Dimensionality Reduction using GPU-accelerated Gradient Descent

Alexey Borisov
Department of Geoinformatics and Information Security
Samara National Research University
Samara, Russia
borisovalexey1996@gmail.com

Evgeny Myasnikov
Department of Geoinformatics and Information Security
Samara National Research University
Samara, Russia
mevg@geosamara.ru

***Abstract*—In this paper, we discuss several possible implementations of GPU-targeted gradient descent algorithm for dimensionality reduction. Four realizations of the gradient descent algorithm are created using HIP, a new framework for GPGPU programming. We get six times performance improvement over a multithreaded CPU version using AMD Radeon RX Vega 56.**

Keywords*—*dimensionality reduction, gradient descent, GPU, GPGPU, HIP

I. Introduction

In the present days, data mining becomes more and more popular research area. Data mining techniques process enormous amounts of data in order to detect significant dependencies in data. In many cases, such data can be interpreted as points in some multidimensional space.

Processing multidimensional data causes problems. The first problem is that multidimensional data requires more storage capacity. The second problem is that the time needed to process these data grows fast with the number of dimensions. But in many cases, multidimensional data contains substantial redundancy in terms of information. Therefore, one can map the data into the space of lower dimensionality without a major information loss. This process can be referred to as dimensionality reduction.

Dimensionality reduction techniques are often used in different problems of image analysis (see [1-4], for example).

There is a variety of dimensionality reduction methods. The most famous methods are Principal component analysis (PCA) and Independent Component Analysis (ICA), which are linear methods. The major disadvantage of linear methods is that they can find only linear dependencies within data. The nonlinear dimensionality reduction methods are Curvilinear Component Analysis (CCA) [5], Curvilinear Distance Analysis (CDA) [6], nonlinear mapping (Sammon's mapping [7]), etc. It was shown [4] that the latter technique can deliver better results compared to some other dimensionality reduction methods, for example, in hyperspectral image analysis.

While there are GPU implementations for linear dimensionality reduction techniques [8, 9], the problem of the long running time is more related to nonlinear techniques. In this paper, we study the performance of the Sammon's mapping dimensionality reduction method based on the gradient descent implemented for GPUs.

II. Dimensionality Reduction using Gradient Descent

Let N denote the number of points, n denote the dimensionality of a high-dimensional base space and m denote the dimensionality of a low-dimensional target space. Our goal is to map points from the n-dimensional space into the m-dimensional space, minimally affecting inter-point distances. This task can be formulated as an optimization problem.

At first, we should define a cost function. In the case of Sammon's mapping the cast function is as follows:

$$\varepsilon = \frac{1}{\sum_{i,j>i}^{N} d_{ij}^{n}} \sum_{i,j>i}^{N} \frac{\left(d_{ij}^{n} - d_{ij}^{m}\right)^2}{d_{ij}^{n}}, \tag{1}$$

where d^n_{ij} is the distance between points i and j in the base space, d^m_{ij} is the distance between points i and j in the target space. In this case, the Euclidean distance was used.

Gradient descent is a common iterative optimization algorithm. Its common definition is:

$$\mathbf{x}(t+1) = \mathbf{x}(t) - \alpha \nabla f, \tag{2}$$

where the $\mathbf{x} = \{x_0, \ldots, x_k\}$ denotes the parameters vector, t denotes the iteration number, α denotes the speed constant and ∇f denotes the gradient of the cost function $f(\mathbf{x})$ in the current point.

If we take the gradient of the Sammon's error with the coordinates of all the points in the lower-dimensional space as a vector of parameters, we will get the following equation, describing the Sammon's mapping procedure:

$$\mathbf{y}_i(t+1) = \mathbf{y}_i(t) + \mu \sum_{j\neq i}^{N} \frac{d_{ij}^{n} - d_{ij}^{m}}{d_{ij}^{n} d_{ij}^{m}} \left(\mathbf{y}_i(t) - \mathbf{y}_j(t)\right), \tag{3}$$

$$\mu = \frac{2\alpha}{\sum_{i,j>i}^{N} d_{ij}^{n}}, \tag{4}$$

where $\mathbf{y}_i = \{y_0^i, \ldots, y_{m-1}^i\}$ denotes the coordinates vector of the i-th point , t denotes the iteration number, d^n_{ij} denotes the distance between points i and j in the base space, d^m_{ij} denotes the distance between points i and j in the target space, and α denotes the descent speed constant.

The computational complexity of this algorithm is $O(N^2(n+m))$. But it is worth noting that all points are updated independently during an iteration and therefore can be processed in parallel. This idea opens the possibility to use the graphics processing units (GPU).

III. HIP

HIP is a new framework for GPU programming. It is developed by the GPUOpen initiative [10]. The HIP is actually an abstraction layer based on C++ macros, which exposes a programming interface and programming language similar to NVIDIA CUDA. Programs written in CUDA can be converted to HIP using a special tool. Depending on a platform used, the HIP code can be compiled using HCC for AMD devices or NVCC for NVIDIA devices.

The terminology and programming model of HIP are inherited from CUDA. The interaction with a GPU is performed using API calls that perform data copy, synchronization, and execution launching. Each command is issued to some command stream. Commands in different command streams are executed in parallel if possible.

A function that was dedicated to perform on a GPU is called a kernel. Each kernel is executed using many GPU threads. GPU threads are grouped into thread blocks, whose size and count are specified on a kernel call. Available memory regions are global memory and shared memory. The global memory is an equivalent of the RAM. The shared memory has very high throughput but also a very small volume, so it was not used in the current work.

IV. Implementation of Gradient Descent

Equation 3 shows that points are updated independently from each other at one iteration. Therefore, several threads can be used to process points in parallel. The more threads used, the better performance may be obtained in theory. Since GPUs can process thousands of threads simultaneously, they may be used to accelerate the processing.

To implement the iterative gradient descent, 3 buffers were used:

1. the buffer to store the coordinates of points in the original space;

2. the buffer to store the coordinates of points in the target space obtained at the previous iteration;

3. the buffer to store the coordinates of points in the target space calculated at the current iteration.

We discuss two approaches for the gradient descent implementation. The first approach (denoted below as a *standard approach*) is to process data according to (3), i.e. for each point all distances to other point was computed, and then the point was updated. The second approach (denoted below as a *modified approach*) uses a fact that $d^n_{ij} = d^n_{ji}$. Therefore we may compute each distance only once and then perform an atomic update of points *i* and *j*. This approach reduces the number of inner cycle iterations from $N(N\text{-}1)$ to $N(N\text{-}1)/2$, but also adds an overhead caused by the atomic operations. In this case, the linear indexing of the virtual distances matrix was used to exclude the conditional operators in the program code, which should be avoided in GPU programs.

We also discuss two possible memory layouts of data points. The points set can be stored as a 2D data matrix. The question is how to place the coordinates of one point. The first layout (denoted below as a *flat layout*) places the point coordinates in rows. This layout allows us to compute the distances between points faster since in this case we are able to use the vector load and store instructions. The second layout (denoted below as a *transposed layout*) places the coordinates of points in columns of the data matrix. This allows the neighboring threads to read the neighboring memory cells when accessing the data. This facilitates memory caching and allows the memory controller to merge many small memory access requests to one (memory coalescing).

V. Experimental Results

Two implementation approaches and two memory layouts give 4 implementations:

1. a standard approach with a flat layout;
2. a modified approach with a flat layout;
3. a standard approach with a transposed layout;
4. a modified approach with a transposed layout;

Proposed implementations were tested using a dataset with random points with coordinates from the range [0, 255]. We measured the average execution time for various thread block sizes and datasets sizes. The time of the data transfer from RAM to GPU was not accounted, only the kernel time was measured.

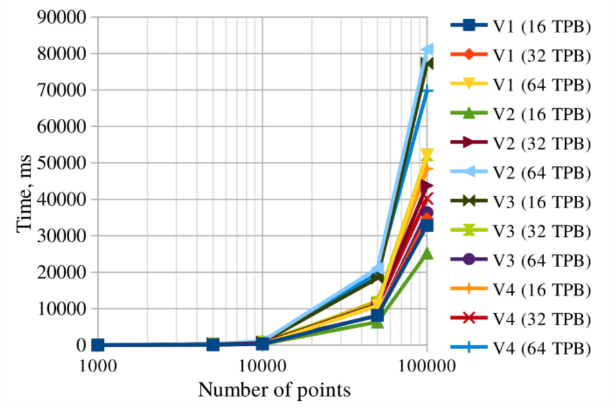

Fig. 1. Time of One Iteration. *TBP means Threads Per Block.

The experiment parameters are:

- *Number of points, N:* 1’000, 2’000, 10’000, 50’000, 10000;
- *Base dimensionality, n:* 200;
- *Target dimensionality ,m:* 3*;*
- *Learning constant,* µ*:* 0.5*;*
- *Thread blocks count:* 1024*;*
- *Thread block sizes:* 16, 32,64;
- *The number of iterations:* 5.

We compared the experimental results with the results of CPU implementation. This implementation was created using an OpenMP as a parallel framework and was compiled with the AVX cycle vectorization allowed. We measured execution time for both single-threaded and 16-threaded versions of this realization.

The experiments were run on the following configuration:

- *CPU:* AMD Ryzen 7 3700X;
- *GPU:* AMD Radeon RX Vega 56;
- *RAM:* 8 GB DDR4-3200;
- *OS:* Ubuntu 18.04 LTS.

The experimental results are shown in Fig. 1 and Fig.2. The best results along with used parameters are shown in Table I. The comparison of CPU and GPU results can be found in Table II. The comparison of CPU and GPU final error values is shown in Table III.

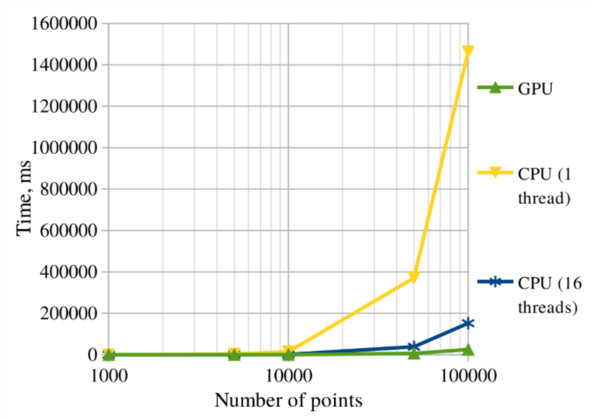

Fig. 2. Time of One Iteration for CPU and GPU.

Let us analyze the results. It is clear that performance improvement grows along with the task size. The most effective implementation is implementation 2 (a modified approach with a flat memory layout). In our case, the ability to compute distances faster was more important than advanced caching and memory coalescing. The data given by the profiler shows that the main performance limiting factor is the HBM memory performance.

TABLE I. THE BEST RESULTS FOR GPU IMPLEMENTATION: RUNNING TIME PER ONE ITERATION

Number of Points	***GPU Time, ms***	***Implementation***	***Thread Block Size***
1 000	7.61	2	16
5 000	88.36	1	32
10 000	279.33	1	64
50 000	6 342.90	2	16
100 000	25 245.20	2	16

TABLE II. THE COMPARISON OF CPU AND GPU: RUNNING TIME PER ONE ITERATION

Number of Points	***CPU Time (1 thread), ms***	***CPU Time (16 threads), ms***	***GPU Time, ms***
1 000	142.46	16.33	7.61
5 000	3 556.06	398.08	88.36
1 0000	14 251.50	1 497.72	279.33
50 000	370 820.00	38 433.20	6 342.90
100 000	1 462 767.20	152 011.00	25 245.20

TABLE III. THE COMPARISON OF CPU AND GPU: DATA MAPPING ERROR

Number of points	***Initial Error***	***CPU Final error***	***GPU Final Error***
100 000	0.785323	0.110451	0.110452

The HBM (High Bandwidth Memory) features the very wide data bus along with relatively low clocks. Since in our case threads access memory cells located relatively far from each other, the caching is ineffective, and the memory requests often cause delays in execution. The use of atomic operations adds an additional load to the memory system. This partially explains why the thread block size of 16 threads was the best. Since threads of one thread block are executed in groups of 64 threads on the AMD GPU, the thread block of only 16 threads causes only 25% occupancy but also reduces the number of simultaneous memory requests. The results from GPUs with different memory types, such as GDDR5/6, probably will be different.

VI. CONCLUSION

In this paper, we have discussed different approaches to implement the dimensionality reduction method based on gradient descent for GPUs. Several experiments were taken using an AMD Radeon Vega 56 GPU. The experimental results show approximately 6 times performance improvement in comparison to the multithreaded version executed on the 8-core CPU, and almost 60 times improvement over single-threaded implementation.

Future work will be devoted to the comparison of the proposed implementations using NVIDIA and AMD GPUs.

ACKNOWLEDGMENT

The work was partly funded by RFBR according to the research project 18-07-01312-a in parts of «2. Method» - «3. Experiments» and by the Russian Federation Ministry of Science and Higher Education within a state contract with the «Crystallography and Photonics» Research Center of the RAS in parts «1. Introduction» and «4. Conclusion».

REFERENCES

[1] E.A. Dmitriev and V.V. Myasnikov, “Comparative study of description algorithms for complex-valued gradient fields of digital images using linear dimensionality reduction methods,” Computer Optics, vol. 42, no. 5, pp. 822-828, 2018. DOI: 10.18287/2412-6179-2018-42-5-822-828.

[2] M.V. Gashnikov, “Optimization of the multidimensional signal interpolator in a lower dimensional space,” Computer Optics, vol. 43, no. 4, pp. 653-660, 2019. DOI: 10.18287/2412-6179-2019-43-4-653-660.

[3] E.V. Myasnikov, “The study of dimensionality reduction methods in the task of browsing of digital image collections,” Computer Optics, vol. 32, no. 3, pp. 296-301, 2008.

[4] E. Myasnikov, “Evaluation of nonlinear dimensionality reduction techniques for classification of hyperspectral images,” CEUR Workshop Proceedings, vol. 2268, pp. 147-154, 2018.

[5] P. Demartines and J. Herault, “Curvilinear Component Analysis: a SelfOrganizing Neural Network for Nonlinear Mapping of Data Sets,” IEEE Trans. Neural Networks, vol. 7, pp. 148-154, 1997.

[6] A. Lee, A. Lendasse, N. Donckers and M. Verleysen. “Robust Nonlinear Projection Method,” Proc. ESANN, Bruges: D-Factopublic, pp. 13-20, 2000.

[7] J.W. Sammon Jr, “A Nonlinear Mapping for Data Structure Analysis,” IEEE Trans. on Computers, vol. C-18, no. 5, pp. 401-409, 1969.

[8] M. Andrecut, “Parallel GPU Implementation of Iterative PCA Algorithms,” J. Comp. Biol., vol. 16, pp. 1593-1599, 2009.

[9] J. Yanshan, W. Zeng, N. Wang, T. Ren, Y. Shi, J. Yin and Q. Xu, “GPU-based Parallel Group ICA for functional magnetic resonance data,” Computer methods and programs in biomedicine, vol. 119, no. 1, pp 9-16, 2015.

[10] Y. Sun, “Evaluating Performance Tradeoffs on the Radeon Open Compute Platform,” ISPASS, pp. 209-2018, 2018.