Efficient Band Approximation of Gram Matrices for Large Scale Kernel Methods on GPUs

Mohamed Hussein
Department of Computer Science
University of Maryland
College Park, MD 20742
mhussein@cs.umd.edu

Wael Abd-Almageed
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742
wamageed@umiacs.umd.edu

ABSTRACT

Kernel-based methods require $O(N^2)$ time and space complexities to compute and store non-sparse Gram matrices, which is prohibitively expensive for large scale problems. We introduce a novel method to approximate a Gram matrix with a band matrix. Our method relies on the locality preserving properties of space filling curves, and the special structure of Gram matrices. Our approach has several important merits. First, it computes only those elements of the Gram matrix that lie within the projected band. Second, it is simple to parallelize. Third, using the special band matrix structure makes it space efficient and GPU-friendly. We developed GPU implementations for the Affinity Propagation (AP) clustering algorithm using both our method and the COO sparse representation. Our band approximation is about 5 times more space efficient and faster to construct than COO. AP gains up to 6x speedup using our method without any degradation in its clustering performance.

1. INTRODUCTION

Kernel-based machine learning methods [15][25][27][32] have gained significant attention within the machine learning community and other applied fields for more than a decade. They are commonly used for many purposes, such as classification [12], regression [29], clustering [10], and dimensionality reduction [25]. The main advantage of kernel methods is their ability to learn non-linear functions using simple linear methods. They achieve this by implicitly mapping the input points from the *input* space to a typically higher, and possibly infinite, dimensional *feature* space. The mapping is realized via a kernel function, which computes a dot product between a pair of data points in the feature space without explicitly performing the mapping to that space. This is popularly known as the *kernel trick*. The pairwise dot product values are stored in the so-called *Gram matrix* or *kernel matrix*.

Given a data set of N points, computing the pairwise kernel values requires $O(N^2)$ computations and the values are stored in an $N \times N$ matrix. For large values of N, the time and space complexities to compute and store the Gram matrix can be prohibitively expensive. A common solution to the space complexity problem is to compute the elements of the kernel matrix on-demand, which trades the memory requirements for a much longer computational time. These complexities limit the use of kernel methods to relatively small problems. However, our era is marked with the availability of tremendous amounts of digital data that needs to be analyzed. Moreover, in many learning applications, increasing the number of training data points significantly improves the model's performance. For example, Munder and Gavrila [17] showed that the classification error of their Support Vector Machine (SVM) classifier, for human detection in images, was reduced by approximately a factor of two by only doubling the size of the training data set. They also noted that the reduction in the classification error obtained by increasing the number of training points exceeded any reduction obtained by using better features or learning algorithms. A similar observation was made by Torralaba et al. [31]. Inspired by such observations, our research focuses on enabling large scale learning with kernel based methods.

Fortunately, the availability of massive datasets nowadays and the increased demand and motivation for large scale learning is accompanied with the emergence of several new computing architectures, such as Graphics Processing Units (GPUs). GPUs have been rapidly advancing towards higher levels of parallelism, and have recently become readily programmable with simple thread-based Application Programming Interfaces (API) instead of using graphics primitives. However, the tremendous computing power provided by such devices is both a bless and a challenge at the same time. The virtue of parallelism offered by GPUs comes at the expense of several restrictions on the algorithm design in order to achieve the promised performance. One of the most critical of these restrictions is on the memory access pattern exhibited by the algorithm.

In this paper, we present a novel approach to address the limitations of kernel based methods for large-scale machine learning applications. Specifically, we introduce a new method to construct a band sparse matrix approximation to the Gram matrix. The idea is to order the input points so that the significant elements of the Gram matrix become confined to a limited band. Having a sparse structure for the Gram matrix is generally one of the ways to address the space com-

plexity problem of kernel methods. However, having a band sparse matrix structure in particular offers more advantages. It allows for a very simple representation that has significantly lower memory overhead than general sparse matrix representations. Moreover, this simple representation naturally adheres to the restrictions on memory access patterns on GPUs for many common matrix operations. Therefore, it allows for significantly more efficient implementations for most algorithms that operate on the matrix.

To construct a band approximation to the Gram matrix, we order the input points so that those that are close to one another in the resulting ordering are more likely to be close in the Euclidean space in which they reside. To efficiently obtain the desired ordering, we rely on the locality preserving properties of space filling curves [24]. To construct the matrix, we evaluate the kernel function only within a fixed neighborhood around each point in the obtained ordering. This results in a band Gram matrix by construction. The assumptions here are that the value of the kernel function is monotonically decreasing with the Euclidean distance between the input points, and the significant values of the function occur between points within the selected neighborhood size.

To illustrate the validity of the proposed approach, we use Affinity Propagation [9][10], an unsupervised clustering algorithm, as an example of kernel methods. Affinity Propagation (AP) operates on a *similarity matrix*. Similar to a kernel matrix, a similarity matrix has an element for every pair of points, whose value represents a measure of similarity between the two points. Typical choices of similarity functions, such as the negative sum of squared differences, and its exponential, can be shown to be dot products in higher dimensional mappings of the input points. We developed two GPU implementations for AP: one is based on our method, and the other is based on the COO (Coordinate) general sparse matrix representation [23]. As a baseline for comparison, we also developed a CPU implementation for AP based on COO. Our results show that the band matrix representation used in our method is 5 times more space and time efficient to construct than the COO representation on GPUs. Moreover, the GPU implementation of AP using our approach is 6 times faster than the GPU implementation using COO and 114 times faster than the CPU implementation. This significant speedup for AP comes with no loss in its clustering performance despite the approximation in our approach.

The main contributions of this paper are:

- An efficient method to construct a band approximation of a Gram matrix, without having to compute all elements of the matrix first. The simplicity of representing a band matrix allows for space efficiency and time efficiency on processing the matrix on GPUs. Hence, our method can effectively address the space and time complexities associated with kernel based learning methods for large scale problems.
- An efficient GPU implementation of the Affinity Propagation algorithm using our method. This implementation achieves 114x speedup over the CPU implementation and 6x speedup over another GPU implementation based on the COO sparse matrix representation. These speedups are achieved without compromising the quality of the output clustering.

The rest of the paper is organized as follows: We briefly present the related work in Section 2. In Section 3, we highlight the main features of GPUs and the performance-critical considerations needed to be taken into account to develop efficient GPU algorithms. Then, we explain our method to construct band approximations to Gram matrices on GPUs in Section 4. We introduce AP and its implementation on GPUs in Section 5. In Section 6, we present the experimental results. Finally, we outline our conclusions and plans for future work in Section 8.

2. RELATED WORK

The work addressing the limitations of large scale kernel methods can be broadly classified into two main categories – (1) methods that depend on constructing low-rank approximations of the kernel matrix and (2) efficient implementations for computing the kernel matrix. Low-rank methods depend on the observation that the eigen-spectrum of the kernel matrix rapidly decays, especially when the kernel function is a Radial Basis Function (RBF) [25][26][33]. Hence, for a kernel matrix K with eigenvalues $\lambda_1 \geq \lambda_2 \cdots \geq \lambda_N \geq 0$ and corresponding eigenvectors $\mathbf{v}_i$, $K = \sum_i^N \lambda_i \mathbf{v}_i \mathbf{v}_i^T$. However, since the eigen-spectrum decays rapidly (i.e. most of the information is stored in the first few eigen vectors), the kernel matrix can be approximated by $\tilde{K} = \sum_i^M \lambda_i \mathbf{v}_i \mathbf{v}_i^T$, and $M << N$. Williams and Seeger [34] use the Nystrom method [6] to compute the most significant M eigenvalues and eigenvectors. The number of computed eigenvectors is inversely proportional to the approximation error. Nystrom-based methods are $O(M^2N)$ where M is the number of computed eignvectors. Also, Drineas and Mahoney [8], and Smola and Schökopf [30], for example, compute a rank-k approximation of the kernel matrix using a subset of the column (or basis functions) of the kernel matrix. These methods generally are $O(N)$ in both space and time.

Due to the importance of kernel-based methods, they have been the target of prior GPU implementations. Ohmer et al. [21] use GPUs to implement the classification step of SVM classifier, in which the kernel values are computed between the input *test* vector and the set of support vectors. While focusing on the classification phase rather than the training phase of the computation can be justified by the higher frequency of using a trained model for classification in practical applications, for large scale learning the training phase becomes the main obstacle. In SVM classifiers, for example, the number of support vectors in a trained model can be much smaller than the training vectors used to train the model. Catanzaro et al. [4] presented an implementation of Platt's Sequential Minimal Optimization (SMO) [22] on GPUs for training SVMs. In this implementation, the kernel size issue was handled by caching recently used values and computing other values on demand upon cache misses. For large scale problems, cache misses are more likely to happen. Hence, computing the kernel values on cache misses is expected to be the computational bottleneck in large scale problems.

The idea of using space filling curves to order points for efficient access on GPUs was recently used by Leiberman et al. [14] with the similarity joint operation and was suggested also for use to approximate k-nearest neighbor search. Our sparse matrix representation actually uses approximate k-nearest neighbor search to obtain the band matrix structure.

In contrast to band reduction techniques, such as the RCM algorithm [5], our method does not start from an already constructed general sparse matrix and *reduce* it to a band matrix. Instead, our method directly *constructs* a band matrix from the input points. This is a fundamental difference since constructing a sparse matrix typically requires the computation of the full dense matrix first to determine which elements to keep. Our method computes only the elements within the projected band. Moreover, band reduction techniques typically use graph algorithms, which are hard to implement in parallel. Our method can be easily implemented in parallel in an efficient way.

3. MODERN GRAPHICS PROCESSING UNITS

We briefly present the main features of the Compute Unified Device Architecture (CUDA), which is the main stream architecture/model used for general purpose computing on GPUs nowadays. For a detailed description, the reader is referred to Nickolls et al. [18], and the NVIDIA CUDA Programming Guide [19].

3.1 Architecture

In CUDA, a parallel compute device, such as the GPU, is referred to as a *device*. A CUDA device is responsible of running CUDA *kernels* in parallel. A CUDA kernel is a C function which specifies the operation performed by a single *thread* of execution. Launching CUDA kernels and controlling the path of execution from one kernel to the next are performed by a separate serial processor, such as the CPU, referred to as a *host*.

Figure 1 is an illustration of CUDA's device architecture. A CUDA device consists of a number of *Streaming Multiprocessors* (SMs) that ranges from 2 to 30. Each SM consists of 8 core *Streaming Processors* (SPs). Each SP has exclusive access to a designated number of registers in its SM's register file. All SP's in the same SM have access to a low latency *shared memory* space. The shared memory is organized in *banks* so that each bank can serve one memory access at a time. All SPs in all SMs have access to three common memory spaces, which are

1. *Global Memory:* A read/write non-cached memory space.
2. *Constant Memory:* A read-only cached memory space.
3. *Texture Memory:* A read-only cached memory space with hardware support for filtering operations and memory access modes needed for texture fetching.

Accessing constant and texture memory spaces is as fast as accessing local registers on cache hits. Accessing shared memory is as fast as accessing registers if there is no memory bank conflict, i.e. if no two SPs access two different locations within the same shared memory bank. On the other hand, accessing global memory is typically up to two orders of magnitude slower. In fact, accessing global memory is also two orders of magnitude slower than floating point multiply and add.

3.2 Execution Model

The execution of CUDA kernels follows a Single-Instruction Multiple-Thread (SIMT) model. Each thread executes a CUDA kernel on a single SP. Threads are virtually organized in a three dimensional discrete space referred to as a *grid*. This space is further divided into equally sized rectangular boxes called *blocks*. The number of threads and dimensionality of the thread blocks and grid are specified by the programmer depending on the operation to be performed and the size and dimensionality of the input data. All threads in the same block execute on SPs of the same SM. Therefore, threads within a block can communicate with one another through the shared memory space in the assigned SM. Threads in the same block can also use efficient barrier synchronization to coordinate their executions. The execution unit of the SM executes threads in parallel in groups of 32, called *warps*. Threads within the same warp need to follow the same execution path to obtain the maximum possible performance. Otherwise, divergent execution paths within a warp are serialized. Different warps are run in parallel by an SM in a time slicing fashion.

3.3 Performance Considerations

There are several important considerations that must be taken into account in order to maximally exploit the computational power of CUDA device. For an extensive discussion of these considerations, the reader is referred to the "CUDA C Programming Best Practices Guide" [20]. The most important of these considerations is optimizing global memory accesses. As noted earlier, accessing global memory is significantly slower than accessing other memory spaces and than compute instructions. One way to alleviate this overhead is through sharing data loaded from global memory among threads by using the shared memory space. A significant gain can be achieved also by considering how accesses to global memory are realized by the hardware. If data is organized in a simple array structure so that each element is either 4, 8, or 16 bytes long, threads within the same half warp access consecutive data elements, and the starting address accessed by a half warp of threads is a multiple of 16 data elements size, these accesses are grouped (coalesced) in one memory access instruction for the entire half warp. If the last condition of address alignment is not satisfied, on devices with compute capability 1.2 or higher, at most two memory access instructions are issued for the entire half warp, while in older devices, 16 memory access instructions are issued. Fortunately, shared memory can be used to significantly reduce the overhead of non-aligned consecutive accesses on both old and new architectures. On the other hand, accessing random array elements by threads in the same warp can lead to launching a memory access instruction for each thread in all devices, which leads to a significant slow down. Other performance considerations include fine grain parallelism, and minimizing thread divergence and shared memory bank conflicts.

Taking these constraints into account is the key to achieving good performance on CUDA devices. As you will see, using

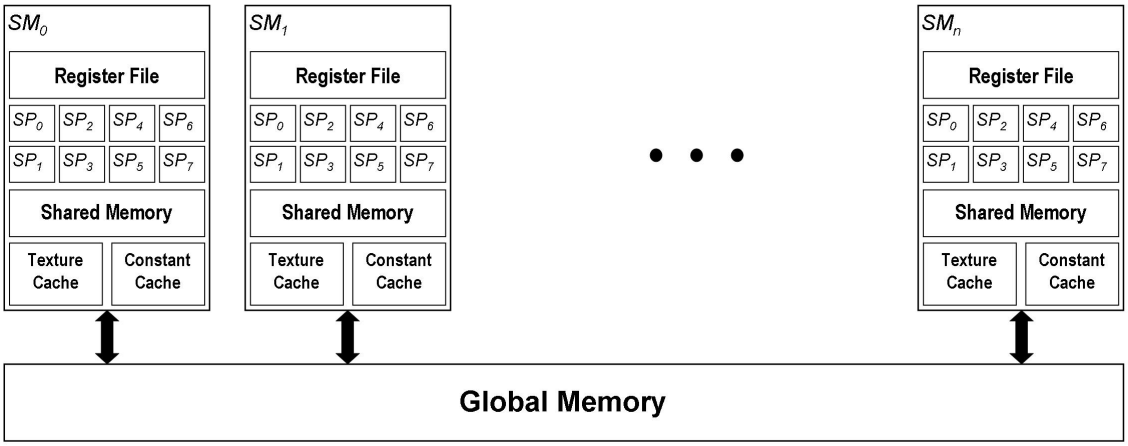

Figure 1: CUDA Architecture.

a data structure that can easily be accessed in a way that respects the global memory access rules can achieve significant speedups.

4. REPRESENTATION OF GRAM MATRICES ON GPUS

Since the Gram matrix often has a rapidly decaying eigenspectrum, as explained in Section 2, especially when using kernel functions with compact support, it is customary to assume that the matrix is approximately sparse and use sparse matrix structures to store (and operate on) its significant values. We are particularly seeking a sparse matrix representation that is efficient to construct, has low space overhead, and efficient to perform common matrix operations on when implemented on GPUs.

The Compressed Sparse Row (CSR) is a common sparse matrix representation on GPUs [11]. It supports efficient sparse matrix-vector multiplication, and other operations, through efficient segmented scans [7]. A closely related representation is the Coordinate (COO) representation [23] [2]. Despite the larger spatial complexity of COO compared to CSR, COO exhibits a better memory access pattern on GPUs for some operations. We use COO as a baseline representation in our experiments. Nevertheless, in the following discussions, all our arguments about COO, except for the space complexity issue, apply equally to CSR.

COO is a general sparse matrix representation that does not assume any special structure for the matrix. As we will show below, due to its generality, the COO representation has several shortcomings in terms of its space overhead and the memory access pattern exhibited with it in common matrix operations on GPUs. To overcome these limitations, we aim to find an approximation to the Gram matrix with a special structure that can be represented efficiently on GPUs in terms of space and computations. In our approach, this special structure is the band matrix structure.

In the rest of this section, we first explain the COO representation and its limitations in Section 4.1. Then, we explain the band matrix structure and its merits in Section 4.2. Finally, we explain how we obtain the band approximation of the Gram matrix in our approach in Section 4.3.

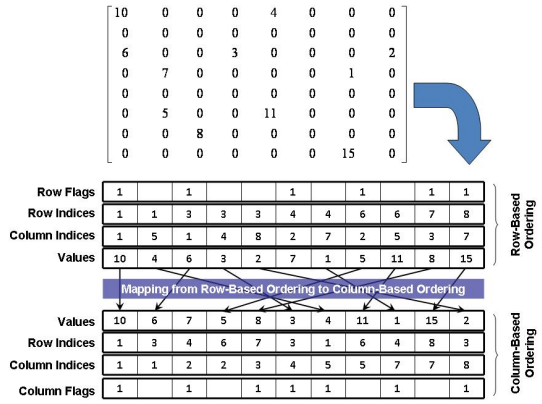

Figure 2: Representation of an example general sparse matrix using the Coordinate (COO) representation. The representation consists of three arrays to store row indices, column indices, and values of non-zero elements. The arrays can be sorted according to either the row indices or the column indices depending on whether we need to perform scan over the rows or the columns of the matrix. If scanning is performed on rows and columns interchangeably, a mapping from one ordering to the other is retained with the structure. Finally flags arrays indicating the beginning of each row and column is needed for the segmented scan operation.

4.1 General Sparse Matrix Representation Using COO

Consider the COO representation of a general sparse matrix, as shown in Figure 2. In this representation, three arrays are used to store the row index, column index, and the value of each significant element in the matrix. For m significant elements to store, we use the space of $3m$ elements, which

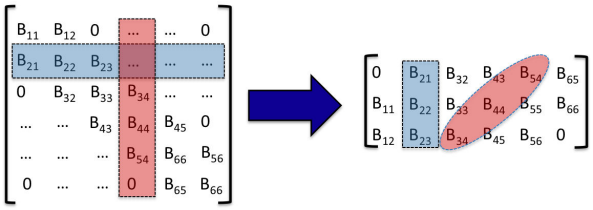

Figure 3: Representation of a band matrix using a 2D array. Each diagonal of the matrix is stored in one row of the array. Each row is stored in one column of the array. And each column of the matrix is stored as a diagonal in the array.

is a significant overhead factor. Beside the space overhead of this representation, when processing each element of the matrix depends on its row and column indices, three arrays need to be accessed in order to process all elements, which results in a significant time overhead if the processing is not compute intensive.

To perform a scan operation on the rows of the matrix efficiently in parallel, the three arrays need to be sorted according to the row index values. Having arrays sorted in such a way, a segmented scan operation can be used to perform the scan operation in parallel. Fortunately, there are efficient algorithms for segmented scans [28] [7]. However, the segmented scan operation requires as input an array of flags, whose elements designate the beginning of each row of the matrix. This is on top of the internal arrays used by the operation itself. Therefore, the operation can be performed efficiently on a GPU with the usage of extra space. Similarly, if we are to perform a scan operation on the columns of the matrix, we need to have the arrays sorted according to column indices, and an array of flags to mark the beginning of each column. Figure 2 shows the COO representations of a sample sparse matrix with using row-based ordering and column based ordering.

Another shortcoming of this representation arises when we need to perform the scan operation on both rows and columns interchangeably. In this case, the COO representation must be extended. One solution is to keep the arrays sorted according to the row indices, for example, and two extra arrays: one to store the mapping from the row-index-based order to the column-index-based order of the arrays, Figure 2, and the other is the flags array of the column-based ordering. When we need to perform a scan operation over the columns, we use the mapping array to reorder the values and perform a segmented scan on the reordered array. Note that we need an extra array to temporarily store the reordered values. Finally, the mapping from one order to the other requires random device memory access during write, which does not respect the conditions for coalescing, as discussed in Section 3.

4.2 Band Matrix Representation

Consider an $N \times N$ band matrix with a bandwidth k, i.e. the non-zero (or significant) elements are confined to at most k diagonals. A simple representation of such a matrix on the GPU is a 2D array, where each diagonal of the matrix is stored as a row of the array, and each row of the matrix is stored as a column of the array [2], as shown in Figure 3. To perform a parallel scan operation on the rows of the matrix, we can assign each thread to a column of the representation array. Each thread loops over the elements of its assigned column and performs the operation. Hence, consecutive threads in a block of threads read consecutive elements in memory. Furthermore, if the array is allocated so that each row starts at a properly aligned memory address, and the block width is selected appropriately, all conditions for memory access coalescing are satisfied, and hence the read operation is performed efficiently.

Note that in our 2D array representation, columns of the matrix are stored in diagonals of the array, as shown in Figure 3. If we need to perform a scan operation on the columns of the matrix rather than the rows, we can still assign each thread to a column in the matrix. Each thread loops over the elements of its assigned column. Consecutive threads still read consecutive memory addresses. However, since columns of the matrix are stored as diagonals in the representation arrays, looping over elements in a column require non-aligned memory access. Fortunately, as we mentioned in Section 3.3, consecutive accesses, even if the addresses are not properly aligned, can always be made to satisfy coalescing requirements either directly through the hardware in latest models, or through software by making good use of the shared memory space.

As we have shown, we can efficiently perform simple scan operations on the rows or columns of a band matrix represented as a 2D array. Another advantage of this representation is that the row and column indices of each element can be calculated instead of being read from separate arrays, which saves a lot of time in memory bound processing. The space overhead of the band representation depends on the location of the significant diagonals with respect to the main diagonal. Suppose that the bandwidth $k = 2h + 1$, so that the bandwidth is divided as the main diagonal, and h diagonals below it and h diagonals above it. In this case, we use a space sufficient for kN elements to actually store $kN - h^2 - h$ elements. Therefore the space overhead is $\frac{h^2+h}{kN}$. Note that the space overhead is always smaller than 1. The scan operations do not require any extra space in the device memory.

4.3 Band Approximation of Gram Matrices

We have shown the advantages of the simple representation for band matrices over the COO representation for general sparse matrices in terms of the space overhead and the efficiency of the memory access pattern for performing simple scan operations over the rows and columns of the matrix. To exploit these advantages for sparse kernel matrices, we need to find an ordering of the rows and columns that puts most of the significant elements within a fixed number of diagonals and use a band matrix representation for the resulting matrix. If the scan operations to be performed are both commutative and associative, changing the order of the rows and columns will not affect the results. We refer to this approach by BAG, for Band Approximation of Gram matrices.

Each element of a Gram matrix is the value of a kernel function on two points. We assume that the data points lie in a

Cartesian space and the kernel value between pairs of points is inversely proportional to the distance between the points in the Cartesian space. These assumptions are satisfied by a variety of kernel functions, including the most popular RBF kernel. A typical kernel function choice in Affinity Propagation is the negative sum of squared differences, which satisfies these assumption too.

In order to obtain a band kernel matrix of bandwidth k, the problem is to find an ordering of the points such that nearby points in the Cartesian space are at most k elements apart in that ordering. This ordering may not optimally exist. Therefore, we need an ordering that satisfies this property for most of the elements. We can formulate the problem of finding such ordering as an optimization problem. This is basically the idea of band reduction techniques for sparse matrices [5]. However, band reduction techniques require the construction of the sparse matrix first, and are complex to parallelize as we explained in Section 2.

We use Space Filling Curves (SFC) [24] to obtain the desired reordering. A space filling curve is a path through the points of a discrete Cartesian space that passes through each point exactly once. There are many types of SFCs. Typically, an SFC is more likely to connect points that are close to one another in the space than to connect points that are far away in the space. However, this *locality preserving* property varies from one type of curves to another. The family of Hilbert curves [3] is known to have good locality preserving properties. However, they are complex to construct. A much simpler curve to construct is the Z-curve [16]. Despite being inferior to the Hilbert curves in locality preserving, it is good enough in many applications. As we will show, it works remarkably well for kernel matrix reordering with the affinity propagation algorithm.

The construction of the space filling curve is performed implicitly (i.e. we need not know how exactly the curve looks like.) Given a set of points in a Cartesian space, all what we need to know is their ordering along the curve, i.**e**. in which order the points are encountered upon traversing the curve from its starting to its ending points. For the Z-curve, this ordering is known as the z-order, or the Morton code. The z-order can be computed very efficiently using bit interleaving of the point coordinates in the Cartesian space [24]. For real valued data, we first map the points to the unit hypercube. Then, we discretize the coordinates by mapping each point to the closest cell of a discrete grid over the unit hypercube . Morton codes are computed using the discrete coordinates of the assigned grid cells, and the points are sorted by their ascending or descending Morton code values. Figure 4 shows an illustration of these steps for a simple 2D example. In our implementation, we use the bitonic sort algorithm [13] to sort the codes. Although the complexity of bitonic sort is $\mathcal{O}(N \log^2 N)$, the time to compute the codes and sort them is negligible with respect to the rest of the computations.

5. AFFINITY PROPAGATION ON GPUS

To illustrate the effectiveness of our band approximation of the Gram matrix, we use Affinity Propagation (AP) as an example of kernel methods. In this section, we briefly explain the AP algorithm, and describe our GPU implementation.

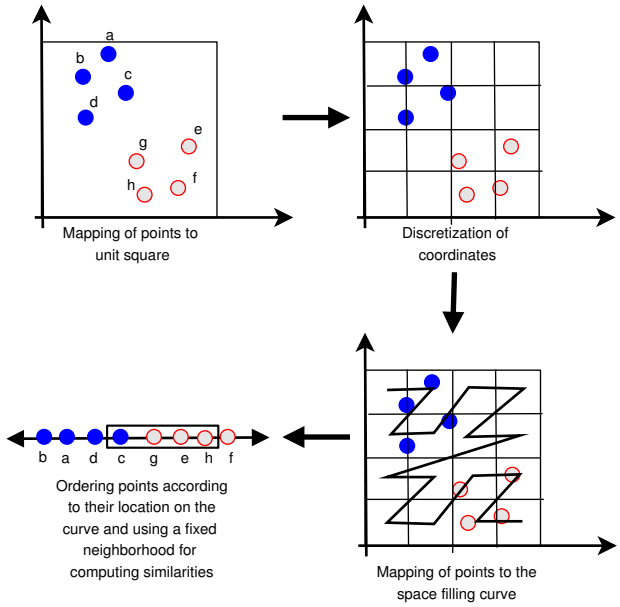

Figure 4: An illustration of the construction of the similarity matrix using space filling curves. The Z-curve is used in this illustration, and in our implementation.

5.1 Affinity Propagation

AP is an unsupervised data clustering algorithm introduced by Frey and Dueck [9][10]. There are two main advantage of data clustering using AP: First, the number of clusters K need *not* be a priori specified. Second, AP operates on pair-wise similarity values which can be computed on non-Euclidean manifolds. For completeness, we briefly describe Affinity Propagation clustering.

Let $\mathbf{X} = \{\mathbf{x}_i; i = 1, 2, \ldots, N\}$ be a set of data points (i.e. observations vectors) with unknown cluster structure and $\mathbf{X} \subset \mathrm{R}^d$. The objective is to find a subset $\mathbf{X}_e = \{\mathbf{x}_k; k = 1, 2, \ldots, K\} \subset \mathbf{X}$ of cluster exemplars where $K \ll N$. This problem is classically handled using the K-center algorithm in which K points are selected at random from $\mathbf{X}$ and the subset is iteratively refined by minimizing the distance between the data points and the exemplars. The procedure is usually repeated more than once in order to converge to the best solution. AP, on the other hand, considers all points to be possible exemplars. Based on similarity (as opposed to distance) $s(i, j)$ between $\mathbf{x}_i$ and $\mathbf{x}_j$. Self similarity values $s(k, k)$ are referred to as the preference values. The higher the preference given to a sample point, the more likely it can be selected as an exemplar by the algorithm.

AP operates by iteratively exchanging two types of messages between data points – availabilities and responsibilities. Responsibility $r(i, k)$ indicates the desire of point i to belong to a cluster for which point k is the exemplar. Availability $a(i, k)$ indicates the willingness of point k to serve as the exemplar of the cluster to which point i belongs. All availabilities are initialized to zeros. Responsibilities are updated as soft assignments using Equation 1.

$$r(i,k) \leftarrow \left\{ s(i,k) - \max_{k' \text{ s.t. } k' \neq k} \left(a(i,k') + s(i,k') \right) \right\} \quad (1)$$

This responsibility update ensures that all potential exemplars compete for data points. Availabilities are updated using Equation 2.

$$a(i,k) \leftarrow \min \left\{ 0, r(k,k) + \sum_{i' \text{ s.t. } i' \notin \{i,k\}} \max\,(0, r(i',k)) \right\} \quad \text{and} \quad i \neq k \tag{2}$$

The self-availability $a(k,k)$ is updated differently in order to reflect the evidence that point k can be an exemplar, as shown in Equation 3.

$$a(k,k) \leftarrow \sum_{i' \text{ s.t. } i' \neq k} \max\,(0, r(i',k)) \tag{3}$$

The algorithm proceeds by iterating over the responsibility and availability update steps in Equations 1, 2, and 3 until convergence or the maximum number of iterations is reached [10].

5.2 GPU Implementation

The dense matrix implementation of AP is $O(N^2)$ in both computational and memory requirements. In practice, the similarity values s_{ij} can be thresholded so that small values are ignored and the pairwise similarity values can be stored in a sparse matrix. Using the massive parallelism available in modern GPUs, we can effectively address the computational complexity problem.

For simplicity of presentation let's assume that full matrices are used to implement AP. To store the similarity values $s(i,j)$, and the preference values $s(k,k)$, we need an $N \times N$ array S. To store the availability and responsibility messages sent from one point to another, we need another two arrays of the same size, A and R, respectively. From equation 1, to update the responsibility values, we need to scan rows of the A and S arrays. Specifically, we need two passes over each row. In the first pass, we compute the maximum two $a(i,k) + s(i,k)$ values in each row. In the second pass, we compute the updated responsibility value for each element in the row, using the two maximums computed in the first pass. From equation 2, to update the availability values, we need to scan the columns of the R array twice as well. In the first scan pass, we compute the sum of all positive elements in the column excluding the self responsibility values. In the second pass, we update the availability value of each element using the sums computed in the first pass. Implementing row and column scans on full matrices on the GPU is straight forward and efficient. However, we cannot store full matrices in memory even for moderately large problems. Therefore, we must use a sparse structure.

Both the COO and BAG representations support row and column scan operations which we need to perform interchangeably in AP. We will show that the COO representation will be highly inefficient for this purpose compared to the BAG representation. The COO structure is constructed by first sampling random pairs of points and computing similarity values between them. Then, we select a threshold below which similarity values are discarded. The threshold is selected based on the random sample and based on a pre-specified limit on the final storage size. Note that to construct the COO structure, we need to compute the similarity values between all pairs of points in order to threshold them and keep the significant ones only. Also, recall from Section 4.1 in order to support both row and column scans in this structure, we need to keep a mapping from an ordering based on row indices to an ordering based on column indices. In our implementation, we construct the structure first ordered by row indices, then use bitonic sort to obtain the ordering based on column indices, and retain the mapping between the two orderings. After constructing the S matrix using this representation, the A and R arrays are represented only as values arrays. They share the row and column indices arrays with the structure for S. In our implementation, we compute a row of the full similarity matrix at a time. Then, we threshold the values and use a compact operation to move the significant elements to the the COO structure.

To implement the BAG structure, the data points are mapped to the unit hypercube, discretized, converted to Morton codes, and sorted based on such codes. Then, the similarity matrix is constructed to include only similarity values between points that are at most h elements apart on the final SFC order, where h is 128 in our implementation. We refer to the value $2h$ as the neighborhood size. The similarity matrix S is represented as a 2D array with $2h+1$ rows and N columns, as shown in Figure 5. Column i of the matrix contains similarity values between element i and elements from $i-h$ to $i+h$ in order. The h^{th} row of the matrix contains the preference values. The responsibility and availability matrices, R and A, are constructed to have the same size and structure of the similarity matrix S.

6. EXPERIMENTAL RESULTS

We implemented the Affinity Propagation on CUDA using both the COO representation and our BAG representation, for the similarity matrix. We also implemented a version for the CPU based on the COO representation. We conducted our experiments on randomly generated point sets. The number of points in these sets ranged from $1K$ to $512K$. We used an NVIDIA Tesla C1060 compute card, which has 240 core processors and 4GB RAM, installed on an Intel Xeon 3.2 GHz workstation with 3GB RAM running 32-bit Windows XP with SP3. We used CUDA version 2.2 for our experiments. We used the CUDA Parallel Primitives Library (CUDPP) [1] in all scan and segmented scan operations.

Due to the limit on grid dimensions, we were not able to experiment with more than $128K$ points with the COO-GPU implementation. This problem arises only with the COO representation since we use scan operations in its implementation. The scan operation in CUDPP creates one thread for every 4 elements of the input array (the similarity matrix values in this case), which results in too many threads required to process the $256K$ points case and beyond. While this issue can be fixed by modifying the kernel functions for segmented scans in CUDPP, we cannot run this implementation with more than $256K$ points anyways because of the memory requirement of the COO representation exceeds the size of the device memory in this case. For the CPU representation, we were not able to run the experiment beyond

	1	...	i	...	N
$-h$			$s_{(h-1)i}$		
$\vdots$			$\vdots$		
-1					
0		...	s_{ii}	...	
1					
$\vdots$			$\vdots$		
h			$s_{(h+1)i}$		

Figure 5: The layout of the similarity matrix, S, used in AP's implementation with the BAG method. Each column of the matrix contains similarities to neighbors ranging from $-k$ to k apart from the column's index. The responsibilities and availabilities matrices, R and A, use the same structure. elements in row index 0 represent preference, self-responsibility, and self-availability values, in the S, R, and A matrices, respectively.

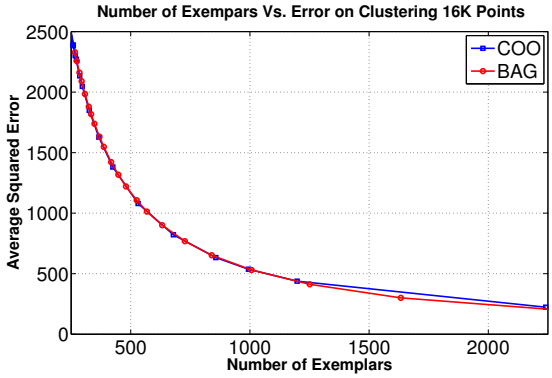

Figure 6: This plot compares the average squared error (distance between each point and its assigned exemplar), of the clustering obtained by affinity propagation on the COO and BAG representations, as a function of the number of exemplars. In both cases, the number of points is fixed at 16K. The plot shows how the usage of the approximate BAG sparse representation does not affect the clustering performance of affinity propagation.

$128K$ points either due to the extremely long time it requires. We used the negative sum of squared differences as the kernel (here similarity) function. The preference value was set to the mean similarity over the elements kept in the matrix representation in use. We set the maximum number of iteration for the AP to 2000. The neighborhood size was fixed at 256 for the BAG representation, which means the bandwidth of the resulting band matrix is 257. For the COO representation, we retain values above some threshold. To have a fair comparison, we select the threshold value to obtain approximately the same number of elements in the BAG representation. We compute this threshold based on a random selection of one million pairs of points.

6.1 Error Versus Number of Exemplars

The BAG representation is an approximation to the sparse kernel matrix, which is in turn an approximation to the full matrix. In this experiment, we want to assess how much the performance of the affinity propagation is affected by using the BAG representation, rather than the COO representation, in terms of the clustering error. We do not compare to the performance using the full matrix representation since the size of such a matrix is prohibitively huge and computing its elements upon need is prohibitively computationally expensive.

The clustering error is measured as the average square distance between each point and its assigned exemplar, $\frac{1}{N}\sum_i s(i, e_i)$, where e_i is the index of the exemplar assigned to point i. The closer a point on average to its exemplar the better the clustering. However, we cannot use this measure without referring to the number of exemplars since increasing the number of exemplars reduces this measure. In Figure 6, we show the clustering error with changing the number of exemplars. In this experiment, we fix the number of points to $16K$ and change the preference value to obtain different points on the curve. We compare between the two sparse matrix representations. The plot clearly shows that the difference between the two representations is negligible in terms of clustering error. Therefore, the approximation introduced by the BAG representation does not have any negative effect on the AP algorithm.

6.2 Time Versus Number of Points

In this set of experiments, we measure the computational time versus the number of points. We vary the number of points from $1K$ to $512K$, except with the COO representation on both CPU and the GPU where the maximum is $128K$. Figure 7 compares between the three implementations based on the convergence time of AP clustering. The time complexity of the three implementations grow almost linearly with the number of points. Since we fix the neighborhood size, this is consistent with the theoretical complexity of the algorithm, which is linear in the number of similarity values used (quadratic in the number of points for a full matrix representation). Most of the time all the implementations run until the maximum number of iterations, 2000. The few exceptions for this are the points that significantly deviate from the linear trend in the plots, which are the $1K$ and $4K$ points on the BAG-GPU curve and the $1K$ point on the COO-CPU curve. Excluding these points, the two GPU implementations consistently outperform the CPU implementation, with up to 18x speedup for the COO representation and up to 114x speedup for the BAG representation. Figure 8 shows the times to construct the similarity matrix representations for the same set of experiments. Since in the COO representation we need to compute all elements of the similarity matrix to compare them to the

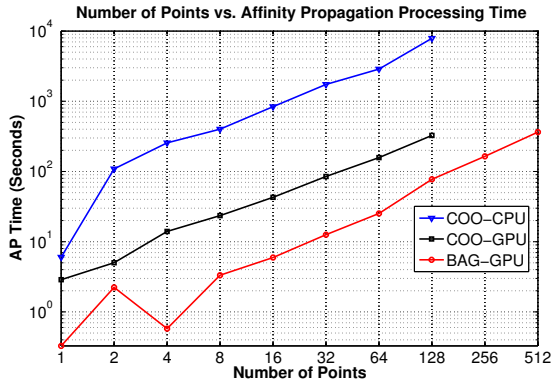

Figure 7: This plot compares the running times of affinity propagation, using the COO and the BAG representations on the GPU and the COO representation on the CPU, as a function of the number of input points. The number of points shown is in units of K (1024). The times shown do not include the time to construct the similarity matrix from the input points. Neither the COO-GPU nor COO-CPU implementations handles more than 128K points. The COO-GPU version achieves up to 18x speedup, while the BAG-GPU version achieves up to 114x speedup over the CPU implementation.

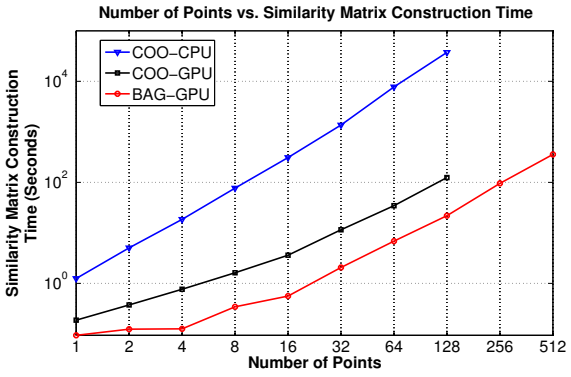

Figure 8: This plot compares the times of constructing the similarity matrix, using the COO and the BAG representations on the GPU and the COO representation on the CPU, as a function of the number of input points. The number of points shown is in units of K (1024). Neither the COO-GPU nor COO-CPU implementations handles more than 128K points. The COO-GPU implementation achieves up to 300x speedup, while the BAG-GPU implementation achieves up to 1700x speedup.

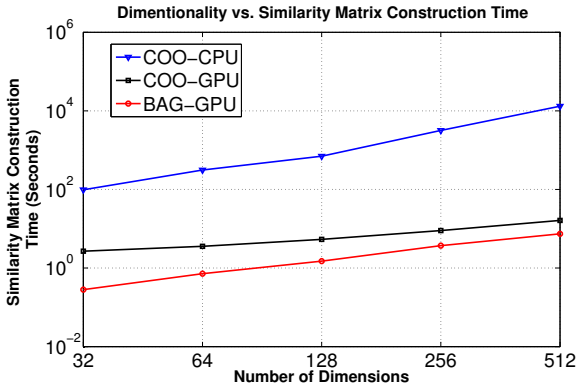

Figure 9: This plot compares the times of constructing the similarity matrix, using the COO and the BAG representations on the GPU and the COO representation on the CPU, as a function of the points dimensionality. As the dimensionality grows, the two GPU implementations achieve around 1000x speedup compared to the CPU implementation.

threshold, the construction of the similarity matrix on the CPU becomes the computational bottleneck as the number of points increase, while the GPU implementations are less affected due to parallelism. The GPU implementations score larger speedups in this part of the computation than the AP part, with the COO achieving up to 300x speedup, and the BAG achieving up to 1700x speedup. The simplicity of the BAG representation is the key to this tremendous speedup.

6.3 Time Versus Dimensionality

In this experiment, we study the effect of point dimensionality on the time to construct the kernel matrix representation. We fix the number of points at $16K$ points. We change the point dimensionality from 32 to 512. Figure 9 shows the results of this experiment. The advantage of using the GPU becomes more evident when the dimensionality increases. At 512 dimensions, both GPU implementations are about 1000 times faster than the CPU. The BAG representation is at least two times faster than the COO representation on the GPU. This again emphasizes the advantage of having a simple representation, such as the BAG over a complex representation such as the COO.

7. CONCLUSION

We presented a novel method to construct a band approximation to Gram matrices, based on space filling curves. The proposed method is very simple to construct and efficient to work with on modern graphics processing units than the conventional Coordinate (COO) representation. We applied the new representation to Affinity Propagation, a recently introduced unsupervised clustering algorithm. Our results show a significant speedup, of up to 114x, when using our algorithm on the GPU compared to the CPU implementation, compare to 18x speedup when using the COO representation. If we include the time to construct the sparse matrix structure, the speedup jumps up to 330x. This speedup does not come at any expense in terms of the clustering performance of the AP algorithm.

There are many interesting experiments to be conducted on

our work, such as studying the effect of neighborhood size on the time and clustering performance of the algorithm, and studying the approximation error to the kernel matrix incurred by our representation compared to the COO representation. Nevertheless, enabling large scale clustering via an effective algorithm such as Affinity Propagation is by itself an important achievement. We are planning on apply this method to real world large scale machine learning applications. Given the success on AP, we are encouraged to investigate the applicability of our approach to other kernel methods, such as SVMs. We are also investigating other types of codes that can be used to order input points other than space filling curves.

8. ACKNOWLEDGMENT

This research was supported by the US Government under the CTA project.

9. REFERENCES

[1] CUDPP: CUDA Data-Parallel Primitives Library.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA, December 2008.

[3] A. R. Butz. Alternative algorithm for hilbert space filling curve. *IEEE Trans. on Computers*, 20:424–42, April 1971.

[4] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML*, 2008.

[5] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th national conference*, 1969.

[6] L. M. Delves and J. Walsh, editors. *Numerical Solution of Integral Equations*. Oxford University Press, 1974.

[7] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, 2008.

[8] P. Drineas and M. W. Mahoney. Approximating a gram matrix for improved kernel-based learning. In *in Proceedings of the 18th Annual Conference on Learning Theory, 2005*, pages 323–337, 2005.

[9] B. Frey and D. Dueck. Mixture modeling by affinity propagation. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 379–386. MIT Press, Cambridge, MA, 2006.

[10] B. J. Frey and D. Dueck. Clustering by Passing Messages Between Data Points. *Science*, 315:972–976, 2007.

[11] M. Garland. Sparse matrix computations on manycore GPU's. In *Annual ACM IEEE Design Automation Conference*, 2008.

[12] B. Heisele, P. Ho, and T. Poggio. Face recognition with support vector machines: global versus component-based approach. In *In Proc. 8th International Conference on Computer Vision*, pages 688–694, 2001.

[13] P. Kipfer and R. Westermann. Improved GPU sorting. In M. Pharr, editor, *GPU Gems2*, pages 733–746. Addison Wesley, 2005.

[14] M. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *IEEE International Conference on Data Engineering*, 2008.

[15] J. Lu, K. Plataniotis, and A. Venetsanopoulos. Face recognition using kernel direct discriminant analysis algorithms. *Neural Networks, IEEE Transactions on*, 14(1):117–126, Jan 2003.

[16] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[17] S. Munder and D. M. Gavrila. An experimental study on pedestrian classification. *IEEE Trans. Pattern Anal. Machine Intell.*, 28(11):1863–1868, November 2006.

[18] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[19] NVIDIA. *NVIDIA CUDA Programming Guide Version 2.2*, April 2009.

[20] NVIDIA Corporation. *NVIDIA CUDA C Programming Best Practices Guide*, July 2009.

[21] J. Ohmer, F. Maire, and R. Brown. Implementation of kernel methods on the GPU. In *DICTA '05: Proceedings of the Digital Image Computing on Techniques and Applications*, page 78, Washington, DC, USA, 2005. IEEE Computer Society.

[22] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in kernel methods: support vector learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[23] Y. Saad. *SPARSKIT: A basic tool kit for sparse computations*, June 1994.

[24] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. The Morgan Kaufmann Series in Computer Graphics, 2006.

[25] B. Schölkopf, A. Smola, and K.-R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10(5):1299–1319, 1998.

[26] M. Seeger. Bayesian model selection for support vector machines, Gaussian processes and other kernel classifiers. In S. A. Solla, , T. K. Leen, and K. R. Müller, editors, *NIPS*, pages 603–609. MIT Press, 2000.

[27] M. Seeger. Gaussian processes for machine learning. *International Journal of Neural Systems*, 14(2), 2004.

[28] S. Sengupta, M. Harris, Y. Zhang, and J. Owens. Scan primitives for GPU computing. In *ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2007.

[29] A. Smola. Regression estimation with support vector learning machines. Master's thesis, Technische Universitat Munchen, 1996.

[30] A. J. Smola and B. Schökopf. Sparse greedy matrix approximation for machine learning. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 911–918, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[31] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *IEEE Conf. on Computer Vision and Pattern Recognition*, pages 1–8, Anchorage, AK, June 2008.

[32] V. Vapnik. *The nature of Statistical Learning Theory*. Springer-Verlag, 1995.

[33] C. K. I. Williams and M. Seeger. The effect of the input density distribution on kernel-based classifiers. In *International Conference on Machine Learning*, 2000.

[34] C. K. I. Williams and M. Seeger. Using nystrom method to speed up kernel machines. In T. K. Leen, T. G. Diettrich, and V. Tresp, editors, *NIPS*, volume 13. MIT Press, 2001.