Portable Parallel Design of Weighted Multi-Dimensional Scaling for Real-Time Data Analysis

Sajal Dash*, Anshuman Verma†, Chris North*, and Wu-chun Feng*†
Department of Computer Science and Department of Electrical and Computer Engineering†*
Virginia Tech, Blacksburg, VA, USA
Email: {sajal, anshuman, north, wfeng}@vt.edu

***Abstract*—Projecting a high-dimensional dataset onto a lower dimensional space can improve the efficiency of knowledge discovery and facilitate real-time data analysis. One technique for dimension reduction, weighted multi-dimensional scaling (WMDS), approximately preserves pairwise weighted distances during the transformation; but its $O(f(n)d)$ algorithm impedes real-time performance on large datasets.**

Thus, we present CLARET, our fast and portable parallel WMDS tool that combines algorithmic concepts adapted and extended from the stochastic force-based MDS (SF-MDS) and Glimmer. To further improve CLARET's performance for real-time data analysis, we propose a preprocessing step that computes approximate weighted Euclidean distances by combining a novel data mapping called *stretching* and Johnson Lindestrauss' lemma in $O(\log d)$ time in place of the original $O(d)$ time. This preprocessing step reduces the complexity of WMDS from $O(f(n)d)$ to $O(f(n)\log d)$, which for large d is a significant computational gain. Finally, we present a case study of CLARET by integrating it into an interactive visualization tool called V2PI to facilitate real-time analytics. To ensure the quality of the projections, we propose a geometric shape matching-based alignment process and a quality metric.

***Keywords*-multi-dimensional scaling (MDS), weighted multi-dimensional scaling (WMDS), OpenCL, visual analytics**

I. INTRODUCTION

The representation of complex scientific data often materializes into points in high-dimensional space. Statistical methods for analyzing these high-dimensional data points are computationally expensive, rendering it infeasible to draw any statistical inferences in a reasonable amount of time. Dimension reduction is an essential computational method for making the data comprehensible. Multi-dimensional scaling (*MDS*) and its extension, weighted MDS (*WMDS*), are popular approaches for dimension reduction.

A. MDS in Science and Visualization

MDS is a tool of choice for many applications. Psychologists use MDS to study the relationship between different stimuli, where each stimulus is a multi-dimensional data point [1]. Biologists use MDS for many applications, including sequence alignment, protein substructure search, and RNA microarray analysis [2].

For visual analytics, high-dimensional data points are projected onto two- or three-dimensional space so that scientists can more easily explore these points. Sometimes, the users inject their domain knowledge through various interactions. This domain knowledge is then utilized to refine the visualization. WMDS introduces weights on different dimensions to enable users to explore a space of projections. For example, Leman et al. [3] used WMDS to create 2D-embedding by translating visual interactions into dimensional weights.

B. MDS for Real-time Visual Analytics

For interactive and real-time visual analytics, it is imperative that MDS runs in real-time on available computing devices. Python Scikit MDS uses the SMACOF [4] method, which requires computing $n^2/2$ pairwise distances. This approach is problematic because storing $n^2/2$ distances in memory can slow down overall system performance. For a dataset of size 683×9, Scikit-MDS [5] takes $30-50$ seconds. So, it is not suitable for real-time interactive visualization. The same is true for virtually every sequential MDS method. While parallelized GPU implementations of MDS exist, they require NVIDIA GPU cards. We aim to develop a fast and portable MDS implementation that can run in parallel on available parallel hardware, such as multi-core CPUs, MICs, or GPU cards made by any vendor.

C. Our Contribution

We created Claret, a parallelized and portable force-based WMDS. We ported and extended Chalmer's stochastic force-based MDS (SF-MDS) [6] to OpenCL, which runs on various platforms, including multi-core CPUs, GPUs, and FPGAs. To support the incremental nature of interactive visualization, we then extended Glimmer's multi-level algorithm to use our OpenCL-based stochastic force calculation.

For high-dimensional data points, the weighted Euclidean distance computation is a bottleneck even for parallel hardware. We prove that with a combination of a new mapping of data points (*Stretching*) and Johnson Lindenstrauss' lemma [7] that we can preserve weighted Euclidean distances and expedite distance computation.

To enable stable visualizations over time, we propose a method to quantify the quality of a layout, compare two embeddings quantitatively, and align embeddings.

II. BACKGROUND

Dimension-reduction tools preserve some essence of the high-dimensional data such as the pairwise dissimilarity and variance. Popular dimension-reduction techniques include principal component analysis (PCA), multi-dimensional scaling (MDS), and linear discriminant analysis (LDA). PCA [8] reduces the dimension of the data by choosing directions along which the total variance of projected data points is maximized. MDS [9] preserves pairwise distances, which is a measure of dissimilarity. Weighted multi-dimensional scaling (WMDS) preserves pairwise weighted distances. LDA [10] finds a linear combination of features to reduce the dimension, and it preserves the class discrimination between points.

Assume there are n points in $\mathbb{R}^d$ and they are represented as a collection of n d-dimensional vectors. W is a d-dimensional weight vector. The projection of these high-dimensional data points onto $2D$ space, L is n two dimensional vectors. H, W, L can be written as $\begin{pmatrix}\{h_{1,1}\dots h_{1,d}\}\\ \vdots \\ \{h_{n,1}\dots h_{n,d}\}\end{pmatrix}$, $\begin{pmatrix}w_1\\ \vdots \\ w_d\end{pmatrix}$, and $\begin{pmatrix}\{l_{1,1}, l_{1,2}\}\\ \vdots \\ \{l_{n,1}, l_{n,2}\}\end{pmatrix}$.

MDS projects data by minimizing some variant of a stress measure, as depicted in equation 1.

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}\left(\sqrt{\sum_{k=1}^{d}(h_{i,k}-h_{j,k})^2}-\sqrt{\sum_{k=1}^{2}(l_{i,k}-l_{j,k})^2}\right)^2 \tag{1}$$

WMDS, on the other hand, preserves weighted high-dimensional distances between points in a low-dimensional space. It minimizes a slightly different stress function, which uses W while computing high-dimensional distances. Stress for WMDS is defined as follows:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}\left(\sqrt{\sum_{k=1}^{d}(h_{i,k}-h_{j,k})^2 w_k}-\sqrt{\sum_{k=1}^{2}(l_{i,k}-l_{j,k})^2}\right)^2 \tag{2}$$

A formal definition of WMDS follows:

Definition II.1 (WMDS)**.** *Given high-dimensional data* $\mathbf{H}$*, and a dimensional weight vector* W*, find a two-dimensional embedding* $\mathbf{L}$ *that minimizes the stress in equation 2.*

III. Related Work

We present the state of the art of MDS tools, focusing mainly on GPU-based tools. We also briefly discuss random projection and layout matching.

A. Force-Directed MDS

We can view the projection of high-dimensional points onto low-dimensional space as a layout optimization problem. Minimizing the stress function narrows the difference between the high-dimensional distance and low-dimensional distance for all pairs. If we start with an initial random layout, i.e. *2D* projection, and guide the points to move around while lowering the difference between distances in two spaces, it will eventually converge to the optimal layout.

In the n-body problem, every point exerts forces on all other $n-1$ points depending on some measure such as mass and distance in the gravitational force field and charge and distance in the electrostatic force field. In layout computation, a given point needs to move towards or further from any other point in the embedding. The amount of these movements depends on how closely their distance in the current layout matches with their high-dimensional counterpart. If we are to attach a force between these two points, it should be proportional to the difference between these two distances.

Thus, the layout computation problem maps to an n-body problem when we apply a force between two points i and j proportional to the measure, $LD(i,j) - HD(i,j)$. If $LD(i,j) - HD(i,j) < 0$, the force is repulsive and it will move the two points apart. If $LD(i,j) - HD(i,j) > 0$, the force is attractive and this will move the two points closer. For the i^{th} point, the total experienced force is

$$F_i = \sum_{j \neq i} K \times (LD(i,j) - HD(i,j))$$

Here K is a constant that we can tune for the dataset and simulation environment. Once we compute force for a point in *2D*, we can estimate acceleration, which is proportional to the force. This acceleration can be used to calculate current velocity, and in turn, the point's next position. Velocity is updated using $v = v_0 + a \times \delta t$, and the next position is updated using $x = x_0 + v \times \delta t$. Here, δt is simulation step. $\langle v_0, x_0\rangle$ and $\langle v, x\rangle$ are the velocity and position at the beginning and at the end of the current timestamp.

Force computation at each step is a $O(n^2)$ operation since we have to compute $O(n^2)$ distances in $2D$. This estimate also assumes that we pre-compute all the high-dimensional distances; otherwise, this computation becomes $O(n^2 \times d)$ operations.

We can compute forces, velocities, and positions of n points independently. There are multiple implementations that use spring-force simulation as a means to perform MDS. Since we can map MDS to an n-body problem, a well-studied and optimized problem in GPU computing, we take this approach as the core of Claret.

B. Stochastic Force-based MDS(SF-MDS)

Force-based MDS can be computed in parallel using many cores; however, the computation workload per thread is still large. Assuming we have n parallel threads at our disposal, the i^{th} thread will compute F_i which is a summation of forces exerted by the $n-1$ other points. For large n, each thread might take a long time to finish its computation.

Chalmers et al. [6] made an observation that we can perform force simulation with much less effort using two small representative sets from the $n-1$ points. In their algorithm, they maintain two sets, a "near set" of size s_n and a "random set" of size s_r. The near-set gradually converges to contain the nearest s_n points the while random set always picks s_r new points at every simulation step. The near-set expedites convergence by providing local structure information, and the random set helps the embedding by enforcing global structure. The sizes used by Chalmers' algorithm are 14 and 10, respectively, which were determined empirically. Though Chalmers' version is a sequential one, we can leverage the fact that n-body simulation can be performed in parallel, and reduce the per-thread workload to a constant amount. This observation also helps us with the stress computation. Instead of adding $O(n^2)$ differences, we can add $O(n)$ distances to approximate the stress.

C. GPU-based MDS

Several realizations of GPU-based *MDS* exist. The field of bioinformatics has produced quite a few GPU-based MDS tools in CUDA, a NVIDIA-specific GPU programming language. Fester et al. [11] implemented a CUDA version of HiT-MDS by employing reduction to add a large group of numbers and computing multi-dimensional distances in parallel. CUDA-based fast multidimensional scaling (CFMDS) [2] dynamically decides whether to run MDS on the entire dataset or divide the data into chunks that can fit into the global memory of the NVIDIA GPU card depending on the input size.

D. Multi-level SF: Glimmer

Glimmer by Ingram et al. [12] uses stochastic force as the base algorithm for their force-based MDS, and they implemented this on a GPU using OpenGL, primarily a graphics programming language. One major contribution of their work is that they optimize the layout at multiple levels. Glimmer divides the dataset into $log_b n$ levels, data ranges in these levels are $[0, max(MIN\_SIZE, \frac{n}{b^{\log_b n}})], \ldots, (\frac{n}{b^2}, \frac{n}{b}], (\frac{n}{b}, n]$, where b is a constant called the decimation factor. Glimmer uses three operators at each level: restrict, relax, and interpolate. The *restrict* operator samples points for that range. *Relax* runs stochastic force to find optimal embedding for all points up to the previous level. *Interpolate* uses all relaxed points up to the previous level to sample the near and random set to run stochastic force on the data at the current level. In the last level, relaxing all data produces the final embedding.

IV. Claret

To develop Claret, we parallelize the sequential MDS algorithm SF-MDS and port the parallelized version into parallel hardware using OpenCL. Claret also uses Glimmer's multi-level approach to obtain faster convergence.

Continuing from sub-section III-B, SF-MDS is a linear approximation of force-based MDS. Instead of computing force from all $n-1$ points, SF-MDS uses two small subsets to do so. At every iteration, the near set is updated by choosing the s_n nearest points (according to their high dimensional distances from the point under consideration) from $pivot\_size = s_n + s_r$ pivot points. We summarize SF-MDS in Algorithm 1, where $f()$ and $g()$ are linear functions to allow tuning simulation parameters.

Algorithm 1 Force Based MDS

```
Require: $highD[n \times d]$, $lowD[n \times 2]$, $velocity[n \times 2]$,
    $force[n \times 2]$ , $pivots[n][pivot\_size]$
 1: while $converge() \neq true$ do
 2:     for $i = 0 \to n$ do
 3:         $near\_set \leftarrow pivots[i][0 \ldots near\_set\_size]$
 4:         $random\_set \leftarrow randomindices$
 5:         $my\_pivots \leftarrow near\_set \cup random\_set$
 6:         for $j = 0 \to pivot\_size$ do
 7:             $k \leftarrow my\_pivots[j]$
 8:             $hDistance[k] \leftarrow dist(highD[i], highD[k])$
 9:             $lDistance[k] \leftarrow dist(lowD[i], lowD[k])$
10:             $force+ = f\big(hDistance[k] - lDistance[k]\big)$
11:         sort $my\_pivots$ based on $hDistance$
12:         $a \leftarrow g\big(force\big)$
13:         $velocity[i] \leftarrow velocity[i] + at$
14:         $lowD[i] \leftarrow lowD[i] + velocity[i] \times t$
15:         $pivots[i] \leftarrow my\_pivots$
```

We want to parallelize algorithm 1 for efficient implementation and fast execution on any OpenCL supported devices. In OpenCL architecture, we have two types of computing devices: host and device. The host is usually a CPU which can launch parallel programs into devices and act as the moderator and controller. The host has host memory, and the device has a hierarchy of memory consisting of global memory, constant memory, and local memory. We load the input data into host memory; the host program then launches parallel programs in SIMD fashion into computing units of one or more devices.

Given the sequential algorithm as depicted in algorithm 1, the goal is to develop a parallel program that achieves similar functionalities and can run on any computing device having the parallel computing architecture specified by the OpenCL standard.

A. Porting SF-MDS to GPU using OpenCL

Any n-body problem can be parallelized across n points. Every point experience force from all other (or in the case of SF-MDS, a subset of) points which are frozen in time and space. At the beginning of every iteration, each point sees the same configuration ($\langle position,\ velocity,\ \ldots\rangle$) of points. In SF-MDS, every point experiences force from $pivot\_size = O(1)$ other points. So, we don't unroll the loop for iterations over time; instead, we unroll/parallelize the outer for loop in line 2 as every point can be processed independently in a given duration of δt.

So, every iteration for every point runs in $O(1) = O(n)/n$ time. The code block consisting from line 3 through line 15 computes the force and updates the position for a given point. We can take this block and put it inside a computing unit, namely thread, to take care of individual points in parallel. Algorithm 2 shows a high-level OpenCL kernel of Claret. At every iteration, n such kernels are launched to update the positions of n points in parallel.

Algorithm 2 Claret Kernel

```
1: $gid \leftarrow get\_global\_id(0)$
2:
3: //copy data from global memory
4: $near\_set \leftarrow pivots[i][0 \ldots near\_set\_size]$
5: $random\_set \leftarrow randomindices$
6: $my\_pivots \leftarrow near\_set \cup random\_set$
7: $v_0 \leftarrow velocity[gid]$
8: $x_0 \leftarrow lowD[gid]$
9:
10: //compute force using pivot points
11: for $j = 0 \rightarrow pivot\_size$ do
12:     $k \leftarrow my\_pivots[j]$
13:     $hDistance[j] \leftarrow dist(highD[gid], highD[k])$
14:     $lDistance[j] \leftarrow dist(lowD[gid], lowD[k])$
15:     $\delta v \leftarrow v_0 - velocity[k]$
16:     $force+ = f\big(hDistance[k] - lDistance[k], \delta v\big)$
17: sort $my\_pivots$ based on $hDistance$
18: globalSynchronization()
19:
20: //update velocity and position
21: $a \leftarrow g\big(force\big)$
22: $v \leftarrow v_0 + at$
23: $x \leftarrow x_0 + v \times t$
24: //copy data back to global memory
25: $velocity[gid] \leftarrow v$
26: $lowD[gid] \leftarrow x$
27: $pivots[gid] \leftarrow my\_pivots$
28: globalSynchronization()
29:
30: //Compute low dimensional distances
31: for $j = 0 \rightarrow pivot\_size$ do
32:     $k \leftarrow my\_pivots[j]$
33:     $lDistance[j] \leftarrow dist(lowD[gid], lowD[k])$
```

There are some implementation/porting challenges which can cripple the performance on different accelerators in OpenCL programming paradigm. We address some of these issues in the remaining part of this section.

B. Memory and Data Management

Solving an n-body problem for large data requires storing a significant amount of data in memory, efficient access to that memory, and minimal data transfer between the host and device.

Storing data into memory: We give an estimate of the in-memory data storage requirement during a single iteration in table I. For the purpose of demonstration, we set $pivot\_size$ to 8.

Buffer	Purpose	Size	Type
highD	high dimensional data	$n \times d$	float
lowD	2D projection	$n \times 2$	float
pivot_indices	Near and Random index	$n \times 8$	unsigned int
hd_distances	HD distances to pivots	$n \times 8$	float
ld_distances	LD distances to pivots	$n \times 8$	float

Table I: The comprehensive list of required memory.

We have to store around $(22 + d) \times n$ floating point numbers in device memory. So, even for an input data as big as $10^6 \times 100$, the required memory is around $500MB$, which can easily fit in the device memory of modern hardware accelerators.

Since pre-computing $O(n^2)$ distances requires a large amount of device memory, we compute distances on the fly. That also helps avoid moving a great deal of data between the host and device.

Low latency in memory access: We coalesced memory access so that whenever possible, the compiler can resort to vector operation. Before the first iteration, we pre-compute $n \times pivot\_size/2$ pivot indices in the range $[0, n)$ and offload the whole data into device's global memory. At every iteration, for every point we generate a random starting point as $si = f(global\_id,\ iteration)$ and access $pivot\_indices[si \ldots si + pivot\_size/2]$ as new random points.

Minimum Data Transfer: Moving data back and forth between the host and device is a time-consuming task. So, we move almost the entirety of the data at the beginning of the first iteration to device global memory, and then between iterations, we only fetch a constant sized data from the device to host.

C. Global Synchronization and Kernel Fusion

At any given iteration of force simulation, every point sees the same configuration, the same high and low-dimensional positions. We want to ensure consistent access to global memory shared by all threads.

In algorithm 2, during a given iteration (same time window), all threads access $lowD$ 4 times in lines 9, 14, 26, and 33. Except for the third access, all other accesses are read accesses. These accesses have a deterministic order, let's call them $R1$, $R2$, $W1$, and $R3$. $W1$ is a write access that can create a data race between threads if the threads do not synchronize before and after this step. So, we put two global synchronization points in the kernel.

Unfortunately, OpenCL does not directly support global synchronization. Furthermore, the global synchronization mechanism offered by Xiao et al. [13] is not viable because OpenCL cannot globally synchronize across workgroups.

Instead, we break the workflow of a single thread across three kernels at the points of required global synchronization. Between two kernel calls, control returns to the host, and thus, all threads get a chance to synchronize globally.

While this approach ensures the correctness of our parallelization, the overhead of coming back to CPU is non-trivial. So, we seam the kernels back together using kernel fusion through double buffering. We maintain two buffers for one array, and at any given step, all threads read from the same buffer and write to the other buffer. This method ensures $W1$ does not create any inconsistency in values read by different threads.

D. Computing Distance in Parallel

Every thread needs to compute $pivot\_size$ pairwise distances, each of them is an $O(d)$ task if computed sequentially within the thread. Ideally, each pairwise distance computation comes down to reducing d values into 1 value. Each main thread is launching $(pivot\_size \times d)$ threads to reduce $pivot\_size$ values. This mechanism is known as dynamic parallelism, and only a handful of GPU cards support this. Since we do not want to restrict Claret to run on only a selected few accelerators, we solve this in software.

We can break the kernel into three segments for three tasks. Each thread will run in parallel to compute pivot indices and then they will sync. After that, we combine all threads' reduction jobs into one big reduction job. Here $(n \times pivot\_size \times d)$ values will be reduced to $(n \times pivot\_size)$ values by $(n \times pivot\_size \times d)$ reduction threads.

Once the reduction threads finish, all distances for all regular threads are completed and available. Now, each regular thread can resume computing forces and positions for the points for which they are responsible.

E. Stress Computation and Convergence

At every iteration, after every point's position is updated, we compute stress by a reduction in the accelerator. We smooth the stress curve by taking moving average, and we use Cauchy Convergence test for deciding termination. In each level, the stress starts from a high point and eventually plateaus.

V. Quantifying Layout Similarity

Embeddings created by different MDS implementations or the same implementation in different phases might appear dissimilar. We want to investigate similarity between such embeddings.

To quantify the similarity between embeddings, we propose a method based on geometric shape matching. Let point set P consist of n points in d-dimensional space. Let L_1, L_2 be two projections created by MDS. We want to quantify the similarity between these two.

If two embeddings are similar under rotation and translation, we align them using center of mass and principal components. First, we compute centers of mass C_1, C_2 of L_1, L_2 respectively. Then, we apply $C_2 - C_1$ translation to L_1 so that their centers coincide. $L_1' = L_1' + C_2 - C_1$. We compute their first principal components $\vec{v_1}, \vec{v_2}$ for L_1', L_2 and the angle θ between $\vec{v_1}, \vec{v_2}$. Finally, we apply θ rotation to L_1' so that $\vec{v_1'}$ and $\vec{v_2}$ are aligned.

To compute similarity between the layouts L_1, L_2, we then pick n corresponding point pairs (p_i, q_i) where $p_i \in L_1'$ and $q_i \in L_2$. We then compute n Euclidean distances between points in each pair, and take their average to get the final score. Formally,

$$similarity(L_1, L_2) = \sum_{\substack{i = 1 \to n \\ p_i \in L_1', q_i \in L_2}} dist(p_i, q_i) \tag{3}$$

The alignment procedure can be used to stabilize $2D$ projections in different phases of interactive visualization.

VI. Stretched Random Projection

In WMDS, we have to compute $O(n^2)$ weighted Euclidean distances in $\mathbb{R}^d$ which requires $O(n^2 d)$ operations. We propose a way to cut this computation.

A. JL Lemma for Weighted Euclidean Distance

The compute kernels compute pairwise distances on the fly To reduce global memory usage. Each distance computation is a $O(d)$ task which can slow down the program for large $d (d \approx n)$.

We solve this problem using a result from geometry. Johnson–Lindenstrauss lemma [7] states that if we project a high dimensional dataset onto a randomly chosen subspace of much smaller dimensions, it preserves the Euclidean distances approximately. Since *WMDS* requires retaining weighted Euclidean distances, we extend JL lemma to prove that similar result can be achieved for weighted Euclidean distance as well.

Definition VI.1. *Given a set of n points in $\mathbb{R}^d$, and a parameter $\epsilon > 0$, a projection of P onto a random k-dimensional linear subspace, a distance $||p - q||_2$ is ϵ-preserved if $(1 - \epsilon)||p - q||_2 \leq \sqrt{d/k}||f(p) - f(q)||_2 \leq (1 + \epsilon)||p - q||_2$.*

Theorem VI.1. ***Johnson-Lindenstrauss Lemma*** *Let P be a set of n points in $\mathbb{R}^d$, let $\epsilon > 0$ be a parameter, and let $k = (1/\epsilon^2)\log n$. Let Q be the projection of P onto a random k-dimensional linear subspace. Then all pairwise Euclidean distances in P are ϵ-preserved by the corresponding pairwise Euclidean distances in Q with probability at least $1/2$.*

Definition VI.2. *Given a point set P in $\mathbb{R}^d$ and a dimensional weight vector W, the weighted Euclidean distance between two points p, q in P is $||p - q||_{w2} = \left(\sum_{i=1}^{k} w_i(x_{pi} - x_{qi})^2\right)^{1/2}$.*

Definition VI.3. *Stretching* $p = [x_{p1}, x_{p2}, \dots x_{pd}]^T$ *by* $W = [w_1, w_2, \dots, w_d]^T$ *is scaling* p *by* w_i *along* i^{th} *dimension for* $i \in \{1, 2, \dots, d\}$ *so that* $p' = [\sqrt{w_1} \times x_{p1}, \dots, \sqrt{w_d} \times x_{pd}]^T$. *Formally,* $p' = p \ominus W = [\sqrt{w_1} \times x_{p1}, \dots, \sqrt{w_d} \times x_{pd}]^T$. *A point set* P *is stretched by* W *if every point* $p \in P$ *is stretched by* W.

Theorem VI.2. ***Johnson-Lindenstrauss Lemma for weighted Euclidean distance*** *Let* P *be a set of* n *points in* $\mathbb{R}^d$, W *is* d*-dimensional weight vector, let* $\epsilon > 0$ *be a parameter, and let* $k = (1/\epsilon^2) \log n$. *Let* Q *be the projection of* $P \ominus W$ *onto a random* k*-dimension linear subspace. Then, all pairwise weighted Euclidean distances in* P *are* ϵ*-preserved by the corresponding pairwise Euclidean distances in* Q *with probability at least* $1/2$.

Proof: Let, $p' = p \ominus W$ and $q' = q \ominus W$. Both, p' and q' are points in $\mathbb{R}^d$ and they can be mapped to their objects p and q. By Theorem VI.1, $||p' - q'||_2$ is ϵ-preserved after projecting $P \ominus W$ onto k-dimensional linear subspace.

$$\begin{aligned}
||p' - q'||_2 &= ||p \ominus W - q \ominus W||_2 \\
&= \Big(\sum_{i=1}^{k} (\sqrt{w_i} x_{pi} - \sqrt{w_i} x_{qi})^2 \Big)^{1/2} \\
&= \Big(\sum_{i=1}^{k} w_i (x_{pi} - x_{qi})^2 \Big)^{1/2} \\
&= ||p - q||_{w2}
\end{aligned} \tag{4}$$

According to Theorem VI.1, $(1 - \epsilon)||p' - q'||_2 < \sqrt{d/k}||f(p') - f(q')||_2 < (1 + \epsilon)||p' - q'||_2$. We know from equation 4, $||p' - q'||_2 = ||p - q||_{w2}$.
So, $(1-\epsilon)||p-q||_{w2} \leq \sqrt{d/k}||f(p')-f(q')||_2 \leq (1+\epsilon)||p-q||_{w2}$. Hence, the weighted Euclidean distance between two points in P are preserved by the corresponding Euclidean distance in Q with probability at least $1/2$.

B. Computing Distance using Random Projection

To compute pairwise weighted Euclidean distances in P, we first compute $P \ominus W$. This can be accomplished by multiplying $(n \times d)$- matrix P with a $(d \times d)$ diagonal matrix W_D, where $W_D[i, i] = w_i$. $P' = P \ominus W = P \times W_D$.

Then, we will project P' onto k-dimensional linear subspace by multiplying P' with a $d \times k$ dimensional random matrix R. So, the projected point set $Q = P' \times R = P \times W_D \times R$.

Once we have computed Q, we will compute pairwise Euclidean distances in Q, and they will be ϵ-approximation of the weighted Euclidean distances in P.

C. Preprocessing Time

Construction of W_D takes $O(d)$ time. $P \ominus W$ takes $O(nd)$ time since W_D is a sparse matrix. R can be a sparse matrix [14] with only $1/3$ non-zero entries. So, the overall runtime for this preprocessing step is $O(n^2 + n \log n + n^2 \times logn)$ = $O(n^2 \log n)$.

For large d, we use this result to create $H' \in \mathbb{R}^{O(\log n)}$ and then run MDS on H'. The saved computation in MDS for the reduced dimension is enough to pay for this preprocessing step.

VII. Result and Discussion

The two main foci of this undertaking of implementing WMDS in OpenCL are:

1) to be able to run this on multiple accelerator types.
2) make the runtime fast enough for interactive visual analytics.

In this section, we will first start with demonstrating the correctness of our implementation by comparing the embedding for several datasets produced by Claret against that of Scikit-MDS and Glimmer. Next, we will show the runtime performance on various accelerators with different configurations.

We use a range of datasets from different sources.

1) **Cancer**: This dataset contains information regarding breast cancer patients. There are 683 patients each with 9 features.
2) **Shuttle**: This dataset is collected from NASA. It contains 43500 data points about shuttle turn correlation, and each data point has 9 features.
3) **Supreme Court Ruling**: We constructed two datasets of size 14000×20 and 14000×100 by running topic analysis on supreme court rulings.
4) **Artificial Data**: We generate artificial data from 10 20-variate normal distributions. These datasets will have 10 clusters with 20 dimensions.

A. Layout Validation

We get a similar output irrespective of our choice of the accelerator for running Claret. We compute embeddings for the same datasets using Claret on GPU, Scikit-MDS on CPU and Glimmer on GPU. As we can see in figure 1, the Claret's embedding is comparable with the Glimmer and Scikit-MDS's embedding. Though the later was only able to compute embedding for the smallest dataset. These embeddings are visually similar. We also compared stress of the embeddings, and the stresses are within $\pm 5\%$ of each other.

B. Performance Baseline

We want to compare Claret's performance against a sequential MDS tool such as Scikit-MDS. As anticipated in earlier sections, Scikit-MDS can not compute embeddings for larger datasets. It ran out of memory for Shuttle and SC Ruling data even in a system with 8GB of memory. So, we compare Claret's performance against that of Scikit-MDS's on the Cancer dataset.

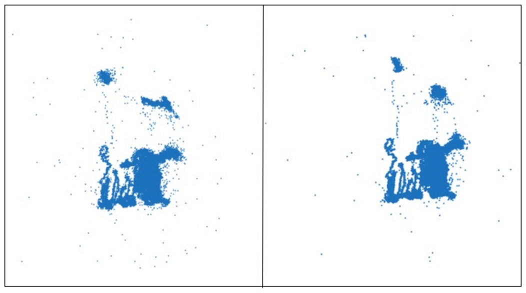


Figure 1: Embeddings produced by Claret(left) and Glimmer(right). The embeddings are visually similar as well as their stresses are within $\pm 5\%$ of each other.

We could run Scikit-MDS only on Cancer dataset (683×9); it took 15s even with 4 parallel threads. Claret took 310ms. We ran both tools in Xeon-E5 with 4 CPU cores.

C. *Comparison Against other GPU-based MDS Method*

There are several GPU-based MDS tools; CFMDS and Hit-MDS are implemented in CUDA and Glimmer are implemented in OpenGL. CFMDS has a dependency on CULA which is discontinued, and Hit-MDS is implemented using very old version of CUDA. Osipyan et al. [15] reported that Glimmer is faster than CFMDS and Hit-MDS. So, we compare Claret's performance against Glimmer's performance on NVIDIA Titan X GPU card. From Figure 2, we see that Claret is $(3-9)X$ faster than Glimmer.

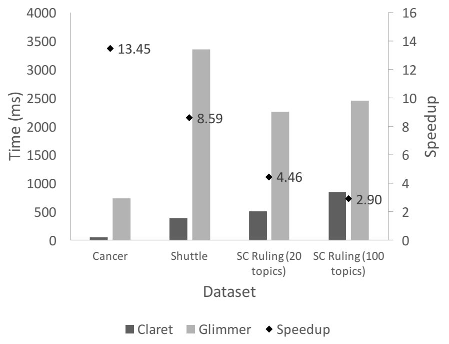


Figure 2: Claret is 3-9 times faster than Glimmer depending on the dataset.

We also experimented with artificial data to see whether the performance is dependent on the values of n. Figure 3 shows the result. Claret's speedup compared to Glimmer is in the range $6.28X - 1.45X$. As the data size increases, the speedup decreases.

D. *Running on Various Accelerators*

The crux of our motivation is portability – the ability to run on many different kinds of hardware. We show that Claret runs on 4 different accelerators including:

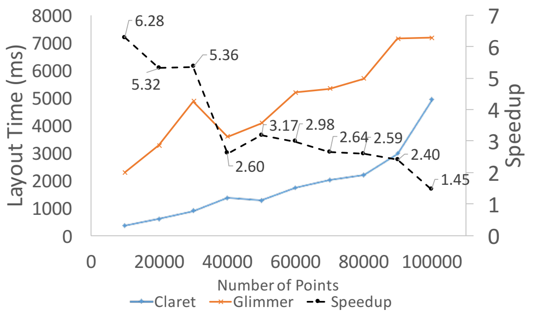


Figure 3: Layout computation time for Claret and Glimmer.

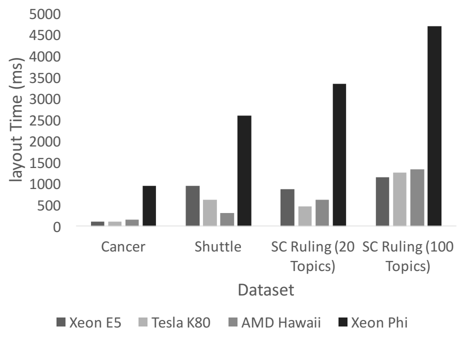


Figure 4: Layout computation time in different accelerators

1) 22-core Xeon E5-2637 CPU @3.50GHz by Intel
2) Tesla K80(Kepler) GPU by NVIDIA with 2496 cores
3) Hawaii GPU by AMD
4) Xeon Phi accelerator by Intel with 61 cores

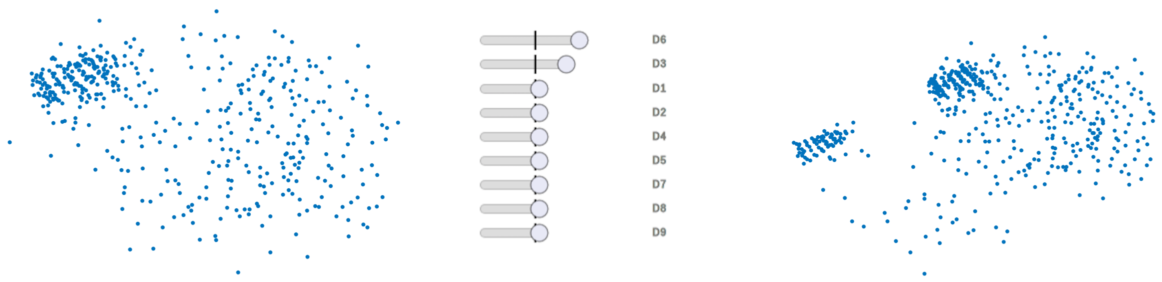
From Figure 4, CPU, and GPU performances are comparable because we have used a powerful CPU with 22 cores. The poorer performance on Xeon Phi in contrast to GPUs suggests that we should take individual core's parallelizability and computing power into consideration when designing parallel tasks.

VIII. Case Study: Interactive Visual Analytics

Visual - Parametric Interaction (V2PI) [3] is a new approach to human-in-the-loop analytics. Instead of relying on MDS to produce the visual layout and then allowing users to tune the weight parameters afterward, V2PI lets users be part of the visualization creation process. The interaction has two directions:

1) Forward: Users set the weights for each dimension/feature by moving sliders. The underlying WMDS computes the $2D$ embedding.
2) Inverse: V2PI translates users' visual feedback into a set of weights that justifies the user-defined arrangement. These weights are used to re-compute a 2D embedding of the entire dataset.

As we can see in Figure 5, the user moves two sliders to change weights of two features, and the resulting projection is displayed.



(a) Initial layout (b) t = 0s, moved sliders (c) t = 0.061s, refined layout

Figure 5: A snapshot of Web Andromeda using Claret. User moves the sliders to up-weight two features and within 61ms Claret recomputes the layout. A new structure emerges in the redrawn layout.

A. *Integrating Claret into V2PI*

A software named Andromeda [16], [17] facilitates V2PI interactions. Web Andromeda is a tool implemented in JavaScript, and an underlying Python back-end drives the computation. This framework presents a problem with using Claret as the host code is written in C++. We use PyOpenCL [18], a Python wrapper library that allows developers to write host code in Python. Since we do not have to change the device kernel, we also provide Python host-side code. With our Python backed host-side code, Claret is then easily called from Web Andromeda's python back-end.

B. *Performance boost-up by Claret*

With Scikit-MDS, a forward interaction for data of size 683×9 took $30 - 50s$. This time is too large for a real-time application. With Claret, the same interaction took $55ms$, which is well below the typical time that humans perceive as a delay ($100ms$ is a standard guideline for real-time response in interactive systems [19]).

IX. Conclusion

We presented Claret, a tool for weighted MDS implemented in OpenCL, which outperforms Glimmer, the previously best performing method designed for GPUs. We show that Claret is indeed a write-once-run-anywhere tool and can run on a plethora of devices. Also, we presented a geometric result claiming that weighted Euclidean distances can be preserved through *stretched*-random projection. The proposed method in this paper for quantification of quality of an embedding has the potential to make visual analytics consistent. We have successfully demonstrated Claret's portability across various accelerators.

The source code and documentation are available at https://github.com/sajal-vt/Claret.

Acknowledgements

This research was supported in part by NSF grant IIS-1447416, a grant from General Dynamics Mission Systems, and a seed grant from the SEEC Center, supported by ICTAS at Virginia Tech.

References

[1] M. L. Davison, "Introduction to multidimensional scaling and its applications," *Applied Psychological Measurement*.
[2] S. Park, S.-Y. Shin, and K.-B. Hwang, "Cfmds: Cuda-based fast multidimensional scaling for genome-scale data," *BMC bioinformatics*.
[3] S. C. Leman, L. House, D. Maiti, A. Endert, and C. North, "Visual to parametric interaction (v2pi)," *PloS one*.
[4] J. De Leeuw and P. Mair, "Multidimensional scaling using majorization: Smacof in r," *Department of Statistics, UCLA*.
[5] "Manifold learning," http://scikit-learn.org/stable/modules/generated/sklearn.manifold.MDS.html.
[6] M. Chalmers, "A linear iteration time layout algorithm for visualising high-dimensional data," in *Visualization'96. Proceedings*.
[7] W. B. Johnson and J. Lindenstrauss, "Extensions of lipschitz mappings into a hilbert space," *Contemporary mathematics*, vol. 26, no. 189-206, p. 1, 1984.
[8] B. Moore, "Principal component analysis in linear systems: Controllability, observability, and model reduction," *IEEE transactions on automatic control*.
[9] W. S. Torgerson, "Multidimensional scaling: I. theory and method," *Psychometrika*.
[10] A. J. Izenman, "Linear discriminant analysis," in *Modern multivariate statistical techniques*.
[11] T. Fester, F. Schreiber, M. Strickert, and I. Gatersleben, "Cuda-based multi-core implementation of mds-based bioinformatics algorithms."
[12] S. Ingram, T. Munzner, and M. Olano, "Glimmer: Multilevel mds on the gpu," *IEEE Transactions on Visualization and Computer Graphics*.
[13] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*.
[14] D. Achlioptas, "Database-friendly random projections: Johnson-lindenstrauss with binary coins," *Journal of computer and System Sciences*, vol. 66, no. 4, pp. 671–687, 2003.
[15] H. Osipyan, M. Kruliš, and S. Marchand-Maillet, "A survey of cuda-based multidimensional scaling on gpu architecture," in *OASIcs-OpenAccess Series in Informatics*.
[16] "Designing usable interactive visual analytics tools for dimension reduction," in *CHI 2016 Workshop on Human-Centered Machine Learning (HCML)*.
[17] "Bridging the gap between user intention and model parameters for data analytics," in *SIGMOD 2016 Workshop on Human-In-the-Loop Data Analytics (HILDA 2016)*.
[18] "Pyopencl," https://mathema.tician.de/software/pyopencl/.
[19] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*.