GPU-FAST-PROCLUS: A Fast GPU-parallelized Approach to Projected Clustering

Jakob Rødsgaard Jørgensen
jakobrj@cs.au.dk
Aarhus University
Aarhus, Denmark

Katrine Scheel
scheel@cs.au.dk
Aarhus University
Aarhus, Denmark

Ira Assent
ira@cs.au.dk
Department of Computer Science
DIGIT Aarhus University Centre for Digitalisation, Big Data and Data Analytics
Aarhus University
Aarhus, Denmark

Ajeet Ram Pathak
ajeet.pathak44@gmail.com
Vishwakarma Institute of Technology
Pune, India

Anne C. Elster
elster@ntnu.no
Norwegian University of Science and Technology
Trondheim, Norway

ABSTRACT

Projected and subspace clustering aim to find groups of similar objects within a subspace of the full-dimensional space. Where subspace clustering tries to identify clusters in all possible subspaces, projected clustering assigns each point to a single cluster within one projected subspace, resulting in a much smaller result set. PROCLUS is an adaptation of the k-medoids clustering algorithm, CLARANS, to projected clustering. Even though PROCLUS is the first projected clustering algorithm, it is still competitive in comparative empirical studies.

PROCLUS is, however, still too slow for large-scale data or real-time interaction when used in information retrieval processes. Therefore, we propose novel algorithmic strategies to reduce computations and exploit the massive parallelism offered by modern graphical processing units (GPUs). To take advantage of their high degree of parallelism, standard sequential algorithms need to be significantly restructured. We therefore also propose a novel GPU-parallelized algorithm, GPU-FAST-PROCLUS, that takes advantage of the computational power of modern GPUs.

We provide experimental studies that demonstrate the benefit of our proposed strategies and GPU-parallelizations. In this experimental evaluation, we obtain 3 orders of magnitude speedup compared to PROCLUS.

1 INTRODUCTION

Clustering, the task of grouping similar objects, is an often employed data mining task, e.g., for finding groups of customers that exhibit similar traits. However, clustering within the full-dimensional space becomes meaningless for higher-dimensional data as distances become increasingly similar [7]. This implies that clusters might only exist within subspace projections of the full-dimensional space, e.g., for a group of customers, a trait like height might not be important for the grouping. The discovery of such clusters can be made by subspace clustering [4, 19], which finds clusters that exist within all possible subspaces. Projected clustering [2], like subspace clustering, aims to find clusters within a subspace of the full-dimensional space but only reports disjoint clusters, each in one subspace projection. By that, projected clustering reduces the size of the result set and makes it easier to understand for the user.

The first projected clustering algorithm is PROCLUS [2], an adaptation of the K-medoids approach, CLARANS [28], to find clusters in projected subspaces. Today, PROCLUS is still one of the fastest subspace or projected clustering algorithms while remaining competitive in evaluations [26]. However, it still takes up to several minutes to perform PROCLUS on small datasets of just a few thousand data points [2, 26].

Data mining is usually part of information retrieval and data science processes. A successful process produces the information for a task quickly [9]. For real-time interaction, this means executing data analysis within $100ms$ [32]. We propose algorithmic strategies to make PROCLUS fast enough for real-time interaction on even a million data points. This includes strategies to reduce computations and to utilize modern multi-core hardware. Furthermore, we address the challenge of determining the right set of parameters for a clustering algorithm, and leverage the fact that we can reuse partial results between parameter settings to provide even higher speedups when computing several PROCLUS clusterings with different parameter settings.

Modern GPUs with their thousands of cores provide high computational throughput. However, this throughput comes with a vastly different computational model. Since PROCLUS is developed for a single-core model, a novel algorithmic approach must be taken for it to benefit from the computational power offered by GPUs.

Our contributions include:

- Algorithmic strategies to reduce the number of computations by reusing distances and partial results between iterations and parameter settings, as well as by an alternative trade-off between speed and space.
- GPU-parallelized versions of PROCLUS and of our proposed algorithmic strategies.
- An experimental evaluation of PROCLUS and our proposed strategies showing 3 orders of magnitude speedup across parameters and data distributions.

Series ISSN: 2367-2005

10.48786/edbt.2022.09

Table 1: Notation

$Data \in \mathbb{R}^{n \times d}$	Dataset with n points and d dimensions
k	Number of clusters
l	Average number of dimensions
$itrPat$	Max number of iterations w/o changes
$minDev$	Threshold to identify bad medoids
A	Constant to determine size of $Data'$
B	Constant to determine size of M
itr	Iteration counter
t, t'	The current and previous usage
p	Point in $Data$
$Data' \subseteq Data$	Random subset of full dataset
$M \subset Data'$	Greedily selected subset of $Data'$
$MCur \subset M$	Set of current medoids selected from M
$MBest \subset M$	Set of best medoids
$MBad \subset MCur$	Set of replaced medoids in $MCur$
$m_i \in MCur$	i'th medoid
$Dist$	Distance matrix
δ_i	Distance to the closest medoid
$L_i \subseteq Data$	Set of points w/ radius δ_i of medoid m_i
$\Delta L_i \subseteq L_i$	Change in set L_i between iterations
λ_i	Indicates increase or decrease in Δ_i
$MIdx_i$	The index of m_i in M
$DistFound$	Indicates the distances computed
$H_{i,j}$	Sum of dist. of $p \in L_i$ to m_i in dim. j
$X_{i,j}$	Avg. dist. to medoid m_i in dim. j
Y_i	Average distance to medoid m_i
σ_i	Standard deviation of $X_{i,j}$
Z_{ij}	Measure of spread for C_i in dim. j
$D_i \subseteq \{1, \ldots, d\}$	i'th subspace projection
$C_i \subseteq Data$	i'th cluster
$CBest$	Best clustering so far
μ_i	Centroid of cluster C_i
w_i	Cost of a cluster i
$cost$	Weighted cost of the full clustering
$costBest$	Lowest cost found so far
Δ_i	Dist. to closest medoid in subspace D_i

2 BACKGROUND

We provide important notations in Table 1. $|A|$ denotes the size of a set A, $||a||$ is the absolute value of a scalar a, $||p||_1$ is the L1-norm of a point p, and $||p||_2$ is the L2-norm. The norm in subspace projection D_i is denoted using a superscript, e.g., $||p - q||_1^{D_i}$. PROCLUS uses Manhattan distance $||p - q||_1$, Manhattan segmental distance $||p - q||_1^{D_i}/|D_i|$, and Euclidean distance $||p - q||_2$.

We use subscript to index matrices, sets, lists, and points, e.g., $Data_{p,j}$ refers to dimension j of data point p. Superscript is used to refer to a version of a matrix, a set, or a list at a specific iteration, e.g., H^t is H in the current iteration t, H^{t-1} is H in the previous iteration.

We use $\leftarrow$ for assignment and $=$ for equality. For simplicity, we use the index of point p as the actual data point $Data_p$, and vice versa. E.g. when computing a distance between point p and medoid m_i we write $Dist_{m_i,p} \leftarrow ||p - m_i||_2$ as shorthand for $Dist_{m_i,p} \leftarrow ||Data_p - Data_{m_i}||_2$.

2.1 PROCLUS

We briefly describe PROCLUS, details are found in [2]. PROCLUS is an adaptation of the k-medoids algorithm, CLARANS [28], to projected clustering. Given input $Data \in \mathbb{R}^{n \times d}$ a d-dimensional dataset with n points, the number of clusters k, the average number of dimensions l, two scalars A, B, minimum deviation $minDev$, and number of rounds without improvement $itrPat$, PROCLUS proceeds in three phases initialization, iterative, and refinement. PROCLUS outputs k disjoint clusters in different subspace projections.

Initialization phase. First, greedily pick potential medoids M from a subset $Data'$ of the dataset $Data$. The set of points $Data'$ is a random sample of size $A \times k$ from $Data$. From $Data'$ PROCLUS greedily selects $B \times k$ points one by one, as the one with the largest Euclidean distance to all other points in M, see Algorithm 1 line 2-3.

Iterative phase. PROCLUS iteratively optimizes the set of medoids $MCur \subset M$ of size k. This is done through multiple subphases, see lines 5-14. First, compute the set of points L_i within a sphere centered at each medoid m_i. Second, find the optimal sets of dimensions for the sets of points L. Third, assign points to the closest medoid within the selected subspace projection for each medoid. Fourth, evaluate the clustering. If the cost of the clustering is smaller than the best found so far, keep the current set of medoids $MCur$ as the best set of medoids $MBest$ and the corresponding clustering as $CBest$. Compute a new $MCur$ for the next iteration by replacing the bad medoids in $MBest$. The bad medoids $MBad$ are the medoids with a cluster size smaller than $|Data|/k \times minDev$ or if no such exists, the medoid with the smallest cluster. The iterative phase stops when no new $MBest$ has been found for $itrPat$ iterations.

Refinement phase. The last phase refines the clusters found so far. First, let $L \leftarrow CBest$ instead of the spheres and use L to find the best set of dimensions for the medoids. Within these dimensions, assign points to the closest medoid. At last, define a sphere for each medoid m_i with radius $\Delta_i \leftarrow \min_{j \neq i} ||m_i - m_j||^{D_i}/D_i$ in subspace D_i. A point is an outlier if it lies outside the sphere for all medoids.

ComputeL. Compute the set of points L_i close to each medoid $m_i \in MCur$. For each medoid m_i, let $\delta_i \leftarrow \min_{j!=i} ||m_j - m_i||_2$ be the distance to the nearest medoid $m_j \in MCur$ and $L_i \leftarrow \{p \in Data| \; ||p - m_i||_2 \leq \delta_i\}$ be the set of points within the sphere centered at m_i with radius δ_i.

FindDimensions. Find the best subspace projections. Compute the average distances $X_{i,j}$ from all points in L_i to m_i along dimensions j. Next, compute the average distances $Y_i \leftarrow \left(\sum_{j=1}^{d} X_{i,j}\right)/d$ for each medoid m_i across all dimensions j, the standard deviation $\sigma_i \leftarrow \sqrt{\left(\sum_{j=1}^{d} X_{i,j}\right)/(d-1)}$ for each medoid m_i across all dimensions j, and a relative measure of spread $Z_{i,j} \leftarrow (X_{i,j} - Y_i)/\sigma_i$ for each pair of dimension j and medoid m_i.

At last, for each medoid m_i pick the two dimensions with the smallest $Z_{i,j}$, and after that pick the dimensions j corresponding to the lowest $Z_{i,j}$ and add them to the subspace D_i until a total of $k \times l$ dimensions has been picked.

AssignPoints. Assign each point p to cluster C_i with smallest Manhattan segmental distance to medoid m_i within the subspace projection D_i.

EvaluateCluster. The cost of the clustering is the average Manhattan segmental distance from centroid $\mu_i \leftarrow \sum_{p \in C_i} p/|C_i|$

Algorithm 1 PROCLUS$(Data, A, B, k, l, itrPat, minDev)$

```
// Initialization Phase
$Data' \leftarrow$ random sample from $Data$ of size $A \times k$
$M \leftarrow$ Greedy$(Data', A, B, k)$
// Iterative Phase
while $itr < itrPat$ do
    $L \leftarrow$ ComputeL$(Data, MCur)$
    $D \leftarrow$ FindDimensions$(Data, MCur, L, k, l)$
    $C \leftarrow$ AssignPoints$(Data, MCur, D)$
    $cost \leftarrow$ EvaluateClusters$(Data, C, D, k)$
    $itr \leftarrow itr + 1$
    if $cost < costBest$ then
        $itr \leftarrow 0$, $costBest \leftarrow cost$, $MBest \leftarrow MCur$
        $MBad \leftarrow$ ComputeBadMedoids$(MBest, minDev)$
    Compute $MCur$ by replacing the bad medoids in $MBest$
    with random points from $M$
// Refinement Phase
$L \leftarrow CBest$
$D \leftarrow$ FindDimensions$(Data, L, MBest, d, k, l)$
$C \leftarrow$ AssignPoints$(Data, MBest, D)$
$C \leftarrow$ RemoveOutliers$(Data, C, MBest, D)$
return $C, D, M$
```

of cluster C_i within subspace D_i:

$$w_i \leftarrow \frac{\sum_{j \in D_i} V_{i,j}}{|D_i|}, V_{i,j} \leftarrow \frac{\sum_{p \in C_i} ||p_j - \mu_{i,j}||}{|C_i|}, \tag{1}$$

$$cost \leftarrow \frac{\sum_i^k |C_i| \times w_i}{|Data|}. \tag{2}$$

Note that PROCLUS uses localized random search for the best set of medoids and subspaces within a random subset of the data. This implies that PROCLUS is non-deterministic, so results can differ with the same parameters and dataset if the random seeds differ. Any of these results are equally correct according to the PROCLUS algorithm.

3 FAST-PROCLUS

We observe that PROCLUS performs many similar computations that we exploit when we devise parallel algorithmic strategies. Some of these computations are not just similar but repeated between iteration and function calls. We propose two strategies: (1) thorough analysis of distance computations and storing partial results for re-use, (2) parallel algorithms for the GPU. To make a clear distinction between the speedup gained by reducing computations performed, and the speedup gained by parallelization, we describe our FAST-PROCLUS for the algorithmic improvements, and in Section 4.1, we parallelize both PROCLUS and FAST-PROCLUS.

The result of PROCLUS depends on user-selected parameters, so the user usually does multiple runs to find the best parameter setting. We propose a strategy that reuses temporary results between iterations and parameter settings to reduce computations. This includes a heuristic to reduce conversion time by reusing the best set of medoids found for the previous parameter setting.

The iterative phase of PROCLUS has several steps with a $O(n \times k \times d)$ running time. These steps are the most time-consuming and, therefore, the focus for improvement.

Compute distances to potential medoids only once. When computing the set of points L_i within the radius δ_i, PROCLUS computes the distances from each medoid m_i to each point p. These computations have a $O(n \times k \times d)$ run-time and are, therefore, one of the more expensive in PROCLUS.

For the medoids used in earlier iterations, the distances have already been computed and do not change between iterations since the distance measure is the Euclidean distance in full-dimensional space. Furthermore, PROCLUS only picks the current medoids $MCur$ from a small set of $B \times k$ potential medoids M where B should be a small number. This implies that the reuse of medoids is likely. We, therefore, propose to save the distances across iterations. For this purpose, we introduce a distance matrix $Dist \in \mathbb{R}^{Bk \times n}$ with all pairs of distances between medoids and points. This requires $O(B \times k \times n)$ space and is, therefore, a trade-off between running time and space. For the cases where space is a limiting factor, we later propose an adaptation of this strategy that reduces the memory used by a factor B, at the cost of a small increase in running time.

The distance matrix $Dist$ allows us to only compute the distances the first time a medoid is used, and therefore reduce the computations in *ComputeL*. To keep track of which distances have been computed, we maintain an indicator vector $DistFound \in \{true, false\}^{Bk}$ that indicates if the distances to a medoid have been computed. Furthermore, we introduce an index $MIdx_i$ to indicate which of the potential medoids the i'th medoid $m_i \in MCur$ corresponds to from the potential medoids M and the distance matrix $Dist$. For each iteration t we check $DistFound$ to see if the distances from each m_i to all points p has been computed, if not we compute the distances $Dist_{MIdx_i,p} \leftarrow ||p - m_i||_2$ for all p. Afterward, we set $DistFound_{MIdx_i} \leftarrow true$ to indicate that the distances have been computed.

Introduce sum of distances to medoids as temporary result. As part of selecting subspaces, we compute the average distances $X_{i,j}$ to each medoid m_i from all points $p \in L_i$ along dimension j. We observe that it is often the case that the set L_i only changes for a fraction of the points between iterations since the potential medoids are selected to be far apart. We denote the change in L_i by ΔL_i.

Theorem 3.1 (Computing the change ΔL_i in L_i between iterations). *For medoid m_i and the set of points L_i in the sphere centered at m_i with radius δ_i, let:*

$$\begin{aligned}\Delta L_i \leftarrow \{p \in Data|\ & \delta_i^{t'} < ||p - m_i||_2 \leq \delta_i^t \vee \\ & \delta_i^{t'} \geq ||p - m_i||_2 > \delta_i^t\}.\end{aligned} \tag{3}$$

Then ΔL_i is the change in set L_i between the current iteration t and the previous usage t'.

Proof. We have two cases, either the radius δ_i has increased, implying that $\delta_i^t > \delta_i^{t'}$, or it has decreased, implying that $\delta_i^t < \delta_i^{t'}$. In the first case, we can split the set L_i^t at the current iteration t into two disjoint sets, the old set $L_i^{t'}$ at iteration t' union with the change ΔL_i:

$$\begin{aligned} L_i^t &= \{p \in Data|\ ||p - m_i||_2 \leq \delta_i^t\} \\ &= \{p \in Data|\ ||p - m_i||_2 \leq \delta_i^{t'}\} \cup \\ &\quad \{p \in Data|\ \delta_i^{t'} < ||p - m_i||_2 \leq \delta_i^t\} \\ &= L_i^{t'} \cup \{p \in Data|\ \delta_i^{t'} < ||p - m_i||_2 \leq \delta_i^t \vee \\ &\qquad \delta_i^{t'} \geq ||p - m_i||_2 > \delta_i^t\} \\ &= L_i^{t'} \cup \Delta L_i. \end{aligned} \tag{4}$$

And analogously for the decrease in L_i. □

To avoid recomputing the entire $X_{i,j}$, we propose to maintain a matrix $H^{t'} \in \mathbb{R}^{Bk \times d}$ with the sum of distances to each medoid m_i to all points in $L_i^{t'}$ along dimension j from the previous usage t':

$$H^{t'}_{MIdx_i,j} = \sum_{p \in L_i^{t'}} ||p_j - m_{ij}||. \tag{5}$$

Theorem 3.2 (Computing H iteratively). *For medoid $m_i \in MCur$, set L_i^t in radius δ_i^t at iteration t and change ΔL_i since the previous usage t' we can split the sum of distances $H^t_{MIdx_i,j}$ into two parts:*

$$H^t_{MIdx_i,j} \leftarrow H^{t'}_{MIdx_i,j} + \lambda_i \times \sum_{p \in \Delta L_i} ||p_j - m_{ij}||, \tag{6}$$

where λ_i is 1 if the sphere increases in size and -1 if it decreases.

Proof. We again have two cases. Either the change is an increase or decrease. For increase, using Theorem 3.1 and since $L_i^{t'}$ and ΔL_i are disjoint, we have:

$$\begin{aligned} H^t_{MIdx_i,j} &= \sum_{p \in L_i^t} ||p_j - m_{ij}|| = \sum_{p \in L_i^{t'} \cup \Delta L_i} ||p_j - m_{ij}|| \\ &= \sum_{p \in L_i^{t'}} ||p_j - m_{ij}|| + \sum_{p \in \Delta L_i} ||p_j - m_{ij}||. \end{aligned} \tag{7}$$

Analogously for decrease. □

We use Theorem 3.2 to update H and then compute the average distance $X_{i,j} \leftarrow H^t_{MIdx_i,j}/|L_i|$ across dimension j from all points in L_i to m_i.

Only updating H with the change in L_i requires that during *ComputeL* we do not calculate the set L_i, but instead the change ΔL_i in points as in Theorem 3.1. Notice that ΔL_i can be computed in the same way as L_i, the only difference being the condition that we keep points between the current δ_i^t and the previous $\delta_i^{t'}$ radius of the set L_i. To maintain radius $\delta_i^{t'}$ we must keep the previous radius for any of the $B \times k$ potential medoids. Furthermore, we maintain the size of the set L_i, $|L^t_{MIdx_i}| \leftarrow |L^{t'}_{MIdx_i}| + \lambda_i \times |\Delta L_i|$, for all potential medoids.

We can now update H between iterations instead of recomputing the sum of distances for each iteration. This implies that we can reuse computations and reduce the running time.

3.1 Multiple parameter settings

A drawback for most subspace and projected clustering algorithms, including PROCLUS, is that the result depends on the selected parameters. In practice, these algorithms are typically run multiple times with different parameters. For PROCLUS, the important parameters are the number of clusters k and the average number of dimensions l.

When running for multiple different parameter settings, PROCLUS performs many similar computations. We observe that if we have the same potential medoids M for all parameter settings, both the distance matrix *Dist* and the sum of distances H can be reused. To achieve this, we only once greedily pick potential medoids for the largest k and use this set M for all parameter settings. Having a constant $|M| = B \times k$ picked from a set of size $|S| = A \times k$ corresponds to an increase in $A = |S|/k$ and $B = |M|/k$ as k decreases. In other words, the first selection of A, B, and k impacts the values for A and B in subsequent parameter settings for different k. Please note that A, B, k, and l are only used to determine the size of M and S and do not otherwise impact greedy picking, so the likelihood of picking a specific M for any given execution is unchanged. We thus trade-off selection of A and B for speed. We implement both this faster version reusing the computations saved in *Dist*, H, M, and S, as well as one that follows the original PROCLUS sampling strategy, and study the speedup gained in the experiments, Section 5.

As an additional speedup option, we introduce a heuristic that reusing medoids found to be good in one setting as initialization in other settings, as this may lead to faster convergence. Therefore, instead of initializing each parameter setting with the current medoids *MCur* as a random subset of the potential medoids M, we initialize the current medoids as a random subset of the previous best medoids *MBest*. In the experiments, Section 5, we study the speedup this initialization provides.

3.2 Trade-off between running time and space

We propose an adaptation of FAST-PROCLUS, called FAST*-PROCLUS, that reduces the space complexity at the cost of a slight increase in running time. Instead of saving the distance matrix *Dist* and the sum of distances H to all potential medoids, which require $O(B \times k \times n)$ and $O(B \times k \times d)$ space, respectively, we only keep these temporary results from the previous iteration $t - 1$. This requires only $O(k \times n)$ space but implies that we can only reuse the computations in *Dist* and H from iteration $t - 1$ instead of any earlier iteration t'. However, it is often the case that few of the current medoids are replaced, and we can, therefore, still reuse most of the computed distances.

Since we no longer need to keep track of which of the potential medoids are in use, we do not use the index *MIdx*. Instead, we use $i \in MBad$ to identify for which of the medoids we need to recompute the distance matrix *Dist* and reset the previous δ_i^{t-1}, size of L_i and the sum of distances H before we compute ΔL_i.

4 GPU-PROCLUS

Modern GPUs provide high computational power through thousands of cores at the cost of a restrictive parallel computational model. This stands in contrast to the sequential model of the CPU that most algorithms follow. Therefore, when developing algorithms for the GPU, several properties must be considered. We use the term parallel to denote parallel execution under the GPU's computational model.

Programming the GPU is more like using vector registers, where the same operation is performed on each element on a given execution cycle. This is known as the Single Instruction Multiple Data (SIMD) model. GPUs similarly use a Single Instruction Multiple Thread (SIMT) model where hundreds or thousands of threads are executed at the same time.

On NVIDIA GPUs, instructions are grouped into vector instructions known as warps, where 32 threads are scheduled together and share a program counter. This implies that all cores in a warp must perform the same instruction at all times. Furthermore, cores are grouped into streaming multiprocessors (SMs). The warps are scheduled on a given SM. Warps on the same SM can share fast access L1-cache associated with that SM and can also synchronize.

In the CUDA programming environment, the CPU program spawns functions, known as kernels, onto the GPU. The functions spawn on invocation of a given number of threads to be executed concurrently. These threads may be organized into blocks. All threads within a thread block are executed within the same SM. Threads in different blocks cannot synchronize automatically, so computations performed in different blocks should be independent to avoid a global synchronization. By independent we mean computations that do not use the partial result of each other. Data accessed by threads in different blocks must be located in global memory, which is slower than the shared memory.

When multiple threads write to the same memory address, race conditions can occur, where changes by one of them may be lost. To avoid such behavior, the GPU provides atomic versions of increment, addition, maximum, etc. However, these are more expensive and should be avoided, if possible.

In this paper, our algorithms are structured such that they may perform for-loops in parallel as threads. This entails that each step of the for-loop iterations is performed concurrently by different threads. We use a similar notation for thread blocks. If the for-loop has more iterations than threads per thread block, each thread handles multiple iterations.

4.1 GPU-friendly parallelization

PROCLUS has a long running time for larger datasets and thus is too slow for interactive settings. We, therefore, propose an algorithm capable of utilizing the computational power of the GPU to reduce the running time. We present GPU-parallelized versions of PROCLUS, FAST*-PROCLUS, and FAST-PROCLUS, called GPU-PROCLUS, GPU-FAST*-PROCLUS and GPU-FAST-PROCLUS.

In our GPU-parallelization approaches, we ensure a correct PROCLUS result by only parallelizing independent computations, or else by synchronizing to ensure that all threads within a block are at the same state. Furthermore, to avoid race conditions between threads that compute part of a common result, we use atomic operations. As mentioned above, PROCLUS is non-deterministic due to local randomization, so results between runs may differ both for the GPU versions and the CPU versions of PROCLUS, but all our results are fully correct with respect to the PROCLUS definition.

Each of the sub-functions *Greedy*, *ComputeL*, *FindDimensions*, *AssignPoints*, and *EvaluateCluster* has a high time complexity and will therefore be the focus of this section.

To avoid costly memory transfers between the CPU and the GPU, all other computations are also performed on the GPU. Each sub-function is formulated as a separate algorithm for readability. However, since it is time-consuming to allocate and free memory on the GPUs, we allocate all required memory at the beginning of GPU-PROCLUS and reuse the same allocated memory for all of the iterations.

Greedy. In PROCLUS [2], the greedy selection of potential medoids repeatedly selects the point that is furthest away from all other potential medoids. Algorithm 2 shows our GPU-parallelized version of the greedy function. At line 1-4 we first pick a random point M_1 in $Data'$ as part of M and in parallel we compute the Euclidean distance $Dist_p \leftarrow ||M_i - p||_2$ to all points p in $Data'$. Each distance computation is completely independent of others and can therefore be computed using different thread blocks. To reduce the number of accesses to global memory, we compute the maximal distance *maxDist* within the same kernel call, line 5. However, to guarantee that the correct maximum *maxDist* is computed, we must ensure that all blocks have finished before using the global maximum *maxDist*. We, therefore, need to check which points have the largest distance in a separate kernel call at line 6-9.

Algorithm 2 Greedy($Data', A, B, k$)

```
1:  Pick $M_1$ at random from $Data'$.
2:  $maxDist \leftarrow 0$ // shared variable
3:  for $p \in Data'$ - in parallel as threads and blocks do
4:      $Dist_p \leftarrow ||M_i - p||_2$
5:      $maxDist \leftarrow \max(maxDist, Dist_p)$ // atomic
6:  for $i \leftarrow 2, \dots, Bk$ do
7:      for $p \in Data'$ - in parallel as threads and blocks do
8:          if $maxDist = Dist_p$ then
9:              $M_i \leftarrow p$
10:     $maxDist \leftarrow 0$
11:     for $p \in Data'$ - in parallel as threads and blocks do
12:         $Dist_p \leftarrow \min(Dist_p, ||M_i - p||_2)$
13:         $maxDist \leftarrow \max(maxDist, Dist_p)$ // atomic
14: return $M$
```

At line 6 to 13, we repeat this procedure until $B \times k$ potential medoids has been picked. The only change is that we keep the smallest distance to any already picked potential medoids from point p.

ComputeL. Computing the set of points L_i for each medoid m_i is done on the GPU as in Algorithm 3. First, pre-compute the distances $Dist_{i,p}$ between each medoid m_i and each point p. Each distance computation is completely independent and can be performed completely in parallel, see line 1-3. Next, in parallel over each pair of medoids m_i, m_j, find the distance δ_i from m_i to the closest medoid $m_{j'}$, see line 4-7. For each medoid m_i, we compute the set of points L_i in the sphere with radius δ_i centered at m_i. This is done by checking if the distance to each point p is within δ_i.

To save time, we allocate memory for the worst-case size of L_i and add points to the first available location in the allocated array. Adding each point p to the set L_i is done using *atomicInc* to increment the location of points without race conditions, see line 8-12.

Algorithm 3 ComputeL($Data, MCur$)

```
1:  for $m_i \in MCur$ - in parallel as blocks do
2:      for $p \in Data$ - in parallel as threads and blocks do
3:          $Dist_{m_i,p} \leftarrow ||m_i - p||_2$
4:  for $m_i \in MCur$ - in parallel as blocks do
5:      for $m_j \in MCur$ - in parallel as threads do
6:          if $m_i \neq m_j$ then
7:              $\delta_i \leftarrow \min(\delta_i, Dist_{m_i,m_j})$ // atomic
8:  for $m_i \in MCur$ - in parallel as blocks do
9:      for $p \in Data$ - in parallel as threads and blocks do
10:         if $Dist_{m_i,p} \leq \delta_i$ then
11:             $l \leftarrow increment(|L_i|)$ // atomic
12:             $L_{i,l} \leftarrow p$
13: return $L$
```

FindDimensions. Finding the dimensions for the projected subspaces follows the formula for the original PROCLUS closely. Each entry of X, Y, σ, Z can be computed completely independently and therefore by different thread blocks. To avoid saving Y and σ to global memory we combine the computation of Y, σ, and Z into one kernel call. This reduces the running time substantially, but since these kernel calls are small the impact on the overall running time is minor.

Algorithm 4 FindDimensions$(Data, MCur, L, k, l)$

```
for $m_i \in MCur$ - in parallel, as blocks do
    for $j \leftarrow 1, ..., d$ - in parallel, as blocks do
        $sum \leftarrow 0$ // local variable
        for $p \in L_i$ - in parallel, as threads do
            $sum \leftarrow sum + ||p_j - m_{i,j}||$
        $X_{i,j} \leftarrow X_{i,j} + sum/|L_i|$ // atomic
for $m_i \in MCur$ - in parallel, as blocks do
    for $j \leftarrow 1, ..., d$ - in parallel, as threads do
        $Y_i \leftarrow Y_i + X_{i,j}/d$ // atomic
        $\sigma_i \leftarrow \sigma_i + (X_{i,j} - Y_i)^2$ // atomic
        synchronize threads
        $\sigma_i \leftarrow \sqrt{\sigma_i/(d-1)}$
        synchronize threads
        $Z_{i,j} \leftarrow (X_{i,j} - Y_i)/\sigma_i$
Pick the dimensions $j$ with the two smallest $Z_{i,j}$ values for
each medoids $m_i$.
Pick next $k \times l - 2 \times k$ smallest $Z_{i,j}$, append associated di-
mensions $j$ to subspace of corresponding medoids $m_i$
return $D$
```

When computing $X_{i,j}$, we sum across a large number of points. To avoid race conditions each addition to the global memory location must be performed by *atomicAdd*. Instead of performing the expensive atomic operations for each point, we let each thread compute a part of the sum locally. Afterward, each thread adds the local sum atomically to $X_{i,j}$ in the global memory. To compute the average, the local sum is divided by $|L_i|$. At last, Z is used to pick the subspaces D.

AssignPoints. Assigning each point p to the closest medoid m_i is done by first computing the distance $Dist_{p,m_i}$ from each point p to each medoid m_i, which again can be done completely in parallel. Remember that the distance measure used to assign points is the Manhattan segmental distance in subspace D_i, and we can therefore not reuse the previously computed distances. Next, for each point p in parallel, we check which medoid is closest and assign the point p to the corresponding cluster C_i. Both steps are joined into one kernel to reduce the number of global memory accesses, see Algorithm 5. Remember that accessing shared memory is faster than accessing global memory, therefore, it has a large effect on the running time. We use *atomicMin* to find the smallest distance to a medoid and synchronize to ensure that all medoids have been checked before selecting the closest. This implies that we must compute the distances from each point to all medoids in the same thread block. Adding the points to set C_i is done the same way as for L_i.

EvaluateCluster. The evaluation of the clustering is the average Manhattan segmental distance to the centroid, not to the medoid as in previous sub-functions. Therefore the first step is to compute the centroid of each cluster.

Algorithm 5 AssignPoints$(Data, MCur, D)$

```
for $p \in Data$ - in parallel, as threads and blocks do
    for $m_i \in MCur$ - in parallel, as threads do
        $minDist_p \leftarrow \infty$ // shared variable
        $Dist_{p,m_i} \leftarrow ||p - m_i||_1^{D_i}/|D_i|$ // local variable
        $minDist_p \leftarrow \min(minDist_p, Dist_{p,m_i})$ // atomic
        synchronize threads
        if $minDist_p = Dist_{p,m_i}$ then
            $l \leftarrow increment(|C_i|)$ // atomic
            $C_{i,l} \leftarrow p$
return $C$
```

The formulation of the cost-function in Eq. 2 is separated into multiple steps that each could be parallelized in its own kernel call; computing the mean $\mu_{i,j}$ of each cluster, the average distance to the mean along each dimension $V_{i,j}$, the average distance among the dimensions for each cluster w_i, and finally the summed weighted cost for the entire clustering. However, this would require saving temporary results of each step to global memory, which is expensive to access. To avoid this, we reformulate the cost-function into a sum of values that can be computed in parallel and where only the final cost must be written to global memory:

$$cost = \frac{\sum_i^k |C_i| \times \frac{\sum_{j \in D_i} \frac{\sum_{p \in C_i} ||p_j - \mu_{i,j}||}{|C_i|}}{|D_i|}}{|Data|} \tag{8}$$

$$= \sum_{i=1}^{k} \sum_{j \in D_i} \sum_{p \in C_i} \frac{||p_j - \mu_{i,j}||}{|D_i| \times |Data|}. \tag{9}$$

Eq. 9 allows both the mean $\mu_{i,j}$ and $cost$ to be computed in parallel using a thread block for each pair of medoids m_i and dimensions j and then distribute the points among different threads within these thread blocks. Since this is the case, we can combine the computation of both into one kernel call, and synchronize all threads in each block to ensure that the computation of $\mu_{i,j}$ has finished before using it. By this, we can avoid writing $\mu_{i,j}$ to global memory but instead keep it as a shared variable. Writing to shared memory is much faster then writing to global memory and therefore provides a large reduction in running time.

To reduce the number of atomic operations in Algorithm 6, we use a local temporary variable that each thread can save its partial result to and then only perform one atomic operation per thread at the very end. This strategy is used both to compute the centroid μ_i and $cost$.

Algorithm 6 EvaluateCluster$(Data, C, D, k)$

```
for $i \leftarrow 1, \ldots, k$ - in parallel as blocks do
    for $j \in D_i$ - in parallel as blocks do
        $\mu_{i,j} \leftarrow 0$ // shared variable
        $tmp \leftarrow 0$ // local variable
        synchronize threads
        for $p \in C_i$ - in parallel as threads do
            $tmp \leftarrow tmp + p_j$
        $\mu_{i,j} \leftarrow \mu_{i,j} + tmp/|C_i|$ // atomic
        $tmp \leftarrow 0$
        synchronize threads
        for $p \in C_i$ - in parallel as threads do
            $tmp \leftarrow tmp + ||p_j - \mu_{i,j}||$
        $cost \leftarrow cost + tmp/(|D_i| \times |Data|)$ // atomic
return $cost$
```

Updated and iterations. We also update the best clustering, the best subspace, the best medoids, the current medoids, and the iteration counter for each iteration. However, this part is not time-consuming and details are therefore omitted.

RemoveOutliers. To remove ourliers, we compute the smallest distance between two medoids m_i, m_j, for each medoid m_i in parallel as block and each m_j using threads within that block and use atomics to find the smallest distance Δ_i. Then in parallel across all points, we check if it lies within the Δ_i radius of any medoid m_i, else, it is reported as an outlier.

4.2 GPU-FAST-PROCLUS

The proposed strategies for reducing computations need modifications to fit the GPU. The lookup in the distance matrix *Dist* and *H* is as for FAST-PROCLUS. However, when computing the distances, in ComputeL, we must ensure that all threads have checked the flag $DistFound_{MIdx_i}$ before marking it as computed.

Since we would like to utilize as many cores as possible, we distribute the distance computations among multiple thread blocks. Instead of using community groups to synchronize across thread blocks, we set the flag afterward in a separate kernel call. Both λ_i and $|L_i^t|$ are computed as in FAST-PROCLUS, but only one thread per medoid m_i needs to compute this. Similarly for FindDimensions, when computing the average distance $X_{i,j}$, we must ensure that H is updated by all threads before computing $X_{i,j} \leftarrow H_{i,j}/|L_i|$. Therefore, $X_{i,j}$ is computed in a separate kernel call. The rest of GPU-FAST-PROCLUS proceeds as GPU-PROCLUS.

5 EXPERIMENTS

We perform real-world experiments on a workstation with Intel Core i7-9750HF 2.6GHz 12-cores, 16 GB RAM, and a GeForce GTX 1660 TI 6 GB dedicated RAM. For the larger synthetic datasets, we move experiments to a workstation with an Intel Core i9 10940X 3.3GHz 14-Core, 258 GB RAM, and a GeForce RTX 3090 with 24 GB dedicated RAM. All algorithms have been implemented in C++ or CUDA. For repeatability, the source code is provided at: https://au-dis.github.io/publications/GPU-FAST-PROCLUS/.

Multi-core CPU-version. Some of the strategies proposed for GPU-parallelization are directly applicable to the CPU as well. We have therefore implemented multi-core CPU versions using OpenMP[1] to study the speedup of parallelization on the CPU vs. the GPU.

Algorithm parameters. The default parameters in all experiments are $k = 10$, $l = 5$, $A = 100$, $B = 10$, $minDev = 0.7$, and $itrPat = 5$.

CUDA kernel configurations. For the CUDA kernel configurations, the block size of 1024 threads is used. If fewer threads are required per block, only the required number of threads are started. To reduce unnecessary synchronizations in AssignPoints, Algorithm 5, 128 threads are used per block.

Synthetic data. For control of data distribution and size, we use the synthetic dataset generator provided by [6]. However, we modify the generator as [18] to generate clusters in any arbitrary subspace. The default parameters for the generated data are 64, 000 points with 15 dimensions, each dimension has values in the range 0 to 100. The points are distributed among 10 Gaussian distributed clusters in a subspace of 5 dimensions and with a standard deviation of 5.0.

Real-world data. For experiments on real-world datasets we use the datasets glass, vowel, pendigits [27] and part of the SkyServer dataset [33]. The glass dataset is of size 214 with 9 features, vowel is of size 990 with 10 features, and pendigits is of size 7, 494 with 16 features. From the SkyServer dataset, we use an area of size 1 × 1 measured in the spherical coordinates, referenced as sky 1 × 1. This subset contains 30, 390 points and we extract 17 features including the spherical coordinates. We also extract a 2 × 2 area with 133, 095 points and a 5 × 5 area with 934, 073 points.

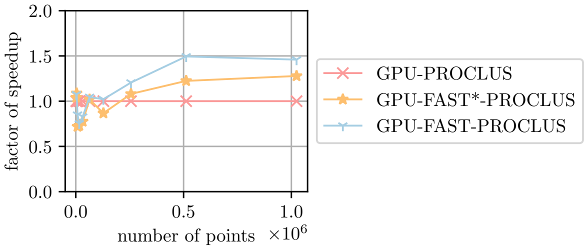


Figure 1: Speedup w.r.t. GPU-PROCLUS.

All reported running times are averages of 10 runs on different generated datasets. The real-world and synthetic datasets are min-max normalized, such that all dimensions have values between 0 and 1.

5.1 Scalability

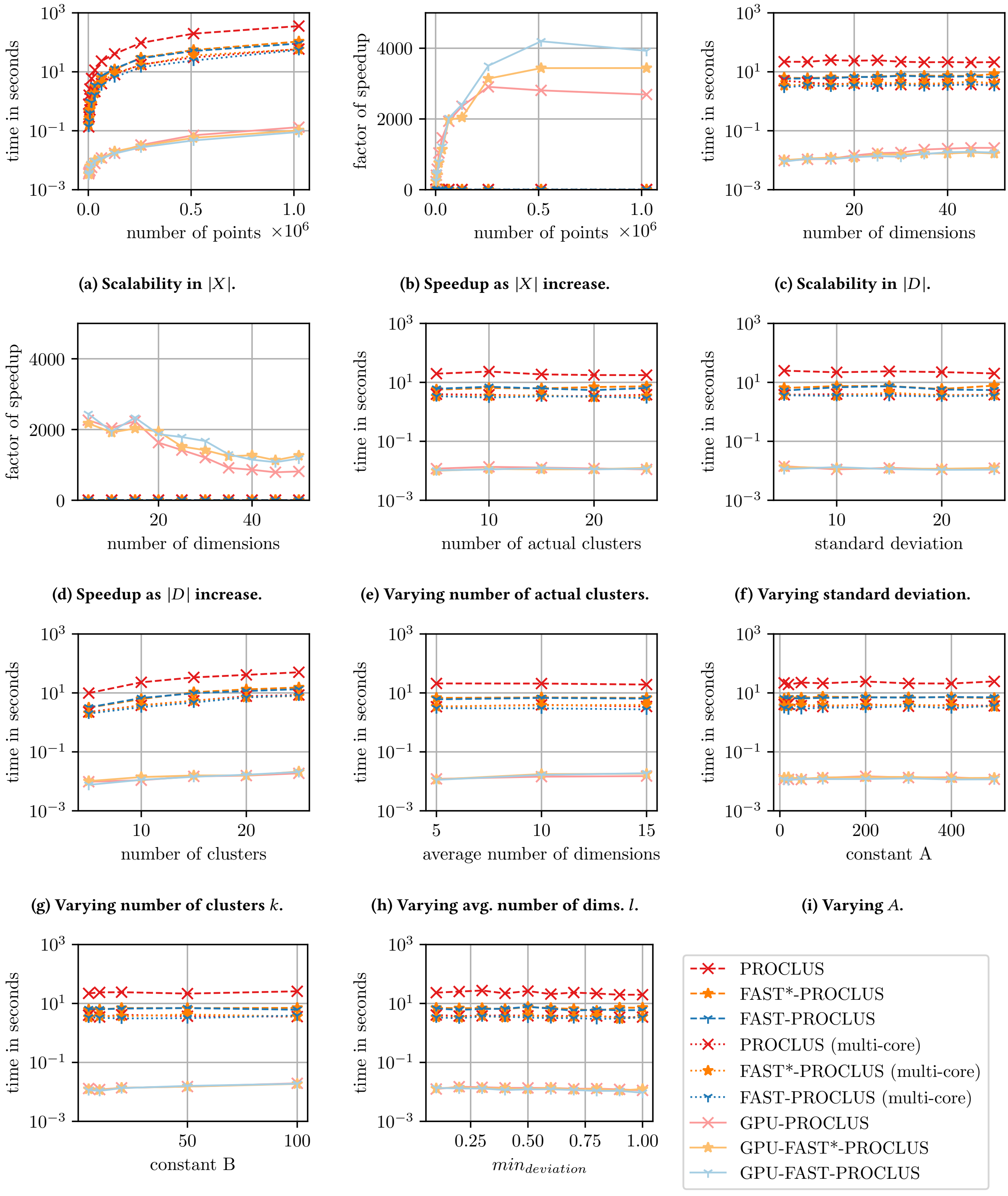
PROCLUS uses randomized search, but besides this random behavior, GPU-PROCLUS and all the algorithmic strategies produce the same clustering as PROCLUS. The important measure in this work is, therefore, not the accuracy but solely the running time. This section investigates how the size of the dataset and its dimensionality affect the running time of our proposed algorithms. We first compare against PROCLUS, where we run with just one parameter setting at a time. Later, in Section 5.3, we show how GPU-FAST-PROCLUS can achieve even higher speedup when allowed to reuse partial computations between parameter settings.

Figs. 2a-2b shows that the algorithmic strategies provide a factor of 1.2 to 1.4× speedup for both PROCLUS and GPU-PROCLUS. However, the GPU-parallelization of each strategy provides an additional 2, 000× speedup. This speedup increases with the input size and stays constant after a certain input size. This is due to the more points, the easier it is to utilize all cores on the GPU. The speedup is so great that we can now perform PROCLUS in less than 100*ms*, the limit for real-time interaction [32], for even 1, 000, 000 data points. Similarly, the multi-core CPU-version provides up to 6× speedup. The comparatively low utilization compared to the GPU could be due to the many context switches.

Figs. 2c-2d shows that the factor of speedup is higher for a lower number of dimensions, ranging from 896 to 1, 265× speedup. This could be caused by not all distance computations being parallelized across dimensions to avoid atomic operations and synchronizations.

Space usage. For FAST* compared to FAST, we see approximately 1.05 to 1.1× slowdown, see Fig. 1, but with the benefit of a reduction in space usage. Fig. 3f investigate the reduction in space usage. Space usage of GPU-FAST*-PROCLUS is approximately half of that of GPU-FAST-PROCLUS and the space usage of GPU-PROCLUS and GPU-FAST*-PROCLUS is similar. Space usage of all algorithms increases linearly in *n*, which is inline with our space complexity analysis.

[1] https://www.openmp.org/



(a) Scalability in $|X|$.

(b) Speedup as $|X|$ increase.

(c) Scalability in $|D|$.

(d) Speedup as $|D|$ increase.

(e) Varying number of actual clusters.

(f) Varying standard deviation.

(g) Varying number of clusters k.

(h) Varying avg. number of dims. l.

(i) Varying A.

(j) Varying B.

(k) Varying $minDev$.

Figure 2: Average running times of runs with a single parameter settings.

5.2 Effect of data distributions and parameters

The performance of clustering, subspace, and projected clustering algorithms is affected by the data distribution. Therefore, we verify that GPU-PROCLUS performs well across data distributions. In Fig. 2e we vary the number of clusters and in Fig. 2f we vary the data distribution using different standard deviations. Here, we see that the running time of PROCLUS and GPU-PROCLUS is largely unaffected.

We show how different parameter settings affect the running time of PROCLUS and our proposed algorithms. In Fig. 2g-2k, we increase each of the parameters in PROCLUS one by one. We observe that the running time stays almost constant for most parameters, except for k and B, where we see that running time for both PROCLUS and GPU-PROCLUS increases with k or B. This is clearly because distances for a larger set of current medoids or potential medoids are computed. However, for all experiments, the factor of speedup remains relatively constant at around 1100×.

5.3 Multiple parameter settings simultaneously

As mentioned in Section 3, the result of PROCLUS depends on the parameters, so it is often run with multiple sets of parameters. GPU-FAST-PROCLUS uses this to reduce the number of distance computations. In Fig. 3a-3e, we show the average running time of testing 9 combinations of k and l. The reported running times are averages per combination to make it easier to compare with running times for just a single parameter setting. We see that GPU-FAST-PROCLUS provides up to around 7000× speedup w.r.t PROCLUS. Furthermore, in Fig. 3e GPU-PROCLUS and GPU-FAST-PROCLUS run on more than $8,000,000$ points, and we see that the average execution time never exceeds a second. At $8,000,000$ points, space becomes the limiting factor, exceeding the 4.2 GB of free memory on our relatively small GPU.

The strategy of reusing computations between parameter settings consists of three parts: **multi-param 1** reuses partial compuations, **multi-param 2** reuses also greedy picking and **multi-param 3** reuses also the previous best set of medoids. Compared to GPU-FAST-PROCLUS executed with one parameter setting at a time, the reuse of partial computations provides approximately a factor 1.4× speedup, also reusing the greedy picking provides approximately a factor 1.6× speedup, and also initializing with the previous best set of medoids provides approximately a factor 2.3× speedup.

5.4 GPU-utilization

The utilization of the GPU is dependent on many factors like memory throughput and and occupancy of the threads. Further more, it can very at lot between the dataset, parameters, and from kernel to kernel. We provide the utilization, inform of The memory throughput, theoretical occupancy, and achieved occupancy provided by NVIDIA Nsight Compute[2], for some of the most interesting kernels and extreme cases. An example of what could decrease the occupancy is if a block of threads uses more than the registries that are available. The most time-consuming kernel is Algorithm 6. Given the parameter settings used in this section and a dataset with $4,096,000$ points and 10 dimensions, it has a theoretical occupancy of 100.00%, achieved occupancy of 99.99%, and memory throughput of 86.54%. Reducing the dataset size to $8,000$ points reduces the utilization to a theoretical occupancy of 78.12%, achieved occupancy of 77.98%, and memory throughput of 50.06%, this is because having $8,000$ points and 10 clusters implies that we spawn around 800 threads per block, which is not a good balance of the warps that can be executed per block. This kernel together with most of our other algorithms has a high utilization since we, in general, parallelize across a large number of points and try to keep the threads that need to communicate within the same block and do not exhaust the resources. On the other hand, a few kernels do not process all points, e.g., Algorithm 3 line 4-7 spans k blocks and k threads per block. If $k < 32$, we do not have enough threads per block to utilize a full warp and if $k \times k$ is less than the number of cores on the GPU, not all cores are engaged. If the preceding and the succeeding kernels were not depending on each other, streams could be used to run two kernels concurrently to engage more cores. This kernel has a theoretical occupancy of 50.00%, achieved occupancy of 3.12%, and memory throughput of 1.64%. This is not a good utilization, but not a time-consuming computation either, so these small kernels do not have a large effect on the overall running time.

5.5 Performance on real world datasets

Running with 9 parameter settings, we show that GPU-FAST-PROCLUS retains the high speedup on different real-world datasets. In Fig. 3g, we show the running time on different real-world datasets, and the experiments confirm that we obtain similar speedups for real-world datasets as we achieve for synthetic data. To be more specific, GPU-FAST-PROCLUS achieves 5490× speedup compared to PROCLUS on the sky 5×5 dataset. As for the synthetic data, the speedup is greatest for large datasets.

6 RELATED WORK

Clustering is the task of grouping similar data points [35]. Distance-based methods like k-means [24] or k-medoids [28] try to minimize intracluster distance. Density-based methods like DBSCAN [14], DPC [31] and SynC [8] find clusters as high density regions separated by sparse regions.

In high dimensional data, clusters might only exist in subspaces of the full-dimensional space, giving rise to subspace and projected clustering [29]. Subspace clustering discovers clusters in any possible subspace projection, allowing a data point to participate in none or several clusters in different subspaces, whereas projected clustering assigns each point to exactly one cluster in one subspace projection. Subspace and projected clustering can be categorized as top-down or bottom-up algorithms.

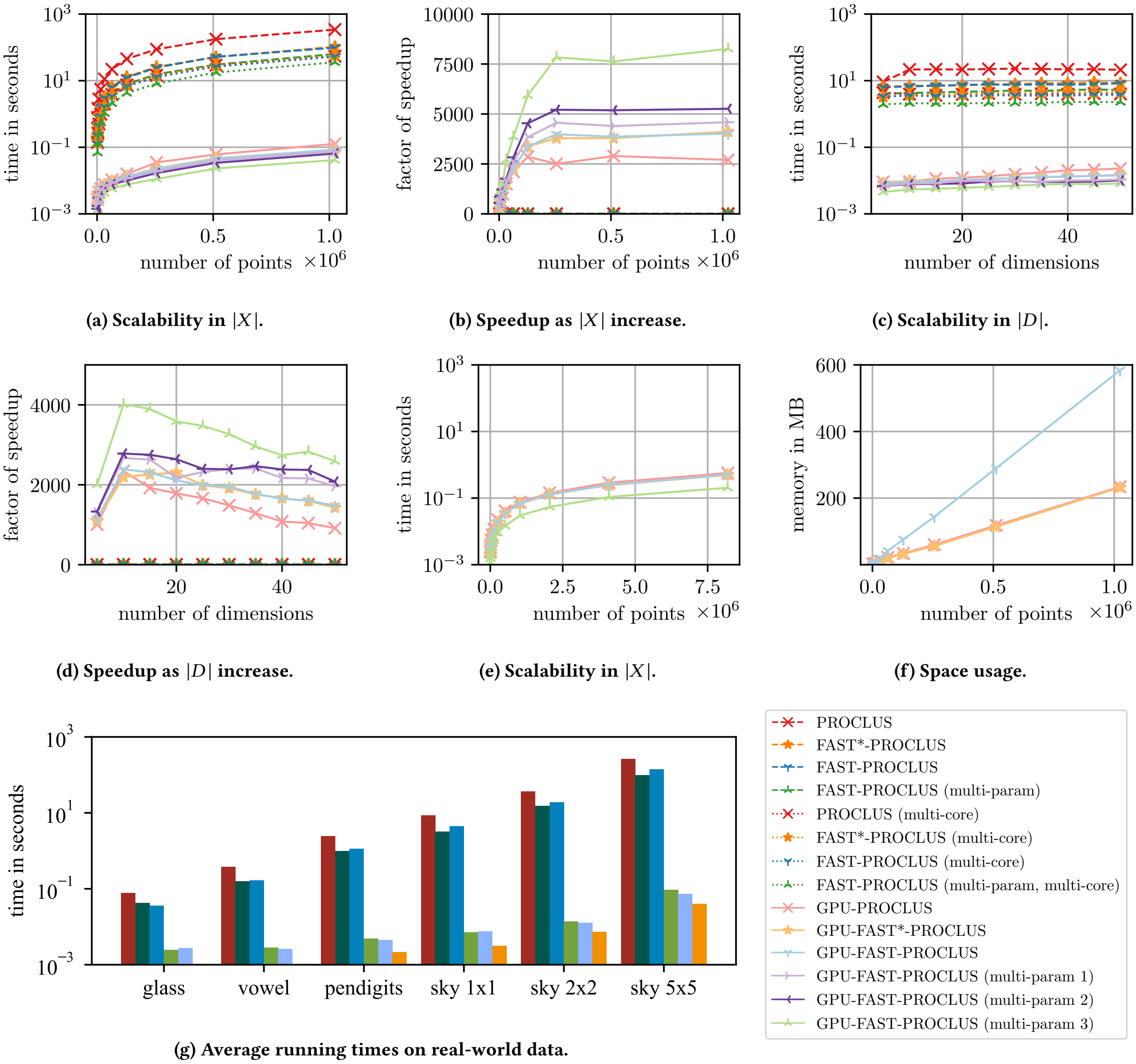
Bottom-up approaches [4, 11, 16, 19] find clusters in $k = 1$ dimensional subspaces, then iteratively combine clusters in k-dimensional subspaces to find clusters in $(k+1)$-dimensional subspaces. Top-down approaches [2, 3, 10, 13, 34] find clusters in the full-dimensional space and iteratively update assigned subspace projections to these clusters.

Subspace and projected clustering are time-consuming to compute due to the number of subspaces increasing exponentially in the number of dimensions, giving rise to works on efficient algorithms. Some algorithms reduce the running time by pruning subspace regions that cannot contain clusters [5, 19], approximating potentially dense areas by e.g. grid cells or histograms [4, 11, 16, 23], or by locally optimizing to iteratively improve candidate clusters and associated dimensions [2, 3, 10, 13, 25, 34].

While much effort has gone into algorithmic improvements of subspace and projected clustering algorithms, the high computational power of the graphics processing unit (GPU) remains largely unexplored. To the best of our knowledge, GPUMAFIA [1], a GPU version of SUBSCALE[12], and GPU-INSCY [18] are the only GPU algorithms proposed for subspace clustering. We review them in turn.

MAFIA is a bottom-up subspace clustering algorithm that combines histogram approximations of overlapping dense subspace regions as it moves from lower dimensional to higher dimensional subspaces [16]. GPUMAFIA efficiently parallelizes these

[2] https://developer.nvidia.com/nsight-compute



(a) Scalability in $|X|$.

(b) Speedup as $|X|$ increase.

(c) Scalability in $|D|$.

(d) Speedup as $|D|$ increase.

(e) Scalability in $|X|$.

(f) Space usage.

(g) Average running times on real-world data.

Figure 3: Average running times of runs with 9 parameter settings at a time.

core steps in different kernels. As PROCLUS uses a different clustering notion that makes use of neither histograms nor dense regions, and that cannot proceed in a bottom-up fashion on subspaces, the algorithmic parallelization strategies in GPUMAFIA are not applicable to the GPU-parallelization of PROCLUS.

SUBSCALE [20] is also density-based, but finds dense units, similar to neighborhoods in DBSCAN, per dimension. The dense units that overlap in dimensions can be combined and subjected to DBSCAN to derive the actual clusters. In the GPU-parallelized version of SUBSCALE[12], the identification of overlapping dense units across dimensions and dense units is parallelized, and possible dense units are precomputed. Again, the clustering notion and algorithmic strategy differ from PROCLUS such that it cannot serve as inspiration for a GPU-parallelization of PROCLUS.

Finally, INSCY [5] is also a density-based subspace clustering approach, but proceeds in a depth-first manner over potential dense subspace regions using a specialized tree structure. GPU-INSCY proposes a GPU-friendly version of the tree structure and devises algorithmic strategies for efficiently handling multiple dense regions in parallel. The GPU-parallelization is tailored to INSCY and does not fit for PROCLUS that does not operate on dense regions, and thus cannot benefit from a tree structured for managing them.

As PROCLUS is an adaptation of k-medoids to projected clustering [28], it is worth considering GPU-parallelized versions of full dimensional k-medoids clustering [21, 30], or of the similar k-means [15, 17, 22]. However, as opposed to k-medoids, which is based on distances between objects that reside in the same space, PROCLUS iteratively adds and removes dimensions from intermediate projected clusters, which results in changes of distance values and changes of projected subspaces. Thus, a GPU version of k-medoids cannot serve as a subroutine of a GPU version of PROCLUS.

7 CONCLUSIONS

We substantially improve the running time of PROCLUS for large-scale data to the extent that real-time interaction with PROCLUS projected clustering becomes possible. We achieve this in our GPU-FAST-PROCLUS, a GPU-parallelized version of PROCLUS that also contributes several algorithmic improvements. Our improvements are two-fold; efficient algorithmic strategies for PROCLUS, here termed FAST-PROCLUS, and an efficient parallelization on the GPU.

The algorithmic improvements target the most costly operations in PROCLUS, restructuring computations such that we can save and reuse distance computations and partial results. In addition, we introduce strategies that reuse partial computations across multiple parameter settings to further speed up FAST-PROCLUS. We demonstrate a trade-off between running time and space consumption, thereby allowing the user to adapt resource consumption as needed.

Our parallelized GPU-PROCLUS and GPU-FAST-PROCLUS are restructured to execute more operations in parallel, and to exploit the memory hierarchy of the GPU.

Our extensive experimental evaluation demonstrates 3 orders of magnitude speedup for GPU-FAST-PROCLUS. This speedup is stable across data distributions and parameter settings. The GPU-parallelizations provide up to 2000× speedup while the algorithmic strategies collectively provide around 2.5× extra speedup.

ACKNOWLEDGMENTS

This work was supported by Independent Research Fund Denmark.

REFERENCES

[1] Andrew Adinetz, Jiri Kraus, Jan Meinke, and Dirk Pleiter. 2013. GPUMAFIA: Efficient subspace clustering with MAFIA on GPUs. In *European Conference on Parallel Processing*. Springer, 838–849.

[2] Charu C Aggarwal, Joel L Wolf, Philip S Yu, Cecilia Procopiuc, and Jong Soo Park. 1999. Fast algorithms for projected clustering. *ACM SIGMoD Record* 28, 2 (1999), 61–72.

[3] Charu C Aggarwal and Philip S Yu. 2000. Finding generalized projected clusters in high dimensional spaces. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 70–81.

[4] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 94–105.

[5] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. 2008. INSCY: Indexing subspace clusters with in-process-removal of redundancy. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 719–724.

[6] Anna Beer, Nadine Sarah Schüler, and Thomas Seidl. 2019. A Generator for Subspace Clusters.. In *LWDA*. 69–73.

[7] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When is "nearest neighbor" meaningful?. In *International conference on database theory*. Springer, 217–235.

[8] Christian Böhm, Claudia Plant, Junming Shao, and Qinli Yang. 2010. Clustering by synchronization. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 583–592.

[9] Stuart K Card, George G Robertson, and Jock D Mackinlay. 1991. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*. 181–186.

[10] Elaine Y Chan, Wai Ki Ching, Michael K Ng, and Joshua Z Huang. 2004. An optimization algorithm for clustering using weighted dissimilarity measures. *Pattern recognition* 37, 5 (2004), 943–952.

[11] Chun-Hung Cheng, Ada Waichee Fu, and Yi Zhang. 1999. Entropy-based subspace clustering for mining numerical data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. 84–93.

[12] Amitava Datta, Amardeep Kaur, Tobias Lauer, and Sami Chabbouh. 2019. Exploiting multi–core and many–core parallelism for subspace clustering. *International Journal of Applied Mathematics and Computer Science* 29, 1 (2019), 81–91.

[13] Zhaohong Deng, Kup-Sze Choi, Fu-Lai Chung, and Shitong Wang. 2010. Enhanced soft subspace clustering integrating within-cluster and between-cluster information. *Pattern recognition* 43, 3 (2010), 767–781.

[14] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[15] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. 2008. A Parallel Implementation of K-Means Clustering on GPUs.. In *Pdpta*, Vol. 13. 212–312.

[16] Sanjay Goil, Harsha Nagesh, and Alok Choudhary. 1999. MAFIA: Efficient and scalable subspace clustering for very large data sets. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Vol. 443. ACM, 452.

[17] Bai Hong-Tao, He Li-li, Ouyang Dan-tong, Li Zhan-shan, and Li He. 2009. K-means on commodity GPUs with CUDA. In *2009 WRI World Congress on Computer Science and Information Engineering*, Vol. 3. IEEE, 651–655.

[18] Jakob Rødsgaard Jørgensen, Katrine Scheel, and Ira Assent. 2021. GPU-INSCY: A GPU-Parallel Algorithm and Tree Structure for Efficient Density-based Subspace Clustering.. In *EDBT*. 25–36.

[19] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. 2004. Density-connected subspace clustering for high-dimensional data. In *Proceedings of the 2004 SIAM international conference on data mining*. SIAM, 246–256.

[20] Amardeep Kaur and Amitava Datta. 2014. Subscale: Fast and scalable subspace clustering for high dimensional data. In *2014 IEEE International Conference on Data Mining Workshop*. IEEE, 621–628.

[21] Kai J Kohlhoff, Marc H Sosnick, William T Hsu, Vijay S Pande, and Russ B Altman. 2011. CAMPAIGN: an open-source library of GPU-accelerated data clustering algorithms. *Bioinformatics* 27, 16 (2011), 2321–2322.

[22] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. 2013. Speeding up k-means algorithm by gpus. *J. Comput. System Sci.* 79, 2 (2013), 216–229.

[23] Guimei Liu, Jinyan Li, Kelvin Sim, and Limsoon Wong. 2007. Distance based subspace clustering with flexible dimension partitioning. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 1250–1254.

[24] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[25] Gabriela Moise and Jörg Sander. 2008. Finding non-redundant, statistically significant regions in high dimensional data: a novel approach to projected and subspace clustering. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 533–541.

[26] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. 2009. Evaluating clustering in subspace projections of high dimensional data. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1270–1281.

[27] David J Newman, SCLB Hettich, Cason L Blake, and Christopher J Merz. 1998. UCI repository of machine learning databases, 1998.

[28] Raymond T. Ng and Jiawei Han. 2002. CLARANS: A method for clustering objects for spatial data mining. *IEEE transactions on knowledge and data engineering* 14, 5 (2002), 1003–1016.

[29] Lance Parsons, Ehtesham Haque, and Huan Liu. 2004. Subspace clustering for high dimensional data: a review. *Acm Sigkdd Explorations Newsletter* 6, 1 (2004), 90–105.

[30] Adhi Prahara, Dewi Pramudi Ismi, and Ahmad Azhari. 2020. Parallelization of Partitioning Around Medoids (PAM) in K-Medoids Clustering on GPU. *Knowledge Engineering and Data Science* 3, 1 (2020), 40–49.

[31] Alex Rodriguez and Alessandro Laio. 2014. Clustering by fast search and find of density peaks. *science* 344, 6191 (2014), 1492–1496.

[32] Ben Shneiderman. 1994. Dynamic queries for visual information seeking. *IEEE software* 11, 6 (1994), 70–77.

[33] Alexander S Szalay, Jim Gray, Ani R Thakar, Peter Z Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. 2002. The SDSS skyserver: public access to the sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 570–581.

[34] Kyoung-Gu Woo, Jeong-Hoon Lee, Myoung-Ho Kim, and Yoon-Joon Lee. 2004. FINDIT: a fast and intelligent subspace clustering algorithm using dimension voting. *Information and Software Technology* 46, 4 (2004), 255–271.

[35] Dongkuan Xu and Yingjie Tian. 2015. A comprehensive survey of clustering algorithms. *Annals of Data Science* 2, 2 (2015), 165–193.