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 that accurately reflects the high-dimensional structure of the original data in the low-dimensional embedding, and converges well. It is designed to exploit modern graphics processing unit (GPU) hardware for a dramatic speedup compared to previous work. We also present GPU-SF, an efficient GPU version of a stochastic force algorithm that we use as a subsystem in Glimmer. We propose robust termination conditions for the iterative GPU-SF computation based the filtered sum of point velocities. Our algorithms can either compute high-dimensional Euclidean distance on the fly from a set of high-dimensional points as input, or handle precomputed distance matrices. The $O(N^2)$ size of these matrices would quickly overflow texture memory, so we propose *distance paging* and *distance feeding* to remove this scalability restriction. We demonstrate Glimmer's benefits in terms of speed, convergence and correctness against several previous algorithms for a range of synthetic and real benchmark datasets.

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

1 Introduction

Multidimensional scaling, or MDS, is a popular technique for dimensionality reduction, where data in a measured high-dimensional space is mapped into some lower-dimensional target space while minimizing spatial distortion. Although some information will necessarily be lost in this mapping, the goal is to retain as much structure as possible. MDS is typically used when the true 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 usually 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 [3].

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 on the fly, 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 Kruskal Stress-1 metric:

$$stress1(D,\Delta)^2 = \frac{\sum_{ij}\left(d_{ij}-\delta_{ij}\right)^2}{\sum_{ij} d_{ij}^2} \quad (1)$$

If the embedded structure matches the original structure of the data, then $stress = 0$. Stress becomes larger as the spatial distortion between the embedding and the original data increases.

Several factors need to be considered when measuring the success of an MDS algorithm. The most straightforward is the **speed** of the computation, which is easy to measure and discussed explicitly in most of the previous work, both in terms of asymptotic complexity and of wall-clock timings. However, the **correctness** of the embedded layout, the degree to which it reflects the high-dimensional structure of the original data, is at least as important. It can be difficult to qualitatively verify the correctness of a layout by simply viewing images of the low-dimensional layout unless the ground truth is known in advance. Some real or synthetic benchmarks have a known shape, while others are pre-categorized into known clusters that should remain grouped in the low-dimensional representation. The stress metric is a more quantitative measurement of correctness, but the $O(N^2)$ cost of computing it has constrained past analyses.

Any iterative approach to optimization requires a **termination condition** used to decide when the computation has **converged** and should halt, returning an answer. If the computation is not halted at the optimal time, then either correctness or speed suffers: halting too soon delivers an incorrect layout, and halting too late wastes time with computations that make no progress. If the function has local minima, it can converge to a state that is different from, and less correct than, the global minimum. We say that one algorithm has better convergence properties than another if it does a better job of avoiding such local minima. Stress is in fact a nonconvex function containing many local minima, so standard nonlinear optimization methods fail to compute acceptable solutions to modestly sized MDS problems in a reasonable amount of time. As a result, a host of specialized MDS algorithms have been proposed. Some previous algorithms are quite susceptible to getting stuck in the local minima of the stress function rather than finding the best possible layout of points.

The primary contribution of our work is a new multilevel MDS algorithm, Glimmer, that is fast, correct, and converges well. Glimmer was designed to exploit modern highly-parallel PC graphics processing unit (GPU) hardware as a computational engine, for a dramatic speedup compared to previous work. We also present GPU-SF, an efficient GPU version of a stochastic force-directed MDS algorithm, inspired by the work of Chalmers [3]. Glimmer uses GPU-SF as a subsystem inside its multilevel architecture. Both Glimmer and GPU-SF are equally fast, for example laying out an 8D dataset of 200,000 points in roughly 30 seconds. Our algorithms can either compute high-dimensional Euclidean distance on the fly from a set of high-dimensional points as input, or handle precomputed distance matrices. The $O(N^2)$ size of these matrices would quickly overflow texture memory, so we propose *distance paging* and *distance feeding* to remove this scalability restriction.

The speed of Glimmer and of GPU-SF allowed us to analyze their behavior across a large number of datasets. We propose more robust conditions for terminating the iterative GPU-SF computation that confer both speed and correctness advantages over more brittle previous approaches. The multilevel Glimmer approach nevertheless has better convergence properties than GPU-SF alone. We compare the performance of Glimmer and GPU-SF to many other MDS algorithms on several datasets, showing that our methods are both faster and more correct than previous work. Many previous MDS approximation approaches have sacrificed correctness for speed, producing incorrect layouts.

- *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.*

2 Previous Work

The foundational ideas behind multidimensional scaling were first proposed by Young and Householder [15], then further developed by Torgerson [14] and given the name of MDS. In the interests of space we focus on the foundational work and the three most competitive categories of current techniques: single-step spectral methods, nonlinear optimization, and 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.

Classic MDS [14, 15] computes coordinates in $O(N^3)$ time using singular value decomposition of a transformation of Δ called the Gram matrix. By using the largest L singular values of this matrix, it effectively finds the global minimum for the *strain* function for a configuration of points in a space of dimensionality L. Although strain is closely related to stress, it may have a very different minimum. Moreover, it has been demonstrated that Classic MDS is one of the least robust methods with respect to input noise [2]. Due to these drawbacks and its high computational complexity, it is not competitive today, although it is sometimes used on a subset of the data as the input to other MDS methods.

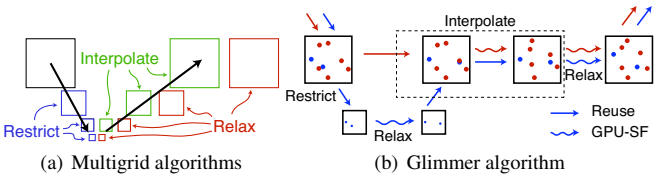
Spectral methods accomplish the same work as classic MDS, but avoid the full singular value decomposition by estimating its first few eigenvalues. A host of Nyström methods [12] have recently been proposed to avoid the $O(N^2)$ computation of Δ altogether, using a subset of that matrix to estimate the eigenvalues. While these techniques achieve dramatic speed improvements over the classic approach and remain globally convergent, they also optimize strain rather than stress We use Landmark MDS [5] as an exemplar in the Glimmer performance comparison of Section 5, since it was recently shown [12] to be the fastest and most accurate classic MDS approximation algorithm.

Optimizing the stress function using gradient descent to find a low-error embedding was pioneered by Kruskal [9]. De Leeuw's SMACOF [4] attempts to avoid local minima by minimizing a quadratic approximation at each iteration to fix the gradient step, resulting in an $O(N^2L)$ cost per iteration and provably linear convergence in $O(N)$ iterations.

The recent Multigrid MDS [1] 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 avoids local minima and makes substantial speed improvements, but the largest example shown in the paper was a layout of 2048 points taking 116.8 seconds. 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).

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. The basic force-directed approach has a complexity of $O(N^3)$, with an $O(N^2)$ cost per iteration for N iterations. The stochastic force (Stochastic) approach introduced by Chalmers [3] 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 $O(N^{5/4})$ [10] and $O(N \log N)$ [7]. Glimmer uses a GPU variant of the Stochastic approach as a subsystem, with an improved termination condition, and we discuss its limitations with respect to correctness and convergence below. We compare Glimmer against three of these approaches in Section 5.

GPUs have been shown to improve the speed of many general purpose algorithms, but have not been previously applied to the problem of MDS. Frishman and Tal [6] take advantage of GPU parallelization 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 a costly $O(N^3)$ precomputation 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.



(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.

3 Glimmer Multilevel Algorithm

Glimmer is a force-based MDS algorithm which uses a recursive hierarchical framework to improve correctness 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 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.

3.2 Multilevel Algorithm

Figure 1 shows a diagram of the Glimmer multilevel algorithm as a v-cycle, and the pseudocode is given in Figure 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 4 between each level, and stop when the size of the lowest level is less than than 1000 points. 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 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.

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

Fig. 2. Pseudocode for the Glimmer algorithm.

At the low levels, only a small subset of the points are involved in the computation, so the system converges very 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 average total time across all levels is roughly the same as with GPU-SF, as we show in Figure 8. The major difference between Glimmer and the GPU-SF subsystem alone is correctness and convergence. When the GPU-SF approach does fall into a local minimum, it will either take longer than Glimmer, or when no further progress can be made it will terminate with an incorrect solution. The multilevel approach usually succeeds in avoiding local minima, which give rise to twisted manifolds in the low-dimensional placement, as shown in Figure 7. Susceptibility to local minima is often cited as a weakness of the force-based methods, but the multilevel approach allows the correct 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.

3.3 GPU Considerations

The Glimmer algorithm can run on a CPU, and we have implemented a MATLAB prototype as a proof of concept. 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 correctness, but brings a dramatic speed improvement over previous MDS approaches.

Modern GPUs include a pipeline of programmable processing stages, each of which is highly parallel. We primarily use the pixel stage, which runs a program, or *shader*, 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 size varies from 16 to 1024 in recent GPUs. This unit has 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.

Glimmer and GPU-SF are general approaches that do not depend on specific hardware features of a particular GPU. They run on any card that supports pixel shaders.

3.4 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. Moreover, handling stochastic operations through permutation is critical for our distance paging approach, as discussed in Section 4.5.

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 [1] must carry out an expensive $O(N^3)$ preprocess to find nearest neighbors. In our approach, neighborhoods around each point are gradually discovered during the stochastic interpolation and relaxation operations.

4 GPU Stochastic Force

We present GPU-SF, a GPU-friendly stochastic force MDS solver used as a subsystem in Glimmer, inspired by the Chalmers [3] algorithm.

4.1 GPU-Friendly MDS

Glimmer's relaxation and interpolation operators both require rapid execution of a simple MDS subsystem, so we need 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.

On the other hand, most of the accelerated MDS algorithms that exploit sparseness fail to achieve correctness. For example, LMDS and the parent-finding approaches of accelerated force-directed MDS [7, 10] achieve their speedups by only considering a small subset of rows of the input distance matrix. While distance matrices frequently exhibit considerable redundancy, these algorithms provide no guarantee that important information is not discarded in the selection of these rows. Section 5 shows that these strategies often yield incorrect results.

We have identified the stochastic force algorithm [3] 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

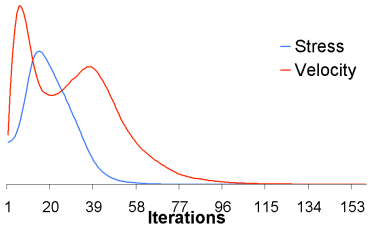


Fig. 3. We use the normalized sum of point velocities in our termination condition. This metric converges shortly after the normalized stress and requires only minimal overhead to compute.

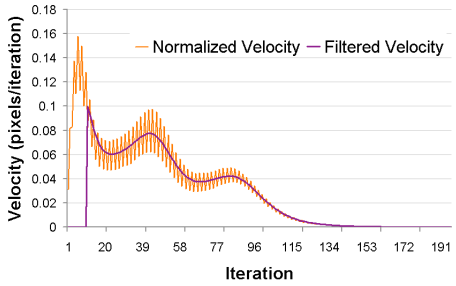


Fig. 4. Velocity is useful in detecting convergence only after filtering. The raw velocity signal is shown in orange, and the filtered in magenta. We also use a heuristic to address mid-frequency noise, and then check against a relative threshold that is smaller than the amplitude of the high frequencies in the original signal.

with a new set of random points. After many iterations, the Near set will converge to the actual set of nearest neighbors.

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 [13]. 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 [7, 10], the computation is run for a fixed number of iterations, usually N. Although linear convergence was proven for the SMACOF algorithm [4], it has been generally assumed for many force-directed approaches. We show that this assumption is not safe to make, frequently leading to overkill that wastes time, or underkill that halts computation before the layout is correct.

A third approach is to terminate when the stress error metric given by Equation (1) stabilizes. Computing stress for a configuration requires N iterations of N^2 computation. Computing stress at each iteration would be far more expensive than the Glimmer algorithm itself.

We instead use a statistic based on point velocities, which are already computed as part of the Stochastic algorithm. Morrison and Chalmers [10] first proposed halting when velocity change is under a specified threshold. The statistic that we use is based on the normalized sum of the velocities of all moving points. Figure 3 shows a representative example of the behavior of velocities versus stress. Although the magnitude of the velocities does not necessarily match the stress, the velocity sum converges shortly after stress. We have empirically verified this property across all datasets that we tested. Velocity is an intuitive metric to use when checking force-directed methods: when the points slow down, the system is probably converging.

While previous approaches simply check for a change smaller than a threshold value, our analysis shows that there is there is high- and mid-frequency noise in the signal, with amplitudes larger than the desired threshold value. Figure 4 shows a typical signal, with high-frequency noise every few iterations, and mid-frequency noise every few dozen iterations that lead to two local minima. We solve the high-frequency noise problem by low-pass filtering with a Hann-windowed sinc function. Using the same strategy for the mid-frequency noise would require a very large window size. We instead use a heuristic where we check if the low-pass filtered signal increases within a second, smaller window. We use a fraction of the highest value from the current invocation of GPU-SF as a relative threshold, rather than an absolute threshold number.

In summary, we use the summed normalized velocities of the moving points as our signal, and after removing high- and mid-frequency noise we check against a threshold relative to the work done in the current level. After empirical testing across many datasets, we arrived at the values of 21 iterations for the high-frequency filter window, 10 iterations for the mid-frequency slope window, and $1/32$ of the current maximum for the relative threshold value. On the relaxation stage on the largest level of the hierarchy, we instead use $1/1000$. Our termination criteria could benefit any iterative, force-based MDS algorithm, including the Stochastic algorithm [3] and others that use it as a subsystem [7, 10].

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 Stochastic 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. 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 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 of all these textures is the number of elements 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. If the input is a distance matrix instead of high-dimensional points, we use the $4N$ `distPage` texture instead of `posHi`.

Figure 4.3 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 4.3, as an estimate of the total work involved. When GPU-SF is invoked as a subsystem of Glimmer, the memory footprint of these textures is

Stage	Passes	Pixels	Input Textures	Output Textures
1 Random Update	1	N_i	`perm`	`index`
2a HighD Distance Calc	$\log_4 H$	N_i	`posHi, index, scratch`	`distHi, scratch`
2b HighD Distance Load	1	N_i	`distPage`	`distHi`
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	$\log_4 N_i$	$N_i/4^j * L$	`velocity`	-

Fig. 5. 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.

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 [11].

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, compute the differences between these points and the current point into 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. If the input data is specified as a matrix, as discussed in Section 4.5, the distances are simply copied from the `distPage` texture into `distHi`. 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 not easy to accomplish on the GPU, because conditionals and loops are expensive. 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. To avoid this problem, 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. We combine 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` from `distHi`, then use the `velocity` texture for damping as described by Chalmers [3]. 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.

Stage 8 The final step of the algorithm checks the termination condition. We can calculate the normalized sum of velocities for our termination condition in $\log_4(N)$ rendering passes using a reduction shader on `velocity`. 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 Stochastic 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 GPUs. The effect of those symmetric forces emerges implicitly as the Near sets of neighboring points gradually converge to include each other.

4.5 Distance Paging for Matrices

In many MDS applications, the input data is given as a precomputed distance matrix. The $O(N^2)$ size of this matrix quickly outstrips available memory, leading to a fundamental scalability challenge. For example, when N is only 4096 points, the matrix would overflow 512MB of texture memory. Our solution, which we call *distance paging*, draws inspiration from texture paging. 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. In this case, we replace the `posHi` texture with the smaller `distPage` texture and distances are simply fetched instead of computed.

Distance paging solves the memory scalability problem of quadratic storage. The time required to precompute distance matrices can be another scalability bottleneck. 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. We support lazy evaluation in the form of a *distance feeder*, a CPU process that takes two points as an argument and returns a distance. Glimmer can thus handle distance matrices far larger than the limits of texture memory on the graphics card.

5 Results and Discussion

We provide an asymptotic analysis of our algorithms, compare our approaches to previous work in terms of speed and correctness, and compare the time required for the distance pager versus the distance feeder.

5.1 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 Table 4.3 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, can range from constant to $O(N)$.

The number of points N_i supplied to GPU-SF at each Glimmer level ranges from 1000 up to N, where $N_{i-1} = N_i/4$, and the number of

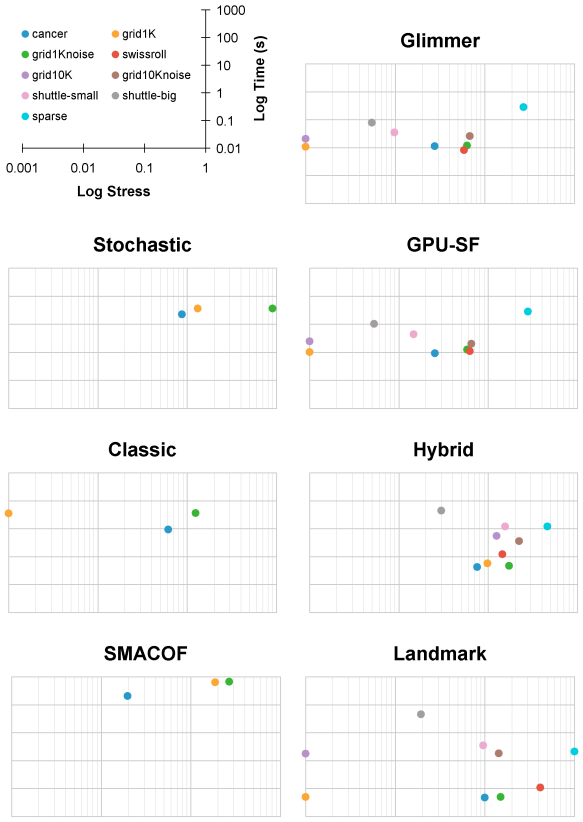


Fig. 6. Log-log small multiples of stress vs. time show the performance for all 7 measured algorithms, across 9 benchmark datasets.

levels is $\log_4 N$. The total number N_t of points processed across all Glimmer levels is bounded above by $4/3\,N$, the infinite sum of $1/4^i * N$. 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 is $O(C\,N\,\log_4 H)$.

We now discuss the effects of GPU parallelism. To oversimplify, a GPU with a SIMD size of k, where k ranges from 16 to 1024 on current cards, speeds up computation up to a factor of k. Since we carefully designed our shaders and render passes to avoid conditionals and loops, our actual speedup is close to this theoretical maximum.

5.2 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 with 1.5 GB of memory and an nVidia 7800GS graphics card with 256MB of texture memory. All timings are averaged across three runs. Unless we explicitly state otherwise, the time includes computing high-dimensional distances on the fly. The timings for Glimmer and GPU-SF includes only layout time, so that they can be directly compared with the other MDS layout algorithms. In our accompanying video, the timings also include render time for interactive display. Stress computations use the normalized metric given in Equation (1).

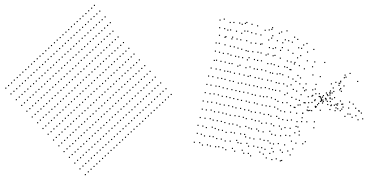


Fig. 7. Glimmer exhibits more stable convergence behavior than GPU-SF alone, which is caught in a local minimum with a twisted grid.

5.2.1 Comparison Algorithms

The MDS algorithms that we chose to compare against are a mix of foundational algorithms and the most-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 Stochastic[2]. These three foundational approaches are known not to be speed-competitive, the main metric of interest for these algorithms is layout stress. We use 40 iterations for the SMACOF algorithm, as suggested by Bronstein [1].

We use our MATLAB implementation of Landmark [5] as the best-of-breed spectral approach, using 15 landmarks as suggested by Platt [12]. We use Jourdan's $O(N \log N)$ Hybrid [7] as the fastest force-directed approach[3]. Bronstein's Multigrid MDS [1] is not publicly available, but we know that it is not speed-competitive with Hybrid or Landmark from the timings given in the paper.

5.2.2 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 and 9 dimensions. The `shuttle_small` dataset, also from UCI, has 14,500 points and 9 dimensions, with `shuttle_big` having the same structure but 43,500 points. 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 the third dimension. The `sparse` dataset is a real-world example of a large collection of unordered document metadata used to study specialized clustering algorithms[4] [8]. 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.

5.2.3 Speed and Correctness

We use the synthetic grid dataset to compare algorithm speed across a large sample of dataset cardinalities. Figure 8 shows that the algorithms fall into three main categories. Glimmer and GPU-SF are clearly the most scalable MDS algorithms, handling the 200,000 point grid in under 30 seconds. Hybrid is the most competitive in terms of speed, but has a much steeper slope than our algorithms. Landmark, while very fast for small datasets, loses its speed advantage at approximately 10,000 points. Glimmer and GPU-SF exhibit approximately 1 second of runtime overhead for data of any cardinality. As a result, they are slower than Hybrid at sizes of less than 3000 points, and Landmark at sizes of less than 10,000 points. As expected, the foundational Classic, SMACOF, and Stochastic algorithms do not scale past roughly 1000 points. Relative to the other approaches, Glimmer

[1] `cobweb.ecn.purdue.edu/~malcolm/interval/2000-025`
[2] `www.lirmm.fr/~fjourdan/Projets/MDS/MDSAPI.html`
[3] `www.ics.uci.edu/~mlearn/MLSummary.html`
[4] Data courtesy of Aaron Krowne.

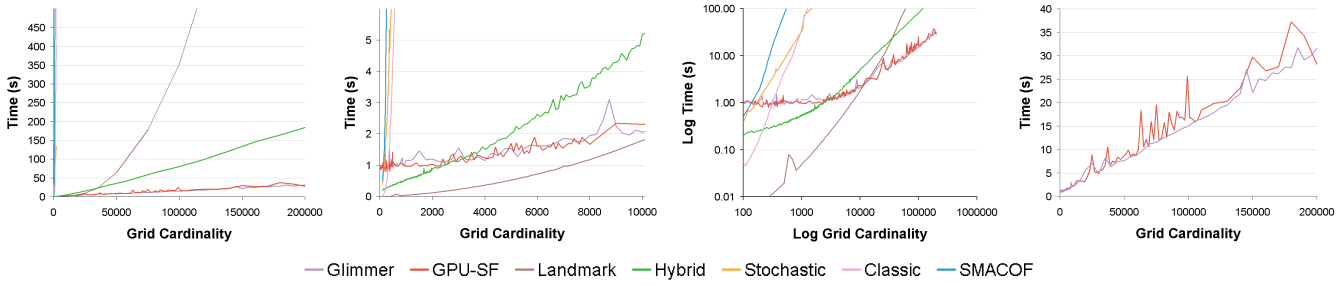


Fig. 8. Timings for synthetic grid datasets of increasing cardinality, with the same data shown at four different scales. On the left, we show the full range. Next, we zoom in on the horizontal cardinality axis. We then show a log-log graph, and on the right is a zoom of the vertical time axis with just Glimmer and GPU-SF.

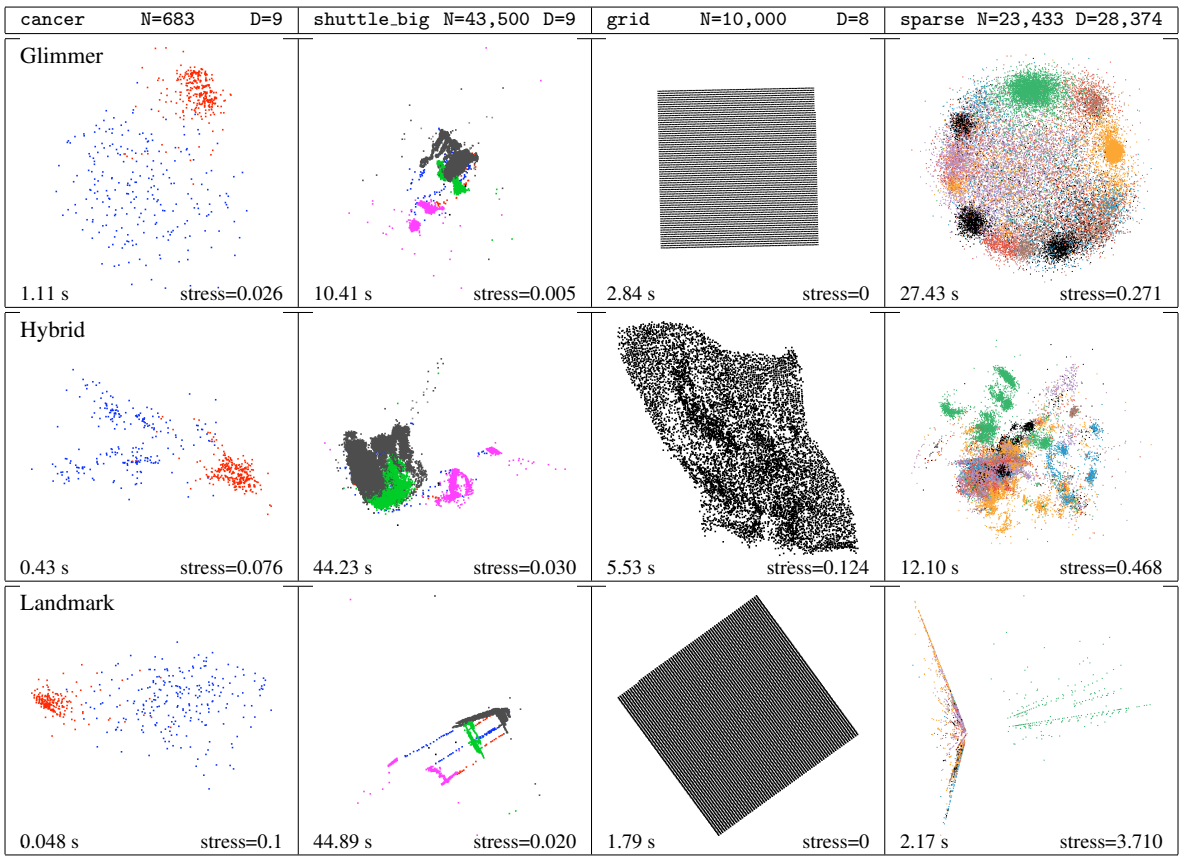


Fig. 9. MDS layouts showing visual quality, time, and stress for the Glimmer, Hybrid, and Landmark algorithms. Dataset name, number of nodes (N), and number of dimensions (D) appear above each column. Time in seconds appears at the bottom left of each entry, with normalized stress on the bottom right.

and GPU-SF are equally fast. However, when compared to each other across many datasets, we can see that GPU-SF has many spikes where it takes longer to converge than Glimmer. This behavior would be hard to spot if we only tested a few datasets. The speed of our new algorithms facilitated rapid testing and analysis, which was critical in designing our robust termination condition for GPU-SF, and in understanding the different convergence properties of GPU-SF and Glimmer.

Figure 9 shows the quality and timing for Glimmer, Hybrid, and Landmark layouts on four datasets with known structure. Points in `cancer` represent tumors, with malignant in red and benign in blue. Qualitatively, both Glimmer and Landmark indicate these two groups clearly with spatial position. Quantitatively, the stress of Glimmer is an order of magnitude lower. Hybrid does separate the two groups, but includes misleading subclusters in the blue group. With `shuttle_big`, Glimmer and Hybrid show the correct structure clearly. Artifacts from the orthogonality constraints of the strain function are visible in Landmark, but it does separate the clusters. Glimmer is four times faster than the other two, and again has significantly lower stress. The 10,000-point grid is correctly embedded with zero stress by Glimmer and Landmark, but Hybrid terminated too soon. The quantitative stress metric is non-zero, and the layout suffers from very obvious distortion.

The Glimmer layout of the `sparse` dataset took nearly 30 seconds and has a stress of 0.271. It shows several spatially distinguishable clusters, color coded by green, red, yellow, and purple. The black cluster is split into three parts. Hybrid is nearly three times faster, but the layout quality is much worse. The stress is nearly twice as high, and the spatial embedding does not clearly separate any of the given clusters. Landmark is very fast, but completely fails to show the dataset structure. The reason is fundamental to the sparse nature of the dataset

and the orthogonality constraints of the algorithm. These sparse matrices are handled very poorly by the approximation algorithms, which achieve their speed improvements by considering only specific rows of the distance matrix. In a sparsely populated document matrix there is no inexpensive way to determine which rows contain enough information to minimize the distortion of the layout. Algorithms based on random search such as stochastic force, which have full access to the distance matrix, are more suited to finding these appropriate distances. Glimmer is the first such algorithm that can scale to datasets of this size.

Although Glimmer and Landmark find the correct embedding, with zero stress, the two approximate force-directed algorithms fail to converge and show a misleading shape with much higher stress. Figure 7 shows that GPU-SF can also fail to converge correctly, whereas the Glimmer multilevel approach succeeds at finding the true global minimum configuration. 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 before the correct 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 pointsets 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.

Figure 6 shows log-log scatterplots of the timing and stress of nine datasets: the five benchmark datasets, two grids of 1,000 and 10,000 points, and two more of the same sizes with noise. We use small multiples, with one scatterplot for each algorithm, to illustrate many speed-accuracy tradeoffs. The Glimmer and GPU-SF scatterplots have the dots on the left high-quality side. The Hybrid and Landmark scatterplots have many dots on the low-quality right. The patterns show the these two approaches sacrifice quality for speed, except in the zero-stress grid case for Landmark. The foundational Stochastic, Classic, and SMACOF algorithms are known to be slow, and, unsurprisingly, the few dots representing the small datasets that they can handle are near the top. We note that these approaches often do not yield ground truth, with the dots on the low-quality right side in many cases. Inspecting a dataset dot of a particular color across the multiple scatterplots shows that those algorithms faster than Glimmer or GPU produce a less correct layout for that dataset. We can also see that GPU-SF is not simply faster than the Stochastic algorithm that inspired it; thanks to the more robust termination condition, it is far more correct.

5.3 Paging and Feeding

We compare the performance of our distance matrix pager and feeder schemes with an example from graph drawing. We use a graph of over 5,000 nodes, where the $O(N^2)$ size of the distance matrix is too large to fit into texture memory. 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. Without precomputation, the layout runs slower, but less work is done in total because unused pairwise distances never need be computed.

6 Conclusion and Future Work

Glimmer and GPU-SF provide dramatic speedups compared to previous work by exploiting GPU parallelism at every stage of their architectures. Our new termination criteria for GPU-SF detects convergence cheaply and accurately. GPU-SF is roughly as fast as Glimmer, but is more prone to getting caught in local minima, whereas the multilevel architecture of Glimmer converges well. The distance pager and distance feeder mechanisms allow us to handle far larger distance matrices than would fit into texture memory. Glimmer avoids the speed-accuracy tradeoff of previous approximation algorithms, as we show on a mix of synthetic and real-world datasets.

It would be interesting future work to adapt the Glimmer approach for optimized force-directed graph placement, to exploit the graph-theoretic connectivity that we must do without for MDS. 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.

7 Acknowledgements

We thank Dan Archambault, Aaron Barsky, Heidi Lam, Peter McLachlan, James Slack, and especially Ciarán Llachlan Leavitt, for feedback on paper drafts.

References

[1] 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.

[2] L. Cayton and S. Dasgupta. Robust euclidean embedding. In *Proc. 23rd Intl. Conf. on Machine Learning (ICML '06)*, pages 169–176. ACM Press, 2006.

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

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

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

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

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

[8] 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.

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

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

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

[12] 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.

[13] 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.

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

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