Università degli Studi di Università di Modena e Reggio Emilia

DIPARTIMENTO DI SCIENZE FISICHE, INFORMATICHE E MATEMATICHE
Corso di Laurea Triennale in Informatica

Tesi di laurea triennale

GPU ACCELERATION IN SVM METHODS FOR MACHINE LEARNING

Candidato:
Andrea Domenico Giuliano

Relatore:
Luca Zanni

Correlatore:
Roberto Cavicchioli

Anno Accademico 2012–2013

Contents

Introduction

In recent years, parallel architectures have evolved considerably and in the future they will become more and more popular, and so it will be necessary to develop applications able to run on multi-core platforms. The development of applications that can run in parallel is obviously more complex than the development of a classical serial application. In addition, for parallel development there is no rule that says (regardless of the programming language) what is the best method, librarie or architecture to implement parallel applications. Despite these disadvantages, there are physical limits that prevent the development of purely serial architectures and make parallel architectures more appealing: the **transmission rate** (the information can not travel faster than the speed of light) and the **limit to miniaturization**. In addition to purely physical limits there are also economic constraints to take into account: it is much less expensive to produce different architectures of normal power rather than trying to produce a single architecture extremely powerful.

For these reasons, we can understand the recent "Race to the multi-core" and the need to convert serial applications into new parallel applications, providing increased performance.

In this thesis we deal with strategies for accellerating the kernel functions evaluations used in the machine learning methodology named Support Vec-

tor Machines (SVM).

These accellerations can be very important for improving the training phase of the methodology, based on large–scale quadratic program whose solution requires many kernel computations.

The aim cosists in to highlight the possible speedups in the optmization solvers used for training SVM, due to a parallel implementation in the NVIDIA CUDA environment of the kernel computations. A wide numerical experimentation shows the numerous computational advantages of the parallel implementation, but at the same time, it emphasizes also the difficulties due to the low amount of memory available on the GPU.

Chapter 1

SVM

The methodology Support Vector Machines (SVM) is a method based on a principle of binary classification generalizable to the case where, in the initial training set, we want to consider more classes [1, 2].
In case you have only two classes, the SVM technique consists in determining an appropriate separation surface between classes by maximizing the distance of the points from the surface.
This surface is obtained by solving a quadratic programming problem and depends on only some points of the training set, called support vectors.

1.1 Optimal separating hyperplane

Consider a set S of N points $x_i \in \mathbb{R}^n$ ($i = 1, 2, \ldots, N$) for which it is known the membership class.
So it is possible to mark every event with some sort of tag y_i indicating the class of x_i.
As we will just consider a binary classification problem, y_i can only take one of two predetermined values, for example $y_i \in \{-1, 1\}$.

The goal to be reached is to determine the equation of a hyperplane that separates S while maintaining all the points of the same class on the same side, maximizes the distance from the hyperplane of the points of the two classes closest to the hyperplane.

Definition 1.1.1 *The set S is linearly separable if there is a vector $w \in \mathbb{R}^n$ and a scalar $b \in \mathbb{R}$ such that:*

$$y_i(w \cdot x_i + b) \geq 1 \qquad con \quad i = 1, 2, \ldots, N. \tag{1.1}$$

The pair (w, b) defines a hyperplane of equation

$$w \cdot x + b = 0 \tag{1.2}$$

known as separating hyperplane

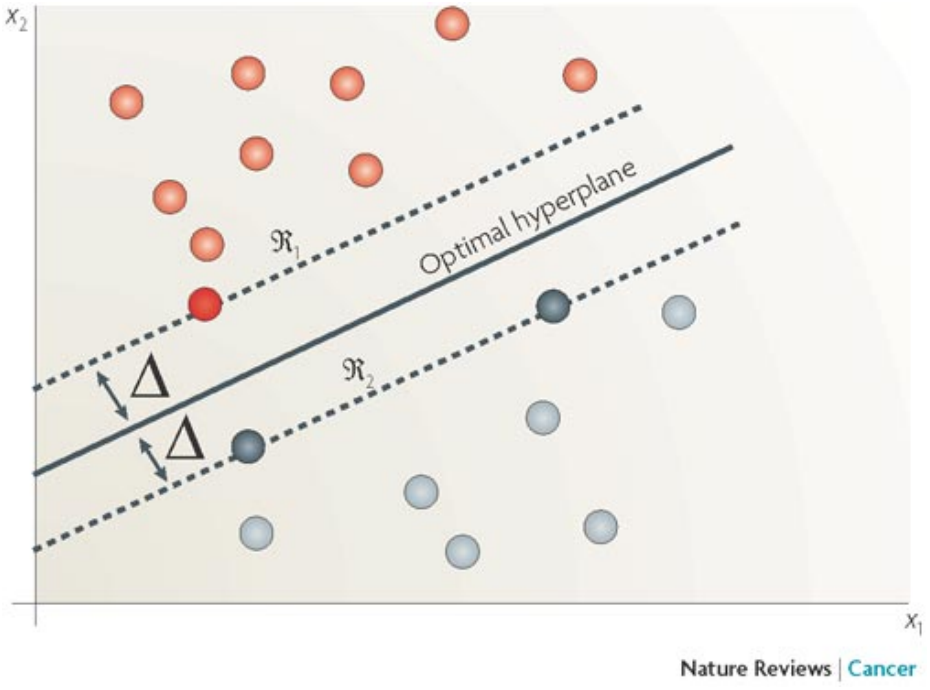

Figure 1.1: Optimal Separating Hyperplane

The signed distance d_i from the point x_i of the hyperplane described by the equation (1.2) is given by:

$$d_i = \frac{w \cdot x_i + b}{\sqrt{w \cdot w}} \tag{1.3}$$

from which we get, using the equation (1.1),

$$y_i d_i \geq \frac{1}{\|w\|_2}. \qquad i = 1, 2, \ldots, n, \tag{1.4}$$

The equation (1.4) means that $\frac{1}{\|w\|_2}$ is the minimum distance between the points x_i and the hyperplane identified by the pair (w, b).
The presence of the scalar 1 as the term on the right of the equation (1.1), besides allowing a constructive definition of "Optimal Separating Hyperplane", provides a one to one correspondence between separating hyperplanes and their parametric representation.
To fix this correspondence, it is necessary to introduce the concept of "canonical representation" of a separating hyperplane.

Definition 1.1.2 *Given a separating hyperplane, defined by the pair (w, b), for a set S linearly separable, its canonical representation is obtained by rescaling the pair (w, b) into the pair (w', b') so as that the distance of the nearest point on the hyperplane is equal to $\frac{1}{\|w'\|_2}$.*

Using this definition, we can get:

$$\min_{x_i \in S} \{y_i(w' \cdot x_i + b')\} = 1. \tag{1.5}$$

Consequently, considering for the separating hyperplane its canonical representation, there exists at least one point (x_i, y_i) for which the relationship (1.4) is valid only with the equality sign.
In the rest of the discussion we will assume that a separating hyperplane is

defined as usual by its canonical representation, and then we will use the pair (w, b) instead of (w', b').
At this point it is possible to specify the way in which we obtain the optimal separating hyperplane (also called OSH).

Definition 1.1.3 *Given a set S linearly separable, the optimal separating hyperplane is the one that maximizes the distance of the nearest point of S.*

Now, as according of (1.4) $\frac{1}{\|w\|_2}$ is the minimum distance of a point in the training set from the hyperplane, the OSH can be obtained as a solution to the problem that maximizes $\frac{1}{\|w\|_2}$ in the presence of the constraints (1.1), i.e.:

$$\min_{y_i(w\cdot x_i+b)\geq 1, i=1,2,\ldots,N.} \frac{1}{2} w \cdot w \tag{1.6}$$

If (w, b) is the pair that resolve the problem (1.6), there exists at least one point $x_i \in S$ such that $y_i(w \cdot x_i + b) = 1$.
Optimal separating hyperplane can therefore be considered as those hyperplane that maximizes the margin (Figure 1.1) between the two classes.
The problem (1.6) can be solved through the technique of Lagrange multipliers, defining with $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_N)^T$ the N non-negative Lagrange multipliers associated with the constraints of the equation (1.1).
This gives a new optimization problem, called "dual problem" of the problem (1.6), which is formulated in the following way:

$$\max_{\substack{\sum_{i=1}^{n} y_i\alpha_i=0 \\ \alpha\geq 0}} -\frac{1}{2}\alpha^T G\alpha + \sum_{i=1} \alpha_i \tag{1.7}$$

where G is a matrix $N \times N$ defined by the relation:

$$G_{ij} = y_i y_j x_i \cdot x_j. \tag{1.8}$$

The pair (w^*, b^*), that define the OSH, is determined in the following way:

$$w^* = \sum_{i=1}^{N} \alpha_i^* y_i x_i \tag{1.9}$$

while b^* is calculated using the Kuhn-Tucker condition:

$$\alpha_i^*(y_i(w \cdot x_i + b) - 1) = 0, \quad i = 1, 2, \ldots, N. \tag{1.10}$$

Equation (1.9) showes that the OSH is a linear combination (non-negative) of the vectors x_i that constitute the training set.

Particularly, only the vectors for which $\alpha_i > 0$ effectively contribute to the definition of the vector w^*.

Note, from (1.10), that the nonzero α_i^* are those for which the constraints (1.1) are satisfied with the equality sign (active constraints).

Since generally the majority of α_i^* it's equal to zero, the vector w^* results a linear combination of a small part of the vectors constituting the initial training set.

Those vectors, called "support vectors", are the points of S nearest to the OSH and are the ones required to determine the OSH.

Assigned a support vector x_j, the scalar b^* can be obtained from the corresponding Kuhn-Tucker condition (1.10):

$$b^* = y_j - w^* \cdot x_j \tag{1.11}$$

So the problem concerning the classification of a new event x is solved calculating

$$sign(w^* \cdot x + b^*). \tag{1.12}$$

It's possible to think about the support vectors as "vantage points" of the initial training set where it's condensed the information necessary to classify new events. [3, 4]

1.2 The GPDT

In this section we propose a recent decomposition technique for the large quadratic program arising in training Support Vector Machines named GPDT (Gradient Projection-based Decomposition Technique).

As standard decomposition approaches, the technique we consider is based on the idea to optimize, at each iteration, a subset of the variables through the solution of a quadratic programming subproblem.

The GPDT approach consist in using a very effective gradient projection method for the inner subproblems and a special rule for selecting the variables to be optimized at each step.

These features allow to obtain promising performance by decomposing the problem into few large subproblems instead of many small subproblems as usually done by other decomposition schemes, with the aim of reducing the computational cost of each iteration.

Consider the numerical solution of the convex quadratic programming (QP) problem arising in training *Support Vector Machines* (SVMs) [3, 4]:

$$\begin{aligned} \min \quad & \mathcal{F}(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T G \boldsymbol{x} - \sum_{i=1}^{n} x_i \\ \text{sub. to} \quad & \textstyle\sum_{i=1}^{n} y_i x_i = 0, \\ & 0 \le x_i \le C, \qquad i = 1, \ldots, n, \end{aligned} \tag{1.13}$$

where the size n is the number of labelled examples of the given training set

$$D = \{(\boldsymbol{z}_i, y_i),\ i = 1, \ldots, n, \quad \boldsymbol{z}_i \in \mathbb{R}^m,\ y_i \in \{-1, 1\}\},$$

and the entries of G are defined by

$$G_{ij} = y_i y_j K(\boldsymbol{z}_i, \boldsymbol{z}_j), \quad i, j = 1, 2, \ldots, n,$$

in which $K : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ is a special kernel function.

Examples of widely used kernel functions are:

- **Linear Kernel** $K(\boldsymbol{z}_i, \boldsymbol{z}_j) = \boldsymbol{z}_i^T \boldsymbol{z}_j$
- **Polynomial Kernel** $K(\boldsymbol{z}_i, \boldsymbol{z}_j) = (1 + \boldsymbol{z}_i^T \boldsymbol{z}_j)^d, d \in \mathbb{N}$
- **Gaussian Kernel** $K(\boldsymbol{z}_i, \boldsymbol{z}_j) = \exp(-\|\boldsymbol{z}_i - \boldsymbol{z}_j\|_2^2/(2\sigma^2)),\ \sigma \in \mathbb{R}$

The problem (1.13) is a genelarization of the problem (1.7) to the case of linearly nonseparable trainign sets and to nonlinear separating surfaces.
Since these kernel functions make the matrix G dense and in many real life applications the size of the training set is very large ($n \gg 10^4$), problem (1.13) cannot be generally solved by traditional approaches based on explicit storage of G.
The most popular strategies to overcome this drawback (the decomposition techniques) usually split the problem into a sequence of smaller QP subproblems that can be stored in the available memory and efficiently solved [5, 6]. At each decomposition step, a subset of the variables is optimized through the solution of the subproblem, in order to obtain a progress towards the minimum of $\mathcal{F}(\boldsymbol{x})$.
One of the main differences among the various decomposition approaches proposed in literature is given by the size chosen for the subproblems.
In fact, if the subproblems are sized 2, they can be solved analytically [7, 9, 10], while an inner numerical QP solver is necessary in decomposition schemes that use subproblems of larger size [11, 12, 13, 6, 14, 15].
Both the frameworks have been explored and have given rise to decomposition packages widely used within the SVM community.
The GPDT introduces some improvements to the decomposition technique recently proposed in [14].
This technique belongs to the class of decomposition schemes based on a numerical solution of the inner subproblems.

It uses as inner QP solver the special gradient projection method introduced in [16].

The gradient projection method used by GPDT exhibits high performance on SVM-type QP problems and enables the decomposition technique to manage efficiently much larger subproblems in comparison to standard decomposition packages.

This ability, combined with an appropriate selection of the variables to optimize at each step and a standard caching strategy for reducing kernel evaluations, allows GPDT to achieve promising performance with respect to the SVM^{light} algorithm, one of the most used and efficient decomposition packages.

Furthermore, the possibility to decompose into large subproblems makes the scheme well suited for an easy and effective parallelization, the numerical experiments carried out in [14] with a parallel version of GPDT give evidence of the importance of this approach in training SVMs on multiprocessor systems.

The improvements to GPDT concern the subproblems solution and a simple trick for a further reduction of the kernel evaluations.

For the subproblems solution the recent gradient projection method proposed in [17] will be considered, that can be more efficient than the inner solver previously used by GPDT.

The GPDT kernel evaluations are reduced by introducing an alternative formula for defining the data of the subproblem at each iteration.

Numerical evidence of these improvements are given by solving well known benchmark problems and by comparison with the SVM^{light} package.

1.3 A gradient projection-based decomposition technique

By following the classical notation, we split the indices of the variables $x_i,\ i = 1, \ldots, n$, into the set $\mathcal{B}$ of *basic* variables, called the *working set*, and the set $\mathcal{N} = \Big\{1, 2, \ldots, n\Big\} \setminus \mathcal{B}$ of *nonbasic* variables.

Furthermore, we denote by $\boldsymbol{x}^*$ a solution of (1.13) and arrange $\boldsymbol{x}$ and G with respect to $\mathcal{B}$ and $\mathcal{N}$ as follows:

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_{\mathcal{B}} \\ \boldsymbol{x}_{\mathcal{N}} \end{bmatrix}, \qquad G = \begin{bmatrix} G_{\mathcal{B}\mathcal{B}} & G_{\mathcal{B}\mathcal{N}} \\ G_{\mathcal{N}\mathcal{B}} & G_{\mathcal{N}\mathcal{N}} \end{bmatrix}.$$

The decomposition techniques for problem (1.13) are iterative procedures that, at each step, solve (1.13) with respect to the basic variables only.

This leads to a sequence of QP subproblems simpler than the original problem since their size n_{sp} is equal to the size of the working set ($n_{sp} = \#\mathcal{B}$), that can be chosen much smaller than n.

At each iteration, the subproblem solution is used to improve the current approximation of $\boldsymbol{x}^*$ and an appropriate updating of the working set is required in order to obtain the convergence.

The special decomposition technique given in [14] can be described as in in the following algorithm, called GPDT2:

- **1.0 Initialization**

  Let $\boldsymbol{x}^{(1)}$ be a feasible point for (1.13), let n_{sp} and n_c be two integer values such that $n \geq n_{sp} \geq n_c > 0$, n_c even;

  Let $\mathcal{L}$ be the largest even number such that $\mathcal{L} \leq \frac{n_{sp}}{10}$.

  Arbitrarily choose n_{sp} indices for the working set $\mathcal{B}$ and set $k \leftarrow 1$.

- **2.0 QP subproblem solution**

  Compute by a Gradient Projection Method the solution $\boldsymbol{x}_{\mathcal{B}}^{(k+1)}$ of

$$\begin{aligned} \min \quad & \frac{1}{2}\boldsymbol{x}_{\mathcal{B}}^T G_{\mathcal{B}\mathcal{B}}\boldsymbol{x}_{\mathcal{B}} + \left(G_{\mathcal{B}\mathcal{N}}\boldsymbol{x}_{\mathcal{N}}^{(k)} - \mathbf{1}_{\mathcal{B}}\right)^T \boldsymbol{x}_{\mathcal{B}} \\ \text{sub. to} \quad & \textstyle\sum_{i\in\mathcal{B}} y_i x_i = -\sum_{i\in\mathcal{N}} y_i x_i^{(k)}, \\ & 0 \le x_i \le C, \quad \forall i \in \mathcal{B}, \end{aligned} \tag{1.14}$$

  where $\mathbf{1}_{\mathcal{B}}$ is the n_{sp}–vector of all ones.
  Set $\boldsymbol{x}^{(k+1)} = \left(\boldsymbol{x}_{\mathcal{B}}^{(k+1)^T}, \ \boldsymbol{x}_{\mathcal{N}}^{(k)^T} \right)^T$.

- **3.0 Gradient updating**

  Update the gradient

$$\nabla\mathcal{F}(\boldsymbol{x}^{(k+1)}) = \nabla\mathcal{F}(\boldsymbol{x}^{(k)}) + \begin{bmatrix} G_{\mathcal{B}\mathcal{B}} \\ G_{\mathcal{N}\mathcal{B}} \end{bmatrix} \left(\boldsymbol{x}_{\mathcal{B}}^{(k+1)} - \boldsymbol{x}_{\mathcal{B}}^{(k)}\right) \tag{1.15}$$

  and terminate if $\boldsymbol{x}^{(k+1)}$ satisfies the KKT conditions.

- **4.0 Updating of $\mathcal{B}$**

  - **4.1**

    Find the indices corresponding to the nonzero components of the solution of

$$\begin{aligned} \min \quad & \nabla\mathcal{F}(\boldsymbol{x}^{(k+1)})^T \boldsymbol{d} \\ \text{sub. to} \quad & \boldsymbol{y}^T \boldsymbol{d} = 0, \\ & d_i \ge 0 \quad \text{for } i \text{ such that } x_i^{(k+1)} = 0, \\ & d_i \le 0 \quad \text{for } i \text{ such that } x_i^{(k+1)} = C, \\ & -1 \le d_i \le 1, \\ & \#\{d_i \mid d_i \ne 0\} \le n_c. \end{aligned} \tag{1.16}$$

    Let $\bar{\mathcal{B}}$ be the set of these indices.

– **4.2**

Fill $\bar{\mathcal{B}}$ up to n_{sp} entries by adding the most recent indices $j \in \mathcal{B}$ satisfying $0 < x_j^{(k+1)} < C$.

If these indices are not enough, then add the most recent indices $j \in \mathcal{B}$ such that $x_j^{(k+1)} = 0$ and, eventually, the most recent indices $j \in \mathcal{B}$ satisfying $x_j^{(k+1)} = C$.

– **4.3**

Set $n_c = \min\{n_c, \max\{10, \mathcal{L}, n_{new}\}\}$, where n_{new} is the largest even number such that $n_{new} \leq \#\{j,\ j \in \bar{\mathcal{B}},\ j \notin \mathcal{B}\}$.

Set $\mathcal{B} = \bar{\mathcal{B}}$, $k \leftarrow k + 1$ and go to step **2.0**.

As we can see the GPDT2 scheme is similar to the well known SVM^{light} algorithm but presents an essential difference in the choice of the inner QP solver.

In fact, GPDT2 uses a gradient projection method, the *Generalized Variable Projection Method* (GVPM) proposed in [16], that is much more effective than the inner solvers (the pr_LOQO or the Hildreth and D'Esopo method) on which SVM^{light} is based.

Thus GPDT2 is able to efficiently solve large subproblems that are hardly managed by SVM^{light} due to the inner optimizer overhead.

As a consequence, while SVM^{light} has been designed and very well optimized for decomposition processes in which n_{sp} is very small (generally less than 10^2), GPDT2 is appropriately developed for decomposing into large subproblems (generally $n_{sp} > 10^3$) in order to exploit the high performance of the inner solver.

The new GPDT2 approach introduces some implementative difficulties but opens interesting scenarios.

On one hand, since the entries of $G_{\mathcal{BB}}$ and $G_{\mathcal{NB}}$ are not in memory, large

values of n_{sp} increase the request of kernel evaluations per iteration.
This means that special strategies for computing the kernels (exploitation of sparseness in training examples) and for reducing their evaluations (caching of G entries and sparseness exploitation in the matrix-vector product (1.3)) become crucial tricks for the GPDT2 performance.
On the other hand, the ability to work with large n_{sp} allows to explore the benefits, in terms of convergence rate, arising from the possibility to optimize many variables at each decomposition iteration.
To this end, an effective rule for updating the working set $\mathcal{B}$ is the essential key.
Unfortunately, for the case of large sized working sets, this topic is not widely investigated in literature since the most popular decomposition approaches are commonly used with very small n_{sp} or they are designed for $n_{sp} = 2$ only, in order to solve the subproblems analytically.
The working set selection described in step **4** of Algorithm GPDT2 has been recently introduced in [14]; it follows the classical SVM^{light} selection rule but introduces new devices useful in case of large working sets.
By proceeding as in SVM^{light}, step **4.1** finds at most $n_c \leq n_{sp}$ indices for the new working set by solving the linear problem (1.16); the aim is to define basic variables that make possible a rapid decrease of the objective function in the new iteration.
This selection idea is exploited by many decomposition approaches and is at the basis of the main theoretical studies on the convergence properties of decomposition techniques [8, 19, 20, 21, 22].
Since the parameter n_c is usually recommended to be less than n_{sp} in order to reduce zigzagging phenomena, $(n_{sp} - n_c)$ indices are required to fill up $\mathcal{B}$.
The filling criterion used in step **4.2** is similar to that introduced in [15]

but it takes into account also how long a variable is in the working set (see also [19] for a discussion about the importance to retain free variables of the previous $\mathcal{B}$).

The last step **4.3** introduces an adaptive reduction of the parameter n_c.

For large n_{sp}, this trick allows the decomposition procedure to start with large n_c (e.g. $n_c = n_{sp}/2$), so many new variables can be optimized in the first iterations, but avoids zigzagging through the progressive reduction of n_c.

More details on the above working set selection can be found in [14] where its effectiveness is evaluated by an extensive computational study on well known benchmark problems.

In [14] a comparison between GPDT2 and SVM^{light} is also performed. The two solvers exhibit opposite behaviour: GPDT2 gets its best performance for large n_{sp} and shows low sensitivity to n_{sp} variations while SVM^{light} is competitive only for very small n_{sp}; if the best performances are compared, GPDT2 is preferable.

Finally, one of the most important aspects of GPDT2 needs to be recalled: its easy parallelization. By decomposing into large subproblems, GPDT2 generally requires very few decomposition iterations in which the most expensive tasks are the solution of the subproblem (1.14) and the kernel evaluations required in step **3**.

These tasks can be easily performed on a multiprocessor system by using a parallel version of the GVPM and by distributing the kernel evaluations among the available processors.

A parallel version of GPDT2, derived by following these ideas and the implementative framework introduced in [15], has been tested in [14] showing promising speedups respect to the serial GPDT2 and the SVM^{light}.

The only thing that remains to be improved both in the serial and in the parallel version are the the computation of the kernel, process that tends to slow down the entire system.

In this thesis we have implemented the computation of the kernels in the GPGPU environments, a new area of computer science which will be presented in the next chapter.

Chapter 2

GPGPU and CUDA

Today the majority of people have a decent graphic processor on his own computer.
Even a recent low-end video card exceeds computational power for the common usage like surfing the web, document and image editing or other applications.
Is therefore questionable if all this power may be exploited in other ways, in different fields from 3D rendering or videogames.
From this need born the GPGPU computing, that stands for General-Purpose computing on Graphic Processing Units [34, 35, 33].

2.1 GPGPU

The GPGPU is an area of computer science that aims to use the GPU for different purposes than the traditional process of creating a three-dimensional image.
In this scope the GPU is used to do extremely complex processing, where traditional CPU architectures do not have enough processing capacity, or

they can not finish the calculations in reasonable time.
The GPGPU uses a GPU (Graphics Processing Units) alongside one CPU for accelerating the elaboration of the scientific and technical applications, offering performance comparable to that of a small cluster of computers, at a fraction of its cost, moving the most challenging portions of the calculations (the ones those running in parallel) of each application to the GPU, while the remaining sections of the code (the serial parts) runs on the CPU.

2.2 Evolution of the GPGPU

The following graph illustrates the reason why today is convenient to "move" portions of the code on the GPU.
Although the displacement of code and data requires a not-negligible amount of time, in case of computation-bound applications, that time is largely recovered during the computation phase, having an average performance increase from ten to twenty times higher than the serial version.
As we can see, the computational power of these architectures is growing significantly, when compared with normal CPU, because they were create with a specific orientation, the processing of 3D graphics, where hundreds of thousands of operations are performed in parallel without the need of having to occupy the little memory of the chip to manage the control logic.
The majority of the operations, in fact, is extremely simple and repetitive, and the conditions under which the parts of code are executed in branch, leaving some cores in idle, are very rare.
Broadly speaking, we can say that in the standard CPU, with a number of cores that nowadays can generally be sixteen, most of the chip area is spent to manage the control, while in the case of the GPU, cores are much simpler,

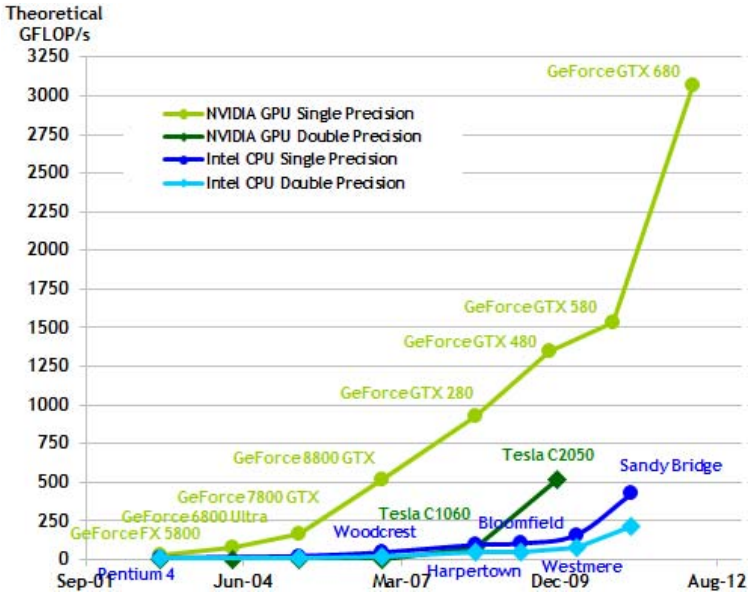

Figure 2.1: Evolution of the GPU

since only the floating-point unit are very fast, so the entire area can be used for them.

For that reason GPUs come to have thousands of cores that run all the same instruction, making them the perfect device for the resolution of problems that require computation over matrices and / or vectors.
GPUs have evolved to the point that several real-world applications can be successfully implemented on them and run so much faster than what happens on the traditional multi-core systems.

2.3 CUDA

NVIDIA has created a specific framework, called CUDA (Compute Unified Device Architecture), to program their own GPUs.
CUDA [23] is both an architectural model, an Application Programming Interface (API) to take advantage of such an architecture and a set of extensions

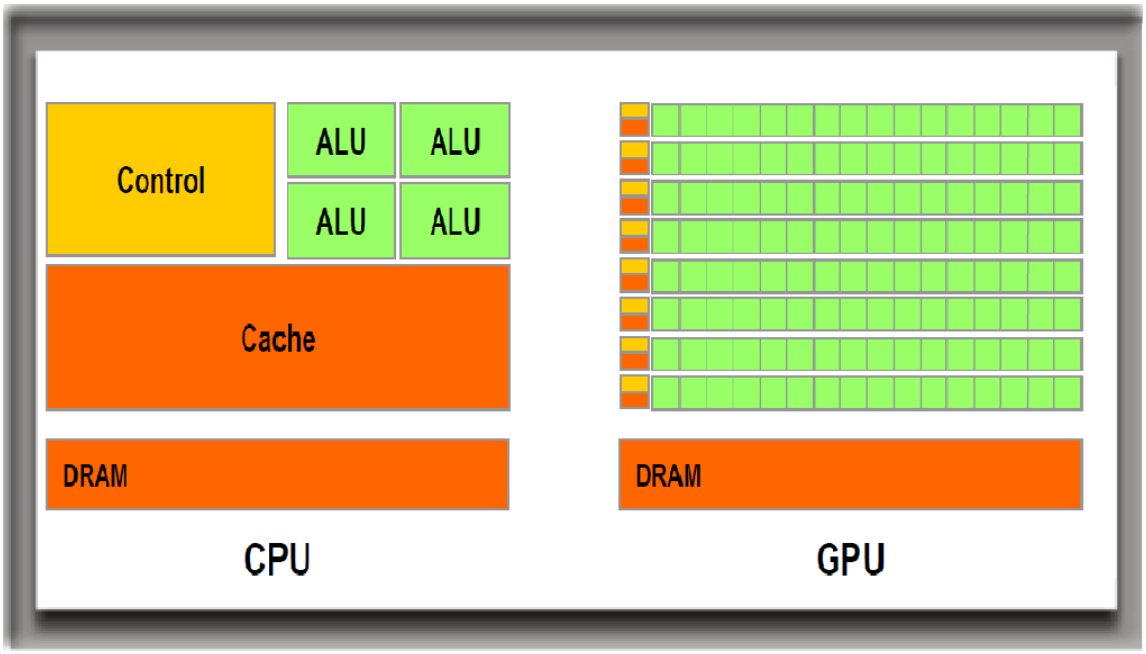

Figure 2.2: Example number of cores

(C for CUDA) to C and C++ language for describing parallel application able to run on the GPU that adopt that model.

Specifically, any NVIDIA GPUs starting from the GeForce 8 generation (G80) can now run applications written in CUDA.

The same goes of course for the generation of graphics cards with chip equivalent to the product families Quadro and Tesla.

2.4 CUDA hardware

In general a "CUDA enabled" graphics card consists of several multiprocessors, called Streaming Multiprocessor (SM), that are independent and their number varies depending on which market segment is intended that and from implementation choices decided by NVIDIA.

Each SM itself is composed of a variable number of Stream Processor (SP); this number depends on the generation of the card itself (G80, GT200, Fermi, Kepler).
Each of these processors can perform basic mathematical operation (addition, multiplication, subtraction, etc..) on integers or floating point numbers in single precision (32-bit) or double precision (64 bits).
The operations in double precision are officially available only from the third generation of GPUs, the Fermi, but in fact these operations are also available on the second-generation GT200 cards: they are not executed atomically but through the usage of two single precision cores together.
In each SM there is also a shared memory, accessible by all SP, caches for instructions and data and a unit that decodes the instructions.
At the software level, the program also splits into three levels nested between them: grid, block and thread.
Each function executable on the GPU is called "kernel" and is defined as a grid that can be divided into mutually independent blocks.
Each block is assigned by the scheduler to a SM, ensuring a first level of big grain parallelism.

For each block, there are many of the fundamental unit of computation, the cores, on which the threads run at a very fine granularity of parallelism. A thread can belong to only one block, and is identified by a unique index for the whole kernel.
For convenience, there is the possibility to use two-dimensional indices for blocks and three-dimensional indices for threads.
The kernels are executed sequentially while blocks and threads are instead logically executed in parallel.

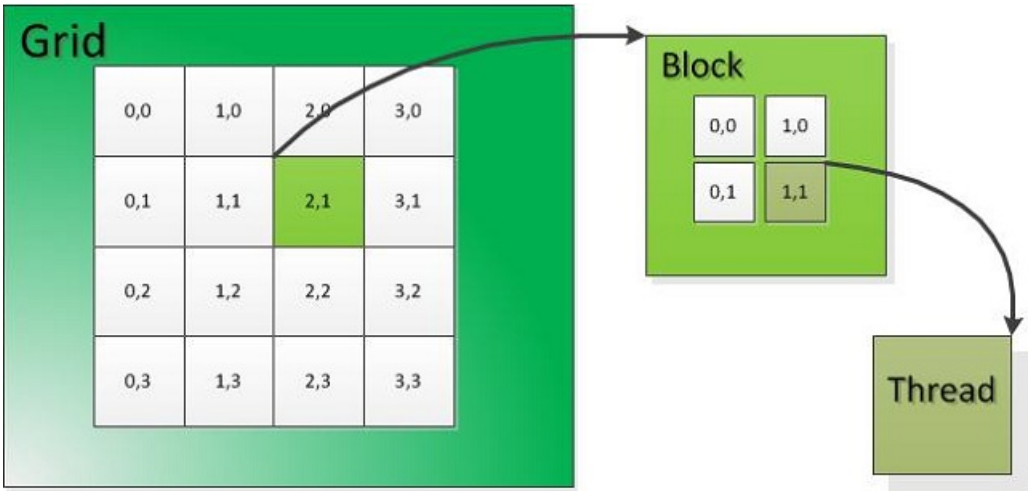

Figure 2.3: Division into grid, blocks and threads

Each thread can access different types of memory:

- **Global memory**

  Is the largest amount of memory on the video card and is accessible by all threads.

  This memory provides a large amount of bandwidth, but also suffers from a high latency.

  Supports the generic instruction of "load and store" and pointing to memory.

- **Local memory**

  Small amount of memory accessible only by the Stream Processor, has a latency like the global memory.

- **Shared memory**

  It's a memory shared between all the stream processors of a multiprocessor (SM), and visible from all threads in the same kernel.

  It's a memory much faster than the global memory, but much smaller

in terms of quantity.

It's useful to perform small operations management between threads and reduce, where possible, the accesses to the global memory.

- **Constant storage**

  High-latency memory, it's shared in read-only mode among all the multiprocessors.

- **Texture memory**

  High-latency memory used only to make quick the operations of linear interpolation.

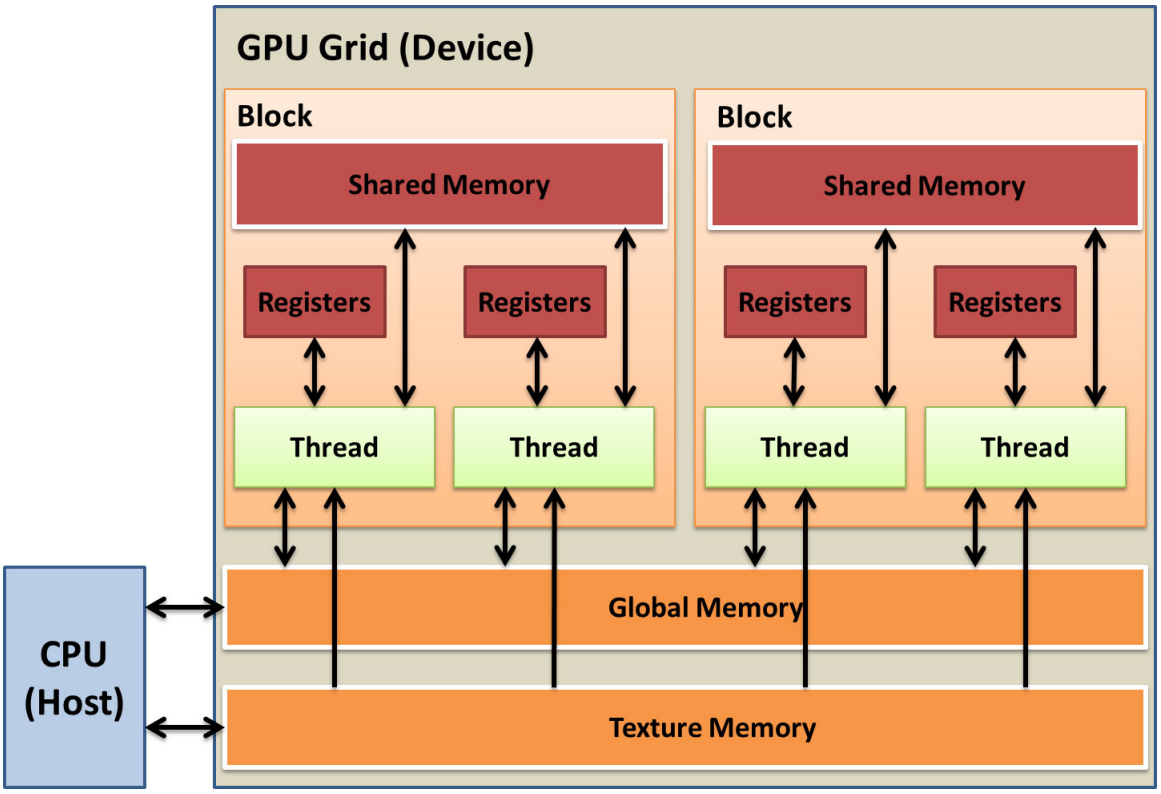

Figure 2.4: CUDA memory model

We can guess, therefore, that the memory type global, local, constant and texture, are physically the same memory, also known as the RAM memory which is equipped with a video card.
They only differ for the different access methods and the different techniques for memory caching.
The CUDA architecture just described, as mentioned previously, is the basis of all the "cuda enabled" NVIDIA GPUs .
There are several generations of GPU architecture, which differ mainly in power and computational capabilities, as well as for numerous innovations in image processing to enhance the game experience, trying to make it, generation after generation, more and more realistic.
These aspects, although very important, will not be discussed in this thesis.

2.5 CUDA generations

Today there are a total of four generations of CUDA GPUs [29] created to support the unified shader architecture.

2.5.1 First generation

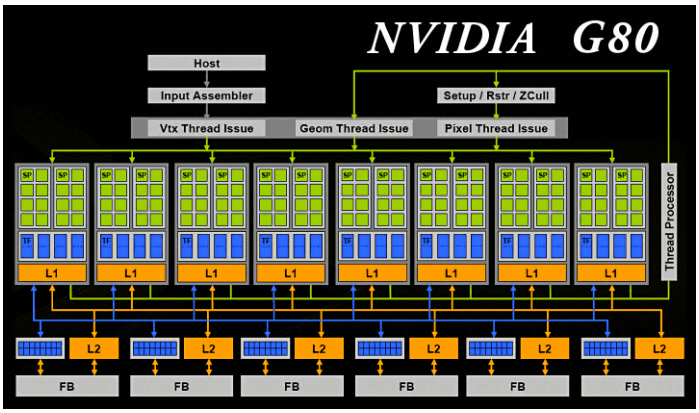

Figure 2.5: G80 Architecture

The first generation is represented by the GeForce 8 series, with the codename G80.

The top of the G80 series is represented by the GeForce 8800 GTX, which has a total of 128 CUDA cores (or stream processors) divided into sixteen Streaming Multiprocessor, each of which contains eight SP.

Each SM operates at a clock of 1500 MHz, while the rest of the chip only operates at 612 MHz, the total amount of RAM is 768 MB and runs at 2160 MHz on a PCI-Express 1.0, giving this card a throughput of 576 GFLOPs for MAD operation (multiplication and an addition in single precision).

A few months later was presented the NVIDIA 9 series, but is not to be considered a "new generation", because it's only a review of the previous series.

2.5.2 Second Generation

The second generation is represented by the GeForce 200 series, codenamed GT200.

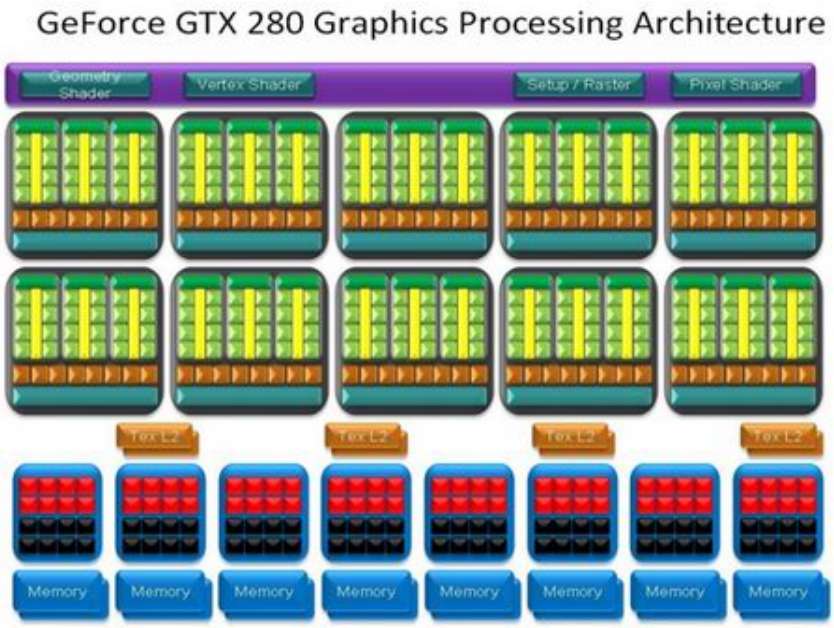

Figure 2.6: GT200 Architecture

The top of the second generation GPU is the GeForce GTX 285, with a total of 240 CUDA cores operating at 1476 MHz and divided into 30 Streaming Multiprocessor.

The rest of the chip operates at a clock of 648 MHz and the 1 GB of RAM which is provided operates at 2484 MHz

Compared to the first generation, the GT200 communicate to the CPU by the PCI-Express 2.0, reaching a throughput of 1062.72 GFLOPs for the MADD + MUL operations (multiplication and an addition + an addition in a single clock).

Furthermore, the GT200 introduce the possibility to operate on double precision data, but these operations require multiple rounds of clock and are not approximated correctly according to the IEEE–754 standard.

2.5.3 Third Generation

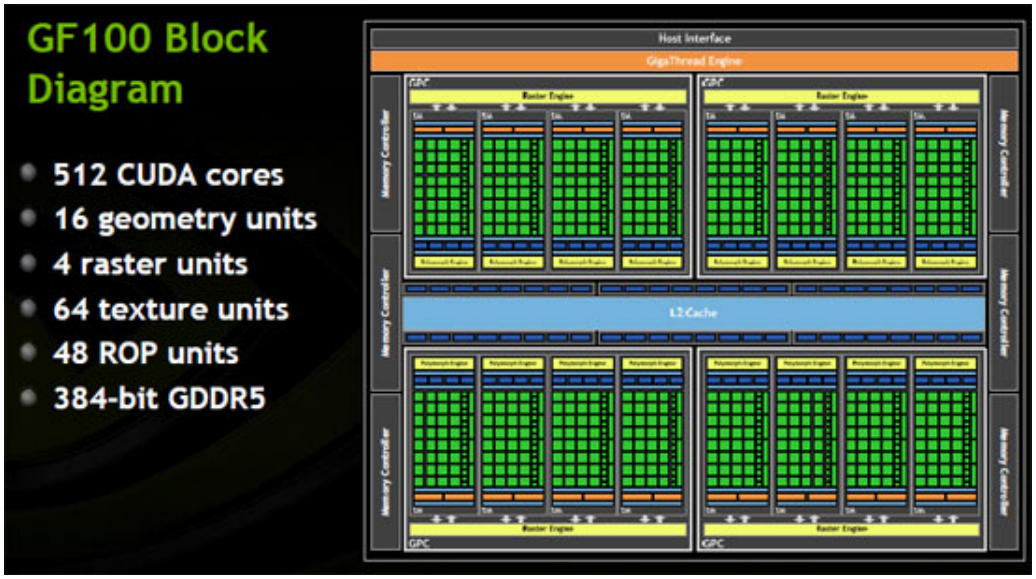

Figure 2.7: Fermi Architecture

The third generation unified shader GPU have the codenamed “Fermi” [31] and represents the GeForce 400 series , but later it was revisited in the GeForce 500 series.
The new key feature introduced in the “Fermi” series are:

- Ability to compute double-precision data in a single round of clock according to the IEEE–754 standard.
- Introduction of two levels of cache for each Streaming Multiprocessor.
- Ability to perform kernel concurrently (up to 16).
- Support for ECC memory (error-correcting code).

The GeForce GTX 480, the top of the generation with codename GF100, has 512 CUDA cores divided into 16 SM, each composed of 32 SP operating at

1401 MHz, against the 700 MHz of the chip.

It is also fitted with 1536 MB of RAM with clock of 3696 MHz for a throughput of 1344.96 GFLOPs for FMA operations (Fused Multiply-Add, which represents the ability to perform a MAD in double precision in a single clock).

2.5.4 Fourth generation

Figure 2.8: Kepler Architecture

The fourth generation, the current one, codenamed “Kepler” [30], is constituted by the GeForce 600 series, but it has been revisited in the new 700 series.

Kepler GPUs introduce important changes at the architectural level and lots for the gaming, but as mentioned earlier, this thesis going to deal with the important news only at the level of pure computation of the data:

- **SMX, Streaming Multiprocessor eXtreme**

  The streaming multiprocessor has been redesigned for the benefit of higher performance and greater energy efficiency.

  It offers three times the performance for watt for the streaming multiprocessor of Fermi.

  Energy efficiency has been achieved by increasing the number of CUDA cores four times, reducing the clock speed of each core, limiting the use of parts of the GPU when they are in idle mode and maximizing the area allocated to the GPU parallel-processing cores.

- **Dynamic Parallelism**

  This feature allows the GPU to adapt dynamically to the data.

  Significantly simplifies parallel programming, enabling GPU-based acceleration to take advantage of a more extensive set of well-known algorithms, such as the mesh refinement, fast multipole and multigrid methods.

  Moreover, thanks to this feature every kernel can invoke a new kernel without necessarily passing control to the CPU.

- **Hyper-Q**

  Reduces the idle time of the CPU by allowing the different CPU cores to use simultaneously a single Kepler GPU, improving programmability and efficiency, for these reasons Hyper-Q is ideal for applications that use MPI cluster.

The top of the generation, the GTX 680, also called GK104, has a total of

1536 CUDA cores, operating at 1.0 GHz, 2048 MB of RAM at 6.0 GHz that provides a card throughput of 3090 GFLOPs for FMA operation.
The above data always refer to the card top for each generation, references have not been made to series revisited as the series 9 or 500 series, to provide greater objectivity in the comparison of the features.

2.5.5 CUDA comparison

In the table below we can see a clear comparison between the architectures previously mentioned:

GPU	G80	GT200	GF100(Fermi)	GK104(Kepler)
Transistor	681 Million	1.4 Billion	3.0 Billion	3.5 Billion
Cuda core (SP)	128	240	512	1536
Graphics Core Clock	612 MHz	648 MHz	700 MHz	1006 MHz
Shader Core Clock	1500 MHz	1476 MHz	1401 MHz	1006 MHz
Memory	768 MB	1024 MB	1536 MB	4096 MB
Memory Clock	2160 MHz	2484 MHz	3696 MHz	6008 MHz
Memory Bandwidth	103.7 GB/s	159.0 GB/s	177.4 GB/s	192.256 GB/s
PCIe x16	1.0	2.0	2.0	3.0
Compute Capability	1.0	1.3	2.0	3.0
GFLOPs	576	1062.72	1344.96	3090
TDP	171 Watt	204 Watt	250 Watt	195 Watt

As can be seen from the data described and summarized in the table, the second generation can be understood as an improvement of the first, while the third and fourth generation have led, in addition to a pure performance increase purely computational, also at numerous technological innovations at the level of gaming and image processing.

2.6 Libraries

CUDA, as it was mentioned previously, is not only an architecture, but also provides a programming environment for C / C + + languages.
More precisely CUDA extends the C language, allowing the programmer to define some special functions.
This extension is also available in Fortran, Python and. NET.
In the C language "extended" (or C for CUDA), it is called device by referring to the GPU and host referring to the CPU.
The device can be accessed directly only to the built-in memory on the GPU but can't access the memory on the motherboard, conversely for the host.
This choice was a position taken from NVIDIA, which has deliberately kept to separate the accesses to the two types of memory.
In the CUDA C–like language we have new type of qualifiers for functions, which can be declared by specifying a prefix which can be:

- **___device___**

  The code written inside these functions is performed only on the device and can be called only from the device.

- **___global___**

  The code written inside these functions is performed only on the device and is callable from the host (in generation Kepler type this function can also be called by the device).

- **___host___**

  The code written inside these functions is performed only on the host and can be called only from the host.

Functions declared as ___host___ represent the typical C functions, but they can be combined with functions ___device___ to declare a function that runs

on both the host and from the device.
Functions declared ___global___ typically represent the kernel, and are run by specifying N parallel copies of each executable with M cuda–cores (threads).
CUDA also includes the nvcc compiler for NVIDIA GPUs, math libraries and tools for debugging and optimization of the sources.
Among the major libraries we can see:

- **Thrust**[27]
  A comprehensive library that provides data structures and algorithms to optimize operations reductions, sorting, scan, transform, random number generation, etc, and also facilitates the management of memory on the host (CPU) and the device (GPU)

- **CUBLAS (Basic Linear Algebra Subroutines)**[25]
  Is a full version of the BLAS libraries, specialized for CUDA, while ensuring acceleration from six to seventeen times higher than the common BLAS.

- **CUSPARSE (Sparse Matrix library)**[28]
  Is a collection of functions of linear algebra devoted to the computation of sparse matrices, accelerating up to eight times faster than the serial ones.

- **NPP (NVIDIA Performance Primitives)**[26]
  libraries that provide functions dedicated to the accelerated computation of images, audio and video signals, ensuring acceleration from five to ten times higher than the CPU–only implementation.

- **CUFFT (Fast Fourier Transform)**[24]

Is a full version and accelerated of the FFT libraries, providing a speed-up of up to ten times higher than serial ones.

Chapter 3

Experimentation

In this chapter we show the numerical results and the speedups related to the implementation of the computations of the various kind of kernel on the GPU, also showing the several implementation choices that we made.

3.1 Development environment

3.1.1 Hardware

The programs developed were tested on different architectures:

- INTEL(R) core i7(R) 860S at 2,53 GHz with 4GB of RAM with NVIDIA GeForce GTX230m with 1GB of dedicated RAM.
- AMD Phenom(tm) II X6 1090T at 3.2GHz with 6GB of RAM with NVIDIA GeForce GTX480 with 1536MB of dedicated RAM.

Both hardware architectures are not very recent, but we wanted to show that even using such platforms may lead to significant improvements in performance.

3.1.2 Software implementation

The tests were conducted on two different codes:

- **simil_gpdt_no_cuda**
  program that simulates the behaviour of GPDT for the calculation of the kernels in serial version.

- **simil_gpdt_si_cuda**
  program that simulates the behaviour of GPDT for the calculation of the kernels using the GPU.

The serial codes are written in C++ and the sources are compiled with gcc version 4.6.3 on operating systems based on Debian-Linux, in particular Kubuntu 12.04.3 64 bit.
The GPGPU codes have been implemented using the CUDA Toolkit 5.0 for the GT200 generation cards (NVIDIA GeForce GTX230m) and Fermi card (NVIDIA GeForce GTX480).
The GPGPU sources are compiled with nvcc V5.5.0 and were written with the target to execute all of the parallelized code on the GPU, in such a way to minimize the latency due to the transfer data between host and device.
More precisely, the data transfer between host and device initialization is done only before the computation of the kernel, then the entire run of the calculation is considered to be totally on the device.
This type of choice must be designed accurately and specifically for each algorithm and architecture.
For this kind of problems, which are purely computational, it's not a bad choice, since there are no data exchange with external processes or user interaction.
During the implementation we faced nevertheless some obstacles:

- reduced on–board memory available for the the consumer version of GPU, which requires to perform preliminary checks before starting the computation process.

  These checks are carried out with this procedure:

```
int col_size = Nr_vet * sizeof(float);
size_t free_byte;
size_t total_byte;
cudaMemGetInfo( &free_byte, &total_byte );
int col_host_mem = (free_byte*0.8)/col_size;
if (col_host_mem > Nr_col_req)
{
  col_host_mem = Nr_col_req;
}
int n_cycles = Nr_col_req/col_host_mem;
int rest = Nr_col_req - (n_cycles * col_host_mem);
```

  In these steps we control how many columns of the results can be stored in the device free memory, and after checking if all the required columns can be stored in the free memory, we compute the required number of cycles to complete the calculation, also checking the number of columns that will be excluded at the end of the cycles. These columns will be calculated at the end of the cycles.

- CUDA systems for the consumer GPU have an internal check that stops the execution of a kernel function on the device if it takes longer than 5-15 seconds to compute (the exact amount depends by the family and the type of the GPU used).

  Unfortunately, this check can't be disabled directly from the program,

but it requires a preliminary procedure for disabling it and it cannot be done for all kind of GPU either.
To solve this issue directly from the program we decided to put a threshold to ensure that the computation can be executed within the time limit for each kernel:

```
/**
 * This value is based on using an Nvidia GTX230m
 **/
int col_time_host = 200;
if (col_time_host > col_host_mem)
{
  col_time_host = col_host_mem;
}
```

- It's very important to split the thread and the blocks correctly, creating enough of them to be sure that every kernel will be computed, otherwise some kernels will be not computed, or they will be stored in the wrong positions.
  Also we have to be sure that the number of threads requested to each block is not greater than the maximum number specified for each GPU, otherwise the program does not compute the results.
  We have used this procedure to split the blocks and the threds:

```
 /**
 * This value is based on using an Nvidia GTX230m, max
    threads 512
 **/
int dimXX =4;
```

```
int dimYY =128;

int numSMs;
cudaDeviceGetAttribute(&numSMs,
    cudaDevAttrMultiProcessorCount, 0);
dim3 blockGridRows;
blockGridRows.x=Nr_vet/dimXX + (Nr_vet%dimXX== 0?0:1);;
blockGridRows.y=col_host_mem/dimYY + (col_host_mem%dimYY==
    0?0:1);
dim3 threadBlockRows;
threadBlockRows.x=dimXX;
threadBlockRows.y=dimYY;
```

dim3 is an integer type vector defined by the CUDA framework that can be used in CUDA codes.
Its most common application is to pass the grid and block dimensions in a kernel invocation.
It can also be used in any user code for holding values of 3 dimensions.
At last, each thread index on the device will be assigned through this procedure:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

The data of the dataset is entirely transferred in the GPU's memory alongside two integer vectors, one containing the number of significant values for each vector and the other containing the indices of the significant data, therefore we consider only the effective non-zero data during the operations, reducing the number of them.

This may seem wasteful in case of datasets with hight sparsity level, especially considering the low amount of memory contained in the GPUs today, but in this way we can remove any operation data management from the GPU, making the whole calculation extremely faster.

3.2 Testing

The tests of the serial and GPU codes were performed on two different data–sets:

- **Web data set**

  The data–set [10], (available at `http://http://www.research.microsoft.com/~jplatt`), concerns web page classification problem with binary representation based on 300 keyword features.

  On average, the sparsity level of the examples is about 96%.

- **MNIST data set**

  The MNIST database of handwritten digits [32] contains 784-dimensional non-binary sparse vectors.

  On average, the sparsity level of the inputs is about 81%.

Each test was performed by calculating 10/100/500/1000 random columns of the kernel matrix.
On both datasets we used $d = 4$ for the Polynomial Kernel and $\sigma = 1800$ for the Gaussian Kernel.

MNIST dataset:

Table 3.1: MNIST 2K Linear

Linear Kernel	MNIST 2K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
	serial (s)	CUDA (s)	speedup
10	0,05	0,0125403	3,9871454431
100	0,49	0,067834	7,2235162308
500	2,43	0,337929	7,1908596184
1000	4,91	0,675122	7,2727595901
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
	serial (s)	CUDA (s)	speedup
10	0,01	0,00193155	5,1771893039
100	0,04	0,0111604	3,5841009283
500	0,32	0,0537019	5,9588208238
1000	0,65	0,108525	5,9894033633

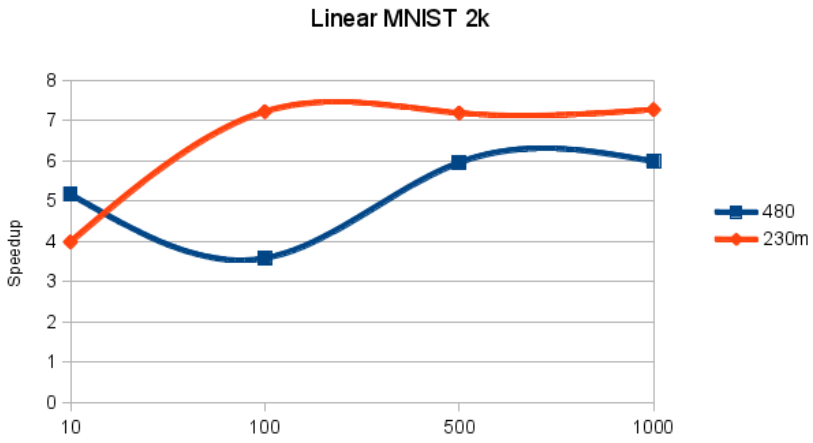

Figure 3.1: Linear MNIST 2K

Table 3.2: MNIST 2K Polynomial

Polynomial Kernel	MNIST 2K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,05	0,01299907	3,8464290138
100	0,51	0,0678525	7,5163037471
500	2,45	0,338379	7,2404020344
1000	4,94	0,675102	7,3174127761
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,01	0,00194298	5,1467333683
100	0,04	0,0112507	3,5553343348
500	0,34	0,0547188	6,2135865553
1000	0,68	0,108591	6,2620290816

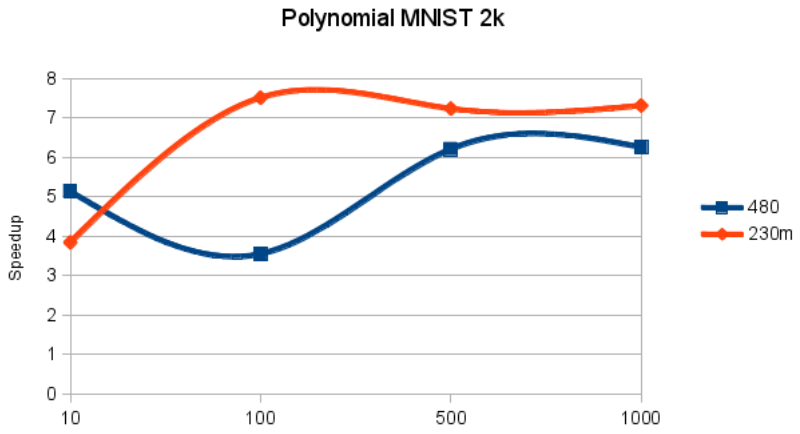

Figure 3.2: Polynomial MNIST 2K

Table 3.3: MNIST 2K Gaussian

Gaussian Kernel	MNIST 2K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,06	0,0236522	2,536761908
100	0,49	0,067834	7,2235162308
500	2,43	0,337929	7,1908596184
1000	4,91	0,675122	7,2727595901
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,01	0,00229184	4,363306339
100	0,05	0,011466	4,3607186464
500	0,36	0,055062	6,5380843413
1000	0,73	0,109012	6,6965104759

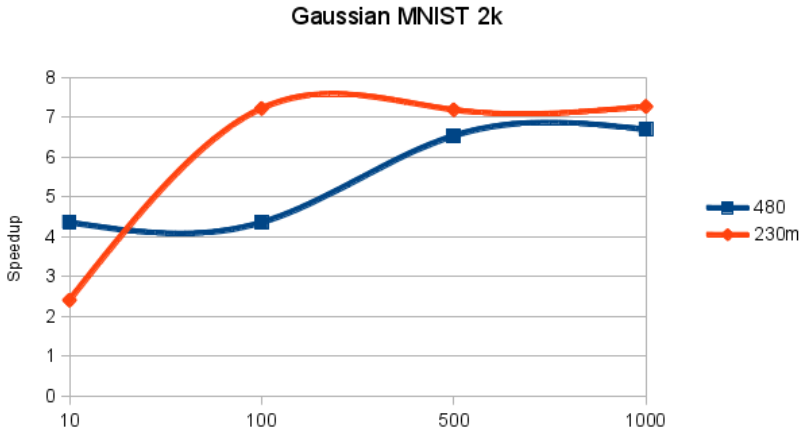

Figure 3.3: Gaussian MNIST 2K

Table 3.4: MNIST 10K Linear

Linear Kernel	MNIST 10K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
	serial (s)	CUDA (s)	speedup
10	0,26	0,0617787	4,2085702677
100	2,62	0,363645	7,2048288853
500	13,12	1,81357	7,2343499286
1000	26,37	3,62682	7,2708322994
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
	serial (s)	CUDA (s)	speedup
10	0,02	0,0096352	2,0757223514
100	0,22	0,0536781	4,0985057221
500	1,71	0,265559	6,4392470223
1000	3,47	0,528442	6,5664727633

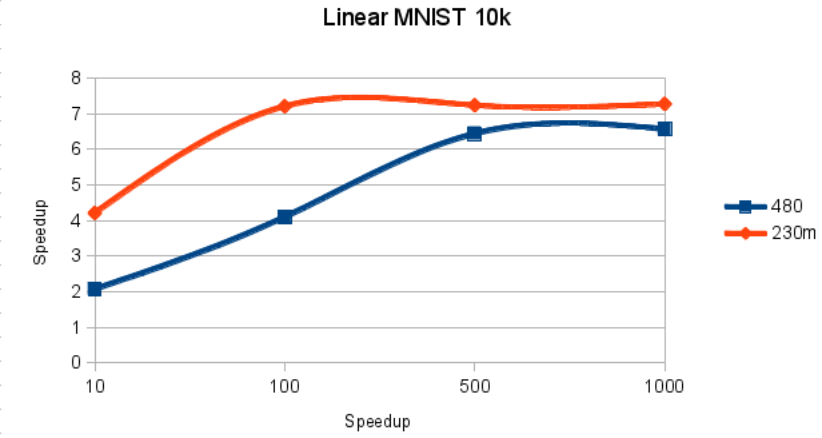

Figure 3.4: Linear MNIST 10K

Table 3.5: MNIST 10K Polynomial

Polynomial Kernel	MNIST 10K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,29	0,061855	4,6883841242
100	2,63	0,363564	7,233939554
500	13,23	1,81392	7,2935961895
1000	26,66	3,62696	7,3505084148
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,03	0,00961008	3,1217221917
100	0,24	0,0537362	4,4662629661
500	1,79	0,265779	6,7349188612
1000	3,65	0,265779	13,7332144376

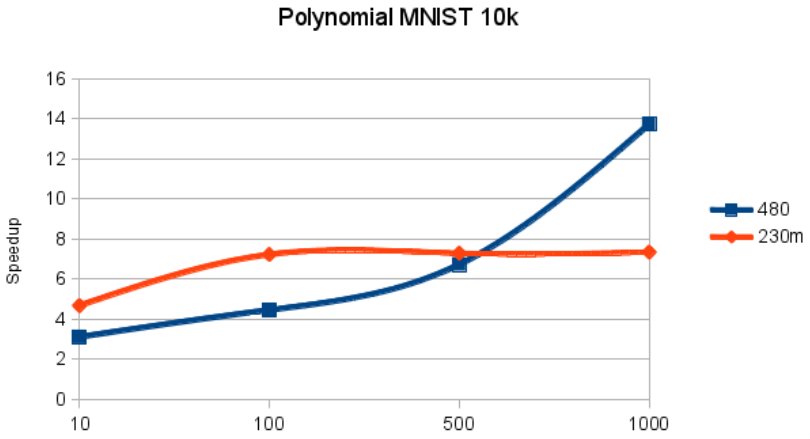

Figure 3.5: Polynomial MNIST 10K

Table 3.6: MNIST 10K Gaussian

Gaussian Kernel	MNIST 10K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,3	0,120545	2,488697167
100	2,87	0,379421	7,5641569655
500	13,53	1,87307	7,2234353228
1000	27,12	3,7453	7,2410754813
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,04	0,01121114	3,5678798053
100	0,27	0,0554252	4,8714303241
500	1,92	0,267367	7,1811405297
1000	3,91	0,530568	7,3694606535

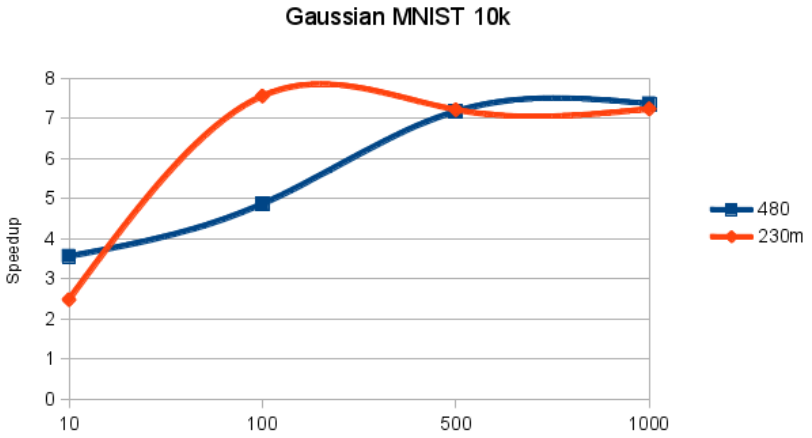

Figure 3.6: Gaussian MNIST 10K

Table 3.7: MNIST 60K Linear

Linear Kernel	MNIST 60K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
	serial (s)	CUDA (s)	speedup
10	1,35	0,354417	3,8090723639
100	14,57	2,0504	7,1059305501
500	73,58	10,2449	7,1821101231
1000	149,38	20,4813	7,2934823473
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
	serial (s)	CUDA (s)	speedup
10	0,14	0,0547692	2,5561812113
100	1,25	0,318244	3,9278038235
500	9,98	1,57576	6,3334517947
1000	20,38	3,14198	6,4863557375

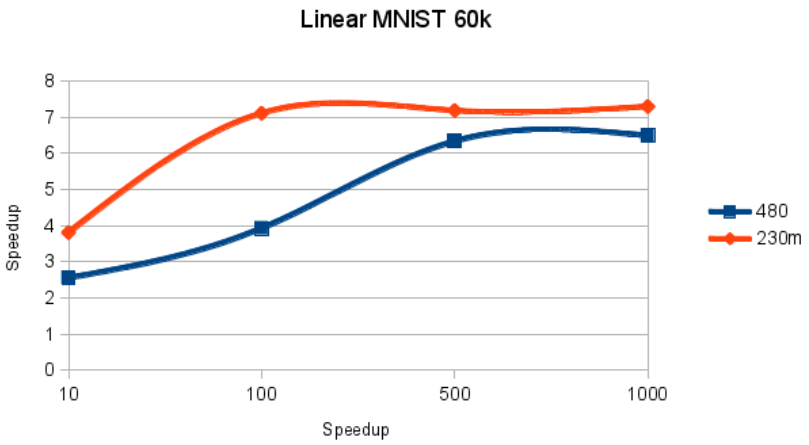

Figure 3.7: Linear MNIST 60K

Table 3.8: MNIST 60K Polynomial

Polynomial Kernel	MNIST 60K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$d = 4$	serial (s)	CUDA (s)	speedup
10	1,47	0,3550704	4,1400240628
100	14,79	2,0501	7,2142822301
500	78,96	10,2439	7,7080018352
1000	156,07	20,4817	7,6199729515
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,15	0,0547801	2,7382206312
100	1,32	0,318311	4,1468877921
500	10,41	1,57657	6,6029418294
1000	21,27	3,14189	6,7698105281

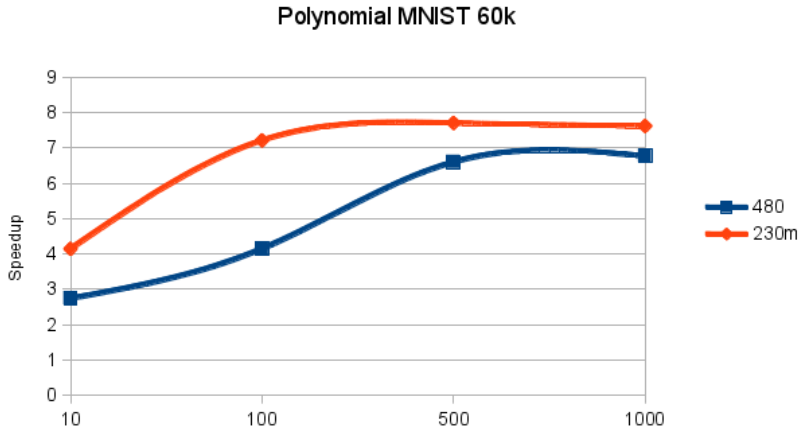

Figure 3.8: Polynomial MNIST 60K

Table 3.9: MNIST 60K Gaussian

Gaussian Kernel	MNIST 60K		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	1,67	0,693274	2,408859989
100	15,2	2,0716	7,3373238077
500	75,63	10,5674	7,1569165547
1000	153,24	19,011	8,0605964968
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,2	0,0646396	3,0940785525
100	1,57	0,328247	4,7829835459
500	11,24	1,58812	7,0775508148
1000	22,86	3,1525	7,2513877875

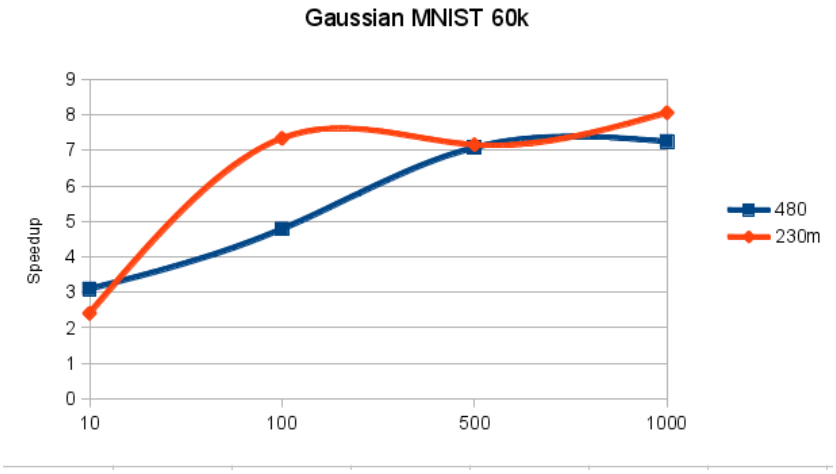

Figure 3.9: Gaussian MNIST 60K

WEB data set:

Table 3.10: Web W2a Linear

Linear Kernel	Web W2a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
	serial (s)	CUDA (s)	speedup
10	0,01	0,00329552	3,0344224887
100	0,08	0,0107934	7,4119369244
500	0,4	0,0532709	7,5087899773
1000	0,79	0,106442	7,4218823397
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
	serial (s)	CUDA (s)	speedup
10	0,001	0,00052544	1,9031668697
100	0,01	0,0025673	3,895142757
500	0,09	0,0125035	7,1979845643
1000	0,26	0,0247697	10,4966955595

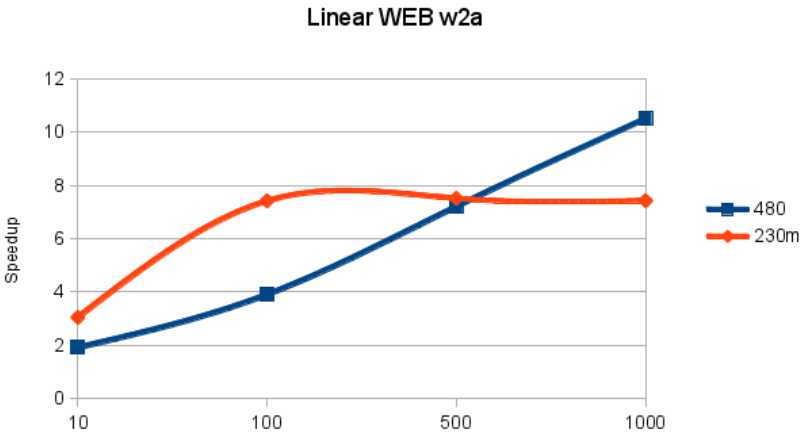

Figure 3.10: Lineare WEB W2a

Table 3.11: Web W2a Polynomial

Polynomial Kernel	Web W2a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,01	0,00371914	2,6887936458
100	0,08	0,0108285	7,3879115298
500	0,45	0,0521863	8,6229527673
1000	0,91	0,106106	8,576329331
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,001	0,000520992	1,9194152693
100	0,01	0,00257779	3,8792919516
500	0,1	0,0123807	8,0770877252
1000	0,29	0,0263467	11,0070710943

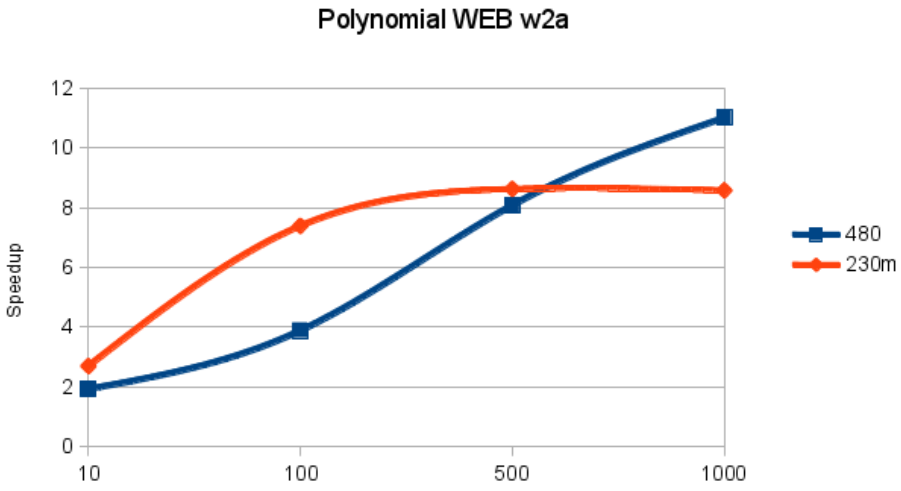

Figure 3.11: Polynomial WEB W2a

Table 3.12: Web W2a Gaussian

Gaussian Kernel	Web W2a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,02	0,0059409	3,3664932923
100	0,12	0,0115097	10,425988514
500	0,55	0,0551013	9,9816156788
1000	1,2	0,111045	10,8064298257
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,001	0,000598656	1,6704083814
100	0,02	0,00277494	7,2073630421
500	0,15	0,0129344	11,5969816922
1000	0,37	0,0252216	14,6699654265

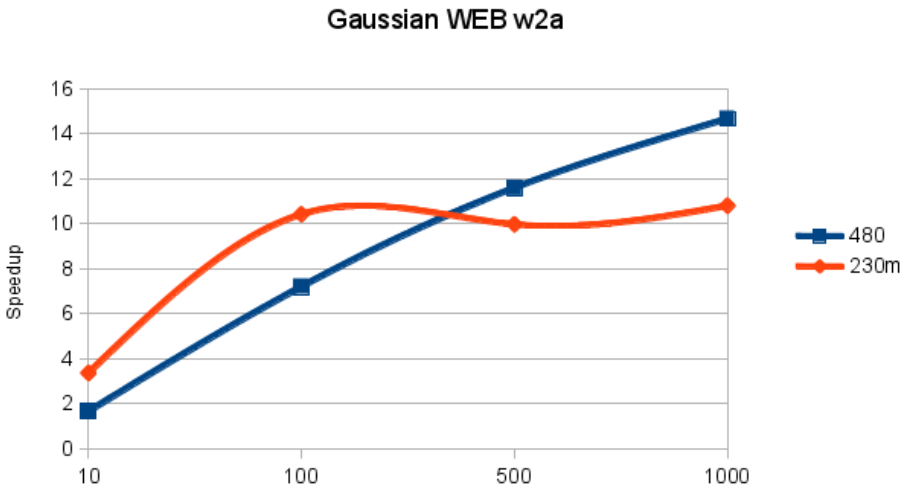

Figure 3.12: Gaussian WEB W2a

Table 3.13: Web W6a Linear

Linear Kernel	Web W6a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
	serial (s)	CUDA (s)	speedup
10	0,01	0,00326442	3,0633313115
100	0,37	0,0500359	7,3946906121
500	1,98	0,255713	7,7430556913
1000	4,01	0,504709	7,9451723667
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
	serial (s)	CUDA (s)	speedup
10	0,01	0,00236877	4,2216002398
100	0,04	0,0116908	3,4214938242
500	0,47	0,057528	8,1699346405
1000	1,37	0,113615	12,0582669542

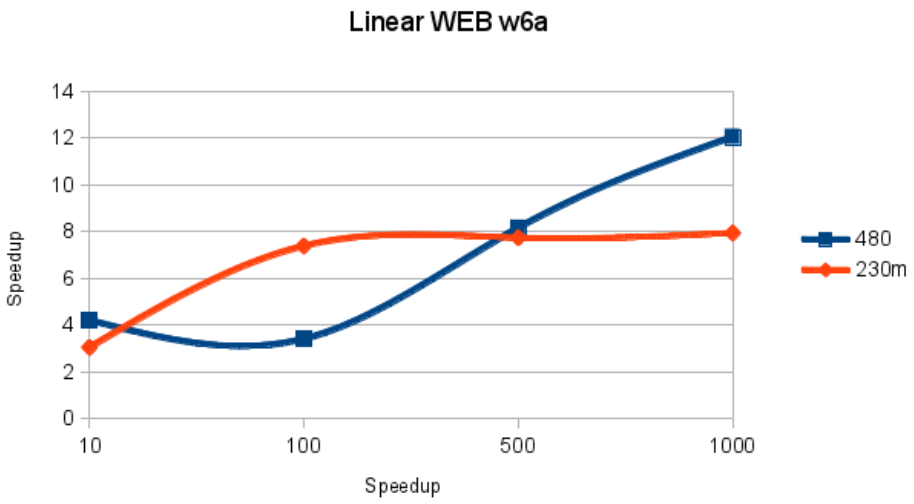

Figure 3.13: Lineare WEB W6a

Table 3.14: Web W6a Polynomial

Polynomial Kernel	Web W6a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,05	0,013566	3,6856848002
100	0,42	0,0511307	8,2142431064
500	2,21	0,256337	8,621463152
1000	4,44	0,510477	8,697747401
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,01	0,00240054	4,1657293776
100	0,05	0,0117557	4,2532558674
500	0,5	0,0573356	8,7205854652
1000	1,45	0,11433	12,6825854981

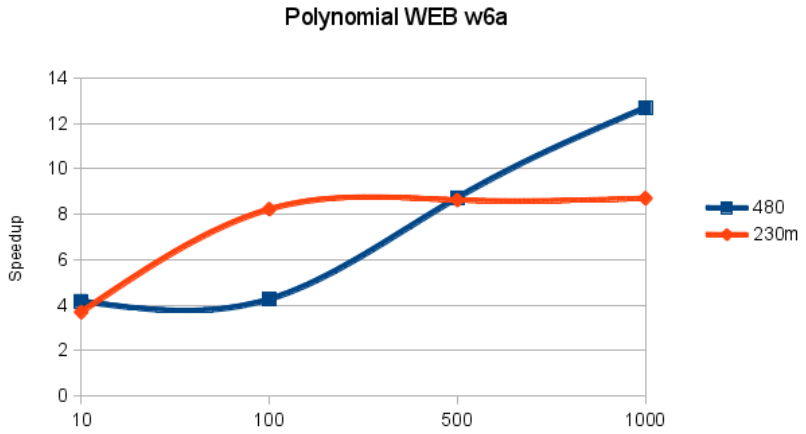

Figure 3.14: Polynomial WEB W6a

Table 3.15: Web W6a Gaussian

Gaussian Kernel	Web W6a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,06	0,0242599	2,4732171196
100	0,51	0,0553841	9,2084190228
500	2,71	0,270197	10,0297190568
1000	5,67	0,537921	10,5405812378
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,002	0,00268989	7,4352482815
100	0,11	0,0121335	9,0658095356
500	0,74	0,0578421	12,7934497537
1000	1,95	0,115106	16,9409066426

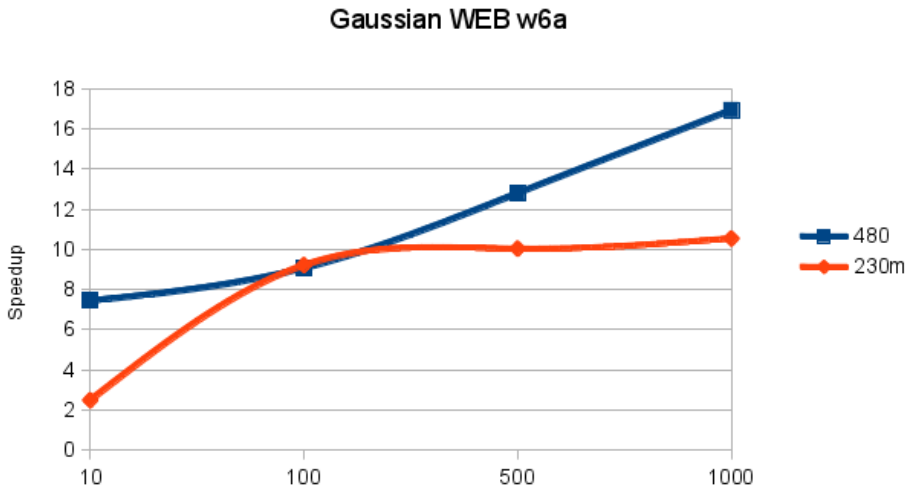

Figure 3.15: Gaussian WEB W6a

Table 3.16: Web W8a Linear

Linear Kernel	Web W8a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
	serial (s)	CUDA (s)	speedup
10	0,01	0,00322403	3,1017081107
100	1,07	0,14351	7,4559264163
500	5,78	0,729963	7,9182095531
1000	11,62	1,45321	7,9960914114
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
	serial (s)	CUDA (s)	speedup
10	0,02	0,00648499	3,0840448482
100	0,13	0,0323527	4,0182117721
500	1,36	0,160225	8,4880636605
1000	3,97	0,321996	12,3293457062

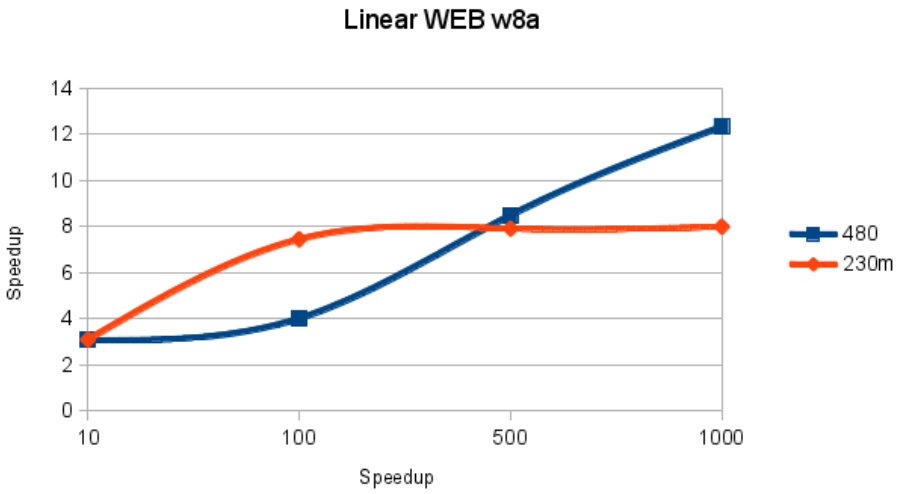

Figure 3.16: Linear WEB W8a

Table 3.17: Web W8a Polynomial

Polynomial Kernel	Web W8a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,16	0,0371677	4,30481305
100	1,21	0,142836	8,4712537456
500	6,33	0,923301	6,8558357459
1000	12,83	1,45356	8,8266050249
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$d = 4$	serial (s)	CUDA (s)	speedup
10	0,02	0,00657395	3,0423109394
100	0,15	0,032406	4,6287724495
500	1,46	0,161989	9,01295767
1000	4,2	0,320264	13,1141808008

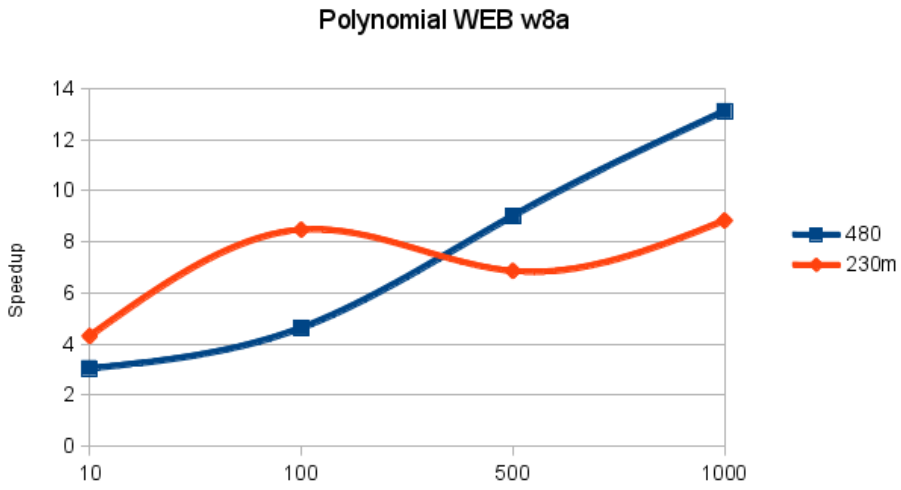

Figure 3.17: Polynomial WEB W8a

Table 3.18: Web W8a Gaussian

Gaussian Kernel	Web W8a		
NVIDIA GTX230m INTEL(R) core i7(R) 860S			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,17	0,0660464	2,5739480123
100	1,47	0,156793	9,3754185455
500	7,83	0,77144	10,1498496319
1000	15,86	1,53483	10,3333919717
NVIDIA GTX480 AMD Phenom(tm) II X6 1090T			
$\sigma = 1800$	serial (s)	CUDA (s)	speedup
10	0,005	0,00747085	6,6926788786
100	0,32	0,0332873	9,6132759341
500	2,16	0,162939	13,2564947618
1000	5,64	0,322644	17,4805668167

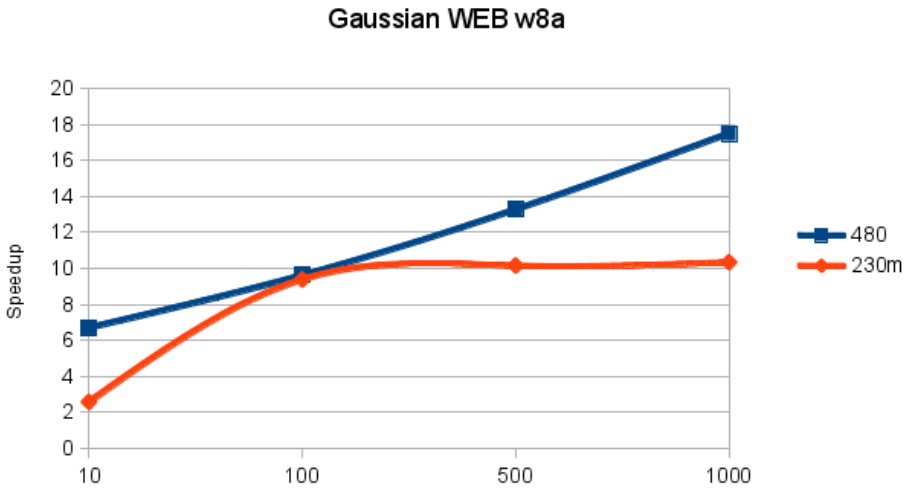

Figure 3.18: Gaussian WEB W8a

Chapter 4

Conclusions

A parallel implementation for facing the kernel function evaluations on GPU is presented. The numerical esperiments on the GPU have shown promising speedups (between 10 to 17) confirming that remarkable improvements can be obtained by including this approach within optimization solvers for machine learning methodologies like Support Vector Machines.

An issue that remains to be solved is concerning the storage of the data on the GPU: our implementation requires that the data involved in the kernel computations are stored on the GPU memory in a nonsparse format and this can be a serious limit in case of very large amount of data.

Future work is necessary for generalizing our approach to the case of sparse storage of the training data.

A way to solve this problem may be the usage of the sparse representation of data, which is already available in the CUSPARSE library, but that needs modifications to be used in our specific application.

Acknowledgements

It's time to thanks all the people who helped me in this long but pleasant journey.

I want to thank Professor Zanni, who followed and helped me during the making of this thesis.
Thanks for the helpful suggestions, for the corrections and especially for his great willingness and patience.
I want to thank Roberto Cavicchioli too, for all the help he has given to me and for all the corrections on my infinite errors in the codes.

I infinitely thank my mother, my father and all my family who have always supported me and for always believing in me even in my most difficult moments.
They have been able to encourage me even when I thought of not succeeding.

Thank to Matteo, Micheal, Fabrizio, Filippo, Mirko, Roberto, Serena, Elisa and all my friends whom I can always found in the physics department's study hall, for helping me all this time and making these years incredible.

Thank to all my Reggio Emilia's friends, perhaps the only people with my family that really was able to put up with me for almost a decade now, always encouraging me and making my life never boring.

Bibliography

[1] C. Cortes, V.N. Vapnik (1995), Support Vector Network, *Machine Learning*, 20, 1-25.

[2] M. Malmusi (1998/1999), Programmazione quadratica di grandi dimensioni in problemi di classificazione binaria, *Tesi di Laurea in Matematica, Università degli Studi di Modena e Reggio Emilia.*

[3] Cristianini N, Shawe-Taylor J (2000) *An Introduction to Support Vector Machines and other Kernel-Based Learning Methods.* Cambridge University Press

[4] Vapnik VN (1998) *Statistical Learning Theory.* John Wiley and Sons, New York

[5] Boser B, Guyon I, Vapnik VN (1992) A training algorithm for optimal margin classifiers. In: Haussler D (ed) *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, ACM Press, Pittsburgh, PA, pp 144–152

[6] Osuna E, Freund R, Girosi F (1997) Training support vector machines: an application to face detection. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR97)*, IEEE Computer Society, New York, pp 130–136

[7] Chang CC, Lin CJ (2001), LIBSVM: a library for support vector machines. URL `http://www.csie.ntu.edu.tw/~cjlin/libsvm`

[8] Keerthi S, Gilbert E (2002) Convergence of a generalized SMO algorithm for SVM classifier design. *Machine Learning* **46**:351–360

[9] Keerthi S, Shevade S, Bhattacharyya C, Murthy K (2001) Improvements to platt's SMO algorithm for SVM classifier design. *Neural Computation* **13**:637–649

[10] Platt JC (1998) Fast training of support vector machines using sequential minimal optimization. In: Schölkopf B, Burges C, Smola A (eds) *Advances in Kernel Methods – Support Vector Learning*, MIT Press, Cambridge, MA

[11] Collobert R, Benjo S (2001) SVMTorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research* **1**:143–160

[12] Hsu CW, Lin CJ (2002) A simple decomposition method for support vector machines. *Machine Learning* **46**:291–314

[13] Joachims T (1998) Making large-scale SVM learning practical. In: Schölkopf B, Burges C, Smola A (eds) *Advances in Kernel Methods – Support Vector Learning*, MIT Press, Cambridge, MA

[14] Serafini T, Zanni L (2005) On the working set selection in gradient projection-based decomposition techniques for support vector machines. *Optimization Methods and Software* **20**:583–596

[15] Zanghirati G, Zanni L (2003) A parallel solver for large quadratic programs in training support vector machines. *Parallel Computing* **29**:535–551

[16] Serafini T, Zanghirati G, Zanni L (2005) Gradient projection methods for quadratic programs and applications in training support vector machines. *Optimization Methods and Software* **20**:353–378

[17] Dai YH, Fletcher R (2005) New algorithms for singly linearly constrained quadratic programs subject to lower and upper bounds. *Mathematical Programming* To appear. Also published as Research Report NA/216, Department of Mathematics, University of Dundee, Dundee, UK

[18] Dai YH, Fletcher R (2005) Projected Barzilai-Borwein methods for large-scale box-constrained quadratic programming. *Numerische Mathematik* **100**(1):21–47

[19] Lin CJ (2001) On the convergence of the decomposition method for support vector machines. *IEEE Transactions on Neural Networks* **12**:1288–1298

[20] Lin CJ (2001) Linear convergence of a decomposition method for support vector machines. Technical report, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

[21] Lin CJ (2002) Asymptotic convergence of an SMO algorithm without any assumptions. *IEEE Transactions on Neural Networks* **13**:248–250

[22] Palagi L, Sciandrone M (2005) On the convergence of a modified version of SVM^{light} algorithm. *Optimization Methods and Software* **20**:317–334

[23] NVIDIA developer site http://https://developer.nvidia.com/what-cuda

[24] NVIDIA: cufft library http://https://developer.nvidia.com/cufft

[25] NVIDIA: cublas library http://https://developer.nvidia.com/cublas

[26] NVIDIA: npp library http://https://developer.nvidia.com/npp

[27] NVIDIA: thrust library http://https://developer.nvidia.com/thrust

[28] NVIDIA: cusparse library http://https://developer.nvidia.com/cusparse

[29] NVIDIA:GPU descriptions http://http://www.geforce.com/hardware/desktop-gpus

[30] NVIDIA: whitepaper Kepler http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[31] NVIDIA: whitepaper Fermi http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[32] LeCun Y, MNIST handwritten digit database. URL http://www.research.att.com/~yann/ocr/mnist

[33] Karnick, Pushpak. *GPGPU: General Purpose Computing on Graphics Hardware.* Pushpak's Home Page. http://http://www.public.asu.edu/~pkarnic/portfolio/papers/IntraVis2006.pdf.

[34] Luebke, David, et al. *GPGPU: general-purpose computation on graphics hardware.* Proceedings of the 2006 ACM/IEEE conference on Supercomputing. ACM, 2006.

[35] Owens, John D., et al *A Survey of general-purpose computation on graphics hardware.* Computer graphics forum. Vol. 26. No. 1. Blackwell Publishing Ltd, 2007.