GPUMAFIA: Efficient Subspace Clustering with MAFIA on GPUs

Andrew Adinetz[1],[2] Jiri Kraus[3], Jan Meinke[1], and Dirk Pleiter[1]
(NVIDIA Application Lab at Forschungszentrum Jülich)

[1] JSC, Forschungszentrum Jülich, 52425 Jülich, Germany
[2] Research Computing Center, Lomonosov Moscow State University
[3] NVIDIA GmbH, Germany

Abstract. Clustering, i.e., the identification of regions of similar objects in a multi-dimensional data set, is a standard method of data analytics with a large variety of applications. For high-dimensional data, subspace clustering can be used to find clusters among a certain subset of data point dimensions and alleviate the curse of dimensionality.

In this paper we focus on the MAFIA subspace clustering algorithm and on using GPUs to accelerate the algorithm. We first present a number of algorithmic changes and estimate their effect on computational complexity of the algorithm. These changes improve the computational complexity of the algorithm and accelerate the sequential version by 1–2 orders of magnitude on practical datasets while providing exactly the same output. We then present the GPU version of the algorithm, which for typical datasets provides a further 1–2 orders of magnitude speedup over a single CPU core or about an order of magnitude over a typical multi-core CPU. We believe that our faster implementation widens the applicability of MAFIA and subspace clustering.

1 Introduction

Cluster analysis is a valuable data mining tool. With high-dimensional data occurring in many real applications, traditional all-attribute clustering algorithms encounter problems. Often data points form a cluster only in some dimensions, called *significant dimensions*, while their coordinates in other dimensions show no correlation. Sets of significant dimensions may differ for different clusters. This is part of what is commonly referred to as the curse of dimensionality [1].

Subspace clustering algorithms attempt to find clusters that exist only in subsets of dimensions of the original data, i.e., in subspaces. As they discard insignificant dimensions, they are especially useful in applications where analysis of high-dimensional data is required. In biological research, subspace clustering is used to study gene expression, e.g., to find groups of genes with similar function or to find individuals with similar gene expression. Customer recommendation systems use subspace clustering to find groups of individuals with similar preferences. It is also used to find groups of thematically related documents [2].

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 838–849, 2013.

Here we consider an application to Monte Carlo simulations of protein folding. Proteins are long chains of amino acids that usually adopt a unique three-dimensional shape. This shape is necessary to perform their particular function, e.g., enable an important chemical reaction at body temperature. How these chain molecules are able to fold reliably into these unique shapes (conformations) remains an open question. During simulations millions of conformations are generated and similar conformations have to be grouped into clusters to determine the relative sizes of the clusters. The relative weights relate directly to the free energy, an important physical quantity that tells us, which is the conformation most likely seen in experiments. If the largest clusters are of similar size, a protein might transition between these different states.

While distance-based clustering works, the complexity of the algorithm makes it infeasible to do an exhaustive clustering of all data points. Subspace clustering offers a chance to find the relevant clusters using a d-dimensional state vector that can be calculated with linear complexity. Nevertheless, these algorithms can be computationally expensive. For a d-dimensional dataset, there are $2^d - 1$ possible axis-parallel subspaces. This may result in long run-times. (In practice, the number of subspaces to consider is often much lower.)

Here we focus on MAFIA (Merging of Adaptive Finite IntervAls) [3], a subspace clustering algorithm which applies an adaptive grid method. This reduces the computational requirements while providing similar quality of cluster search. We improve MAFIA by, first, using a number of algorithmic techniques, and second, providing a GPU implementation.

The contributions of this paper are as follows:

- We present a number of algorithmic improvements to the MAFIA subspace clustering algorithm, which gives 1–2 orders-of-magnitude performance increase for the use cases considered here while producing the same output.
- We present a GPU port of MAFIA, which gives an additional 1–2 orders-of-magnitude improvement over a single CPU core, or an order-of-magnitude improvement over a typical multi-core CPU-only system. To our knowledge, this is the first implementation of a subspace clustering algorithm on GPU.
- We present performance analysis, which enables us to justify the algorithmic improvements and parallelization for CPU and GPU architectures.

This paper is organized as follows. Section 2 presents a brief overview of subspace clustering algorithms together with some performance data. It also provides a summary of existing GPU implementations of clustering algorithms. MAFIA is described in section 3 together with an analysis of the operation count. We present the details of algorithmic improvements and GPU implementation in section 4. Finally, we analyse and discuss performance improvements in section 5 and present our conclusions in section 6.

2 Related Work

Subspace clustering is a relatively new field of research. The first two algorithms for finding clusters in subspaces, CLIQUE [4] and PROCLUS [5] were proposed in

1998 and 1999, respectively. MAFIA [3] was introduced shortly thereafter. For a parallel verion, pMAFIA, see [6]. Alternative subspace clustering algorithms are: SeqClus [7], LCM-nCluster [8], Maxncluster [9], and DiSH (Detecting Subspace cluster Hierarchies) [10]. The latter scales super-linearly with the number of points. A good survey of existing subspace clustering algorithms can be found in [2] which, however, does not include a comparison of performance or quality. This has been done elsewhere. For example, in [11] MAFIA and FINDIT are compared. In 2004, both required around 10 minutes to process a 20-dimensional 4-million point set with 5 hidden 5-dimensional clusters. [12] indicates that runtimes of SUBCLU can be on the order of several hours even for relatively small dataset. However, SUBCLU is also better at finding subspace clusters, compared to CLIQUE. In [9] Maxncluster is compared to MAFIA and STATPC. Another good survey of subspace clustering performance is [13], which, however, excludes MAFIA.

As clustering algorithms are generally computationally intensive and parallelizeable, they are prime candidates for implementation on GPUs. K-means is perhaps the most widely implemented GPU clustering algorithm [14, 15, 16]. CAMPAIGN, an open-source clustering library [17], comprises an implementation of K-means and other algorithms. Other clustering algorithms implemented on GPUs include fuzzy clustering [18, 19], multi-level clustering [20] and density-based clustering [21]. A distinguishing characteristic of MAFIA is that most of its time is spent in operations on sets of data points, as opposed to the points themselves. To the best of our knowledge, there is no publicly available GPU implementation of a subspace clustering algorithm.

3 Algorithm Analysis

MAFIA starts with breaking each dimension into bins, counting points in each bin, and building *windows* as part of an adaptive grid approach. Initially, each window is a *candidate dense unit* of dimensionality 1 (1-CDU). CDUs that are dense enough become *dense units* (DUs). MAFIA then builds CDUs of increasing dimensionality; an a-CDU is build by merging two $(a-1)$-DUs which are the same in $(a-2)$ dimensions. $(a-1)$-DUs which were not joined into a-DUs are then added to the list of *terminal DUs*. Finally, connected groups of DUs of the same dimensionality are merged into clusters. For a high-level pseudo-code version of the algorithm see Alg. 1. Note that the collections used are arrays, not sets, and may therefore contain duplicate elements.

The algorithm can be broken into three phases: the loop over the dimension of CDUs is the *middle phase*, and what precedes and follows are the *initial* and *final* phases, respectively. The main kernel of the initial phase is the histogram construction **histogram**. During the same phase also the adaptive grid is built (**windows**). The kernels of the middle phase are CDU generation (**gen**), CDU deduplication (**dedup**), finding dense CDUs or point counting (**pcount**), and check for unjoined DUs (**unjoin**). For **gen**, **dedup** and **unjoin**, an $O(N^2)$ algorithm is used, while for **pcount**, each point is just checked against bounds of

Algorithm 1. High-level pseudo-code for MAFIA algorithm

n — number of points, d — number of dimensions
p_{ij} — point coordinates, $1 \leq i \leq n$, $1 \leq j \leq d$
α — threshold parameter, N_b — min. #bins, N_M — max. # windows, N_{uw} — #windows for uniform dimensions
for $j = 1 \rightarrow d$ **do**
  $D_j \leftarrow \max_{1 \leq i \leq n} p_{ij} - \min_{1 \leq i \leq n} p_{ij}$
  $h_j \leftarrow histogram(p, j, N_b)$
  $W \leftarrow W \cup adaptiveGrid(h_j, N_{uw}, N_M)$
end for
$w \in W | t_w \leftarrow \frac{\alpha n (r_w - l_w)}{D_{j_w}}$
$CDUs \leftarrow \{\{w\} | w \in ws\}$, $a \leftarrow 1$
while $DUs \neq \emptyset \vee a = 1$ **do**
  if $a > 1$ **then**
    $CDUs \leftarrow \{u_1 \cup u_2 | (u_1, u_2) \in pairs(DUs) \wedge canMerge(u_1, u_2)\}$
  end if
  $CDUs \leftarrow dedup(CDUs)$ ▷ deduplication
  $u \in CDUs | ns_u \leftarrow \|\{i \in 1 \rightarrow n | \forall w \in u : l_w \leq p_{ij_w} < r_w\}\|$ ▷ point counting
  $newDUs \leftarrow \{u \in CDUs | \forall w \in u : ns_u \geq t_w\}$
  $termDUs \leftarrow termDUs \cup \{u \in DUs | \nexists u_1 \in newDUs : u \in u_1\}$ ▷ unjoined check
  $a \leftarrow a + 1$, $DUs \leftarrow newDUs$
end while
$G \leftarrow \{termDUs, \{(u_1, u_2) \in pairs(termDU) | haveCommonFace(u_1, u_2)\}\}$
$cs \leftarrow connectedComponents(graph)$
$c \in cs | inds_c \leftarrow \{i \in 1 \rightarrow n | p_i \in c\}$ ▷ index lists

each window belonging to each CDU. The final phase consists of building the DU graph and finding connected components (**graph**), as well as building lists of points belonging to given clusters (**list**). The **graph** and **windows** kernels do not process large amounts of data, and are ignored in our performance analysis.

We now estimate the computational complexity of the kernels in a special case. We assume that the dataset consists of n d-dimensional points, from which a fraction f belongs to m clusters, all of dimensionality k. Furthermore, to simplify analysis, we assume that clusters are arranged in such a way that $(a-1)$-DUs belonging to different clusters do not merge, i.e., they do not have a common $(a-2)$-sub-DU. If the overlap does occur, the complexity of the middle phase becomes even higher, and so does the gain from using GPUs.

Table 1 lists the operation counts for each kernel for fixed DU dimensionality a and the total costs which is obtained by summing over $a = 1, \ldots, k$. We dropped components with lesser order-of-magnitude and used Stirling's approximation. When a-CDUs are being generated, there are $m\binom{k}{a-1}$ $(a-1)$-DUs. The number of operations to check any pair for merging is a, which gives the cost of generating a-CDUs. There are $m\binom{k}{a}$ a-CDUs generated. Each a-CDU is generated $a(a+1)/2$ times, as this is the number of $(a-1)$-DU pairs which merge into this a-CDU, which gives the cost of deduplication. Points must be counted for each a-CDU, and a dimensions must be checked for each point, which gives the cost of counting

Table 1. Operation count for fixed dimensionality a as well as the total costs

Kernel	**Costs for fixed a**	**Total costs**
histogram	—	$O(nd)$
gen	$\frac{a}{2}m\binom{k}{a-1}(m\binom{k}{a-1}-1)$	$O(m^2\sqrt{k}4^k)$
dedup	$\frac{a}{2}\frac{ma(a+1)}{2}\binom{k}{a}(\frac{ma(a+1)}{2}\binom{k}{a}-1)$	$O(m^2k^4\sqrt{k}4^k)$
pcount	$m\,a\,n\binom{k}{a}$	$m\,n\,k\,2^{k-1}$
unjoin	$am^2\binom{k}{a-1}\binom{k}{a}$	$O(m^2\sqrt{k}4^k)$
list	—	$O(nf)$

points. For unjoined check, each $(a-1)$-DU should be checked against all a-DUs, and the cost of a single check is a.

4 Optimization

We first introduce a number of algorithmic improvements. Kernel costs in Table 1 show that there are two possible performance bottlenecks:

- If the number of points is large and the cluster dimensionality is small then the **pcount** kernel will dominate.
- For small to average numbers of points and big cluster dimensionality, kernels with operation counts independent of the number of points, i.e., **gen**, **unjoin**, and, most importantly, **dedup** will start to dominate.

We first optimize the **dedup** kernel by replacing the originally used $O(N^2)$ algorithm by $O(N \log N)$ set deduplication, where the set is implemented as a tree. (N is the number of CDUs at current iteration.) CDU order is defined as lexicographic order of sequences of dimension and window numbers. We also merge the kernels **dedup** and **gen**: A newly generated CDU will be added to the set only if there has not been one previously generated.

We then consider the kernels **gen** and **unjoin**. For **gen**, we build a map from $(a-2)$-subsequences to lists of $(k-1)$-DUs containing that subsequence. A $(a-1)$-DU belongs to the list only if it contains the subsequence. For **unjoin**, we similarly build a set of possible $(a-1)$ subsequences of a-DUs and check each $(a-1)$-DU against that set. Both kernels then also have $O(N \log N)$ complexity.

The optimizations described are simple but, to the best of our knowledge, they have never been implemented so far. As will be shown in the next section the improvement can be large.

For **pcount**, no point index optimization is possible due to high data dimensionality. What MAFIA really needs to do is to compute the intersection of window point sets, and then calculate its cardinality. We use bit arrays to represent those sets, and store the sets of dense windows only. The intersection can now be computed using simple bit-wise and operations. The number of points is given by the number of enabled bits. The actual operation count thus reduces by

Table 2. Same as Table 1 but for improved kernels

Kernel	**Costs for fixed** a	**Total costs (optimized)**	**Total costs (unoptimized)**
bitarray	—	$O(mnk)$	—
gen	$(a-1)am\binom{k}{a-1}\log\{m\binom{k}{a-2}\}$	$O(m(k+\log m)k^2 2^k)$	$O(m^2\sqrt{k}4^k)$
dedup	$a\frac{ma(a+1)}{2}\binom{k}{a}\log\{m\binom{k}{a}\}$	$O(m(k+\log m)k^3 2^k)$	$O(m^2k^4\sqrt{k}4^k)$
pcount	$\frac{man\binom{k}{a}}{32}$	$\frac{mnk}{64}2^k$	$\frac{mnk}{2}2^k$
unjoin	$2am(a-1)\binom{k}{a}\log\{m\binom{k}{a-1}\}$	$O(m(k+\log m)k^2 2^k)$	$O(m^2\sqrt{k}4^k)$

about 32× because now 1 operation taking 32-bit operands is sufficient to process 32 points. This also reduces memory bandwidth requirements while upfront costs for building the bit arrays are small.

In Table 2 we show how the algorithmic improvements and the use of bit arrays improves the operation count. The costs of **gen**, **dedup** and **unjoin** still grow exponentially with k, but the exponent is reduced from 4 to 2, which makes the algorithm applicable to datasets with higher k. Costs of point counting are cut by a factor of 32×, which makes the algorithm applicable to larger datasets with the same cluster configuration. Note that as a side effect, the algorithmic improvements also reduced dependency on the number of clusters of **gen**, **dedup** and **unjoin** kernels from m^2 to $m \log m$, which is beneficial for real-world applications with datasets containing many clusters.

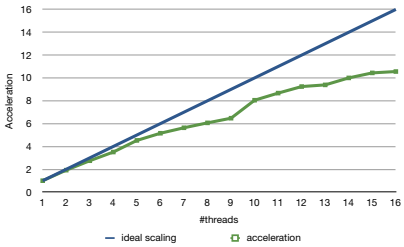
We now consider parallelization and porting of the most performance critical kernels to the GPU. For CPU parallelization we use OpenMP, GPU kernels are implemented using CUDA.

In case of the kernel **histogram** we parallelize both the dimension and point loops. For the latter, CPU threads compute private histograms, which are then summed up. On the GPU shared-memory atomics are used. (Further speed-up may be obtained using warp-synchronous non-atomics operations.)

pcount is a doubly nested loop, the outer over the CDUs and the inner over bit array words. On the CPU both are parallelized. On the GPU, the loops are mapped into different dimensions of the CUDA thread-block grid. Each thread adds up points from several words (128 in the current implementation) of bit arrays using `__popc()`, a CUDA built-in function, to count bits in each word, and global-memory atomics to compute the final point count. We also implemented precomputing of bit array indices in shared memory, and allocated bit arrays using `cudaMallocPitch()`, which cumulatively resulted in a 73% performance improvement over the initial GPU version.

bitarray is also a doubly nested loop, the outer over windows and the inner over words or points. Parallelization is thus similar as for **pcount**. On the GPU global-memory atomics are used to set individual bits. We found the memory access pattern to be better for the point-per-thread approach than for using a separate thread for each word.

(a) Speed-up due to algorithmic improvements as a function of the cluster dimensionality k.



(b) Scaling of MAFIA as a function of the number of OpenMP threads.

Fig. 1. CPU-only implementation of MAFIA

5 Performance Evaluation and Discussion

We implemented MAFIA as a standalone application. For benchmarking we used a dual-socket server comprising 2 8-core Intel Xeon E5-X2670 CPUs running at 2.60 GHz, plus an NVIDIA Tesla K20X (Kepler) GPU. For the experiments, hyper-threading and frequency scaling on CPU were turned off, and ECC was turned on on the GPU.

We first conducted a CPU-only single-core test to evaluate the algorithmic improvements. For this, we generated a series of datasets, each containing 10^5 30-dimensional points and a single embedded cluster, whose dimensionality k varied from 3 to 17. The results of the evaluation are shown in Fig. 1a. For small k, the cost of building the histograms dominates and optimization has little effect. For increasing k the algorithmic improvements start to have an increasingly large effect when the kernels **pcount** and later **dedup** start to dominate execution time. For $k = 17$ we observe almost two orders of magnitude speed-up. For further discussion, we consider only the version with all algorithmic improvements applied.

Fig. 1b shows scaling of MAFIA as a function of the number of OpenMP threads for 10^5 10-dimensional points and a single cluster with $k = 10$. We observe a large deviation from perfect scaling for increasing number of threads. Performance improvement does not saturate before all the cores are used.

To investigate acceleration on the GPU as well as CPU parallelization we generated datasets comprising 10^7 20-dimensional points and 3 clusters, whose dimensionality k varied from 3 to 16. The different clusters did not intersect in any dimension, which ensures their successful detection by MAFIA. In Fig. 2a and 2b we plot the execution time of each kernel as a function of k for the CPU-only and the GPU-accelerated version, respectively. Runtime of MAFIA grows exponentially with k in all cases, approximately doubling for each successive k, which agrees with operation counts in Table 2. For small k, initial and final phases take significant part of the overall computing time. For higher values of k, the exponentially-growing middle phase, and most importantly for this

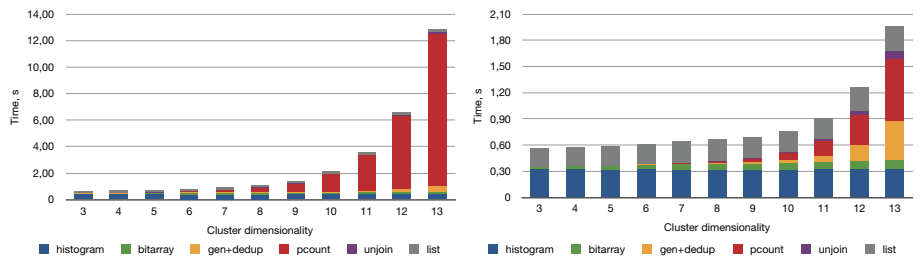
parameter combination, point counting, dominate execution time. For CPU, **pcount** dominates clearly. With k further increasing, **gen**, **dedup** and **unjoin** start to play a greater role for both architectures, because their operation count grows as $k^4 2^k$. For GPU, times spent in **pcount** and **gen** + **dedup** are already comparable even for mid-range k, as only the former kernel has been ported to GPU. Both CPU and GPU parallelization give considerable performance improvement over sequential version.

The **histogram** part for GPU time breakup mostly consists of transferring initial data to device. As actual histogram computation takes only a small fraction of the **histogram** time, it is not possible to hide data transfer.

For the **pcount** kernel, as the number of memory reads is known exactly, we can estimate effective memory bandwidth achieved by the kernel. For K20X, this is 368 GB/s, much higher than the specification value (250 GB/s) and Stream benchmark value (180 GB/s). However, the kernel is still memory-bound, and the high bandwidth achieved is due to sharing of the same windows, and therefore same bit arrays, between neighboring CDUs, which enables caching to take effect.

Overall execution times are given in Fig. 3. **seq** stands for sequential version, **par4**, **par8** and **par16** are 4-, 8- and 16-thread parallel versions, respectively, and **k20x** is the GPU version. Accelerations for parallel vs. sequential CPU are given in Fig. 4a, while Fig. 4b gives acceleration of GPU vs. sequential CPU version. The GPU-accelerated version does outperform the parallel CPU-only implementation in all the cases. For small k the acceleration is small due to the time needed to transfer data to the GPU. For larger k, as the relative contribution of **pcount** grows, so does the acceleration, peaking at $7\times$ for $k = 14$. As not all systems have 16 CPU cores, we believe that for a typical system, GPU will give at least an order of magnitude acceleration over the parallel CPU version. When compared to a single CPU core, GPU gives more than two orders of magnitude acceleration. However, for $k > 15$ this value starts decreasing, due to increasing role of sequential **gen** + **dedup** kernels.

To test the applicability of MAFIA to protein folding simulations, we used data from a parallel tempering Monte Carlo simulation of the last 16 residues of protein G. This simulation produced 160000 independent conformations. For ease of



(a) 12 x86 CPU cores (b) Kepler K20X GPU

Fig. 2. MAFIA execution time breakdowns in various settings

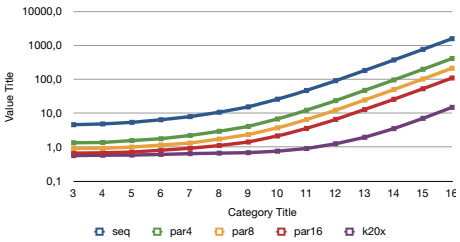
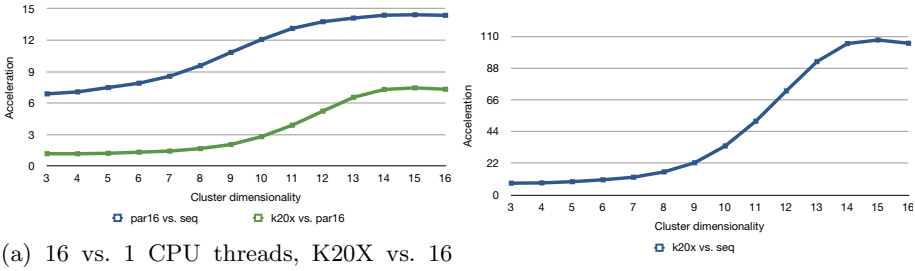


Fig. 3. MAFIA execution times in different settings



(a) 16 vs. 1 CPU threads, K20X vs. 16 CPU threads

(b) K20X vs. 1 CPU core

Fig. 4. MAFIA acceleration in different settings

visualization, we take only 4 properties of the conformation to form the state vector: the temperature at which this confirmation was found, the energy of the conformation, the fraction of residues that is part of a strand called the strand content, and the root-mean-square deviation (RMSD) to the native conformation. This data set is not meant to stress the performance of the algorithm in terms of the run-time (approximately 1 s). Rather, comparing expected and obtained results is an additional check of both algorithm and implementation. Furthermore, it allows to test how the parameters need to be adapted to deal with a real world data. Finally, in combination with our analysis of the algorithm's complexity the example demonstrates that datasets of realistic size could be processed in just $O(1)$ minutes in case of larger cluster dimensionality $k \simeq 10$.

Figure 5 shows the clusters that we obtained using 15 windows and $\alpha = 0.075$. The density threshold needs to be so small to detect the low energy clusters. At $\alpha = 1.5$ MAFIA didn't detect any clusters. We chose 15 windows since there are 15 discrete values for the strand content.

MAFIA identified several low-energy as well as several high-energy clusters. While the separation of structures into different clusters is not obvious, the fact that we retain the original axes makes interpretation of the clusters easier than interpreting clusters obtained from principal component analysis, for example.

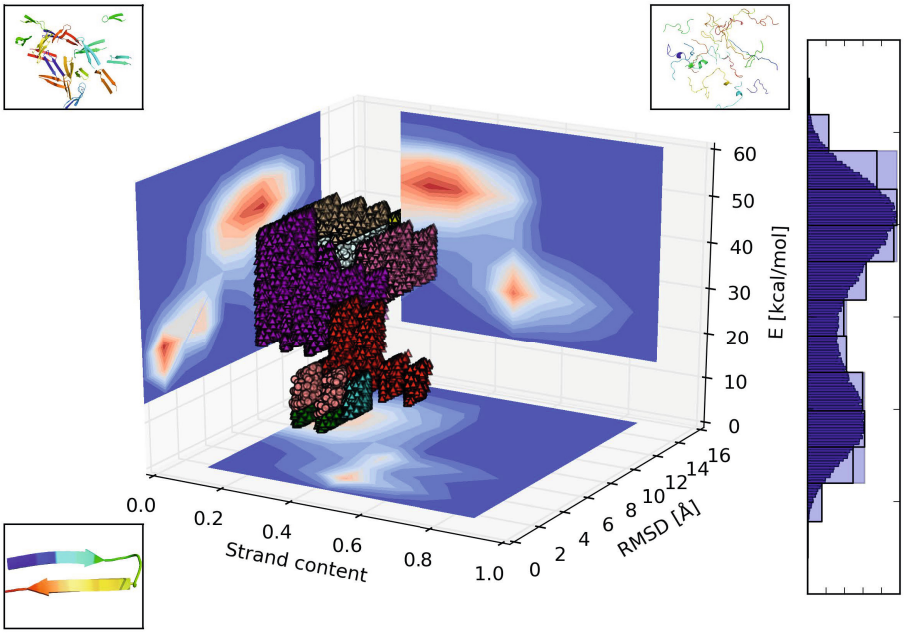


Fig. 5. Clusters of conformations from a Monte Carlo protein folding simulation in the S-R-E subspace including a low-energy and a high-energy 4D cluster (spheres). The images in the corners of the figure show the reference structure (lower left corner) and a random selection of conformations from the low-energy (upper left corner) and the high-energy (upper right corner) 4D clusters. The background contour plots show the 2D histograms in the corresponding planes. On the right is the 1D histogram of energy showing the initial fine bins, the 15 initial windows (lines) and the final merged windows (light blue filled bars).

6 Conclusions

In this paper, we studied porting MAFIA to GPUs. We first performed a number of algorithmic improvements, and then developed a GPU implementation. The algorithmic changes resulted in an almost two-orders-of-magnitude improvement. Porting to GPU accelerated the application by another order of magnitude. Our results put MAFIA subspace clustering well within the limits of interactivity even for large datasets with moderate cluster dimensionality (≤ 15).

Our MAFIA implementation can still be improved. The kernels **gen**, **dedup** and **unjoin** can dominate execution time if the number of points is small. Thus, parallelizing them should be considered, though it might be difficult, as they all access a single set for reading and writing. Porting them to a GPU may also be considered, though this is an even more difficult task. For datasets with larger number of points, parallelization across multiple GPUs or even across cluster

nodes is to be considered; as **pcount** performs only per-point operations, this should be simple.

Also, as the run-time of the algorithm is significantly improved, I/O, and more specifically, converting large input text files into data point arrays, which is part of many work-flows involving MAFIA, becomes a bottleneck. One way to remove it is to accelerate string-to-double parsing on GPU, and do this in pipelined fashion. This would enable hiding initial data transfer to GPU, and improve the overall performance without changing the current workflow. Restructuring the implementation into a library accepting points from CPU or GPU memory rather than reading them from a file is another possibility for performance improvement.

Acknowledgements. We would like to thank Wilhelm Homberg for discussions and support of this work.

References

[1] Bellman, R.: Dynamic Programming (Dover Books on Computer Science). Dover Publications (2003)
[2] Kriegel, H.P., Kröger, P., Zimek, A.: Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. ACM Trans. Knowl. Discov. Data 3(1), 1:1–1:58 (2009)
[3] Nagesh, H.S.: High Performance Subspace Clustering for Massive Data Sets. Master's thesis (1999)
[4] Agrawal, R., Gehrke, J., Gunopulos, D., Raghavan, P.: Automatic subspace clustering of high dimensional data for data mining applications. SIGMOD Rec. 27(2), 94–105 (1998)
[5] Aggarwal, C.C., Wolf, J.L., Yu, P.S., Procopiuc, C., Park, J.S.: Fast algorithms for projected clustering. SIGMOD Rec. 28(2), 61–72 (1999)
[6] Nagesh, H., Goil, S., Choudhary, A.: Parallel Algorithms for Clustering High-Dimensional Large-Scale Datasets. Kluwer (2001)
[7] Wang, H., Chu, F., Fan, W., Yu, P.S., Pei, J.: A fast algorithm for subspace clustering by pattern similarity. In: Proceedings of the 16th SSDBM, pp. 51–62 (2004)
[8] Liu, G., Li, J., Sim, K., Wong, L.: Distance based subspace clustering with flexible dimension partitioning. In: IEEE 23rd International Conference on Data Engineering, ICDE 2007, pp. 1250–1254 (April 2007)
[9] Liu, G., Sim, K., Li, J., Wong, L.: Efficient mining of distance-based subspace clusters. Statistical Analysis and Data Mining 2(5-6), 427–444 (2009)
[10] Achtert, E., Böhm, C., Kriegel, H.-P., Kröger, P., Müller-Gorman, I., Zimek, A.: Detection and visualization of subspace cluster hierarchies. In: Kotagiri, R., Radha Krishna, P., Mohania, M., Nantajeewarawat, E. (eds.) DASFAA 2007. LNCS, vol. 4443, pp. 152–163. Springer, Heidelberg (2007)
[11] Parsons, L.: Evaluating subspace clustering algorithms. In: Workshop on Clustering High Dimensional Data and its Applications, SIAM International Conference on Data Mining (SDM 2004), pp. 48–56 (2004)
[12] Kröger, P., Kriegel, H.P., Kailing, K.: Density-Connected Subspace Clustering for High-Dimensional Data. In: SDM (2004)

[13] Müller, E., Günnemann, S., Assent, I., Seidl, T.: Evaluating clustering in subspace projections of high dimensional data. Proc. VLDB Endow. 2(1), 1270–1281 (2009)
[14] Cao, F., Tung, A.K.H., Zhou, A.: Scalable clustering using graphics processors. In: Yu, J.X., Kitsuregawa, M., Leong, H.-V. (eds.) WAIM 2006. LNCS, vol. 4016, pp. 372–384. Springer, Heidelberg (2006)
[15] Wu, R., Zhang, B., Hsu, M.: Clustering billions of data points using GPUs. In: UCHPC-MAW 2009, pp. 1–6. ACM, New York (2009)
[16] Hong-Tao, B., Li-li, H., Dan-Tong, O., Zhan-Shan, L., He, L.: K-Means on Commodity GPUs with CUDA. In: 2009 WRI World Congress on Computer Science and Information Engineering, March 31-April 2, vol. 3, pp. 651–655 (2009)
[17] Kohlhoff, K.J., Sosnick, M.H., Hsu, W.T., Pande, V.S., Altman, R.B.: CAMPAIGN: An open-source Library of GPU-accelerated Data Clustering Algorithms. Bioinformatics (2011)
[18] Kim, S., Wunsch, D.: A GPU based Parallel Hierarchical Fuzzy ART clustering. In: The 2011 International Joint Conference on Neural Networks (IJCNN), July 31-August 5, pp. 2778–2782 (2011)
[19] Anderson, D., Luke, R., Keller, J.: Speedup of Fuzzy Clustering Through Stream Processing on Graphics Processing Units. IEEE Transactions on Fuzzy Systems 16(4), 1101–1106 (2008)
[20] Chiosa, I., Kolb, A.: GPU-Based Multilevel Clustering. IEEE Transactions on Visualization and Computer Graphics 17(2), 132–145 (2011)
[21] Böhm, C., Noll, R., Plant, C., Wackersreuther, B.: Density-based clustering using graphics processors. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, pp. 661–670. ACM, New York (2009)