

# GPUML: Graphical processors for speeding up kernel machines

Balaji Vasan Srinivasan, Qi Hu, Ramani Duraiswami

Perceptual Interfaces and Reality Laboratory,

Institute for Advanced Computer Studies & Department of Computer Science,

University of Maryland, College Park, MD, USA.

[balajiv,huqi,ramani]@umiacs.umd.edu

## Abstract

Algorithms based on kernel methods play a central role in statistical machine learning. At their core are a number of linear algebra operations on matrices of kernel functions which take as arguments the training and testing data. These range from the simple matrix-vector product, to more complex matrix decompositions, and iterative formulations of these. Often the algorithms scale quadratically or cubically, both in memory and operational complexity, and as data sizes increase, kernel methods scale poorly. We use parallelized approaches on a multi-core graphical processor (GPU) to partially address this lack of scalability. GPUs are used to scale three different classes of problems, a simple kernel-matrix-vector product, iterative solution of linear systems of kernel function and QR and Cholesky decomposition of kernel matrices. Application of these accelerated approaches in scaling several kernel based learning approaches are shown, and in each case substantial speedups are obtained. The core software is released as an open source package, GPUML.

**Keywords:** kernel machines, matrix decomposition, kernel density estimation, Gaussian process regression, ranking, SRKDA

## 1 Introduction

During the past few decades, it has become relatively easy to collect huge amounts of data. Examples include data in astronomy, internet traffic, meteorology and surveillance. A goal of this collection is to mine the data for useful information and thus build meaningful statistical patterns that allow one to predict/recognize unseen patterns. Several machine learning algorithms [1] have been proposed and used; *kernel machines* are a particular class of approaches that are very popular for their robustness. Widely used methods such as support vector machines, kernel density estimation, Gaussian process regression are subclasses of kernel machines. However, the computational complexity of these are either quadratic or cubic, thus hindering their

application to very large datasets.

A core computation in many kernel approaches is the weighted summation of kernel functions

$$(1.1) \quad f(x_j) = \sum_{i=1}^N q_i K(x_i, x_j), \quad \mathbf{f} = \mathbf{K}\mathbf{q},$$

which may also be treated as the product of a kernel matrix  $\mathbf{K}$  with a vector  $\mathbf{q}$ . Here  $x_i$  is the  $d$ -dimensional observation. Typically,  $f(x)$  needs to be evaluated at  $M$  points, resulting in an overall complexity of  $O(MN)$ . By evaluating the kernel function *on the fly*, the space complexity can be kept to  $O(M + N)$ . Other computations with the matrix  $\mathbf{K}$ , or its relatives, may also be sought, including solution of linear systems, eigen decomposition and others, and usually the complete matrix has to be stored in some of these cases, increasing the memory complexity to  $O(MN)$ .

Existing approaches to accelerate kernel methods, either approximate the kernel summation/decomposition or parallelize them. Approaches like the Improved Fast Gauss Transform [24, 16] and dual-trees [8] evaluate kernel sums in linear time using efficient approximations. Message Passing Interface (MPI) on clusters [9] and thread-based parallel approaches on graphical processing units (GPU)[20, 13] have been used to parallelize and thus speed up kernel machines. Most of the GPU based parallelization have primarily casted the underlying problems in terms of pixel and fragment shaders. With the emergence of CUDA (Compute Unified Device Architecture), it is possible to remove this additional overhead to exploit the computational capabilities of the GPU. This has been exploited to accelerate the popular kernel machine, SVM in [21, 3]. Although CUDA based GPU algorithms have been used in some applications, a comprehensive work on the use of GPU for kernel machines has never been done and in this paper, we try to address this by accelerating the following categories of kernel based algorithms using CUDA on GPU,

1) simple matrix-vector product involving kernel

matrices (eg. for kernel density estimation)  
**2)** solution of linear system of kernel matrices (eg. for kernel regression)  
**3)** decomposition (like Cholesky, QR) of kernel matrices (eg. for spectral clustering)  
 We feel that the computational bottleneck in most of the kernel machines falls into one of these three categories. We propose approaches to accelerate each of these on a GPU and illustrate the speedup on applications like kernel density estimation, mean shift clustering, Gaussian process regression, ranking and kernel discriminant analysis.

In section 2, we discuss GPU architecture and its improving trend and also introduce NVIDIA CUDA. We discuss the various limitations of GPUs that need to be considered when designing the algorithms. In section 3, we discuss the mapping of various kernel problems on to the GPU. We then introduce accelerated kernel matrix-vector product and matrix decomposition on a GPU. In section 4, we present the various experiments performed to illustrate the speedups obtained in each case. Finally we provide our conclusions in section 5 and discuss the limitations and future prospects of our approach.

## 2 Graphical Processors:

Computer chip-makers are no longer able to easily improve the speed of processors, with the result that computer architectures of the future will have more cores, rather than more capable faster cores. This era of multicore computing requires that algorithms be adapted to the data parallel architecture. A particularly capable set of data parallel processors are the graphical processors, which have evolved into highly capable compute coprocessors. A graphical processing unit (GPU) is a highly parallel, multi-threaded, multi-core processor with tremendous computational horsepower. In 2008, while the fastest Intel CPU could achieve only  $\sim 50$  Gflops speed theoretically, GPUs could achieve  $\sim 950$  Gflops on actual benchmarks [12]. Fig. 1 shows the relative growth in the speeds of NVIDIA GPUs and Intel CPUs as of 2008 (similar numbers are reported for AMD/ATI CPUs and GPUs). The recently announced FERMI architecture significantly improves these benchmarks. Moreover, GPUs power utilization per flop is an order of magnitude better. GPUs are particularly well-suited for data parallel computation and are designed as a single-program-multiple-data (SPMD) architecture with very high arithmetic intensity (ratio of arithmetic operation to memory operations). However, the GPU does not have the functionalities of a CPU like task-scheduling. Therefore, it can efficiently be used to assist the CPU in its operation rather than replace it.

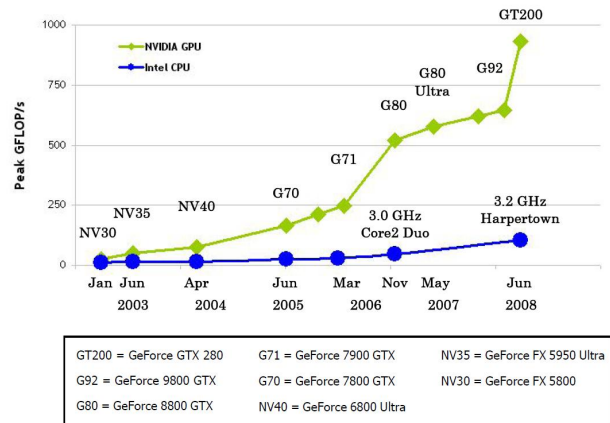


Figure 1: Growth in the CPU and GPU speeds over the last 6 years on benchmarks (Image from [12])

In November 2006, NVIDIA introduced *Compute Unified Device Architecture (CUDA)*[12], a parallel programming model that leverages the parallel compute engine in NVIDIA GPUs to solve general purpose computational problems. With CUDA, GPUs can be seen as a bunch of parallel co-processor that can assist the main processor in its computations. The OpenCL initiative seeks to provide a similar non-proprietary API for general purpose GPU computing.

Fig. 2 shows how current GPU coprocessors appear to a user through CUDA. Each GPU has a set of multiprocessors, each with 8 processors. All multiprocessors communicate with a global memory, which can be as high as 4GB and a constant memory. More capable GPUs share more multiprocessors and more global memory. The 8 processors in each multiprocessor share a local shared memory and a local set of registers. The instructions in the GPU are designed to be executed as parallel threads on multiple data. Therefore, the computations are organized into grids, which are groups of thread blocks. A thread block is defined as a patch of threads that are executed on a single multiprocessor. A maximum of 512 threads can be housed in a single thread block. Each thread performs its operations independently and halts when a synchronization barrier is reached. *While GPUs can do double precision, most advantage is gained on single precision computations, and double precision is advised only when it is absolutely essential for algorithmic correctness.* Newer GPUs that are to be released in coming years relax this restriction. In all our experiments in this paper, we have used only single precision computations.

The main processor (the *host*) controls the computations and provides the data on which the GPU (the *device*) can work on. This data is generally transferred from the host memory to the device's global memory.

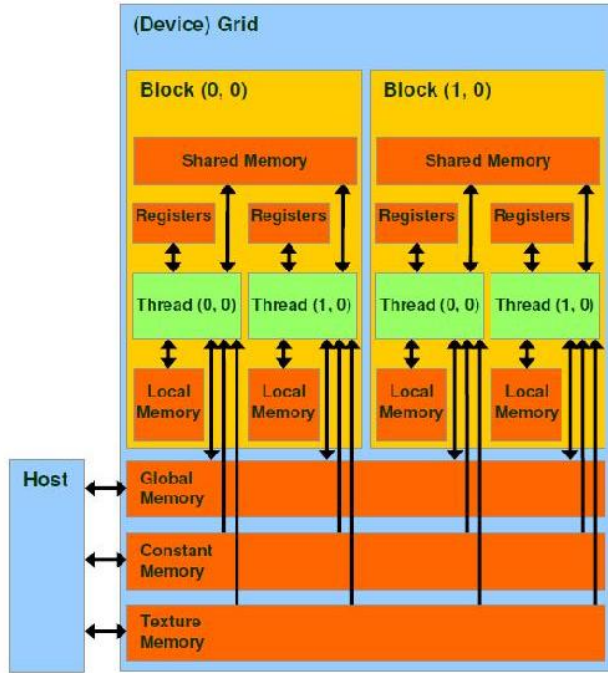


Figure 2: Logical organization of the GPU memories as seen through CUDA (Image from [12])

The global memory is large enough to hold many of the large datasets usually encountered ( $\sim 4\text{GB}$  on current GPUs). It is important to note that access times to different memories in the device are significantly different. Accesses to global memory are the most expensive and it takes approximately 400 clock cycles for one access. However, if each thread in a block accesses consecutive global memory locations, it takes lesser time than a random access. This is referred as *memory coalescing*. Accesses from the cached constant and texture memory, which can be written to from the host, are cheaper. Read and write local memory is provided by shared memory (which is shared between all processors in a multiprocessor), and per-processor register memory, and takes only as long as one instruction. The key difference between an efficient algorithm on a sequential processor and a graphics processor is that the former requires to have as less computation as possible while the latter needs to minimize memory access to and from the global memory. In other words, an efficient GPU algorithm should ensure a coalesced transfer of data from the global memory to the local shared memory, a parallelization strategy that results in most of the work being done on data that is in local registers or shared memory, and a well defined patterns of access to global memory.

### 3 Accelerating kernel methods on GPUs

**Kernel summation:** We will refer to the data points  $x_i$  in Eq. (1.1) as *source* points, and the points at

which the kernel sums are evaluated as *evaluation* points (in line with  $N$ -body algorithms, which have a similar computational structure). There are several ways each thread can be designed. One obvious parallelization approach is to assign each thread to process the effect of each source point, another one is to assign each thread to process individual evaluation points independently.

If each thread is assigned to evaluate the effects of a particular source on all evaluation points, it would have to update the value at each evaluation point in the global memory, thus requiring a number of global memory writes, resulting in a memory inefficient algorithm. However, if each thread is assigned to evaluate a particular evaluation point, it would require only one global memory write per thread. This would however result in several global reads per thread. In this case the use of shared memory and registers can reduce the number of global accesses.

We assign each thread to evaluate the kernel sum on an evaluation point. Suppose there are  $N$  source points, each thread would be required to read  $N$  source points from global memory. We reduce the total accesses to global memory, by transferring source points to the shared memory. The shared memory is not large enough to house the entire data. So it is required to divide the data into chunks and load them according to the capacity of the shared memory (the number of source points that can be loaded to the shared memory is limited by its size and data dimension). The size of each chunk is set to be equal to the number of threads in the block, and each thread transfers one source element from the global memory to the shared memory, thus ensuring coalesced memory reads. The weights corresponding to each source is also loaded to the shared memory in the same way. Once the source data is available in the shared memory, all threads update the kernel sums involving the source points in the shared memory.

In order to further reduce the global memory accesses, we use local registers for each evaluation point and the evaluation sum. Once all the source data are evaluated, the sum in the register is written back to global memory. The algorithm is summarized in Table 1. If  $d$  is the dimension of the data points, then we use  $d + 1$  shared memory location per thread (one source point and its kernel weight) and  $d+1$  registers per thread (one evaluation point and the corresponding kernel sum). The proposed approach is generic and can be extended to any kernel, an important distinction from the CPU based approximation algorithms [8, 24, 16, 11]. **Accelerating iterative algorithms:** Several kernel machines involve solution of linear or least square systems with kernel matrices, or computation of a few

**GPU based acceleration for kernel summation**

Data: Source points  $x_i, i = 1, \dots, N$ , evaluation points  $y_j, j = 1, \dots, M$   
 Each thread evaluates the sum corresponding to one evaluation point:  
 Step 1: Load evaluation point corresponding to the current thread in to a local register.  
 Step 2: Load the first chunk of source data to the shared memory.  
 Step 3: Evaluate part of kernel sum corresponding to source data in the shared memory.  
 Step 4: Store the result in a local register.  
 Step 5: If all the source points have not been processed yet, load the next chunk, go to Step 3.  
 Step 6: Write the sum in the local register to the global memory.

Table 1: *Data-parallel kernel summation on the GPU*

eigenvalues and eigenvectors of a kernel matrix. Iterative approaches are used for problems of this type. These include conjugate gradients [6] for Gaussian process regression, power iterations for eigenvalue computations, more sophisticated Krylov and Arnoldi methods, and others. In each case, the core computation per iteration is the matrix vector product, with the computation of a residual or error term. We discuss individual cases in the experimental section. As far as GPU implementations are concerned, accelerations are achieved by having the above sum evaluated in the iterative procedure. Better speedups can be obtained if the data between iterations are allowed to stay on the GPU, thus avoiding data transfers between host and device. The other way to accelerate these algorithms is to reduce the number of iterations via techniques such as preconditioning. GPU implementation of these are subject of current research.

**Accelerating kernel matrix decompositions:** Several matrix decompositions are already available on the GPU [23] and here, the strategy is to use CPU algorithms, with part of the computation performed on the GPU. To accelerate kernel methods, these libraries can be used as is, similar to the way Lapack and other libraries are used to accelerate CPU versions of kernel methods. Given the training and test data, the kernel matrix needs to be constructed before decomposition. Constructing the matrix has a computational complexity of  $O(dN^2)$  in most kernels,  $d$  being the dimension of the input data. For higher dimension ( $d > 50$ ) the matrix construction cost can become as significant as the decomposition itself, because of the availability of optimized implementations of the decomposition algorithms. However, if the structure of the kernel matrices is utilized, the matrix construction can also be parallelized on the GPU. Notice that, the matrix decompositions return a matrix which requires  $O(N^2)$  memory, and memory requirements in these approaches cannot be reduced by generating the kernel matrices on-the-fly as in the kernel summation.

We construct the matrix on the GPU and utilize this matrix for decomposition using approaches in [23].

The proposed approach is summarized in Table 2, where we use “*mat-decomp*” to denote the algorithms in [23]. We load a chunk of the training and test points to the shared memory and generate the block of the kernel matrix that involving these training/test points in the current thread block. This is repeated across all the thread blocks to construct the entire kernel matrix, which can be used for matrix decomposition.

**4 Experiments**

We tested the GPU accelerated kernel methods with three classes of problems. The first problem class used the accelerated summation on different kernels and tested the speedup on a synthetic data. Further we extended our approach to speed-up kernel density estimation. We also compare GPU based Gaussian kernel summation with a linear algorithm, FIGTREE [11]. In the second problem class, we looked at different iterative approaches which employ the kernel summation over each iterations. Finally we look at algorithms that use kernel matrix decompositions and accelerate them using our acceleration (Table 2).

**Host and device:** In all experiments the host processor is an Intel Xeon Quad-Core 2.4GHz with 4GB RAM. The GPU is a Tesla C1060 which has 240 cores arranged as 30 multi-processors. It has 4GB of global memory, 16384 registers per thread block and 16kB shared memory per multiprocessor.

For all experiments, GPU codes were written in CUDA and compiled with Matlab linkages. Similarly, the CPU codes were written in C++ with Matlab linkages. This allowed for convenient execution of machine learning algorithms.

**4.1 Experiment1 - Kernel summations:** We accelerated widely-used kernels namely the Gaussian (Eq. 4.2), Matern (Eq. 4.3), periodic (Eq. 4.4) and Epanechnikov (Eq. 4.5) kernels given by,

$$(4.2) \quad K(x_i, x) = s \times \exp\left(-\frac{(d(x_i, x))^2}{2}\right),$$

$$(4.3) \quad K(x_i, x) = s \times (1 + \sqrt{3}d(x_i, x)) \times \exp(-\sqrt{3}d(x_i, x)),$$

$$(4.4) \quad K(x_i, x) = s \times \exp(-2 \sin^2(\pi * d(x_i, x))),$$

### Accelerated kernel matrix construction for matrix decomposition

Data: Source points  $x_i, i = 1, \dots, N$ , evaluation points  $y_j, j = 1, \dots, M$

Each thread evaluates one element of the kernel matrix

Step 1: Load the source points from global memory into the shared memory.

Step 2: For large data dimension which can not fit into shared memory, divide the each source vector into several chunks of constant size and load them consecutively.

Step 3: Compute the distance contribution of the current chunk in a local register, and load the next chunk. Repeat this until the complete dimension is spanned.

Step 4: Use the computed distance for evaluating the matrix entry.

Step 5: Write the final computed kernel matrix entries into global memory.

Step 6: Use the kernel matrix with *mat-decomp*

Table 2: Our accelerated kernel construction using the GPU for matrix decomposition

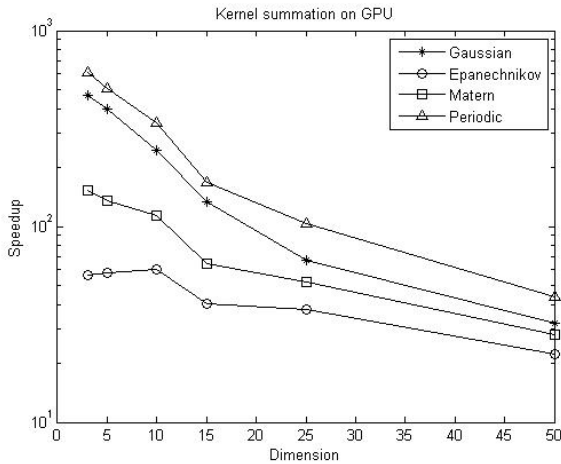


Figure 3: Speedup obtained on a GPU for various kernel summations for a data size of 10,000. The mean absolute error between the CPU and GPU based summation in all the cases were less  $10^{-5}$ .

$$(4.5) \quad K(x_i, x) = s \times (1 - d(x_i, x)^2) \times 1(d(x_i, x) < 1),$$

where  $d(x_i, x)$  is the Euclidean distance between the points  $x_i$  and  $x$ ,  $s$  is a scaling parameter and  $1(d(x_i, x) < 1)$  is an indicator function. Also there is a bandwidth  $h$  associated with distance  $d(.,.)$  such that,

$$(4.6) \quad d(x_1, x_2) = \sum_{k=1}^d \frac{\|x_{1,k} - x_{2,k}\|^2}{h^2}.$$

The synthetic data for this experiment were generated by choosing a random number between 0 and 1 uniformly for each dimension of the source and evaluation points. Datasets of varying dimensions are generated in this fashion. The resulting speedup for each kernel for a 10,000-size data is shown in Fig. 3. The speedup obtained is lower for a simple kernel like Epanechnikov, but as the complexity of the kernel increases, the speedup obtained is significant, like for pe-

riodic/Gaussian. This can be attributed to the fact that for simpler kernels, the data transfer time is more dominant than for a complex kernel. As the dimension increases, the number of threads that can be accommodated on each processor is reduced to fit the data in the shared memory, and hence there is a reduction in the speedup.

**Comparison with FIGTREE:** There are several approximation algorithms that evaluate the kernel summations in linear time, for example, FIGTREE [11] for the Gaussian kernel. In spite of the speedups obtained by the GPU-based approach, the asymptotic dependence on data size is still  $O(N^2)$ . Therefore, a linear approach like FIGTREE [11] will outperform it at some point. In order to explore this, we compared the performance of our GPU algorithm with FIGTREE which combines two popular linear approximation algorithms, Improved Fast Gauss Transform (IFGT) [24, 16] and tree based approaches, and automatically chooses the fastest method for a given data. We expected FIGTREE to beat our implementation at some point, this was observed only for a data size greater than 128,000 as seen in Fig. 4a.

Although, the linear approach eventually outperforms the GPU version, the performance of these approaches are found to have great dependence on data dimensions and kernel bandwidth. Also, these approaches are restricted to the Gaussian kernel and require a fixed bandwidth over the data. In contrast, our GPU based approach can be used with any kernel for varying bandwidths. We compared the performance of our algorithm with FIGTREE for various dimensions and kernel bandwidths and the results are shown in Fig. 4b and Fig. 4c. While the GPU approach performs consistently across bandwidths, the automatically tuned approach has varying performances across bandwidths. It was also observed that the speedup obtained by the GPU approach over FIGTREE increases with data dimensions.

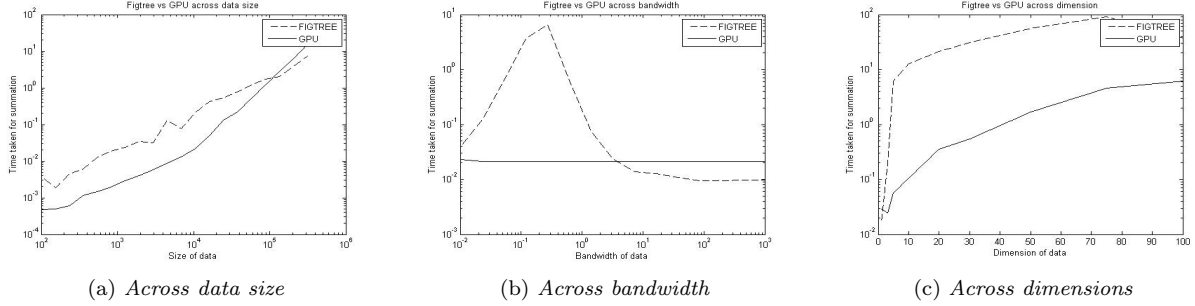


Figure 4: Speedups obtained on the Gaussian kernel compared to FIGTREE[11], the linear algorithm

Mean CPU time for Gaussian kernel	25.144s
Mean GPU time for Gaussian kernel	0.022s
Mean absolute error between estimates	$\sim 10^{-7}$
Mean CPU time for Epanechnikov kernel	25.117s
Mean GPU time for Epanechnikov kernel	0.011s
Mean absolute error between estimates	$\sim 10^{-7}$

Table 3: Performance of kernel density estimation on the 15 normal mixture densities in [10] for a data size of 10000

**Kernel Density Estimation (KDE):** KDE (or Parzen window based density estimation) is a non-parametric way of estimating the probability density function of a random variable. Given a set of observations  $D = \{x_1, x_2, \dots, x_N\}$ , the density estimate at a new point  $x$  is given by,

$$(4.7) \quad f(x) = \frac{1}{Nh} \sum_{i=1}^N K\left(\frac{x - x_i}{h}\right).$$

Two kernels widely used for density estimation are the Gaussian kernel (Eq. 4.2) and the Epanechnikov kernel (Eq. 4.5). We performed KDE based on GPU acceleration on the 15 normal mixture densities in [10] and compared performance with a direct approach. The error between the two approaches was less than  $10^{-6}$  for each distribution. The results are tabulated in Table 3.

**4.2 Experiment2 - Iterative approaches using kernel summations:** In this experiment, we explored different algorithms, that uses the kernel summation (Eq. 1.1) within an iterative algorithm like conjugate gradient.

**Mean Shift Clustering:** Mean shift clustering [4] is a non-parametric clustering approach based on kernel density estimation. It involves running a **gradient ascent** over the kernel density estimate in order to move each data point towards the local mode. Finally, it returns the set of modes (centers) to which each data point converges. A typical result of mean-shift based clustering on an image is shown in Fig. 5.

In this experiment, we apply our approach based density estimates over each iteration of the gradient

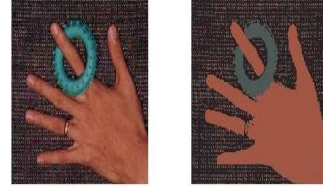


Figure 5: An example of mean shift clustering on a color image **Left:** Original Image; **Right:** Segmented image

Mean CPU time	390.847s
Mean GPU time	0.369s
Mean abs error	$< 10^{-5}$

Table 4: Performance on the optimal bandwidth estimation problem on the 15 normal mixture densities in [10] for a data size of 10000

ascent to speed up the mean shift clustering approach in [4]. For the image in Fig. 5, it was observed that the naive direct implementation took almost 13.5 hours, whereas the corresponding GPU implementation took only 35s for clustering.

**Optimal Bandwidth Estimation:** Determining the bandwidth  $h$  of the kernel in density estimation is of paramount importance for the performance of the estimator [19]. For a Gaussian kernel, there have been several approaches to evaluate the optimal bandwidth for a given data points. We looked at the plug-in approach in [18] and accelerated it using the GPU. The key computation in the bandwidth selection approach in [18] is the evaluation of a weighted sum of the *GaussianDerivative* kernel,

$$(4.8) \quad K(x_i, x) = s \times H_r(d(x_i, x)) \times \exp\left(-\frac{(d(x_i, x))^2}{2}\right),$$

where  $H_r$  is the  $r^{th}$  Hermite polynomial. We used a GPU based *GaussianDerivative* kernel summation with a **Levenberg-Marquardt based non-linear least squares** to solve for the bandwidth of the Gaussian kernel like in [18] and the results are shown in Table 4.

**Gaussian Process Regression:** Gaussian process regression is a probabilistic kernel regression approach

Dataset	$dxN$	CPU Time (s)	GPU Time (s)
Diabetes	2x43	0.0473	0.1639
Abalone	7x4177	235.8	0.79
Bank7FM	8x4499	631.2	2.19
CompAct	22x8192	1884.2	9.3
PumaDyn8NH	8x4499	467.69	1.72
Stock	9x950	13.34	0.27

Table 5: Performance on Gaussian process regression with standard datasets;  $d$  denotes the input dimension and  $N$  the size of the input data. The mean absolute error in each case was less than  $10^{-5}$  for a the Gaussian covariance (kernel) (Eq. 4.2)

which uses the prior that the regression function ( $f(X)$ ) is sampled from a Gaussian process. For regression, it is assumed that a set of datapoints  $D = \{X, y\}_{i=1}^N$ , where  $X$  is the input and  $y$  is the corresponding output. The function model is assumed to be  $y = f(x) + \epsilon$  where  $\epsilon$  is a Gaussian noise with variance  $\sigma^2$ . Rasmussen et al. [15] use the Gaussian process prior with a zero mean function and has a covariance function defined by a kernel  $K(x, x')$  which is the covariance between  $x$  and  $x'$ , i.e.  $f(x) \sim GP(0, K(x, x'))$ . They show that with this Gaussian process prior, the posterior of the output  $y$  is also Gaussian with mean  $m$  and covariance  $V$  given by,

$$\begin{aligned}
 m &= k(x_*)^T (K + \sigma^2 I)^{-1} y \\
 V &= K(x_*, x_*) - k(x_*)^T (K + \sigma^2 I)^{-1} k(x_*)
 \end{aligned}$$

where  $x_*$  is the input at which prediction is required and  $k(x_*) = [K(x_1, x_*), K(x_2, x_*) \dots, K(x_N, x_*)]$ . Here  $m$  is the prediction at  $x_*$  and  $V$  the variance estimate of prediction. Some popular kernels used with Gaussian process regression are Gaussian (Eq. 4.2), Matern (Eq. 4.3) and periodic kernels (Eq. 4.4).

The parameters of the kernels  $s$  and  $h$  are called the *hyperparameters* of the Gaussian process and there are different approaches to estimate these [15]. Given the hyperparameters, the core complexity in Gaussian processes involves solving a linear system involving the kernel covariance matrix and hence is  $O(N^3)$ . Gibbs et al. [6] suggest a *conjugate gradient* based approach to solve the Gaussian process problem in  $O(kN^2)$ ,  $k$  being the number of conjugate gradient iterations. Over each iteration of the conjugate gradient, the key computation is a weighted summation of the covariance kernel functions. In this experiment, we used our GPU based kernel summation to speed up each iteration of the conjugate gradient. Table 5 shows the performance of Gaussian process regression on various standard datasets [22] for a Gaussian kernel.

**Learning a Ranking function:** In information retrieval, a ranking function is a function that ranks data matching a given search query point according to their relevance. In order to rank the data, a preference relation needs to be learned, which is given by the ranking function. A ranking function  $f$  maps a pair of data-points to a score value, which can be sorted for ranking. There are several approaches to learn the ranking function for a given dataset. We particularly looked at the preference graphs based approach in [17].

Raykar et al. [17] maximize the generalized Wilcoxon-Mann-Whitney (WMW) statistic [14] using a *non-linear conjugate gradient* approach to learn the ranking function. Instead of maximizing the WMW statistic, they use a continuous surrogate based on the log-likelihood, thus maximizing a relaxed statistic. They use a sigmoid function to model the pair-wise probability, and approximate it using a *erfc* function. They thus reduce the core computation of the ranking problem to the evaluation of a weighted sum of *erfc* functions. A linear algorithm for the summation of the *erfc* functions is proposed in [17], which is then used for the efficient learning of the ranking function.

We use GPU based summation of *erfc* functions for learning the ranking function and compare it with the linear approach in [17] on the datasets used in [17] and the results are shown in Table 6. We also report the absolute error in the WMW statistic evaluated on the training and testing data, based on the ranking functions learnt from the approach in [17] and our GPU based approach. It can be seen that our approach consistently outperforms the approach in [17]. Note that the approach in [17] is linear in computational complexity, whereas our approach is quadratic and still outperforms the linear approach for large datasets.

**4.3 Experiment3: Matrix decompositions** Often, in many algorithms, it is required to perform LU, QR or Cholesky decomposition of the kernel matrices, for example [2] and a direct decomposition will have a cubic complexity. The GPU-based summation cannot be used in these scenarios, but however there are many algorithms for performing efficient matrix decompositions accelerated on GPUs. In this experiment, we discuss these approaches for kernel matrices.

Volkov et al. [23] claim that their implementation is the fastest LU, QR and Cholesky decomposition as of 2008. We adapted their approach for kernel matrices in this experiment. As the dimension of the input data increases, the construction of the kernel matrices becomes the dominant part of the decomposition, due to the high efficiency of these algorithms. In order to illustrate this, we performed Cholesky and QR decompositions of kernel matrices for data of various dimen-

Dataset	$dxN$	Raykar [17]	GPU	Error in WMW on training data	Error in WMW on test data
Auto	8x392	0.7499s	0.5280s	0.000	0.000
California Housing	9x20640	105.2s	27.82s	0.003	0.003
CompAct	22x8192	5.67	5.56s	0.000	0.003
Abalone	8x4177	9.838s	5.003s	0.020	0.020

Table 6: Performance of WMW-statistic based ranking, GPU based approach vs the linear algorithm in [17], error reported is the total error on the WMW statistic

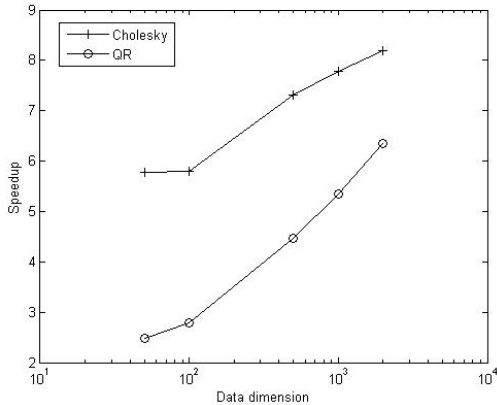


Figure 6: Speedup obtained for Cholesky and QR decomposition of a Gaussian kernel matrix, based on [23] on a 10,000-size data (size of the kernel matrix) for various dimensions of the input data; the speedups reported here are for the matrix construction on GPU using Table 2 and decomposition using [23] against the matrix construction on CPU and GPU decomposition using [23]. The mean absolute error in each case was less than  $10^{-4}$ .

sions for a 10,000-size synthetic data (generated as before). We decomposed a Gaussian kernel matrix on the GPU using [23], compared the performance for the matrix constructed on the CPU with the GPU counterpart, and the results are shown in Fig. 6. As the data dimension increases, the speedup obtained also increases. This suggests that for large dimensions, matrix construction takes significant time and hence the proposed approach would increase speedup.

**Spectral Regression for Kernel Discriminant Analysis:** Linear discriminant analysis (LDA) is a statistical projection approach, where the projections are obtained by maximizing the inter-class covariance, while simultaneously minimizing the intra-class covariance. For non-linear problems, the LDA is performed on the kernel space and is termed as Kernel discriminant analysis (KDA). KDA requires the eigen decomposition of the kernel matrix and hence is computationally expensive for large datasizes. Cai et. al [2] address this problem by casting the eigen decomposition problem

DataSize	Direct	GPU
1000	0.61s	0.2830s
2500	4.41s	2.1337s
5000	22.06s	12.467s
7500	60.30s	37.28s

Table 7: SRKDA [2] - Comparison between the GPU implementation and a CPU implementation on Caltech-101 dataset [5]; Mean absolute error (measured on the projection of a test data of size 100) in each case was  $\sim 10^{-5}$

as a spectral regression and have proposed a spectral-regression based KDA (SRKDA). At the core of SRKDA is a Cholesky decomposition of the kernel matrices, this has resulted in a 27 times theoretical speedup of SRKDA over KDA.

However, the Cholesky decomposition remains the core computational task of SRKDA, and its cubic complexity is still computationally expensive for large datasizes. We address this problem using the proposed kernel matrix decomposition. We performed SRKDA-based decomposition on the SIFT features extracted from the 10 classes of the CalTech-101 dataset [5]. The results are tabulated in Table 7. It is evident that there is a significant improvement in the performance compared to a direct implementation.

## 5 Conclusions

In this paper we have looked at accelerating popular kernel approaches using the GPU. We have reported the speedups obtained for the summation and decomposition of various kernels on GPUs. Our approaches are not just limited to the kernels reported and can be extended to any generic kernel. Further, we have shown the improvement in performance in different kernel machines like kernel density estimation, Gaussian process regression, learning a Ranking function and kernel discriminant analysis using spectral regression. We have also compared our performance with a linear algorithm [11]. With the increasing speeds in GPUs compared to the CPUs (as shown in Fig. 1), the performance can only improve further and can provide effective solution to computation bottlenecks in various kernel machines. We have made these algorithms in CUDA with Matlab

linkages available under LGPL as an opensource<sup>1</sup> and we hope to keep evolving the library. Possible areas of improvement include the following.

**Accelerating linear algorithms:** In Section 4.1, we compared our implementation with a linear version of Gaussian kernel summation. It was evident that inspite of the speedup obtained by our approach, a linear algorithm will eventually beat it. Motivated by this, we tried to map parts of the linear algorithm in [11] on the GPU and achieved speedups up to 3X for some stages. The chief bottle-neck here is the construction of underlying data-structures and if there can be an appropriate map of the data structures to the GPU, the speed up can be substantially increased like in [7].

**Accelerating kernel summation for higher dimensions:** The proposed approach for kernel summation is limited by the size of the shared memory of the graphics processor. Thus, for very large dimensions ( $> 2000$ ), it is impossible to house even one datapoint in shared memory. While it is still possible to use the GPU to improve the performance by using all the data from the global memory, this approach will be suboptimal. This is particularly interesting in the context of problems with a large number of training and test points, with each point in a very high dimensional space.

## References

- [1] C. BISHOP, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer-Verlag New York, Inc., 2006.
- [2] D. CAI, X. HE, AND J. HAN, *Efficient kernel discriminant analysis via spectral regression*, in IEEE International Conference on Data Mining, IEEE Computer Society, 2007, pp. 427–432.
- [3] B. CATANZARO, N. SUNDARAM, AND K. KEUTZER, *Fast support vector machine training and classification on graphics processors*, in International Conference on Machine Learning, 2008, pp. 104–111.
- [4] D. COMANICIU AND P. MEER, *Mean shift: a robust approach toward feature space analysis*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 24 (2002), pp. 603–619.
- [5] L. FEI-FEI, R. FERGUS, AND P. PERONA, *Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories*, 2004, p. 178.
- [6] M. GIBBS AND D. MACKAY, *Efficient implementation of Gaussian processes*, tech. rep., 1997.
- [7] N. GUMEROV AND R. DURAISWAMI, *Fast multipole methods on graphics processors*, Journal of Computational Physics, 227 (2008), pp. 8290–8313.
- [8] D. LEE, A. GRAY, AND A. MOORE, *Dual-tree fast Gauss transforms*, in Advances in Neural Information Processing Systems 18, 2006, pp. 747–754.
- [9] S. LUKASIK, *Parallel computing of kernel density estimates with MPI*, in International conference on Computational Science, Springer-Verlag, 2007, pp. 726–733.
- [10] J. MARRON AND M. WAND, *Exact mean integrated squared error*, The Annals of Statistics, 20 (1992), pp. 712–736.
- [11] V. MORARIU, B. SRINIVASAN, V. RAYKAR, R. DURAISWAMI, AND L. DAVIS, *Automatic online tuning for fast Gaussian summation*, in Advances in Neural Information Processing Systems, 2008.
- [12] NVIDIA, *NVIDIA CUDA Programming Guide 2.0*, 2008.
- [13] J. OHMER, F. MAIRE, AND R. BROWN, *Implementation of kernel methods on the GPU*, in Digital Image Computing: Techniques and Applications, Dec 2005, pp. 543–550.
- [14] G. OMER, R. ROSALES, AND B. KRISHNAPURAM, *Learning rankings via convex hull separation*, in Advances in Neural Information Processing Systems, 2006, pp. 395–402.
- [15] C. RASMUSSEN AND C. WILLIAMS, *Gaussian Processes for Machine Learning*, The MIT Press, 2005.
- [16] V. RAYKAR AND R. DURAISWAMI, *The improved fast Gauss transform with applications to machine learning*, in Large Scale Kernel Machines, 2007, pp. 175–201.
- [17] V. RAYKAR, R. DURAISWAMI, AND B. KRISHNAPURAM, *A fast algorithm for learning a ranking function from large-scale data sets*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 30 (2008), pp. 1158–1170.
- [18] S. SHEATHER AND M. JONES, *A reliable data-based bandwidth selection method for kernel density estimation*, Journal of the Royal Statistical Society, 53 (1991), pp. 683–690.
- [19] B. SILVERMAN, *Density Estimation for Statistics and Data Analysis*, Chapman & Hall/CRC, April 1986.
- [20] D. STEINKRAUS, I. BUCK, AND P. SIMARD, *Using GPUs for machine learning algorithms*, in International Conference on Document Analysis and Recognition, vol. 2, Sep 2005, pp. 1115–1120.
- [21] D. THANH-NHGI AND V. NGUYEN, *A novel speed-up SVM algorithm for massive classification tasks*, July 2008, pp. 215–220.
- [22] L. TORGO. Available at <http://www.liaad.up.pt/~ltorgo/Regression/DataSets.html>.
- [23] V. VOLKOV AND J. DEMMEL, *LU, QR and Cholesky factorizations using vector capabilities of gpus*, Tech. Rep. UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.
- [24] C. YANG, C. DURAISWAMI, AND L. DAVIS, *Efficient kernel machines using the improved fast gauss transform*, in Advances in Neural Information Processing Systems, 2004.

<sup>1</sup>[www.umiacs.umd.edu/users/balajiv/GPURL.htm](http://www.umiacs.umd.edu/users/balajiv/GPURL.htm)