Seeking Performance In Nonlinear Dimensionality Reduction COMP4240 Course Project

Jukka Jylänki

June 2, 2008

Contents

1 Introduction **2**

2 Algorithms **2**
2.1 Sammon's Nonlinear Mapping 2
2.1.1 Newton's Method 3
2.1.2 NLM In Practice 4
2.2 Isomap 5
2.3 Landmark Isomap 5
2.4 Isotop 5
2.4.1 Vector Quantization Using Simple Competitive Learning . 6
2.4.2 Kohonen's Self-Organizing Maps 7
2.4.3 Comparisons 8

3 Implementation **8**
3.1 Early Matlab Tests 9
3.2 NLM Implementation 10
3.2.1 High-Performance NLM on GPU 11
3.3 Isomap Implementation 12

4 Benchmarks **13**
4.1 Scalability Concerns 14

5 Conclusion **15**
5.1 Future Work 15

6 References **17**

1 Introduction

Dimensionality reduction is a common name for an operation that processes a data set to reduce its size in dimension while trying to preserve most of the interesting information. **Nonlinear dimensionality reduction** (NLDR) techniques produce low-dimensional embeddings that don't lose the important nonlinear dependencies in the data. In this project, some common NLDR techniques and their properties are examined, with the aim of selecting a suitable method for parallelized computation and further implementation. Hence, the focus is put on practical issues with performance considerations in mind. C++ source code snippets of implementations of the selected algorithms, along with a high-performing parallelized version of a chosen method for computation on modern graphics hardware (GPU) are included.

In the next chapter, the algorithms that were reviewed are presented along with some necessary mathematics needed to understand them. The implementations are discussed in the third chapter and benchmarks are presented in the fourth chapter. The final chapter describes conclusions and proposes future work for a follow-up project.

2 Algorithms

The algorithms that were chosen for investigation are **Sammon's Nonlinear Mapping** (NLM) [nld07, Chapter 4.2.3.], **Isomap** [Tea00], **Landmark Isomap** [dST03] and **Isotop** [LV02]. Two of these, NLM and Isomap, were implemented as a generic use C++ library. Finally, a highly optimized parallelized version of NLM was written as a proof-of-concept application to demonstrate the power of GPUs in stream processing computations.

2.1 Sammon's Nonlinear Mapping

Iterative versions of MDS operate on a so-called **cost** or **stress** function, that evaluates how good an embedding is. When applying Sammon's NLM on a set of N data points, we work on a NxN matrix that contains the pairwise distances of the data points. A straightforward cost function is the **raw stress** function, which is given by

$$\text{Cost} = \sum_{i,j} (||x_i - x_j|| - ||y_i - y_j||)^2$$

where y_i are the data in the high-dimensional space and x_j are the points mapped to the low-dimensional space. The idea here is to preserve the pairwise distances between all data points as well as possible.

Another commonly used cost function is the **Sammon cost** function, which is defined by

$$\text{Cost} = \sum_{i,j} \frac{(||x_i - x_j|| - ||y_i - y_j||)^2}{||x_i - x_j||}$$

The Sammon cost function operates on relative distance differences instead of absolute distances, which favors preserving small distances.

The dimensionality reduction process is recast as a nonlinear optimization problem and a variety of algorithms can be used to minimize the cost function. Techniques include eigendecomposition, the conjugate gradient method or a pseudo-Newton method for nonlinear equations [vdM07]. The method of steepest descent is used by [Kru64] and kernel methods are utilized by [Web02].

For implementation in this project, the pseudo-Newton optimization method was selected for minimizing the cost function in Sammon's NLM. This method was chosen as the high-performance candidate for GPU implementation as well, due to the simplistic structure of the algorithm. Because the inner loop of the pseudo-Newton method does not contain either random-access data lookups or any conditionals, it is ideal from GPU performance viewpoint.

2.1.1 Newton's Method

This section is adapted from [nld07, Appendix C].

Newton's method is an iterative technique to find a zero of a C_∞ function. We approximate our function f by a first-order Taylor expansion, $f(x+\epsilon) \approx f(x) + \epsilon f'(x) + \mathcal{O}(n^2)$. Then, to convert this to an update rule we denote $x_{new} := x + \epsilon$, $x_{old} := x$ and set $f(x_{new}) = 0$ to find an approximate location of the zero. With a little manipulation we get

$$x_{new} = x_{old} - \frac{f(x_{old})}{f'(x_{old})}.$$

To find the minimum of a function, we're looking for the zero of the function's first derivative. Thus our update rule gets the form

$$x \leftarrow x - \frac{f'(x)}{f''(x)}.$$

There is one problem though. At the function's maxima the derivative is also zero. Fortunately we can distinguish between these, since at maxima the second derivative is negative and at minima the second derivative is positive.We can make the iteration diverge from all maxima by taking absolute values of the second derivative. This yields

$$x \leftarrow x - \frac{f'(x)}{|f''(x)|}.$$

For this application, we are interested in the multidimensional case. If $f : \mathbb{R}^n \to \mathbb{R}$, the same update rule takes the form

$$x \leftarrow x - \mathbb{H}^{-1} \left[\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_n} \right]^T$$

where $\mathbb{H}$ denotes the nxn Hessian matrix. Calculating $\mathbb{H}$ is not feasible as its dimension is far too big to be practical. The solution is to make a simplifying assumption that the Hessian is zero outside the diagonal, i.e.

INITIALIZE $x_1, \ldots, x_n$ RANDOMLY.
WHILE(NOT CONVERGED)
  FOR($i = 1, \ldots, N$)
    FOR($k = 1, \ldots, p$)
      UPDATE $x_{i,k}$ USING THE RULE $x_{i,k} \leftarrow x_{i,k} - \alpha \frac{\frac{\partial E}{\partial x_{i,k}}}{|\frac{\partial^2 E}{\partial x_{i,k}^2}|}$.

Figure 1: The NLM algorithm.

$\mathbb{H} = diag\left(\frac{\partial^2 f(x)}{\partial x_1^2}, \ldots, \frac{\partial^2 f(x)}{\partial x_n^2}\right)$. This brings a considerable amount of instability into the optimization and to account for that, we introduce a step factor α ($0 < \alpha < 1$) to stabilize the search. The final update rule is then

$$x_k \leftarrow x_k - \alpha \frac{\frac{\partial f(x)}{\partial x_k}}{|\frac{\partial^2 f(x)}{\partial x_k^2}|}.$$

2.1.2 NLM In Practice

To perform pseudo-Newton optimization in NLM, we need to calculate the first and second derivatives of Sammon's cost function. Denote by $x_1, \ldots, x_N$ the output points in the low-dimensional space with dimension p and let $y_1, \ldots, y_N$ be the input points in the high-dimensional space. Also denote by $x_{i,k}$ the k-th component of the vector x_i. Then let

$$E = \sum_{i<j} \frac{(d(x_i, x_j) - d(y_i, y_j))^2}{d(y_i, y_j)}$$

Note that to reduce extra computational ballast, calculating a normalizing factor $\frac{1}{\sum_{i<j} d(y_i, y_j)}$ as was done in [nld07, Chapter 4.2.3.] is ignored. By careful calculation, and ignoring constant multiplicative factors, we get

$$\frac{\partial E}{\partial x_{i,k}} = \sum_{j=1, j\neq i}^{N} \frac{d(y_i, y_j) - d(x_i, x_j)}{d(y_i, y_j)}(x_{j,k} - x_{i,k})$$

and

$$\frac{\partial^2 E}{\partial x_{i,k}^2} = \sum_{j=1, j\neq i}^{N} \left(\frac{(x_{i,k} - x_{j,k})^2}{(d^3(x_i, x_j)} - \frac{d(y_i, y_j) - d(x_i, x_j)}{d(y_i, y_j) d(x_i, x_j)} \right).$$

These formulas are deliberately missing the multiplicative constants in front of them, because for our purposes they are insignificant as they are melded into the step factor α.

Applying NLM then maps quite straightforwardly to a practical implementation. Note that there is a $\mathcal{O}(N)$ loop hidden from the algorithm in Figure 1, so the total complexity for running a single iteration is $\mathcal{O}(N^2 p)$. This linear pass in calculating the partial derivatives proposes some challenges for the GPU implementation, but we shall see how to handle it later on.

2.2 Isomap

Isomap [Tea00] is a clever extension to MDS that tries to better capture the local topology of the data. When MDS is applied to a data set that lies on an extremely curved manifold, it can fail to extract the local structure of the data and instead groups up points that are close to each other (according to the Euclidean metric) whereas they in reality are far away from each other along the manifold. Isomap tries to recapture the local properties by building the pairwise distance matrix differently. First it generates a connectivity graph of the data points where edges represent adjacencies between points along the manifold. These connections are created using the k-nearest neighbors rule, or an ϵ-distance limit. The second step of the algorithm is to generate the pairwise distance matrix according to the found adjacencies in the graph, and to fill in the blanks between non-pairs in the graph representation, an all-pairs shortest paths algorithm is run. The last step of the algorithm is to apply a traditional MDS algorithm, but now by using the distance matrix that was generated. This gives a true nonlinear embedding that has better local features than MDS by itself.

The implementation of Isomap is discussed in Chapter 3.3. This algorithm has the obstacle that running the all-pairs shortest paths on a GPU is a technical step, but it certainly can be done [?]. Isomap optimizations were run with the graph building stage operated on the CPU and the NLM phase was computed on the GPU.

2.3 Landmark Isomap

A particular consern of all dimensionality reduction techniques is in their scalability. Running an all-pairs shortest paths algorithm in Isomap can be a noticeable issue in practice even though the adjacency graph is desirably sparse. Johnson's algorithm runs in $\mathcal{O}(V^2 \log V + VE)$ [cor01, Chapter 25] and if the MDS mapping is computed using the Eigenvalue-method, the complexity is $\mathcal{O}(N^3)$ [dST03]. With possibly more than tens of thousands of data points, these algorithms start to get infeasible in practice.

In *Landmark Isomap*, a small fraction of the points are designated as so-called landmarks, and the distances are calculated from each data point to those points only. This drastically reduces the number of operations that need to be done in the two algorithms above. The non-landmark points are positioned by examining their distances from the landmarks [dST03].

Landmark Isomap was not implemented in the library, but writing one would only require decimating the inputs to the original Isomap algorithm, and implementing a triangulation method to position the rest of the data points.

2.4 Isotop

Isotop performs nonlinear dimensionality reduction in the same vein than Isomap does, by seeking to produce a projection that preserves pairwise distances in

local neighborhoods rather than on a global scale, as classic metric multidimensional scaling does. Isotop also tries to solve the performance scaling problem by decimating the number of points to process using **Vector Quantization** (VQ). The original paper mentions three main stages in the algorithm [LV02], but I would like to identify four. The first phase is to perform VQ on the data points to reduce it to a smaller fixed size set of so-called **prototypes** that characterize the distribution as well as possible. VQ is a common problem in other areas, like color quantization, and there exists several different algorithms to perform the task. For more depth, see [HB98]. The algorithm of choice in the original paper, **Simple Competitive Learning** (SCL) is presented in the next chapter. The second stage of the Isotop method is the same as in Isomap and it consists of building a neighborhood graph between the prototypes using k-nearest neighbors or an ϵ-rule. This stage includes running the all-pairs shortest paths algorithm to determine the distances of the data points along the estimated manifold. In the third phase the actual projection of the prototypes is constructed. This projection could be done for example by calculating the classic analytical solution for metric MDS or by Sammon's Nonlinear Mapping, but in the Isotop method, an approach based on **Kohonen's Self-Organizing Maps** (SOM) is chosen. Finally, in the fourth stage, the rest of the points of the input data are projected to the low-dimensional space by estimating their positions relative to the already projected prototypes.

2.4.1 Vector Quantization Using Simple Competitive Learning

In Vector Quantization, we are given N input vectors $P = \{p_1, \ldots, p_N\}$ and the task is to reduce them to $n < N$ prototypes $Q = \{q_1, \ldots, q_n\}$ that "best" represent the input data. Note that in general, $Q \nsubseteq P$. We define a **distortion** function to formalize what is meant by best representation, as follows. For a point p_i, define the closest quantizer q_j as one having the property that $\forall q \in Q. ||q_j - p_i||^2 \leq ||q - p_i||^2$. For each prototype q_i, we associate C_i to be the set of input vectors that have q_i as their closest quantizer. The collection $\{C_i\}_i$ forms a partition of the set P and we define our distortion function δ as

$$\delta = \frac{1}{N} \sum_{i=1}^{n} \sum_{x_j \in C_i} ||x_j - q_i||^2.$$

Now to perform the quantization itself, we employ the algorithm shown in Figure 2. For initialization, we start by picking initial values for $q_1, \ldots, q_n$. Taking n random elements of P works. Then, for each input vector p_i we find the nearest prototype q_w and reposition it using so-called 0-Kohonen update rule. This update is performed over several epochs, until we have achieved convergence. Particular care must be taken when constructing the **learning rate function** $\varepsilon(t)$, since too big learning rates will cause instability and a rate too small will make the convergence slow. As such, the function is usually constructed to satisfy two conditions, $\sum_{t=1}^{\infty} \varepsilon(t) = \infty$ and $\sum_{t=1}^{\infty} \varepsilon^2(t) < \infty$, which are called the Robbins-Monro conditions.

```
INITIALIZE q_1, ..., q_n.
FOR(t = 1, ..., ∞)
    FOR(i = 1, ..., N)
        FIND q_w THAT IS NEAREST TO p_i.
        UPDATE q_w USING THE RULE q_w ← q_w + ε(t)(p_i − q_w).
    RECALCULATE δ.
    IF δ HAS CONVERGED, STOP AND OUTPUT q_1, ..., q_n.
```

Figure 2: The SCL algorithm.

Intuitively, we can think of the data points p_i to "compete" over the influence of the prototype points q_j. Large clusters of nearby points will attract the prototype points more effectively than single isolated points and as a result the distribution of the prototype points will closely match the input data.

2.4.2 Kohonen's Self-Organizing Maps

The low-dimensional embedding in Isotop is performed using Kohonen's SOM algorithm. SOM works by performing the same competitive learning iteration scheme as does the SCL algorithm that was described in the previous chapter, but the major difference is in the update rule. In SOM, instead of pushing one prototype q_w towards an input point at a time, we find several neighboring prototypes of q_w and update all of them towards the input point. This has been shown to greatly reduce the iterations needed to achieve convergence [dBCLV03].

As a result of the previous stage, we have n prototypes $q_1, \ldots, q_n \in \mathbb{R}^P$ and a pairwise distance function $d(q_i, q_j)$ that gives an estimation of the prototype distances along the manifold. We now seek to embed these points to the low-dimensional space, that is, to come up with $m_1, \ldots, m_n \in \mathbb{R}^p, p < P$ that minimize the error in pairwise distances, i.e. $\sum_{i<j} |d(q_i, q_j) - d(m_i, m_j)|^2$ is as small as possible.

We define $\varepsilon_1(t)$ to be a learning rate function satisfying Robbins-Monro conditions as before, and $\varepsilon_2(t)$ an another learning rate function that is nonincreasing and tends to 0 as epochs go by. Also, define $\underline{G}(t)$ to be a p-dimensional standard Gaussian random variable (zero mean, unit norm), i.e., $\underline{G}(t) \sim N^p(0, 1)$. This random variable is used for sampling stimulus points to be used in the learning[1]. To formally define the neighborhoods of points as required by SOM, we use a neighborhood factor function $\nu(t, q_i, q_j)$ that yields values close to 1 if the two points are close to each other, and values near 0 when they are far. This function is defined in terms of the pairwise distance function $d(q_i, q_j)$ as follows:

$$\nu(t, q_i, q_j) = e^{-\frac{1}{2}(\frac{d(q_i,q_j)}{\varepsilon_2(t)})^2}.$$

[1] These can be seen to correspond to the points q_i in the SCL algorithm.

```
INITIALIZE m_1, ..., m_n AT RANDOM.
FOR(t = 1, ..., ∞)
    LET g_t ← G(t)
    FIND m_w THAT IS NEAREST TO g_t.
    UPDATE ALL m_i USING THE RULE m_i ← m_i + ε_1(t)ν(t, q_i, q_w)(g_t − m_i).
    RECALCULATE δ.
    IF δ HAS CONVERGED, STOP AND OUTPUT m_1, ..., m_n.
```

Figure 3: SOM for NLDR embedding.

Using these definitions, we proceed to compute the algorithm in Figure 3. It is worth to stop and examine why this works. After all, our choices of g_t are more or less arbitrary. A very important role is in the function ν, which determines how strongly the points m_i are moved. The g_t merely act as stimulants for the learning process. The role of $\varepsilon_2(t)$ here is to make the neighborhoods become smaller as epochs go by, eventually shifting the algorithm from global to local scale optimization.

2.4.3 Comparisons

Isotop is very close to Landmark Isomap in its characteristics. The VQ of the first phase corresponds to the selection of landmark points, the aim in both being to scale down the number of data points to process. They both estimate the manifold structure in high-dimensional space using graph distances rather than employing Euclidean distances, as MDS does. The choice of the prototype points in Isotop feels more justified than just randomly picking the landmark points, but at the same time, Isotop may suffer from losing fine-grained details in manifolds due to the averaging nature of the VQ process. Producing the embedding using SOM instead of Sammon's NLM does generate different results. See [nld07, Chapter 6.1.1] for plots of practical comparisons. The idea of employing SOM instead of MDS or Sammon's NLM to develop an embedding is interesting since it might be computable by stream processing kernels on a GPU better than NLM. One such implementation can be found in [LZ].

For the GPU implementation however, Isotop was not chosen. This was due to the simplicity of NLM over Isotop. Isotop involves two separate iteration stages which would have required implementations of their own. Moreover, Isotop has the same obstacle of requiring an all-pairs shortest paths implementation as Isomap.

3 Implementation

The early goal of the project was to measure the candidate algorithms to see how good embeddings they produce and how well they scale. Synthetic datasets were used to test the algorithms. **Maaten Matlab Toolbox for dimensionality**

reduction [vdM07] was used for early tests, to get an idea of the speed of the algorithms, and later on when the C++ implementations were ready, as a guide to compare and confirm proper functionality.

3.1 Early Matlab Tests

To compare NLM and Isomap feature-wise, the standard Swiss Roll dataset was used, because it clearly visualizes the difference in how these two techniques operate. In the following, the value of $k = 12$ was used for the nearest neighbor estimation parameter in Isomap. Results are shown below.

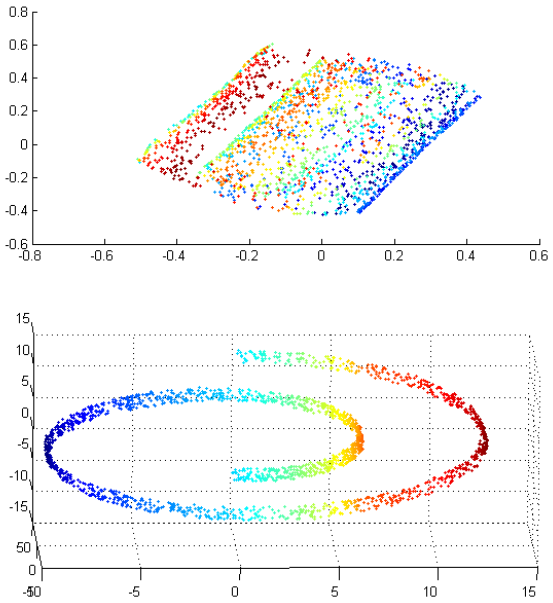


Figure 4: Bottom: The Swiss Roll data set in 3D space. Top: The 2D embedding produced by NLM.

Figure 4 shows how NLM is unable to reconstruct the proper structure since preserving all the global as well as local pairwise distances are considered important.

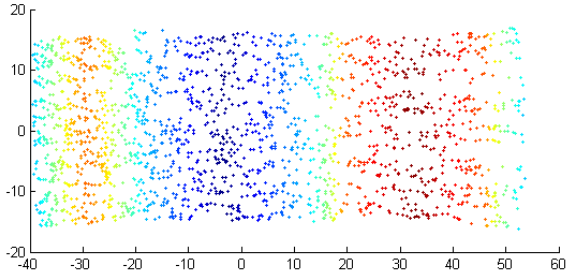


Figure 5: The 2D embedding of the Swiss Roll produced by Isomap. The structure is correctly unfolded.

Isomap has no problems with the Swiss Roll as can be seen in Figure 5. Careful selection of the neighbor value k or the distance value ϵ is very important for Isomap to produce proper embeddings. When noise is present, the performance of Isomap is not as effective. For more detailed examination, see [eELS02].

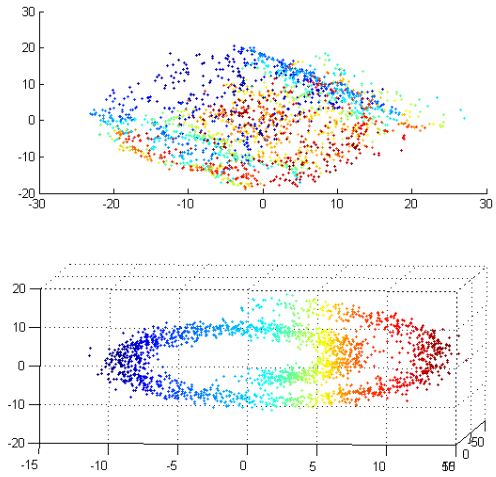


Figure 6: The Swiss Roll with noise value of 0.5. The performance is not as impressive.

Figure 6 shows how Isomap processes a noisy version of the Swiss Roll. It takes quite a lot to break down the procedure. Noise value of 0.35 (Maaten Toolbox units) still gave adequate results, but noise of 0.5 is enough to confuse the algorithm. On the other hand, even the high-dimensional structure is not so clear anymore.

3.2 NLM Implementation

Sammon's NLM was implemented as a self-contained C++ library. The implementation closely follows the description that was given in chapter 2.1. For the sake of comparison, a few notes should be made. The main iteration routine, `SammonNLMDist`, takes as an input both a convergence threshold value `epsilon` and a `maxIterations` parameter, and the iteration is carried out until either of these criterions are met. The NLM procedure in Maaten Toolbox (v0.4b) however does only a fixed number of iterations (set to 30) which makes comparing these algorithms directly quite difficult. As such, tests against Maaten Toolbox weren't done based on either the quality of embeddings or in the number of iterations required to converge.

Invoking the NLM routine can be done in one of two ways. If the input is a set of coordinates, call the `SammonNLM()` function. That will automatically convert the data points to a distance matrix and call `SammonNLMDist()` to perform the embedding. If instead the coordinates are not known and the input is a matrix of pairwise dissimilarities, just directly call the `SammonNLMDist()` function.

3.2.1 High-Performance NLM on GPU

The pseudo-Newton method admits a fairly good capability for parallelization and this was the major topic of investigation for the project. For the experiment, commodity graphics hardware (Ati Mobility Radeon X1600) was used. Modern GPUs contain several stream processing units that are well suited for batch processing large amounts of data. Therefore, the CPU implementation of the NLM algorithm was refactored and converted to be run on a graphics card, and several benchmark tests were run for comparison. This experiment immediately showed the performance gains as initial unoptimized GPU version yielded roughly a 2x speedup, and the final results are even more promising. Unfortunately, this implementation is more of a proof-of-concept nature than applicable for general use, since the software architecture it runs is restricted on the DirectX9 architecture. See Figure 7 for details.

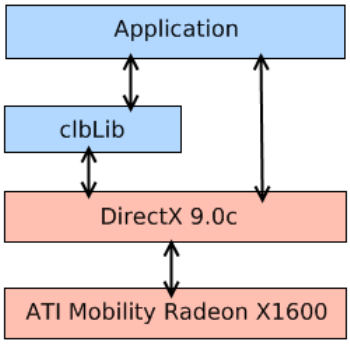


Figure 7: The architecture of NLM/GPU implementation.

The application runs on top of Microsoft's DirectX platform and uses a graphics framework library called clbLib. This combination was chosen mostly due to lack of alternatives at this point, but also because of the ease and familiarity of programming withing the framework. There are a few possibilities for performing real general-purpose computation on GPUs, most promising of them being CUDA[?], developed by nVidia Corporation. Unfortunately due to not having access to nVidia hardware, this alternative was not tested.

For successful implementation, some technicalities need to be addressed. One such is that the limit on the number of iterations in a for loop in shader code was found to be 255 iterations. Therefore, when computing the first and second order partial derivatives, the computation needs to be done in passes of 255 elements. In general, the code follows the common GPGPU path - using two sets of data, where one set is used as a source and one as a destination, the computations proceed to ping-pong between textures. A hard limit is caused by the maximum texture sizes of 4096×4096 and hence, the input data set size can only consist of 4096 points at maximum. This limitation could have been overcome by using arrays of textures or 3D textures, but was not tested due to the clumsiness of the API for the purpose.

As usual in GPGPU programming, the actual computation is carried out in Pixel Shaders (using Shader Model 3.0) by rendering a fullscreen overlay quad to a destination render target. Texture filtering needs to be closest-point, edges set to clamp and no mipmapping is utilized. Vertex Shaders need not be used.

The biggest bottleneck was identified to lie in the huge amount of texture accesses (data lookups) that need to be performed per kernel. To alleviate the issue, two tricks were employed. The first one was to perform an extra computation phase that precalculates the distances between low-dimensional data points at the beginning of an iteration. As expected, this resulted in a major increase in the overall performance.

Even still, the bottleneck lies in data access, and to further cut down on this, an idea to simplify the Hessian matrix even more by introducing zeroes on the diagonals was tested out. That is, instead of $\mathbb{H} = diag\left(\frac{\partial^2 f(x)}{\partial x_1^2}, \ldots, \frac{\partial^2 f(x)}{\partial x_n^2}\right)$, we would for example decimate to $\frac{1}{3}$rd of the values, that is, $\mathbb{H} = diag\left(\frac{\partial^2 f(x)}{\partial x_1^2}, 0, 0, \frac{\partial^2 f(x)}{\partial x_4^2}, 0, \ldots, \frac{\partial^2 f(x)}{\partial x_n^2}\right)$, or $\frac{1}{4}$th of the values, resulting in $\mathbb{H} = diag\left(\frac{\partial^2 f(x)}{\partial x_1^2}, 0, 0, 0, \frac{\partial^2 f(x)}{\partial x_5^2}, 0, 0, \ldots, \frac{\partial^2 f(x)}{\partial x_n^2}\right)$. In general, we can cut the number of texture accesses to a fraction of the original, which greatly simplifies the computations. This decimation parameter is controlled in the source code by the variable `derivCalcDecimateFactor`, a value of 1 meaning that no decimation is to be done and a value of k meaning that only every kth element of the Hessian is calculated.

The decimation trick naturally results in greater instability in convergence and to counteract that, three methods were employed. The first one is obviously to scale down our step factor α. In practice, values between 0.3 and 0.4 were used, whereas without decimating values as high as 0.7 to 0.9 were acceptable. Second, the implementation introduces a clamping mechanism to avoid data points being moved too far at any single iteration. The rationale for this lies only in the visual observation that, due to instability, the second derivative can take values arbitrarily close to zero, which in turn makes the step length arbitrarily long and some data points would end up being pushed really far from their real destination. The third one is to make sure that each diagonal element of the Hessian gets applied in turns. Which elements of the Hessian to calculate are rotated at each iteration, that is, if at the first iteration we would compute derivatives at indices $1, 4, 7, 10, \ldots$, then at the next iteration we shift all by one and calculate indices $2, 5, 8, 11, \ldots$ instead. An important remark was made that on data sets with relatively small number of points (< 512), decimation caused bigger instabilities than on large data sets and in fact for small data sets, decimating only increased the time to compute the embedding.

3.3 Isomap Implementation

Implementing Isomap on top of NLM requires writing algorithms for determining the k-nearest neighbors or ϵ-proximities that define the connectivity graph, and running all-pairs shortest paths algorithm on that data. In the implementation, Dijkstra's algorithm with Fibonacci Heaps is used, giving a $\mathcal{O}(kN^2 logN)$

performance. Isomap uses the NLM procedure as a subroutine, and as such requires strictly more time than NLM to compute.

4 Benchmarks

Different Sammon's NLM implementations were benchmarked on a 1.6GHz AMD Turion 64 x2 platform with an ATI Mobility Radeon X1600 graphics card. **Maaten MDS** is the implementation that was used as a baseline for the comparisons. **NLM/CPU** is the C++ implementation of the NLM algorithm running on a conventional single-core CPU, whereas **NLM/GPU** is the same algorithm running on graphics hardware. Two variants of the **NLM/GPU** were considered, **5-NLM/GPU** and **10-NLM/GPU**, which process only $\frac{1}{5}$th and $\frac{1}{10}$th of the diagonals of the Hessian, respectively.

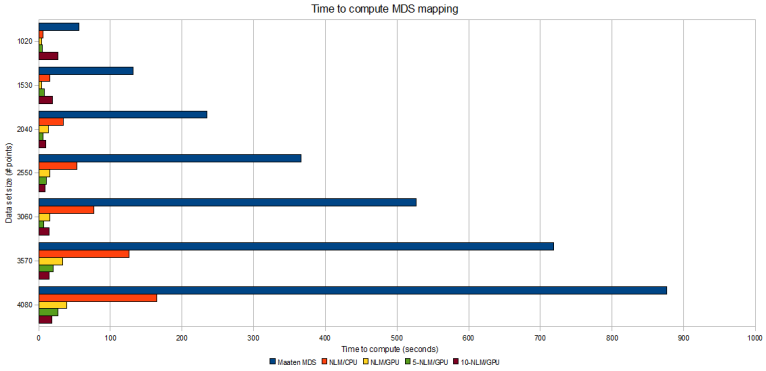


Figure 8: Running times on different data sizes (smaller is better).

The benchmarks were run on the traditional Swiss Roll dataset, generating random points on the manifold without any noise. Figure 8 shows the time the algorithms needed to finish. Note that the MDS algorithm in Maaten Matlab Toolbox only calculated a fixed number of, namely 30, iterations and returned afterwards. NLM/CPU and all the NLM/GPU variants finish instead when the stress function values level off below a certain threshold. It is clear that Matlab code suffers from some overhead, but this big difference in performance compared to NLM/CPU is quite surprising. The performance advantage of NLM/GPU over NLM/CPU is rather visible as well. The low performance of the aggressively decimating version 10-NLM/GPU on small data sets (1020 points) is explained due to the largely increased number of iterations that were needed to converge. Note that with data sets of 4080 points, the effect is the opposite.

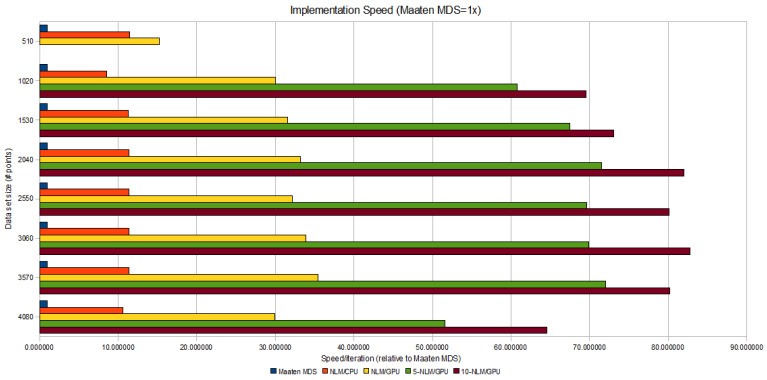


Figure 9: Speed relative to Maaten MDS (higher is better).

To see how efficient the implementations are at computing a *single iteration*, see Figure 9. This gives a better apples-to-apples comparison of Maaten MDS against NLM/CPU since Maaten MDS only does a fixed number of iterations. Additionally, it shows that the GPU implementations are indeed constrained by data accesses. The decimating versions 5-NLM/GPU and 10-NLM/GPU did not finish on 510 data points, since their behavior was too unstable when applied on a data set this small.

4.1 Scalability Concerns

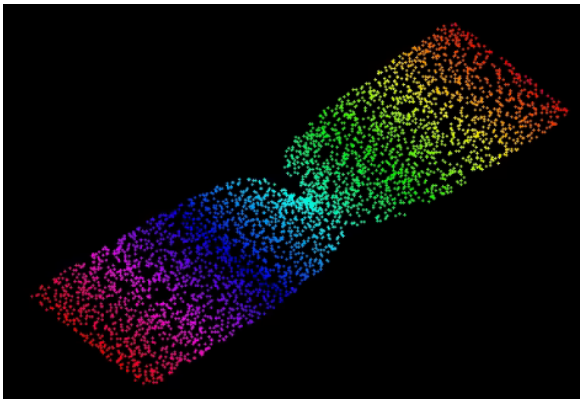


Figure 10: NLM erroneously converges to a steep local minimum.

Data set size	# errors
512	1
1024	19
2048	23
4096	35

Figure 11: The number of NLM/GPU convergence to a local instead of global minimum on Swiss Roll data sets of different sizes, out of 100 runs.

For getting more feedback from the GPU optimization process, the intermediate iteration stages are visualized as 2D plots. What was discovered is that as the data set sizes increase, the chance of converging to a local minimum

increases greatly. When running Isomap on the Swiss Roll with 4096 points, about one third of the time the iteration reached convergence to a configuration like that in Figure 10. Instead of flattening the Swiss Roll to a 2D plane, the optimization produced a stable state where the 2D plane is twisted once around itself. On the other hand, for optimizing the Swiss Roll using traditional MDS, where the output is resembles linear projection from $3D$ to $2D$, this never happened. The only explanation found for this is that when there are lots of data points that lie nearby in a stable configuration like the two ends of the plane in Figure 10, they have too much combined influence on other data points for the optimization process to overcome it.

Incidentally, after implementing the decimating variants of NLM/GPU, this issue seemed to be alleviated due to the increased instability that causes jittering in the output point coordinates. Tests were performed to determine how well this enhanced the optimization, but the results were highly inconsistent between runs and hence omitted.

In overall, it seems that a big problem with regards to scalability lie in the lack of robustness of the optimization method and in the increasing possibility of not finding the global minimum. It is unknown that by which extent real world data sets would exhibit these issues, though.

5 Conclusion

Some common NLDR techniques were reviewed, along with details on how to implement them. Two of these algorithms were developed to general-purpose C++ libraries and the applicability for parallellization in Sammon's NLM was demonstrated by converting it to a highly performing GPU program. The superiority of NLM/GPU over single-core CPU versions in speed was benchmarked. Additionally, the performance of custom C++ code was compared and found to well outperform that of the Maaten Toolbox Matlab code.

Feasibility of converting the other techniques as GPU programs was evaluated even though actual implementations were not written. A notable issue for scalability in NLM was found to lie in the quality of the optimization method used, instead of time or memory consumption as would've been expected.

5.1 Future Work

There are several things that could have been explored further. First and foremost, the uneasiness of DirectX as a GPGPU platform is notable and should be rid of. This would naturally lift the restrictions on data set sizes and would certainly improve the performance as well. Testing how big data sets are feasible after that, and whether the Isomap assumption "more data $\rightarrow$ better embedding" still holds in practice could be worthwhile.

Implementing the other examined algorithms, (Landmark) Isomap and Isotop, and comparing them against the performance of NLM would be an interesting topic as well.

What comes to the implementation of NLM/GPU, there are several open questions. As was seen, when data set sizes increase, the chance of getting stuck to a local minimum increases as well and can become a serious problem. Examining how local search techniques or simulated annealing -types of methods could help overcome this might be useful. In all the benchmarks, the learning rate was kept fixed and the iteration finish threshold was not changed - either of these could help in avoiding local minimas.

Reducing the amount of data lookups by blindly decimating the number of elements of the Hessian to calculate might not be the most reasonable method. Perhaps some kind of adaptive technique that estimates which elements of the Hessian actually affect the data point and discarding the calculation of too small values would give better performance.

6 References

References

[cor01] *Introduction to Algorithms*. McGraw-Hill, 2001.

[dBCLV03] E. de Bodt, M. Cottrell, P. Letremy, and M. Verleysen. the use of self-organizing maps to accelerate vector quantization, 2003.

[dST03] V. de Silva and J. Tenenbaum. Global versus local methods in nonlinear dimensionality reduction, 2003.

[eELS02] Mukund Balasubramanian et Eric L. Schwartz. The isomap algorithm and topological stability. *Science*, 2002.

[HB98] T. Hofmann and J. Buhmann. Competitive learning algorithms for robust vector quantization, 1998.

[Kru64] J. B. Kruskal. Nonmetric multidimensional scaling: A numerical method. *PsychoMetrica*, 1964.

[LV02] J. Lee and M. Verleysen. Nonlinear projection with the isotop method, 2002.

[LZ] Yang Zhengping Wu Xincai Luo Zhongwen, Liu Hongzhi. Self-organizing maps computing on graphics processing unit. In *"ESANN'2005 proceedings"*.

[nld07] *Nonlinear Dimensionality Reduction*. Springer, 2007.

[Tea00] "Joshua B. Tenenbaum and et al.". A global geometric framework for nonlinear dimensionality reduction. *Science*, 2000.

[vdM07] L.J.P. van der Maaten. An introduction to dimensionality reduction using matlab. Technical report, MICC, Maastricht University, 2007.

[Web02] Andrew Webb. A kernel approach to metric multidimensional scaling. *SSPR et SPR*, 2002.