JOURNAL OF CHEMICAL INFORMATION AND MODELING

ARTICLE

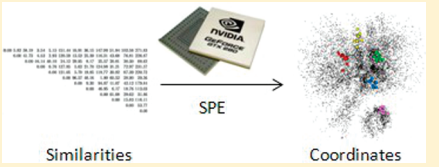
pubs.acs.org/jcim

Stochastic Proximity Embedding on Graphics Processing Units: Taking Multidimensional Scaling to a New Scale

Eric Yang,* Pu Liu, Dimitrii N. Rassokhin, and Dimitris K. Agrafiotis

Johnson & Johnson Pharmaceutical Research and Development, L.L.C., Welsh and McKean Roads, Spring House, Pennsylvania 19477, United States

ABSTRACT: Stochastic proximity embedding (SPE) was developed as a method for efficiently calculating lower dimensional embeddings of high-dimensional data sets. Rather than using a global minimization scheme, SPE relies upon updating the distances of randomly selected points in an iterative fashion. This was found to generate embeddings of comparable quality to those obtained using classical multidimensional scaling algorithms. However, SPE is able to obtain these results in $O(n)$ rather than $O(n^2)$ time and thus is much better suited to large data sets. In an effort both to speed up SPE and utilize it for even larger problems, we have created a multithreaded implementation which takes advantage of the growing general computing power of graphics processing units (GPUs). The use of GPUs allows the embedding of data sets containing millions of data points in interactive time scales.



INTRODUCTION

Multidimensional scaling (MDS) is a class of statistical methods that attempt to generate low-dimensional representations of data objects related to each other through distances or similarities.[1] These similarities may be obtained directly through observation or indirectly through distance calculations from a higher dimensional representation. The classical application of MDS is the conversion of 3D coordinates of cities around the globe to a 2D map that best maintains the distances between them. This technique is widely applied to aid in the visualization of inherent structures within a given data set and is used in fields where high-dimensional data are present, such as pyschoanalytics,[2] market research,[3] and bioinformatics,[4] to name a few. In most cases, the goal is to reduce high-dimensional data into two to three dimensions so that they can be visualized with conventional graphs and make it easier for researchers to form intuitions about the underlying structure of the data.

Most traditional MDS algorithms are formulated as minimization problems where the objective is to minimize a so-called stress function that measures the difference between the distances computed in the original high-dimensional data set (or input dissimilarities) and those in the lower dimensional mapping. This formalism allows for the application of several well-known gradient minimization techniques, such as steepest descent[5] or Newton−Raphson.[6] However, these methods are unable to handle large numbers of data points because the objective functions and their respective gradients require $O(n^2)$ operations to compute. Stochastic proximity embedding[7] (SPE) was proposed as an elegant method that could solve the same underlying problem without requiring explicit evaluation of the stress function. The method starts with a (typically) random initial low-dimensional configuration and iteratively refines it by repeatedly selecting two points at random and adjusting their coordinates so that their distance on the low-dimensional map matches more closely their corresponding distance (or dissimilarity) in the original data set. The fundamental idea behind SPE is that the average change over multiple pairwise refinements represents a good approximation of the gradient of the stress function. Thus, SPE is in effect able to approximate the direction of descent needed for minimization without having to explicitly calculate the computationally expensive stress function. The algorithm is widely applicable,[8−11] extremely fast, scales linearly with the size of the data set in both time and memory, is relatively insensitive to missing data, and is programmatically trivial to implement. More importantly, it can be generalized to embed data objects whose proximities are known only approximately or within certain bounds. This feature has allowed SPE to produce isometric embeddings that preserve geodesic rather than Euclidean distances and thus reveal the intrinsic dimensionality of the data[12−14] and has also made it an ideal alternative to distance geometry[15] for generating molecular conformations and for solving an array of related problems in computational chemistry and biology.[16−25]

While SPE represents an enormous step in efficiency over traditional MDS; data sets on the order of a million data points or larger still require several minutes to run. However, in addition to its computational efficiency, SPE and especially the modified version that takes advantage of randomly selected pivot points for refinement are highly parallelizable because the pairwise updates can be performed independently. Given the ease by which the algorithm can be parallelized, we expect to see significant benefits by targeting graphics processor units (GPUs) or through multithreaded implementations that take advantage of multiple cores in modern CPUs.

Received: September 6, 2011
Published: October 01, 2011

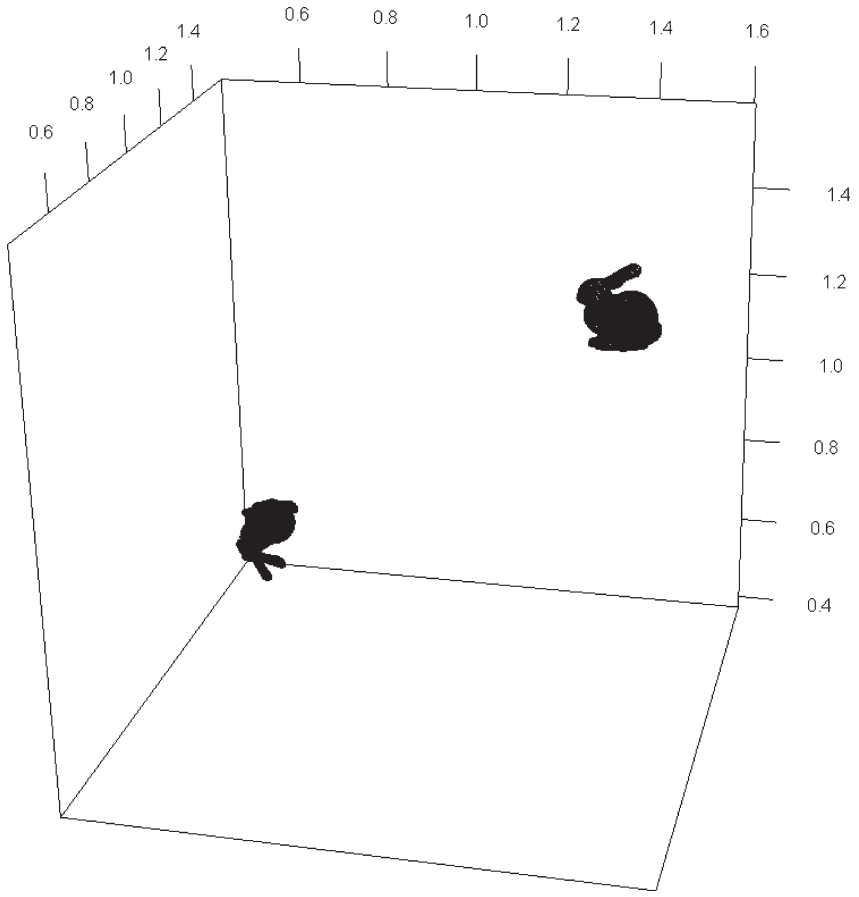


Figure 1. An illustrative example as to what happens if the problem is simply decomposed into disjointed subprocesses without synchronization. Without utilizing global communication between blocks or a global pivot element, we will obtain N disjoint lower dimensionality embeddings.

GPUs represent low cost, highly parallel processors, with anywhere from ten to several hundred independent computational units, each with a small amount of high-speed local memory. They have been used successfully to speed up the highly parallel operations, such as large database and string searches, molecular dynamics, and Monte Carlo simulations, and medical image processing.[26] In those cases where the calculations can be broken down into many independent subproblems, GPUs have shown speed-ups of 100× or greater, but even in less easily parallelizable algorithms, speed-ups of 10−20× can still be easily obtained.[27] In the present work, we utilized a GeForce 580 GTX, a board that costs $500 U.S. dollars and has 512 independent cores, 1.5 GB of local memory, and a theoretical throughput of nearly 1.5 trillion floating point operations per second.

Our overall goal was to determine whether, through parallelization, we can speed up SPE to a sufficient degree such that large data sets can be tackled with commodity hardware common on many laptops and desktops. In this work, we evaluate a parallel implementation of the pivoted SPE algorithm proposed by Rassokhin and Agrafiotis[8] on both a GPU as well as a modern multicore CPU and investigate the bottlenecks associated with these implementations. Our parallel algorithms were tested using three data sets, a moderate-sized 3D point cloud containing 32 000 points, a much larger chemical library containing 908 000 points, and a conformational library utilizing approximately one million separate conformations of gleevec.

METHODS

The fundamental idea behind most MDS implementations is that a higher dimensional representation of a point cloud can be mapped to a lower dimensional point cloud while preserving the distances between any two points. Though the mapping may not be perfect, MDS algorithms in general attempt to minimize the differences between the pairwise distances in the higher and lower dimensions. The differences in the two distance matrices is captured via so-called stress functions, such as Kruskal's stress[2] (eq 1) or Sammon's stress[28] (eq 2). In these equations, d_{ij} represents the distance between any two points (i,j) in the lower dimensional mapping, whereas r_{ij} represents the distance (dissimilarity) between any two points in the original data set.

$$S_k = \sqrt{\frac{\sum_{i<j}(d_{ij} - r_{ij})^2}{\sum_{i<j} d_{ij}^2}} \tag{1}$$

$$S_s = \frac{\sum_{i<j}(d_{ij} - r_{ij})^2 / r_{ij}}{\sum_{i<j} r_{ij}^2} \tag{2}$$

Therefore, MDS is often cast as a minimization problem and solved via various standard minimization techniques. However,

the use of such techniques requires the calculation of the stress function and its gradient in which $O(n^2)$ distance computations must be performed. This makes traditional MDS algorithms impractical for large data sets. The fundamental idea behind SPE is that rather than trying to refine the coordinates of all the data points at once, it updates only one pair at a time but does so repeatedly until a satisfactory embedding is obtained. The magnitude of the correction is controlled by a parameter called learning rate which is gradually reduced during the course of the refinement to avoid oscillatory behavior. More specifically, given a set of N objects, a symmetric matrix of proximities (or dissimilarities) between these objects r_{ij} and a set of images on a m-dimensional display map $\{\mathbf{x}_i, i = 1, 2, .., N; \mathbf{x}_i \in \mathscr{R}^m\}$, SPE attempts to place $\mathbf{x}_i$ into the map in such a way that their Euclidean distances $d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|$ approximate as closely as possible the corresponding values r_{ij}. The algorithm proceeds as follows:

1. Initialize the coordinates $\mathbf{x}_i$ and select an initial learning rate λ.
2. Select two points i,j at random and calculate their distance $d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|$. If $|d_{ij} - r_{ij}| > \delta d$, update the coordinates $\mathbf{x}_i$ and $\mathbf{x}_j$ by:

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \lambda \frac{1}{2} \frac{r_{ij} - d_{ij}}{d_{ij} + \varepsilon} (\mathbf{x}_i - \mathbf{x}_j)$$

$$\mathbf{x}_j \leftarrow \mathbf{x}_j + \lambda \frac{1}{2} \frac{r_{ij} - d_{ij}}{d_{ij} + \varepsilon} (\mathbf{x}_j - \mathbf{x}_i)$$

3. Repeat step 2 for S steps.
4. Decrease the learning rate λ by a prescribed decrement $\delta\lambda$.
5. Repeat step 2 → step 4 for C cycles.

The outer loop adjusts the learning rate, i.e., the magnitude of the refinement between each pair of selected points, whereas the inner loop randomly selects pairs of points and performs the coordinate update. However, this implementation is difficult to parallelize because of the possibility of race conditions. If the inner loop were subdivided into P parallel tasks, two concurrent processes may update the same data point leading to race conditions, corrupted data, and finally incorrect results. A naïve approach may be to break up the problem into P independent subproblems. However, without a mechanism to synchronize the results between the different subproblems, the result would be P disjointed results in which the local embeddings are correct, but the global embedding is not.

This can be solved by utilizing the pivoted variant of SPE (pSPE) in which a global pivot point is chosen prior to entry into the inner loop. Then, in the inner loop, randomly selected points are compared and updated against the pivot. This allows us to break the operation of the innermost loop into P independent tasks, while having them all update versus a common frame of reference. The only requirement for this task to be successful is that the threads must be synchronized after every iteration so that the pivot point is not subject to any race condition. This greatly simplifies the parallelization task. The algorithm proceeds as follows:

1. Initialize the coordinates $\mathbf{x}_i$ and select an initial learning rate λ.
2. Select a pivot point i at random.
3. Select a point j at random and calculate its distance to the pivot, $d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|$. If $|d_{ij} - r_{ij}| > \delta d$, update the coordinates $\mathbf{x}_i$ and $\mathbf{x}_j$ by:

$$\mathbf{x}_j \leftarrow \mathbf{x}_j + \lambda \frac{1}{2} \frac{r_{ij} - d_{ij}}{d_{ij} + \varepsilon} (\mathbf{x}_j - \mathbf{x}_i)$$

4. Repeat step 3 for S steps.
5. Decrease the learning rate λ by a prescribed decrement $\delta\lambda$. Synchronize all threads/kernels (only in the parallel version).
6. Repeat step 2 → step 5 for C cycles.

To parallelize the pSPE algorithm, step 3 is split such that each processor handles N/P data points in the data set. In addition, we introduce a thread synchronization operation in step 5, nominally implemented in a multithreaded implementation as a thread join operation. The GPU implementation requires an additional step, which is a relatively slow copy from main memory onto the device memory on the GPU prior to entry into the pSPE operation as well as a relatively slow copy of the results from the device back to the main memory once the embedding is complete. In the timings described below, the time required to copy the data is included.

In this work, to evaluate the speed-up of the algorithm, we ran the outer loop for 10 000 cycles (denoted by C above) and the inner loop for $0.1 \times N$ steps (denoted by S above). This allows us to get highly consistent measurements of runtime between trials. Rather than running the algorithm for a set number of cycles, alternative implementations could utilize cutoffs for the stress values or determine whether further iterations in the inner loop do not lead to improvements in the embedding. However, for the purpose of evaluating the performance increase associated with parallelization and hardware acceleration, a constant number of iterations is quite appropriate.

HARDWARE

As mentioned previously, the hardware used in our evaluation was a NVIDIA GeForce 580 GTX for the GPU implementation and dual Xeon 5530 processors for the CPU implementation. At the time of this writing, the GeForce 580 GTX represented NVIDIA's highest level consumer grade video card. It consists of 512 independent execution blocks each running at 1.54 GHz. Each execution block has access to 48 K of high-speed shared memory, which can be accessed by multiple threads in a given block as well as 32 K of register space. The card is also equipped with 1.5 Gb of GDDR5 DRAM running at 2 GHz. Its theoretical peak performance is rated at 1.5 TFlops. The Xeon 5530 on the other hand is a slightly older Intel processor and was launched in 2009. With two processors, the CPU hardware has a theoretical peak performance is 76.8 GFlops. Taking a simplistic view based on the theoretical performance numbers, the GPU should have roughly 20× the raw performance of the CPU in our experiments.

The GPU implementation of the pSPE algorithm relies on C for CUDA, which is an extension of the C programming language developed by NVIDIA. When GPGPU programming was in its infancy, programming was done through the use of vertex and pixel shaders which operated upon data that was loaded into texture memory. However, while vertex and pixel shaders exposed through graphics frameworks, such as OpenGL and DirectX, showed great promise, they were still difficult to work with and more importantly did not allow for direct access to the

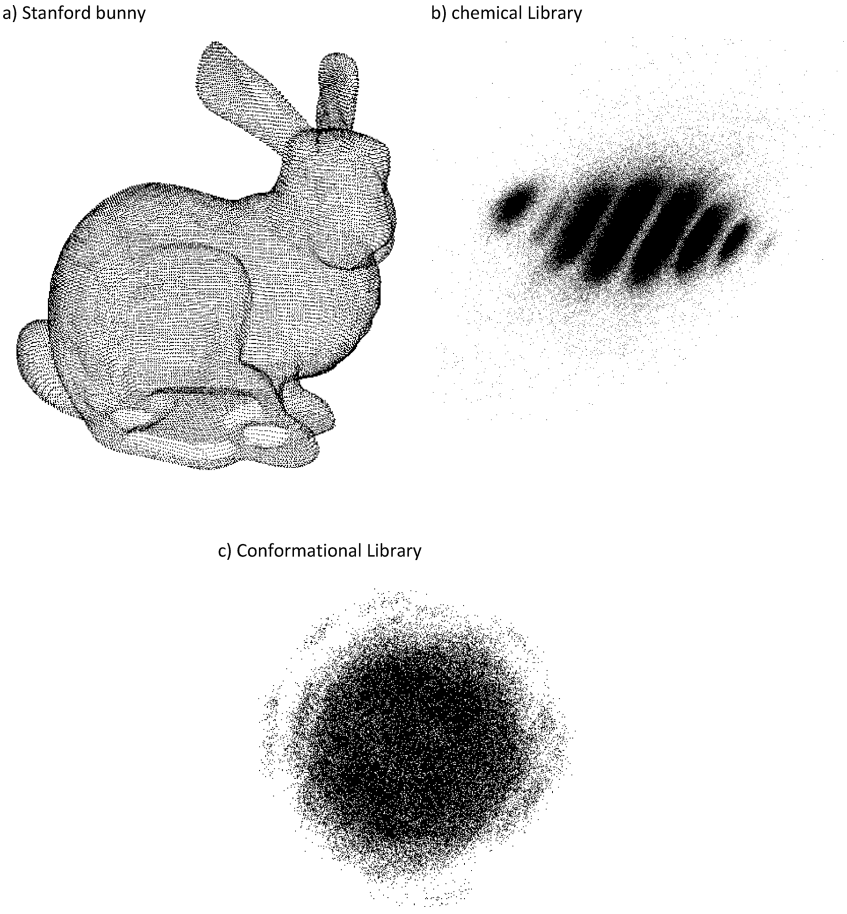


Figure 2. The result of our parallel implementation of pSPE on the Stanford bunny (a) and the chemical library of Golovin and Henrick. In the case of the Stanford bunny, we see that an $R^3 \rightarrow R^3$ mapping leads to a perfect reconstruction of the original image. In the case of the chemical library (b), we see that pSPE has produced an embedding that reveals several families of related molecules. The stress of the Stanford bunny after 10 000 iterations was 4.5×10^{-6}, whereas that of the 2D embedding of the chemical was 0.08. In the embedding of the conformational library, we see that most of the conformations are present within a diffuse shell, with certain disjoint areas indicating significantly different conformations. The resultant stress of the embedding is 0.29.

high-speed shared memory or nonsequential memory access. C for CUDA was developed to address these issues. Other alternative implementations for GPGPU programming have also been proposed, such as OpenCL and DirectCompute. These two frameworks have the advantage of being applicable to GPUs from other manufacturers, such as ATI, as well as the Intel family of integrated graphics units. However, they show significantly lower performance on NVIDIA hardware than C for CUDA.

In terms of utilizing the GPU, parallel processes are broken up into individual threads. These threads are then assigned to a specified number of computational blocks, the precise number of which is determined by the problem size. However, it is important to note that each computational block is capable of running multiple threads due to the presence of multiple floating point arithmetic logic units (FPALU). We have empirically determined that 256 threads per computational block leads to the best utilization of FPALU's within a given computational block.

The CPU implementation of the pSPE algorithm used standard C++ and relied upon the Visual Studio threading libraries as well the standard Visual C++ 2005 compiler. The main drawbacks of using the Visual Studio compiler rather than the Intel C++ compiler (ICC) were the lack of support for SSE3/SSE4 operations as well as the lack of some more aggressive optimizations present in the ICC. Given the extensive use of distance calculations in the pSPE implementation, SSE3/SSE4-specific code may have led to non-negligible improvements in execution speed. However, we do not believe that speeding up the distance calculation will decrease the execution time by more than 50%.

Similarly to the GPU hardware, the Intel Xeon processor is capable of executing two threads per core. However, we had determined that assigning multiple threads per core was less efficient than running one thread per core due to increased synchronization costs and slightly lower performance of running multiple threads on a single core.

DATA SETS

The performance of our parallel SPE algorithm was tested against three data sets. The first was the vertex data from the Stanford bunny. This data set was obtained from the Stanford 3D scanning repository (http://graphics.stanford.edu/data/3Dscanrep/) and was obtained from scanning a statue of a bunny with a swept-stripe laser triangulation range scanner. The Stanford bunny contains 35 948 data points in R^3. The task was to convert these 3D coordinates into pairwise distances and use those

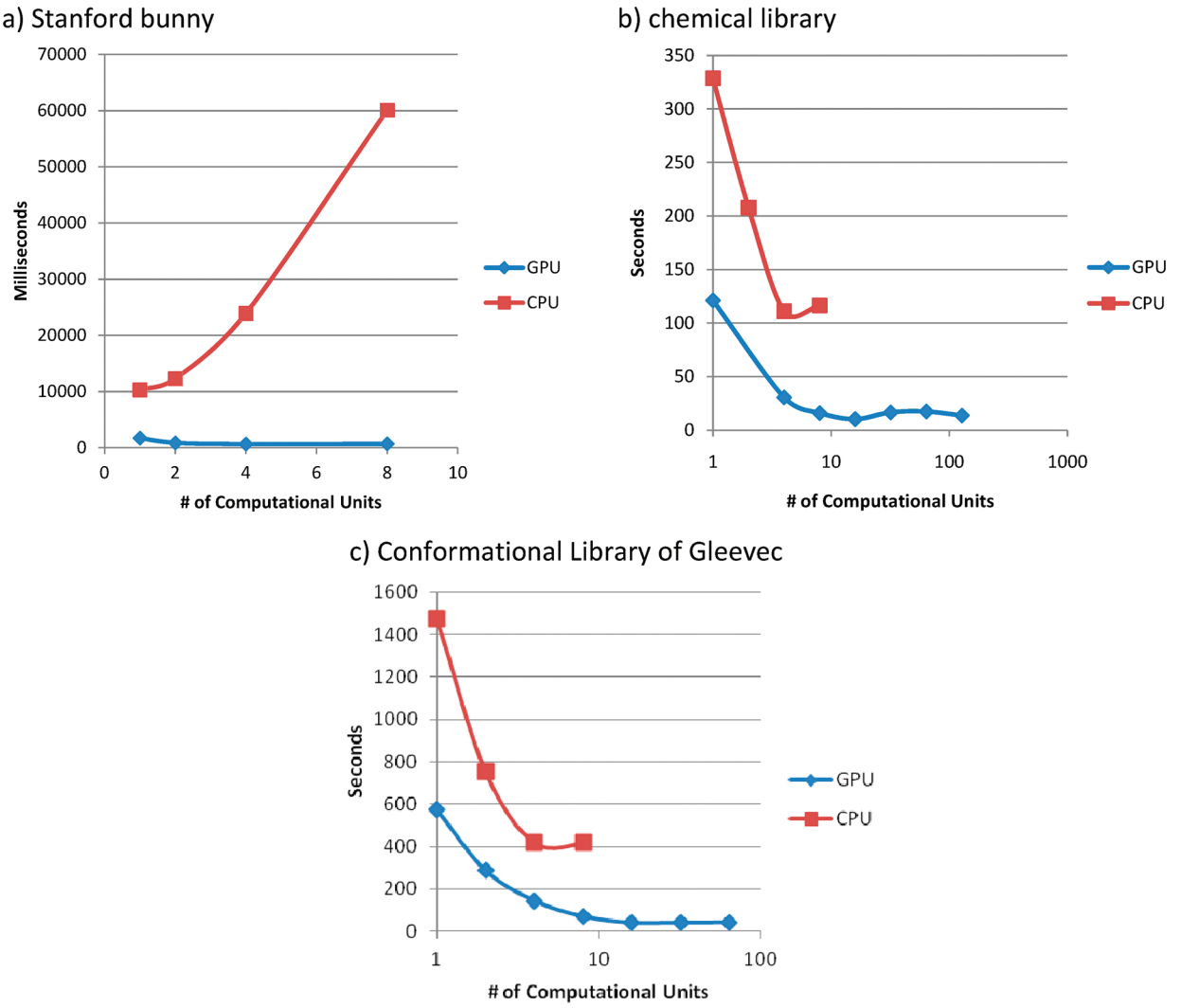


Figure 3. The scaling properties of the multithreaded pSPE algorithm on the different data sets. In the case of the Stanford bunny, we see that while the GPU scales well with the number of computational units, the CPU implementation does not. This suggests that the cost of thread synchronization is greater than the cost of updating the coordinates of the data points. With the chemical library, which is more than an order of magnitude larger, we see that for up to four processors speed-up is nearly linear but falls off after that due to the cost of synchronization. With the GPU implementation, we reach the point of diminishing return at 16 computational units. With the conformational library, we see very similar results as the chemical library, though with an optimal number of computational units being obtained at 16 computational units.

distances to embed the data points back into R^3. If the algorithm works as expected, the resulting image should be indistinguishable from the original one (subject of course to translation, rotation or reflection, since we are starting from random initial coordinates and two mirror images result in identical distance matrices). Though the Stanford bunny is not specifically a MDS problem, it allows us to visually assess the quality of the embeddings and detect problems with different SPE implementations, as seen in Figure 1.

The second data set consisted of a chemical library comprising 907 588 unique molecules based on a slightly larger library originally compiled by Golovin and Henrich and used to test several new substructure search algorithms.[29−31] For each of these molecules, 117 molecular connectivity descriptors were computed[32] and were subsequently normalized in the interval 0−1 and decorrelated into a latent set of 24 orthogonal variables using principal component analysis. The Euclidean distances of the molecules in this 24-dimensional space were used as input distances (dissimilarities) for the SPE embeddings.

The final data set consists of 998 254 conformations of gleevec, a well-known cancer drug. Since gleevec consists of 39 atoms excluding hydrogens, each conformation was described by 117 data points representing the coordinates of each atom in 3D space. Briefly, the conformations were generated via the self-organizing superpositioning (SOS) algorithm developed by our group,[33−35] a technique that defines a set of geometric constraints derived from the molecule's connection table and standard covalent geometry rules and then attempts to generate configurations that satisfy these constraints by iterative local adjustments of atomic coordinates. In SOS, each run starts from a different random initial atomic configuration and as a result generates a conformation that is unrelated to those generated by previous runs. Here, we employed a further enhancement known as conformational boosting,[36] where each run produces a small number of conformations (typically three), each of which is

successively more extended that the previous one (but still stochastically generated). The advantages of the SOS method over alternative conformational sampling techniques have been documented in a number of comparative studies.[37,38] By embedding all the energetically accessible conformations of a given molecule into a lower dimensional space using rmsd as a pairwise measure of similarity, we can visually identify the degree and quality of sampling.

RESULTS

The embeddings obtained by pSPE are shown in Figure 2. In terms of quality, the output of the GPU and CPU implementations were identical, and both behaved as expected. In the output of the Stanford bunny, we see that there is a perfect reproduction of the rabbit showing that the algorithm works exactly as expected. The embedding of the chemical library reveals several clusters that represent families of related molecules with similar descriptors. Performing SPE on the conformations of gleevec yielded a large circular cluster (with some smaller clusters on the outer shell), reflecting the very significant conformational flexibility of that molecule. In terms of the Kruskal stress, the Stanford bunny's final stress was less than 1×10^{-5} after 10 000 iterations, whereas that of the chemical library was 0.08 and the gleevec conformations was 0.29 (which indicates that the embedding is substantially more distorted, that is, that the interconformation similarities are not as well preserved).

However, aside from validating that the output of the multithreaded implementations is correct, of interest is also their performance with varying numbers of parallel processes. Figure 3 shows the run times of the GPU and CPU implementations for different number of computational units. In the case of the GPU, we varied the number of blocks (computational units), whereas in the CPU we varied the number of cores. For the Stanford bunny data set, the single-threaded implementation on the CPU was the fastest, taking approximately 10 s to complete 10 000 iterations, whereas the 8-threaded implementation was the slowest, taking 60 s for the same number of iterations. For the GPU, a single computational block could obtain the final embedding in 1.2 s, and the best performance with 4 blocks would lead to a runtime of 656 ms. With greater number of GPU blocks, we start seeing a slight degradation in performance.

With the chemical library we observe similar behavior but much better scaling characteristics. In the CPU implementation, the embedding of 907 K data points with a single processor takes around 5 min to compute, but with 4 processors, the average processing time can be brought down to around 2 min. With the GPU, we see that running a single block yields a runtime of slightly over 2 min. It was initially surprising to us that a single computational block on the GPU would perform roughly as well as utilizing four cores on the CPU. Utilizing 16 GPU blocks, we found that the runtime could be further reduced to a little over 10 s. Thus, for this moderately sized problem and utilizing only a small subset of computational resources available on the GPU, we were able to obtain a speed-up of 31× over that of a single-threaded application, as was initially implemented, and a speed-up of around 10× over conventional multithreading. In general, it was found that on the GPU having each computational unit handle around 5000 points led to the best performance relative to the synchronization overhead. However, given the fact that only a small fraction of computational units were used in this problem, we believe that with larger problem sizes, the speed advantage of the GPU would only increase.

With the conformational library containing close to a 998 K objects, we have a data set that is close to the same number of data points as the chemical library, though of much higher apparent dimensionality (117 vs 24). This data set allows us to assess the impact of dimensionality upon the speed of the embedding. Utilizing a CPU, a single core is able to compute the embedding in 24 min, and through the use of four blocks, we have been able to reduce the time to 7 min. Running this data set, we find that a single computational block takes on the order of 10 min; by utilizing more blocks, we were able to get a minimum embedding time of 40 s. Though the linear relationship between dimensionality and execution time was not surprising, it is important to verify that there were not any issues, such as synchronization, cache misses, etc., that would have negatively impacted our GPU implementation.

DISCUSSION

In the overall results, we see that the SPE algorithm running on a GPU is 20−30× faster compared to a single-threaded implementation running on a conventional CPU. However, when compared to a multithreaded implementation running on a dual CPU system with 8 cores, the GPU implementation shows a 10× improvement in performance versus the CPU. This is in line with what was previously reported when comparing parallel GPU implementations with multithreaded implementations for the CPU. We believe that for larger data sets, the difference between the CPU and GPU implementations will be even greater. This is because for the size of the data sets which we have chosen to evaluate, we could not take full advantage of all the available computational units of the GPU. For instance, in the case of the Stanford bunny which contains only 32 K points, utilizing the full power of the GPU, 512 blocks running 512 threads, gave more independent threads than data points. In the case of the chemical library, each computational thread would have only been used to process four to five data points depending on the partition. Thus for the size of data sets used in this manuscript, we rapidly reached a point of diminishing returns in terms of adding more computational cores.

The primary problem leading to poor scaling is that our outer loop requires the threads to synchronize prior to the next iteration, and this adds a great deal of overhead to the running of the algorithm. This can be seen in both the CPU and GPU implementations. With data sets under a certain size, it makes no sense to apply more computational resources to the problem. Testing the overhead associated with synchronization, we coded an incorrect implementation that did not require block level synchronization. Without requiring block synchronization at every iteration through the outer loop, the runtime for the bunny data set was on the order of 60 ms, the runtime for the larger chemical library was 300 ms, and the runtime for the chemical conformation library was on the order of 1200 ms. Thus, given the problem sizes which we were encountering, the overhead for synchronization was on the order of 10× of the time that was actually spent doing useful work. While we have not yet developed an algorithm which allows us to do without the thread synchronization, we have at least established that by cutting down the number of block level synchronizations, it may be possible to speed up the algorithm further.

One of the reasons that the CPU implementation did not scale quite as well as the GPU implementation is due to the relatively greater computational expense of thread synchronization of the CPU. This can be seen by the fact that the scaling for a small number of cores is nearly linear for the GPU but not the CPU. In the case of the chemical library, we see that increasing from one to eight blocks, we get an equivalent speed-up in the overall algorithm, whereas in the CPU implementation, though we can see significant speed up through parallelization, diminishing returns for adding additional cores occurs even when moving from one to two processors. Possible reasons for this discrepancy may be the lower performance of Win32 threads versus pthreads. However, even with a corresponding performance in thread synchronization, we see that even comparing single computational units the GPU outperforms the CPU by at least a factor of 3. We hypothesize that this is primarily due to the fact that memory access latency is several times faster on the GPU than on the CPU. In the CPU computations, the accesses are to relatively slow DDR3 memory running at 800 MHz, whereas the GPU has access to much faster DDR5 memory running at 2.0 GHz. Given the random reads associated with each thread, this memory overhead is a significant factor, and thus the superior memory bandwidth of the GPU is what contributes to the increase in performance.

One of the more encouraging aspects of SPE is that even with very large data sets, utilizing the graphics processing units allows us to handle them without a large amount of computational resources. Though we have based our implementation upon the GeForce 580 GTX, a top of the line graphics card, our inability to utilize even a significant fraction of the computational units on this card means that for problems of the same size, pSPE can be successfully run on less powerful graphics processors such as those that are present in most modern laptops. With respect to the multithreaded implementation running on the CPU, we were still able to get a significant speed increase while utilizing multiple cores. However, given that the scaling of CPU cores is more challenging and given the less ideal scaling of the CPU implementation, targeting the GPU appears to be the optimal solution for most applications.

CONCLUSION

Pivoted SPE was initially formulated as an efficient method for deriving lower dimensional embeddings of high-dimensional data sets. While it was shown to be more efficient than the original SPE implementation, it has the additional advantage of being easily parallelized. With minimal work, we were able to create a parallel implementation that would function on a dedicated graphics processor and were able to speed up the execution of the algorithm by more than 30× compared to our original single-threaded implementation and more than 5× compared to a multithreaded implementation on a conventional CPU.

AUTHOR INFORMATION

Corresponding Author

*E-mail: eyang3@its.jnj.com. Telephone: (215) 628-7114.

REFERENCES

(1) Borg, I.; Groenen, P. J. F. *Modern multidimensional scaling: theory and applications*; Springer: New York, 1997.

(2) Kruskal, J. B.; Wish, M. *Multidimensional scaling*; Sage Publications: Beverly Hills, CA, 1978.

(3) Cooper, L. G. A review of multidimensional scaling in marketing research. *Appl. Psychol. Meas.* **1983**, *7*, 427–450.

(4) Mooney, C.; Vullo, A.; Pollastri, G. Protein structural motif prediction in multidimensional phi-psi space leads to improved secondary structure prediction. *J. Comput. Biol.* **2006**, *13*, 1489–1502.

(5) Cox, T. F.; Cox, M. A. A. *Multidimensional scaling*, 2nd ed.; Chapman & Hall/CRC: Boca Raton, FL, 2001.

(6) Yoshio, T.; Forrest, Y.; Jan, L. Nonmetric individual differences multidimensional scaling: an alternating least squares method with optimal scaling features. *Psychometrika* **1977**, *42*, 7–67.

(7) Agrafiotis, D. K. Stochastic proximity embedding. *J. Comput. Chem.* **2003**, *24*, 1215–1221.

(8) Agrafiotis, D. K. A new method for analyzing protein sequence relationships based on Sammon maps. *Protein Sci.* **1997**, *6* (2), 287.

(9) Izrailev, S.; Agrafiotis, D. K. A method for quantifying and visualizing the diversity of QSAR models. *J. Mol. Graphics Modell.* **2004**, *22*, 275–284.

(10) Smellie, A.; Wilson, C. J.; Ng, S. C. Visualization and interpretation of high content screening data. *J. Chem. Inf. Model* **2006**, *46*, 201–207.

(11) Allor, G.; Jacob, L. Distributed wireless sensor network localization using stochastic proximity embedding. *Comp. Comm.* **2010**, *33*, 745–755.

(12) Agrafiotis, D. K.; Xu, H. A self-organizing principle for learning nonlinear manifolds. *Proc. Natl. Acad. Sci. U.S.A.* **2002**, *99*, 15869–15872.

(13) Agrafiotis, D. K.; Xu, H. A geodesic framework for analyzing molecular similarities. *J. Chem. Inf. Comput. Sci.* **2003**, *43*, 475–484.

(14) Farnum, M.; Xu, H.; Agrafiotis, D. K. Exploring the nonlinear geometry of sequence homology. *Protein Sci.* **2003**, *12*, 1604–1612.

(15) Blaney, J. M.; Dixon, J. S. Distance geometry in molecular modeling. *Perspect. Drug Discov Design* **1993**, *1*, 301–319.

(16) Xu, H.; Izrailev, S.; Agrafiotis, D. K. Conformational sampling by self-organization. *J. Chem. Inf. Comput. Sci.* **2003**, *43*, 1186–1191.

(17) Agrafiotis, D. K.; Bandyopadhyay, D.; Carta, G.; Knox, A. J. S.; Lloyd, D. G. On the effects of permuted input on conformational sampling of druglike molecules: an evaluation of stochastic proximity embedding (SPE). *Chem. Biol. Drug Des.* **2007**, *70* (2), 123–133.

(18) Izrailev, S.; Zhu, F.; Agrafiotis, D. K. A distance geometry heuristic for expanding the range of geometries sampled during conformational search. *J. Comput. Chem.* **2006**, *27* (16), 1962–1969.

(19) Zhu, F.; Agrafiotis, D. K. A self-organizing superposition (SOS) algorithm for conformational sampling. *J. Comput. Chem.* **2007**, *28*, 1234–1239.

(20) Agrafiotis, D. K.; Gibbs, A.; Zhu, F.; Izrailev, S.; Martin, E. Conformational sampling of bioactive molecules: a comparative study. *J. Chem. Inf. Model.* **2007**, *47*, 1067–1086.

(21) Bonnet, P.; Agrafiotis, D. K.; Zhu, F.; Martin, E. J. Conformational analysis of macrocycles: finding what common search methods miss. *J. Chem. Inf. Model.* **2009**, *49*, 2242–2259.

(22) Bandyopadhyay, D.; Agrafiotis, D. K. A self-organizing algorithm for molecular alignment and pharmacophore development. *J. Comput. Chem.* **2008**, *29*, 965–982.

(23) Liu, P.; Zhu, F.; Rassokhin, D. N.; Agrafiotis, D. K. A self-organizing algorithm for modeling protein loops. *PLoS Comput. Biol.* **2009**, *5* (8), e1000478.

(24) Klenner, A.; Weisel, M.; Reisen, F.; Proschak, E.; Schneider, G. Docking of flexible molecules into receptor binding sites by ligand self-organization in situ. *Mol. Inf.* **2010**, *29*, 189–193.

(25) Agrafiotis, D. K.; Xu, H.; Zhu, F.; Bandyopadhyay, D.; Liu, P. Stochastic proximity embedding: methods and applications. *Mol. Inf.* **2010**, *29*, 758–770.

(26) Owens, J. D.; Houston, M.; Luebke, D.; Green, S.; Stone, J. E.; Phillips, J. C. GPU computing. *Proc. IEEE* **2008**, *96*, 879–899.

(27) Lee, V.; Kim, C.; Chhugani, J.; Deisher, M.; Kim, D.; Nguyen, A.; Satish, N.; Smelyanskiy, M.; Chennupaty, S.; Hammarlund, P.; Singhal, R.; Dubey, P. Debunking the 100X GPU vs. CPU myth: an

evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News* **2010**, *38*, 451–460.

(28) Sammon, J. W. A nonlinear mapping for data structure analysis. *IEEE Trans. Comput.* **1969**, *18*, 401–409.

(29) Golovin, A.; Henrick, K. Chemical substructure search in SQL. *J. Chem. Inf. Model.* **2009**, *49*, 22–27.

(30) Agrafiotis, D. K.; Lobanov, V. S.; Shemanarev, M.; Rassokhin, D. N.; Izrailev, S.; Jaeger, E. P.; Alex, S.; Farnum, M. Efficient substructure searching of large chemical libraries: the ABCD chemical cartridge. Submitted.

(31) Liu, P.; Agrafiotis, D. K.; Rassokhin, D. N. *Power keys: a novel class of topological descriptors based on exhaustive subgraph enumeration and their application in substructure searching*. Submitted.

(32) Kier, L. B.; Hall, L. H. *Molecular connectivity in structure-activity analysis*. Research Studies Press; Wiley: Letchworth, Hertfordshire, England, 1986.

(33) Zhu, F.; Agrafiotis, D. K. *Self-Organizing Superimposition Algorithm for Conformational Sampling. J. Comput. Chem.* **2007**, *28*, 1234–1239.

(34) Liu, P.; Zhu, F.; Rassokhin, D. N.; Agrafiotis, D. K. A self-organizing algorithm for modeling protein loops. *PLoS Comput. Biol.* **2009**, *5* (8), e1000478.

(35) Xu, H.; Izrailev, S.; Agrafiotis, D. K. Conformational sampling by self-organization. *J. Chem. Inf. Comput. Sci.* **2003**, *43*, 1186–1191.

(36) Izrailev, S.; Zhu., F.; Agrafiotis, D. K. A distance geometry heuristic for expanding the range of geometries sampled during conformational search. *J. Comput. Chem.* **2006**, *27*, 1962–1969.

(37) Agrafiotis, D. K.; Gibbs, A.; Zhu, F.; Izrailev, S.; Martin, E. J. Conformational sampling of bioactive molecules: a comparative study. *J. Chem. Info. Model* **2007**, *47*, 1067–1086.

(38) Bonnet, P.; Agrafiotis, D. K.; Zhu, F.; Martin, E. J. Conformational analysis of macrocycles: finding what common search methods miss. *J. Chem. Info. Model.* **2009**, *49*, 2242–2259.