# GPU computation for Kernel Methods in Machine Learning

Idriss Alaoui-Amini

February 2022

## 1 Introduction

The main purpose of this project is to use parallel computations to improve (drastically) Kernel matrix computations. It is linked to a course given in the master's degree *Kernel Methods for Machine Learning.* GPU computing gives us the ability to parallelize computations in order to improve performances during computations, and avoid time constraints. Yet it requires to think about the design of the problem and to make sure that it is adapted to parallel computations. From the understanding of Kernel methods, it appears that it is not only particularly fit for parallelization, but it can also be a very good introduction to the topic because of the relative ease of the problem. Fundamentally, kernel projections rely on basic vectors and matrices multiplications (see next section). In this project we will go from the limit of kernel computations (it's lack of scalability) and use a Graphic Processing Unit to improve performances. We will develop several types of kernels, and compare results to a benchmark in one of the most widely used scientific computing stack (Python's Numpy). Because machine learning and kernel methods are generally used in python, the work will be done using this language. Originally we were supposed to wrap C code in python using Cython to generate the functions, but it turns out that there is a very convenient alternative to Cython. For this project we will use Numba and its cuda implementation in python to create the necessary functions.

## 2 Kernel methods in a nutshell

If we had to give a basic explanation of kernels, it would just be the projection of some data in what is called a Reproducible Kernel Hilbert Space space RKHS such that it becomes linearly separable. In order to do this, a Gram matrix $\mathbf{K} \in \mathbb{R}^{N\times N}$ is computed. The Gram matrix corresponds to a square matrix where each element $K_{ij}$ is a scalar product of a kernel function $\Phi : \mathcal{X} \mapsto \mathcal{H}$, and is defined like this: $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle_{\mathcal{H}}, \quad \forall i, j \in \{1, \dots, N\}$.

The function $\Phi$ can be any known kernel. For the sake of demonstration, only three kernels will be developped, the linear kernel, the polynomial kernel and the Gaussian kernel. These three kernels will be enough to make the point and are very common.

### 2.1 The linear kernel

The most basic kernel used when data are linearly separable. It is used for the sake of example.

$$K(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{x}\boldsymbol{x'}^T$$

### 2.2 The Gaussian kernel

The Gaussian kernel or radial basis function (RBF) kernel appears quite obviously to be a similarity measure where $||\boldsymbol{x} - \boldsymbol{x'}||^2$ corresponds to a squared Euclidean distance between the two vectors $\boldsymbol{x}$ and $\boldsymbol{x'}$. It is perhaps the most used kernel. But it is also one that scales very poorly either because of the large size of the sample or the large size of the features.

$$K(\boldsymbol{x}, \boldsymbol{x'}) = \exp\left(\frac{||\boldsymbol{x} - \boldsymbol{x'}||^2}{2\sigma^2}\right)$$

### 2.3 The polynomial kernel

The polynomial kernel consists in the projection of the vectors where the similarity is made in a larger polynomial space and their combinations are taken into account rather than just the basic similarity.

$$K(\boldsymbol{x}, \boldsymbol{x'}) = \left(\boldsymbol{x}^T\boldsymbol{x'} + c\right)^d$$

## 3 Why are Kernel methods fit for GPU computing

Before the development and advent of deep learning, kernel methods were some of the most successful and most widely spread methods of machine learning in the literature. Yet, they have a major drawback, they scale very poorly. Indeed because it relies very often on the computation of a square matrix the complexity is squared, $\mathcal{O}(n^3)$. This can cause problems when the size of data goes beyond a certain value of $n$ (typically from 10000). But because they rely solely on scalar products and element wise multiplications, they are well adapted to parallelization. Indeed, matrix multiplication is a very common theme in parallel computation and High performance computing. So there is no need to go further than these methods in order to at least bring great improvements in Gram matrix computations. Actually the first implementation would be one similar to the methods that have been presented in class.

## 4 Implementation

The implementation will go as follows. From the matrix multiplication methods seen in class, some basic kernel functions will be developped as *kernels* (in the cuda sense) to be used as algorithms to compute the kernel. As mentionned in the introduction we will rely on the high performance python compiler Numba in order to develop and test our kernels. There exist already a perfectly functioning and optimized scientific computing package in python, it is numpy. But numpy still faces lack of speed and scales rather poorly. Numba is supposed to make scientific computing in python (including numpy) much faster than pure python. It also relies a lot on Just-in-time compilation mechanisms which execute a code on the fly. In the case of our matrix multiplication, we believe that using numba should greatly enhance the performances of numpy. Furthermore, numba also contains a cuda API which allows us to code and program at the GPU level without the need to use C/C++. The syntax is very close to original cuda programming, which makes it quite straightforward to use the API. This why the code and benchmarks will be done in Python. In the second section three kernels have been presented, and only these three kernels will be implemented. Each kernel will be developped using a numpy version, a numba jit version and a cuda version. Then, for each kernel, we will make a computation with matrices of different sizes of 1024, 2048, 4096, 10240. Every time, we will compute the time it took the function to return the result. These results will be compared to each others with the benchmark being the numpy version as it is the standard way these computations are done in python (and generally in machine learning). These results will be shown in plots showing performance and speedup compared to the numpy version.

## 5 Comparing Kernel computing efficiency (regular VS GPU)

The basic hypothesis was that for all kernels, the regular numpy version should be the slowest version of them all, followed by the just-in-time numpy computation whereas the cuda implementation should always remain superior to the other two. It turns out that for both linear and polynomial kernel, the basic numpy version happens to be faster than the jit version. But for the Gaussian kernel, the jit version of is much faster than the version we have developped previously using numpy and scipy. Using scipy makes it a much slower and it is a very complicated function to deal with and to parallelize.

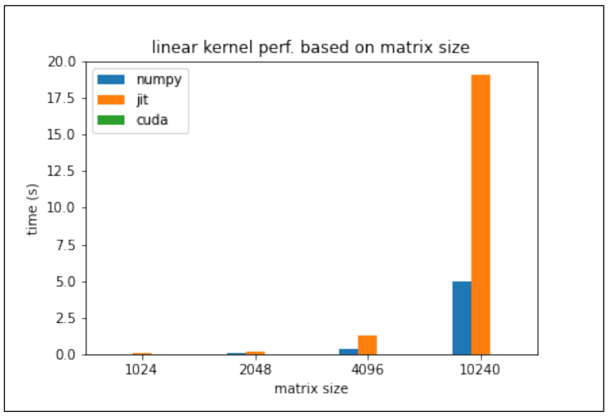

Figure 1: Linear kernel performances of a plane

Fig 1, contains performances of the linear kernel for different size of matrices. Fig 2 contains the speedup relatively to the numpy version of the algorithm. Naturally the numpy version is always 1. In the case of the linear kernel, cuda is always faster and can reach a 300 times speedup for the 10240 matrix (a lot more compared to the jit version). It appears that the computation time increases exponentially for the jit version.

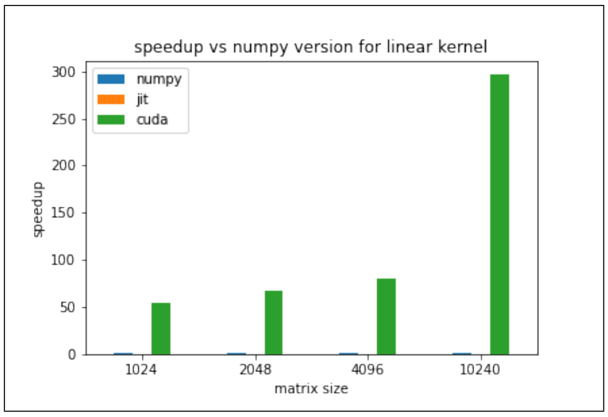

Figure 2: Linear kernel speedup

Now let us see what is happening for the polynomial kernel. Since the polynomial kernel's core is the matrix multiplication, performances should be similar to the linear kernel.

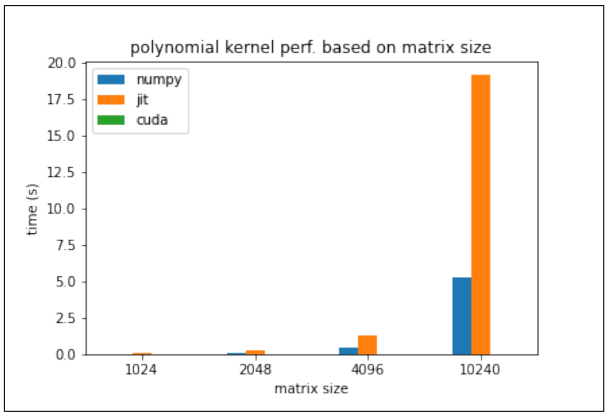

Figure 3: Polynomial kernel performances

As mentionned above, the performances are almost the same as the linear kernel. Let us now see if cuda's performance and speedup is still the same.

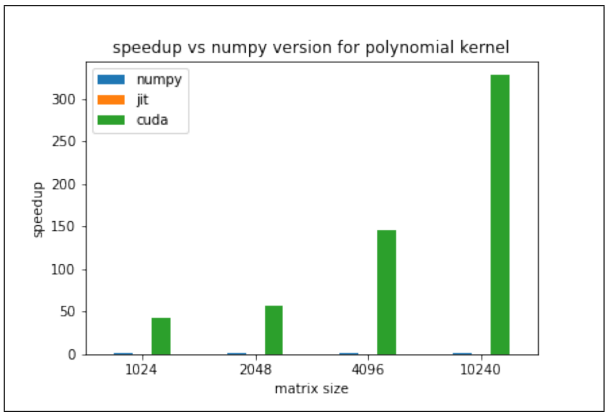

Figure 4: Polynomial kernel speedup

It is the same. Finally, the last kernel is the Gaussian kernel, and the results should be a lot more interesting.

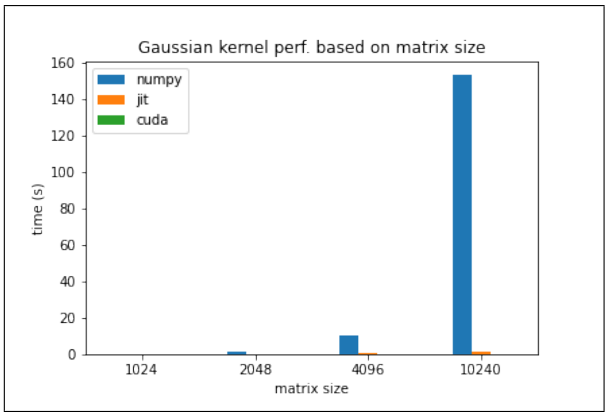

Figure 5: Gaussian kernel performances

The Gaussian kernel presents the behaviour originally expected in terms of performances. The numpy version is the slowest with an exponential increase and a poor scalability based on the size. And this time, the jit version is a lot better, because it is possible to make very good use of the vectorization made by numpy and the just in time compilation, we can get the best of both worlds. Below in Figure 6 is shown the speedup of the jit version relatively to the basic numpy version. Exceptionally, the speedup of the cuda version is not shown in this figure, because the number is so high (cuda is almost 10000 times faster than the numpy version) that it must be considered with caution. And also there are some doubts as to its results.

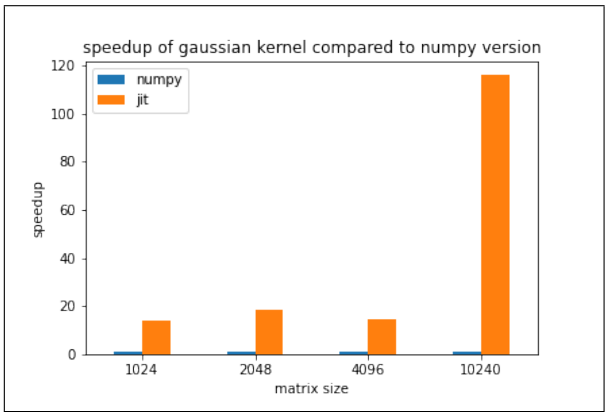

Figure 6: Gaussian kernel speedup

One last and yet important thing worth mentioning, is the fact that cuda computations take place in the GPU, but the plots shown above do not take into consideration the time to transfer the data from the device (the gpu) to the host, where we can actually view them. This operation is very time consuming when executed. And it not only reduces the speedup, it actually reverses it. The transfer from device to host makes the numpy version better than even the cuda version. This is what figure 7 below shows.

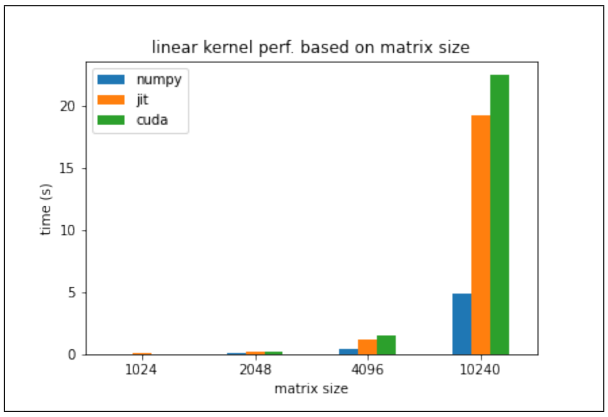

Figure 7: Linear kernel perfs with data transfer from device to host

We strongly suspect this phenomenon to be the result of a bank conflict. Because the use of shared memory has already been used to improve computation speed within cuda. The bank conflict issue happens when two threads in the same warp access different elements in the same bank. This causes memory accesses to be treated sequentially and decreases performance. It is possible to reduce bank conflict issues with the transformation shown in the course (and practice) and a very common technique that is to make sure that the shared memory parts of both matrices for multiplication follow the same order. Below are the two figures for performance and speedup without bank conflicts.

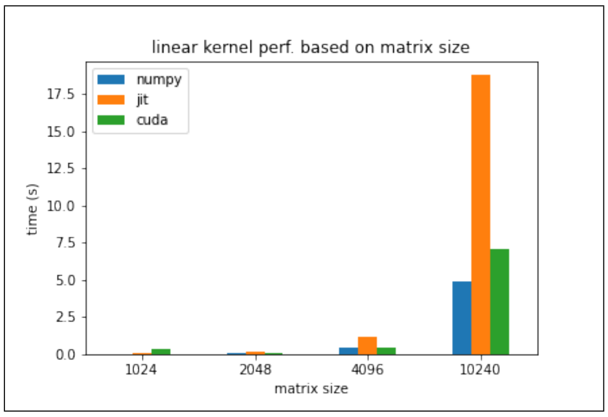

Figure 8: Linear kernel performances without bank conflicts

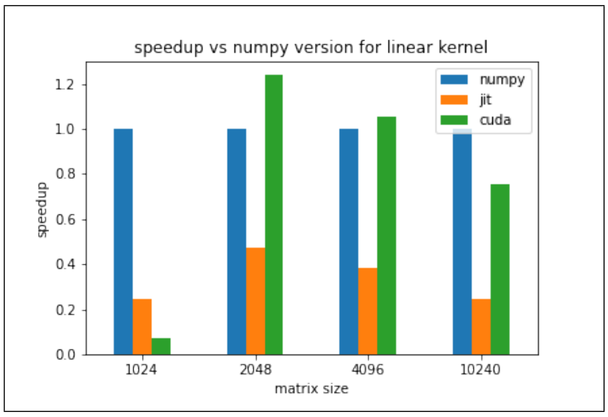

Figure 9: Linear kernel speedup with no bank conflicts

Compared to figure 7, figure 8 shows a drastic decrease in computation time for the cuda version. The cuda version becomes better than the jit version, but remains worse than the basic numpy one. This is reflected in the speed plot in figure 9.

## 6 Conclusion

The purpose of this project was to get an introduction to gpu and high performance computing. Kernel methods for machine learning happened to be a very convenient way to apply basic yet concrete methods in high performance computing with poor scalability, great complexity, and basic matrix multiplication. The results that have been shown above are a reflection of the personal work done and are by no means meant to be relevant because of the lack of mastery of the topic. Yet it allowed to understand how and when to use gpu computing. The data transfer from device to host happens to be a problem and further investigations should be made to find a faster way to transfer data. Also, the work has been done on a laptop with a Nvidia RTX 3060 Laptop GPU, which allowed to use gpu, but whose size and capacity is limited and might have been a drawback. Perhaps performances (especially in data transfer) could have been greatly improved with more resources. But it was a choice, and the objective was not as much performance as understanding. Also, numpy is a highly efficient and the most acclaimed scientific computing package right now, so it is hard to do better without expert knowledge. Yet the Gaussian kernel proved that gpu or better compilation logic can greatly improve performances and scale very well.