Glimmer: Multilevel MDS on the GPU

Stephen Ingram, Tamara Munzner, *Member, IEEE,* and Marc Olano, *Member, IEEE*

***Abstract*— We present Glimmer, a new multilevel 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 makes local minima less likely 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 also show that the performance of Glimmer on GPUs is substantially faster than a CPU implementation of the same algorithm.**

***Index Terms*— Multidimensional scaling, multilevel algorithms, optimization, GPGPU.**

I. 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 dimensionality of the dataset is conjectured to be smaller than dimensionality of the measurements. When dimensionality reduction is 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 [4].

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.

MDS algorithms work by minimizing an objective function based on the discrepancy of these distances. A standard *stress* error metric is the the normalized stress metric between D, the matrix of low-dimensional distances d_{ij}, and Δ, the matrix of high-dimensional distances δ_{ij}:

$$stress(D,\Delta) = \sqrt{\frac{\sum_{ij}(d_{ij}-\delta_{ij})^2}{\sum_{ij}\delta_{ij}^2}} \qquad (1)$$

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 II 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 stress. Finally, we devise a simple multilevel strategy that demonstrably reduces convergence to 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.

Below, we discuss the rich previous work in Section II, and then present the core ideas of the Glimmer multilevel algorithm and GPU-SF algorithm in Section III. We cover GPU considerations in Section IV, providing the details of our GPU-based algorithms. In Section V 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.

II. Previous Work

The foundational ideas behind multidimensional scaling were first proposed by Young and Householder [25], then further developed by Torgerson [24] 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

Stephen Ingram and Tamara Munzner are with the University of British Columbia, E-mail: {sfingram,tmm}@cs.ubc.ca.

Marc Olano is with the University of Maryland, Baltimore County, E-mail: olano@umbc.edu.

the dimensionality of the low-dimensional target space, while H is the dimensionality of the high-dimensional input space.

A. Classical Scaling

Classical scaling methods compute exact or approximate analytical solutions to the minimum of the *strain* function. Although strain is closely related to stress, it may have a very different minimum. These spectral methods find embedding coordinates by computing the top eigenvectors of a "double-centered" transformation of the distance matrix sorted by decreasing eigenvalue. The original algorithm, Classic MDS [24], [25] 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 [20] 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 [7], LLE [23], Landmark MDS [6], and PivotMDS [2]. We use PivotMDS as an exemplar in the Glimmer performance comparison of section V, since it was shown to be a fast and accurate classical scaling approximation algorithm [2]. All of these techniques achieve dramatic speed improvements by reducing the complexity to essentially $O(N)$. However, in Section V we discuss the limitations of these approaches in handling sparse datasets. The Glimmer approach of distance scaling yields higher quality layouts in these cases, and has competitive speeds whenever the visual quality is equal.

B. Distance Scaling by Nonlinear Optimization

Optimizing the stress function using gradient descent to find a low-error embedding was pioneered by Kruskal [15]. 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. De Leeuw's accurate SMACOF [5] monotonically converges to a stationary point by minimizing a quadratic approximation at each iteration, resulting in provably linear convergence but at a large cost of $O(N^2L)$ per iteration. Gansner *et al.* [10] 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 an $O(N^2)$ pre-processing procedure. Accelerations of this technique are not straightforward to apply in high dimensions.

The recent Multigrid MDS [3] 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 reduces convergence to 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 III-A).

C. 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. They can be considered a type of gradient descent with local linear gradients. 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.

The basic force-directed approach has a complexity of $O(N^3)$, with an $O(N^2)$ cost per iteration for N iterations. The CPU-based stochastic force approach introduced by Chalmers [4] 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 $O(N^{5/4})$ [16] and $O(N \log N)$ [11]. 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 Section V.

D. Graph Layout Algorithms

MDS has strong a connection to graph drawing. In fact, performing MDS on a dataset is equivalent to performing Kamada and Kawai's energy-based graph layout on a complete graph whose vertices correspond to points in the dataset and whose edges are weighted by the high-dimensional distance between the corresponding points.

Many fast graph drawing algorithms such as ACE and Subspace Optimization [13], [12] have been proposed that make order of magnitude speed gains with quality results by leveraging the sparseness of the graph Laplacian matrix. The sparseness of the Laplacian matrix depends on the distribution of edges in the graph. When you consider the problem of a fully connected graph as we do in MDS, these algorithms become $O(N^2)$ or worse.

One may argue that nearest-neighbor strategies may be used to build a graph over the original vertices with a sparse Laplacian, thus permitting fast graph layout algorithms to compute layouts for the data. In practice, such graph-building techniques always make potentially faulty assumptions regarding the underlying topology of the data. First, nearest neighbor search algorithms that are not computationally exhaustive degrade as a function of the dimension of the data. For example, the popular Approximate-Nearest-Neighbor algorithm [1] computes a $(1+\varepsilon)$-approximate nearest neighbor of a point in $O((H\lceil 1+6H/\varepsilon)^H \log N)$ time. This approach is far too expensive for high-dimensional data where $H = 28,374$ as it is in one of our test datasets in section V. Second, measurement noise can destabilize the topology of such graphs making the results sensitive to the parameters of the algorithm used to construct the graph. Finally, care must be taken to ensure the resulting graph is fully connected. Due to the number of complications involved with this strategy, we do not consider graph layout algorithms that rely on a sparse topology to be examples of MDS algorithms.

These arguments do not imply that MDS algorithms cannot employ subsampling strategies for sparse iterations. For example, the stochastic force algorithm uses a different random sampling of distances at each iteration. We further discuss the advantages of the constant-size random selection strategy in section IV-B.

E. 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 [21] 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 [9], [8] take advantage of GPU parallelism to increase the speed of graph layout algorithms. As mentioned above, force-directed graph layout does have deep similarities to force-directed MDS. However both algorithms' acceleration strategies break down in the case of weighted complete graphs. In the dynamic algorithm, an edge collapsing step requires computing $O(N^2)$ edge weights. In the static algorithm their initial partitioning strategy uses graph Laplacian which is $O(N^2)$ in the case of a complete graph. As with other fast graph algorithms, they are able to make productive use of graph-topology assumptions that may not hold for the full MDS problem. Furthermore, the energy function they minimize on the GPU ignores pairwise distances, and thus does not minimize stress. Finally, they use the CPU for initial placement and for spatial partitioning, whereas Glimmer runs all stages entirely on the GPU.

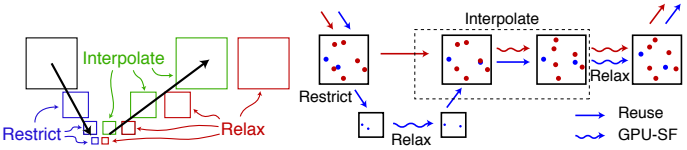
We further discuss the suitability of previous algorithms for speedup using GPU parallelism in Section IV-A.

III. 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. Simliarly, our multilevel heuristic is justified empirically, rather than analytically.

A. 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 1. Loosely speaking, restriction performs the decimation to build the hierarchy, relaxation is the core computation operator that reduces



(a) Multigrid algorithms (b) Glimmer algorithm

Fig. 1. **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 new points allowed to move. Lower levels untwist complex layouts while higher levels converge quickly because of computation at the lower levels.

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.

B. Multilevel Algorithm

Figure 1 shows a diagram of the Glimmer multilevel algorithm as a single v-cycle. The pseudocode is given in Figure 3. 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 subsystem 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.

The major difference between Glimmer and the GPU-SF subsystem alone is accuracy and convergence. Figure 2 illustrates the convergence problems of GPU-SF compared to Glimmer. After a threshold of approximately 12,000 points, the gray GPU-SF algorithm consistently converges to a much higher stress configuration than the purple Glimmer line. The existence of

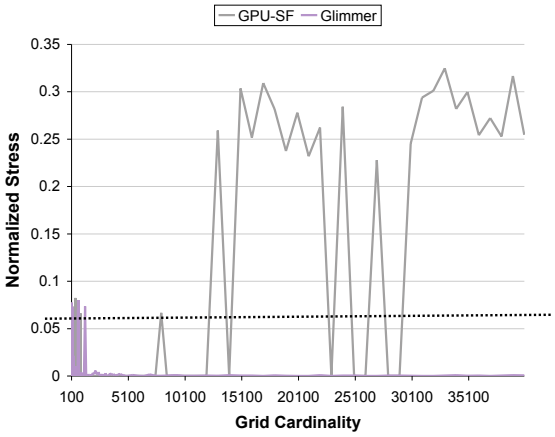


Fig. 2. Graph of final stress of single-level GPU-SF versus multilevel Glimmer on a large range of input dataset cardinalities. After approximately 12,000 points, GPU-SF terminates too soon. Consistent problems with convergence on large datasets disqualifies GPU-SF as a scalable MDS algorithm.

```
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); // interpolate
  runGPUSF(emptyset, points);        // relax
```

Fig. 3. Pseudocode for the Glimmer algorithm.

some purple spikes in the figure also provides evidence that Glimmer is not immune to these local minima, but is much less likely than GPU-SF to converge to them. Local minima, can give rise to twisted manifolds in the low-dimensional placement, as shown in Figure 4. 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.

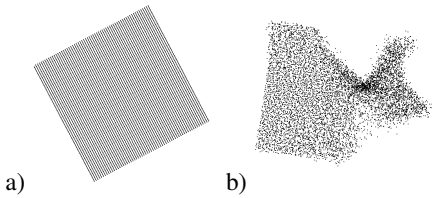


Fig. 4. Visual quality differences between a) Glimmer and b) GPU-SF for `grid` instance with cardinality 8000. Glimmer exhibits more stable convergence behavior than GPU-SF, which more frequently yields a twisted layout when it is caught in a local minimum and terminates with a high stress value. This layout corresponds to the spike at 8000 for GPU-SF in Figure 2.

C. GPU Considerations

The Glimmer algorithm can run on a CPU, and we have implemented an optimized C prototype 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. We refer the reader to Owens et al. [18] for a good survey on the use of GPUs for general purpose computation.

Glimmer and GPU-SF are general approaches that do not depend on specific hardware features of a particular GPU. The most recent nVidia GPUs (G80 and later) handle all three shader types with a shared set of SIMD clusters that can be programmed with a general-purpose parallel languaged called CUDA [17]. Although our algorithms could be implemented on 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 V.

D. 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 merely 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 [3] must carry out an preprocessing step to find nearest neighbors. In our approach, neighborhoods around each point are gradually discovered during the stochastic interpolation and relaxation operations.

IV. GPU Stochastic Force

GPU-SF is our GPU-friendly stochastic force MDS solver used as a subsystem in Glimmer, inspired by the Chalmers [4] 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.

A. 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 [2], [16] 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 [4] as especially appropriate for our requirements. Each point only references a small set of other points during an iteration step, and the selection of this set changes each iteration and 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.

B. 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. 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.

The heuristic behind the selection rules is simple: a point's coordinates are derived from both local and global position information. The local information comes from the iteratively refined Near set and the global information comes from the always changing Random set. The heuristic has two advantages over preselection and sparse-graph construction strategies. First, a fixed number of neighbors are referenced and so GPU storage is known a priori. This is in contrast to techniques that select all neighbors under a given threshold because the number of neighbors cannot be known for an arbitrary dataset. Second, it reduces the bias of global position information. In strategies where landmark points are randomly chosen once at the start of the algorithm, the layout of all points biased in favor of those points. In the case of Chalmers, the global distance information is a weighted combination of a much larger random set.

C. 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 [22]. 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 [11], [16], the computation is run for a fixed number of iterations, usually N. Although linear convergence was proven for the SMACOF algorithm [5], it has been generally assumed for many force-directed approaches. We show in Section V 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) 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.

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

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

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 low-pass filter, a windowed sinc in our implementation. The resulting smooth signal closely mimics the behavior of the true normalized stress function, as shown in Figure 5. Since we are interested in the behavior of 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

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

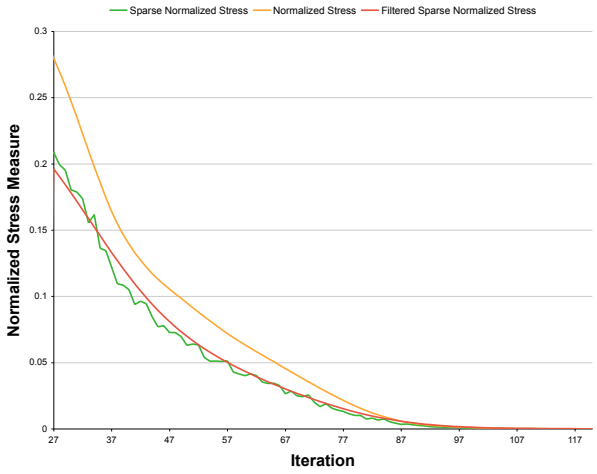


Fig. 5. GPU-SF uses a sparse approximation (green) of the normalized stress function (orange), which converges simultaneously and requires only minimal overhead to compute. We use a low-pass filter (red), because the noise in the unfiltered signal is much larger than the convergence threshold of $\varepsilon = .0001$.

where $\star$ is the convolution operator and *deriv* is 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 [4], and others that use it as a subsystem [11], [16].

D. 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. Figure 7 lists the textures used to store this information. The vectors are `posHi` and `posLo`, the high- and low-dimensional position of the points. Each element of `posHi` has size H, where H is the dimensionality of the high-dimensional space. The size of `posLo` elements is L, the dimensionality of the low-dimensional space, which

Tex. Name	Size	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

Fig. 7. The GPU-SF algorithm uses textures as storage. This table lists each texture used by the algorithm, the size in pixels of the individual elements dedicated to each point and a brief description of the purpose of the texture.

in Glimmer is 2. The `velocity` texture keeps track of point velocities in the low-dimensional space, and also has size L elements. The tables all have 8 elements, divided 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.

The remaining three 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.

Figure 6 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 6, 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.

a) 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 [19].

b) Stages 2 & 3: We need to compute `distHi`, the Euclidean distances in high-dimensional space. 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.

c) Stage 4: Updating the Near set with points in Random that are closer is slightly tricky. If we 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-

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`

Fig. 6. 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 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. All GPU sorting is done using an even-odd sorting network.

d) 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.

e) Stages 6 & 7: We integrate the `scratch` forces into `velocity` in one pass, then integrate `velocity` and update `posLo` in another pass.

f) 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 well supported on GPUs. The effect of those symmetric forces emerges implicitly as the Near sets of neighboring points gradually converge to include each other.

V. 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 Classic MDS, SMACOF, and Chalmers. 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 PivotMDS [2] as the classical scaling approach, using 50 landmarks except where noted. We use Jourdan's $O(N \log N)$ Hybrid [11] as the fastest force-directed approach. Bronstein's Multigrid MDS [3] 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 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.

We also compare against an implementation of Glimmer on the CPU to separate the speedup achieved by the multilevel algorithm and the subsequent GPU speedup.

All algorithms are implemented in C by the authors. Optimized third-party, matrix-multiplication routines were used for Classic, SMACOF, and PivotMDS. Our C implementations were tested against Java[1] and MATLAB[2] versions of the algorithms where available and verified to have between a 1% to 80% speed improvement.

A. 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 6 yields a per-iteration cost of $(7+\log_4 H+\log_4 L+5.33\,L) * N_i = O(N_i \log_4 H)$.

[1]PivotMDS software courtesy of Christian Pich. Hybrid implementation from `www.lirmm.fr/~fjourdan/Projets/MDS/MDSAPI.html`

[2]Classic implementation from `cobweb.ecn.purdue.edu/~malcolm/interval/2000-025`

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 IV-C, 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 computation 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 $O(C N \log_4 H \,/\, p)$. If we assume C to be $O(N)$, and both $\log_4 H$ and p to be a small constants, then the complexity of Glimmer is essentially $O(N^2)$.

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.

B. Performance Comparison

We compare Glimmer to several MDS algorithms, across a range of real and synthetic datasets. All benchmarks 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 paper 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) Datasets: We use a mix of of synthetic and real-world benchmark datasets. The small `cancer` dataset from the UCI ML Repository[3] 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[4] [14]. 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.

This group of datasets permit us to characterize the speed and stress of each MDS algorithm in different dimensionality scenarios. In the case of the regular `grid`, the true dimensionality is equal to the embedding dimensionality and so an MDS algorithm should produce stress results very close to zero and the regularity should be visually apparent in the layout. For `shuttle`, where the true dimensionality is conjectured to be slightly greater than the embedding dimension, both the global structure and the local proximity of the data may be important but neither can be reconstructed without some distortion. However, some cluster structure can be distinguished. For `docs`, because the true dimensionality is believed to be at least an order of magnitude greater than the embedding dimension, the global relationships between points are less important and potentially misleading. Again, the local cluster relationships and their distinguishability from each other should be emphasized.

2) Layout Quality: Figure 8 shows the visual quality, normalized stress, and timing of Glimmer, 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, 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 Glimmer and PivotMDS algorithms indicate these two color-coded groups clearly with spatial position. Quantitatively, the stress of Glimmer 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`, Hybrid produces a readable layout separating the red cluster from the other two, but is slower by several hundred percent. 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 and PivotMDS in comparable times. Hybrid is again 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 green

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

[4] Data courtesy of Aaron Krowne.

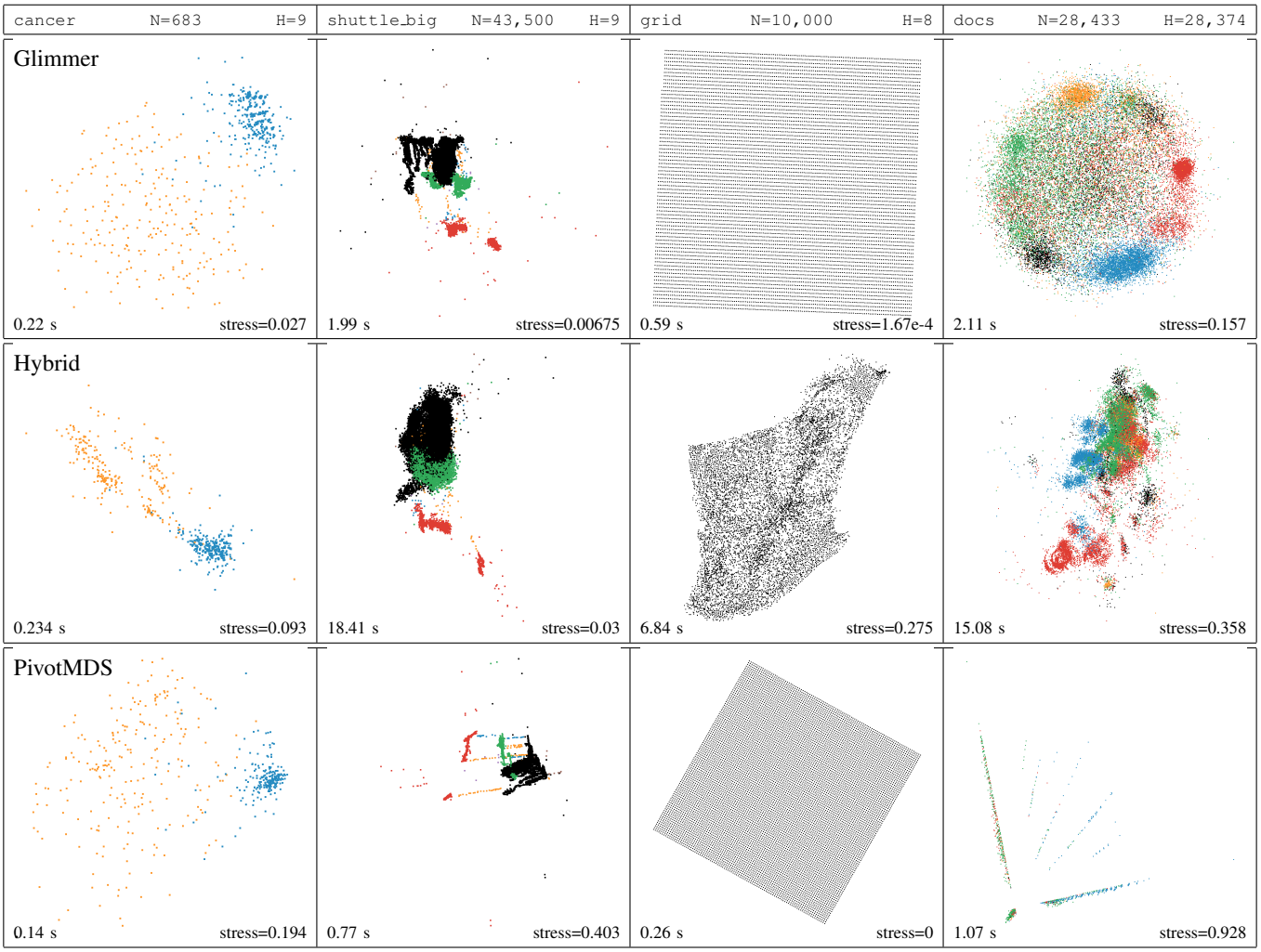


Fig. 8. MDS layouts showing visual quality, time, and stress for the Glimmer, Hybrid, and PivotMDS algorithms. Dataset name, number of nodes (N), and number of dimensions (D) appear above each column. Time in seconds appears at the bottom left of each entry, with normalized stress on the bottom right.

cluster is split into three parts. It took approximately 2 seconds with normalized stress of 0.157. Hybrid suffers from cluster occlusion. The stress is nearly twice as high as Glimmer, 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.

3) Speed and Stress: We use the synthetic `grid` dataset and parameterized random permutations of `shuttle` and `docs` to compare algorithm speed and accuracy across a large interval of dataset cardinalities.

The timings in Figures 9abc all exhibit the same pattern of three equivalence classes. The first and slowest class of algorithms are the foundational algorithms: Classic in pink, SMACOF in blue, and Chalmers in orange. These algorithms all show timing curves that are quadratic or worse. Assuming enough computational resources were present, completion time on large datasets of 100,000 points or more for these algorithms would be on the order of many hours or even days. The second class of algorithms are Hybrid and GlimmerCPU. These algorithms terminate in approximately one minute on very large datasets of 100,000 points. The final and fastest class of algorithms are Glimmer and PivotMDS. These algorithms have a much smaller slope compared to the other classes, requiring only a handful of seconds to compute layouts of approximately 100,000 points.

The stress measurements in Figures 9def, with log-scale vertical axes, each exhibit a different pattern depending on the dataset. We have placed dashed lines on the graphs to roughly delineate the boundary where the visualization goals for each dataset are satisfied (below) and where they are not (above). We determined the positions of these lines empirically by computing layouts with different stress values and making qualitative judgements about their visual structure. We used the criteria discussed in Section V-B.1: regularity for `grid`, and cluster structure for `shuttle` and `docs`. We characterize an algorithm as outperformed by a competitor when the algorithm's average stress across dataset cardinalities falls above this line, even if the competitor is slower.

All the algorithms satisfy the visual quality test for the `shuttle` dataset. For `grid`, the Hybrid algorithm in green regularly results in a distorted grid above the dashed line. The remaining algorithms all regularly produce layouts with no visible distortion. Chalmers, PivotMDS, and Classic all produce zero-stress layouts, so are not visible on the graph because they coincide with the horizontal axis. In the case of `docs`, where the intrinsic

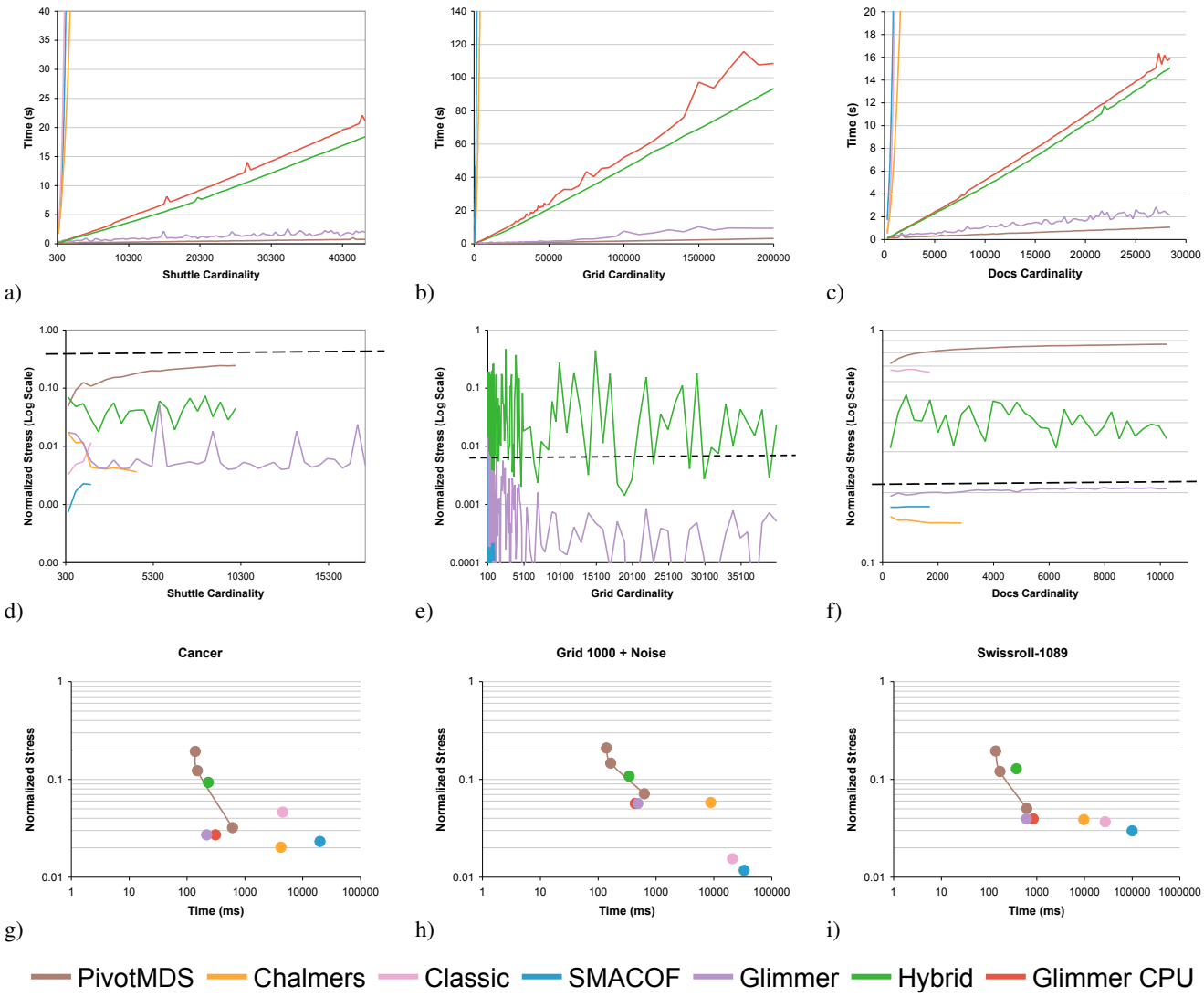


Fig. 9. **a-c)** Detailed graphs of timings for the seven measured MDS algorithms on `shuttle`, `grid`, and `docs` datasets of increasing cardinality. The graphs exhibit the same three speed classes of algorithms. The Glimmer GPU algorithm is a member of the fastest speed class. **d-f)** Graphs of layout stress for the seven MDS algorithms on `shuttle`, `grid`, and `docs` datasets of increasing cardinality. Each graph has a dashed black line demarcating unacceptable visible distortion. Glimmer is the only algorithm in the fastest speed class to regularly fall beneath each of the visible distortion lines. **j-l)** 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 outliers Glimmer CPU (red) and Glimmer (violet) on the side of the tradeoff with lower stress in shorter time. All timings include distance calculations and layout and all stresses are the full normalized stress calculation.

dimensionality is very high, brown PivotMDS, pink Classic, and green Hybrid are all well above the dashed line. Glimmer in purple, Chalmers in orange, and SMACOF in blue all produce results that group similar points together and separate clusters.

Figures 9g- 9i further illustrate the relationship of speed and stress, showing log-log scatterplots of the timing and stress of the seven algorithms on three samller 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 Chalmers (orange), PivotMDS (brown), Hybrid (green), Classic (pink), and SMACOF (blue), indicating a simple speed-accuracy tradeoff for these algorithms. Glimmer (violet) and Glimmer-CPU (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. On these smaller datasets, the GPU does not significantly improve speed over the Glimmer on the CPU.

4) Summary: The Glimmer algorithm satisfies the visual quality test for each dataset and is in the fastest equivalence class. The other algorithm in the fastest equivalence class, PivotMDS, does not produce a usable layout for the `docs` dataset. The other algorithms that satisfy the visual quality test for all datasets, SMACOF and Chalmers, do not scale to large datasets, either running out

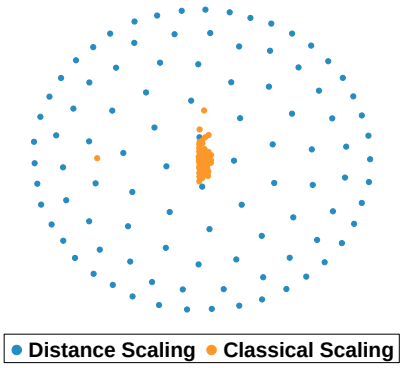


Fig. 10. Layouts of a regular 100-simplex produced by distance scaling and classical scaling. Both methods distort the simplex. Distance scaling algorithms like Glimmer produces less point occlusion and better preserves the diameter of the simplex.

of memory or requiring hundreds of hours to compute.

C. 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. When k is greater than the desired embedding dimension ($L = 2$ in this paper), 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 10 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.

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 occlusion may sometimes be advantageous, because clusters may be more easily distinguished from each other. However, we argue that for sparse, very high dimensional datasets such as `docs` or for tagged datasets, distance scaling is very likely to be a better choice than classical scaling. The PivotMDS layout of the `docs` dataset shown in Figure 9i, produced by minimization of the strain objective, demonstrates that no two-dimensional 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.

D. GPU Speedup

We now provide quantitative measurements of the GPU speedup for Glimmer. Figure 11 shows the speed improvement Glimmer algorithm on two different GPUs versus the completely CPU-based implementation. The graph is was constructed by dividing the CPU run times by the GPU times for 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. The graph clearly shows considerable speed improvements of the Glimmer GPU algorithm. The older nVidia 7900GS card converges to a constant speedup around 2.5. The newer nVidia 8800GTX reaches a variable speedup factor between 10 and 15 for grids of cardinality greater than 10,000.

The GPU speedup comes with startup and overhead costs. These include shader compilation, shader optimization, and data initialization-upload/download. Figure 12 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.

VI. Conclusion and Future Work

Glimmer provides 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. 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. It is competitive with previous classical scaling approximations in speed, and yields readable results for sparse datasets where these approximations fail.

It would be interesting future work to adapt the Glimmer approach for optimized force-directed graph placement. Also, Glimmer

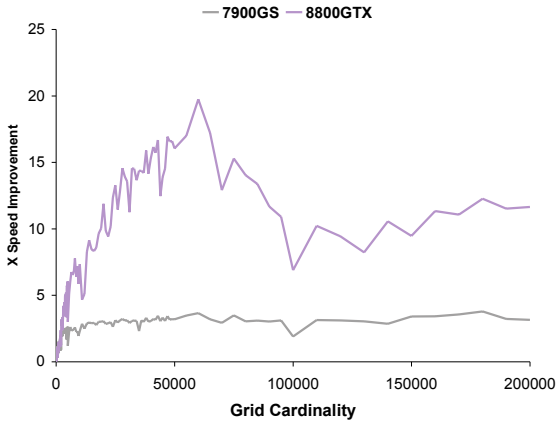


Fig. 11. GPU speedup for two different cards, calculated by dividing the time required to complete a layout using Glimmer on the CPU by the time required on a GPU.

	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

Fig. 12. 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.

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.

The source code and executable for Glimmer is available at `www.cs.ubc.ca/~sfingram/glimmer`.

Acknowledgements

We thank Dan Archambault, Aaron Barsky, Heidi Lam, Peter McLachlan, James Slack, and especially Ciarán Llachlan Leavitt, for feedback on paper drafts. We also thank Dan Archambault for help with timing experiments. We appreciate the thoughtful comments of anonymous reviewers.

References

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

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

[3] 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.

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

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

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

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

[8] Y. Frishman and A. Tal. Multi-level graph layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, 2007.

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

[10] E. Gansner, Y. Koren, and S. North. Graph drawing by stress majorization. In *Proc. Graph Drawing 2004, LNCS 3383*, pages 239–250. Springer, 2004.

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

[12] Y. Koren. Graph drawing by subspace optimization. In *Proc. Eurographics/IEEE TCVG Symp. on Visualization (VisSym '04)*, pages 65–74, 2004.

[13] Y. Koren, L. Carmel, and D. Harel. ACE: A fast multiscale eigenvectors computation for drawing huge graphs. In *INFOVIS '02: Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, page 137, Washington, DC, USA, 2002. IEEE Computer Society.

[14] 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.

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

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

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

[18] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

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

[20] J. 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.

[21] 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.

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

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

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

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

Stephen Ingram is a PhD student in the Computer Science Department of the University of British Columbia. He received his MSc degree from UBC in 2008 and his BSc Honors degree in Computer Science from Georgia Tech in 2004. He has worked on visualization at Emory University and the BC Cancer Agency. His research interests are dimensionality reduction, information visualization, and computational finance.

Tamara Munzner is an associate professor in the Computer Science Department of the University of British Columbia. She was a technical staff member at the University of Minnesota Geometry Center from 1991 to 1995, received the PhD degree in 2000 from Stanford, and was a research scientist at the Compaq Systems Research Center from 2000 to 2002. Her research interests are information visualization, graph drawing, and dimensionality reduction.

Marc Olano received the BS degree in Electrical Engineering from the University of Illinois in 1990 and the PhD degree in Computer Science from the University of North Carolina in 1998. He is currently an associate professor in the Computer Science and Electrical Engineering department, and Director of the Computer Science Game Development Track at the University of Maryland, Baltimore County. His PhD dissertation described the first interactive graphics hardware system supporting a "shading language". After his PhD, Dr. Olano worked at SGI, creating shading languages for commercial graphics hardware. Since 2002, Dr. Olano has been at UMBC, where his research continues to revolved around all aspects of design and use of programmable graphics hardware. He is a member of ACM SIGGRAPH, Eurographics, the IEEE and the IEEE Computer Society.