A Survey of CUDA-based Multidimensional Scaling on GPU Architecture

Hasmik Osipyan[1], Martin Kruliš[2], and Stéphane Marchand-Maillet[3]

1 **National Polytechnic University of Armenia**
Yerevan, Armenia
`hasmik.osipyan.external@worldline.com`

2 **Charles University in Prague**
Prague, Czech Republic
`krulis@ksi.mff.cuni.cz`

3 **University of Geneva**
Geneva, Switzerland
`stephane.marchand-maillet@unige.ch`

Abstract

The need to analyze large amounts of multivariate data raises the fundamental problem of dimensionality reduction which is defined as a process of mapping data from high-dimensional space into low-dimensional. One of the most popular methods for handling this problem is multidimensional scaling. Due to the technological advances, the dimensionality of the input data as well as the amount of processed data is increasing steadily but the requirement of processing these data within a reasonable time frame still remains an open problem. Recent development in graphics hardware allows to perform generic parallel computations on powerful hardware and provides an opportunity to solve many time-constrained problems in both graphical and non-graphical domain. The purpose of this survey is to describe and analyze recent implementations of multidimensional scaling algorithms on graphics processing units and present the applicability of these algorithms on such architectures based on the experimental results which show a decrease of execution time for multi-level approaches.

1998 ACM Subject Classification C.1.2 Multiple Data Stream Architectures (Multiprocessors), D.1.3 Concurrent Programming, I.2.6 Learning, I.3.1 Hardware Architecture

Keywords and phrases Dimensionality Reduction, Pattern Recognition, Graphics Hardware, Compute Unified Device Architecture, Parallel Programming

Digital Object Identifier 10.4230/OASIcs.ICCSW.2015.37

1 Introduction

In different pattern recognition problems, a large amount of measurements is obtained from hardware devices. Applying clustering methods or classifiers to these measurements is problematic due to the parameter estimation problem which arises when learning methods applied to high-dimensional data sets with a limited number of samples. In order to handle these problems, the dimensionality reduction techniques are used which helped to extract a small number of useful features from the set of many measurements [1].

High-dimensional data sets exist in most scientific domains such as visualization [4], classification [5], recommender systems/filtering [6, 7], signal/image processing [8, 9], etc. That is why the contributions to this field have come from many disciplines and approaches to the problem are quite different. In the recent years, many different linear and non-linear dimensionality reduction techniques such as multidimensional scaling (MDS) [14], t-SNE algorithm [2] and locally linear embedding (LLE) [3] have been developed for using in different applications. But the main idea behind all these dimensionality reduction techniques is the same: transformation of high n-dimensional input data $X = x_1, x_2, ..., x_n$ into a relevant k-dimensional space $Y = y_1, y_2, ..., y_k$, where $k \leq n$ (and possibly $k << n$). Based on the existing implementations/adaptations of these techniques on multi-core architectures and due to a limited space, in this paper we will only focus on the MDS algorithm. Albeit an early dimensionality reduction method, the MDS is still very popular in the implementation of interactive visualization technique in the context of visual analytics [29] triggered by the need to inspect Big Data visually.

Unfortunately, classical MDS quickly became quite expensive for high-dimensional data sets, where both the number of variables and samples are high. Therefore, there is a real need to reduce execution time of these algorithms by suggesting a faster algorithms or implementing existing ones on parallel architectures such as graphics processing units (GPUs).

With the evolution of the NVIDIA CUDA API (Compute Unified Device Architecture Application Programming Interface) [15], GPU hardware becomes more power-efficient than central processing units (CPUs). Unlike CPUs, GPUs give the low cost alternative for implementing non-graphical problems in parallel. However, achieving a good utilization of GPU requires not only careful implementation which needs detailed understanding of the underlying architecture, but also the right usage of all the capabilities provided by these units.

In this survey, we review the multidimensional scaling – a more recent state-of-the-art dimensionality reduction technique – and discuss its adaptation on the GPU architecture. To analyze the performance issues of MDS algorithms in a good way we have also run experiments for the same hardware and data set. Earlier survey papers [10, 11, 12, 13] review dimensionality reduction methods on classical CPUs and, to the best of our knowledge, this is the first survey paper which reviews the adaptation of MDS algorithms on the GPU architecture.

The rest of this paper is organized as follows. Sections 2 briefly reviews the MDS algorithm. Section 3 introduces the fundamentals of GPU architecture as well as the parallel programming model adapted to program using the all advantages of these units. Section 4 shows the adaptation of MDS methods on the GPU. Finally, we discuss the results based on the experiments in Section 5 and conclude in Section 6.

2 Multidimensional Scaling

In this section we review the classical MDS and emphasize the main computationally intensive components of the algorithm. Classical MDS algorithms take an $n \times n$ square matrix D containing all possible dissimilarities between n data objects and map these dissimilarities into a lower dimensional Euclidean space. The goal of MDS is to minimize a loss function which measures the lack of fit between the distances of the lower dimensional objects $\|x_i - x_j\|$ (where $\|.\|$ is the Euclidean distance) and the dissimilarities d_{ij} of the full dimensional data objects (Eq. 1) .

$$\sum_{i,j}(\|x_i - x_j\| - d_{ij})^2 \rightarrow min \tag{1}$$

In order to compute classical MDS, the input matrix D is first converted into a dot product matrix (Eq. 2).

$$J = I - \frac{1}{n}[1], B = -\frac{1}{2}JD^2J \tag{2}$$

Then, the m largest positive eigenvalues $\gamma_1, \ldots, \gamma_m$ and eigenvectors $\varepsilon_1, \ldots, \varepsilon_m$ of B matrix are extracted from which the final low-dimensional data set $x_1, x_2, ..., x_n$ is computed as

$$X = E_m\sqrt{\gamma_m} \tag{3}$$

where γ_m is the diagonal matrix of the m eigenvalues and E_m is the matrix of m eigenvectors.

Solving the eigenpairs problem is the main bottleneck of the original MDS algorithm in terms of execution time. The demand of working with large scale data sets creates a need for a new algorithms with reduced time complexity. For example, Morrison [16] suggested a new hybrid approach which is based on sampling, interpolation, and the use of spring models [17]. They showed the effectiveness of their approach by reducing the computational complexity from $O(n^2)$ to $O(n\sqrt{n})$. In another work of Yang et al. [18], the authors presented new sampling-based fast approximation method by dividing the original matrix into sub-matrices and then combining sub-solutions. For his part, this method reduced the time complexity to $O(n \log n)$. But even with these optimizations, more rapid processing (e.g. parallel) is required when the number of input objects reaches the order of millions.

Besides the classical MDS algorithm, other types such as metric MDS, non-metric MDS, or generalized MDS exists. Furthermore, there are various approaches to implement a given MDS problem. For example, conventional methods for classical scaling can be solved by eigendecomposition, while the distance scaling methods demand iterative minimization which is much more expensive in terms of execution time [10, 11, 12, 13]. The methods which have proven to be applicable for GPU architectures are compared in Section 4.

3 GPU Fundamentals

GPU architectures differ from CPUs in multiple ways. In this section, we present the GPU architecture fundamentals with particular emphasis on aspects, which have a great importance in the light of the studied problem.

A GPU card is a peripheral device with its own memory which is connected to the host system via the PCI-Express (PCIe) bus. It has several *streaming multiprocessor* units (SMPs), which share only the main memory bus and the L2 cache of the GPU.

The GPU architecture follows the Single Instruction Multiple Thread (SIMT) execution model by processing multiple data items simultaneously by the same function called *kernel*. When a kernel is invoked, the caller specifies, how many threads are attached to this kernel. Each thread executes the kernel code, but has an unique thread ID, which is used to identify the portion of the work processed by the thread.

Threads are grouped together into blocks of the same size. Threads from different blocks run independently as they are usually assigned to different SMPs. On the other hand, threads within one block may synchronize their work efficiently using barriers and closely cooperate using internal resources of the SMP, especially the shared memory. Threads in a block are divided into subgroups called *warps* (all current NVIDIA architectures use 32 threads per warp) and they are executed simultaneously in a lockstep, which means they are all issued the same instruction at a time.

GPU hardware has its own memory hierarchy which is shown in Figure 1. The *host memory* is the operational memory of the computer which directly cannot be accessed by

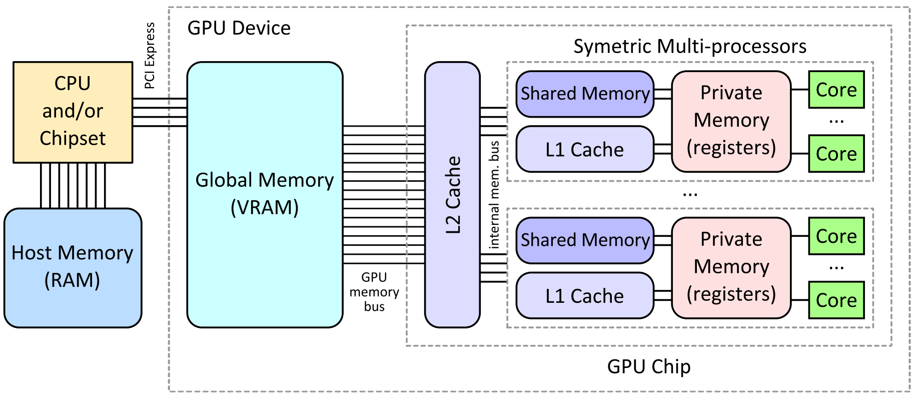

Figure 1 Host and GPU memory organization scheme.

the GPU device. Before execution, the input data need to be transferred from the host memory (RAM) to the graphic device global memory (VRAM). Data are transferred via the PCI-Express bus which is rather slow (8 GB/s) in comparison to the internal memory buses.

The *global memory* can be accessed from the GPU cores, so the input data and the results computed by a kernel are stored here. The *shared memory* is shared among threads within one group. It is rather small (tens of KB) but almost as fast as the GPU registers which makes it quite effective for intermediate results exchange between the threads in the block. The *private memory* belongs exclusively to a single thread and corresponds to the GPU core registers. Its size is very limited (tens to hundreds of words per thread), so it is suitable only for a few local variables. Furthermore, a GPU employs two level of caches to reduce the latency of memory transfers. The *L2 cache* is shared by all SMPs and caches all access to global memory. The *L1 cache* is private to each SMP and caches data from global memory selectively by a specialized loading instruction.

One of the critical aspects of developing GPU algorithms is the thread synchronization and data exchange. These operations disrupt the work flow of individual threads and reduce the utilization of GPU cores. Hence, they can severely limit the performance of any parallel algorithm.

4 MDS Approaches Utilizing GPU Architectures

In this section, we are going to review the most recent approaches of MDS algorithm on GPUs by highlighting the main performance issues.

4.1 Single-level Approaches

Fester et al. [19] proposed a CUDA implementation of MDS algorithm based on the high-throughput multidimensional scaling (HiT-MDS) [20]. As the time consuming part of the HiT-MDS algorithm is the computation of Pearson correlation coefficient, the steps of HiT-MDS which implemented on the GPU are as follows:

1. Partial-sum calculation – time complexity of this task for parallel implementation is $O(\log n)$ performing only $O(n)$ addition operations [21]. To use all the advantages of CUDA parallel model, authors used the shared memory of SMPs to load data portion by portion and compute partial-sum by threads within the block.

2. Euclidean distance calculation – as this operation should be done for each data point it best suits on the GPU architecture when each thread is responsible for calculating the distance of one point.
3. Floyd-Warshall algorithm – an additional operation for getting information out of sparse graphs. It is based on the work of Harish [22] and uses the texture memory access method.
4. Degree based distance manipulation – also an additional operation for visualizing network structures by computing the distances out of adjacency. In this step, CUDA was used only for visualization of the node positions.

In another approach [25], authors suggest a new efficient parallel algorithm for MDS mapping based on virtual particle dynamics (VPD-MDS) [26] and demonstrated the performance of this algorithm on the GPU architecture. The VPD-MDS uses cost function converted into a more general form (Eq. 4).

$$V(X) = \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} V(\|D_{ij}^k - d_{ij}^k\|) \tag{4}$$

The $V()$ is the difference of distances d_{ij} and respective dissimilarities D_{ij} of points in X where the points interact with each other via partial forces $f_{ij} = - \bigtriangledown V(\|D_{ij} - d_{ij}\|)$. As the most expensive part of this algorithm in terms of complexity is the computation of the partial forces ($O(M^2)$), in the paper authors concentrated on the parallelization of that procedure.

The implementation computes only the partitioning of the data and the calculation of the partial forces on the GPU. In addition, they utilize asynchronous data transfers, which means that the read/write operations from/to global memory overlapped with the calculations. One of the greatest challenges solved by this implementation was the problem of the size of the dissimilarity matrix which may not fit the (rather limited) memory of the GPU. To overcome this problem, authors proposed a parallel implementation using MPI (Message Passing Interface) library on a cluster where the data can be distributed among several GPUs and processors.

4.2 Multi-level Approaches

Though the iterative approaches of MDS algorithm are more flexible because they find a stress-optimal configuration, they do not always guarantee an optimal solution due to the difficulty of finding an appropriate termination point. To overcome this problem, the Glimmer algorithm [23] for the GPU architectures was proposed. It divides the input data into hierarchical levels and executes the algorithm recursively. The method is based on a stochastic approach [17].

In brevity, the algorithm extracts subsets from the input data where each level is obtained from the parent level (*restriction*). The stochastic force algorithm is performed on each level starting at the bottom (*relaxation*) and the results of each level are interpolated up to the parent level (*interpolation*). The algorithm finishes when the stochastic force algorithm have processed the highest level of the hierarchy which contains all points of the data set.

The restriction procedure randomly splits data into the levels and due to the inefficient implementation on the GPU, this portion kept on the CPU side. During stochastic force, a set of *near points* and a set of *random points* are defined for every data point. These sets are filled randomly at the beginning and after each iteration, the sets are updated according to the distances between data points. The stress error metric, which is used in the termination condition, is defined to approximate $\epsilon = \frac{1}{10000}$ value.

After the restriction procedure, the GPU stochastic force algorithm (GPU-SF) keeps the data in the texture memory. At the beginning, a random index set index is defined for each data point and a set of high and low dimensional (Euclidean) distances is calculated to simplify the subsequent updates of the near set index. In the final step, a proper force is calculated and each point is shifted to according to that force. In calculation of the Euclidean distances the sum of the values was implemented by a parallel reduction algorithm and the near sort by even-odd sorting network.

Another recent technique showed CUDA-based fast multidimensional scaling (CFMDS) approximate solution [24]. It implements two approaches: If the data can fit the memory of the GPU entirely, a classical algorithm is executed. Otherwise, the input data are divided into smaller portions that fit the global memory. The required memory is computed at the beginning and the program selects appropriate version of the MDS.

For the partial MDS the sampling of the input data was done in two ways: randomly and by MaxMin approach. During the MaxMin approach, data points were chosen one at a time and each new point maximized the minimal distance to any of the previously sampled points.

5 Discussion

To better analyze the performance issues for each algorithm (HiT-MDS, VPD-MDS, Glimmer, and CFMDS) we present the results obtained on the same GPU hardware for the same data sets.

Due to a limited space, we present only the results for fixed set of parameters. The results of each presented algorithm were measured for three data sets of sizes 0.2m, 0.5m, and 1m (m is a million) extracted from CoPHIR data set [27] which consists of 106-million MPEG-7 global visual descriptors. The descriptors are scalable color, color structure, color layout, edge histogram, and homogeneous texture. The vectors in each data set are constructed by combining the first three descriptors, which result in vectors of size 208-dimension.

Our experiments were conducted on a PC with an Intel Core i3-4010U CPU clocked at 1.7 GHz, which have 4 physical cores and 4 GB of RAM. The desktop PC is equipped with NVIDIA GeForce GT 740M (Kepler architecture [28]), which have 2 SMPs comprising 192 cores each (384 cores total) and 4 GB of global memory. The host used Windows 8.1 as operating system and CUDA 5.1 framework for the GPGPU computations.

The experiments were timed using the real-time clock of the operating system. Each experiment was conducted 10× and the arithmetic average of the measured values is presented as the result. All measured values were within 1% deviation from their respective averages. The comparison is mainly based on the speedup of the algorithms, since the accuracy of the results were presented in the related papers.

Due to a limited hardware availability in our experiments for VPD-MDS method data were not distributed among several GPUs. Instead, the data were loaded to global memory of the GPU portion by portion which significantly affected the final results. The portions of the work could have been processed on several GPUs in some cases to increase the overall speed; however, the main objective of this survey is to compare the methods relatively.

Table 1 shows the execution time (seconds) of each presented algorithm on the GPU using different thread block configurations. In case of HiT-MDS algorithm, the Floyd-Warshall algorithm and the degree based distance manipulation points (step 3 and 4) are not included in the final results. Furthermore, we include only the MaxMin results of the CFMDS implementation, since it has proven to be faster than the random method.

We have run experiments for different block sizes (2, 4, 8, 16, 32 warps – i.e., 64 – 1,024

Table 1 Performance comparison of presented GPU implementations.

Warps	Glimmer			CFMDS			HiT-MDS			VPD-MDS		
	0.2m	**0.5m**	**1m**	**0.2m**	**0.5m**	**1m**	**0.2m**	**0.5m**	**1m**	**0.2m**	**0.5m**	**1m**
2	1032	1615	2375	1051	1465	2817	1118	1652	3005	1362	1589	3206
4	512	742	1297	564	680	1422	524	755	1503	604	795	1597
8	410	602	1231	422	620	1404	416	699	1405	565	759	1540
16	417	594	1220	425	619	1394	421	711	1400	557	784	1563
32	**416**	**597**	**1211**	**415**	**620**	**1390**	**426**	**694**	**1401**	**557**	**754**	**1541**

threads) to analyze the effect of the GPU configuration on the performance of the algorithms. On the Kepler architecture, at least 4 warps must be used in order to achieve full occupancy of the cores, no matter which algorithm was tested. Due to a limited space, we analyze only the results when 32 warps (1, 024 threads) were used, since these results are the best (or very close to the best) for every algorithm.

The results indicate that the best performance was achieved with the Glimmer algorithm. It can be explained by the fact that stochastic approach is very well suited for the GPU architectures, because each point of the data set references only a small set of the other points during the main computation. Furthermore, the Glimmer algorithm spends the most time performing the relaxation procedure, which also performs well on GPUs. In terms of execution time, CFMDS algorithm is quite competitive with Glimmer algorithm, which can be explained by the multilevel origin of the two methods. CFMDS gives a little bit slower results, since the classical MDS algorithm is performed for each level of CFMDS and it is less suited for GPUs than the stochastic approach. Finally, the HiT-MDS solution outperformed VPD-MDS method because of the shared memory usage for partial-sum calculation.

6 Conclusion

Being a most popular method of dimensionality reduction, MDS algorithm is a main research topic in different science fields. Due to the rapid increase of the number of processed data, utilization of massively parallel hardware such as GPUs becomes a necessity. We have analyzed the performance issues of existing MDS algorithms on GPU architecture and compared their performance.

Comparison of existing works reveals that GPU solutions showed a promising potential for future scaling. Meanwhile, the differences in execution times of presented implementations are not very significant due to the similarity of methods. We believe that the effectiveness of each described method strongly depends on the used data set and the application, which means that the provided results could be roughly different for other inputs. However, comparison shows that a multi-level approaches show better results for large scale data.

The nature of dimensionality reduction algorithms allows adaptation of these methods for heterogeneous system consisting of multiple GPUs and multi-core CPUs which can significantly improve the performance even further. Based on the performance results obtained for the Glimmer algorithm, we are planing to use this method to visualize Maya hieroglyphs on a multi-GPU system.

Acknowledgements. This paper was supported by the Swiss National Foundation (SNF) under the project MAAYA (grant number 144238) and by Czech Science Foundation (GAČR) project no. P103-14-14292P.

References

1 C. J. C. Burges. *Dimension Reduction: A Guided Tour*. Foundations and Trends in Machine Learning 2, 2010

2 L. J. P. van der Maaten and G. E. Hinton. *Visualizing High-Dimensional Data Using t-SNE*. Journal of Machine Learning Research, 9(Nov), pp. 2579–2605, 2008.

3 S. Roweis and L. Saul. *Nonlinear dimensionality reduction by locally linear embedding*. Science, 290(5500):2323–2326, 2000.

4 D. Engel, L. Huttenberger, and B. Hamann. *A Survey of Dimension Reduction Methods for High-dimensional Data Analysis and Visualization*. Proceedings of IRTG 1131 Workshop, Dagstuhl, Germany, pp. 135–149, 2012.

5 F. Plastria, S. de Bruyne, and E. Carrizosa. *Dimensionality Reduction for Classification*. In ADMA (Lecture Notes in Computer Science), Springer, 411–418, 2008.

6 B. M. Sarwar, G. Karypis, J.A. Konstan, and J.T. Riedl. *Application of Dimensionality Reduction in Recommender System – A Case Study*. In ACM WEBKDD Workshop, 2000.

7 P. Imkeller, S. Namachchivaya, N. Perkowski, and H. C. Yeong. *Dimensional Reduction in Nonlinear Filtering: a Homogenization Approach*. Timo Sepplinen (Ed.), Vol. 23, The Annals of Applied Probability, 2013.

8 M. Ishteva, L. de Lathauwer, P. A. Absil, and S. van Huffel. *Dimensionality reduction for higher-order tensors: algorithms and applications*. International Journal of Pure and Applied Mathematics 42, 337–343, 2008.

9 I.F.L. Journaux and P. Gouton. *Multispectral Satellite Images Processing through Dimensionality Reduction*. In Signal Processing for Image Enhancement and Multimedia Processing, 2008.

10 I. K. Fodor. *A survey of dimension reduction techniques*. Technical Report UCRL-ID-148494, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-560, Livermore, CA 94551, USA, 2002.

11 L. J. P. van der Maaten, E. O. Postma, and H. J. van den Herik. *Dimensionality Reduction: A Comparative Review*. Technical Report TiCC-TR, Tilburg University, 2009.

12 N. Varghese, V. Verghese, P. Gayathri, and N. Jaisankar. *A survey of dimensionality reduction and classification methods*. International Journal of Computer Science and Engineering Survey (IJCSES) 3, pp. 45–54, 2012.

13 A. Sarveniazi. *An actual survey of dimensionality reduction*. American Journal of Computational Mathematics 4, pp. 55–72, 2014.

14 W. S. Torgerson. *Multidimensional scaling: I. Theory and method*. Psychometrika 17, pp. 401–419, 1952

15 NVIDIA Corporation. *NVIDIA CUDA: Compute Unified Device Architecture. Reference Manual*. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, 2008.

16 A. Morrison, G. Ross, and M. Chalmers. *Fast multidimensional scaling through sampling, springs and interpolation*. Information Visualization 2, pp. 68–77, 2003.

17 M. Chalmers. *A Linear Iteration Time Layout Algorithm for Visualising High-dimensional Data*. In Proceedings of the 7th Conference on Visualization. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

18 T. Yang, J. Liu, L. McMillan, and W. Wang. *A fast approximation to multidimensional scaling*. In Proceedings of the ECCV Workshop on Computation Intensive Methods for Computer Vision, pp. 354–359, 2006.

19 T. Fester, F. Schreiber, and M. Strickert. *CUDA-based Multi-core Implementation of MDS-based Bioinformatics Algorithms*. In GCB (LNI), Ivo Grosse, Steffen Neumann, Stefan Posch, Falk Schreiber, and Peter F. Stadler (Eds.), Vol. 157, 67–79, 2009

20 M. Strickert, S. Teichmann, N. Sreenivasulu, and U. Seiffert. *High-Throughput Multidimensional Scaling (HiT-MDS) for cDNA-Array Expression Data*. In ICANN (1) (Lecture Notes in Computer Science), Wlodzislaw Duch, Janusz Kacprzyk, Erkki Oja, and Slawomir Zadrozny (Eds.), Vol. 3696. Springer, pp. 625–633, 2005.

21 M. Harris, S. Sengupta, and J.D. Owens. *Parallel Prefix Sum (Scan) with CUDA*. In GPU Gems 3, 2007.

22 P. Harish and P. J. Narayanan. *Accelerating Large Graph Algorithms on the GPU Using CUDA*. In Proceedings of the 14th International Conference on High Performance Computing. Springer-Verlag, Berlin, Heidelberg, pp. 197–208, 2007.

23 S. Ingram, T. Munzner, and M. Olano. *Glimmer: Multilevel MDS on the GPU*. IEEE Trans. Vis. Comput. Graph. 15, pp. 249–261, 2009.

24 S. Park, S. Y. Shin, and K. B. Hwang. *CFMDS: CUDA-based fast multidimensional scaling for genome-scale data*. BMC Bioinformatics 13, S-17, S23, 2012.

25 P. Pawliczek, W. Dzwinel, and D. A. Yuen. *Visual exploration of data by using multidimensional scaling on multicore CPU, GPU, and MPI cluster*. 2014.

26 W. Dzwinel and J. Blasiak. *Method of particles in visual clustering of multi-dimensional and large data sets*. Future Generation Comp. Syst. 15, pp. 365–379, 1999.

27 P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. *CoPhIR: a test collection for content-based image retrieval*. CoRR, 2009.

28 NVIDIA. *Kepler GPU Architecture*. `http://www.nvidia.com/object/nvidia-kepler.html`.

29 D. Keim, G. Andrienko, J. D. Fekete, C. Görg, J. Kohlhammer, and G. Melançon. *Visual Analytics: Definition, Process, and Challenges*. In Andreas Kerren, John T. Stasko, Jean-Daniel Fekete, and Chris North (Eds.), Information Visualization – Human-Centered Issues and Perspectives, pp. 154–175, Lecture Notes in Computer Science 4950, Springer Berlin Heidelberg. (2008).