Multilevel Multidimensional Scaling on the GPU

by

Stephen F. Ingram

B.S., Georgia Institute of Technology, 2004

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

November, 2007

© Stephen F. Ingram 2007

Abstract

We present Glimmer, a new multilevel visualization algorithm for multidimensional scaling designed to exploit modern graphics processing unit (GPU) hardware. We also present GPU-SF, a parallel, force-based subsystem used by Glimmer. Glimmer organizes input into a hierarchy of levels and recursively applies GPU-SF to combine and refine the levels. The multilevel nature of the algorithm helps avoid local minima while the GPU parallelism improves speed of computation. We propose a robust termination condition for GPU-SF based on a filtered approximation of the normalized stress function. We demonstrate the benefits of Glimmer in terms of speed, normalized stress, and visual quality against several previous algorithms for a range of synthetic and real benchmark datasets. We show that the performance of Glimmer on GPUs is substantially faster than a CPU implementation of the same algorithm. We also propose a novel texture paging strategy called distance paging for working with precomputed distance matrices too large to fit in texture memory.

Contents

Abstract ii

Contents iii

List of Figures v

Acknowledgements vii

Disclaimer viii

1 Introduction 1

2 Previous Work 5

2.1 Classical Scaling 5

2.2 Distance Scaling by Nonlinear Optimization 7

2.3 Distance Scaling by Force Simulation 8

2.4 GPU Layout Approaches 9

3 Glimmer Multilevel Algorithm 10

3.1 Multigrid/Multilevel Terminology 10

3.2 Multilevel Algorithm 11

3.3 Multilevel Parameter Selection 13

3.4 GPU Considerations 15

3.5 Restriction 16

4 GPU Stochastic Force 20

4.1 GPU-Friendly MDS 20

4.2 Stochastic Force Algorithm 21

4.3 Termination 22

4.3.1 Average Point Velocity 23

4.3.2 Sparse Normalized Stress 25

4.4 Stochastic Force on the GPU 28

5 Scalability With Distance Matrices 34

5.1 Paging and Feeding Performance 38

6 Results and Discussion 41

6.1 Complexity 42

6.2 Performance Comparison 43

6.2.1 Datasets 44

6.2.2 Layout Quality 44

6.2.3 Speed and Stress 46

6.2.4 Summary 49

6.3 Comparing Distance To Classical Scaling 50

6.4 GPU Speedup 52

7 Conclusion and Future Work 62

Bibliography 64

List of Figures

1.1 MDS Case Studies 2

3.1 Multigrid and Glimmer Diagrams 11
3.2 Pseudocode for the Glimmer algorithm. 13
3.3 Stress and Timing Results for Decimation Factor Tests 17
3.4 Stress Results for Minimum Set Size Tests 18
3.5 Timing Results for Minimum Set Size Tests 19

4.1 Filtered and Unfiltered Velocity Signals 24
4.2 Physical Signals Comparison 26
4.3 Stress and Sparse Stress per Iteration 27
4.4 GPU-SF Algorithm Stages 29
4.5 GPU-SF Algorithm Textures 30

5.1 GPU-SF Texture Memory Requirements 36
5.2 GPU-SF Memory Limitations 38
5.3 A Graph Layout With Distance Feeder 40

6.1 MDS Layout Comparison 54
6.2 Glimmer Versus GPUSF Layout 55
6.3 `grid` Stress and Timing Graphs 56
6.4 `shuttle` Stress and Timing Graphs 57

6.5 `docs` Stress and Timing Graphs 58
6.6 Log-log Stress Versus Time Scatterplots 59
6.7 Classical Scaling And Distance Scaling Simplex Layouts 60
6.8 Glimmer CPU Versus GPU Speed 60
6.9 Glimmer GPU Speedup 61
6.10 GPU Startup and Overhead Costs 61

Acknowledgements

This work required far more labor than I originally estimated. Like many of my personal projects it would have languished without the help of several key individuals. Obviously my supervisor, Tamara Munzner, was key in its development. Without her rigorous standards, tireless efforts, and (most importantly) good humor, this thesis would be a showcase of the inane. I credit her also with bringing aboard my co-supervisor, Marc Olano. Marc's sound, creative insight repeatedly simplified and enriched the algorithm in this thesis. My colleagues in the Infovis group at UBC were also a great source of ideas, constructive criticism, and friendship. Most of all, I thank my long-suffering companion Kelsey and daughter Eleanor. You both transformed what could have been drudgery into an adventure.

Disclaimer

The following paper [13] is based on the material contained in this thesis and has been submitted for publication:

- Stephen Ingram, Tamara Munzner, Marc Olano, *Glimmer: Multilevel MDS on the GPU*, submitted for publication.

Chapter 1

Introduction

Multidimensional scaling, or MDS, is a technique for dimensionality reduction, where data in a measured high-dimensional space is mapped into some lower-dimensional target space while minimizing spatial distortion. MDS is used when the inherent dimensionality of the dataset is conjectured to be smaller than dimensionality of the measurements. When dimensionality reduction is commonly used for information visualization applications, the low-dimensional target space is 2D or 3D and the points in that space are drawn directly, in hopes of helping people understand dataset structure in terms of clusters or other proximity relationships of interest [6]. MDS is only one of many dimensionaly reduction algorithms. Others include Isomap [28], Locally Linear Embedding [26] (LLE), and Gaussian Process Latent Variable Models [18].

In MDS, the goal is to find coordinates for N points in a low-dimensional space, where the low-dimensional distance d_{ij} between points i and j is as close as possible to the corresponding high-dimensional distance, or *dissimilarity*, δ_{ij}. Input can consist of high-dimensional points, with δ_{ij} computed from coordinates, or of an $N \times N$ distance matrix, Δ, allowing an arbitrarily complex distance metric.

As an example of where MDS produces meaningful results, consider a dataset from the Wisconsin Breast Cancer Database[1]. It contains 683 points representing tumors and 9 measurements about each tumor such as cell uniformity and clump

[1] Taken from the UCI ML Repository: `www.ics.uci.edu/~mlearn/MLSummary.html`

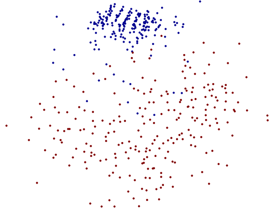


Figure 1.1: MDS layouts may reveal useful structures, as in the tumor dataset from the Wisconsin Breast Cancer Database where blue represents benign and red represents malignant. In typical MDS usage, the color labels do not exist and the proximity relationships such as clusters are used to discern classification.

thickness. We say that this dataset is 9-dimensional and the distance δ_{ij} between two tumors i and j is the Euclidean distance between the 9-dimensional points constructed from the measurements of each tumor. Figure 1.1a shows the resulting visualization of this dataset after using MDS to map the data from 9 to 2 dimensions. The figure shows that points (tumors) either belong to a tightly clustered low-variance group or are part of the more spread out, higher-variance cluster of points. We color points based on ground-truth labels to show that the blue points are benign tumors and the red points are malignant tumors. In this instance, there is a strong correlation between the spatial strctures observed after applying MDS and a latent feature of the data.

Alternatively, MDS may result in the loss or distortion of structure. For example, consider a three dimensional dataset constructed by regularly sampling points from the surface of a sphere. There is no embedding of this dataset in two dimensions without some distortion because the sphere's intrinsic dimensionality is three.

MDS algorithms work by minimizing an objective function based on the discrepancy of high-dimensional and embedding distances. A standard *stress* error

metric is the normalized stress metric:

$$stress(D,\Delta)^2 = \frac{\sum_{ij}(d_{ij}-\delta_{ij})^2}{\sum_{ij}\delta_{ij}^2} \tag{1.1}$$

which has a significant cost of $O(N^2)$ to compute for the N points of the dataset. If the embedded distances match the original distances of the data, then $stress = 0$. Stress becomes larger as the spatial distortion between the embedding and the original data increases.

MDS algorithms vary in precisely what form the stress function takes and in how they minimize the stress function. Some are approximate while others are exact, some are iterative while others are completely analytical. Such diversity in algorithms leads to diversity in the quality of the results and the speed at which they are computed. Section 2 gives a brief overview of various relevant classes of existing MDS algorithms and their underlying characteristics.

One class of MDS algorithms that has had significant influence in information visualization is the class of iterative, force directed algorithms. In such algorithms, data points are modeled as particles in space attached to other particles with springs with an ideal length proportional to the original distance δ. The algorithm computes a simulation by integrating forces until the physical system settles down into a state of minimal energy. At this point computation halts and the final positions of the particles are assigned the resulting coordinates of the data. Naïve implementations of such algorithms can be computationally expensive and prone to converge to local minima.

We present three substantial improvements to the iterative class of MDS algorithms based on simulated forces. First, we improve algorithm speed by exploiting the modern PC graphics processing unit (GPU) as a computational engine. Second, we introduce a cheap and reliable linear-time termination condition based on the convergence of an approximation of the normalized stress function. Finally, we

devise a simple multilevel strategy that demonstrably helps to avoid local minima. We compare the resulting algorithm, called Glimmer, to a wide variety of MDS algorithms showing the advantages of our approach in terms of speed and accuracy. While our algorithm is applicable to produce output in N dimensions, our implementation only scales to embeddings in two dimensions.

Below, we discuss the rich previous work in Chapter 2, and then present the core ideas of the Glimmer multilevel algorithm and GPU-SF algorithm in Chapter 3. We cover GPU considerations in Chapter 4, providing the details of our GPU-based algorithms. In Chapter 5 we propose an algorithm for handling the general-case where high-dimensional distances are not necessarily Euclidean. In Chapter 6 we compare Glimmer to several other MDS algorithms in terms of complexity, speed, quantitative accuracy with respect to the stress error metric, and qualitative accuracy of layouts for datasets where ground truth is known for shape or clustering.

Chapter 2

Previous Work

The foundational ideas behind multidimensional scaling were first proposed by Young and Householder [30], then further developed by Torgerson [29] and given the name of MDS. Considerable research has gone into devising faster and more robust solutions. In the interests of space we focus on the foundational work and the three most commonly employed categories of current techniques: classical scaling methods, distance scaling by nonlinear optimization, and distance scaling by force-directed approaches. In the descriptions below, N is the number of points, and L is the dimensionality of the low-dimensional target space, while H is the dimensionality of the high-dimensional input space.

2.1 Classical Scaling

Classical scaling methods compute exact or approximate analytical solutions to the global minimum of the *strain* function

$$strain(X) = \|XX^T - B\|_F^2$$

where X is the matrix of the coordinates of the low-dimensional configuration, $B = \frac{1}{2}J\Delta J$, and J is the so-called centering matrix $J = I - n^{-1}1^T1$ which centers the high-dimensional coordinates around the origin. If the underlying distances in the distance matrix Δ are Euclidean, then B is the transformation from a distance

matrix to an inner product, or Gram matrix. Rather than minimizing the discrepancy between distances as in stress, classical scaling minimizes the discrepancy between inner products. Although strain is closely related to stress, it may have a very different minimum. These spectral methods find embedding coordinates X by computing the top eigenvectors of B sorted by decreasing eigenvalue. Classical scaling is equivalent to Principal Component Analysis, another popular dimensionality reduction method, when the coordinates are centered at the origin and normalized such that the largest distance is equal to 1. The original algorithm, Classic MDS [29, 30] computed a costly $O(N^3)$ singular value decomposition of this matrix. Modern classical scaling methods quickly estimate the eigenvectors using the power method or other more sophisticated iterative methods that employ $O(N^2)$ matrix-vector products.

A host of Nyström methods [23] have recently been proposed to avoid the $O(N^2)$ computation of Δ altogether, using a subset of that matrix to approximate the eigenvectors. These include FastMap [9], Landmark MDS [8], and PivotMDS [4]. We use PivotMDS as an exemplar in the Glimmer performance comparison of section 6, since it was shown to be the most accurate classical scaling approximation algorithm [4]. All of these techniques achieve dramatic speed improvements by reducing the complexity of classical scaling to essentially $O(N)$. However, in Section 6 we discuss the limitations of these approaches in handling sparse, high-dimensional datasets for visualization . The Glimmer approach of distance scaling yields higher quality layouts in these cases, and has competitive speeds whenever the visual quality is equal.

2.2 Distance Scaling by Nonlinear Optimization

Optimizing the stress function using gradient descent to find a low-error embedding was pioneered by Kruskal [17]. Optimization approaches can easily incorporate weights to emphasize certain types of distances over others, or handle missing values gracefully, in a way that is difficult using spectral methods. Unlike classical scaling methods, optimization approaches are subject to local minima. De Leeuw's accurate SMACOF [7] algorithm monotonically converges to a stationary point by minimizing a quadratic approximation of the stress function at each iteration, resulting in provably linear convergence but at a large cost of $O(N^2L)$ per iteration. Gansner *et al.* [11] use a SMACOF-based approach to stress majorization for graphs, but the sparsification and edge-weighting modifications they propose are not suitable for general MDS because in general, data topology is unknown. Computing the nearest-neighbor topology of general datasets is naively an $O(N^2)$ pre-processing procedure. Accelerations of this technique are not straightforward to apply in high dimensions due to the fact that the k-nearest neighbor graph generated from k-d trees or farthest-point sampling may not be connected.

The recent Multigrid MDS [5] algorithm employs the multigrid method for discretized optimization problems, using SMACOF as a relaxation operator and terminating in a small, constant number of iterations. The hierarchical approach helps avoid local minima and makes substantial speed improvements over SMACOF alone, but the scalability is still limited, with a layout of 2048 points taking 117 seconds and requiring precomputation of the data topology. We were inspired by the power of a hierarchical multigrid approach in the design of Glimmer, but use very different operators for the three multigrid operations of restriction, relaxation, and interpolation (described in more detail in Section 3.1).

2.3 Distance Scaling by Force Simulation

Force-based MDS algorithms use a mass-spring simulation to optimize the stress function, generating forces in proportion to the residual between low and high-dimensional distances. This is a kind of nonlinear optimization where each point computes a linear estimate of the gradient and moves in a fixed length according to the integration time-step. However, the previous work and underlying machinery of the force-simulation approach is different enough from standard nonlinear optimization to warrant its own category. These methods are intuitive to understand, easy to program, can support weights and interactivity, and typically produce lower-stress results than Classic MDS. Their drawbacks include numerous parameters to the physical system such as damping constants and time-step size, the introduction of oscillatory minima, and the possibility of local minima.

Force-simulation MDS is only tangentially related to other particle simulation algorithms. N-Body and particle simulation accelerations such as Barnes-Hut [1] and the Fast Multipole Method [12] rely on force models where forces are entirely generated from the configuration of the embedding space. In distance scaling, forces come from the difference between the configurations of two different spaces, the original dataset space and the low-dimensional embedding space, and so the accelerations proposed in these methods are not applicable.

The basic force-directed approach has a complexity of $O(N^3)$, with an $O(N^2)$ cost per iteration for N iterations. We discuss the lurking assumptions surrounding the use of N iterations in section 4.3. The CPU-based stochastic force approach introduced by Chalmers [6] reduces the per-iteration cost to $O(N)$, for a total $O(N^2)$ cost. This stochastic algorithm is used as a subsystem to two further refinements, with complexity $O(N^{5/4})$ [20] and $O(N \log N)$ [15]. Glimmer uses a GPU variant of the stochastic approach (GPU-SF) with an improved termination condition as

a subsystem. We discuss its limitations with respect to accuracy and convergence below. We compare Glimmer against three of these approaches in Chapter 6.

2.4 GPU Layout Approaches

GPUs have been shown to improve the speed of many general purpose algorithms including graph layout and classical scaling, but have not been previously applied to minimizing the stress function directly.

Reina and Ertl [24] proposed a GPU version of the FastMap algorithm, a classical scaling approximation algorithm, achieving considerable speedup over a CPU implementation. However, the technique only accelerates the mapping into low dimensional space. The initial computation of the high dimensional distances, the costliest part of the Nyström algorithms, is not sped up.

Frishman and Tal [10] take advantage of GPU parallelism to increase the speed of their dynamic graph layout algorithm. Force-directed graph layout does have deep similarities to force-directed MDS. However, their edge-collapsing coarsening stage relies on the graph topology as input, which would require precomputation of a nearest neighbor graph for the more general case of arbitrary MDS data. The energy function they compute on the GPU ignores pairwise distances, and thus does not minimize stress. They use the CPU for initial placement and for spatial partitioning, whereas Glimmer runs all stages entirely on the GPU.

We further discuss the suitability of previous algorithms for speedup using GPU parallelism in Section 4.1.

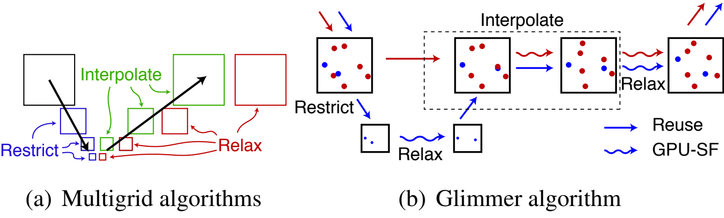
Chapter 3

Glimmer Multilevel Algorithm

Glimmer is a force-based MDS algorithm which uses a recursive hierarchical framework to improve accuracy and to reduce computation. Unlike other hierarchical MDS algorithms, Glimmer is specifically designed to exploit GPU parallelism at every stage of the algorithm. We use the multigrid vocabulary, because we were inspired by those methods, but we call our algorithm *multilevel* because our final formulation differs from the strict definition of multigrid algorithms.

3.1 Multigrid/Multilevel Terminology

In our description of the multilevel hierarchy, we consider the highest level to be the input data, with lower levels being nested subsets of that data reduced in size by a fixed decimation factor. Multigrid methods use three operators at each level: *restriction*, *relaxation*, and *interpolation*, as shown in Figure 3.1. Loosely speaking, restriction performs the decimation to build the hierarchy, relaxation is the core computation operator that reduces the error at a specific level, and interpolation passes the benefit of the latest relaxation computation up to the next level. In typical multigrid methods, a so-called *v-cycle* of restriction, relaxation, and interpolation is repeated several times. However, the Glimmer operators were designed to converge in a single cycle.



(a) Multigrid algorithms (b) Glimmer algorithm

Figure 3.1: Multigrid and Glimmer Diagrams. **a)** The multigrid *v-cycle*. **b)** The Glimmer multilevel algorithm. The restriction operator builds the hierarchy by sampling points. GPU-SF is used as the relaxation operator at each level, with all points allowed to move, and as the interpolation operator, with only the points newly added to the level allowed to move. Lower levels untwist complex layouts while higher levels converge quickly because of computation at the lower levels.

3.2 Multilevel Algorithm

Figure 3.1 shows a diagram of the Glimmer multilevel algorithm as a single v-cycle. The pseudocode is given in Figure 3.2. The restriction operator we use to construct the multilevel hierarchy simply extracts a random subset of points from the current level. In Glimmer, we use a decimation factor of 8 between each level, and stop when the size of the lowest level is less than 1000 points. These parameter choices were empirically chosen after analyzing the speed/quality behavior for decimation factors of several powers of 2 and a variety of minimum set sizes above and below our final choices. Then, we traverse upwards to the top, alternating runs of the relaxer for the current level with interpolating the results up to the next level. In this traversal, we use stochastic force as our relaxation operator; that is, we perform iterations of a stochastic force MDS algorithm (GPU-SF) for all the points at a particular level until the system converges. Perhaps surprisingly, we also use the stochastic force algorithm as our interpolation operator. We fix the locations of previously relaxed points, moving just the newly added points to fit the current configuration. Again, we stop the interpolation step when the stochastic force sub-

system converges. We continue with the traversal, freeing the formerly fixed points for the relaxation step. We halt after running the relaxation operator on the highest level that contains all points.

At the low levels, only a small subset of the points are involved in the computation, so the system converges quickly. The higher levels converge in few iterations because the points placed at lower levels are likely to be close to their final positions. In particular, although the relaxation step at the highest level involves running stochastic force on all the points in the input dataset, the system converges more quickly than it would if the stochastic force algorithm were run with the points at random initial positions. Glimmer terminates after completing a single v-cycle.

The average total time required across all levels of the Glimmer multilevel approach is roughly the same as when the stochastic GPU-SF force algorithm used alone, as shown in Figure 6.3(a). The major difference between Glimmer and the GPU-SF subsystem alone is accuracy and convergence. The multilevel Glimmer approach is more successful at avoiding local minima, which can give rise to twisted manifolds in the low-dimensional placement, as shown in Figure 6.2. Susceptibility to local minima is often cited as a weakness of the force-based methods, but using a multilevel approach atop a force-based subsystem allows the accurate global structure of the point set to be found during the cheap iterations at the lower levels. At the higher levels, the local structure is refined within the global context inherited from lower levels through interpolation. When GPU-SF falls into a local minimum that Glimmer avoids, GPU-SF may terminate quickly but with visually unacceptable results.

```
restrict( points ):
  if(size(points) < threshold)
    return emptyset;
  return randomsubset(points);
runGPUSF( fixed, free ):
  while(!converged)
    stochasticforce(points in free)
glimmer( points ):
  if(points == emptyset)
    return;
  subset = restrict(points);          // restrict
  glimmer(subset);
  runGPUSF(subset, points - subset); // interp
  runGPUSF(emptyset, points);         // relax
```

Figure 3.2: Pseudocode for the Glimmer algorithm.

3.3 Multilevel Parameter Selection

This section describes our method for determining appropriate values for the decimation factor F and minimum set size M parameters. We construct sets of test values for each parameter; for F we use the set $f = \{2,4,8,16,32\}$ and for M we use $m = \{100,500,1000\}$. We then construct the cartesian product of the two sets $t = f \times m$ and run Glimmer using each element of t as input parameters on a series of cardinalities of different datasets, recording the time and final stress for each run of the algorithm. We use the stress and timing data to directly compare the effects of ranges of multilevel parameters on stress and time for different datasets.

Figure 3.3 shows the effect of different decimation factors F for stress and timing across a range of randomly-sampled cardinalities of a real-world dataset, `shuttle`, described in section 6.2.1. By taking different size samples from the dataset we can simultaneously observe the sensitivity of a parameter choice to initial random configuration and dataset size. Each parameter choice exhibits some noise in the stress output, but only when the stress is above 0.03 is the local min-

ima visually different. When F is 2, 16, or 32, we observe spikes in the stress function indicating the presence of these noticable local minima. Using only stress as a metric, 4 or 8 are sensible choices for the parameter F with relatively stable convergence behavior. Looking at the timing graph, the correct choice of parameter value becomes clearer. When $F = 2$, the algorithm suffers a noticable time penalty. Likewise, when F is 16 or 32, significant spikes appear. These are uncorrelated with the spikes in stress. Comparing the overall running times when $F = 4$ and $F = 8$, we observe a performance improvement of approximately 2 seconds of $F = 8$ over $F = 4$. We have observed that the behavior of the parameter F is independent of dataset and the minimum set size parameter M.

Similarly, Figure 3.4 shows the effect of using different minimum set sizes M on final stress across cardinalities of the dataset `docs` (see section 6.2.1). We add the final stress of GPU-SF on the same data as a reference to compare the stress convergence behavior. There is a noticeable plateau in the stress results for GPU-SF which indicates convergence to a local minimum. When we set the minimum set size M before this region, the suboptimal convergence pattern disappears. Figure 3.5 illustrates why we do not make M as small as possible. On datasets smaller than 1000, such as the one shown in Figure 3.4, there is a noticable performance penalty for the multilevel approach. In the interests of maximizing speed for smaller datasets, it is useful make the value of M as large as possible. Rather than require the user to find regions which are prone to local minimum, we empirically observe that GPU-SF is more prone to local minima on datasets with cardinality greater than 1000.

3.4 GPU Considerations

The Glimmer algorithm can run on a CPU, and we have implemented an optimized Java prototype as a proof of concept and to allow direct timing comparisons. However, our restriction, relaxation, and interpolation operators are all carefully designed to exploit GPU parallelism. Our use of the GPU does not affect convergence or accuracy, but brings a dramatic speed improvement over previous MDS approaches.

Modern GPUs have a user-programmable pipeline of highly parallel processing stages, called *shaders*. The first stage operates on a stream of vertices, the second stage operates on a stream of geometry, and the final stage operates on a stream of pixels. The GPU pixel processors can be considered as a single-instruction multiple-data (SIMD) unit operating in parallel on a subset of pixels in the stream, where the SIMD batch size varies from 16 to 1024 in recent GPUs. These units have random read/write access to data stored in texture memory, so textures can be used in place of arrays. Computation occurs when a textured polygon is rendered using a shader. Typical computations take multiple rendering passes, where the only communication channel between processing units is writing a texture in one pass, then reading from it in a later pass.

The porting of general purpose algorithms to graphics hardware is subject to certain pitfalls. For example, memory accesses are a common bottleneck for general purpose GPU programs. The number of accesses per shader program execution GPUs should be kept relatively small and constant to avoid performance problems. We discuss how this pitfall impacts our choice of MDS subsystem in section 4.1.

Glimmer and GPU-SF are general approaches that do not depend on specific hardware features of a particular GPU. The most recent G80 nVidia GPUs handle all three shader types with a shared set of SIMD clusters that can be programmed

with a general-purpose parallel languaged called CUDA [21]. Although our algorithms could be implemented CUDA, we can operate across several generations of GPUs by using a more generic model of GPU processing. Our algorithms run on any card that supports pixel shaders, and we compare speeds on two different generations of cards in Section 6.

3.5 Restriction

The restriction operator creates a multilevel hierarchy from nested subsets of the input data, randomly sampled from the enclosing set. We first run an $O(N)$ preprocessing step to randomly permute the input data on the CPU before loading it into texture memory on the GPU. We then can easily access nested rectangles in texture memory to solve the sampling problem. Traversing the hierarchy from bottom to top in the second leg of our v-cycle is handled by enlarging the size of the rendering polygon, with no shader code or extra storage required to create the hierarchy of levels. Our solution avoids the need to do random sampling on the GPU, which would be slow.

Our restriction operator does not require any explicit extra computation, and specifically does not rely on having any geometric locality information. In contrast, the previous Multigrid MDS approach [5] must carry out an preprocessing step to find nearest neighbors in the high-dimensional space. In our approach, high-dimensional neighborhoods around each point are gradually discovered during the stochastic interpolation and relaxation operations.

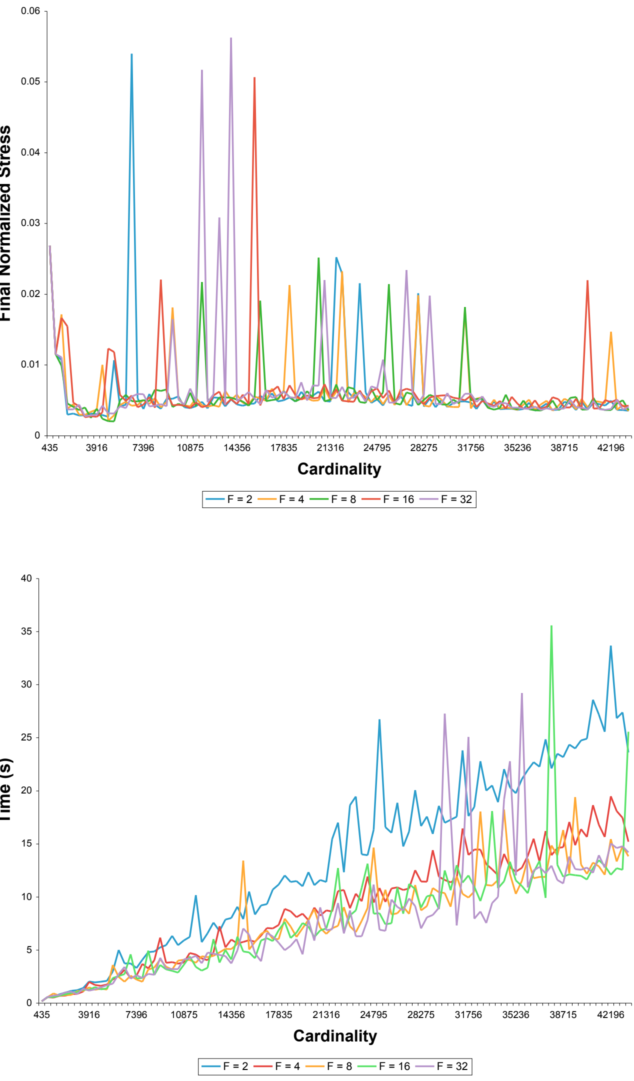


Figure 3.3: Stress and Timing Results for Decimation Factor Tests. **a:** Graph showing the effect of decimation factor F on the final stress of a configuration computed by Glimmer across a range of random samples from a dataset. Only when F is 4 or 8, do we not observe spikes representing local minima. **b:**. Graph showing the effect of decimation factor F on the final time across the same range of dataset cardinalities. When F is 8 we observe the fastest average convergence time.

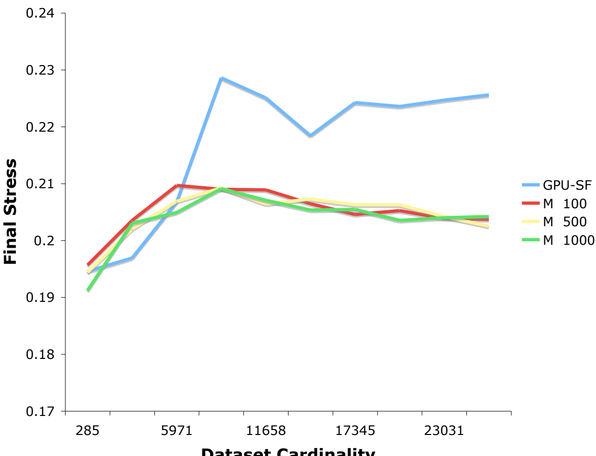


Figure 3.4: This graph shows the effect on final stress of different values for the Glimmer minimum set size parameter *M* across a range of randomly sampled datasets. Also included are the final stress results for GPU-SF on the same data. When processing datasets with more than 10000 points, GPU-SF regularly terminates at a local minimum. By setting the minimum set size *M* before this region, we can help avoid the problem.

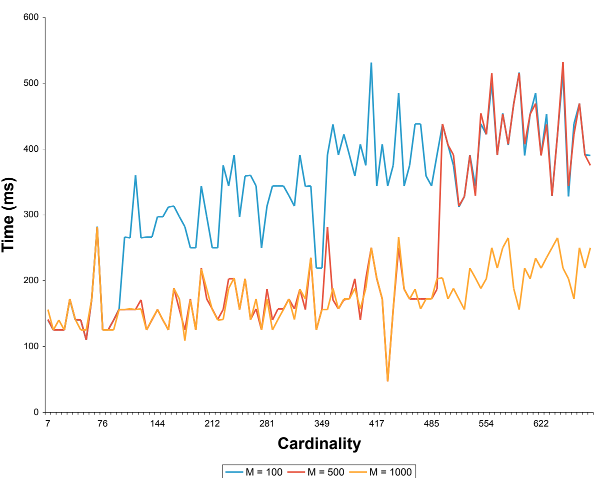


Figure 3.5: This graph shows the effect on convergence time of different values for the Glimmer minimum set size parameter M across a randomly sampled range from a small dataset. On larger datasets, increasing M is relatively imperceptible, but on datasets less than 1000, the effect is significant and motivates the need to maximize M.

Chapter 4

GPU Stochastic Force

GPU-SF is our GPU-friendly stochastic force MDS solver used as a subsystem in Glimmer, inspired by the Chalmers [6] algorithm. Without GPU acceleration, the GPU-SF algorithm has nearly identical runtime characteristics with the CPU-based Chalmers one. The only differences are the new termination criteria that we propose, and the asymmetric force calculations.

4.1 GPU-Friendly MDS

Glimmer's relaxation and interpolation operators both benefit from rapid execution of a simple MDS subsystem, so we propose a GPU-friendly MDS algorithm. In general, algorithms whose iterations exploit a form of *sparseness* perform best on graphics hardware. By sparse, we mean a limited number of computations and non-local accesses per point, a number far less than the total number of points N. This restriction immediately disqualifies most MDS algorithms because of their reliance on *dense* matrices or submatrices for matrix-matrix or matrix-vector operations. Traditional force-based MDS is also dense, since each point must access every other point to compute its force.

On the other hand, most of the accelerated MDS algorithms that exploit sparseness may fail to achieve accuracy on certain datasets. For example, PivotMDS, Landmark MDS, and the parent-finding approaches of accelerated force-directed

MDS [4, 20] achieve their speedups by only considering a subset of rows of the input distance matrix. While distance matrices frequently exhibit considerable redundancy, these algorithms may discard important information in the selection of these rows.

We have identified the stochastic force algorithm [6] as especially appropriate for our requirements. Each point only references a small fixed set of other points during an iteration step, and the selection of this fixed set is not limited to any subset of the input. Thus, in a single iteration of the stochastic force algorithm, each point performs a constant amount of computation and accesses only a constant number of other points, regardless of dataset size.

4.2 Stochastic Force Algorithm

The stochastic force algorithm iteratively moves each point until a stable state is reached, but the forces acting on a point are based on stochastic sampling rather than on the sum of all pairwise distance residuals. More specifically, two sets of a small, fixed size are maintained for each point: a Near set, and a Random set. The forces acting on a point are computed using only the pairwise distances between the points in its two associated sets. Each set initially contains random points. After each iteration, any members of the Random set whose high-dimensional distance to the point is less than those in the Near set are swapped into that Near set. The Random set is then replaced with a new set of random points. After many iterations, the Near set will converge to the actual set of nearest neighbors. The retention of nearest-known-neighbors has the effect of maintaining the topology of the data in the embedding space, while the Random set functions both as a means of resolving the Near set and maintaining the global structure of the dataset in the embedding space. Chalmers proposes a Random set of size 10, and a Near set

of size 5. We use 4 for the size of each set to match the 4-element vector types supported by the GPU.

As with any force simulation algorithm, stochastic force has an integration method and parameters. We use Euler integration with a time step of 0.3 of both force and velocity. We normalize the sum of forces by a size factor of $1/(|Near| + |Random|)$. Furthermore, we dampen these forces by computing the relative velocity vector between each point and the points in its Near and Random sets, scaling it by a damping factor 0.3 and subtracting it from the force vector between these vertices.

4.3 Termination

Some previous iterative MDS algorithms do not have an explicit termination criterion, and depend on the user to monitor the layout progress and halt the computation when deemed appropriate [25]. Because we use the GPU-SF algorithm as a subsystem in Glimmer, we need to quickly and automatically determine the correct time to terminate computation. In other approaches [15, 20], the computation is run for a fixed number of iterations, usually N. Although linear convergence was proven for the SMACOF algorithm [7], it has been generally assumed for many force-directed approaches. We show in Chapter 6 that this assumption is not safe to make, frequently leading to overkill that wastes time, or underkill that halts computation before the layout is accurate.

A standard termination criterion for nonlinear optimization is to terminate when the gradient of the function converges to zero. In MDS, this criterion implies that the difference between iterations in the stress error metric given by Equation (1.1) converges to some small number ε. Computing stress for a configuration requires $O(N^2)$ computations. Producing this value at each iteration would be far more

expensive than the Glimmer algorithm itself. To remain speed competitive, the algorithm needs an termination condition requiring only $O(N)$ computations. We devised and evaluated two different approaches, one using the average point velocity and the other a sparse approximation of the stress function. The sparse stress approximation proved to be a superior termination condition because it exhibits monotonic behavior in a multilevel context.

4.3.1 Average Point Velocity

Previous algorithms such as Morrison's subquadratic MDS algorithm [19] terminate by checking whether the change in velocity of the particle system between iterations falls beneath a fixed threshold. This linear-time function returns the sum of the total velocities of each particle which have already been computed as part of the force simulation. We set the units of velocity to the number of pixels in the embedding layout traversed per second. We found that this strategy has several inherent problems. First, the magnitude of system velocity is dependent on the precise number of particles in the system. We normalize the signal by dividing total velocity by the number of active particles in the system. Second, the velocity of the system is very noisy and subject to oscillations, even with damping. We solve this problem by low-pass filtering the signal, specifically with a Hann-windowed sinc filter.

Third, using a single fixed termination threshold is problematic in the context of a multilevel algorithm. That is, an appropriate convergence threshold for a low level of the algorithm may be too high or low for another level. The initial stages of the Glimmer algorithm are generally the highest energy whereas later stages might see lower changes in velocity while making just as much qualitative progress. Our solution to this problem was to set the threshold to a fraction α of the highest

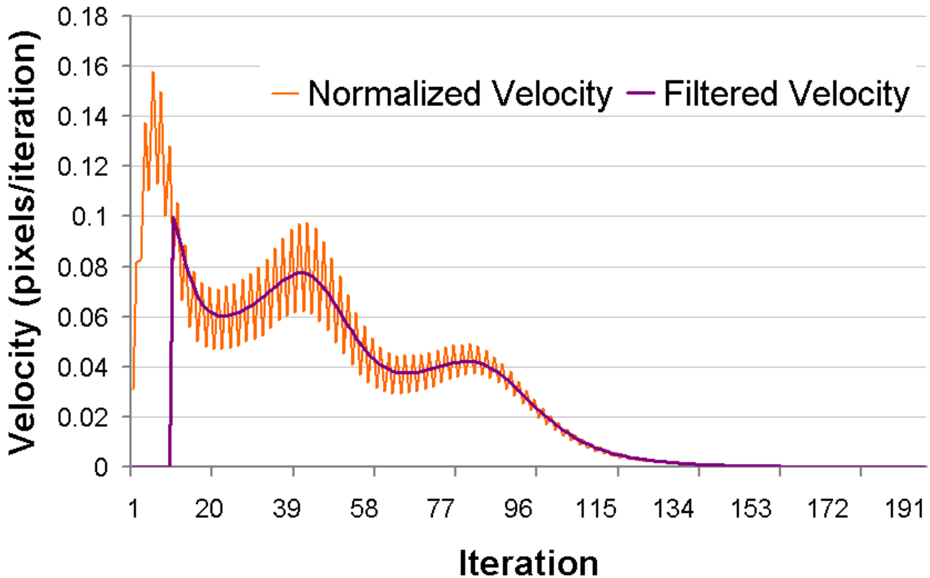


Figure 4.1: Graph of normalized velocity per iterations for GPU-SF algorithm on a grid dataset. After filtering high-frequency noise from the signal, low frequency noise remians resulting in non-monotonicity of the signal. This phenomenon makes normalized point velocity an undesirable choice in termination condition.

average velocity observed in that level. α should not be too low or the algorithm will waste time with cycles that do nothing. If α is too high the algorithm will terminate too soon. We found $\alpha = 1/10000$ to be a useful compromise.

Fourth, the most difficult problem with using system velocity as a termination criteria is low-frequency oscilliatory behavior that we observed in the signal that was not removed by our low-pass filter. Figure 4.1 shows a velocity signal with this behavior. We conjecture that the large-scale oscillation is a result of either the asymmetry of the forces applied in our algorithm, the randomness of the neighbor selection process, or that the system is ultimately based on spring-like forces. Whatever its cause, non-monotonicity in a termination condition poses difficult

problems for detecting system convergence. Eliminating such oscillations with filtering would require a window so large that much of the algorithm would be spent computing convolutions and far too many iterations would be spent in each stage of the algorithm computing enough signal to fill a window, frequently after the signal had converged. We tried a heuristic for detecting false convergence in the presence of such non-monotonicity. To detect oscillations, we checked if a positive slope exists across a window smaller than the window used to detect convergence. If positive slope exists, then we ignore convergence. This ad-hoc solution ultimately proved too sensitive, and the heuristic's window size and parameters required adjustment according to different datasets.

We examined two other physical characteristics of the system other than velocity in hopes of finding an appropriate and inexpensive termination condition. The first of these is the average *force* of the system, defined as the average sum of the active point forces. Computing this involves summing already computed force values. The second is the average *displacement*. Computing displacement over a window of D iterations requires storing the set of D previous iterations of layout coordinates and subtracting the Dth oldest set from the current coordinates and summing the absolute value of the results. Figure 4.2 shows a graph of these signals with respect to raw system velocity, filtered velocity, and normalized velocity over several stages of a Glimmer run. Normalized velocity, force, and displacement exhibit essentially identical behavior making none of them obviously superior to the other as a termination criterion.

4.3.2 Sparse Normalized Stress

We instead use an approximation of stress that we call *sparse normalized stress* based on the differences in distance values already computed. More specifically,

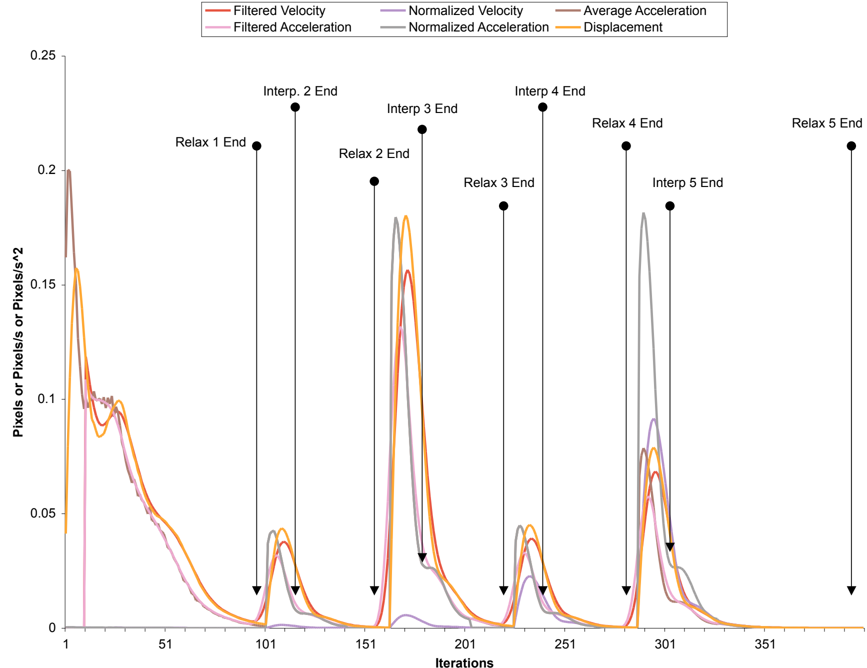


Figure 4.2: The behavior of several physical signals of the simulation over a Glimmer run. Since the signals share the same characteristics, none of them are more useful than the others as a linear-time termination condition. Instead, Glimmer relies on a sparse approximation of the stress function.

sparse normalized stress is defined as

$$sparsestress(D,\Delta)^2 = \frac{\sum_i \sum_{j \in Near(i) \cup Random(i)} (d_{ij} - \delta_{ij})^2}{\sum_i \sum_{j \in Near(i) \cup Random(i)} \delta_{ij}^2} \tag{4.1}$$

Here, $Near(i) \cup Random(i)$ is the union of the index sets for point i, requiring only $O(N)$ computations to compute the stress for a configuration.

Because the contents of these sets change at each iteration, the sparse stress value is noisy, making the raw function values inadequate as a convergence criterion. To remove this noise we treat sparse stress as a signal and apply a temporal low-pass filter, a windowed sinc in our implementation. The resulting smooth sig-

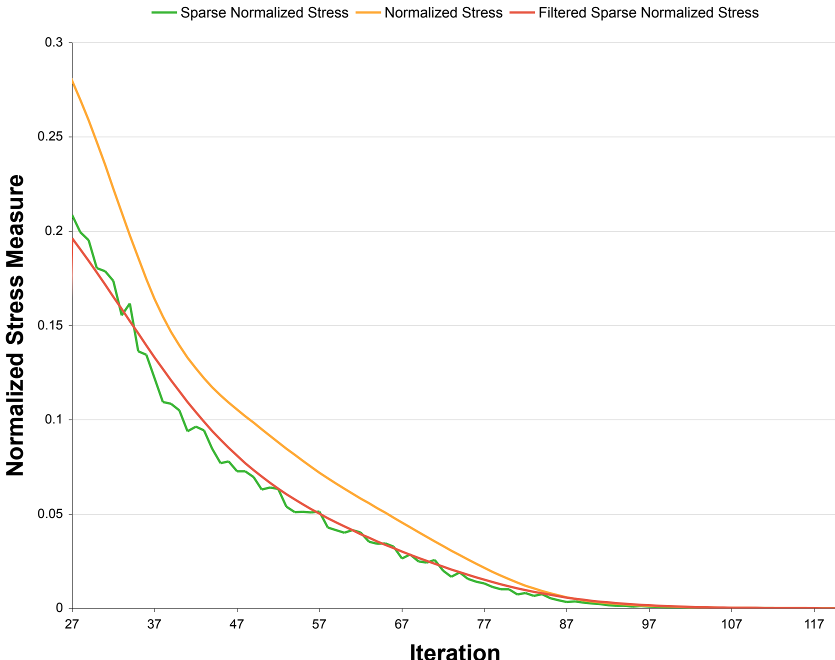


Figure 4.3: Stress and Sparse Stress per Iteration. GPU-SF uses a sparse approximation (green) of the normalized stress function (orange), which converges simultaneously and requires only minimal overhead to compute. We use a low-pass filter (red), because the noise in the unfiltered signal is much larger than the convergence threshold of $\varepsilon = .0001$.

nal closely mimics the behavior of the true normalized stress function, as shown in Figure 4.3. Since we are interested in the behavior of the derivative of the stress function and not the function itself, we convolve the sparse stress signal with the derivative of the low-pass filter. This optimization follows from the theorem that

$$deriv(f \star g) = deriv(f) \star g = f \star deriv(g)$$

where $\star$ is the convolution operator and $deriv()$ represents the derivative. The algorithm thus terminates by comparing the filtered signal directly to ε.

After empirical testing across many datasets, we arrived at the value of 50 iterations for the low-pass filter window. The termination criterion ε controls the accuracy of the layout; in our experiments we chose $\varepsilon = 1/10000$. Our linear-time termination criteria could benefit any iterative MDS algorithm relying on the convergence of stress, including SMACOF, the Chalmers algorithm [6], and others that use it as a subsystem [15, 20].

4.4 Stochastic Force on the GPU

GPU-SF is a version of the stochastic force algorithm that runs on the GPU as a series of pixel shaders, with data storage in texture memory. The first stage of GPU-SF updates the random index set of each point. Next, the set of high and low dimensional distances are computed or fetched. This information is reorganized to update the near index set. The final series of steps uses this information to calculate the proper force to apply to the point and move it accordingly. Control is then shifted back to the first step unless the termination condition is triggered.

In order to minimize GPU overhead and to work within system constraints, GPU-SF has a quite different organization of code and data from the original Chalmers algorithm. Each point in the stochastic force algorithm maintains a fixed-size cache of state information such as low-dimensional position and near-set membership.

The per-point state information is divided into vectors and tables which are stored in texture memory on 2D textures. Figure 4.5 lists the textures used to store this information and the sizes of the elements stored on the textures. Because the smallest unit of texture memory is an RGBA pixel containing four 32-bit floats, the number of pixels devoted to an element must equal the ceiling of number floats divided by four. The per-point vectors are `posHi` and `posLo`, the high- and low-

Stage	Passes	Pixels	Input Textures	Output Textures
1 Random Update	1	N_i	`perm`	`index`

2 HighD Distance Calc	$\log_4 H$	N_i	`posHi`
			`index`
			`scratch`
				`distHi`
				`scratch`

3 LowD Distance Calc	$\log_4 L$	N_i	`posLo`
			`index`
			`scratch`
				`distLo`
				`scratch`

4 Near Sort	6	N_i	`distHi`
			`distLo`
			`index`
				`distHi`
				`distLo`
				`index`

5 Force Calc	1	$N_i * L$	`index`
			`distHi`
			`distLo`
			`posLo`
			`velocity`
				`scratch`

6 Velocity Calc	1	$N_i * L$	`scratch`	`velocity`
7 Position Update	1	$N_i * L$	`velocity`	`postLo`

8 Termination Check	$\log_4 N_i$	$N_i/4^j * L$	`distHi`
			`distLo`
			`scratch`
				`scratch`

Figure 4.4: GPU-SF Algorithm Stages. The GPU-SF algorithm carries out a single layout iteration in eight stages. We list the number of rendering passes each stage requires, the number of pixels affected by each pass, the textures read as input arrays, and the textures written as output arrays. These stages repeat until the termination check succeeds.

dimensional position of the points. Each element of `posHi` has size H floats, where H is the dimensionality of the high-dimensional space. The size in floats of a `posLo` element is L, the dimensionality of the low-dimensional space, which in Glimmer is 2. The `velocity` texture keeps track of point velocities in the low-dimensional space, and also has size L floats. The tables all have 8 entries, divided

Tex. Name	Size (pixels)	Description
`posHi`	$\lceil H/4 \rceil$	high-d point coords
`posLo`	1	low-d point coords
`velocity`	1	point velocity
`index`	2	Near & Random set indices
`distHi`	2	high-d distances to pts in `index`
`distLo`	2	low-d distances to pts in `index`
`perm`	1	random number resource
`scratch`	$2\lceil H/4 \rceil$	holds temporary results

Figure 4.5: GPU-SF Algorithm Textures. The GPU-SF algorithm uses textures as storage. This table lists each texture used by the algorithm, the size in pixels of the elements dedicated to each point and a brief description of the purpose of the texture.

into two equal sections for points in the Near and Random sets. The `distHi` and `distLo` textures contain the high- and low-dimensional distance between the point in question and the items in the Near and Random sets. The `index` table contains the pointers to the items in these sets. The total size in bytes of each texture is the element size in pixels $\times$ 4 floats per pixel $\times$ 4 bytes per float $\times$ N, the number of points in the input dataset.

Two textures are used as resources in the computation. The `perm` texture contains a permutation of all indices that was precomputed on the CPU, of total size N. The $2HN$ `scratch` texture is used for intermediate storage.

Information is packed into two dimensional textures as follows. We first compute the number of pixels per element on `posHi` as $P = \lceil H/4 \rceil$. We then compute the total number of pixels required to store the data as $P \times N$. To best fit this number of pixels into a square region we compute the ceiling square root of this number, $R = \lceil \sqrt{PN} \rceil$. Unfortunately, this may produce a texture dimension R that is not divisible by the width of an element, which may force us to “wrap” an element

across a row of texels. To eliminate this phenomenon we divide R by the width of an element and take the floor of the result. The final width in pixels of our texture `posHi` is $W = P\lfloor R/P \rfloor$. To compute the height H of the texture, we just divide the total number of points by the number of points in a row and round up. When packing the remaining textures, we assume that they are W/P elements wide. This strategy makes addressing point data uniform across textures.

Figure 4.4 summarizes the overall organization of GPU-SF, showing the seven stages and which textures they update. A single iteration step is carried out in $10 + \lceil \log_4(L*H*N) \rceil$ texture rendering passes. The number of pixels, N_i, processed in each pass is also given in Figure 4.4, as an approximation of the total work involved. When GPU-SF is invoked as a subsystem of Glimmer, the memory footprint of these textures is always a function of the entire dataset size N, but the number of pixels processed in each pass changes depending on the Glimmer level.

Stage 1 The first step of GPU-SF is to update the Random section of the `index` set using `perm`. We acquire new random indices by sampling at a location in this resource determined by $P[P[x] + iteration]$ where P is the permutation array, x is the cardinality of the point, and *iteration* is the overall iteration number. This strategy is inspired by the Perlin noise algorithm [22].

Stages 2 & 3 We need to compute `distHi`, the Euclidean distances in high-dimensional space. Other distance metrics are possible with the use of distance matrices. We discuss the use of distance matrices in Glimmer in Chapter 5. We indirectly reference the points in `posHi` using the `index` set to compute the differences between these points and the current one, storing them in the `scratch` texture. We square each item in `scratch`, sum them together, and put the square root of that number into `distHi`. The fast approach to summing k values on the

GPU is a reduction shader that takes $\log_4 k$ passes, which is far cheaper than looping through the values. A similar computation produces `distLo` from `posLo`, with $\log_4 L$ passes.

Stage 4 Updating the Near set with points in Random that are closer is slightly tricky. If we simply sort by distance and pick the first 4 to be in the Near set, then an item that appears in both Near and Random would be duplicated in the Near set. Instead, we first sort by `index`, mark duplicates as having infinite high-dimensional distance, and then resort by `distHi`. We sort each of the three textures `index`, `distHi`, and `distLo` twice, using six rendering passes, combining the duplicate-marking operation with the first sorting pass.

Stage 5 To do the force calculation, we compute the vectors between the point and the 8 others in the Near/Random sets using `index` to look up their low-dimensional positions in `posLo`. We scale these vectors by the difference between `distLo` and `distHi`, then use the `velocity` texture for damping. Damping is designed to inhibit excessive particle oscillation and improve convergence. Our damping scheme computes the relative velocity vector between each vertex and its indexed vertices and subtracts it from the force vector between these vertices. We sum these damped force vectors, and save the resulting vector into the `scratch` texture.

Stages 6 & 7 We integrate the `scratch` forces into `velocity` in one pass, then integrate `velocity` and update `posLo` in another pass. Both integrations are done using Euler integration with parameters mentioned in Section 4.2.

Stage 8 The final step of the algorithm checks the termination condition. We can calculate the normalized sum of squared distance differences in `distHi` minus

`distLo` for our termination condition in $2\log_4(N)$ rendering passes using a reduction shader on `scratch`. The 4^j factor in the pixel size indicates the size reduction by a factor of four each pass, for a total of $4/3N_i * L$ pixels processed.

In the Chalmers algorithm, forces are applied symmetrically between two points, so that point i is affected not only by forces from its own Near and Random sets, but also by any forces from other points that contain i in their Near or Random sets. In our GPU-SF version, forces are applied from points in the Near/Random sets to point i, but not vice versa. We abandon this explicit symmetry because it would require a *scatter* random access write operation, which is not supported on current GPUs without slow, memory-intensive workarounds. The effect of those symmetric forces emerges implicitly as the Near sets of neighboring points gradually converge to include each other.

Chapter 5

Scalability With Distance Matrices

In our discussion thus far, we have assumed that the input for the MDS algorithm is points in high dimensional space and the desired distance metric is Euclidean. In this case, high dimensional Euclidean distances are computed on the fly as part of the work of the MDS algorithm. Although the calculation of high dimensional distances is one of the most expensive stages of many MDS algorithms, Euclidean distance is one of the most straightforward and cheap metrics compared to many of the other possibilities.

In many applications, the required distance metric may be so costly or complex to compute that doing so on the fly every time that it is used would be infeasible, for example the Earth Mover's Distance between images [27]. Moreover, in many cases there is no direct access to any formulation of points in a high dimensional space: the only available data is pairwise distances. In these cases, an MDS algorithm must accept precomputed distances as the input data, typically in a precomputed distance matrix. A naive approach would be to store the matrix in a texture on the graphics card. The texture, called `dmatrix`, would replace `posHi`. Since distance matrices are symmetric, we need only store $N^2/2$ of the entries. The GPU-SF algorithm would then copy the proper distances from `dmatrix` during stage 2 instead of computing them from high dimensional coordinates making

the number of passes for stage 2 only 1 instead of $\log_4 H$. Also, we reduce the size of the elements of the `scratch` texture from $2\lceil H/4 \rceil$ pixels to 2 because 2 pixels is the element size required for computing low-dimensional embedding distances of dimension 2 in stage 3 of the GPU-SF algorithm.

We now compare the texture memory requirements of the Euclidean case and the case where we store the distance matrix in texture memory. Figure 4.5 gives the size in RGBA pixels of individual elements for each texture the Euclidean case. Using the element size entries of the table as a guide, for a dataset of N points of dimension H, the `posHi` texture requires allocating a texture of $\lceil H/4 \rceil N$ pixels. Because each pixel is 4 floats and each float is 4 bytes, `posHi` requires $4(4\lceil H/4 \rceil N) = 16\lceil H/4 \rceil N$ bytes of texture memory. We then proceed to multiply the texture element size gathered from Figure 4.5 by 16 for each texture used by Glimmer, thus computing the total size of the each texture in bytes and store the sizes as the entires in Figure 5.1. The sum of the sizes in bytes of all the textures yields the number $(144 + 48\lceil H/4 \rceil)$, which is the total number of bytes of texture memory required by the Glimmer algorithm in the case of computing Euclidean distances on the fly from N coordinates of dimension H. To derive the maximum number of points of dimension H that Glimmer can compute on a card with 256MB of texture memory we solve for N in the equation

$$totalBytes = (bytesPerElement)N$$

$$256,000,000 = (144 + 48\lceil H/4 \rceil)N$$

and get $N = 256,000,000/(144 + 48\lceil H/4 \rceil)$. If $H = 9$ then we theoretically can fit $888,888$ points in texture memory.

Computing the total texture memory needed for the case where we store the distance matrix requires changing two values in Figure 5.1. First, we replace the $16\lceil H/4 \rceil N$ from `posHi` with $16N^2/2 = 8N^2$, the total number of bytes required

Texture Name	Total Texture Size (bytes)
`posHi`	$16\lceil H/4 \rceil N$
`posLo`	$16N$
`velocity`	$16N$
`index`	$32N$
`distHi`	$32N$
`distLo`	$32N$
`perm`	$16N$
`scratch`	$32\lceil H/4 \rceil N$
Sum Total	$(144+48\lceil H/4 \rceil)N$

Figure 5.1: GPU-SF Texture Memory Requirements. The GPU-SF algorithm uses textures as storage. This table lists each texture used by the algorithm in the case where Euclidean distances are computed on the fly and their respective sizes in bytes of texture memory.

to store $N^2/2$ floats in the `dmatrix` texture. Second, because the element size of `scratch` has changed to 2 pixels, the total size of the `scratch` texture is $32N$ bytes. Thus, the total number of bytes required by Glimmer for N points in the distance matrix case is $8N^2 + 176N$ bytes. To derive the maximum number of points that Glimmer can compute using this method on a card with 256MB of texture memory, we solve for N in the quadratic equation

$$256,000,000 = 8N^2 + 176N$$

and get a theoretical maximum of $N = 5,645$ points. Unlike in the case where distances are computed on the fly, doubling memory does not mean doubling the number of points handled. On a card with $512MB$ of texture memory, the theoretical maximum is still only $7,989$ points. Clearly, storing the entire distance matrix on the graphics card has scalibility problems.

We present a scalable solution that exploits our use of precomputed permutations for handling stochastic operations. Our methods, which we call *distance pag-*

ing and *distance feeding*, solve the texture memory scalability problem of quadratic storage. Although we have only implemented a proof of concept for the GPU version of Glimmer, the idea could benefit CPU-based MDS algorithms as well.

Distance paging draws inspiration from texture paging, used when a texture is too large to fit in texture memory but the application designer knows that only a small region of the texture is visible at one time. The designer splits the texture into chunks which are loaded from main memory only when necessary. Because we use a precomputed random number resource when updating our Random set, we know in advance the precise sequence of high-dimensional distances the program will access per iteration. We arrange the required distances in order of access, either in advance or online, and a *pager* running on the CPU loads these blocks from main memory into texture memory at every GPU-SF iteration.

We now compute the texture memory requirements for the distance paging case. We replace the `posHi` texture with the smaller `distPage` texture and distances are simply fetched instead of computed. Rather than $N^2/2$ floats, `distPage` requires storage for only $4N$ floats or $16N$ bytes. As with the case of storing the entire distance matrix, the element size of `scratch` has changed to 2 pixels and the total size of the `scratch` texture is $32N$ bytes. We again sum the individual terms to find that the texture memory required by Glimmer with distance paging is $192N$ bytes. The theoretical maximum number of points on a card with 256MB of memory is then $1,333,333$ points. Figure 5.2 summarizes the memory limitations of the Glimmer algorithm in all the previously discussed cases.

Distance feeding allows further scalability by supporting lazy evaluation. Because we use a stochastic method, many pairwise distances are not needed at all. Our use of precomputed permutations allows us to know in advance which distances will be required in the computation. A *distance feeder* is a CPU process that takes two points as an argument and returns a distance, which can then be up-

Method	Total Bytes per N	Max Points on 256MB card
On-the-fly Euclidean	$(144+48\lceil H/4\rceil)N$	888,888 ($H=9$)
Distance Matrix	$8N^2+176N$	5,645
Distance Paging	$192N$	1,333,333

Figure 5.2: Theoretical memory limitations of the GPU-SF algorithm on a 256MB card when using different techniques to compute inter-point distances. We introduce Distance Paging to solve the quadratic storage requirements with using distance matrices.

loaded to the textures on the graphics card. With distance feeding, there is no need to precompute the entire distance matrix. Often, the time required to precompute distance matrices is the most significant scalability bottleneck for MDS applications. With precomputation, the layout runs slower, but less work is done in total because unused pairwise distances never need be computed. Using distance paging and distance feeding, Glimmer can handle distance matrices far larger than the limits of texture memory on the graphics card.

5.1 Paging and Feeding Performance

We compare the performance of our distance matrix pager and feeder schemes with an example from graph drawing. We use a graph named `bcspwr10` from the Matrix Market collection [3] of over 5,300 nodes, where the $O(N^2)$ size of the distance matrix is too large to practically fit into available texture memory on our 256MB card but slightly below the theoretical limit. MDS can be used to lay out the graph because stress is closely related to the Kamada-Kawaii force-directed placement energy. Graphs are a good example of datasets where precomputing the full distance matrix is expensive: solving the all pairs shortest path problem is $O(N^3)$, taking 623 seconds. When we use the pager to work with this distance

matrix, there is no slowdown in the performance of Glimmer; in fact, loading the texture is cheaper than computing the high-dimensional distances. Computing the layout with paging took only 5.5 seconds. In contrast, the feeder-based layout took 172 seconds which is less than the total 628.5 seconds required by paging. Figure 5.3 shows the graph layout produced by Glimmer using the distance matrix computed from the `bcspwr10` graph.

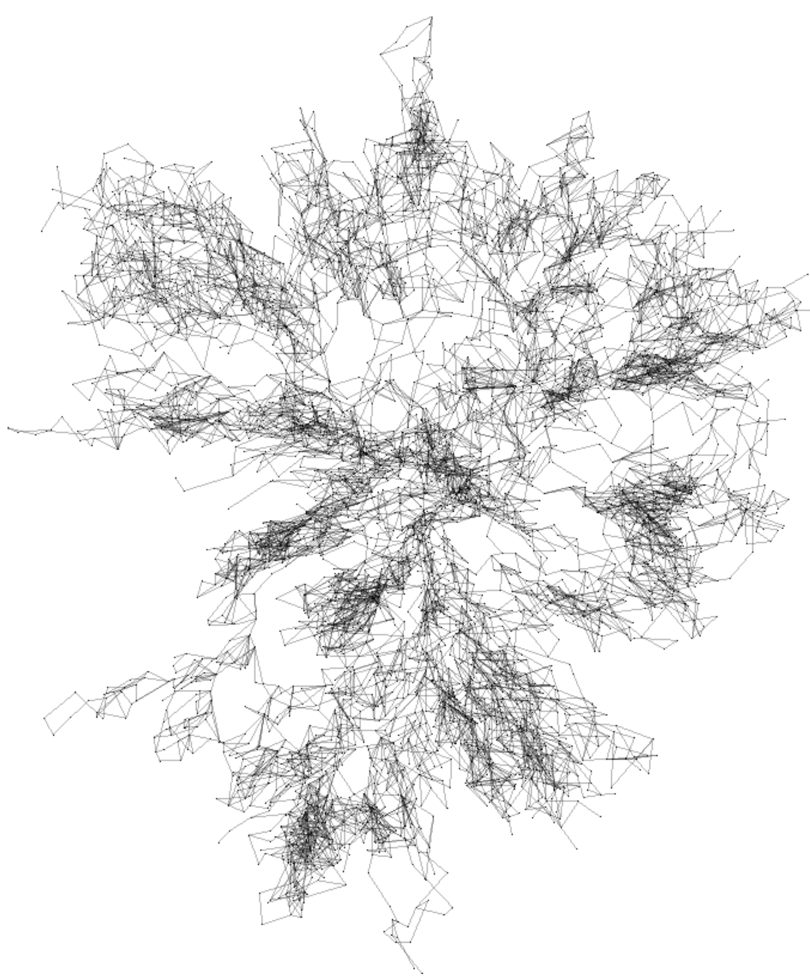


Figure 5.3: The `bcspwr10` graph with layout by Glimmer using distances precomputed by graph search. Rather than compute the entire $O(N^2)$ distance matrix, we need only compute enough distances for the stress function to converge. Because we know this sequence in advance, we compute distances using a distance feeder and drastically reducing compute time over the distance paging approach.

Chapter 6

Results and Discussion

We compare our approaches to previous work in terms of asymptotic complexity, speed, the quantitative metric of normalized stress, and the qualitative visual analysis of layouts.

The MDS algorithms that we chose to compare against are a mix of foundational algorithms and competitive exemplars of the major approaches. The foundational algorithms are a MATLAB version of Classic MDS[1], our MATLAB implementation of SMACOF, and a Java implementation of Chalmers[2]. These three foundational approaches are known not to be speed-competitive, so measures of stress and layout quality are more interesting than the time performance. We terminate SMACOF when the change in the normalized stress function falls below $1/10000$, the same criterion used for GPU-SF and Glimmer.

We use a Java implementation of PivotMDS[3] [4] as the classical scaling approach, using the default of 50 landmarks except where noted. We use Jourdan's $O(N \log N)$ Hybrid [15] as the fastest force-directed approach[2]. Bronstein's Multigrid MDS [5] is not publicly available, but we know that it is not speed-competitive with Hybrid or PivotMDS from the timings given in the paper.

While Classic and PivotMDS are designed to minimize strain rather than stress, we report on the success of their layout using the stress metric. We do so for

[1] `cobweb.ecn.purdue.edu/~malcolm/interval/2000-025`

[2] `www.lirmm.fr/~fjourdan/Projets/MDS/MDSAPI.html`

[3] software courtesy of Christian Pich

consistency, and also because we consider stress to be the most suitable quantitative metric that captures our qualitative judgement about layout quality for visualization purposes. In other MDS applications outside of information visualization, where direct visual inspection of the layout is not required, stress may be a less suitable metric.

6.1 Complexity

The cost of one GPU-SF iteration is proportional to the number of rendering passes multiplied by the number of pixels affected at each pass. Multiplying these values from Figure 4.4 yields a per-iteration cost of $(7 + \log_4 H + \log_4 L + 5.33\, L) * N_i = O(N_i\ \log_4 H)$. The cost of a full GPU-SF invocation is $O(C\, N_i\ \log_4 H)$ where C is the number of iterations performed before the system converges. As we discuss in Section 4.3, C is not necessarily N. We have observed that it varies depending on dataset characteristics, ranging from constant to $O(N)$.

The number of points N_i supplied to GPU-SF at each Glimmer level using decimation factor F ranges from 1000 up to N, where $N_{i-1} = N_i/F$, and the number of levels is $\log_F N$. The total number N_t of points processed across all Glimmer levels is bounded above by $(F/(F-1)) * N$, the infinite sum of $(1/F^i) * N$. The cost of each Glimmer level is two invocations of GPU-SF, one for interpolation and one for relaxation. The restriction stage of Glimmer does not incur any extra costs that we need to consider in our asymptotic analysis, because the sampling is built into the algorithm. Thus, the total complexity of Glimmer on the CPU is $O(C\, N\ \log_4 H)$.

We now discuss the effects of GPU parallelism. Asymptotic analysis of parallel programs is difficult to present concisely. To oversimplify, a GPU with a SIMD size of p, where p ranges from 16 to 1024 on current cards, speeds up compu-

tation up to a factor of p. Since we carefully designed our shaders and render passes to avoid conditionals and loops, our actual speedup is close to this theoretical maximum. The computational complexity of Glimmer on the GPU is thus approximately $O(C\,N\ \log_4 H\ /\ p)$.

In contrast, the complexity of Hybrid is $O(N \log N)$, Chalmers is $O(N^2)$, SMACOF is $O(N^2)$, and Classic MDS is $O(N^3)$. Pivot MDS has a complexity of $O(k^3 + k^2 n + kn)$, and for a fixed number k of landmarks and a large number of points N it is typically considered linear.

6.2 Performance Comparison

We compare Glimmer and GPU-SF to each other and to several previous MDS algorithms, across a range of real and synthetic datasets. All benchmarks are run on a Pentium 4 3.2 GHz CPU with 1.5 GB of memory and an nVidia 7800GS graphics card with 256MB of texture memory, except for the 8800GTX timings for Glimmer and GPU-SF which are run on an Intel Core 2 QX6700 2.66 GHz CPU with 2 GB of memory and an nVidia 8800GTX graphics card with 768MB of texture memory. No timings in this thesis include file loading time or rendering time for any algorithm. However, in the accompanying video, the timings for GPU-SF and Glimmer do include render time for interactive display. All layout times below include computing high-dimensional distances on the fly. Although some algorithms use an approximation of the stress function while finding the embedding, all stress figures reported below use the full normalized metric given in Equation (1.1).

6.2.1 Datasets

We use a mix of of synthetic and real-world benchmark datasets. The small `cancer` dataset from the UCI ML Repository[4] has 683 points in 9 dimensions. The ground truth for the two major clusters of malignant versus benign tumors is shown with color coding of orange and blue, respectively. The `shuttle_small` dataset, also from UCI, has 14,500 points in 9 dimensions, with `shuttle_big` having the same structure but 43,500 points. The ground truth for the seven clusters is shown with color coding. We generated the well-known synthetic `swissroll` benchmark, a 2D nonlinear manifold of 1089 points embedded in 3 dimensions. We generated a set of synthetic datasets of smoothly varying cardinality, where a 2D grid is embedded in 8 dimensions. We also tested the effects of adding noise to those grids, specifically 1% noise in a third dimension. The `docs` dataset is a real-world example of a large collection of unordered document metadata used to study document clustering algorithms[5] [16]. These collections can be represented as highly sparse matrices where a row represents a document and a column represents a text feature. In Glimmer and GPU-SF, we store this matrix compactly in texture memory as a value-index pair. There are 28,433 points in 28,374 dimensions, with the ground truth of six clusters again shown by color coding.

6.2.2 Layout Quality

Figure 6.1 shows the visual quality, normalized stress, and timing of Glimmer, GPU-SF, Hybrid, and PivotMDS layouts on four datasets with known structure. In the case of `grid`, the correct shape is known. In the other three cases, the correct partitions of the points into clusters are available with these benchmark datasets,

[4] `www.ics.uci.edu/~mlearn/MLSummary.html`

[5] Data courtesy of Aaron Krowne.

so the extent to which the color coding matches the spatial grouping created by an algorithm is a measure of its accuracy.

Qualitatively, with `cancer` the three algorithms Glimmer, GPU-SF, and PivotMDS indicate these two color-coded groups clearly with spatial position. Quantitatively, the stress of Glimmer and GPU-SF is an order of magnitude lower than PivotMDS. Hybrid does separate the two groups, but produces misleading subclusters in the orange group.

With `shuttle_big`, GPU-SF fails to separate the clusters and has clearly terminated prematurely. Hybrid produces a somewhat more readable layout separating the red cluster from the other two, but is very slow. Glimmer and PivotMDS both produce useful and qualitatively comparable layouts separating the clusters. The PivotMDS layout is twice as fast, but has noticeable occlusion and much higher stress than the Glimmer layout.

The 10,000-point `grid` is accurately embedded by Glimmer, GPU-SF, and PivotMDS in comparable times. Hybrid is much slower but nevertheless terminated too soon, suffering from very noticable qualitative distortion and with a much higher quantitative stress metric compared to the other layouts.

The Glimmer layout of the `docs` dataset is qualitatively better than the other three. It shows several spatially distinguishable clusters, color coded by blue, red, orange, and green. The blue cluster is split into three parts. It took nearly 16 seconds with normalized stress of 0.271. GPU-SF is three times faster but terminated prematurely with a poor layout: although the points are grouped into clusters, the clusters all occlude one another. Hybrid also suffers from cluster occlusion. The stress is nearly twice as high, and the spatial embedding does not clearly separate any of the given clusters. PivotMDS is very fast, but almost completely fails to show the dataset structure. The normalized stress value of 0.928 is extremely high.

Figure 6.2 illustrates the very noticeable difference in visual quality of a `grid`

layout between a fast but inaccurate GPU-SF layout that failed to converge accurately because of premature termination at a local minimum, and a correct layout from Glimmer. GPU-SF can get caught in local minima where the low-dimensional manifold is twisted, and will either take more time to slowly unfold or stop too soon before the accurate solution is reached because the termination condition is fulfilled. Glimmer combats such situations by unfolding these twists at the highest tiers in the multilevel hierarchy. Twists in layouts of small point sets are higher energy states relative to the overall energy of the dataset and more likely to be properly resolved before the termination condition is met. This strategy does not make Glimmer immune to such states, but helps to reduce their probability. The Glimmer multilevel approach succeeds more often at finding the global minimum configuration.

6.2.3 Speed and Stress

We use the synthetic grid dataset and parameterize random permutations of `shuttle` and `docs` to compare algorithm speed and accuracy across a large interval of dataset cardinalities.

With respect to speed, Figures 6.3(a), 6.4(a), and 6.5(a) show that the algorithms fall into two main categories. As expected, the polynomial-time foundational Classic, SMACOF, and Chalmers algorithms do not scale past thousands of points, taking minutes or hours to compute such layouts. The remaining group of algorithms scale to hundreds of thousands of points in under a minute, as shown in more detail in Figures 6.3(b), 6.4(b), and 6.5(b). Hybrid is the slowest. The timing relationship of PivotMDS versus Glimmer and GPU-SF depends on the generation of the graphics card. PivotMDS (brown) is consistently faster than Glimmer and GPU-SF on the older 7900GS card (violet and red respectively), but slower than

Glimmer and GPU-SF on the newer 8800GTX card (grey and plum respectively). The exception is the sparse `docs`, where PivotMDS is fastest but yields incorrect results.

The stress graphs of Figures 6.3(c), 6.4(c), and 6.5(c), with log-scale vertical axes, are a critical part of the story, showing where there is a speed-accuracy trade-off. For each dataset, we draw a dashed black line as a rough indication of the stress threshold where visual quality is affected based on our empirical inspection of the layouts. We characterize an algorithm as outperformed by a competitor when its accuracy falls under this line, even if the competitor is faster.

For `grid`, in Figure 6.3(c), classical scaling algorithms like Classic (pink) and PivotMDS (brown) produce perfect, zero-stress layouts. SMACOF (blue) and Glimmer (violet) also produce excellent layouts with stress less than 0.01. The dashed black line shows that for `grid`, layouts with stress higher than approximately 0.009 have perceivable inaccuracies. GPU-SF (red) produces accurate low-stress layouts until around 15,000 points. At cardinalities beyond that, it terminates early but the layout is inaccurate, so GPU-SF is outperformed by Glimmer. Finally, Hybrid (green) and Chalmers (orange) are both inaccurate, producing comparable layouts with noticable distortion and stress an order of magnitude greater than the competitive algorithms.

For `shuttle`, in Figure 6.4(c), the location of the dashed black line above the measured stress for all algorithms indicates that all yielded acceptably accurate results, except for the special case of GPU-SF. We show a second red dashed line to indicate that GPU-SF layouts above it are qualitatively inaccurate. The stress in these failure cases is numerically lower than the acceptable PivotMDS and Classic MDS layouts, showing that there is not always a direct correlation between stress and visual quality, especially when comparing the results of differing optimization strategies. At high cardinalities, GPU-SF is very likely to terminate too

soon and produce unreadable layouts where clusters are not spatially separated, as in Figure 6.1. We thus argue that Glimmer outperforms GPU-SF in this case as well. The approximate PivotMDS algorithm (brown) yields higher-stress layouts than the foundational Classic approach (pink) but qualitatively they are the same. SMACOF (blue) produces the lowest-stress layouts, followed by Glimmer (violet). Hybrid (green) falls in the gap between the classical scaling methods and the other distance scaling techniques.

In `docs`, shown in Figure 6.5(c), we see much higher stress levels, with 0.1 to 1.0 as the axis range, because the intrinsic dimensionality of the data is much higher than 2. We see considerable separation between the accuracy of the algorithms. Although the magnitude of separation between the GPU-SF and Glimmer stress measurements may not seem large, the visual difference is indeed very perceptible, as can be seen in Figure 6.1. Only Glimmer is underneath the dashed black line showing qualitatively correct threshold. It produces layouts of acceptable quality quite quickly, taking 2 seconds on the 8800 card and 12 seconds on the 7900 card for the 28,433 point dataset. SMACOF also provides acceptable quality layouts of lower stress than Glimmer, but would require several hours to compute them. Both PivotMDS and Classic produce very inaccurate layouts, as shown by their high stress values.

Figure 6.6 further illustrates the relationship of speed and stress, showing log-log scatterplots of the timing and stress of the seven algorithms on three small datasets: `cancer`, `swissroll`, and a grid of 1000 points with 1 percent noise. Each algorithm is represented by a single colored dot, except for PivotMDS where we show a brown line connecting three runs of 50, 100 and 300 pivots. Dots closer to the lower left corner represent algorithms outperforming those further towards the upper right.

The plots show an almost linear relationship between the stress and timing of

PivotMDS (brown), Hybrid (green) Classic (pink), and SMACOF (blue), indicating a simple speed-accuracy tradeoff for these algorithms. Chalmers (orange) is an obvious outlier in the underperforming upper right quadrant, with slow times and high stress. Glimmer (violet) and GPU-SF (red) are outliers in the overperforming lower left quadrant, with both fast times and low stress. Our two algorithms break the pattern by achieving higher-speed layouts without an accuracy penalty. We can also see that GPU-SF is not simply faster than the Chalmers algorithm that inspired it; thanks to the more robust termination condition, it achieves lower stress.

SMACOF produces the lowest stress layouts, but is unacceptably slow. The Hybrid method substantially reduces the time required to produce a layout, but the resulting layout configuration can contain substantial artifacts. GPU-SF harnesses the GPU to converge rapidly but often terminates prematurely for large datasets. Glimmer uses a multilevel approach to avoid premature termination and computes a low-stress configuration in seconds by exploiting the GPU, scaling to datasets beyond 100,000 points.

6.2.4 Summary

Glimmer is much faster than the foundational SMACOF, Classic, and Chalmers algorithms. For datasets of 8 dimensions and 50,000 points, this improvement is on the order of 40,000X. Glimmer reliably achieves lower stress than Chalmers, but is higher stress than SMACOF, and the victor for stress between Classic and Glimmer depends on the dataset.

Glimmer produces results with lower stress and better visual quality than Hybrid. Glimmer is also faster, with the exception of `docs` on the 7900GS GPU, where Hybrid is faster but yields very uninformative layouts.

GPU-SF and Glimmer are very close in speed. In all cases where GPU-SF

is much faster than Glimmer, it has terminated too soon and yields uninformative layouts. The stress graphs show that early termination by GPU-SF is correlated with larger dataset cardinality. Glimmer often produces results with lower stress and better visual quality.

Finally, Glimmer is faster than PivotMDS when using the latest 8800GTX card, but slower when using the older 7900 card. These two specific algorithms are examples of very different approaches to speeding up the MDS computation. Given the current hardware trend of GPU speeds increasing more quickly than CPU speeds, algorithms such as Glimmer that exploit GPU parallelism may have an increasing speed advantage in the future.

The exception is again `docs`, where PivotMDS is faster than Glimmer even when running on the 8800GTX. However, it produces visually uninformative results. For all other datasets that we tested, the two algorithms produce results of comparable visual quality. PivotMDS and Glimmer are exemplars of two very different approaches to MDS, and we now discuss the tradeoffs between those approaches more generally.

6.3 Comparing Distance To Classical Scaling

It is interesting to consider the advantages and disadvantages of distance scaling approaches that use stress such as Glimmer, GPU-SF, Chalmers, Hybrid, and SMACOF versus classical scaling approaches that use strain such as PivotMDS, Landmark MDS, and Classic.

In distance scaling, individual distances are computed in an embedding space of specified dimension L. In contrast, classical scaling does not specifically parameterize embedding dimension. Layout in L dimensions occurs by simply choosing the first L eigenvectors. If the intrinsic dimensionality of the layout is k, then k

eigenvectors will contain layout information. By intrinsic dimensionality, we mean the number of dimensions needed to achieve a layout where strain is zero[6]. When k is greater than the desired embedding dimension ($L = 2$ in this thesis), classical scaling implicitly uses more degrees of freedom in minimizing its objective function than distance scaling. The resulting layout may occlude points, clusters or other features in lower dimensions.

We illustrate this phenomenon by embedding the endpoints of a regular simplex. A simplex is a geometric object whose endpoints are all a distance of unit length from each other. For example, a line segment is a regular 1-simplex and an equilateral triangle is a regular 2-simplex. Figure 6.7 shows the results of embedding a regular 100-simplex in two dimensions using classical scaling and distance scaling. While there is no way to embed such a high dimensional object without loss of some information, distance scaling constructs a layout without point occlusion roughly the diameter of the simplex while classical scaling places most of the points in a region much smaller than the simplex diameter.

The so-called curse of dimensionality [2] states that the majority of points sampled in high dimensional space will be equidistant. That is, the volume of space exponentially increases as a function of dimension and the likelihood that two points are close to each other becomes less and less. Thus data sampled in very high dimensions is more likely to exhibit simplicial structure.

When the intrinsic dimensionality of the dataset is less or equal than the embedding dimension, then classical scaling methods are likely to work very well. Even if the dimensionality is greater, the greater likelihood of intra-cluster occlusion may sometimes be advantageous, because clusters may be more easily distin-

[6]Distance gathering techniques like Isomap [28] may find even lower intrinsic dimensionality layouts using more complex distance metrics than the Euclidean one we discuss. Our arguments still apply in this case.

guished from each other. However, we argue that for sparse, very high dimensional datasets such as `docs`, distance scaling is probably a better choice than classical scaling. The PivotMDS layout of the `docs` dataset shown in Figure 6.1, produced by minimization of the strain objective, demonstrates that no two-dimensional orthogonal basis in the text-feature space can be constructed to visually separate the relevant clusters. We consider the smearing of the ground-truth color coding into disparate spatial regions to be evidence of the disadvantages of minimizing strain when dealing with sparse datasets. To confirm this analysis, we tested the PivotMDS algorithm on this dataset using 5000 landmarks, and the visual appearance was not improved. We argue that algorithms based on distance scaling and random search such as stochastic force, are more suited to visualizing these datasets. Glimmer is the first such algorithm that can scale to sparse datasets of this size and produce useful results in a matter of a dozen seconds.

6.4 GPU Speedup

Figure 6.3(a) shows the speedup of GPU-SF over the CPU-based Chalmers algorithm. We now provide quantitative measurements of the GPU speedup for Glimmer. Figure 6.8 shows the running times in milliseconds for the Glimmer algorithm on two different GPUs versus a completely CPU-based proof-of-concept implementation. Timings are shown for the synthetic grid dataset over several sample sizes. Each implementation performs roughly the same number of computations, allowing us to very directly gauge the magnitude of the GPU speedup. Figure 6.9 is a graph of the CPU timing values in Figure 6.8 divided by the GPU timings for each cardinality. The speedup factors converge to a constant value for each GPU, approximately 6 times faster on the 7900GS and approximately 30 times faster on the 8800GTX. The two speedup factors do not conform to the precise number of

processors per GPU which are 24 and 128 respectively. This discrepancy is due to architectural differences between the individual CPU and GPU processors.

The GPU speedup comes with startup and overhead costs. These include shader compilation, shader optimization, and data initialization-upload/download. Figure 6.10 shows the costs in milliseconds for each of these steps on a variety of sample sizes of the grid dataset. The GPU-SF and Glimmer layout times do include the overhead of uploading data from the CPU to the GPU. Shader compilation/optimization is a step required only once for any number of subsequent layouts and thus is not included in any performance runtimes. For both GPU-SF and Glimmer, shader compilation and initialization requires 4 seconds of dataset-independent startup overhead when the program begins, which is not included in any of our timings below. Similarly, we do not count startup times for MATLAB itself or the Java VM for the other algorithms.

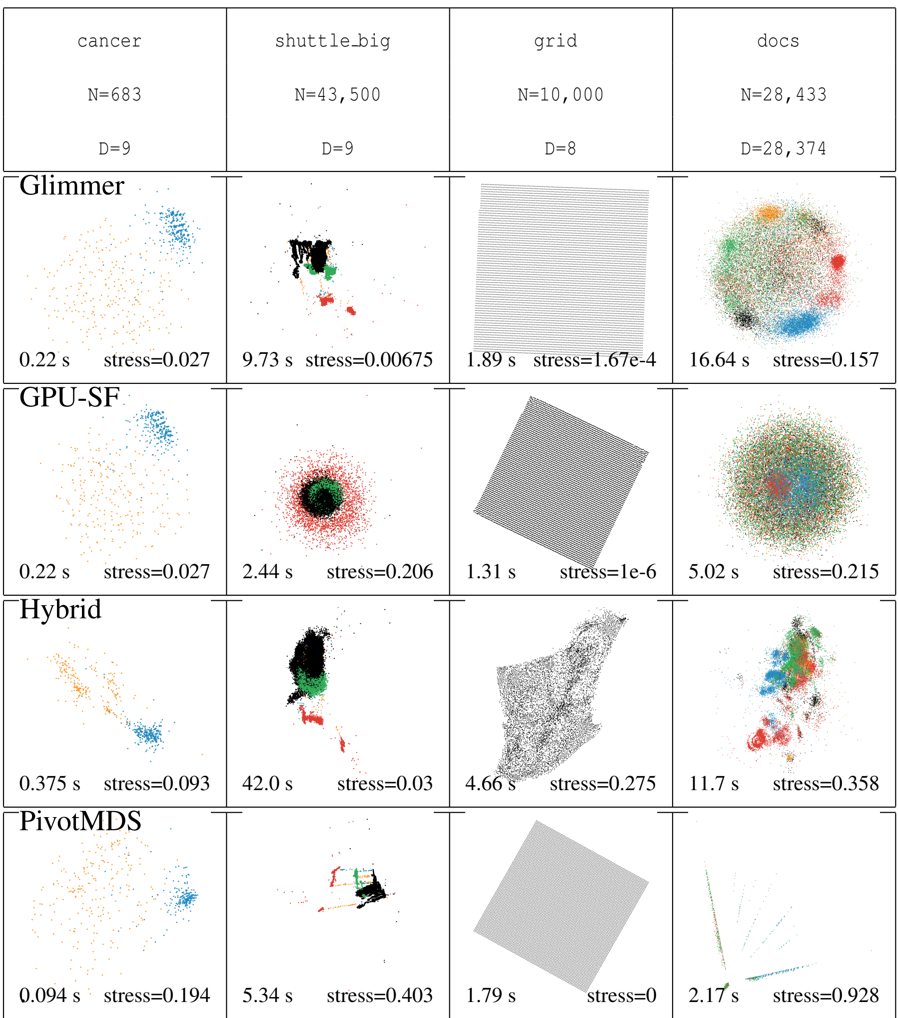


Figure 6.1: MDS layouts showing visual quality, time, and stress for the Glimmer, GPU-SF, Hybrid, and PivotMDS algorithms. Dataset name, number of nodes (N), and number of dimensions (D) appear above each column. Time in seconds appears at the bottom left of each entry, with normalized stress on the bottom right.

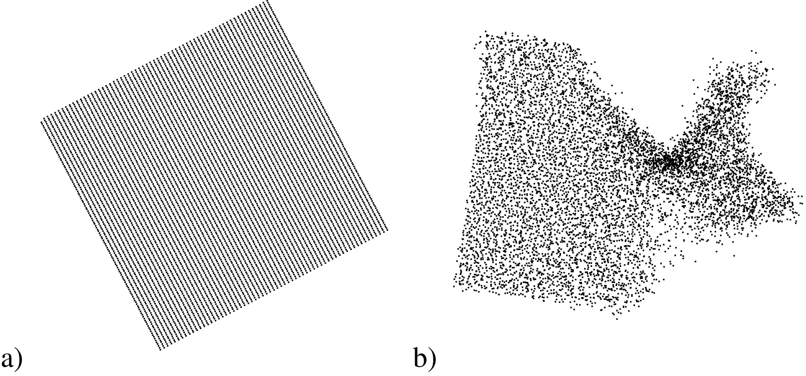


Figure 6.2: Visual quality differences between a) Glimmer and b) GPU-SF for `grid` instance with cardinality 8000. Glimmer exhibits more stable convergence behavior than GPU-SF, which more frequently yields a twisted layout when it is caught in a local minimum and terminates with a high stress value. This layout corresponds to the spike at 8000 for GPU-SF in figure 6.3(a).

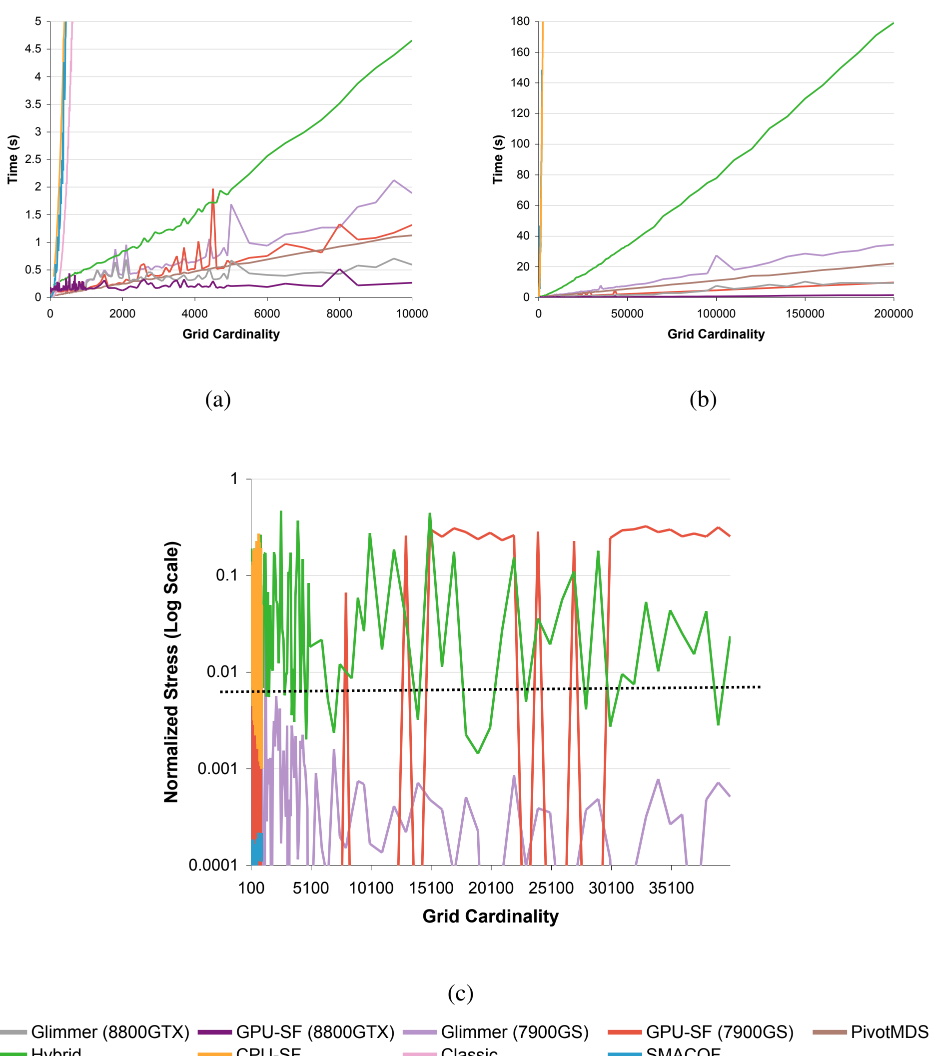


Figure 6.3: `grid` Stress and Timing Graphs. **(a)** Seconds versus cardinality for `grid` up to 10000 points. Chalmers (orange), SMACOF (blue), and Classic (pink) are all orders of magnitude slower than faster approximation algorithms. While GPU-SF appears to compute the fastest layouts, its layout quality is much lower than Glimmer in most cases due to local minima. **(b)** Seconds versus cardinality for `grid` up to 200000 points. Chalmers, SMACOF, and Classic are not visible at this scale. Again GPU-SF finishes fastest due to local minima, however Glimmer on the 8800GTX card is second fastest with higher quality layouts. **(c)** Normalized stress versus cardinality for `grid` up to 40000 points. Both Chalmers, Hybrid, and GPU-SF are prone to distorted layouts registering above the heuristic, dashed-black visible-distortion-line. Both PivotMDS and Classic are not visible due to producing zero stress layouts.

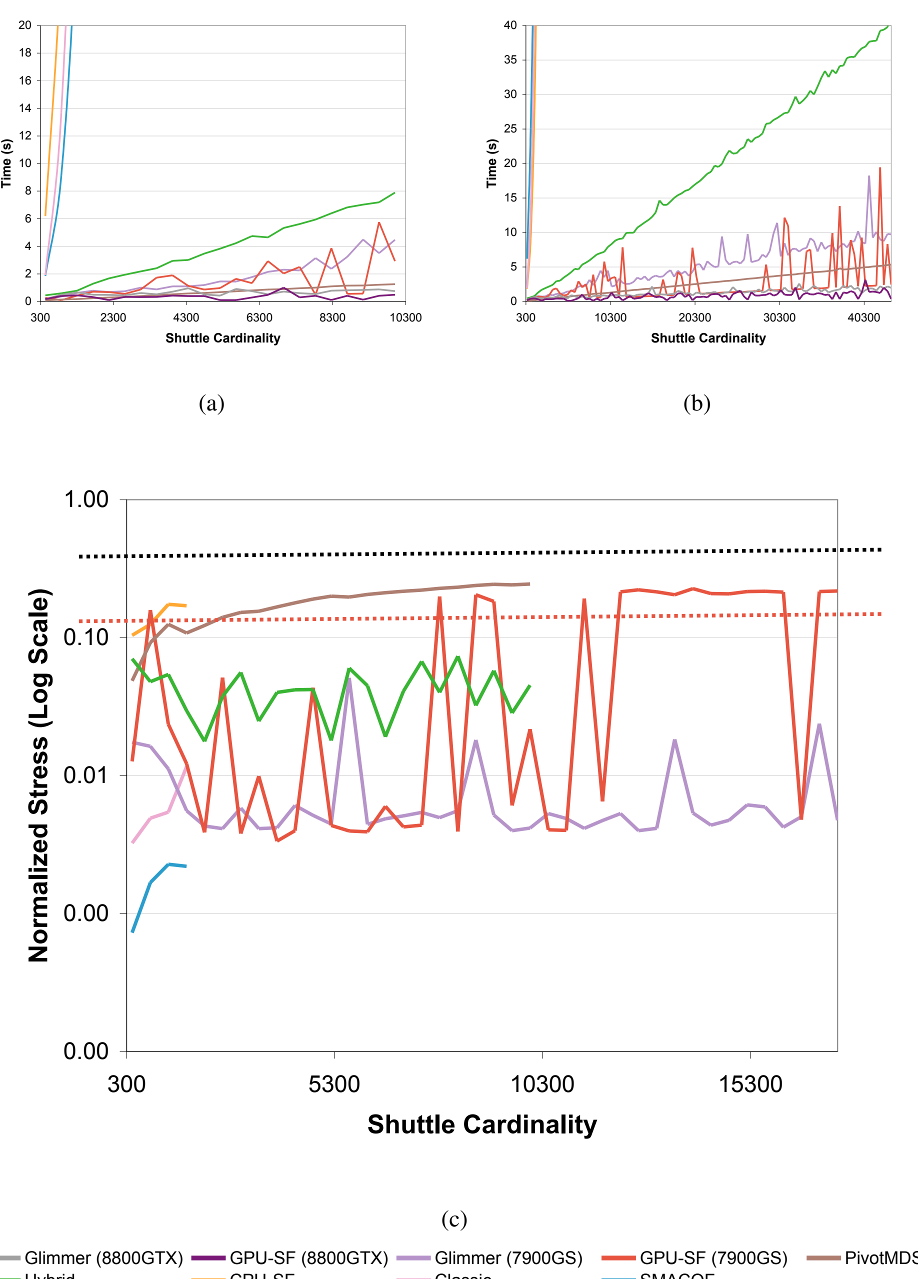


Figure 6.4: `shuttle` Stress and Timing Graphs. **(a)** Seconds versus cardinality for `shuttle` up to approximately 10000 points. **(b)** Seconds versus cardinality for `shuttle` up to 45000 points. **(c)** Normalized stress versus cardinality for `shuttle` up to approximately 10000 points. We include a dashed black line to indicate that all algorithms produce acceptable results, except GPU-SF. To show where GPU-SF terminates with low-quality results at a local minimum, we use a red dashed line.

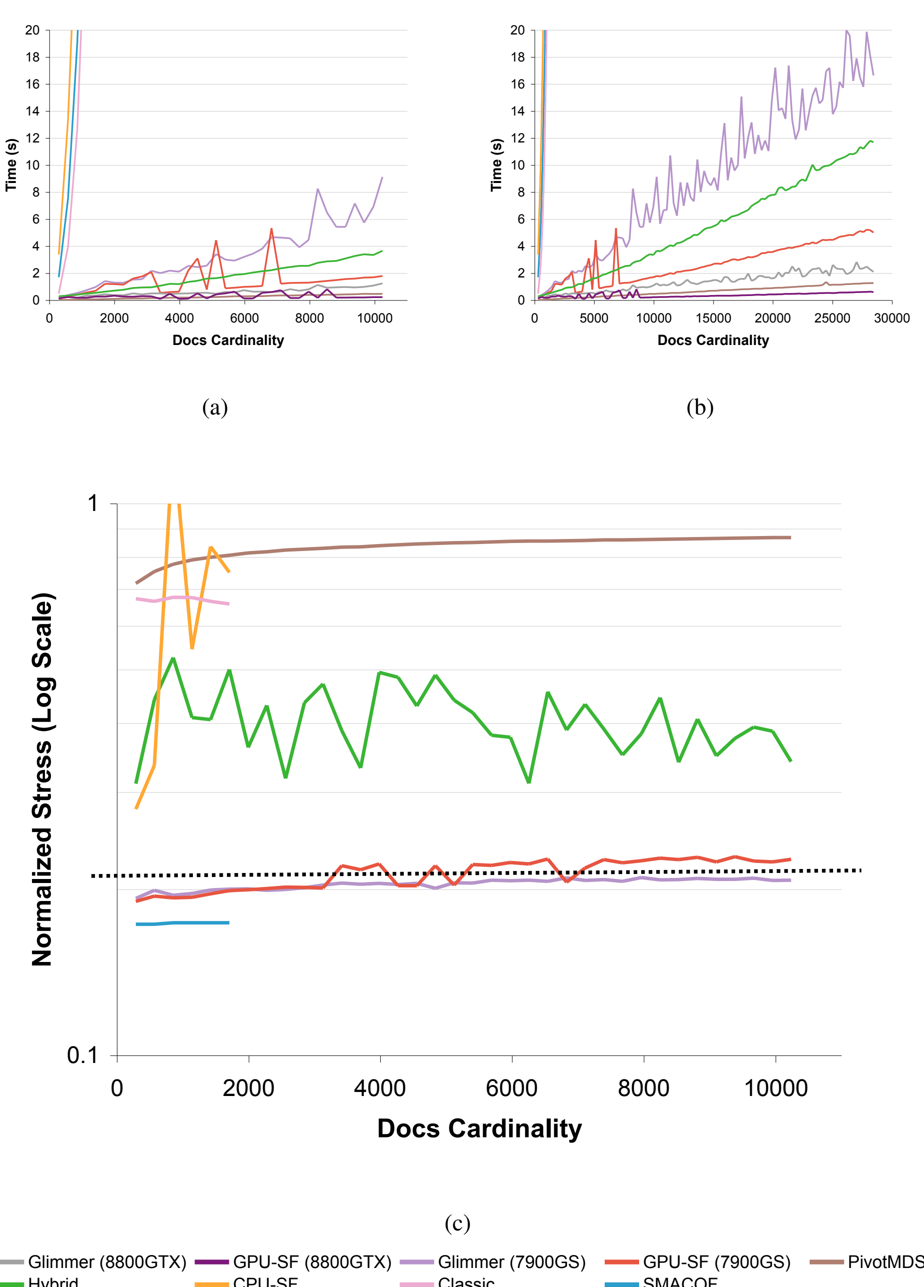


Figure 6.5: `docs` Stress and Timing Graphs. **(a)** Seconds versus cardinality for `docs` up to approximately 10000 points. **(b)** Seconds versus cardinality for `docs` up to 28433 points. **(c)** Normalized stress versus cardinality for `docs` up to approximately 10000 points.

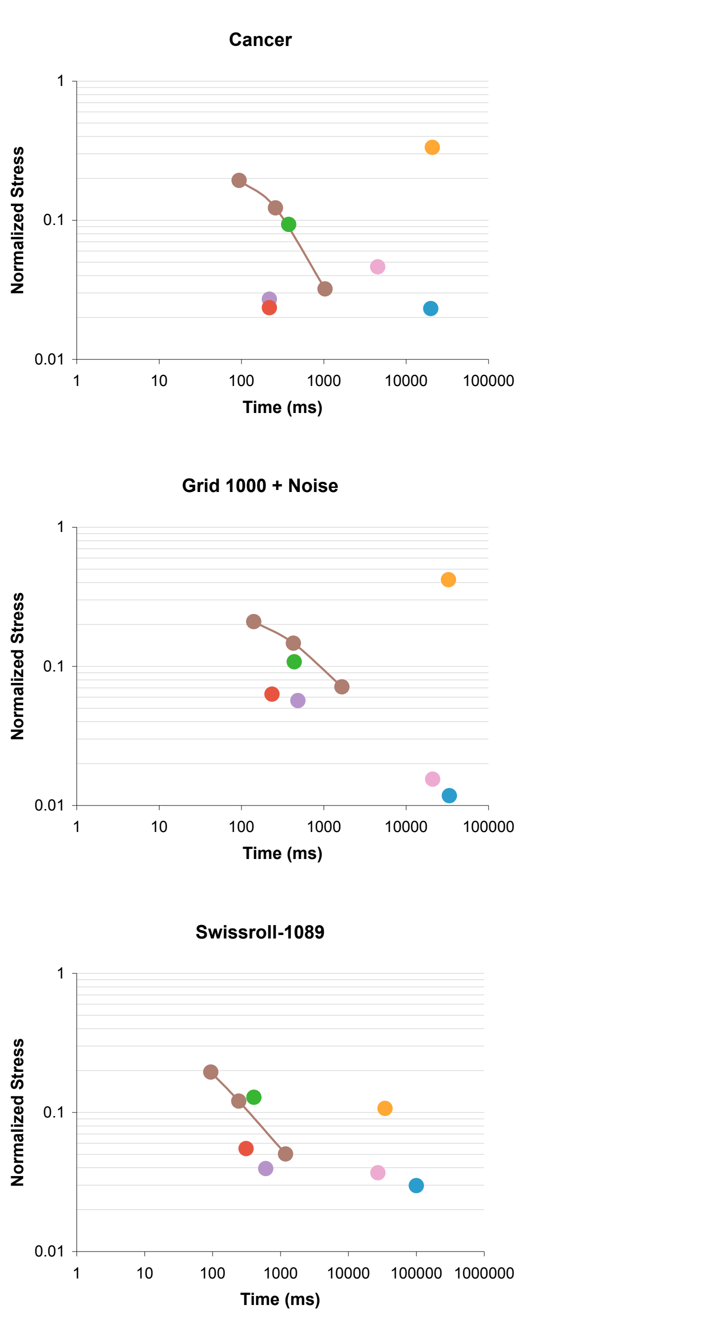


Figure 6.6: Log-log scatterplots of stress versus time for the seven measured MDS algorithms on `cancer`, `swissroll`, and `grid1knoise` datasets of increasing cardinality. These graphs illustrate a stress-time tradeoff with distinct outliers Chalmers (orange) on the slower side of the tradeoff and GPU-SF (red) and Glimmer (violet) on the side of the tradeoff with lower stress in shorter time. We include a dashed black line to indicate the stress level at which cluster separation is noticable.

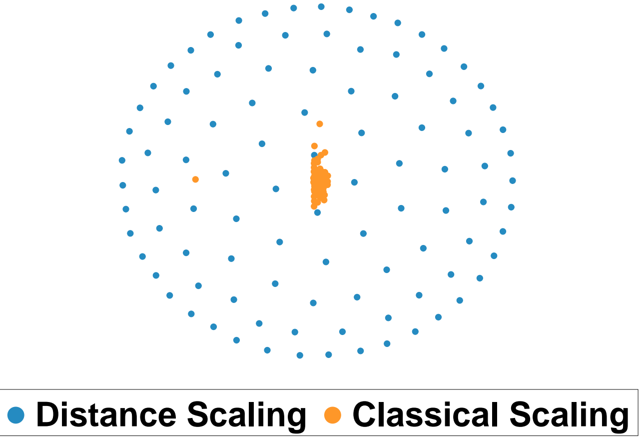


Figure 6.7: Layouts of a regular 100-simplex produced by distance scaling and classical scaling. Both methods distort the simplex. Distance scaling produces less point occlusion and better preserves the diameter of the simplex.

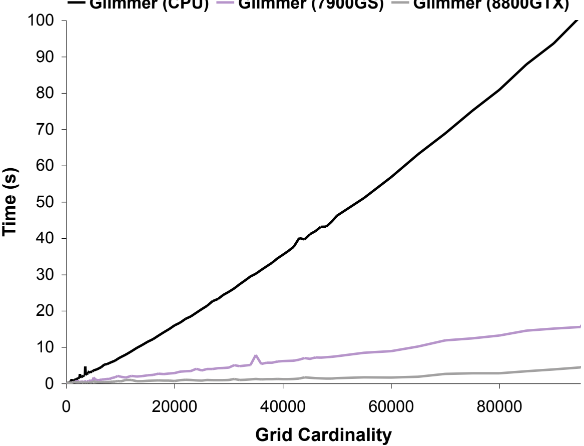


Figure 6.8: GPU vs. CPU speed, in seconds. We show Glimmer on two different GPUs compared to a purely CPU-based implementation. The dataset is the synthetic grid across a range of cardinalities. Glimmer was carefully designed to fully exploit the possibilities of GPU parallelism in modern hardware, achieving an order of magnitude speedup at 100,000 points.

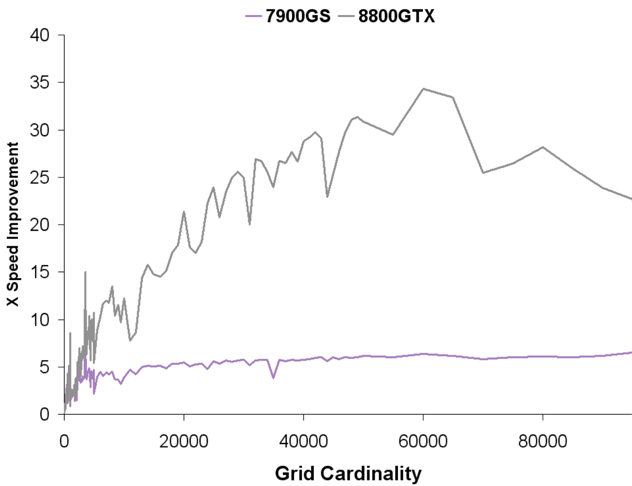


Figure 6.9: Glimmer GPU Speedup. By dividing the time required to complete a layout using Glimmer on the CPU by the time required on a GPU, we can directly calculate the speedup provided by the two graphics cards. The graph shows that both cards converge to an approximately constant speedup factor.

	startup (ms)		overhead (ms)	
Size	Shdr. Comp	Shdr. Opt	Init+Upload	Dload
20	3922	812	16	0
200	3891	797	31	0
2000	3875	797	15	0
20000	3859	813	47	0
200000	3875	813	312	16

Figure 6.10: Startup costs and texture overhead, in milliseconds. Shader compilation and optimization are single-step startup costs that can be amortized over many layouts. Texture initialization and data upload and download are costs incurred by an individual dataset, but this overhead is very small compared to overall runtime.

Chapter 7

Conclusion and Future Work

Glimmer and GPU-SF provide dramatic speedups compared to previous distance scaling approximation algorithms by exploiting GPU parallelism at every stage of their architectures. Our new termination criterion for GPU-SF detects convergence cheaply by approximating the normalized stress function. GPU-SF is roughly as fast as Glimmer, but is more prone to getting caught in local minima, especially on large datasets. The multilevel architecture of Glimmer is more likely to converge to a lower stress embedding. Glimmer avoids the speed-accuracy tradeoff of previous distance scaling approximation algorithms, as we have shown on a mix of synthetic and real-world datasets. With distance scaling and distance paging, we avoid any disadvantages that GPU texture memory restrictions may impose over CPU-based MDS algorithms. It is competitive with previous classical scaling approximations in speed, and yields higher quality results for sparse, high-dimensional datasets.

Glimmer should be straightforward to generalize from the current $L = 2$ implementation to handling target spaces of any dimension. The force calculation pass at stage 5 of GPU-SF might be the main bottleneck, possibly taking more passes as dimensionality increases. During the force calculation in step 5 of GPU-SF, normalized velocity vectors are computed for damping purposes. We currently use optimized shader instructions for calculating normalization factors. As the number of embedding dimensions exceeds 3, these instructions would no longer be applicable. A general multipass scheme similar to step 2 of the algorithm must be used

to calculate velocity vector length which would increase the number of passes for step 5 from 1 to $\log_4 L$.

We adapted the GPU-SF algorithm to perform force-directed graph placement and called the algorithm GLUG [14]. GLUG makes three key changes to GPU-SF. First, rather than fixed-size *Near* and *Random* index sets with dynamic entries, we replaced these sets with two different index sets with fixed entires called *Near* and *Landmark*. Users can control the size of the Near and Landmark sets to respectively control the local and global fidelity of the graph layout. Second, we initialize the contents of `distHi` as a CPU-based preprocess. Third, we use the velocity-based termination condition rather than the sparse normalized stress condition from Chapter 4.3. While the velocity termination is problematic, the sparse normalized stress condition works well only in the context of a multilevel approach and GLUG is a single-level algorithm. GLUG computes layouts with superior speed and comparable quality to other state-of-the-art, graph placement algorthms like FM^3, but suffers from suboptimal results due to local minima and the need to tweak velocity termination parameters. It would be interesting future work to adapt the multilevel Glimmer approach for optimized force-directed graph placement, to exploit the more robust sparse stress termination criterion and avoid local minima.

Bibliography

[1] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(6270):446–449, 1986.

[2] Richard Ernest Bellman. *Dynamic Programming*. Courier Dover Publications, 1957.

[3] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard Barrett, and Jack J. Dongarra. The Matrix Market: A web resource for test matrix collections. In Ronald F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137, London, 1997. Chapman and Hall.

[4] Ulrik Brandes and Christian Pich. Eigensolver methods for progressive multidimensional scaling of large data. In *Proc. Graph Drawing 2006, LNCS 4372*, pages 42–53. Springer, 2006.

[5] M. M. Bronstein, A. M. Bronstein, R. Kimmel, and I. Yavneh. Multigrid multidimensional scaling. *Numerical Linear Algebra with Applications (NLAA)*, 13:149–171, March-April 2006.

[6] M. Chalmers. A linear iteration time layout algorithm for visualising high dimensional data. In *Proc. IEEE Visualization*, pages 127–132, 1996.

[7] J. de Leeuw. Applications of convex analysis to multidimensional scaling. *Recent developments in statistics*, pages 133–145, 1977.

[8] V. de Silva and J. Tenenbaum. Sparse multidimensional scaling using landmark points. Technical report, Stanford, 2004.

[9] Christos Faloutsos and King-Ip Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. ACM SIGMOD*, pages 163–174, 1995.

[10] Y. Frishman and A. Tal. Online dynamic graph drawing. In *Proc. Eurographics/IEEE VGTC Symp. on Visualization (EuroVis'07)*, 2007.

[11] Emden Gansner, Yehuda Koren, and Stephen North. Graph drawing by stress majorization. In *Proc. Graph Drawing 2004, LNCS 3383*, pages 239–250. Springer, 2004.

[12] Leslie Frederick Greengard. *The rapid evaluation of potential fields in particle systems*. PhD thesis, New Haven, CT, USA, 1987.

[13] S. Ingram, T. Munzner, and M. Olano. Glimmer: Multilevel MDS on the GPU. submitted for publication.

[14] S. Ingram, T. Munzner, and M. Olano. GLUG: GPU Layout of Undirected Graphs. Technical Report TR-2007-23, University of British Columbia, 2007.

[15] Fabien Jourdan and Guy Melancon. Multiscale hybrid MDS. In *Proc. Intl. Conf. on Information Visualization (IV'04)*, pages 388–393, 2004.

[16] A. Krowne and M. Halbert. An initial evaluation of automated organization for digital library browsing. In *Proc. of the 5th ACM/IEEE-CS Joint Conf. on Digital Libraries (JCDL '05)*, pages 246–255, 2005.

[17] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[18] Neil D. Lawrence. Gaussian process latent variable models for visualization of high dimensional data, 2003.

[19] A. Morrison, G. Ross, and M. Chalmers. A hybrid layout algorithm for sub-quadratic multidimensional scaling. In *Proc. IEEE Symposium on Information Visualization (InfoVis'02)*, pages 152–158, 2002.

[20] A. Morrison, G. Ross, and M. Chalmers. Fast multidimensional scaling through sampling, springs and interpolation. *Information Visualization*, 2(1):68–77, 2003.

[21] nVidia. *nVidia CUDA Compute Unified Device Architecture programming guide*, January 2007.

[22] Ken Perlin. An image synthesizer. In *Proc. ACM SIGGRAPH '85*, pages 287–296, 1985.

[23] John Platt. FastMap, MetricMap, and Landmark MDS are all Nyström algorithms. In *Proc. 10th Intl. Workshop on Artificial Intelligence and Statistics*, pages 261–268. Society for Artificial Intelligence and Statistics, 2005.

[24] G. Reina and T. Ertl. Implementing FastMap on the GPU: Considerations on General-Purpose Computation on Graphics Hardware. In *Theory and Practice of Computer Graphics '05*, pages 51–58, 2005.

[25] G. Ross and M. Chalmers. A visual workspace for constructing hybrid multidimensional scaling algorithms and coordinating multiple views. *Information Visualization*, 2(4):247–257, December 2003.

[26] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500), Dec 22 2000.

[27] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. A metric for distributions with applications to image databases. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 59, Washington, DC, USA, 1998. IEEE Computer Society.

[28] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, Dec 22 2000.

[29] W. S. Torgerson. Multidimensional scaling: I. theory and method. *Psychometrika*, 17:401–419, 1952.

[30] G. Young and A. S. Householder. Discussion of a set of points in terms of their mutual distances. *Psychometrika*, 3(1), January 1938.