Parallel Kernel K-Means on the CPU and the GPU

Mohammed Baydoun[1], **Mohammad Dawi**[1], **and Hassan Ghaziri**[1]
[1]Beirut Research and Innovation Center, Beirut, Lebanon

Abstract – *K-Means is probably the leading clustering algorithm with several applications in varying fields such as image processing and patterns analysis. K-Means has been the basis for several clustering algorithms including Kernel K-Means. In machine intelligence and related domains Kernelization transforms the data into a higher dimensional feature space by calculating the inner products between the different pairs of data in that space. This work targets Kernel K-Means and presents parallel implementations of the clustering algorithm using CUDA on the GPU and OpenMP, Cilk-Plus and BLAS on the CPU. The implementations are tested on different datasets leading to different speedups with CUDA achieving the faster runtimes.*

Keywords: Kernel K-Means; clustering; CUDA; OpenMP; BLAS

1 Introduction

Clustering consists of finding partitions of a data set such that similar data points are grouped together. This definition is vague on purpose. In fact, there is no standard definition to clustering and it remains at the end a subjective procedure allowing the end user to detect some hidden structures or regularities inside the data set. Clustering has a wide variety of applications such as data mining, pattern recognition, knowledge discovery, text mining, and many others [1].

Definitely, there are several clustering algorithms and perhaps the most famous one is K-Means due to its simplicity and behavior. K-Means has had several modifications and amongst the notable ones are those that utilize Kernelization which are generally termed Kernel K-Means. These methods are used to bypass some of the limitations of K-Means related to data representations. Kernel Methods are mainly used since the data in its raw representation often needs to be explicitly transformed into feature vector representation via a user-specified feature map. Kernel methods are relatively computationally cheap methods for achieving this objective using a selected kernel, i.e., a similarity function over pairs of data points in raw representation. Kernel methods owe their name to the use of kernel functions, which enable them to operate in a high-dimensional implicit feature space [2].

High Performance computing is a major area of research. In particular, parallel programming leads to implementing a parallel version of the required algorithm while targeting a suitable hardware. The aim is essentially speeding up the performance of the algorithm in comparison with other hardware. While a sequential or traditional program runs serially, a parallel program utilizes the independences or parallelizable parts of the algorithm to speed-up the performance taking into consideration the used hardware and its limitations. It is worth mentioning that the speedup of any program is limited by its sequential part according to Amdahl's law.

Applications to parallel computing and programming are limitless since there are limitless algorithms and various parallel architectures or platforms. These include Field Programmable Gate Arrays (FPGA), Graphics Processing Units (GPU), Multi-Core Central Processing Units (CPU), Networks or Clusters, ASICs, and others developed for the main purpose of enhancing performance of various algorithms.

In regards to CPUs, several tools exist and perhaps the most common one is Open Multi Processor (OpenMP). Others include PThreads and Cilk Plus. Concerning GPUs, the main available tools are General Purpose Programming on the GPU (GPGPU), Open Computing Language (OpenCL), and Compute Unified Device Architecture (CUDA).

This work addresses using the CPU and implements Kernel K-Means using OpenMP and Cilk Plus. In addition, we target the GPU and provide the Kernel K-Means implementation on an Nvidia CUDA capable GPU.

Also, this work considers various artificial and real datasets having different numbers of patterns and features without accounting for big data cases which is definitely an important issue but is a problem on its own and it can be targeted in future works. It is important to note that in order to obtain accurate comparisons; this work optimizes the serial and the parallel versions of the algorithm on the handled architectures.

The remainder of the paper discusses the different ideas regarding the proposed Kernel K-Means implementations. In section II, we provide a brief review of previous literature on the related subjects. Afterwards, section III explains the Kernel K-Means clustering algorithm. Section IV details the serial and parallel CPU related implementations and section V dwells on the CUDA GPU implementation. After that,

section VI presents the main results and the last section provides the conclusion.

2 Literature Review

K-Means have been the subject of much research. Kernel K-Means is part of this research, where works concentrate on the accuracy of the clustering algorithm. The main ideas related to Kernel K-Means were discussed in [2]. Other ideas were discussed in different research. The work in [3] added a kernel step to the global K-Means algorithm discussed in [4] to deal with nonlinearly separable clusters. In [4] the work proposed a way to eliminate high sensitivity of K-Means to the initial guess. In [3], the authors also presented a fast version to get comparable running times. In [5], the authors addressed implementing the Kernel K-Means algorithm on large data. They discussed changing the clustering order from the sequence of the sample to the sequence of the Kernel, which enabled an efficient way of handling the Kernel Matrix along with using all the available disk space. The work in [5] aimed at improving the speed of the kernel k-means by using a new kernel function termed conditionally positive definite kernel (CPD). They mentioned getting running times superior to that of the K-means clustering algorithm on artificial and real data. In regards to parallelizing the Kernel K-Means algorithm, previous work is lacking and therefore, we consider the literature related to parallel implementations of K-Means.

The work in [6] implemented several algorithms on the GPU and K-Means was one of them. Moreover, the work in [7] discussed the first implementation on a CUDA device by mainly parallelizing the part accounting for the minimum distance calculation. In [8], the work provided a detailed CUDA implementation of K-Means and discovered that a speedup of 14 times is possible in comparison to a single threaded CPU implementation. Also, MapReduce was used in several works in order to present a parallel K-Means algorithm such as the algorithm in [9]. Additionally, the work in [10] considered the parallel implementation of K-Means on MPI, OpenMP and CUDA and concluded that for relatively small data, OpenMP performs the best, while for larger data, CUDA obtains the best speedup.

In [11], the objective was to utilize the advantages of both MPI and OpenMP to parallelize the K-Means algorithm, namely parallel processing inside the node and distribution of tasks inside the cluster obtained through MPI. The latter work showed good speedups especially on large datasets. Moreover, [12] focused on distributed ways to parallelize K-Means and proved scalability over large datasets.

Thus, and despite the lack of any parallel implementation of Kernel K-Means, there are lots of relevant works on the parallel implementation of the K-Means algorithm. So, the main contribution of this work lies in being the first parallel implementation, as far as we know, of Kernel K-Means using the targeted architectures or tools.

3 Kernel K-Means

Kernel K-Means utilizes the Kernelization approach to divide given data into a set of clusters using an approach that is mainly based on K-Means.

First, it is important to provide a basic idea about Kernelization. This is a common approach that is used with various algorithms including PCA, SOM, etc… It can be explained in the following.

Given a set S in the input space, its image is $\phi(S)$ in the feature space. The center of mass of set $\phi(S)$ is the vector given by:

$$\phi_{mean}(S) = \frac{1}{l}\sum_{i=1}^{l}\phi(x_i) \tag{1}$$

As with all points in the feature space we will not have an explicit vector representation of this point. However, a point x_{mean} whose image under ϕ is $\phi_{mean}(S)$ may not exist. In other words, we are now considering points that potentially lie outside $\phi(X)$ that is the image of the input space X under the feature map ϕ.

Despite this apparent inaccessibility of the point $\phi_{mean}(S)$, we can compute its norm using only evaluations of the kernel function on the inputs:

$$\begin{aligned}\left\|\phi_{mean}(S)\right\|^2 = \left\langle \phi_{mean}(S), \phi_{mean}(S)\right\rangle &= \left\langle \frac{1}{l}\sum_{i=1}^{l}\phi(x_i), \frac{1}{l}\sum_{j=1}^{l}\phi(x_j)\right\rangle \\ &= \frac{1}{l^2}\sum_{i,j=1}^{l}\left\langle \phi(x_i),\phi(x_j)\right\rangle \\ &= \frac{1}{l^2}\sum\nolimits_{i,j=1}^{l} k(x_i,x_j)\end{aligned} \tag{2}$$

Where l is the number of samples in the set $\phi(S)$, or in other words it is the number of samples per class or cluster.

Hence, the square of the norm of the center of mass is equal to the average of the entries in the kernel matrix.

Incidentally this implies that this sum is greater than or equal to zero, with equality if the center of mass is at the origin of the coordinate system. Similarly, we can compute the distance of the image of a point x from the center of mass $\phi_{mean}(S)$.

$$\begin{aligned}\left\|\phi(x)-\phi_{mean}(S)\right\|^2 &= \left\langle \phi(x)-\phi_{mean}(S), \phi(x)-\phi_{mean}(S)\right\rangle \\ &= \left\langle \phi(x),\phi(x)\right\rangle - 2\left\langle \phi(x),\phi_{mean}(S)\right\rangle + \left\langle \phi_{mean}(S),\phi_{mean}(S)\right\rangle \\ &= k(x,x) - \frac{2}{l}\sum\nolimits_{i=1}^{l} k(x,x_i) + \frac{1}{l^2}\sum\nolimits_{i,j=1}^{l} k(x_i,x_j)\end{aligned} \tag{3}$$

In regards to clustering in general and K-Means in particular, the main aim is to determine the clusters of a certain input set X = {x1,...xN}.

We first provide a basic explanation of K-Means. K-Means starts with the data randomly labeled according to a selected number of clusters. Then, and on an iterative basis, the label of each pattern is chosen based on the distance from the center of the cluster which is constantly updated until convergence.

For Kernel K-Means, the data is initially transformed into the Kernel feature space using a predefined kernel function such as the ones mentioned in the following equations:

$$k(x,y)=\exp(-\frac{\|x-y\|^2}{2\sigma^2}) \text{ (Gaussian)} \quad (4)$$

$$k(x,y)=\exp(-\frac{\|x-y\|}{2\sigma^2}) \text{ (Radial Basis)} \quad (5)$$

$$k(x,y)=\tanh(\alpha x^T y+\beta) \text{ (Sigmoid)} \quad (6)$$

$$k(x,y)=1-\sum_{n=1}^{N}\frac{(x_i-y_i)^2}{\frac{1}{2}(x_i+y_i)} \text{ (Chi-Square)} \quad (7)$$

In this work, we only utilize the Gaussian Kernel noting that other kernels can be performed in a similar manner without greatly affecting the performance of the algorithm especially that this only affects the first phase of the algorithm (Phase I), where the Kernel Matrix needs to be computed. It is worth emphasizing that this phase can be used in several other Kernel methods such as Kernel SOM [13] and Kernel PCA, so its implementation is generally useful.

In order to provide a complete explanation of the Kernel K-Means algorithm, the pseudo code of the algorithm is provided noting that we divide the algorithm into two phases.

Given a training set of *N* samples $\leftarrow X_1,.....,X_N$

(Phase I) $K_{NxN} \leftarrow$ Kernel Matrix:

$K(i,j) \leftarrow \langle \phi(X_i),\phi(X_j) \rangle = \exp(-\frac{\|X_i-X_j\|^2}{2\sigma^2})$ (If the Gaussian Kernel is used)

Number of clusters $\leftarrow k$, $cluster\ centers(C_1,.....,C_k)$

Distance matrix: D (*N* rows, *k* columns),

$D(i,j)=\left\|\phi(X_i)-\phi(C_j)\right\|^2$

Randomly assign each pattern to a cluster: Labels matrix A (*N* rows, *k* columns), label rule:

$A_{ij}=\begin{cases}1 & if\ X_i\ in\ cluster\ j;\\ 0 & otherwise\end{cases}$

change $\leftarrow$ 1
while change ≠0 do
(Phase II) Iterative
for j=1:*k* do
calculate the cluster size *l*
for i=1:*N* do
calculate distance of each sample from the mean of the cluster in feature space using:

$$\phi(C_j)=\frac{1}{l}\sum_{s=1}^{l}A_{ij}\phi(X_i)$$

$$D(i,j)=K(X_i,X_i)-\frac{2}{l}\sum_{m=1}^{N}K(X_i,X_m)+\frac{1}{l^2}\sum_{m=1}^{N}\sum_{n=1}^{N}A_{mj}A_{nj}K(X_m,X_n)$$

end for
end for
old A = A
update labels of all samples according to minimum distance
A =indices(minimum of D along the columns)
if old A = A then
change=0
end if
end while

4 CPU Implementation

This work mainly tested the algorithm using C on different artificial and real datasets with varying number of patterns and features without accounting for big data cases since this is a domain on its own and should be the subject of future work.

We optimized both the serial and the parallel implementations on a multicore CPU to attain the highest possible speedups whilst ensuring the correctness of the results.

Concerning the serial version, the implementation relies on using a Matrix-like form that utilizes functions similar to matrix and vector operations which is generally optimal in Matrix Based Software such as Matlab. This matrix like form is possible due to the nature of the problem as can be observed in the pseudo code. For example, phase I is relatively similar to the matrix multiplication, but by replacing the multiplication with a subtraction, squaring and division followed by the exponential calculation. Also, the distance calculation involves a matrix multiplication for calculating the second term in a single kernel for all the values followed by multiplying by (2/*l*). So, these can be directly optimized.

Thus, in the C sequential version, we initially implemented a direct serial version and afterwards used and modified the Open-BLAS library [14], where BLAS stands for Basic Linear Algebra Subprograms, in order to optimize the implementation using matrix-like operations. Open-BLAS includes one of the fastest sequential and parallel matrix multiplication algorithms, so we implemented similar approaches that suit the Kernel-K Means algorithm which helped achieve the faster serial solution.

Concerning the parallel multicore CPU implementation, we implemented the Kernel K-Means algorithm in C using two very common libraries. These are the OpenMP and Cilk-Plus parallel libraries. Also, and due to the matrix properties of the problem, we used Open-BLAS which allows for selecting the number of threads and is therefore parallel. The OpenMP and Cilk Plus implementations are relatively similar since we basically parallelized the sequential version by dividing the threads according to the number of patterns, which allows for the highest level of parallelism in this algorithm. In regards to the BLAS related versions, we based on the parallel BLAS version in addition to using OpenMP when BLAS was not applicable or even when it achieved lower times to obtain the fastest possible times.

5 CUDA Implementation

The CUDA implementation is rather similar to the parallel CPU one but with important differences that require explanation.

Initially, CUDA requires transferring the required data to the GPU, which is in this case the dataset itself including the patterns and the features in addition to the initial random labels, although these can be initiated on the GPU. Definitely, these variables and others as mentioned in the pseudo code require to be allocated in the GPU memory, so this can be termed (Phase 0) and is relatively time consuming depending on the size of the data. In general, memory transfers can be a bottleneck in GPU implementations, except when used with multi-streams, but this is not applicable here, since the complete data is required from the start of the Kernel K-Means algorithm.

After transferring the data, the implementation requires computing the Kernel Matrix, which is similar to matrix multiplication as already mentioned in the previous section. Thus, this can be performed using a kernel similar to the (cublas) kernels that are already available with CUDA, or to the kernels provided in the CUDA sample codes, which is adopted here using a modified version of the matrix multiplication kernel. This is probably the best possible implementation according to the CUDA guidelines.

Afterwards, the iterative phase needs to be performed. The first part accounts for counting the size of each cluster, which is simply performed using the atomicAdd function. Then, one needs to compute the distance value for each pattern with each cluster. This is composed of three terms. The first term is from the Kernel Matrix, which is already computed. The second term can be computed for all the patterns and the clusters in a similar operation to a single matrix multiplication operation as already noted in the serial section. The last term is common for each cluster meaning that there can be only a number of clusters' terms. Thus, it needs to be computed only once for all the patterns and used according to each cluster. This term is composed of two summation steps and is computed using a reduction operation. Reduction is usually used for summations and in this case we have summation of several variables that are a collection of multiplied values, which means reduction can be used. Like other matrix operations, reduction is provided in the CUDA sample codes, so these were modified to suit the calculation of this term.

After obtaining the three terms, the distance matrix can be computed, where each value is computed by a single CUDA thread while ensuring that the accesses to the variables in the memory are coalesced to achieve better speedups.

Afterwards, the minimum distance and the corresponding cluster for each pattern need to be determined. Like the distance matrix computation, this is done for each pattern in a single CUDA thread and this leads to the new labels matrix whilst checking if convergence was achieved through the Boolean "change" variable. This completes the second phase, which is repeated for a certain number of iterations until convergence is achieved.

After the completion of the algorithm, the final labels need to be transferred back to the host from the GPU, which completes the implementation.

Thus, the proposed CUDA implementation mainly relies on modifying tasks that are generally provided with the CUDA sample codes in order to achieve the best possible speedups.

6 Results

The Kernel K-Means algorithm, and as already noted in the pseudo code can be divided into two distinct phases. The first phase (PI) calculates the Kernel matrix (K), and the second phase (PII) is part of (the while loop) and is thus an iterative procedure that is required to update the distances, and obtain the clusters and the labels of the patterns. (PII) is performed for a number of times until the convergence of the algorithm.

Moreover, we can consider a phase zero (P0), which is necessary for allocating the data and copying the required values only in the CUDA case. P0 accounts for copying the data to and from the GPU.

Thus, we need to report on the results of these different phases by considering different datasets. So, we considered artificial and real datasets as noted in Table 1. Several of these datasets are shown in Figure 1. Besides the ones in the figure, we generated a Gaussian random dataset of four clusters where each pattern has three features. Also, for the MNIST dataset [15], we utilized the test data for the images of “0” and “1” which means that there are only two clusters.

Table 1. Timing Results for the different implementations

Dataset	Patterns	Features	Clusters	C (ms)		C+BLAS (ms)		OpenMp (ms)		Cilk (ms)		MP+BLAS (ms)		CUDA		
				PI	PII	PI	PII	PI	PII	PI	PII	PI	PII	P0	PI	PII
Circular	1012	2	2	13.2	3.8	4.3	3.3	5.4	1.9	4.2	1.3	1	1.9	1.1	0.17	0.6
Full Moon	1000	2	2	11.8	4.1	2.5	3.3	5.8	1.8	4.9	1.2	1	1.8	1.1	0.17	0.6
Atom	800	2	2	9.2	2.5	1.9	2.2	3	1.5	3.4	1.3	0.5	1.5	1	0.12	0.5
Chainlink	1000	3	2	12.9	3.8	2.2	3.4	5.1	2	3.4	1.6	0.9	2	1.1	0.19	0.6
Spiral	2000	2	2	60	44	18	43	15	11	15	14	9.3	25	2.9	0.65	0.5
Gaussian	2000	3	4	50	32	12.2	21	14	10	13	8.2	8.8	14	7.7	1.7	0.9
EngyTime	4096	2	2	198	65.5	69.3	59.1	54.9	22	56.4	21.4	36.9	38	10	2.7	1.1
Mnist (0,1)	2115	400	2	1877	18	91.1	15.7	356.6	6.6	359.5	6.3	37	6.6	5.4	0.92	0.6

The results are presented in Table 1 for the various implementations using the different datasets. The table shows the time in milliseconds. All the CPU related tests were performed using a 2.3GHz i7 Intel CPU with 4 cores. The GPU implementation was performed on a Tesla C2050 device.

It is worth comparing the obtained times of Kernel K-Means with that of a serial C K-Means implementation and although the timing is not mentioned, all the times for K-Means are less than 0.5 milliseconds, which indicates that Kernel K-Means is much more time consuming.

Besides, it is important to note that the CPU implementation varied according to the used CPU and the number of available cores whose increase directly meant higher speedups although this is not detailed here. Moreover, the CUDA implementation directly depends on the used GPU where a newer GPU should yield better results. The discussion related to such ideas should be provided in future works.

In regards to the errors and the accuracy of K-Means versus Kernel K-Means, there is a drastic difference with Kernel K-Means being much more accurate and having near 100% accuracy in all the cases except the Gaussian and Spiral case while the K-Means accuracy is generally much lower except when the data is linearly separable.

The results indicate that computing phase I and phase II is clearly faster on the CUDA device and even if the memory timings or phase zero is accounted for, the CUDA device still proves to be the faster albeit not by much when compared with the OpenMP+BLAS version except in relatively larger datasets such as MNIST and Energy Time.

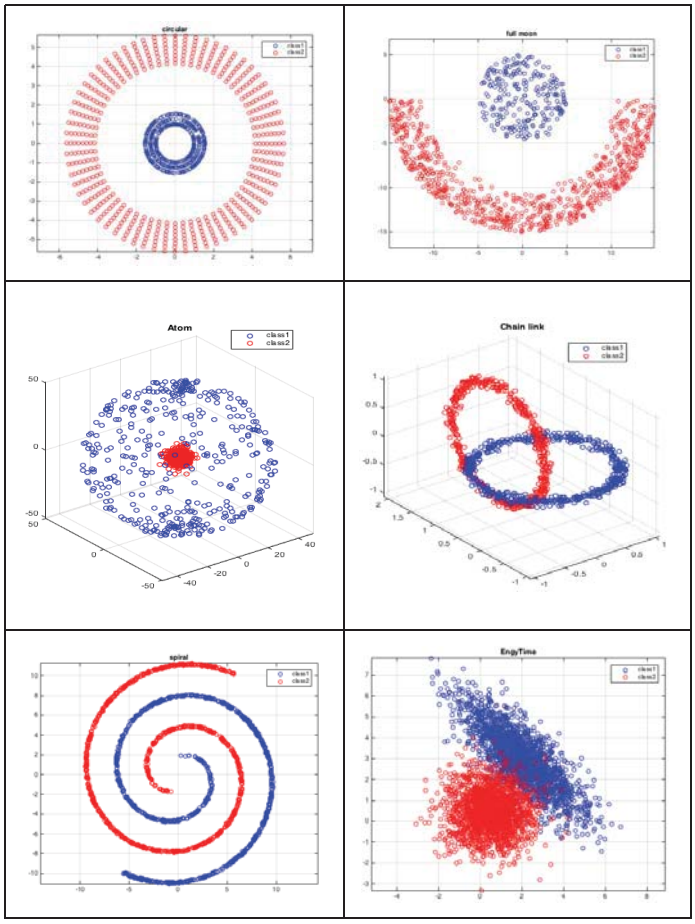

Figure 1. Circular, Moon, Atom, Chainlink, Spiral and Energy-Time Datasets

7 Conclusion

This work presented several parallel implementations of the Kernel K-Means algorithm using the CPU and the GPU. The CPU implementations involved using Cilk-Plus and OpenMP in addition to OpenBLAS due to the matrix-like nature of the problem, while the GPU used CUDA and the available sample codes. The CUDA implementation proved to yield the faster

results despite the relatively high time consumption caused by memory allocation and data transfers between the CPU and the GPU. The work also notes that there is a large room for further improving the parallel implementations with emphasis on large data.

8 References

[1] C. C. Aggarwal and C. K. Reddy, Data Clustering: Algorithms and Applications. CRC Press, 2013.

[2] J. Shawe-Taylor and N. Cristianini, Kernel Methods for Pattern Analysis. Cambridge university press, 2004.

[3] G. F. Tzortzis and A. C. Likas, "The global kernel-means algorithm for clustering in feature space," Neural Networks, IEEE Transactions on, vol. 20, pp. 1181-1194, 2009.

[4] G. Tzortzis and A. Likas, "The global kernel k-means clustering algorithm," in Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on, 2008, pp. 1977-1984.

[5] R. Zhang and A. I. Rudnicky, "A large scale clustering scheme for kernel k-means," in Pattern Recognition, 2002. Proceedings. 16th International Conference on, 2002, pp. 289-292.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," Journal of Parallel and Distributed Computing, vol. 68, pp. 1370-1380, 2008.

[7] R. Farivar, D. Rebolledo, E. Chan and R. H. Campbell, "A parallel implementation of K-Means clustering on GPUs." in Pdpta, 2008, pp. 212-312.

[8] M. Zechner and M. Granitzer, "Accelerating K-Means on the graphics processor via cuda," in Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on, 2009, pp. 7-15.

[9] W. Zhao, H. Ma and Q. He, "Parallel K-Means clustering based on mapreduce," in Cloud ComputingAnonymous Springer, 2009, pp. 674-679.

[10] J. Bhimani, M. Leeser and N. Mi, "Accelerating K-Means clustering with parallel implementations and GPU computing," in High Performance Extreme Computing Conference (HPEC), 2015 IEEE, 2015, pp. 1-6.

[11] L. M. Rodrigues, L. E. Zárate, C. N. Nobre and H. C. Freitas, "Parallel and distributed kmeans to identify the translation initiation site of proteins," in Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on, 2012, pp. 1639-1645.

[12] K. Stoffel and A. Belkoniene, "Parallel k/h-means clustering for large data sets," in Euro-Par'99 Parallel ProcessingAnonymous Springer, 1999, pp. 1451-1454.

[13] K. W. Lau, H. Yin and S. Hubbard, "Kernel self-organizing maps for classification," Neurocomputing, vol. 69, pp. 2033-2040, 2006.

[14] http://www.openblas.net/

[15] Y. LeCun, C. Cortes and C. J. Burges, The MNIST Database of Handwritten Digits, 1998.