UNIVERSITÀ DEGLI STUDI "ROMA TRE"

DIPARTIMENTO DI MATEMATICA E FISICA

Dottorato di ricerca in Matematica
XXXI ciclo

Parallel algorithms for cryptography, interacting particles systems and machine learning

Dottorando:
Louis Nantenaina ANDRIANAIVO

Tutor:
Prof. Francesco PAPPALARDI
Prof. Elisabetta SCOPPOLA
Coordinatore del dottorato:
Prof. Angelo Felice LOPEZ

Ottobre, 2019

UNIVERSITÀ DEGLI STUDI "ROMA TRE"

Abstract

Dipartimento di Matematica e Fisica

Doctor of Philosophy

Parallel algorithms for cryptography, interacting particles systems and machine learning

by Louis Nantenaina ANDRIANAIVO

The exponential growth of the new technologies and the development of hardware in modern computers are due to multi-core CPU[1] and powerful GPU[2]. The *High Performance Computing* greatly improved the performance in solving several problems. I discuss in this thesis three different problems:

1) Factorization problem
General Number Field Sieve (GNFS) is known to be the perfect candidate for the factorization task. The procedure is dominated by a step called sieving. Due to the size of the number we want to factor (ex: RSA modulus), the procedure needs to perform the same operation on a large set (Single Instruction Multiple Data). For this purpose, we exploit the features of the GPU to handle this operation, implementing the sieving procedure to run on the GPU. A benchmark that measure the performance of GPU Tesla P100 compared to the CPU (serial and parallel) is provided. We obtained a speed-up of the algorithm which is parameterized by the features of the GPU (amount of cache memory) and the size of the input (cardinality of the factor-base). This experiment is implying a proof of concept which shows that the procedure can benefit from the new generation GPU.
2) Numerical methods in statistical mechanics
The problem of describing the phase transition for the 2D Ising model is approached numerically. As a Markov Chain, the dynamics of this model can be simulated by means of Probabilistic Cellular Automata. Recently, dynamic known as shaken dynamics was introduced on two layers of square lattice. Using this dynamics we are able to estimate numerically the critical curve which separates the ordered and disordered phases on the parameter region (J, q). Furthermore, it induces a procedure suitable for a parallel environment to simulate in real-time the dynamics on GPU. Our experiment can be generalized as a study of the numerical aspect of shaken dynamics. In particular, we compare it with the alternate dynamics on the critical line (bisector $J = q$), also we are able to evaluate numerically the equilibrium distribution of the dynamics in a given region (J, q).
3) Machine learning
Machine learning is one of our daily life applications which causes the fast development of dedicated hardware for High Performance Computing. Image classification belongs to the intersection between machine learning and computer vision which aims to detect the features from input pixel images in an elegant and fast way. As a fact, Deep Convolutional Neural Networks is among the best preforming techniques when dealing with this task. We applied this technique on architectural images to provide a light model with a good performance that can be used in a mobile device. The project can be seen as an instruction guide, starting from scratch for the building of mobile applications. It can be considered as a proposal for the interaction of Artificial Intelligence(AI) with the urban context, a starting point that leads to more complicated tasks in architecture that can be faced by means of AI.

[1]Central Processing Unit.
[2]Graphics Processing Unit.

La crescita esponenziale di nuove tecnologie e lo sviluppo dell'*hardware* nei moderni computer sono dovuti alle CPU *multi-core* e alla potenza delle GPU. L'*High Performance Computing* ha ampiamente migliorato le performance nella risoluzione di molti problemi. In questa tesi, discuto tre diversi problemi:

1) Il problema della fattorizzazione
Il *General Number Field Sieve* (GNFS) è ritenuto un perfetto candidato per il problema della fattorizzazione. La procedura è dominata da un passaggio chiamato *sieving*. A causa della dimensione del numero che si vuole fattorizzare (RSA *modulus* ad esempio), la procedura deve ripetere la stessa operazione su un ampio insieme (*Single Instruction Multiple Data*). A questo fine, si sfruttano le caratteristiche della GPU per gestire l'operazione, implementando la procedura di *sieving* su GPU. Si fornisce un *benchmark* per misurare la performance di una GPU Tesla P100 rispetto alla CPU (seriale e parallela). Si ottiene un'accelerazione dell'algoritmo che dipende dalle caratteristiche della GPU (quantità di memoria cache) e dalla dimensione dell'input (cardinalità della *factor-base*). L'esperimento prevede una *proof of concept* per mostrare come la procedura possa beneficiare della nuova generazione di GPU.
2) Metodi numerici in meccanica statistica
Si affronta numericamente il problema della descrizione della transizione di fase per il modello di Ising bidimensionale. Poiché il modello è una catena di Markov, la dinamica può essere simulata attraverso automi cellulari probabilistici. Di recente, dinamiche conosciute come *shaken dynamics* sono state introdotte su un reticolo quadrato a due livelli. Usando queste dinamiche è possibile stimare numericamente la curva critica che separa le fasi ordinata e disordinata nella regione di parametri (J, q). Inoltre, si introduce una procedura per simulare le dinamiche in tempo reale in ambiente parallelo su GPU. Gli esperimenti condotti possono essere generalizzati come studio degli aspetti numerici delle *shaken dynamics*. In particolare, queste sono messe a confronto con dinamiche alternative sulla retta critica (bisettrice $J = q$) e si può inoltre calcolare numericamente la distribuzione d'equilibrio delle dinamiche in una data regione (J, q).
3) Machine Learning
Il machine learning è una delle applicazioni che producono il rapido sviluppo di *hardware* dedicato per l'*High Performance Computing*. Il riconoscimento di immagine fa parte dell'intersezione tra *machine learning* e *computer vision*, finalizzata a individuare *features* in una immagine *raster* di input in maniera elegante e veloce. Di fatto, i modelli *Deep Convolutional Neural Networks* sono tra le tecniche che mostrano le migliori performance nell'affrontare questi problemi. Queste tecniche sono applicate a immagini con soggetto d'architettura, al fine di ottenere un modello leggero che possa essere usato con buone performance su un dispositivo mobile. Il progetto può essere visto come una guida che parte da zero per la costruzione di applicazioni mobili. Lo stesso può essere considerato una proposta per l'interazione dell'*Artificial Intelligence* (AI) con il contesto urbano, un punto d'inizio che conduca a più complessi problemi nel campo dell'architettura affrontati con AI.

Acknowledgements

I would like to express my special appreciation and thanks to : ***Roberto D'Autilia****, it would never have been possible for me to take this work to completion without him. My supervisors* ***Francesco Pappalardi*** *and* ***Elisabetta Scoppola*** *for their help and support throughout this work.* ***Stefano Guarino****,* ***Alessio Troiani****,* ***Valerio Palma*** *for their suggestions and comments, I benefited a lot on working with them especially on writing this thesis.*
I also thank ***Marco Pedicini****,* ***Roberto Maieli****,* ***Flavio Lombardi*** *and* ***Marco Cianfriglia*** *for giving me the opportunity to work with them.*

Contents

1. Introduction

Computer Science is a field of mathematics which studies algorithms to be implemented on computers. The properties of an algorithm depend both on the problem and the hardware available. There are two main approaches: adapting the algorithm to a given hardware architecture or designing a realizable hardware which can handle the given computation. Many problems known to be computationally hard can be solved using both techniques. Here I focus on the first approach, trying to adapt the algorithms on a given hardware and exploit all its features to obtain a computational speed-up.

A general purpose CPU [1] of last generation has more than one core (multi-thread). To exploit this multiple processor technology, **parallel computing** approach is necessary, i.e., the design of algorithms and data structures that can be processed in parallel. This research field is often referred as High Performance Computing, boosting the performance of the computational power for solving computationally complex problems. In principle, this involves both the property of the hardware and the architecture of the optimized algorithm.

In this thesis, I explore parallel algorithms design and optimization in three different scientific fields, **cryptography, statistical mechanics and machine learning**, assuming that a High Performance Computer is available, that is a dedicated hardware which can be used to speed-up computation. I used last generation GPU[2], which was originally designed to accelerate the graphic operations to produce output on a graphic display. Recently GPUs have been used in scientific computing to obtain important results in several scientific domains. During this project I had the opportunity to use Nvidia workstations DGX-1 occupied by 2×40 cores Intel Xeon CPU and 8 Nvidia Tesla P100 16 Gb of video memory GPU.

I address three different projects: the first is about the factorization problem from which I describe the fastest known algorithm for factorization [BLP92] and provide a possible improvement of its dominant steps. The second is about statistical mechanics, I exploit several numerical simulation methods to study the planar Ising model. This contribution is described in [DNT19]. Finally, I consider machine learning. I explore deep learning methods for classification problems. This work is collected in [ADP19].
The three projects have no direct interaction between them although the common feature is that I treated them by means of High Performance Computing.

In *cryptography*, security depends on a hard mathematical problem. For instance, given a one-way function[3] the challenge is that observing a random output it is difficult to figure out the input. However brute force can break this by enumerating all the possible solutions (note that using random procedure takes the same complexity in the worst cases). An option to overcome this problem is to use an infinite domain however this possibility is not feasible on a computer. Indeed, this level of security

[1] Central Processing Unit
[2] Graphics Processing Unit
[3] non invertible, easy to compute.

must rely on the size of the domain of such functions. A Cryptographic system is said to be computationally secure when any known generic attack is practically unfeasible. More precisely, a cryptosystem is computationally secure when any known generic attack has a complexity bigger than the brute force (or equivalent). Otherwise the cryptosystem is said to be breakable.
Since the running time is a function of the size of the key (private input) it is expected that the brute force has exponential complexity over this size. In this thesis, I discuss a cryptosystem for a problem in *Number Theory* which has been studied for millenniums, the *factorization problem*. Euclid proved the *Fundamental theorem of Arithmetic*: every positive integer greater than 1 can be decomposed as a product of primes. The task is to compute explicitly the prime decomposition. In practice, this is difficult in theory and in application, as well as deciding if a given number is prime or not.
Many theories have been developed and studied to find a computer algorithm to tackle these problems and several attempts have been made to combine deterministic and probabilistic strategies. Probabilistic algorithms based on deterministic techniques offer good running time. For testing primality, the algorithms often belong to the *Monte Carlo* [4] class, unlike for factorization where *Las-Vegas* [5] is used. Here, I go through a probabilistic algorithm for factorization, exploiting the parallelism of the dominant step of the *Gneral Number Field Sieve* so that we can expect a factor speed up of the procedure using the dedicated hardware latest GPU.

One of the challenging problems in *Statistical mechanics* is the description of the phase transitions together with the equilibrium probability measure (state) of the particles system. The *Ising model* is a mathematical model that was used to deal with this task.
The problem can be approached algorithmically by simulating the evolution of the state of the particle system using Markov Chains. A general problem for simulations is that the equilibrium distribution is difficult to compute. A fruitfull approach is to use statistical sampling techniques as *Monte Carlo*, *Gibbsian sampling*, etc[HÖ00]. Based on the recent results in [ADS+19b] and [ADS+19a], I use parallel computing techniques both on multi-CPU and GPU to develop parallel numerical simulation of a large class of 2-dimensional Ising models. Numerically, We recover the critical curve which separates the two phases on the region (J, q), and verify some of the results proved in [ADS+19b, ADS+19a].

Machine learning (ML) is a set of methods in Artificial Intelligence (AI) that aim to teach an artificial agent to perform a particular task. It is based mainly on probability, statistics and algorithmic techniques. ML has extremely vast applications that can be classified in three subjects, *supervised learning*, *unsupervised learning* and *recurrent learning*. In this thesis, I explore the *supervised learning*, that is the ML-agent learn from a labeled dataset to be able to predict the label of unseen data, using *deep learning*, a technique in ML based on Artificial Neural Networks (ANNs). Nowadays, deep learning has become a very active research area in several domains. For

[4] Deterministic strategy, probabilistic correctness of the output.
[5] Always output correct, probabilistic running time.

instance in computer vision traditional methods have been replaced by deep neural networks and offer optimal result on qualitative (good) and quantitative (fast) aspects(see [HGDG17] for example). The principal task on using this kind of technique is the training of the network. The challenge is to train an efficient model with a good latency that can be used in mobile phones.
For training the model, powerful resources are required. I used GPU to handle this step, together with parallel computing techniques that exploit the power of multicore processors and modern graphic cards for high computing. As a result of our experiment, we have trained a model that can be used to recognized architectural objects, implemented in a mobile phone. This work is collected in [ADP19].

Part I

Factorization algorithms

2. Introduction

2.1 Problem definition

In this part, we explore an important algorithm in Number Theory. The factorization problem is a big challenge in number theory and its application. To this purpose many approaches have been exploited by means of techniques derived from mathematics and computational science. Without mentioning the quantum algorithm, the General Number Field Sieve (GNFS) revealed to be the best candidate for this purpose. It uses some theoretical ideas from mathematics and sophisticated algorithmic techniques. It can be considered as an improvement of the sieving algorithm, a technique suggested for the first time in 1975 [AMB75]. The goal is to find a particular pair of integer which can be used to factorize the input number N. Instead of randomly extracting this pair we generate different candidates, called *smooth* candidates. This is indeed the heart of the algorithm.

2.2 Current state of the art and contributions

There are many existing implementations of the General Number Field Sieve, for instance the cado-nfs [Tea17]. As we discuss through this thesis the GNFS algorithm benefits of the hardware architecture design for a parallel environment. This procedure consists of a sequences of steps in which the dominant term is the sieving.
In principle, the **sieving step** consists of finding a sufficient number of *smooth* candidates to perform a prime decomposition. These candidates are supposed to be much smaller than the original input number N so that we can use trial division or other deterministic or probabilistic algorithm to handle the factorization. Basically, the sieve is done in a given interval parameterized by N. Checking for smoothness is independent per each candidate.
The number of sufficient candidates is approximately the running time of the algorithm. Each candidate analysis can be performed at each node in a parallel system. This can be done in any implementation of GNFS since the sieving itself is an *embarrassingly parallel problem*. In particular, we check the implementation in [Tea17], where the sieve is implemented to run on a multi-thread processor.

The contribution of the candidate is a proposal of an implementation for the *sieving step* of the GNFS on GPU by means of lattice sieve. We experiment our implementation of the procedure on GPU and CPU. We try to verify the expected speed-up we observed from the theoretical analysis of the algorithm.

2.3 Preamble

Computing a non trivial divisor for a given N is a fundamental problem in arithmetic. More precisely, the prime decomposition of an integer (as well as giving a witness to confirm that it is indeed prime [LN15]) has been addressed by means of different number theory techniques, and is considered a difficult computational problem. In some cryptographic systems such as RSA, the security relies on the hardness of the factorization problem.

The most difficult case in the factorization problem is when the number N is composed by two prime numbers of similar size, exactly the case in RSA. The main reason of the active contribution of cryptanalisys on the improvements of the factorization techniques. The use of trials is computationally impracticable. The method discussed in this thesis suggests a less computationally expensive procedure than the trial division. The improvement of the algorithm using a dedicated device and a distributed system, plays the most important role exploiting the independence of computation and data so that the running time can be parameterized by the number of processors or nodes used. The sieving methods is the best candidate among such algorithms. It is intended to sieve a B-smooth[1] number in a bounded interval of $\mathbb{Z}$ (B and the bound of the interval are parameterized by the input N).
For this tasks, a randomized algorithm is always the best choice because it offers a realistic running time even if it still depends on the size of the input. The Number Field Sieve [Pol93a] is a probabilistic algorithm, the theory behind the procedure is deterministic but the running time rely on the distribution of B-smooth integers in a given interval.
The procedure took advantage of the arithmetic on a Number Field, i.e., one works on polynomials over an algebraic number. The method was first introduced by Pollard [Pol93a] for a specific number. Later, it was generalized for an arbitrary number in [BLP92]. The procedure is composed of four principal steps: *polynomial generation, sieve, linear algebra* and *GCD computation*. Every step has an important role in the algorithm as it can be seen as a succession of procedures which determine the success of its successor. For instance the success of the GCD computation relies hardly on the choice of the number field (polynomial), the sieve and the linear algebra.
On the other hand, such algorithm can be classified as a sieving algorithm, where the running time is dominated by the sieving procedure, finding a sufficient number of B-smooth candidates. An intuitive observation is that one can proceed the sieving by using an iterative search over the given interval. Given a positive bound B, we extract the list of primes less than B. We mark each element in the candidates if it is divisible by one prime. The candidates which have more marks are likely to be B-smooth. Such technique is known as *line sieve*.
The sieving for the NFS algorithm is different from other technique in factorization algorithm. The candidates must be smooth simultaneously in two domains, rational and algebraic, more precisely, its polynomial evaluation. According to this property, the elements of the factor-base are prime ideal and can be approached by means of line sieve. On the other side, the element of the factor-base can be seen as two dimensional lattice. Pollard [Pol93b] presented a method called *lattice sieve* which is designed to handle the sieving step for NFS which exploits this form of factor-base. Looking closer (chapter 4) at this procedure, one can exploit quite easily its parallel properties.
This part addresses the lattice sieve. The main goal is understanding whether an implementation of the procedure on GPU may yield a significant speed-up of the sieving step and hence of the NFS algorithm. This part is planned as follow: In *Chapter 3*, we have an overview of the Number Field Sieve algorithm and its improvement. In *Chapter 4*, *The lattice sieve*, we give the theoretical description of this procedure, followed by the current state of the art and the discussion of our proposed parallel implementation. In *Chapter 5*, we summarize this part with a possible extension of the project.

[1] All its prime divisor are less than B

3. Literature review

3.1 The Number Field Sieve

The following description is based on [LHWL93]. This contains several papers related to the Number Field Sieve algorithm (NFS), and some improvements. Indeed, the algorithm was first proposed by John Pollard [Pol93a], who suggested its use, by using the property of an algebraic number, and illustrate that method on the seventh Fermat number. Later on, the method was used in [LLMP93a] who succeed to factor the ninth Fermat number. Basically the algorithm is done using the arithmetic on the Number Field. In [LLMP93b] a detailed description of the procedure for a special form integer is given. However, the number sieve algorithm could be generalized for an arbitrary integer. We refer to [BLP92] for this procedure, a such algorithm usually called the General Number Field Sieve (GNFS).

3.1.1 Theoretical description

The Number Field Sieve algorithm depends on the properties of an algebraic number. We construct the number field, by assuming that we have a monic irreducible polynomial f of degree d, which has a root m in the ring $\mathbb{Z}_N$. Let α be a complex root of f, and consider the following homomorphism

$$\begin{array}{rccc} \Phi: & \mathbb{Z}[\alpha] & \longrightarrow & \mathbb{Z}_N \\ & \alpha & \longmapsto & m \end{array}$$

Assumption 3.1.2. $\mathbb{Z}[\alpha]$ is an *unique factorization domain*.

Most of the time, the assumption (3.1.2) does not occur, or $\mathbb{Z}[\alpha]$ is not even an order, although the following description is still valid where the algorithm tries to reduce this amount of error. The case when it is not an order is discussed in the generalization of the Number Field Sieve algorithm.
The elements of $\mathbb{Z}[\alpha]$ are polynomial of degree $d-1$ in $\mathbb{Z}$, and the operation is similar in any ring of polynomial modulo f.

In the Number Field Sieve, the generation of the couple (x, y) is different from the other methods based on the difference of squares (quadratic sieve [Pom82]). Here, we need a set S of coprime (a, b) such that $a - bm$ and $a - b\alpha$ are simultaneously smooth in $\mathbb{Z}$ and $\mathbb{Z}[\alpha]$ respectively. For $a - b\alpha$, we refer to the ideal of $\mathbb{Z}[\alpha]$ generated by this element, the smoothness of an ideal stands when it can be written as a product of prime ideals (UFD). This means that we compute the norm and verify its smoothness in $\mathbb{Z}$, because the prime ideals of $\mathbb{Z}[\alpha]$ are those one which have norm to be a power of prime numbers.
Before giving the detailed description of the algorithm, we give a notification on the factor-base. We consider the two cases for $\mathbb{Z}_N$ and $\mathbb{Z}[\alpha]$.

1. **The rational side $\mathbb{Z}_N$**
   Let B be a positive integer and set

$$Fb(B) = \{p \text{ prime} : p \leq B\}$$

Let $a - bm$ the element we defined before, for $p \in Fb(B)$ divides $a - bm$ implies that

$$\frac{a}{b} \equiv m \mod (p)$$

thus one can define our factor base as

$$Fb(B) = \{(r, p) : p \leq B \text{ prime }, r \equiv m \mod (p)\}$$

This motivates the choice of (a, b) to be coprime, the sieve procedure is done by checking the congruence matching between $\frac{a}{b}$ and r modulo p, for each $p \in Fb(B)$.

2. **The algebraic side** $\mathbb{Z}[\alpha]$
Under the assumption (3.1.2). We set $\beta = a - b\alpha \in \mathbb{Z}[\alpha]$, and let σ_i, for $i = 1, \ldots, d$, the d-conjugates of α in $\mathbb{Q}[\alpha]$, we have

$$\begin{aligned} N(\beta\mathbb{Z}[\alpha]) &= \text{Norm}_{K/Q}(\beta) \\ &= \text{Norm}_{K/Q}(a - b\alpha) \\ &= \prod_{i=1}^{d} \sigma_i(a - b\alpha) \\ &= \prod_{i=1}^{d} (a - b\sigma_i(\alpha)) \\ &= b^d \prod_{i=1}^{d} (\frac{a}{b} - \sigma_i(\alpha) \\ &= b^d f(\frac{a}{b}) \end{aligned}$$

Let B be a positive integer and define the factor-base by

$$Fb(B) = \{p : \text{prime ideal }, N(p) \leq B\}$$

Since $\mathbb{Z}[\alpha]/p$ is a finite extension of $\mathbb{Z}_p$ with degree e_p with p prime number such that $N(p) = p^{e_p}$.

Now, let consider only the extension of degree 1, it means that p is a prime ideal with $[\mathbb{Z}[\alpha]/p : \mathbb{Z}_p] = 1$, define the homomorphism

$$\begin{array}{rccc} \Phi_p : & \mathbb{Z}[\alpha] & \longrightarrow & \mathbb{Z}[\alpha]/p \\ & \alpha & \longmapsto & \alpha + p \end{array}$$

The element of the kernel of Φ_p can be presented by $r \mod (p)$ where $f(r) = 0$ in $\mathbb{Z}_p$. This rise to the definition of the algebraic factor-base as follows,

$$Fb(B) = \{(r, p) : p \leq B \text{ prime }, f(r) \equiv 0 \mod (p)\}$$

and since

$$N((a - b\alpha)\mathbb{Z}[\alpha]) = b^d f(\frac{a}{b})$$

thus we need to verify the congruence matching of r and $\frac{a}{b}$ modulo p during the sieving.

Notices that the positive integer B has a very important contribution of the running time in the algorithm. In the last section, we give the expression of B, which is a parameter that depends on the input N. The algorithm 1 presented below can be seen as the base description of the Number Field Sieve, all the improvements of this are those which deals with the non-occurrence of the assumption (3.1.2).

Algorithm 1 The number field Sieve

Input integer N that we want to factor.
Output a proper factor of N.

1: Choose m and compute f such that $f(m) = 0 \mod (N)$ or $\mod (kN)$ for a chosen positive integer k.
2: Compute the bound B and the two factor-base.
3: Find a sufficient number of candidates $a - bm$ and $a - b\alpha$ $B-$smooth by sieving.
4: Compute the matrix M of the exponents in $\mathbb{F}_2$, each column represents the list of candidates and each row for the exponents.
5: Compute the kernel v of M in $\mathbb{F}_2$ ($Mv = 0$).
6: $X \leftarrow \prod_{i\in I}(a_i - b_i m)$, $Y \leftarrow \prod_{i\in I}(a_i - b_i\alpha)$ such that $I = \{i : v_i = 1\}$
7: $x \leftarrow \sqrt{X}, y \leftarrow \sqrt{Y}$
8: Return $\gcd(x \pm y, N)$

In general, the algorithm 1 can be summarized as follows:
Polynomial Selection (step 1), Sieving (step 2,3), Linear algebra (step 4,5), Square Root (step 6,7), and GCD. Throughout this chapter, we give a description of some existing approach for each of these steps and their improvements.

The assumption (3.1.2) is actually a strong assumption on f, considered in [Pol93a] and [LLMP93b] for particular form of N. Also we assumed in the algorithm that those elements in $\mathbb{Z}[\alpha]$ with even inertial degree are square in $\mathbb{Z}[\alpha]$. In [BLP92], the authors generalized these by a four obstructions.

Let consider the set S to be the set of coprime (a, b) found in the step 5 in the algorithm above, which verify

$$\sum_{(a,b)\in S} e_p(a - b\alpha) \equiv 0 \mod (2)$$

for all $p \in Fb(B)$. Here we consider the algebraic side,

1. The ideal $\prod_{(a,b)\in S}(a - b\alpha)\mathcal{O}_K$ may not be a square of an ideal. Due to the property that this is an ideal of $\mathcal{O}_K$ where one works in $\mathbb{Z}[\alpha]$.
2. The ideal $\prod_{(a,b)\in S}(a - b\alpha)\mathcal{O}_K$ is a square of an ideal but may not be a principal ideal.
3. The ideal $\prod_{(a,b)\in S}(a - b\alpha)\mathcal{O}_K$ is a square of a principal ideal but the element $\prod_{(a,b)\in S}(a - b\alpha)$ is not square.
4. The ideal $\prod_{(a,b)\in S}(a - b\alpha)\mathcal{O}_K$ is a square of a principal ideal and the element $\prod_{(a,b)\in S}(a - b\alpha)$ is square but its square root is not in $\mathbb{Z}[\alpha]$.

These four obstructions have been estimated for Number Field algorithm. We give an improvement of the procedure by taking care of these obstructions.

The first obstruction is the more occurring case, which says that $\mathbb{Z}[\alpha]$ is not the maximal order of $\mathbb{Q}[\alpha]$. An attempt is to minimize the quantity $[\mathcal{O}_K : \mathbb{Z}[\alpha]]$ during *polynomial selection*. An other idea is to estimate the difference and making sure that the element we found has better chance to be in the order $\mathbb{Z}[\alpha]$ rather than only $\mathcal{O}_K$, this is based on a probabilistic strategy.

In [BLP92] the authors gave a bound for the amount of this obstruction. Indeed, for any order A of K, and for all prime p ideal of A, let the group homomorphism $e_p : K^* \longrightarrow \mathbb{Z}$. The existence of e_p is showed in [BLP92] and it satisfies the following conditions,

1. $e_p(x) \geq 0$ for all $x \neq 0 \in A$.
2. If x is non-zero in A, then $e_p(x) > 0$ if and only if $x \in p$.
3. For each $x \in K^*$ one has $e_p(x) = 0$ for all but finitely many p, and

$$\prod_p (N(p))^{e_p(x)} = |Norm(x)|$$

p ranges over the primes ideal of A.

In our case, where $A = \mathcal{O}_K$, e_p is the inertial degree. Now by letting

$$V_A = \left\{ x \in K^* \ : \ e_p(x) \equiv 0 \mod 2 \ , \forall p \text{primes ideal of } A \right\}$$

which is a subgroup of K^*, an upper bound of its quotient with the group of square element K^{*2} of K^* gives an estimation of the above difference. It is given in the following theorem

Theorem 3.1.3 ([BLP92]). *Let n, d be integers with $d \geq 2$ and $d^{2d^2} < n$, and let m, f be as produced by the base m algorithm[1]. Given a number field K and V as defined above, we have $dim_{\mathbb{F}_2}(V/K^{*2}) < \log(n)/\log(2)$.*

Notices that the assumption of this theorem (3.1.3) can be used in the first step of the number field sieve algorithm to generate f, the polynomial selection. The proof of theorem (3.1.3) is based on algebraic techniques, it deals fundamentally on the discriminant of f and is based on the first three obstruction. In practice, if we use these arguments, in the polynomial selection, then it requires a computation of the discriminant of f and factor it, follow by the computation of the Ideal Class Group of the number field. To summarize, we try to make the maximal order to be a Principal Ideal Domain(PID) and this can handle all the first three obstructions. However, this is very expensive in terms of running time and we do not want the algorithm to be dominated by the polynomial selection.

On the other hand, Adleman [Adl91] idea to deal with the second and third obstructions was to add more rows (rows of character) in the *linear algebra* steps. The methods was also used in [BLP92], and has been shown that it also can be used to handle the first obstruction. The idea is to design a probabilistic strategy, saying that if the *Legendre* symbol of the candidate we found is 1 for a sufficient number of prime then the candidate has more chances to be square, otherwise it is definitely

[1] This is a digits extension of N in the base m

not square for a given set of an ideal (Quadratic character factor-base). According to these properties, The following theorem was exploited to improve the algorithm

Theorem 3.1.4. *Let S be a finite set of coprime integer (a, b) with the property that*

$$\prod_{(a,b)\in S} (a - b\alpha)$$

is the square of an element in K. Further let q be an odd prime number and s with $f(s) \equiv 0 \mod (q)$ such that $a - bs \neq 0 \mod (q)$ for each $(a, b) \in S$ and $f'(s) \neq 0 \mod (q)$. Then

$$\prod_{(a,b)\in S} \left(\frac{a - bs}{q}\right) = 1$$

The modification of the Number Field Sieve algorithm uses the converse of this theorem, one reason makes the procedure to be probabilistic, saying that the candidate is not only of the even e_p but has its Legendre symbol equal to 1. We can add the following steps in the linear algebra (step 4 in algorithm 1),

1. Define a Quadratic character factor-base $Qb(B_1)$ for a given bound B_1 (B_1 is a parameter to the probability of the element we get from the linear algebra to be a square), and for every $(q, s) \in Qb(B_1)$, q is prime such that $f(s) \equiv 0 \mod (q)$, $f'(s) \neq 0 \mod (q)$ and $(q, s) \notin Fb(B)$.
2. In the Matrix used in the linear algebra, the entries of the character columns is 0 if $\left(\frac{a - bs}{q}\right) = 1$ and 1 if $\left(\frac{a - bs}{q}\right) = -1$.

For the forth obstruction, we use $f'(\alpha) \prod_{(a,b)\in S}(a - b\alpha)$ in the square root part, and this is indeed in $\mathbb{Z}[\alpha]$.

3.2 The General Number Field Sieve

The procedure introduced [Pol93a] by Pollard was used to factor the F_7 seventh Fermat number. More precisely, the number field was an extension of the root of $f(X) = X^3 + 2$. In [LLMP93b], a number of particular form $N = r^e - s$ (where r and $|s|$ are a small positive integer with a large e) was presented. These methods were based on the fact that it is possible to construct a monic irreducible polynomial with small size easily and has a better chance to overcome the four obstructions we gave before.
In [BLP92] the generalization of the Number Field Sieve was suggested. It introduces the homogeneous polynomial which extends the procedure to be valid for an arbitrary number. Let

$$f(X) = a_d X^d + \ldots + a_0 \quad \in \mathbb{Z}[X]$$

and

$$F(X, Y) = a_d X^d + a_{d-1} X^{d-1} Y + \ldots + a_1 X Y^{d-1} + a_0 Y^d$$

its homogenized polynomial. We assume that $F(X, Y)$ is irreducible over $\mathbb{Z}[X, Y]$ (otherwise we find a factor and consider the irreducible factor), and let (m_1, m_2) be a couple of integer such that

$$F(m_1, m_2) \equiv 0 \mod (N)$$

let α be a complex root of $F(X,1)$ and consider Φ the homomorphism defined by

$$\begin{array}{rccc} \Phi: & \mathbb{Z}[\alpha] & \longrightarrow & \mathbb{Z}_N \\ & \alpha & \longmapsto & \frac{m_1}{m_2} \end{array}$$

Notices first that $\mathbb{Z}[\alpha]$ is not even an order here. As in the original procedure, we have defined the first degree prime ideal over an order, and used it to describe the algebraic factor-base.
Let ω be a complex root of $F(X,a_d)$, obviously w is an algebraic integer, and set $\alpha = \frac{\omega}{a_d}$ a complex root of $F(X,1)$.

Proposition 3.2.1. [BLP92]
Let $\beta_0,\ldots,\beta_{n-1} \in \mathbb{Z}[\alpha]$ such that $\sum_{i=0}^{n-1} \beta_i X^i = \frac{F(X,1)}{(X-\alpha)}$. Define

$$A = \mathbb{Z} + \sum_{i=0}^{n-1} \mathbb{Z}\beta_i$$

we have A is an order in the number field $\mathbb{Q}(\alpha)$ with $A = \mathbb{Z}[\alpha] \cap \mathbb{Z}[\alpha^{-1}]$

Using the order A defined above, we can give a description for the first degree prime ideal. Let

$$\begin{array}{rccc} \Phi_p: & A & \longrightarrow & A/p \\ & \gamma & \longmapsto & \gamma + p \end{array}$$

where p is a prime ideal such that $[A/p : \mathbb{Z}_p] = 1$ with p prime number.
The element $\gamma + p$ in the kernel of Φ_p can be seen as a projective line $(r_1 : r_2)$ such that $F(\frac{r_1}{r_2}, 1) = 0$ in $\mathbb{Z}_p$, more precisely we do have the following two cases,

case $r_2 \neq 0$, we take the kernel intersects with $\mathbb{Z}[\alpha]$.

case $r_2 = 0$, we take the kernel intersects with $\mathbb{Z}[\alpha^{-1}]$, in here a_n is divisible by p, we refer this as the point at infinity.

During sieving, as we will consider the couple (a,b) the exponent in the norm becomes

$$exp(a-b\alpha) = \begin{cases} e_p(a-b\alpha) & \text{if } r_2 \neq 0 \\ e_p(a-b\alpha) + v_p(a_n) & \text{if } r_2 = 0 \end{cases}$$

The algebraic factor-base is given by,

$$\begin{aligned} Fb(B) = &\left\{[(r:1),p] \in P^1(\mathbb{Z}_p) : p \leq B, F(r,1) \equiv 0 \mod (p)\right\} \\ &\cup \left\{[\infty,p] \in P^1(\mathbb{Z}_p) : p \leq B, a_n \equiv 0 \mod (p)\right\} \end{aligned}$$

Let N be an integer we want to factor, we give the following algorithm 2 as the generalization of the Number Field Sieve:

In this algorithm 2, we have the *polynomial selection* in the steps 1, 2 and 3, the optimal choice of m_1, m_2 and the construction of F are given in the next section. The *Sieve*

Algorithm 2 GNFS

Input integer N that we want to factor.
Output a proper factor of N.

1: Choose an optimum m_1, m_2.
2: Compute $F(X,Y) = a_d X^d + a_{d-1}X^{d-1}Y + \ldots + a_0 Y^d$ such that $F(m_1, m_2) \equiv 0 \mod (N)$
3: Set $G(X,Y) = m_2 X - m_1 Y$.
4: Compute the bounds for the factor-base (rational, algebraic).
5: Find a sufficient number of candidates (a,b) with $F(a,b)G(a,b)$.
6: Compute the character factor-base and construct the matrix M of exponent in $\mathbb{F}_2$.
7: $v \leftarrow \text{kernel}(M)$.
8: $I \leftarrow \{i : v_i = 1\}$.
9: $\gamma \leftarrow \left(\frac{F_X(\omega, a_d)}{a_d}\right)^2 \prod_{i\in I}(a_d a_i - \omega b_i), \sigma \leftarrow \sqrt{\gamma} = \sum_{i=0}^{n-1} v_i \omega^i$
10: $s \leftarrow \sum_{i=0}^{d-1} v_i a_d^i m_1^i m_2^{d-1-i}$, $e \leftarrow m_2^{\frac{\#I}{2}}$, $h \leftarrow a_d^{(d-2)+\frac{\#I}{2}} F_X(m_1, m_2)$ in $\mathbb{Z}_N$.
11: $X^2 \leftarrow \prod_{i\in I}(m_2 a_i - m_1 b_i)$
12: Return $\gcd(hX \pm es, N)$

part are composed by the steps 4 and 5. *Linear algebra*, the steps 6, 7. Notices that the character factor-base should not have an intersection with the algebraic factor base, the construction of the matrix M can be seen as follows

- The first row is the sign of G
- The next rows are the exponent of the prime in the factor-base rational and algebraic.
- The next rows are the contents of the character factor-base, here we have 0 if $\left(\frac{a-bs}{q}\right) = 1$ and 1 otherwise.
- A last row which assigned to 1.

The need of the last row is because we want the cardinal of I to be even. The steps 8,9,10 and 11 are the setting for the *square root* part, here F_X represents the partial derivative of F with respect the variable X. The setting of these variables are given as follows, since γ is square in $\mathbb{Z}[\omega]$, and we have

$$\Phi(\frac{\omega}{a_d}) = \frac{m_1}{m_2} \mod (N)$$

which implies

$$\Phi(\omega) = a_d \frac{m_1}{m_2} \mod (N)$$

and

$$\Phi(m_2^{d-1}\sigma) = s \mod (N)$$

therefore,

$$
\begin{aligned}
e^2 s^2 &= \Phi(m_2^{2(d-1)+\#I}\sigma^2) \\
&= \Phi(m_2^{2(d-1)+\#I}\gamma) \\
&= \Phi\left(\left[m_2^{d-1}\frac{F_X(\omega, a_d)}{a_d}\right]^2 \prod_{i\in I} m_2(a_d a_i - \omega b_i)\right) \\
&= \Phi\left(\left[\frac{F_X(a_d m_2\alpha, m_2 a_d)}{a_d}\right]^2 \prod_{i\in I} a_d(a_i m_2 - m_2 b_i\alpha)\right) \\
&= a_d^{2(d-2)} F_X(m_1, m_2)^2 a_d^{\#I} X^2 \mod (N) \\
&= h^2 X^2 \mod (N)
\end{aligned}
$$

and this proved the GCD from the last step of the algorithm 2.

3.3 Steps of the Number Fields Sieve

All the steps presented in the algorithm 2 have been improved to make the GNFS to be the fastest known algorithm for factorization on classical computers. The techniques used are based on the mathematical properties of the algorithm. The exploitation of the independence of the steps in the algorithm makes it to be suitable for parallel environment. In this section, we give a description of each steps of the NFS and some of their improvements.

3.3.1 Polynomial Selection

Let d be the degree of the polynomial, and $m \approx N^{\frac{1}{d}}$, the base-$m$ expansion was the first candidate. From [BLP92], a wise choice of d could result a monic polynomial f for the NFS. However, for $0 \leq i \leq d-2$ we have $a_i \leq m$, the coefficients can not be controlled even making a few manipulation on them.
In [BLP92], the author suggest the homogeneous polynomial especially for the GNFS, an optimization for the coefficients of the polynomial is also introduced. This later is due to the fact that small polynomial produces small number which are more likely to be smooth. The authors suggested three methods based on improvements of the polynomial selection.
Let assume that we have an integer N as input, and let a_d be the leading coefficient of the degree d polynomial, m_1 and m_2 the pair root of the polynomial F modulo N, by considering three cases, the generation of $F(X, Y)$ is given as follows

case $m_2 = 1$: The procedure is exactly the base-m_1 expansion (with $m_1 \approx N^{\frac{1}{d+1}}$).

case $a_d = 1$: Let $m_1 \approx N^{\frac{1}{d+1}}$, $m_1 \approx N^{\frac{1}{d+1}}$ such that $(m_1, m_2) = 1$ and $N - m_1^d$ be a multiple of m_2. The rest of the coefficients a_i of F can be deduced by

$$\frac{N - m_2^d}{m_2} = a_{d-1}m_1^{d-1} + \ldots + a_0 m_2^{d-1}$$

modulo m_2^i for $1 \leq i \leq d-2$.

case $a_d \neq 1$ and $m_2 \neq 1$: We consider the lattice

$$L = \left\{ (x_i)_{i=0}^{d} : \sum_{i=0}^{d} x_i m_1^i m_2^{d-i} \equiv 0 \mod (N) \right\}$$

with a trivial basis $(0, 0, \ldots, N)$ and $(0, \ldots, 1, -\frac{m_2}{m_1} \mod (N), \ldots, 0)$. All the coefficients of the polynomial are given by a short vector basis such that

$$\sum_{i=0}^{d} x_i m_1^i m_2^{d-i} \neq 0$$

Notice that m_1 is far smaller than N so if we find a divisor of N the procedure is unnecessary.

An attempt is to manipulate the coefficients, also one allows the use of a positive integer multiplier k and apply the same procedure for kN. This technique is similar to the *continued fraction* [AMB75] methods. In fact, the multiplier k is not only needed to prevent the short period but also to induce a parameter to quantify the smoothness of the Q's candidate, the same purpose for the *multi-polynomial quadratic sieve* [DS87]. By defining the Knuth-Schroppel function, one deduces k from the maximum of a such function.

B. Murphy [Mur99] in his Ph.D thesis, used the same approach to quantify the smoothness of the polynomial used in the Number Field Sieve. The idea is to define a new parameter that will quantify the root property of the chosen polynomial. This is due to the fact that, if a polynomial has many roots modulo small prime then it has a large chance to be smooth.
In principle, Murphy's idea was to compare, for the sieving interval, the behavior of the polynomial value with a random integer with the mean of its smoothness.

Definition. Let S be a sample of value, and $v \in S$, p a prime and B a positive integer. Define $ord_p(v)$ the largest power of p which divides v and $cont_p(v)$ is the expected value of $ord_p(v)$

For a chosen v in the sample S, we have

$$\log(v) = \sum_{p \leq B} cont_p(v) \log(p)$$

If we set S to be a sample of F-value (resp. f), for an homogenized polynomial F (resp. for a polynomial $f(X) = F(x, 1)$), we give an estimate for the value $\log(F)$ (resp. $\log(f)$) which are given by

$$cont_p(i_r) = \frac{1}{p-1}$$

$$cont_p(f) = \frac{r_p}{p-1}$$

$$cont_p(F) = \frac{r_p p}{p^2-1}$$

which are the expected values of the ord_p for each case, where r_p represent the number of distinct roots of f(or F) modulo p. By considering the difference between the

F and a random i value we have

$$\log(i_r) - \log(F) = \sum_{p \leq B} \left(1 - \frac{r_p}{p+1}\right) \frac{\log(p)}{p-1}$$

From this we define a new **parameter** $\alpha(F)$, which for f-value it is given by

$$\alpha(f) = \sum_{p \leq B} (1 - r_p) \frac{\log(p)}{p-1}$$

This parameter is used as an adjustment of the property of the polynomial F (or f) used during the sieving. One observes that for a negative $\alpha(F)$ the value of the polynomial f is more likely to be smoother than the random integer. This parameter is called the *root property* since it depends on the root of the polynomial, if the number of distinct roots of F modulo p is big then the value of $\alpha(F)$ is negative.
Murphy described a procedure to use this quantifier, a pre-sieving alike procedure, assuming that we can use one of the algorithm above to produce the polynomial F and G. The algorithm takes a pair of polynomials (F, G) and a bound B, and computes the root property for several polynomial rotated by an affine polynomial $j_1 x - j_0$ with $|j_0| < J_0$ and $|j_1| < J_1$, where J_0, J_1 are a fixed bound such that $J_0 \ll J_1$. For a given polynomial F, G and f, g their dehomogenized polynomials, we refer the rotated polynomial with $ux - v$ by

$$f_{u,v} = f(x) + (ux - v)g(x)$$

This procedure can be coded as follows.

Algorithm 3 Murphy root property

Input Polynomial f, g, B, J_1 .
Output List Global$\alpha(f_{j_0,j_1})$
1: Global$\alpha(f_{j_0,j_1}) \leftarrow$ List()
2: Partial$cont_{p^k} \leftarrow$ List()
3: **for** each $(k, p) : p^k < B$ **do**
4: **for** each $j_1 < J_1$ **do**
5: **for** each $l < p^k$ **do**
6: $j_0 \leftarrow$ solve(Mod$(f_{j_1 j_0}(l) \equiv 0, p^k)$)
7: update(Partial$cont_{p^k}(f_{j_1 j_0})$
8: **end for**
9: **end for**
10: merging(Global$\alpha(f_{j_0,j_1})$)
11: **end for**
12: Return Global$\alpha(f_{j_0,j_1})$

Using the algorithm 3, we identify the good root property polynomial from a two-dimensional list. Shi Bai [ShI11] gave an improvement of this procedure, by observing that the complexity of the Murphy algorithm is heavy. His idea is to be able to lift the root for the power of prime which can be identified by the Hensel theorem. The number of polynomials that we quantify can be very large, and depends on the bound of the rotation, it is approximately $p_i - 1$ for each $p_i^k \leq B$. The procedure is done in two steps. The first one is to identify the polynomial which has many roots modulo small prime (this can be done for each small prime and using Chinese

Reminder Theorem to construct one polynomial). The second step, is to use the rotation with a given large bound.

On the other hand, one needs to quantify the size of the polynomial. Let $F(x,y)$ be the homogenized of f we use in GNFS. In [BLP92], the degree d is chosen using the complexity of the algorithm given in section 3.5.2. We assume to sieve on a rectangle (a,b) so that the probability of the polynomial is $y-$smooth (using the same notation as in the study of complexity in chapter 3.5) and is given by

$$\rho\left(\frac{\log(|F(a,b)|)}{\log(y)}\right)$$

the same for the affine polynomial in the rational side G

$$\rho\left(\frac{\log(|G(a,b)|)}{\log(y)}\right)$$

Therefore the number of candidates (a,b) we are expecting will be approximately

$$\frac{6}{\pi^2}\iint\limits_{(-u,u)\times(0,u)}\rho\left(\frac{\log(|F(a,b)|)}{\log(y)}\right)\rho\left(\frac{\log(|G(a,b)|)}{\log(y)}\right)\mathrm{d}a\mathrm{d}b \tag{3.3.1}$$

The constant $\frac{6}{\pi^2}$ comes from the fact that we collect coprime candidates, also notice the $G-$ value does not really affect (3.3.1) much this expression. We will assume that the F-value contributes more, the optimization of the size of F is the *size property* .

To summarize, the polynomial selection generates many polynomials and we classify them using the root and size property, by assuming that we can find one candidate which are simultaneously passed the root and size property, we can perform the next step considering this *polynomial*.
We give a technique mainly used to optimize the size of the polynomial. Notices that in the original description of the General Number Field Sieve [BLP92] the coefficients of the polynomial were adjusted. The authors claimed that for a particular choice of m, the base-m algorithm gives a monic polynomial which is intended to minimize the absence of the assumption (3.1.2). The same idea could be used for size property. We keep the rotation and add a translation by some amount on the root of polynomial. To obtain a polynomial with good size property, we use a similar gradient descent over the amount t. In Murphy [Mur99] the quantification of the size is done before the root property.

The following idea is used for the size property, more precisely by computing the quantity (3.3.1). Let u_1 and u_2 be the respective bound on a and b in the sieving region, to make everything in harmony, u_1 and u_2 will have the same magnitude as the optimal choice in section 3.5.2. We define a **skewness** s, a constant equal to $s=\frac{u_2}{u_1}$. The rectangle of the sieving becomes $((-u\sqrt{s},u\sqrt{s})\times(0,\frac{u}{\sqrt{s}}))$, by using the Log L^2-norm (3.3.1) can be written as,

$$\frac{1}{2}\log\left(s^{-d}\int_{-1}^{1}\int_{-1}^{1}F^2(as,b)\mathrm{d}a\mathrm{d}b\right)$$

and we now have more variables to find a minimization of the expression.

Bai [ShI11] used a quadratic rotation $wx^2 + ux + v$, a translation t, and uses a local gradient descent over the fives w, u, v, t, s variables to quantify the size fo the polynomial. Assume that we have a polynomial of good root property, we could try to compute an optimal candidate polynomial in size generated by translation since it keeps the root property. The use of the parameter s changes the shape of the sieving area, which are obtained from the size quantifying algorithm.

To summarize, the search for a good polynomial in size and root can be approached by rotation and translation, also the use of skewness s is important since it changes the shape of area in which we can control the upper bound of the polynomial value. The following procedure can be used for the **polynomial selection** step, the algorithm is based on Murphy and Montgomery idea.
Given a number N , degree d and $a_{d,max}$ a bound for the leading term, we start by choosing $a_d = 1$,

Algorithm 4 Polynomial selection

Input Integer $N, d, a_{d,max}$.
Output Polynomial (f, g)
1: choose a_d {a product of prime, $\leq a_{d,max}$}
2: $m \leftarrow \left\lfloor \sqrt[d]{\frac{N}{a_d}} \right\rfloor$
3: $a_i \leftarrow \text{base}_m(N)$
4: **if** a_{d-2} is not sufficiently small **then**
5: **go to** 1
6: **else**
7: $f \leftarrow a_d x^d + \ldots + a_0$
8: $g \leftarrow x - m$
9: **end if**
10: $f, g \leftarrow$ **Rootproperty**(f, g)
11: $f, g \leftarrow$ **Sizeproperty**(f, g)
12: **if Sizeproperty** fails **then**
13: **go to** 1
14: **else**
15: Return (f, g)
16: **end if**

Remarks.

- In steps 1 and 4, we have already an optimization of the coefficients, this makes the step 11 less heavy.
- In steps 10 and 11, we applied all the methods we have discussed above, also we can combine the optimal check for the root and size by keeping the following small

$$\alpha(F) + \frac{1}{2} \log \left(s^{-d} \int_{-1}^{1} \int_{-1}^{1} F^2(as, b) \mathrm{d}a\mathrm{d}b \right)$$

The idea behind the generation of the two homogeneous polynomials F and G for

the GNFS is based on the algorithm 4. An improvement of this procedure was suggested by Kleinjung [Kle06]. It was a modification on the choice of m to control the coefficients a_{d-1} and a_{d-2}. This idea can be seen as follow.
Assume N, d and a_d given, we choose m_2 to be a product of small prime, and m_1 is the solution of

$$a_d x^d \equiv N \mod (m_2)$$

with the constraint $m_1 \approx \left(\frac{N}{a_d}\right)^{\frac{1}{d}}$ by defining the following recursive relationship for $d-1 \geq i \geq 0$, we construct the base-(m_1, m_2) expansion algorithm to get the rest of the coefficients of F and $G(X) = m_2 X - m_1$

$$r_i = \frac{r_{i+1} - a_{i+1} m_1^{i+1}}{m_2}$$
$$a_i = \frac{r_i}{m_1^i} + \delta_i$$

where $0 \leq \delta_i \leq m_2$, $r_i \equiv a_i m_1^i \mod (m_2)$ and $r_d = N$. It is a generalization of the case 2 of the suggestion in [BLP92].
In this procedure, we have a parameter to adjust the amount of coefficient generated. For instance the error term in $m_1 \approx \left(\frac{N}{a_d}\right)^{\frac{1}{d}}$ plays an important rule, as well as the δ_i. The main objective is to have as much polynomials, pre-optimized size on the coefficients. We consider a bound to control a_d, and by the base-(m_1, m_2) algorithm it can be used over the other coefficients. Also in [Kle06], the author suggest the use of Log $L^\infty-$norm over the coefficients. In fact, let $M < N^{\frac{1}{d+1}}$, the skewness s such that $a_d s^{\frac{d}{2}} < M$ and define the log $L^\infty-$norm as follow,

$$\log(L^\infty(F)) = \log(\max_{0 \leq i \leq d} |a_i s^{i-\frac{d}{2}}|)$$

Their practice results showed that an optimal skewness s could be chosen in $\left(\frac{m_0}{M}\right)^{\frac{2}{d-2}} \leq s \leq \left(\frac{M}{a_d}\right)^{\frac{2}{d}}$ with $m_0 = \left(\frac{N}{a_d}\right)^{\frac{1}{d}}$. Therefore a_d is in the range $a_d \leq \left(\frac{M^{2d-2}}{N}\right)^{\frac{1}{d-3}}$. According to this a_{d-1} is bounded by $\left(\frac{M^2}{m_0}\right)$, even though a good choice of m_1 will produce a small a_{d-1}. The coefficient a_{d-2} is bounded by $\left(\frac{M^{2d-6}}{m_0^{d-4}}\right)^{\frac{1}{d-2}}$.
To make this in practice, the following are the formal technique behind Kleinjung algorithm.

Setup :

- Set $m_0 = \left(\frac{N}{a_d}\right)^{\frac{1}{d}}$ and $m_2 = \prod_{i=1}^{l} m_{2,i}$ where $m_{2,i}$ are small primes with

$$m_{2,i} \equiv 1 \mod (d)$$

and

$$a_d x^d \equiv N \mod (m_2)$$

has d solutions.

- We write each solution as
$$x_\mu = \sum_{i=1}^{l} x_{i,\mu_i}$$
with $\mu_i = 1, \ldots, d$. We have $(i, x_{i,j})$, $1 \leq j \leq d$ the d solutions of
$$N \equiv a_d x^d \mod (m_{2,i})$$
with $0 \leq x_{i,j} \leq m_2$
- Set $\overline{m_0} = [m_0]$, for m_2 divides $\overline{m_0}$, we define
$$m_{i,j} = \begin{cases} \overline{m_0} + x_{i,j} & i = 1 \\ x_{i,j} & i > 1 \end{cases}$$
For each μ such that
$$m_\mu = \sum_{i=1}^{l} = \overline{m_0} + x_\mu$$
we perform the base-$(m_{1,\mu}, m_2)$ to generate the other coefficients. The $a_{d-1,\mu}$ and $a_{d-2,\mu}$ can be written in function of x_{i,μ_i} using the following lemma.

Lemma 3.3.2 ([Kle06]). Given N, m_2, d, a_d, μ and $m_{1,\mu}$, there exist an integer $0 \leq e_{i,j} \leq m_2$ for $1 \leq i \leq l$ and $1 \leq j \leq d$ such that
$$a_{d-1,\mu} = \sum_{i=1}^{l} e_{i,\mu_i}$$
satisfy
$$a_{d-1,\mu} m_{1,\mu}^{d-1} \equiv \frac{N - a_d m_{1,\mu}^d}{m_2} \mod (m_2)$$
with
$$\begin{cases} e_{1,j} & \equiv & a_{d-1,(j,1,\ldots,1)} \mod (m_2) \\ e_{i,1} & \equiv & 0 \text{ for } i > 1 \\ e_{i,j} & \equiv & (a_{d-1,(1,\ldots,j,\ldots,1)} - a_{d-1,(1,\ldots,1)}) \mod (m_2), i > 1, j > 1 \end{cases}$$
From this lemma, for $1 \leq k \leq l$ with a fixed d, the difference $(a_{d-1,\mu} - a_{d-1,\mu'})$ mod (m_2) depends on $x_{k,\mu} - x_{k,\mu'}$ with $\mu \neq \mu'$ for all $i \neq k$ where $\mu = (\mu_1, \ldots, \mu_l)$ and $\mu' = (\mu'_1, \ldots, \mu'_l)$. The d^l coefficient $a_{d-1,\mu}$ now can be expressed in $O(ld)$ variables. A similar idea can be applied for $a_{d-2,\mu}$ by defining a new parameter in which we express $a_{d-2,\mu}$ as
$$\begin{cases} f_0 & = & \frac{N - a_d \overline{m_0}^d}{\overline{m_0}^{d-1} m_2^2} \\ f_{i,j} & = & -\frac{d a_d x_{i,j}}{m_2^2} - \frac{e_{i,j}}{m_2} \end{cases}$$
where $1 \leq i \leq l$ and $1 \leq j \leq d$, also notices that the d^l $a_{d-2,\mu}$ coefficients generated

could be expressed again in $O(ld)$ variables from the estimates

$$\frac{a_{d-2,\mu}}{m_{1,\mu}} \approx f_0 + \sum_{i=1}^{l} f_{i,\mu_i}$$

However, this procedure does not optimize the other coefficients, for example a_0, a_1, a_2 in a polynomial of degree $d = 5$. Putting this together we could summarize as the following procedure:

Polynomial generation
Given N , degree d, l number of primes in m_2 and a bound B for these primes as inputs, the algorithm generates many polynomial with an optimized coefficients,

1. Compute $M \leq N^{\frac{1}{d+1}}$, and $c_{d,max} \leq \left(\frac{M^{2d-3}}{N}\right)^{\frac{1}{d-3}}$
2. For each a_d in the range of $a_{d,max}$
   (a) Construct m_2 by finding all $m_{2,i} < B$ prime such that $a_s x^d \equiv N \mod (m_{2,i})$ has d solutions.
   (b) Compute $m_0 = \left(\frac{N}{a_d}\right)^{\frac{1}{d}}$, $a_{d-1,max} = \frac{M^2}{m_0}$ and $a_{d-2,max} = \left(\frac{M^{2d-6}}{m_0^{d-4}}\right)^{\frac{1}{d-2}}$
   (c) For each subset of the primes $m_{2,i}$ of cardinality l , Compute $m_2 \prod_{i=0}^{l} m_{2,i}$ such that
   $$m_2 \leq a_{d-1,max}$$
   Compute $x_{i,j}$, $m_{i,j}$, $e_{i,j}$, f_0 and $f_{i,j}$.
   (d) Set $\epsilon = \frac{a_{d-2,max}}{m_0}$, identify μ such that $f_0 + \sum_{i=1}^{l} f_{i,\mu_i}$ lies in the $\epsilon-$ neighborhood of an integer, continue the base $(m_{1,\mu}, m_{2,\mu})$ and add the two polynomials F and G in the output list.

The output of this polynomial can pass in the size and root properties in the algorithm 4 given by Murphy and Montgomery. Originally [Kle06] was a preparation of [KAF+10] the RSA-768 bits challenge, in the same paper [Kle06] Kleinjung proposed an other variant for the polynomial of degree less than $d = 5$.

3.3.3 Sieving

The sieving part of the algorithm is the dominant step in the algorithm based on the sieve methods. For GNFS, we have an approximate attempt on the number of candidates expected which allows to perform the next step. According to the definition of the factor-base, the sieving over the two sides can be done in a similar way. Given two polynomial F and G we are searching for (a, b) such that $F(a, b)G(a, b)$ is smooth. The preferable procedure was described as the *line sieve*. It consists of sieving each candidates over the sieving region (rectangular) inline using a serial method or in parallel.

An algorithm based on the sieving method uses *line sieve* to handle this part, it sieves over a given area and marks the candidate which more likely to be smooth, try to factor it into its prime factors. This process is repeated until we found a sufficient candidates (depending on the size of the factor-base) for the next step.

The first version of GNFS uses the same procedure. This can be done in two steps, for each (a, b) relatively prime in the sieving area. First, we mark those which have small value after subtracting their polynomial value by the prime in the factor base (in practice, log is used to save memory). Second, we use probabilistic or a deterministic factorization algorithm (this is fast for a small number) on these marked candidates. From the study of the complexity we use trial division on the second steps, although we could use some probabilistic algorithm to factor or to test the primality of some large factor. This procedure can be formalized as follows.

Line sieve

1. We start from an empty two dimensional list C,
2. For each (a, b) relatively prime in $(-A\sqrt{s}, A\sqrt{s}) \times (1, \frac{B}{\sqrt{s}})$
   - $C[a, b] = \log(|G(a, b)|)$ or F
   - for each (p, r) in the factor base if $a \equiv rb \mod (p)$ then $C[a, b] \leftarrow C[a, b] - \log(p)$
3. For each (a, b) relatively prime in $(-A\sqrt{s}, A\sqrt{s}) \times (1, \frac{B}{\sqrt{s}})$
   - if $C[a, b]$ is small enough for some bound, factorize $G(a, b)$

During sieving, as suggested from the improvement of an algorithm based on sieving method such as continued fraction and quadratic sieve, one allows large prime in the factor-base. This makes sense for the use of a modified probabilistic primality test in the sub-step 3. However it is not allowed to be in the quadratic character factor-base.

This technique was improved for the GNFS by Pollard [Pol93b]. The idea is to consider these large prime referred as special prime, and the sieving is performed over the lattice (a, b) generated by this special prime. We give more detail about this improvement in chapter 4 which is the main contribution of the author of this thesis.

3.3.4 Linear algebra

After collecting a sufficient number of candidates from the sieving, a further step consists of finding a correspondences over these candidates. In fact we assume that we have a matrix of the exponents as described in algorithm 2 modulo 2, in which we want to compute its kernel. The Gaussian methods could be used to handle this kind of problems, although the choice of the optimized parameter studied in the running time of the GNFS, the Gaussian is not an option, since the sieving part is the dominant term in the complexity of the algorithm.

Given the matrix A we want to find a vector x such that $Ax = 0$. Using an elementary result from linear algebra, if we assume that A is square with determinant zero and f the characteristic polynomial of A, there exist a polynomial h such that $Xh(X) = f(X)$. Then by Cayley-Hamilton we can construct our x by means of $h(A)$. The purpose is not to compute the characteristic polynomial of A but try to construct a polynomial with the same property. Before describing the method used to tackle the linear algebra part in the algorithm, we give the following procedure called Berlekamp-Massey, originally used to find the minimal polynomial of linearly

Algorithm 5 Berlekamp-Massey

Input Linear sequence (s_i) for $0 \leq i < 2n$.
Output Polynomial generator of s_i

$u_0 \leftarrow x^{2n}$
$u_1 \leftarrow \sum_{i=0}^{2n-1} s_i x^i$
$v_0 \leftarrow 0$
$v_1 \leftarrow 1$
while $\deg(u_1 \geq 1)$ **do**
  $q, r \leftarrow \text{QuoRem}(u_0, u_1)$
  $u_0 \leftarrow u_1$
  $u_1 \leftarrow r$
  $tmp \leftarrow v_0 - qv_1$
  $v_0 \leftarrow v_1$
  $v_1 \leftarrow tmp$
end while
$d \leftarrow \max(\deg(v_1), \deg(u_1) + 1)$
$P \leftarrow x^d v_1(\frac{1}{x})$
Return Normal(P)

recurrent sequence, especially linear Feed Back Register.
The QuoRem is an Eucludian division between polynomials, the algorithm output a normalization of the coefficient of P, by dividing with its leading coefficient. To make use of the algorithm 5, we give more theory that will be used to describe the linear algebra part of GNFS.

Definition (**Krylov subspace** [Wik18]). Given a non singular matrix $A \in K^{n \times n}$ and $b \neq 0 \in K^n$ the order-r Krylov subspace $\mathcal{K}_r(A, b)$ generated by A and b is given by

$$\mathcal{K}_r(A, b) := \text{span}(b, Ab, \ldots, A^{r-1}b)$$

Our goal is to find the solution of the system $AX = b$. Using the Krylov subspace $\mathcal{K}_r(A, b)$, we construct the solution by a similar method inspired by the fixed point iteration. In fact, there exist a non trivial linear dependency relation between the first $(k+1)$ vectors of $\mathcal{K}_r(A, b)$ say

$$a_0 b + a_1 Ab + \ldots + a_k A^k b = 0$$

where $a_0, \ldots, a_d \in K$ with $k \leq r$ and $a_0 = 1$ then

$$b = -A(a_1 b + a_2 Ab + \ldots + a_k A^{k-1} b)$$

therefore the solution is

$$x = -(a_1 b + a_2 Ab + \ldots + a_k A^{k-1} b)$$

Wiedemann [Wie86] used a similar approach where we construct a generator f of the Krylov sequence such that

$$f(X) = a_0 + a_1 X + \ldots + a_d X^d$$

and define

$$f^*(X) = \frac{f(X) - f(0)}{X}$$

the solution is given by

$$x = -f^*(A)b$$

The algorithm uses Berlekamp-Massey algorithm. Unlike the linear dependency relation from the Krylov sequence we consider the generator of a sequence of scalar element in K. Let u be a random vector, and consider the sequence $\{(u, A^i b)\}_{i\geq 0}$ of the inner product of u with $A^i b$, if we assume that f is a polynomial generator of this sequence, since this is not necessarily minimal, let f_u the minimal polynomial generator, we have that $f_u | f$, by Berlekamp-Massey we can compute f_u by the first $2n$ element from the sequences, now suppose $f = f_u$ then we construct the solution of the system by

$$x = \sum_{i=1}^{d} f[i] A^{i-1} b$$

where $f[i]$ is the coefficient of f. The procedure can be described as follows

Algorithm 6 Wiedemann[Wie86]

Input Matrix A, vector b.
Output Vector x such that $Ax = b$.

1: $k \leftarrow 0$
2: $b_0 \leftarrow b$
3: $y_0 \leftarrow 0$
4: $d_0 \leftarrow 0$
5: **while** $b_k \neq 0$ **do**
6: $u_k \leftarrow$ random vector.
7: $f \leftarrow$ Berlekamp$(\{(u_{k+1}, A^i b)\})$ {$2(n - d_K)$ terms are enough}
8: $y_{k+1} \leftarrow y_k + f^*_{k+1}(A) b_k$
9: $b_{k+1} \leftarrow b_0 + A y_{k+1}$
10: $d_{k+1} = d_k + deg(f_{k+1})$
11: $k \leftarrow k + 1$
12: **end while**
13: $x = -y_{k+1}$

Remarks.

1. This algorithm is probabilistic, in [Wie86] it is proved that it will stop after three passes (while step 5-12) with probability at least 70%, although letting u_k to be the $k-$unit vector will turn the algorithm into deterministic.

2. In step 7, we only needs to compute $2(n - d_k)$, in fact, for $k = 1$ at b_1 we got f_1 with f_{u_1} its minimal polynomial such that $f_{u_1} | f_1$ and $\deg(f_1) = \deg(f) - \deg(f_1) \leq n - \deg(f_1)$ which actually means that we can run Berlekamp-Massey by only $2(n - \deg(f_1))$ elements of the sequences $\{(u_{k+1}, A^i b)\}$ to obtain f_{u_1}.

3. The equality

$$y_k = (f_k \dots f_1)^*(A) b$$

can be shown by induction, for $k = 1$ then $y_1 = f_1^*(A)b$, let suppose that its hold for k, and

$$\begin{aligned} y_{k+1} &= y_k + f_{k+1}^*(A)b_k \\ &= (f_k \dots f_1)^*(A)b + f_{k+1}^*(A)(f_k \dots f_1)(A)b \\ &= (f_k \dots f_1)^*(A)b + (f_{k+1}f_k \dots f_1)^*(A)b - (f_k \dots f_1)^*(A)b \\ &= (f_{k+1}f_k \dots f_1)^*(A)b \end{aligned}$$

4. The same for

$$b_0 + Ay_{k+1} = f_{k+1} \dots f_1(A)b$$

Indeed, for $k = 1$,

$$\begin{aligned} b_0 + Ay_1 &= b_0 + A(y_0 + f_1^*(A)b_0) \\ &= b_0 + Ay_0 + f_1(A)b_0 - b_0 \\ &= f_1(A)b \end{aligned}$$

by assuming that the equality holds for k, and we have for $k+1$,

$$\begin{aligned} b_0 + Ay_{k+1} &= b_0 + A(y_k + f_{k+1}^*(A)b_k) \\ &= b_0 + Ay_k + f_{k+1}(A)b_k - b_k \\ &= f_{k+1}(A)(b_0 + Ay_k) \\ &= f_{k+1}f_k \dots f_1(A)b \end{aligned}$$

Recall that the aim is to find a solution for $Ax = 0$. Most of the time, the matrix A is non-square and we consider the matrix to have a dimension $(n \times n+1)$. The vector b can be obtained by the last column. We can perform the algorithm 6.
The case where A is singular has been discussed in [Wie86]. It might happen that $a_0 = 0$, from the original idea using Krylov subspace, a solution can be deduced from $A(c_1 b + \dots + c_d A^{d-1}b) = 0$ otherwise the algorithm fails.

This means that there is some dependency between the rows or columns of the matrix. We could erase these columns or rows and reduce the problem in the remaining matrix. For GNFS in particular, we need to keep trace if the column is part of the factor base.
In practice, we are dealing with a huge *sparse*[2] matrix, which is very suitable for parallel environment. Coppersmith [Cop94] gave a parallel variant of the idea proposed by Wiedmann, called the *Block of Wiedmann*. Using a version of the Berlekamp-Massey procedure which can be used in a polynomial with matrix coefficient, before we give the procedure of Coppersmith, we state some details about the improvement.
The goal is to find a non zero solution w of $Aw = 0$. We begin with two random vectors u, z, and we set $y = Az$
For $0 \leq i \leq 2N$, we compute $a^{(i)} = u^T A^i y$ and let

$$a(X) = \sum_{i=0}^{2N} a^{(i)} X^i$$

[2] most of the element are zero.

a generator of the sequence $\left\{a^{(i)}\right\}_i$ can be deduced from Berlekamp-Massey, there are f and g such that

$$f(X)a(X) \equiv g(X) \mod (X^{2N+1}) \tag{3.3.2}$$

If $\hat{f}$ is the minimal polynomial of A then

$$\hat{f}^{rev}(X) = X^{\deg(\hat{f})}\hat{f}(\frac{1}{X})$$

Similar to the the original procedure of Weidmann, we construct $\hat{f}$ by the least common multiple of some f^{rev} and set

$$f^*(X) = \frac{\hat{f}(X)}{X^k}$$

for k the highest possible power of X. We apply a power of A in $f^*(A)z$ for any z until

$$A^i f^*(B)z = 0$$

then

$$w = B^{i-1}f^*(B)z$$

This procedure is based on the fact that we constructed a linear generator of $\{A^i y\}$ which is orthogonal to $\{u^T A^i\}$. Coppersmith suggested to view the Berlekamp-Massey algorithm as the extended Euclidian algorithm. Which means that we perform an Euclidian division between the polynomial $a(X)$ and X^{2N+1} until the polynomial f in (3.3.2) has a degree N. More precisely, we have

$$f(X)a(X) + g(X) = e(X)X^{2N+1}$$

with $\deg(e) = N - 1$ the process decreases $\deg(a)$ and increases f and g. Now if we replace X with $\frac{1}{X}$ and multiply it by X^{3N} the equality becomes

$$\underline{f}(X)a^{rev}(X) + \underline{g}(X)X^{2N} = \underline{e}(X)$$

Using the initial value $\deg(\underline{f}) = 0$ and $\deg(\underline{g}) = 0$, the algorithm output an $\underline{f}$ if $\deg(\underline{f}) = N$. This method can be used when the coefficients of the polynomial are matrices. Coppersmith idea supposes that the coefficients are vectors in F_2^n where n is the width of the block, applying the improvement of Wiedmann with a random matrix for u and z. A detail of this step is described in [Cop94]. The block of Wiedmann algorithm is again probabilistic, although it is very well suited for parallel environment. A detailed analysis of this method was given in [Kal95]. There are many improvement of this algorithm and other application in many field. For instance, the linear algebra in GNFS, Thomé [Tho02] gave an improvement of this technique, it was proposed to be a speed-up of Coppersmith method.
On the other side, we can work in the orthogonal subspace of the Krylov to solve the linear system. Lanczos introduced a procedure to solve a system $Ax = b$ in [Lan52] for a matrix symmetric square A in $\mathbb{K}^{N\times N}$ and $b \in \mathbb{K}^N$. The idea is to construct the

sequences $\{w_n\}_{n\geq 0}$ as follows

$$\begin{cases} w_0 = b \\ w_{n+1} = Aw_n - \sum_{i=0}^{n} \frac{(Aw_i)^T Aw_n}{(Aw_i)^T w_i} w_i \end{cases}$$

From this definition, $(Aw_i)^T w_j = 0$ for $i \neq j$, thus for $j < n-1$

$$\begin{aligned} (Aw_j)^T Aw_n &= \left(w_{j+1} + \sum_{i=0}^{j} \frac{(Aw_i)^T Aw_j}{(Aw_i)^T w_i} w_i \right) Aw_n \\ &= (Aw_{j+1})^T w_n + \sum_{i=0}^{j} c_i (Aw_i)^T w_n \\ &= 0 \end{aligned}$$

for a constant c_i, the computation of the sequences could be simplified as follows,

$$\begin{cases} w_0 = b \\ w_{n+1} = Aw_n - \frac{(Aw_n)^T Aw_n}{(Aw_n)^T w_n} w_n - \frac{(Aw_{n-1})^T Aw_n}{(Aw_{n-1})^T w_{n-1}} w_{n-1} \end{cases}$$

and stops when for $w_n = 0$. Indeed, if $n > N$, then $w_0, \ldots, w_n$ are linearly dependent, which means there are $a_i \neq 0$ such that

$$\sum_{i=0}^{n} a_i w_i = 0$$

so that

$$\sum_{i=0}^{n} a_i (Aw_n)^T w_i = 0$$

which implies $w_n = 0$. There were many variants of this technique, not only focusing in solving system but for several problem in linear algebra [CW85].

Now, if we assume that m is the smallest index which verify $w_{m-1} = 0$, we set

$$x = \frac{w_0^T b}{(Ab)^T b} b + \sum_{i=0}^{m-1} \frac{(w_i)^T b}{(Aw_i)^T w_i} w_i$$

so that

$$Ax = b + A \sum_{i=1}^{m-1} \frac{(w_i)^T b}{(Aw_i)^T w_i} w_i$$

and therefore $Ax - b \in \text{span}\{Aw_0, \ldots, Aw_{m-1}\}$ Since $(Aw_i)^T x = (Aw_i)^T b$ for $i \leq m-1$ we have

$$Ax - b = 0.$$

Again, we want to solve the system $Ax = 0$ for a rectangular matrix A. Similar idea as in Wiedmann modified algorithm could be used to transform this system into $Ax = b$. To make the matrix symmetric, we multiply it by its transposed. Originally, Lanczos algorithm was used with $\mathbb{K} = \mathbb{R}$, for the sieving based algorithm we work in $\mathbb{K} = \mathbb{F}_2$. The conversion into a symmetric matrix is not always working, since it

might be orthogonal with itself. To avoid this we can embed $\mathbb{F}_2$ into $\mathbb{F}_{2^k}$ for $2^k \geq N$. Also P. Montgomery [Mon95] pointed out that the expression $(Aw_i)^T w_i$ in the denominator can be equal to zero even if $w_i \neq 0$.

Coppersmith gave an improvement of Lanczos algorithm using block in [Cop93], the *Block of Lanczos* algorithm. There were many variant of a such technique, dedicated to a very large sparse matrix and for an arbitrary domain $\mathbb{K}$ [CW85].

In [Mon95] Montgomery inspired from these ideas, proposed an implementation of the block Lanczos algorithm. Instead of producing the sequences $\{w_i\}_{i\geq 0}$ in the original algorithm, one constructs a sequences of subspace $\{\mathcal{W}_i\}_{i\geq 0}^{m-1}$ of $\mathbb{K}^N$ which are pairwise $A-$orthogonal.
Indeed, given the matrix $B \in \mathcal{M}_{N_1\times N_2}(\mathbb{F}_2)$ from the sieving step, with $N_1 < N_2$, there are at least $N_2 - N_1$ linearly independent vectors $X \in \mathbb{F}_2^{N_2}$ which satisfy $BX = 0$, the improvement of Lanczos proposed by Montgomery needs symmetric matrix say $A = B^T B$ in $\mathcal{M}_{N\times N}(\mathbb{F}_2)$ for $N = N_2$. Let n_b be the size of block, we select a random matrix $Y \in \mathcal{M}_{N\times n_b}(\mathbb{F}_2)$ and by trying to find a matrix $X \in \mathcal{M}_{N\times n_b}(\mathbb{F}_2)$ such that $AX = AY$, therefore the column vectors of $X - Y$ are in the null space of A, and if $\text{rank}(A)$ is at least $\text{rank}(B) - n_b + 1$ a null space of B can be formed from the combination of the vector in the null space of A. To give a detail of this technique we start from the following definition.

Definition. A subspace $\mathcal{W} \subseteq \mathbb{K}^N$ is $A-$ invertible if it has a basis ω of column vector such that $\omega^T A\omega$ is invertible, the choice of the basis is arbitrary.

Let $u \in \mathbb{K}^N$ and $\mathcal{W} \subseteq \mathbb{K}^N$ is $A-$invertible. u can be written as $v_0 + v_1$ with $v_1 \in \mathcal{W}$ such that $v_1 = \dfrac{\omega}{\omega^T A\omega}\omega^T Au$ and $\mathcal{W}^T Av_0 = \{0\}$. The theory behind the construction for the sequence of subspace can be seen as follows

$$\begin{array}{ll} \mathcal{W}_i \ \ A-\text{invertible} & \\ \mathcal{W}_j^T A\mathcal{W}_i = \{0\}, & \text{for } i \neq j \\ A\mathcal{W} \subseteq \mathcal{W}, & \text{with } \mathcal{W} = \mathcal{W}_0 + \ldots + \mathcal{W}_{m-1} \end{array}$$

So given $b \in \mathcal{W}$, we construct $x \in \mathcal{W}$ such that $Ax = b$. In fact, we set $x = \sum_{j=0}^{m} v_{1j}$ for $v_{1j} \in \mathcal{W}_j$ with $Av_{1j} - b$ is orthogonal to all $\mathcal{W}_i$. By taking a ω_j basis for $\mathcal{W}_j$ we set

$$x = \sum_{j=0}^{m-1} \frac{\omega_j}{\omega_j^T A\omega_j}\omega_j^T b$$

Based on this, we give the sketch-procedure of the block Lanczos algorithm suggested by Montgomery [Mon95].
Let V_0 be a $N \times n_b$ a matrix, one defines the following recurrences

$$\begin{aligned} \omega_i &= V_i S_i \\ V_{i+1} &= A\omega_i S_i^T + V_i - \sum_{j=0}^{i} \omega_j C_{i+1,j} \end{aligned}$$

for $i \geq 0$, until $V_i^T AV_i = 0$ and $\mathcal{W}_i =< \omega_i >$.

- S_i is a $n_b \times n_i$ projection matrix such that $\omega_i^T A \omega_i$ is invertible which making $n_i \leq n_b$ as large as possible.
- The elements of S_i are all zero except for exactly one 1 per column and at most one 1 per row. This is done to ensure that $S_i^T S_i = I_{n_i}$ and $S_i S_i^T$ a sub-matrix of I_{n_b} reflecting the vector selected from V_i.
- $C_{i+1,j} = \dfrac{1}{\omega_j^T A \omega_j} \omega_j^T A (A \omega_i S_i^T + V_i)$.

Indeed we would construct the subspace $\mathcal{W}_i$ $A-$invertible by selecting the basis ω_i as many as the column of V_i. The $C_{i+1,j}$ is the constraint to ensure that $\omega_j^T A \omega_{i+1} = \{0\}$ as in the original idea.
Let m be the first index which makes $V_i^T A V_i = 0$, then the above procedure can be used to construct the sequence of subspace defined in the general idea, and $\omega_j^T A V_i = 0$ for $0 \leq j < i \leq m$ therefore $\mathcal{W}_j^T A \mathcal{W}_i = \{0\}$ for $i \neq j$.
To reduce the computation, we rewrite the expression to generate V_i. For $j < i$ we have that

$$\begin{aligned}
\omega_j^T A^2 \omega_i &= (S_j^T S_j)(\omega_j^T A^2 \omega_i) \\
&= S_j^T (A \omega_j S_j^T)^T A \omega_i \\
&= S_j^T (V_{j+1} - V_i + O(\omega_0 + \ldots + \omega_j))^T A \omega_i \\
&= S_j^T V_{i+1}^T A \omega_i - \omega_j^T A \omega_i \\
&= S_j^T V_{j+1}^T A \omega_i
\end{aligned}$$

and if $S_{j+1} = I_{n_b}$ then $\omega_{j+1} = V_{j+1}$ thus $\omega_j^T A^2 \omega_i = 0$, since the choice of S_i is free.
We have

$$V_{i+1} = A \omega_i S_i^T + V_i - \omega_i C_{i+1,i} - \omega_{i-1} C_{i+1,i-1} - \omega_i C_{i+1,i-2}$$

To make this valid for $j \leq i - 3$, V_{j+1} must be $A-$orthogonal to ω_{j+1} through ω_m. That is

$$< V_{j+1} > \subseteq \mathcal{W}_0 + \ldots + \mathcal{W}_{j+2}$$

for $j \geq -1$.
To simplify the notation, we set

$$\omega_i^{inv} = \frac{S_i}{\omega_i^T A \omega_i} S_i^T = \frac{S_i}{(V_i S_i)^T A V_i S_i} S_i^T$$

hence

$$V_{i+1} = A V_i S_i S_i^T + V_i - V_i \omega_i^{inv} V_i^T (A V_i S_i S_i^T + V_i) - V_{i-1} \omega_{i-1}^{inv} V_{i-1}^T A^2 V_i S_i S_i^T - V_{i-2} \omega_{i-2}^{inv} V_{i-2}^T A^2 V_i S_i S_i^T$$

Manipulating this expression,

$$V_{i+1} = A V_i S_i S_i^T + V_i D_{i+1} + V_{i-1} E_{i+1} + V_{i-2} F_{i+1}$$

for $i \geq 0$ with

$$\begin{aligned}
D_{i+1} &= I_{n_b} - \omega_i^{inv} (V_i^T A^2 V_i S_i S_i^T + V_i^T A V_i) \\
E_{i+1} &= -\omega_{i-1}^{inv} V_i A V_i S_i S_i^T \\
F_{i+1} &= -\omega_{i-2}^{inv} (I_{n_b} - V_{i-1}^T A V_{i-1} \omega_{i-1}^{inv}) (V_{i-1} A^2 V_{i-1} S_{i-1} S_{i-1}^T + V_{i-1}^T A V_{i-1}) S_i S_i^T
\end{aligned}$$

for $i < 0$, we define ω_j^{inv} and V_j to be zero and S_j to be I_{n_b}.
To use this procedure, we start by the initial value $V_0 = AY$ and construct the sequence of subspace until $V_i^T AV_i = 0$ at $i = 0$, computes

$$X = \sum_{i=0}^{m-1} \frac{\omega_i}{\omega_i^T A \omega_i} \omega_i^T V_0 = \sum_{i=0}^{m-1} V_i \omega_i^{inv} V_i^T V_0$$

By the property of the sequences of subspace generated, if we set $\mathcal{W} = \mathcal{W}_0 + \ldots + \mathcal{W}_{m-1}$ and $\mathcal{W}_m =< V_m >$ where $\mathcal{W}_m$ is A-orthogonal to $\mathcal{W}$ and to itself, then $AX - V_0 \in \mathcal{W} + \mathcal{W}_m$ and if $V_m = 0$ we have $AX = AY$.

Remarks.

1. Most of the time, the procedure stops when $V_m^T AV_m = 0$ with $V_m \neq 0$, so that the vector of the null space of A is not only the combination of $X - Y$. Indeed, V_m is A-orthogonal to ω_j for $j < m$ and we have that $A\mathcal{W}_m \subset \mathcal{W}_m$ such that ω_j is $S-$invertible for $j < m$, which makes $\omega_j = 0$ therefore $AV_m \in \mathcal{W}_m$. A vector which span the kernel of A can be obtained by the linear combination of $X - Y$ and V_m. Montgomery[Mon95] claimed that the total rank of $X - Y$ and V_m is at most $2n_b$ so that one can use Gaussian algorithm to find a such vector. To get back to B, one let Z to be a $N \times 2n_b$ matrix (the concatenation of $X - Y$ and V_m), and a matrix U of size at most $2n_b \times 2n_b$ such that $BZU = 0$, then a basis of ZU can be used for the output of the algorithm.

2. This procedure can be generalized as a generation of S_i and V_{i+1} or more precise ω_i^{inv}. Montgomery gave a strategy to generate this recursively at each iteration, it output $V_i^T AV_i$ and S_{i-1}, the methods is a similar as Gaussian pivot algorithm, from the property of S_i it constructs the diagonal for S_i and ω_i^{inv}.

The algorithm can be summarized as the following,

Algorithm 7 Block of Lanczos algorithm

Input $B \in \mathcal{M}_{N_1 \times N}(\mathbb{F}_2)$, with $N > N_1$.
Output Matrix kernel of B.

$A \leftarrow B^T B \in \mathcal{M}_{N \times N}$
$Y \leftarrow \text{random}(\mathcal{M}_{N \times n_b})$
$V_0 \leftarrow AY$
$T \leftarrow V_0 A^T V_0$
$S_0 \leftarrow I_{n_b}$
$i \leftarrow 1$
while $T \neq 0$ **do**
  $(\omega_i^{inv}, \text{diag}(S_i)) \leftarrow \text{Montgomery}(T, S_{i-1})$
  $\text{Compute}(V_i)$
  $T \leftarrow V_i^T A V_i$
end while
$m \leftarrow i$
$\text{Compute}(X)$
$Z \leftarrow X - Y||V_m$
$U \leftarrow \text{Gaussian}(BZ)$
Return ZU

Chronologically the Coppersmith Block of Lanczos was suggested one year before the Block of Wiedmann, the goal is the same, to be able to use the property of the large matrix of being sparse. In computer science, this has been studied so that the operations on such kind of data is faster. Working in $\mathbb{F}_2$ makes the use of block natural since we are dealing on bits operation. Both techniques are probabilistic and output n_b vectors of the null space of B.
The implementation [Mon95] of Montgomery with an improvement of the block of Lanczos can be used for linear algebra step in the number field sieve, although most of the literature in this area still utilizes the block of Wiedmann, this has a huge advantage in parallel and distributed environment. However, Thomé [Tho16] suggested a parallel version of the block of Lanczos.

3.3.5 The square roots algorithm

After finding a sufficient dependencies I from the linear algebra step, we have n_b probable solutions that we could use. On the algebraic side we have

$$\gamma = \left(\frac{F_X(\omega, a_n)}{a_n}\right)^2 \prod_{i \in I}(a_n a_i - \omega b_i)$$

and we would like to compute its square root. In [BLP92] the author suggested a method using factorization of polynomial over the number field. Choosing an odd prime q such that F reduced modulo q is still irreducible (inert prime), so that one can compute a δ_0 such that $\delta_0^2 \gamma \equiv 1 \mod (q)$ where q is the ideal $q\mathbb{Z}[\omega]$, (we assume it, otherwise we must add more ideals in the quadratic factor base). By giving a bound estimation of the coefficient of this square root as polynomial in ω say B. The lifting root method by Newton [Lip76] iteration gives

$$\delta_i = \frac{\delta_{i-1}(3 - \delta_{i-2}^2 \gamma)}{2} \mod (q^{2^i})$$

We observe that the coefficient of this element as polynomial in ω is bounded by $\frac{q^{2i}}{2}$ so that the algorithm stops when q^{2i} is twice larger than B, by hoping that we reach the coefficient in $\mathbb{Z}[\omega]$.
This technique is known as p-adic approximation, often used in *computational algebra*. A detail description of the application of such strategy could be found in many literature for instance([GCL92] chap 6).

The above method is very efficient, the only drawback is the size of the number we are attempting to compute. In fact, from the form of γ, the cardinality of the index interval I is roughly the same as the complexity of the algorithm. This concerns every step in the procedure, the computation of γ, the factorization of $x - \gamma$ and the lifting operation over a huge size number.

In [Cou93], Couveignes suggested an improvement of this method to avoid the problem of size, more precisely his procedure consider a several inert prime modulo q_i, and construct the solution by using CRT[3]. As well as the first method, the existence of such q_i inert is probabilistic. On the bright side the complexity seems to be improved and the algorithm can be done in parallel. Indeed, the running time is in the order of the size of $q_i^{k_i}$ (which is determined by the bound of the coefficient of the square root as polynomial in γ). The method requires a square root for every $q_i^{k_i}$ and to be sure that it has exactly two different positive and negative roots, also we need the further assumption that the degree of the number field must be odd.

Thomé [Tho12] gave a variant of this technique. It is an improvement of the two suggested methods above. The procedure works for any degree number field and the existence of inert prime is no longer required. It could be considered as a combination of the two methods, using lifting and CRT. Without loss of generality we consider f to be a monic polynomial and using the same notation as in [Tho12]. Let α be an algebraic integer, and σ_i is the embedding of the $Q[\alpha]$ for $i = 1, \ldots, d$. We have $S(\alpha)$ and we aim to compute its square root $T(\alpha)$ and

$$\log |\sigma_i(T(\alpha))| = \frac{1}{2} \log |\sigma_i(S(\alpha))|$$

can be used to compute the bound over the coefficient of $T(\alpha)$ since for $i = 1, \ldots, d$

$$\log |\sigma_i(S(\alpha))| = \sum_{j \in J} \log |\sigma_i(a_j - b_j \alpha)|$$

If we assume that a such bound can be computed, say M, one can give a precision for the lifting stop steps by setting $k = \lceil \log_p M \rceil$ for p^k modulo.

Thomé new CRT-based lifting strategy
Assuming the above hypothesis, let $\mathcal{P}$ be a set of l primes which totally splits f, set $P = \prod_{p \in \mathcal{P}} p$ and $\lambda = \lceil \frac{\log(M/\epsilon)}{\log P} \rceil$ where $\epsilon \leq 1$ and so $M < \epsilon P^\lambda$ we denote $B = \frac{\lambda}{l} \log_2 P$ the bit-size of p^λ for each $p \in \mathcal{P}$.
For each p_i one has $r_{i,j}$ roots for $j = 1, \ldots, d$ and by lifting that root modulo p_i^λ we obtain $\bar{r}_{i,j}$. We next compute the p_i-adic lift of $\sqrt{S(\bar{r}_{i,j})}$ with precision λ (as in the first suggested method), and say $T'_{i,j}$ be the result after lifting. So that we have

[3]Chinese Remainder Theorem.

$T'_{i,j} \equiv s_{i,j} T_{i,j} \mod (p_i^\lambda)$ where $s_{i,j}$ is the sign of the square root. A CRT-similar is used to obtain $T(x)$. Indeed, let ,

$$Q_i := \left(\frac{P}{p_i}\right)^\lambda = \prod_{p \in P,\, p \neq p_i} p^\lambda$$

and

$$H_{i,j}(x) := \frac{f(x)}{(x - \overline{r}_{i,j})} = \prod_{j' \neq j} (x - \overline{r}_{i,j})$$

so then

$$T(x) = \left(\sum_{i,j} Q_i H_{i,j}(x) T_{i,j} \frac{1}{Q_i f'(\overline{r}_{i,j})} \right) \mod (P^\lambda)$$

Remarks.

- The algorithm uses $l = t \times r$ so that one can partition $\mathcal{P}$ into subset of r element.
- The introduction of the parameter ϵ interacts directly on the sign $s_{i,j}$. Let consider the last expression constructed $T(x)$, we shall analyze its coefficients. Without loss of generality we could consider the leading coefficient and one will deduce a generalization for all, one has

  $$[x^{d-1}]T(x) := \sum_{i,j} Q_i T_{i,j} \frac{1}{Q_i f'(\overline{r}_{i,j})} \mod (P^\lambda)$$

  then

  $$\frac{1}{P^\lambda}[x^{d-1}]T(x) := \sum_{i,j} \frac{1}{p_i^\lambda} \left(T_{i,j} \frac{1}{Q_i f'(\overline{r}_{i,j})} \mod (p_i^\lambda) \right) \mod (1)$$

  we set the two real number

  $$\begin{aligned} x_{i,j} &= \frac{1}{p_i^\lambda} \left(T_{i,j} \frac{1}{Q_i f'(\overline{r}_{i,j})} \mod (p_i^\lambda) \right) \\ y_{i,j} &= \frac{1}{p_i^\lambda} \left(T'_{i,j} \frac{1}{Q_i f'(\overline{r}_{i,j})} \mod (p_i^\lambda) \right) \end{aligned}$$

  where $x_{i,j}$ and $y_{i,j}$ are in $[0,1]$ and $y_{i,j} \equiv \pm x_{i,j} \mod (1)$, the choice of $\epsilon \leq 1$ is arbitrary and $|\frac{1}{P^\lambda} T(x)| \leq MP^{-\lambda} \leq \epsilon$ then $\sum_{i,j} x_{i,j}$ and $\sum_{i,j} s_{i,j} y_{i,j}$ are in $[-\epsilon, \epsilon] + \mathbb{Z}$ which rise to a combinatorial problem. In [Tho12], the author claimed that for a modest number of primes this could be solved in practice.
  Assume now that $s_{i,j}$ are computed, so the closer integer to $\sum_{i,j} x_{i,j}$ say e_{d-1} and $T_{i,j} = s_{i,j} T'_{i,j}$ therefore

  $$\frac{1}{P^\lambda}[x^{d-1}]T(x) := \sum_{i,j} \frac{1}{p_i^\lambda} \left(T'_{i,j} \frac{1}{Q_i f'(\overline{r}_{i,j})} \mod (p_i^\lambda) \right) - e_{d-1}$$

  and we could generalize the procedure by defining the following,

$$c_{i,j,k} = [x^k]\left(T'_{i,j}\frac{H_{i,j}(x)}{Q_i f'(\bar{r}_{i,j})} \mod (p_i^\lambda)\right)$$

$$c^*_{i,j,k} = s_{i,j}c_{i,j,k}$$

Thus

$$T(x) = \sum_{i,j,k} x^k\left(Q_i s_{i,j} c_{i,j,k} - e_k P^\lambda\right)$$

and use $x_{i,j} = \frac{c^*_{i,j,k}}{p_i^\lambda}$ and $y_{i,j} = \frac{c_{i,j,k}}{p_i^\lambda}$ to compute $s_{i,j}$.

This procedure could be seen as follows, using the same hypothesis described in [Tho12] where f is monic.

Thomé Algorithm

Input: f monic irreducible, N, m and the indexes I.

Output: $T(m) \mod (N)$

Parameter: $l = t \times r$

1. Choose a set $\mathcal{P}_1, \ldots, \mathcal{P}_t$ of r primes (totally splits) and partition I into t disjoints subsets $I_1, \ldots, I_t$
2. For $k = 1, \ldots, t$
   - Computes λ and M
   - $S_k(x) = \prod_{i \in I_k}(a_i - xb_i)$
3. For $i = 1, \ldots, l$
   - Computes p_i^λ, Q_i, $\frac{1}{Q_i} \mod p_i^\lambda$

   3-1. For $j = 1, \ldots, d$
   - Computes $r_{i,j}$ and its lifted $\bar{r}_{i,j}$
   - $\frac{H_{i,j}(x)}{f'(\bar{r}_{i,j})}$
4. For $\tau = 1, \ldots, t$
   - Computes $S_\sigma(x) \mod \mathcal{P}_\tau$ for $\sigma = 1, \ldots, t$

   4-1. For p_i in $\mathcal{P}_\tau$
   - For $j = 1, \ldots, d$
     - one computes $S_\sigma(\bar{r}_{i,j}) \mod p_i^\lambda$ for $\sigma = 1, \ldots, t$
     - the product $S(\bar{r}_{i,j})$ and $T'_{i,j}$
     - $c_{i,j,k}$, $\frac{c_{i,j,k}}{p_i^\lambda}$ and $c_{i,j,k}m^k \mod (N)$ for $k = 0, \ldots, d-1$
1. Combinatorial to find $s_{i,j}$ and e_k
2. return $\sum_{i,j,k} Q_i s_{i,j} c_{i,j,k} m^k - e_k P^\lambda \mod (N)$

3.4 Notes

So far, we have given the main steps for the GNFS, with some of their improvement. However in practice, after sieving and decompose each candidate into its prime factor, we notice that some candidate can be sieved more than once especially when using lattice sieve. Also, the sieving step does not guaranty that each element of the factor base will appear in all the candidates (it may appear only in one candidate). Of course, we need to take care of these problems before feeding the data into the linear algebra step. Such a procedure is very relevant, for example when we do not need a trivial vector in the kernel.

Filtering. This step is applied before the linear algebra. In fact, it does more than taking care of the above restriction, it reduces the size of the matrix much smaller in which the sparse property is held.
This procedure can be sketched as, first, it eliminates all the candidate with the problem mentioned above (duplicates, ...), after reducing the exponent matrix into $\mathbb{F}_2$, it eliminates all the column zeros and the same for zero rows (free relation). A more detailed description of this method can be found in [Cav00].

3.5 Running time analysis

Definition. Let $Fb = \{p_1, \cdots, p_m\}$ be a set of prime (preferably small), Fb is called the ***factor-base***. We say that an integer x is smooth over Fb if all of the primes which divide x are in Fb. We say that an integer x is B-smooth for a positive integer B if all its primes factor are less than B, in this case, the factor-base is given by

$$Fb(B) = \{p \text{ prime } : p < B\}$$

We mentioned that the factorization algorithm based on the sieving is probabilistic. For instance, the Fermat difference of square method applied in a naive way relies upon the distribution of the square on the interval $[1, N]$, and the running time approximates the effort needed to find the couple (x, y). This gives an explicit running time for the Lehmer [Leh28] suggestion since we have approximately $\sqrt{N}$ of squares in the interval $[1, N]$. The use of *smooth* number was introduced in [AMB75], it did give an improvement of the complexity of the algorithm.

Definition. Let B be a positive integer, for a given X positive integer we define $\psi(X, B)$ be the number of $B-$smooth integer in the interval $[1, X]$, that is

$$\psi(X, B) = \#\{n \, : \, 1 \leq n \leq X \, , \, n \text{ is } B\text{-smooth}\}$$

This function has a very important implication in algorithmic number theory. We can deduce from this that the probability of random integers taken from $[1, X]$ to be B-smooth is equal to

$$\frac{\psi(X, B)}{X}$$

Karl Dickman studied this function, in [Dic30]. He showed the existence of a positive function $\rho(u)$ named after him "*Dickman function*" which satisfy

$$\frac{\psi(x, x^{\frac{1}{u}})}{x} \approx \rho(u)$$

where x tends to infinity and u bounded.
More precisely, $\psi(x, x^{\frac{1}{u}}) = \rho(u)x + O(\frac{x}{\log(x)})$ for any $U \geq 0$ s.t $0 \leq u \leq U$ and all $x > 2$, $\rho(u)$ has the following property on $(0, \infty)$,

$$\begin{cases} u\rho'(u) = -\rho(u-1) & u > 1 \\ 1 & \text{o/w} \end{cases}$$

In particular, ρ tends to zero rapidly as u goes to infinity.

Further results about ρ can be found in [Bru51],[Dic30]. In fact, Bruijn [Bru51] got an asymptotic approximation on ρ

$$\rho(u) = \exp\left(-u\left\{\log(u) + \log\log(u) - 1 - \frac{1}{\log(u)} + \frac{\log\log(u)}{\log(u)} + O(\frac{(\log\log(u))^2}{(\log(u))^2})\right\}\right)$$

where $x \longrightarrow \infty$ and $u \longrightarrow \infty$.
In [CEP83] a new approximation obtained by Canfield, Paul Erdös and Carl Pomerance is given:

$$\rho(u) = u^{-u(1+o(1))}$$

where $x \longrightarrow \infty$ and $u \longrightarrow \infty$.

An algorithm complexity is usually expressed in term of **Big** O**-notation**, it gives an approximation of the time spent by the algorithm in the worst case. In [Pom82], Carl Pomerance used a different notation known as L**-notation** to express the running time of the quadratic sieve algorithm. It became the reference notation for the new factorization algorithm based on the sieve technique. The L-notation has a very good behavior by carefully choosing its variable parameters. A new promising on the computational side since Factorization was considered to be a hard problem.

Definition (L-**Notation**). Let $0 \leq a \leq 1$, and $c \in \mathbb{R}_{>0}$. The L-function for the parameters a and c is defined by

$$L_N(a, c) = exp[c\log(N)^a(\log\log(N))^{1-a}]$$

In our purpose, N is the number we want to factor. Notices for $a = 0$,

$$L_N(0, c) = (\log(N))^c$$

is equivalent to a *polynomial* running time, and for $a = 1$ we have

$$L_N(1, c) = N^c$$

equivalents to an *exponential* running time. The case $L_N(a, c)$ *subexponential* when $0 < a < 1$ is between the above two classes.

Theorem 3.5.1 ([BLP92]). *Suppose g is a function defined for all $y \geq 2$ which satisfies $g(y) \geq 1$ and $g(y) = y^{1+o(1)}$ for $y \longrightarrow \infty$. Then as $x \longrightarrow \infty$*

$$\frac{xg(y)}{\psi(x,y)} \geq L_x\left[\frac{1}{2}, \sqrt{2} + o(1)\right]$$

uniformly for for all $y \geq 2$.
In particular, for $x \longrightarrow \infty$

$$\frac{xg(y)}{\psi(x,y)} = L_x\left[\frac{1}{2}, \sqrt{2} + o(1)\right]$$

if and only if

$$y = L_x\left[\frac{1}{2}, \frac{\sqrt{2}}{2} + o(1)\right]$$

The above theorem (3.5.1) gives the lower bound for the running time of the sieving steps. Indeed, the left hand side is the number of efforts we need to make on the interval $[1, x]$ to obtain a sufficient y-smooth numbers. The function g can be seen as the order of the number of operations we perform to each of the smooth candidates to obtain its prime decomposition.

3.5.2 The Complexity of the GNFS

Here we consider the setup for the GNFS algorithm. We chose a unique bound on the two factor-base, say y. Often the bound of the rational is smaller than the algebraic side so that here we consider y to be the maximum. We sieve on a two dimensional area defined by (a, b) relatively prime, such that $|a| < u$ and $0 < b < u$ with a positive integer u. The asymptotic running time of the algorithm has been conjectured by Pomerance [BLP92], it uses the following lemmas 3.5.3 and 3.5.4 with the approximation from theorem 3.5.1.

Lemma 3.5.3. For a real number $k \geq e$ and $l \geq 1$. we set $v = v(k, l)$ such that

$$\frac{v^2}{\log(v)} = kv + l$$

for $v \geq e$, then we have

$$2v = (1 + o(1))\left(k\log(k) + \sqrt{(k\log(k))^2 + 2l\log(l)}\right)$$

as $k + l$ goes to infinity.

Lemma 3.5.4. For each pair of positive integer n, d satisfying $n > d^{2d^2} > 1$, Let consider the real number $u = u(n,d) \geq 2$, $y = y(n,d) \geq 2$ and $x = x(n,d) = 2dn^{2d}u^{d+1}$ which satisfies

$$\frac{u^2\psi(x,y)}{x} \geq g(y)$$

for $g(y) \geq 1$ and $g(y) = y^{1+o(1)}$ as $y \to \infty$ then

$$2\log(u) \geq (1 + o(1))\left(d\log(d) + \sqrt{(d\log(d))^2 + 4\log(n^{1/d})\log\log(n^{1/d})}\right)$$

for $n \to \infty$ uniformly in d.

Notices that the last lemma 3.5.4 is equivalent to the theorem in 3.5.1, the square in u is due to the fact that we sieve on a two-dimensional area. This rises to a lower bound for the running time of the algorithm.

Conjectured running time of the GNFS[BLP92]
For any integer input $N > 256$, the running time of the general number field sieve is given asymptotically by

$$L_N\left[\frac{1}{3}, \left(\frac{64}{9}\right)^{\frac{1}{3}} + o(1)\right]$$

for $N \to \infty$ with

$$u = y = L_N\left[\frac{1}{3}, \left(\frac{8}{9}\right)^{\frac{1}{3}} + o(1)\right]$$

and

$$d = \left(3^{\frac{1}{3}} + o(1)\right)\left(\frac{\log(n)}{\log\log(n)}^{\frac{1}{3}}\right)$$

is the degree which minimize the running time such that $N > d^{2d^2} > 1$.
In here, one tries to find $g(y)$ which satisfy the condition in lemma 3.5.4. The degree d is the value which minimizes the complexity, which means that the lower bound is reached. A recent result which describes more details about this expression can be found in [LV18].
However, the above given asymptotic complexity is the reason which makes the General Number Field Sieve(GNFS) to be the best-known factorization algorithm .

4. The lattice sieve

In Quadratic Sieve [Pom85], the technique of sieving has been improved. The idea is inspired by Eratosthenes technique, but applied over the roots of the quadratic polynomial. This step was originally implemented as the traditional line sieve to perform the sieve at each element in the sieve interval. As we have mentioned, this step can be implemented differently by using data parallelism. Indeed, we broadcast the elements of the factor-base over the available nodes and each node can handle independently each element over the sieving area.
On the other hand, the General Number Field Sieve [BLP92] benefits from the form of its two sides factor-base. Each element of the factor-base can be expressed as a lattice point generator, finding a short basis which can be used as a pattern to construct all the elements of the lattice. This technique was first proposed by Pollard [Pol93b], and used to address the sieving step of the NFS algorithm. In this chapter, we give a theoretical description followed by the practical discussion of the lattice sieve.

4.1 Theoretical description

4.1.1 General idea

Pollard [Pol93b] first used the lattice sieve technique on the NFS to factor the seventh Fermat number. The sieving technique has been later used in the GNFS algorithm. The main goal is finding sufficiently many pair of candidate coprime (a, b) which satisfy

1. $a - bm$ smooth.
2. $N(a - b\alpha)$ smooth.

For this purpose, we give the following setup over the factor-base, inspired from [DHS85]. Let B_0, B_1 be two positive integers such that $\frac{B_0}{B_1} \in [0.1, 0.5]$, and let B be a positive bound of the factor-base as $Fb(B)$. We split $Fb(B)$ as follows:

$$\begin{array}{ccc} S & \text{small prime} & p \leq B_0 \\ M & \text{medium prime} & B_0 \leq p \leq B_1 \\ L & \text{large prime} & B_1 \leq p \leq B \end{array}$$

A prime in M is often called a special prime. The lattice sieve strategy can be described as follows:

1. Choose a region $\mathcal{R}$ of the (a, b) candidates to be sieved.
2. Choose a fixed prime q in M, and sieve only those (a, b) with

$$a - bm \equiv 0 \mod (q)$$

we sieve those as

(a) Sieve the numbers $a - bm$ with the primes $p < q$ including $p \in S$.
(b) Sieve the numbers $N(a - b\alpha)$ with the set which have more elements than the rational side (due to the size).

For the both sides factor-base rational and algebraic, one or few large primes up to B are allowed. In general we do not need all of the elements in the region $\mathcal{R}$ to be sieved on the set L since we tried to keep the sieving region as small as the G and F values.
It was pointed in [Pol93b] that the number of element sieved is reduced with respect to the traditional line sieve used in NFS. Precisely by a factor

$$\sum_{q \in M} \frac{1}{q} \approx \frac{\log(B_2/B_1)}{\log(B_1)}.$$

4.1.2 Sieving procedure

The techniques used to sieve over the rational and algebraic sides are similar according to the definition of the two factor-base, without loss of generality. Let $q \in M$ and (a, b) in $\mathcal{R}$ such that

$$a - br_q \equiv 0 \mod (q) \tag{4.1.1}$$

A trivial solution of this equation is $(r_q, 1)$ and $(q, 0)$, thus any point (a_i, b_i) in the lattice

$$(r_q, 1)\mathbb{Z}^2 \oplus (q, 0)\mathbb{Z}^2 \tag{4.1.2}$$

satisfies the equation (4.1.1).
The *lattice sieve* technique starts by looking for a short basis which generates the defined lattice in (4.1.2) (notice that finding a short basis for a 2-dimensional lattice is equivalent to an extended Euclidean algorithm on r_q, q). Let us assume that $U = (u_1, u_2)$ and $V = (v_1, v_2)$ are a short basis, we sieve over the elements (a_i, b_i) in the intersection of this lattice with $\mathcal{R}$, that is:

$$\begin{aligned} a_i &= cu_1 + dv_1 \\ b_i &= cu_2 + dv_2 \end{aligned}$$

where c, d determine the new index of the sieving area. In fact, the sieve is done on (c, d) coprime to conserve the hypothesis from the description of GNFS. Formally, we want $a_i - r_p b_i \equiv 0 \mod (p)$ for each (a_i, b_i) where (r_p, p) is in the factor-base, thus

$$c(u_1 - r_p u_2) + d(v_1 - r_p v_2) \equiv 0 \mod (p)$$

Now let C and D be two positive integers, and consider the rectangle $\mathcal{R} = [-C, C] \times [0, D]$ the sieve region. In [Pol93b] the author used this setup to avoid negative candidate by changing the sign of a given point from the Lattice.
For (r_p, p) element of the factor base, set

$$\omega_1 = u_1 - r_p u_2$$

and

$$\omega_2 = v_1 - r_p v_2$$

If $\omega_1 \equiv 0 \mod (p)$ then we need to check for the whole row with p (respectively if $\omega_2 \equiv 0 \mod (p)$).
When $(\omega_1, p) = 1$ the following two methods could be used during the sieve:

> *Sieving by rows.* This method could be seen as, at each row (fixed c in $L_{(r_q,q)}$) checking for $c\omega_1 + d\omega_2 \equiv 0 \mod (p)$. This is good for small primes but bad for large primes.

Sieving by vectors. This method is based on the fact that taking a couple (c, d) we construct a sub-lattice $L(r_p, p)$ of $L(r_q, q)$, where $p < q$. This is where the plane (c, d) intersects with the lattice $L(r_p, p)$. This lattice can be written as

$$\left(\frac{v_1 - r_p v_2}{r_p u_2 - u_1} \mod (p), 1\right) \mathbb{Z}^2 \oplus (p, 0)\mathbb{Z}^2$$

and one computes again the short basis of this lattice.

In [Pol93b], to factor the seventh Fermat number[1], the lattice sieve was used where both methods succeed. However, in practice sieving by vectors requires less memory than the sieving by rows as we will better explain in the next section. Therefore, a first criterion for choosing between the two methods is the amount of memory available on the hardware.

4.2 Practical description

4.2.1 Current state of the art

A practical use of lattice sieve can be found in this implementation of GNFS [BL93], where an analysis of the time and space used by the lattice sieve is also given. In their implementation, *sieving by rows* was used with a parallel variant. In [GLM94], *sieving by vectors* was used with a parallel version, their suggestion was a work in progress of an improvement for the sieving part of the GNFS. Moreover these two implementations were designed for MIMD system. In [FK], a new variant of the lattice sieve was presented with a parallel suggestion of the strategy, this can be seen as an improvement of the *sieving by rows*.

4.2.2 Motivation

We present an implementation of the *lattice sieve* on a SIMD system. In particular, we exploit the performance of modern GPU (Graphic Processing Units) to face the sieving part of GNFS. The two versions of the *lattice sieve* are very suitable for parallel environment, in fact they can be illustrated as in the two algorithms 8 and 9.

Algorithm 8 Sieving by rows

```
for q ∈ M do
    Computes a short basis for L_q
    for (a, b) ∈ L_q do
        for p ∈ S do
            Update location [a, b]
        end for
    end for
end for
```

In algorithm 9, we assume that we can split the set of small primes into SL (large in the small) and SS (small in the small).
We observe that in algorithm 9, the splitting of the small prime is a reduction of the number of ideals we used to sieve for every lattice point. This confirms the fact that the sieving by vector is used when the system does not have sufficient memory to

[1] $F_n = 2^{2^n} + 1$

Algorithm 9 Sieving by vectors

```
for q ∈ M do
  Computes a short basis for L_q
  for p ∈ SL do
    Compute a sort basis for L_pq
    for (a, b) ∈ L_pq do
      for p ∈ SS do
        Update location [a, b]
      end for
    end for
  end for
end for
```

fit these small prime ideals. In our attempt we give a description of the two sieving by row and vector trying to figure out their differences and limitation with respect to the hardware.

4.2.3 Observations

For the following we consider a original square sieving area $[-u/2, u/2] \times [1, u]$.

- ***Parallel line sieve***
  The steps preformed by the normal line sieve can be approximated by

  $$T = Sint \times A \times f$$

  where $Sint$ = # sieve interval, f = # factor-base and A is the average of steps performed to check the congruence and updating the location at each candidate coordinate. Indeed, for each candidate we need to perform $A \times f$ steps. A parallel implementation on a m processors system can split the sieve interval in m piece in which we gain a factor of m on the speed-up of the algorithm.

- ***Sieving by rows***
  Algorithm 8 describes the serial version of the sieve by rows. The number of steps performed by this procedure is roughly given by

  $$T = Mp \times e \times Sint \times Sp \times A$$

  where $Mp = \#M$, e is the average cost of the lattice reduction, A is the average number of steps performed to check the congruence and update the location at candidate coordinates, $Sp = \#S$ and $Sint$ the sieve interval which is not the same as for the line sieve, Razvan in [Bar16] showed that this can be estimated as $2^i \times 2^i$ where $u = 2^i \times q^{\frac{1}{2}}$. Notice that e is the same as the cost of an extended Euclidean algorithm.
  In our proposal, we split the sieve interval into pieces (with respect to the available amount of memory). Each piece can be referred as run, which means that each run performs a part of the sieve interval (we assume to work on a single GPU). For each run, one element in the sieve intervals handled by one thread, and one element in M is handled by a group of threads (thread-blocks). For this parallel purpose the factor Mp in T disappears and $Sint$ is reduced by factor of the number of runs (or the number of GPUs since in a multi-GPU system each run can be performed independently). Apparently we gain speed-up

using this suggested parallel sieve by rows even though it misses some candidates found by the line sieve, those are the smooth number with respect to the small factor-base.

- ***Sieving by vectors***
  Algorithm 9 describes the serial version of the sieve by vectors. An obvious difference between the two lattice sieves is that here the procedure tries to break the factor Sp and hence the $Sint$ since we now work on a lattice point of an intersection L_{pq}. In fact, we can consider a similar analysis as the sieve by rows,
  $$T = Mp \times e_1 \times Sl \times e_2 \times Sint \times Ss \times A$$
  Where $Mp = \#M$, e_i is the cost of the lattice reduction, $Sl = \#SL$, $Sint$ the sieving area and $Ss = \#SS$. In our proposal, Mp and Sl disappear however, we have one additional factor: the cost of the reduction for the lattice L_{pq}. Notice that the sieve by vectors miss more candidates than the sieve by rows.

4.2.4 Configuration and implementation details

Note. Here, we followed the setup used in the cado-nfs implementation from [Tea17] since we use this software to generate the polynomial and the factor-base. According to the sieve properties of the GNFS, we sieve over (a, b) such that $F(a, b)$ is smooth. That is, let p be a prime such that $F(a, b) \equiv 0 \mod (p)$, we have the two following cases in the factor-base:

case 1: $f(\frac{a}{b}) \equiv 0 \mod (p)$ with $b \not\equiv 0 \mod (p)$. The root of $F(X, Y)$ in a projective notation is $(a : b) \in \mathbb{P}^1(\mathbb{F}_p)$. This is referred as the root of the dehomogenized polynomial f of F in the factor-base and represented as (r, p) i.e,
$$a - br \equiv \mod (p)$$
.

case 2: $F(a, b) \equiv 0 \mod (p)$ with $b \equiv 0 \mod (p)$. This root is written as $(1 : 0)$ in a projective notation (the point at infinity). This is referred as the projective root in the factor-base and represented as $(v + p, p)$ i.e,
$$F(1, b) \equiv 0 \mod (p)$$
for $b \equiv v \mod (p)$.

We use the outputs from cado-nfs and feed them to our lattice sieve implementation, as well as the cado-nfs parameters since these are optimized from the polynomial selection (skewness, bounds on the factor base, etc).
We present in Appendix A.1 the properties of the Tesla P100 GPU we used. Such device memory architecture is important since the difference with respect its level between the number of cycles has a huge interaction in the algorithm and should be set carefully.

In the experiment, we have an implementation of the Lattice sieve for 8 bytes signed integer (int64). For this we divided the factor-base into two sets, the compatible factor-base (those which can be fitted in a 64 bits) and the large factor-base. From now one, we refer to the compatible factor-base Fb.

Let q_{min} and q_{range} the parameter generated by cado-nfs. We split the factor-base Fb into $S_q = \{(r, p) : p < q_{min}\}$, the *small* factor-base and $M = Fb \setminus S_p$. We split M into q_{range} pieces and serialize each slice M_p as mentioned in the above description, however the q_{range} can be modified depending on the properties of the device.

Sieving by rows

Our implementation of the sieve by row (Algorithm 8) consists of two steps: it computes the short basis of the lattice generated by each element in the medium prime (M_p) interval, followed by the sieve over the small prime S_p. This could be detailed as follows:

1. Compute a short basis for each two dimensional lattices generated by the elements (r_q, q) of M. That is, the lattices $(r_q, 1)\mathbb{Z}^2 \oplus (q, 0)\mathbb{Z}^2$. This step can be done by using Lagrange's method which is similar to the extended Euclidean algorithm.

2. For each lattice, for a given sieving interval we perform the sieve over the small prime S_q.

In 1. we set each lattice on one thread. We implemented an optimized lattice reduction which computes the short basis on the GPU. In our experiments we design this to work with double precision. This is implemented in one kernel. The drawback for this kernel is one may run out of register memory in one multiprocessor which may slow down the code.
In 2. we are given the sieving interval (two dimensional array depending on a offset as the entire interval can not be fitted on the memory), we set the dimension of the thread-block equal to that size so that the index of each thread corresponds to the index of the sieve interval, each block handles one lattice. Note that it is not the same interval for each lattice, and the sieving interval is considered for all case in which the coprime test is done during the computation of the polynomial value after the sieve.

Sieving by vectors

In *sieving by rows*, we sieve for each element in the small prime interval. We tried to fit the small factor-base (root on constant, prime on texture) on the cached memory. This later is limited depending on the GPU we then have to perform the sieve by vectors.
We implemented the sieve by vectors within the sieve by rows, that is the code runs on the sieve by rows unless the number of small prime does not fit on the cached memory. It contains two kernels as above but in this case we run the lattice reduction one more time for each of the lattices L_{pq}. This can be described as follows:

1. Compute a short basis for each two dimensional lattices generated by the elements (r_q, q) of M. That is, the lattices $(r_q, 1)\mathbb{Z}^2 \oplus (q, 0)\mathbb{Z}^2$, and we output a short basis by the Lagrange's method say (u_1, u_2) , (v_1, v_2).

2. For each output from 1., compute the short basis of L_{pq}, a sub-lattice of L_q: where $(r_p, p) \in SL$.

$$L_{pq} = \left(\frac{v_1 - r_p v_2}{u_1 - r_p u_2} \mod (p), 1 \right) \mathbb{Z}^2 \oplus (p, 0)\mathbb{Z}^2$$

3. For each of these $Mp \times Sl$ lattices and a given sieving interval we perform the sieve over the small prime SS.

A similar setup and configuration as in the sieving by rows can be applied here.

Implementation issue

In the experiment, we used Tesla P100[2]. Some configuration parameters of the code may be different for a different hardware.
For both sieve, the small factor-base must be accessed on the global memory, sorted with respect to the prime coordinates and divided into two arrays where the root is copied into the constant memory and the difference of the primes into the texture memory as the whole factor-base cannot fit into the constant or texture memory.

We use the implementation in [Tea17] to generate the polynomial within some parameters. In fact, we are given: two polynomials, skew, factor-base, q_{min} and q_{range}, bound which control the bits of the elements in the sieving area.
The q_{min} is needed to determine the value which splits the factor-base. We need to sieve over the candidates divisible by all primes $q > q_{min}$, which means that q_{min} is optimized so that the proportion of the candidates which are not divisible by those primes q are negligible [Bar16]. The q_{range} is obtained by the optimization of our code, in fact it can be omitted and the lattice sieve is applied as long as the sieving area can fit on the global memory. The detailed description of the implementation is the following :

Lattice reduction:
Consists of one kernel, the input is (a part of) the medium factor-base where $q > q_{min}$, and the output is the reduced basis with the same dimension. The principal key here is the function which computes the short basis, our implementation worked on a 64 bits integer. After tracking the amount of memory in runtime used in the function, we calculate the occupancy estimation of the kernel to figure out the dimension of grid and block. This is the reason why the parameters used in the kernel are different for different hardware.

Lattice sieve:
Consists of one kernel, the input is the reduced basis and the output is the array indexed by the candidates which contains their smooth marks. It follows a similar process as in the lattice reduction, the occupancy estimation is calculated to obtain the dimension of the grid and block.
Other parameters are given as input, the attempt is that each block sieve over the same area but with different lattice. Indeed, we use square sieving area

$$[-u/2, u/2] \times [1, u]$$

depending on the bound $u > 0$ we provide an x_0 and y_0 to determine the start point of the sieve. Also we sieve over all the element on the square without checking coprime to avoid thread divergence. This procedure is shown in Figure 4.1 where the coordinates in red indicate the thread indexes, and in black the sieve interval. The black filled squares indicate the value where we do not have coprime coordinates while the blue represents one piece of sieving area handled by one threads-block. The arrow indicates that in this case the sieve

[2] https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf

is done in 4 times by assuming that the blue area is handled by one launch (notice that the the area is the same but the candidates differs depending of the lattice). Each piece of the sieve area is mapped into a file so that we can reuse the memory (mmap).

CPU code:
Using the similar indexed area as above we use CPU to compute the logarithm of the polynomial value of the candidates. We skip the non-coprime indexes (can be seen as a pre-filter). The list of this value is mapped to a memory file.

Smoothness test:
This kernel takes two inputs, the two sieving area GPU and CPU. These two arrays are the results from the GPU sieving which contains the logarithm contribution of the prime and from the CPU code which contains the logarithm value of each polynomials. It outputs the difference between the two arrays.

Trial division:
Finally we make the trial division by walking through the sieving area. Also this step is done on the CPU and consists of collecting the potential candidates (with respect to a given threshold) followed by a trial division on the rational factor-base, for the case when it is smooth we perform the trial division on the other side.

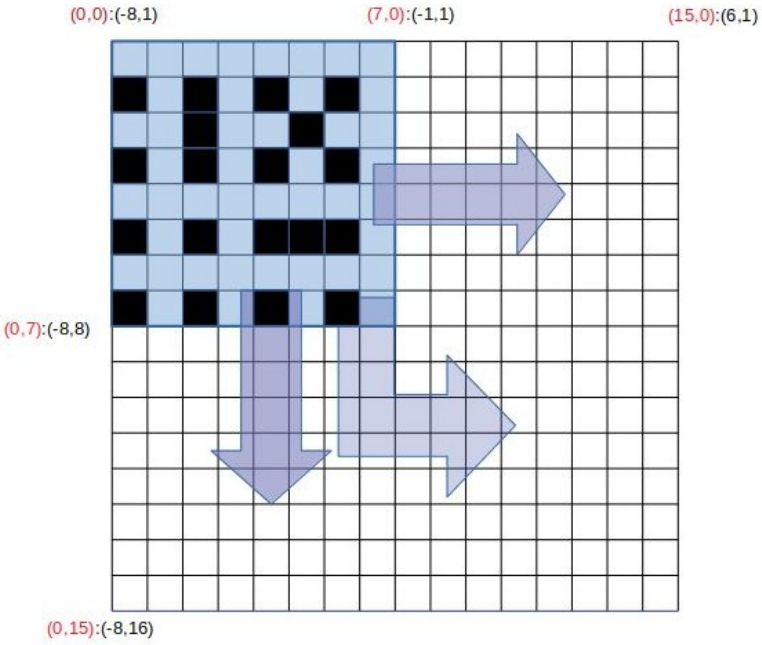


FIGURE 4.1: A grid of threads which perform the sieving.

Remarks.

- This is a standard configuration for the two side (rational and algebraic). For the trial division we always tried to factor the other side polynomial and allow some large prime factor not included in the factor-base but controlled by the user (referring to the command line in cado-nfs[Tea17], these are the `mfb0` and `mfb1`).

- Theses sequences of configuration can be used in a multi-GPU system. This is very useful when the grid-dimension of the GPU does not fit the medium prime interval for a fixed dimension of the thread-block (the sieving area). However the main challenge here is setting the parameters such that the small prime interval can be fitted in the constant memory. It confirms the fact that *sieving by vectors* is recommended for large number of factor-bases. Furthermore a similar configuration can be used for the sieve by vectors. The CPU codes are used to handle large number, the polynomial value (compute the logarithm and factorization), since this requires multi-precision operation.

Experiment results

We have accessed the setup, steps and results from the cado-nfs [Tea17] but the code is impossible to read line by line. For instance for each given range of medium prime factor-base, the software computes the lattice reduction followed by the detection of smooth and the factorization of the candidates (over the sieving area parameterized by these medium factor-base) in one launch. In this we are able to find the same total number but different value of candidates.
The main purpose of this task was to propose an implementation of the sieving step over the GPU by means of lattice sieve. The overview of this work can be detailed as the following Table 4.1; An usual drawback of GPU code due the memory struc-

Procedure	GPU	CPU
Lattice reduction	✓	
Sieving	✓	
Polynomial-value		✓
Detect smooth	✓	
Trial division		✓

TABLE 4.1: Overview.

ture compared to CPU, we have to perform the operation on large number over the CPU. To give a benchmark in this task, it is enough to compare the sieving step on GPU and CPU. The full implementation (from lattice reduction to trial division) is to verify the correctness of our procedure. We compare the two implementations of the lattice sieve (lattice reduction and sieving) CPU and GPU by their running time proportional to the area of sieving as this depends mainly on the sieve area. In this way we also could experiment on the parameters used on the kernel function, this is detailed in Appendix 5.A.

For the *sieve by vectors*, the pre-processing for the basis of the second lattice L_{pq} is performed on the CPU and its reduction is done on GPU, this is done for each element of the medium factor-base, also we save the reduced basis into a file to be used in the next step (trial division). Due to these, our implementation of the sieving by vectors is much slower than the sieving by rows. Notices that the size of factor-base

type	vectors	rows
contents	• basis reduction • sieving	• basis reduction level 1 • basis reduction level 2 • sieving
CPU (pre-processing)	No	Yes
Sieving area	$S \times Sint$	$S \times Sl \times Sint$
$Sint$	large	small
non-coprime candidates w.r.t $Sint$	49%	47%

TABLE 4.2: Summary.

does not affect much on the sieving part, this steps is mainly depend on the size of small factor-base. The two lattice sieve can be summarized as in the Table 4.2

Benchmark

To measure the performance of our implementation of the lattice sieve on GPU, we compare the time taken by the two implementations GPU and CPU. For the CPU, we have a parallel implementation of the sieving using Multithreading in C++. This is presented in Figure 4.2. The fluctuation at the 70-digits number is due to the cardinal of its small factor-base. In Figure 4.3, we have single-GPU vs single-CPU.

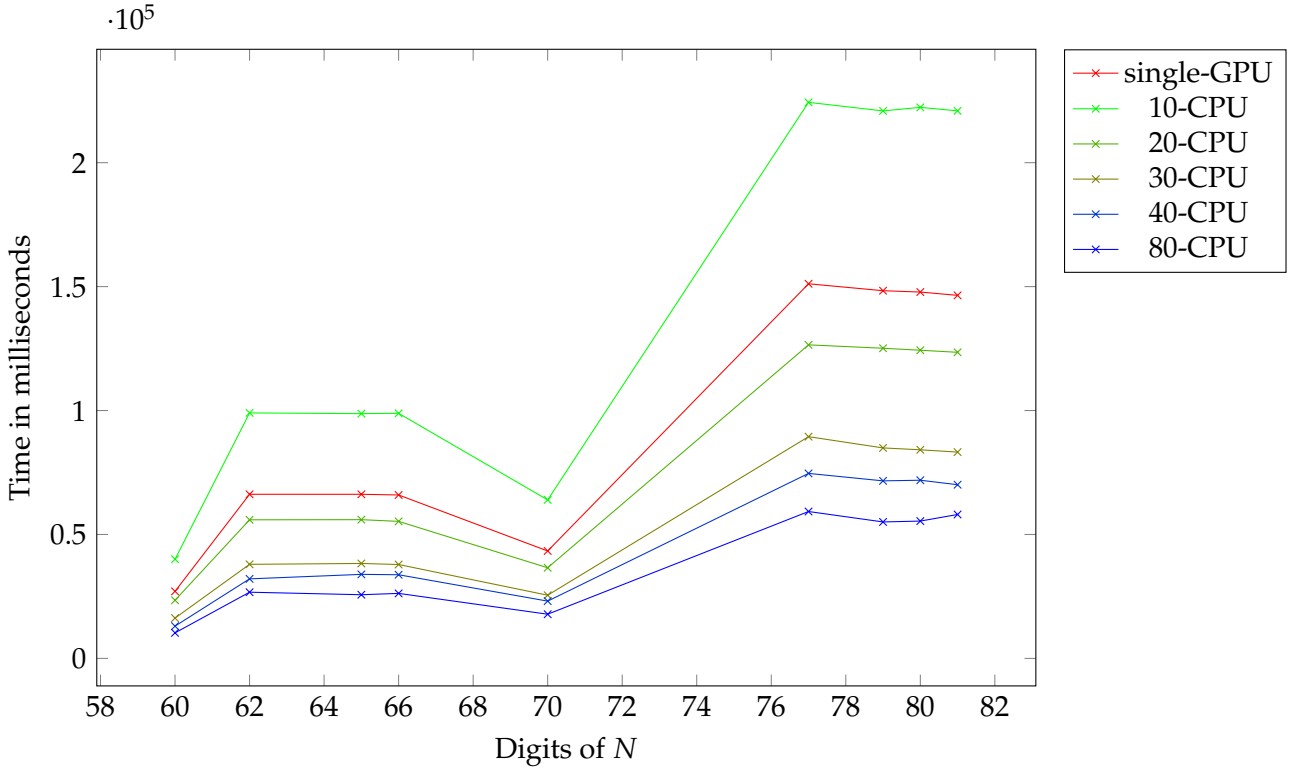


FIGURE 4.2: GPU vs CPUs.

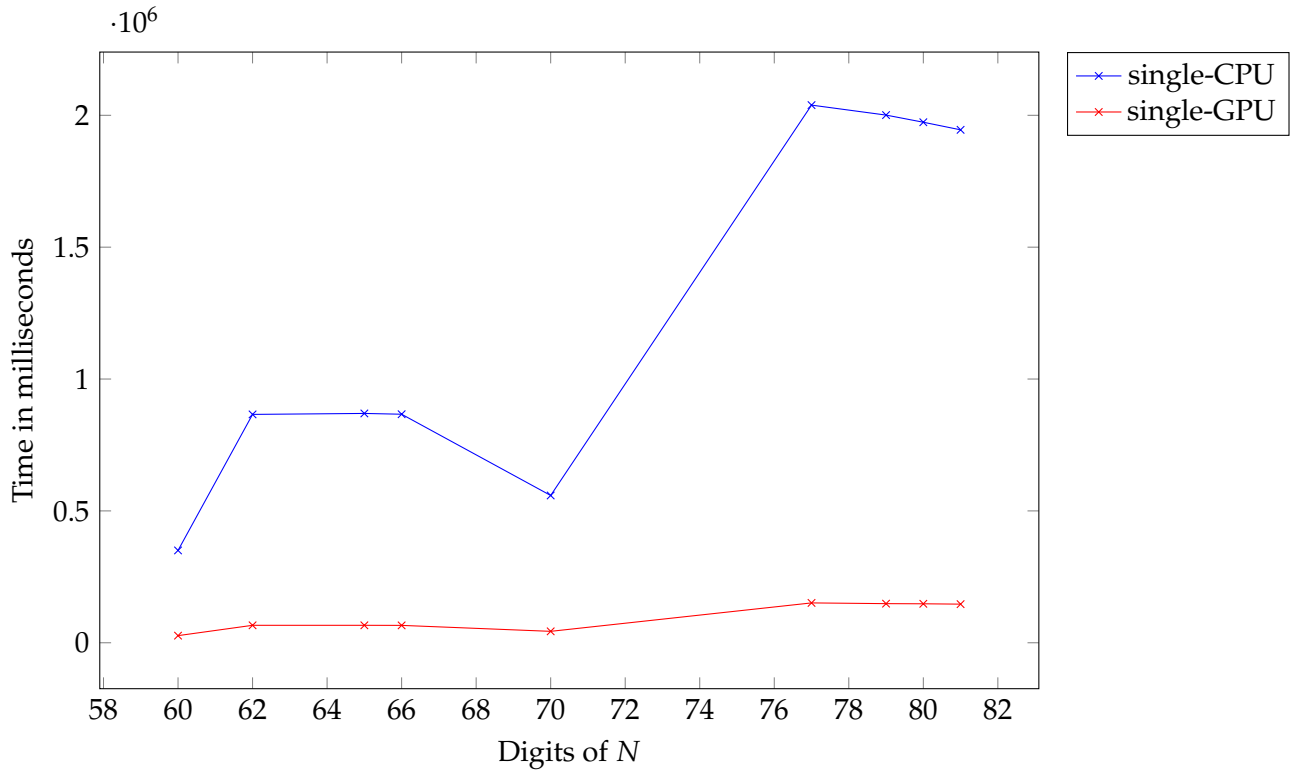


FIGURE 4.3: single-GPU vs single-CPU.

5. Summary and future work

During this project, we explored the lattice sieve procedure, a proposal algorithm for the sieving steps of the General Number Field Sieve. In which, we highlighted that it is the dominant step of this algorithm. Using parallel computing resources we are able to gain a speed-up of the algorithm.
From the property of the lattice sieve, we can observe an independence of the data and its computation, that makes the procedure to be easy to implement in a parallel environment. In fact, there are many parallel version of this step implemented on a multi-core CPU for example in [Tea17]. In our interest, we exploited the latest graphic card oriented for high performance computing by the help of CUDA (parallel computing platform and programming model invented by NVIDIA) to deal with this task.

In the present work, we have implemented the two lattice sieve methods, by rows and vectors on GPU. We gave the benchmark of the implementation for the sieving by rows with respect to the CPU version.
For the sieve by vectors, we are not able to give a benchmark as our implementation is slow (both CPU and GPU) which needs to be optimized. Indeed, according to the structure of sieve by vectors procedure, we process the sieve over the $(e, f)-$plane that requires two level of the basis reduction on the lattice and sublattice. On the second reduction, the original basis needed a pre-processing phase where we perform these operations on the CPU. For the number of operations, we have per each lattice L_q overall its sub-lattice L_{pq} ($\#M \times \#SL$). Also, the sieve is performed on large candidates as we pointed in our experiment (Appendix 5.A).
To summarize, we are able to confirm that the sieving part of the GNFS can be implemented on the GPU. However, as the implementation we presented here requires the use of cache memory which are limited, we try to use this limit to be our parameter on the choice between the two procedures (rows and vectors). Some of the operations are still performed on the CPU, in general when one needs to perform an operation on large number.

The implementation presented in this thesis can be considered as one of the initiative to the exploitation of the GPU performance in factorization algorithm. Doing this project, we met several constraints that some of them have been handled.

There are possibility of an extension of this project. A combination of the two lattice sieve procedures to speed up the algorithm, i.e if we can reduce the amount of SL factor-base (only the elements which have higher probability to be appear in the smooth candidates) then we apply the sieve by rows for the remains part of SL by adding them in the SS. Also, the prime power in the SS factor-base can be omitted but we estimate the threshold T used in the smooth-checker ($|\log(F(a,b)) - \sum_p \log(p)| < T$). In this way, we do not miss any candidates as we mentioned that the candidates which are smooth only over the M and SS factor-base are apparently missed in the sieve by vectors.
An other issue is the operation on large number. Regarding to the available register memory on one processor, an optimized fixed length multi-precision operation is needed to avoid the use of CPU and the transfer of data between the GPU and CPU.

References

(Adl91) Leonard M. Adleman, *Factoring numbers using singular integers*, Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '91, ACM, 1991, pp. 64–71.

(AMB75) Michael A. Morrison and John Brillhart, *A method of factoring and the factorization of f7*, Mathematics of Computation, vol. 29, 01 1975, pp. 183–205.

(Bar16) Razvan Barbulescu, Lecture notes, *summer school in New Delhi*, 14-22 July 2016, contact in person.

(BL93) Daniel J. Bernstein and A. K. Lenstra, *A general number field sieve implementation*, in Lenstra and Hendrik W. Lenstra [LHWL93], pp. 103–126.

(BLP92) J. P. Buhler, H. W. Lenstra, and C. Pomerance, *Factoring integers with the number field sieve*, in Lenstra and Hendrik W. Lenstra [LHWL93], p. 50.

(Bre89) D. M. Bressoud, *Factorizations and primality testing*, 1989. MR 91e:11150

(Bru51) N.G. Bruijn, de, *The asymptotic behaviour of a function occuring in the theory of primes*, Journal of the Indian Mathematical Society. New Series **15** (1951), 25–32 (English).

(Cav00) Stefania Cavallar, *Strategies in filtering in the number field sieve*, Proceedings of the 4th International Symposium on Algorithmic Number Theory (London, UK, UK), ANTS-IV, Springer-Verlag, 2000, pp. 209–232.

(CEP83) E.R Canfield, Paul Erdös, and Carl Pomerance, *On a problem of oppenheim concerning* ***factorisatio numerorum***, Journal of Number Theory **17** (1983), no. 1, 1 – 28.

(Cop93) Don Coppersmith, *Solving linear equations over GF(2): block lanczos algorithm*, 33–60.

(Cop94) ———, *Solving homogeneous linear equations over GF(2) via block wiedemann algorithm*, no. 205, 333.

(Cou93) Jean-Marc Couveignes, *Computing a square root for the number field sieve*, The development of the number field sieve (Berlin, Heidelberg) (Arjen K. Lenstra and Hendrik W. Lenstra, eds.), Springer Berlin Heidelberg, 1993, pp. 95–102.

(CW85) Jane Cullum and Ralph A. Willoughby, *A survey of Lanczos procedures for very large real symmetric eigenvalue problems*, Journal of Computational and Applied Mathematics **12-13** (1985), 37–60 (en).

(DHS85) J A Davis, D B Holdridge, and G J Simmons, *Status report on factoring (at the sandia national labs)*, Proc. Of the EUROCRYPT 84 Workshop on Advances in Cryptology: Theory and Application of Cryptographic Techniques (New York, NY, USA), Springer-Verlag New York, Inc., 1985, pp. 183–215.

(Dic30) K. Dickman, *On the frequency of numbers containing prime factors of a certain relative magnitude*, Arkiv för matematik, astronomi och fysik, Almqvist & Wiksell, 1930.

(DS87) Robert D. Silverman, *The multiple polynomial quadratic sieve*, Mathematics of Computation - Math. Comput., vol. 48, 01 1987, pp. 329–329.

(FK) Jens Franke and Thorsten Kleinjung, *CONTINUED FRACTIONS AND LATTICE SIEVING*, 11 (en).

(GCL92) Keith O. Geddes, Stephen R. Czapor, and George Labahn, *Algorithms for computer algebra*, first ed., Kluwer, Boston, 1992.

(GLM94) Roger A. Golliver, Arjen K. Lenstra, and Kevin S. McCurley, *Lattice sieving and trial division*, ANTS, Lecture Notes in Computer Science, vol. 877, Springer, 1994, pp. 18–27.

(KAF+10) Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman Te Riele, Andrey Timofeev, and Paul Zimmermann, *Factorization of a 768-bit rsa modulus*, Proceedings of the 30th Annual Conference on Advances in Cryptology (Berlin, Heidelberg), CRYPTO'10, Springer-Verlag, 2010, pp. 333–350.

(Kal95) Erich Kaltofen, *Analysis of coppersmith's block wiedemann algorithm for the parallel solution of sparse linear systems*, Mathematics of Computation **64** (1995), no. 210, 777–806.

(Kle06) Thorsten Kleinjung, *On polynomial selection for the general number field sieve*, Mathematics of Computation, vol. 75, 2006, pp. 2037–2047.

(Knu97) Donald E. Knuth, *The art of computer programming, volume 2: Seminumerical algorithms*, third ed., Addison-Wesley, Boston, 1997.

(Lan52) C. Lanczos, *Solution of systems of linear equations by minimized iterations*, no. 1, 33.

(Leh28) Derrick H. Lehmer, *The mechanical combination of linear forms*, American Mathematical Monthly **35** (1928), 114–121.

(LHWL93) Arjen K. Lenstra and Jr. Hendrik W. Lenstra (eds.), *The development of the number field sieve*, Lecture Notes in Mathematics, vol. 1554, Springer-Verlag, Berlin, 1993.

(Lip76) John D. Lipson, *Newton's method: A great algebraic algorithm*, Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation (New York, NY, USA), SYMSAC '76, ACM, 1976, pp. 260–270.

(LLMP93a) A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and John M. Pollard, *The factorization of the ninth fermat number*, vol. 61, 1993, p. 319.

(LLMP93b) ———, *The number field sieve*, in Lenstra and Lenstra [LHWL93], pp. 11–42.

(LN15) Andrianaivo Louis N., *Primality and compositeness tests*, Master's thesis, AIMS-Senegal, (African Institute for Mathematical Science), Mbour, 2015.

(**LV18**) Jonathan D. Lee and Ramarathnam Venkatesan, *Rigorous analysis of a randomised number field sieve*, vol. 187, 2018, pp. 92 – 159.

(**Mon95**) Peter L. Montgomery, *A block lanczos algorithm for finding dependencies over gf(2)*, Proceedings of the 14th Annual International Conference on Theory and Application of Cryptographic Techniques (Berlin, Heidelberg), EUROCRYPT'95, Springer-Verlag, 1995, pp. 106–120.

(**Mur99**) Brian Murphy, *Polynomial selection for the number field sieve integer factorisation algorithm*, Ph.D. thesis, 1999.

(**Pol93a**) John M. Pollard, *Factoring with cubic integers*, in Lenstra and Lenstra [LHWL93], pp. 4–10.

(**Pol93b**) John M. Pollard, *The lattice sieve*, in Lenstra and Hendrik W. Lenstra [LHWL93], pp. 43–49.

(**Pol09**) John M. Pollard, *A monte carlo method for factorization*, BIT, vol. 15, 1975-09, pp. 331–334.

(**Pom82**) Carl Pomerance, *Analysis and comparison of some integer factoring algorithms*, Computational Methods in Number Theory (Math Centrum, Amsterdam) (H. W. Lenstra and R. Tijdeman, eds.), Math Centre Tracts-Part 1, 1982, pp. 89–139.

(**Pom85**) Carl Pomerance, *The quadratic sieve factoring algorithm*, Advances in Cryptology (Thomas Beth, Norbert Cot, and Ingemar Ingemarsson, eds.), vol. 209, Springer Berlin Heidelberg, 1985, pp. 169–182.

(**ShI11**) Bai ShI, *Polynomial selection for the number field sieve integer factorisation algorithm*, Ph.D. thesis, 2011.

(**Tea17**) The CADO-NFS Development Team, *CADO-NFS, an implementation of the number field sieve algorithm*, 2017, Release 2.3.0.

(**Tho02**) Emmanuel Thomé, *Subquadratic Computation of Vector Generating Polynomials and Improvement of the Block Wiedemann Algorithm*, Journal of Symbolic Computation **33** (2002), no. 5, 757–775 (en).

(**Tho12**) Emmanuel Thomé, *Square root algorithms for the number field sieve*, 4th International Workshop on Arithmetic in Finite Fields - WAIFI 2012 (Bochum, Germany) (Ferruh Özbudak and Francisco Rodríguez-Henríquez, eds.), Lecture Notes in Computer Science, vol. 7369, Springer, July 2012, The original publication is available at www.springerlink.com, pp. 208–224.

(**Tho16**) Emmanuel Thomé, *A modified block lanczos algorithm with fewer vectors*, vol. abs/1604.02277, 2016.

(**Wie86**) D. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory **32** (1986), no. 1, 54–62 (en).

(**Wik18**) Wikipedia contributors, ***Krylov subspace*** *— Wikipedia, the free encyclopedia*, 2018, [Online; accessed 29-september-2018].

Appendices

5.A Implementation issue of the lattice sieve

We have implemented two methods for the lattice sieve, by rows and vectors. In fact, the two methods are chosen depending on the GPU features with respect the size of the factor-base. In our experiment, we used the GPU Tesla P100 occupied by $64Kb$ of constant memory so that we have $c = 8192$ of 8 byte integer at most (this number is can be reduced in constraint of the memory for some variable used in running time). If the size of small factor-base is more than this number then we split the small factor-base into two sets SS and SL and use the sieving by vectors.

5.A.1 Configuration of the sieving by rows

Let S and M be the small and medium factor-base respectively. We assume that $\#S < c$ (taking care the other use of constant memory for the cuda kernel). We first pre-process the set S in which we copy the roots on the constant memory and the difference of prime on the texture memory.
Let $(q, r_q) \in M$, we have the lattice L_q defined by $(r_q, 1)\mathbb{Z}^2 \oplus (q, 0)\mathbb{Z}^2$. The sieving by rows is composed by two kernels:

- **basis reduction**
  In here, we performs the lattice reduction per each L_q handled by one thread. The Lagrange's method is implemented for a two-dimensional lattice of 64 bits which requires 31 registers memory. We deduced the dimension of the block-thread by calculating the warps occupancy of the code with respect the features of our GPU, see Figure 5.A.1.

- **sieve**
  Let $I > 0$ such that $u < 2^I$ and the sieving area is defined by $[-u/2, u/2] \times [1, u]$ in which we sieve over the $(c, d)-$plane. We consider $\#M$ pieces of the sieving area, in this way we could perform the sieve for all lattice L_q for $q \in M$ in one launch (notice that each piece are equal but for different value $(a, b) \neq (c, d)$). Each piece is handled by one block-thread of dimension $u \times u$ (we prefer to sieve over a square). The code uses 32 registers memory, the value of u is deduced by calculating the warp occupancy Figure 5.A.2.

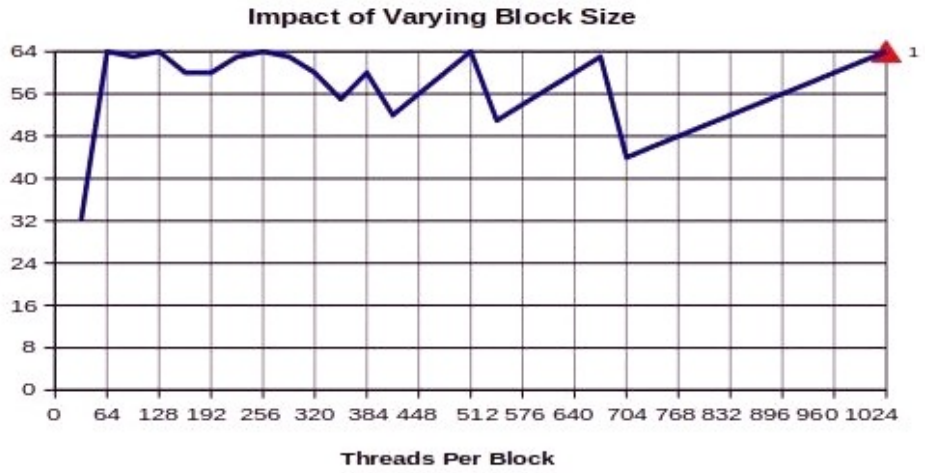


FIGURE 5.A.1: warps occupancy for the basis reduction.

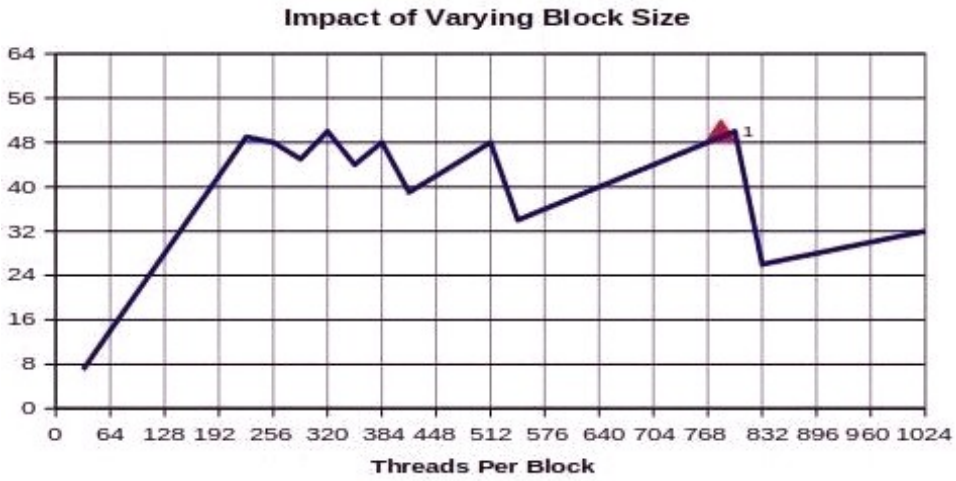


FIGURE 5.A.2: warps occupancy for the sieve (sieving by rows).

The source codes of the kernel used in sieve by rows is presented in the following,

```
/*
 ============================================================================
 Name        : kernelsrows.cu
 Author      : Louis
 Version     :
 Copyright   :
 Description : CUDA implementation of the Lattice sieve by rows for GNFS
 ============================================================================
 */

#include "inlinetools.cuh"

/*********************************
***** kernel reduce basis ******
*********************************/

__global__ void ker_reducebasis(myInteger *o_basis,
```

```
                                VecArray<myInteger> primes,
                                VecArray<myInteger> roots)
    {
    int indx = threadIdx.x + blockIdx.x*blockDim.x;
    int stride = blockDim.x*gridDim.x;

    for(int i = indx; i < primes._size ; i+= stride)
        {
        myInteger u_reg[2],v_reg[2];
        u_reg[0] = *(roots._array + i);
        u_reg[1] = 1;
        v_reg[0] = *(primes._array + i);
        v_reg[1] = 0;
        LLL(u_reg,v_reg);
        *(o_basis + 4*i) = u_reg[0];
        *(o_basis + 4*i + 1) = u_reg[1];
        *(o_basis + 4*i + 2) = v_reg[0];
        *(o_basis + 4*i + 3) = v_reg[1];
        }
    }

/*********************
**** kernek sieve ****
*********************/

__global__ void ker_sieve(double *candidates,
                          myInteger *prime,
                         int x_0, int y_0,
                         int size, myInteger *medium,
                         int dim_s)
    {
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    __shared__ double lattice[maxthreads][maxthreads];
    __shared__ myInteger u[4];
    __shared__ myInteger reg_c[maxthreads];
    __shared__ myInteger reg_d[maxthreads];

    lattice[tx][ty] =  log((double) *(prime + blockIdx.x));
    u[0] = *(medium + 4*blockIdx.x);
    u[1] = *(medium + 4*blockIdx.x + 1);
    u[2] = *(medium + 4*blockIdx.x + 2);
    u[3] = *(medium + 4*blockIdx.x + 3);
    reg_c[tx] = tx + x_0;
    reg_d[ty] = ty + y_0;
    __syncthreads();

    myInteger reg_p = 0;
    myInteger reg_r; //reg_c, reg_d;
    for(int s_idx = 0; s_idx < dim_s; s_idx++)
```

```
        {
        reg_r = Smallroot[s_idx];
        reg_p += (myInteger) tex1Dfetch(Smallptex,s_idx);
        lattice[tx][ty] +=
            ((((reg_c[tx]*u[0]
                + reg_d[ty]*u[2]) - reg_r*(reg_c[tx]*u[1]
                + reg_d[ty]*u[3])) % reg_p) == 0)
                ? log((double) reg_p) : 0.0;
        }
    __syncthreads();

    int linthread = tx + ty*blockDim.y + blockIdx.x*blockDim.x*blockDim.y;
    if (linthread < size)
        {
        *(candidates + linthread) = lattice[tx][ty];
        }
}
```

5.A.2 Configuration of the sieving by vectors

We assume that $\#S > c$, we split S into SS and SL. Let assume again that the two factor-bases does not share the same prime. Similar steps as in the sieving by rows except that here the sieve part is done in two phases: lattice reduction and sieve.
In fact, from 4 we have a description of this process saying that given the lattice L_q for all $(q, r_q) \in M$ we convert the sieve part over the (c, d)-plane. The idea is to sieve over the set of small factor-base which are accessible on the global cache memory, this procedure is good as far as the small factor-base could be fitted on these memories (constant and texture). Now for each lattice L_q we define the sub-lattice

$$\left(\frac{v_1 - r_p v_2}{r_p u_2 - u_1} \mod (p), 1\right)\mathbb{Z}^2 \oplus (p, 0)\mathbb{Z}^2$$

where $(p, r_p) \in SL$ and $(u_1, u_2); (v_1, v_2)$ is a short basis for L_q. The sieve is converted on the (e, f)-plane, more precisely over the lattice point of L_{pq}.
This function is composed by two kernels as in the sieving by rows: basis reduction and sieve. The arguments configuration of the lattice reduction is the same as in Figure 5.A.1.

- **sieve**
  According to the property of sieving by vectors, we now sieve over the (e, f)-plane. This is done as in the following. For a each lattice L_q we do a pre-processing over the basis of the sub-lattice L_{pq} for all $p \in SL$, per each of these basis we compute their reduction. And finally we perform the sieve step.
  Notice first that L_{pq} may not be defined $\left(\frac{v_1 - r_p v_2}{r_p u_2 - u_1} \mod (p) = 0\right)$ in which we skip these prime $p \in SL$. Second, we need to control the size of the lattice point (a, b) as we have a fixed size (8 bytes) for all the arithmetic operations. This is due to the fact that we now sieve over an element in the sub-lattice L_{pq} which are technically the candidates divisible by p and q simultaneously, which imply that the sieving area must be controlled by an optimized bound.
  The kernel uses 37 registers memory, the sieving area bound is given by the

blockthreads dimension. With the above remark, we may obtain the smooth candidates in a small area for instance $[-2, 2] \times [1, 6]$ and this occupied 50% on the warp-threads, the variation of this choice can be seen in Figure 5.A.3.

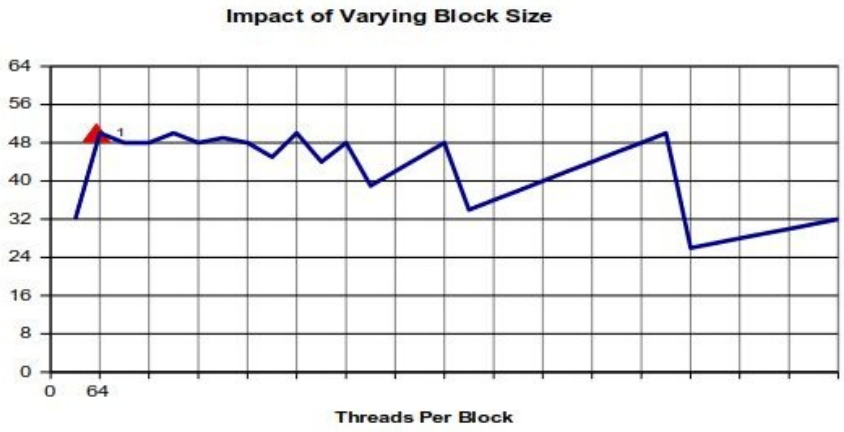


FIGURE 5.A.3: warps occupancy for the sieve (sieving by vectors).

In the following source code, we present our kernel implementation of the sieve by vectors,

```
/*
 ============================================================================
 Name        : kernelsvectors.cu
 Author      : Louis
 Version     :
 Copyright   :
 Description : CUDA implementation of the Lattice sieve by vectors in GNFS
 ============================================================================
 */

#include "inlinetools.cuh"


/*************************
 ***** basis reduction *****
 *************************/

__global__ void ker_reducebasis(myInteger *o_basis,
                                VecArray<myInteger> primes,
                                 VecArray<myInteger> roots)
    {
    int indx = threadIdx.x + blockIdx.x*blockDim.x;
    int stride = blockDim.x*gridDim.x;

    for(int i = indx; i < primes._size ; i+= stride)
        {
        myInteger u_reg[2],v_reg[2];
        u_reg[0] = *(roots._array + i);
        u_reg[1] = 1;
        v_reg[0] = *(primes._array + i);
        v_reg[1] = 0;
        LLL(u_reg,v_reg);
        *(o_basis + 4*i) = u_reg[0];
        *(o_basis + 4*i + 1) = u_reg[1];
        *(o_basis + 4*i + 2) = v_reg[0];
        *(o_basis + 4*i + 3) = v_reg[1];
        }
    }

/*********************
**** sieve kernel ****
*********************/

__global__ void ker_sieve_v2(double *candidates,
            myInteger prime,
            myInteger f_basis[4],
            VecArray<myInteger> _prime,
            myInteger *s_basis,
```

```
        int dim_s,
        int size)
{
int tx = threadIdx.x;
int ty = threadIdx.y;

__shared__ double lattice[maxthreadsX][maxthreadsY];
__shared__ myInteger sh_sbasis[4];

lattice[tx][ty] = log((double) prime);
lattice[tx][ty] += log((double) *(_prime._array + blockIdx.x));

sh_sbasis[0] = *(s_basis + 4*blockIdx.x);
sh_sbasis[1] = *(s_basis + 4*blockIdx.x + 1);
sh_sbasis[2] = *(s_basis + 4*blockIdx.x + 2);
sh_sbasis[3] = *(s_basis + 4*blockIdx.x + 3);

__syncthreads();

myInteger reg_p = 0;
myInteger reg_r;

myInteger reg_c = f_basis[0]*(tx - 2) + f_basis[2]*(ty + 1);
myInteger reg_d = f_basis[1]*(tx - 2) + f_basis[3]*(ty + 1);

myInteger reg_a = reg_c*sh_sbasis[0] + reg_d*sh_sbasis[2];
myInteger reg_b = reg_c*sh_sbasis[1] + reg_d*sh_sbasis[3];

for(int s_idx = 0; s_idx < dim_s; s_idx++)
    {
    reg_r = Smallroot[s_idx];
    reg_p += (myInteger) tex1Dfetch(Smallptex,s_idx);
    lattice[tx][ty] +=
        (((reg_a - reg_b*reg_r) % reg_p) == 0)
        ? log((double) reg_p) : 0.0;
    }
__syncthreads();

int linthread = tx + ty*blockDim.y + blockIdx.x*blockDim.x*blockDim.y;
if (linthread < size)
    {
    *(candidates + linthread) = lattice[tx][ty];
    }
}
```

Part II

Statistical Mechanics

6. Studied topic: planar Ising

In this chapter, we give the detail of the procedure we carried out in this part in which we present our numerical results. We start by introducing the theoretical backgrounds we utilized throughout this project, these information were taken from [ADS+19b], [ADS+19a]. The results of the following chapter are part of my contribution in the paper [DNT19].

6.1 Introduction

The Gibbs sampling of lattice spin models is a major task for statistical mechanics. The numerical techniques developed for its realization are based mainly on Markov chain dynamics for single and cluster spin flip [Gla63][SW87][Wol89], and can be easily implemented by means of random mapping representation [HÖ0] techniques. A theory of parallel Markov chains as a Probabilistic Cellular Automaton (PCA) dates back to 1989 [GKLM89]. These processes are characterized by a factorized transition matrix on the configuration space, and their simulation updating all spins by means of the same random map [ADS+19b]. More recently a class of PCAs where transition probabilities are defined in terms of a *pair Hamiltonian* and where the spins are simultaneously updated at each time step has been the subject of several works, e. g., [LS13, DSS12] where PCA are exploited to study the Ising model on planar graphs.
We explore the computational possibilities of this pair Hamiltonian model to generalize the random sampling algorithms for Ising spin systems on a set of two-dimensional lattices.

Formally a PCA is a Markov Chain $(X_n)_{n\in\mathbb{N}}$ whose transition probabilities are such that given two generic configurations $\tau = (\tau_1, \dots, \tau_k)$ and $\sigma = (\sigma_1, \dots, \sigma_k)$

$$P\{X_n = \tau | X_{n-1} = \sigma\} = \prod_{i=1}^{k} P\{(X_n)_i = \tau_i | X_{n-1} = \sigma\} \tag{6.1.1}$$

so that for each time n, the components of the "configuration" are independently updated. From a computational point of view, the evolution of a Markov Chain of this type is well suited to be simulated on parallel processors and GPUs.
In this framework, a new PCA parameterized by J and q, called *shaken dynamics* has recently been introduced [ADS+19b]. The equilibrium measure of the shaken dynamics has been extensively investigated in [ADS+19a] and a critical curve in the plane (q, J) has been explicitly determined.
In particular in [ADS+19a] a model has been proposed where the configurational variables are split into two groups $\tau = (\tau_1, \dots, \tau_k)$ and $\sigma = (\sigma_1, \dots, \sigma_k)$, where $\tau_i, \sigma_i \in \{-1, 1\}$ for each i, and are arranged on a bipartite graph. Different interactions among the τ and σ variables give rise to the possibility of interpolation among different lattice geometries.
The PCA we take into account is a parallel and irreversible version of the heath bath dynamics and is obtained by concatenating two different update rules [ADS+19b]. By means of the Hamiltonian defined in [ADS+19a], which depends on the (J, q) parameters, we identify numerically two regions of the space (J, q) characterized by different behaviors of the dynamics.

In this framework, a new PCA parameterized by J and q, called *shaken dynamics*, has recently been introduced in [ADS+19b]. The equilibrium measure of the shaken dynamics has been extensively investigated in [ADS+19a] and a critical curve in the plane (q, J) has been explicitly determined.
The elementary step of the shaken dynamics is naturally defined on the a finite subset Λ of the square lattice $\mathbb{Z}^2$ and consists of a sequence of two inhomogeneous half steps. However, in both [ADS+19b, ADS+19a] it has been pointed out that the shaken dynamics can be seen as an *alternate dynamics* on a subset of the honeycomb lattice.
The proposed dynamics, although not faster than *ad hoc* dynamics for specific models, allows to simulate a whole class of statistical mechanics models spanning from the one-dimensional Ising model to the square lattice and hexagonal one across all the intermediate models.
Depending on the values of J and q, the shaken dynamics "formalism" defined on the square lattice can be used to simulate a class of Ising models on the honeycomb lattice (as pointed out in [ADS+19a]). Some of the values of J and q are particularly interesting because they allow to use the shaken dynamics to simulate

- the Ising model on the isotropic hexagonal lattice for $J = q$
- (an approximation to) the Ising model on the square lattice for $q >> 1$
- the Ising model on a collection of weakly interacting unidimensional systems for small values of q.

The numerical investigation we put forward is aimed at:

- illustrating a simple heuristic method to numerically determine the critical curve
- evaluating the mixing time of the chain as a function of J and q
- studying the spin-spin correlations as a function of J and q.

Further, for $J = q$ we compare the mixing time of the shaken dynamics with that of a single spin flip dynamics for the Ising model on the hexagonal lattice and, for $q >> 1$ we also compare the mixing time of the shaken dynamics with that of a single spin flip and an alternate parallel dynamics for the Ising model on the square lattice and evaluate the distance between the equilibrium measure of the shaken dynamics from the Gibbs measure for the Ising model on the square lattice.

6.2 The model

Consider the Ising Hamiltonian on a graph $G(V, E)$

$$H_G(\sigma) = -\sum_{(x,y)\in E} J_{xy}\sigma_x\sigma_y \tag{6.2.1}$$

where $\sigma_x \in \{-1, 1\}$ for all the $x \in V$ and $J_{xy} \in \mathbb{R}^+$.

We assume that $V = \Lambda_1 \cup \Lambda_2$, where Λ_1 and Λ_2 are finite squared subsets of the square lattice with L^2 sites and periodic boundary conditions

$$\Lambda = \Lambda_1 = \Lambda_2 = (\mathbb{Z}/L\mathbb{Z})^2 \tag{6.2.2}$$

and all edges in E have one endpoint in Λ_1 and the other in Λ_2. The σ and τ variables denote the Ising configuration on the vertices of Λ_1 and Λ_2. Each σ_u, with $u \in \Lambda_1$, can be put in a one-to-one correspondence with τ_u with the same index $u \in \Lambda_2$.

Let $x = (i, j)$ be a vector of coordinate on the torus $(\mathbb{Z}/L\mathbb{Z})^2$. Then

$$x^{\uparrow} = (i, j+1), \quad x^{\rightarrow} = (i+1, j), \quad x^{\downarrow} = (i, j-1), \quad x^{\leftarrow} = (i-1, j) \tag{6.2.3}$$

are the coordinates of the four points at unit distance from x. Set $J_{xy} = J$ for all $(x, u) \in E$ with $x \neq u$ and $J_{xy} = q$ if $x = y$.
With this notation we obtain the Hamiltonian studied in [ADS+19a, ADS+19b]

$$\begin{aligned} H(\sigma, \tau) &= -\sum_{x \in \Lambda} \left[J\sigma_x(\tau_{x^{\uparrow}} + \tau_{x^{\rightarrow}}) + q\sigma_x\tau_x \right] \\ &= -\sum_{x \in \Lambda} \left[J\tau_x(\sigma_{x^{\downarrow}} + \sigma_{x^{\leftarrow}}) + q\tau_x\sigma_x \right] \end{aligned} \tag{6.2.4}$$

on the pairs of Ising configurations σ on Λ_1 and τ Λ_2. The interactions of this Hamiltonian can be visualized on the induced bipartite graph represented in Figure 6.1 and 6.3. The parameter q is also referred to as the *self interaction parameter*.

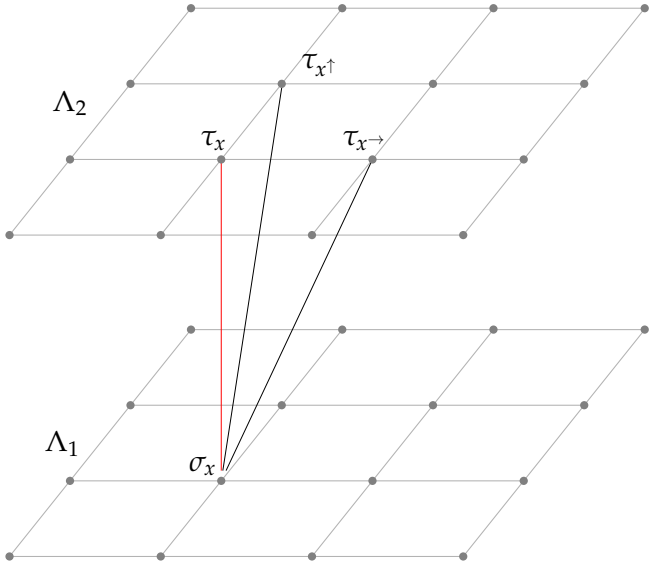


FIGURE 6.1: The lattices Λ_1, Λ_2 with the q (red) and J (black) interactions.

As pointed out in [ADS+19a] a careful look to the Hamiltonian (6.2.4) and to the graph of Figure 6.1 shows that the bipartite graph is isomorphic to the hexagonal lattice $G^{\hexagon}(V, E)$ with edges J and q on whose vertices are arranged the variables σ and τ as shown in Figure 6.2.
The Gibbs measure at temperature $1/\beta$ for the Hamiltonian (6.2.4) is

$$\pi_2(\sigma, \tau) = \frac{e^{-\beta H(\sigma,\tau)}}{\sum_{(\sigma,\tau) \in \mathcal{X} \times \mathcal{X}} e^{-\beta H(\sigma,\tau)}} \tag{6.2.5}$$

where $\mathcal{X} \times \mathcal{X} = \{-1, 1\}^{|\Lambda|} \times \{-1, 1\}^{|\Lambda|}$ is the configuration space of the variable (σ, τ). The critical value of β_c separates the ordered phase where all the spin have the same probability to take the values $+1$ or -1 from the ordered phase where the measure is polarized [Gal72].

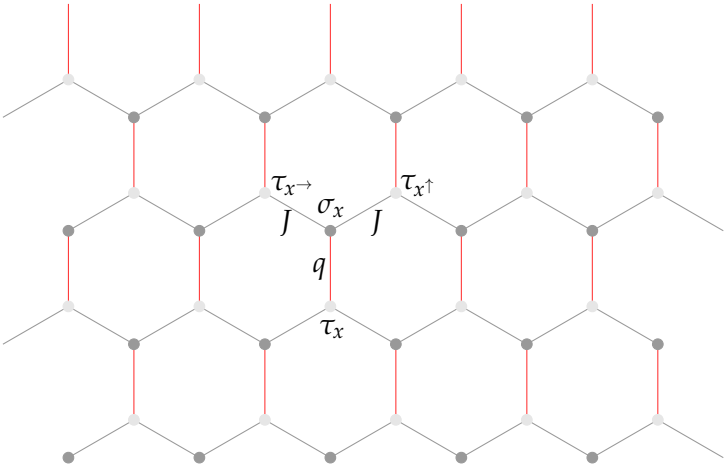


FIGURE 6.2: The hexagonal graph $G^{\hexagon}(\Lambda_1 \cup \Lambda_2, \{J, q\})$

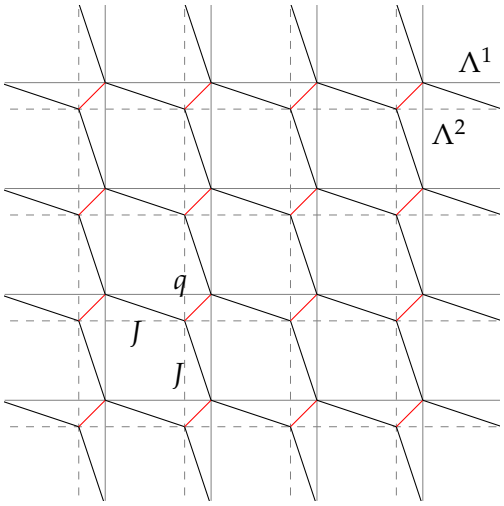


FIGURE 6.3: A representation of the hexagonal graph $G^{\hexagon}(\Lambda_1 \cup \Lambda_2, \{J, q\})$ that highlights the relation with the two square lattices Λ_1 and Λ_2

Rescaling the interactions J and q by β

$$\beta J \to J, \qquad \beta q \to q \tag{6.2.6}$$

it has been proven in [ADS+19a] that there exists a function $J_c(q)$, shown in Figure 6.4, which separate the ordered phase from the disordered one.

The partition function of the Ising model on the honeycomb lattice $G^{\hexagon}$ is

$$Z(J,q) = \sum_{(\sigma,\tau)\in\mathcal{X}\times\mathcal{X}} \prod_{u\in\Lambda} \cosh^2 J \cosh q \left(1+\sigma_x\tau_{x^\uparrow}\tanh J\right)\left(1+\sigma_x\tau_{x^\rightarrow}\tanh J\right)\left(1+\sigma_x\tau_x \tanh q\right) \tag{6.2.7}$$

The graph $G^{\hexagon}$ is a weighted planar graph, non degenerate, finite and doubly periodic. The periodic boundary conditions for Λ_1 and Λ_2 guarantee that the graph $G^{\hexagon}$ is immersed in the torus.

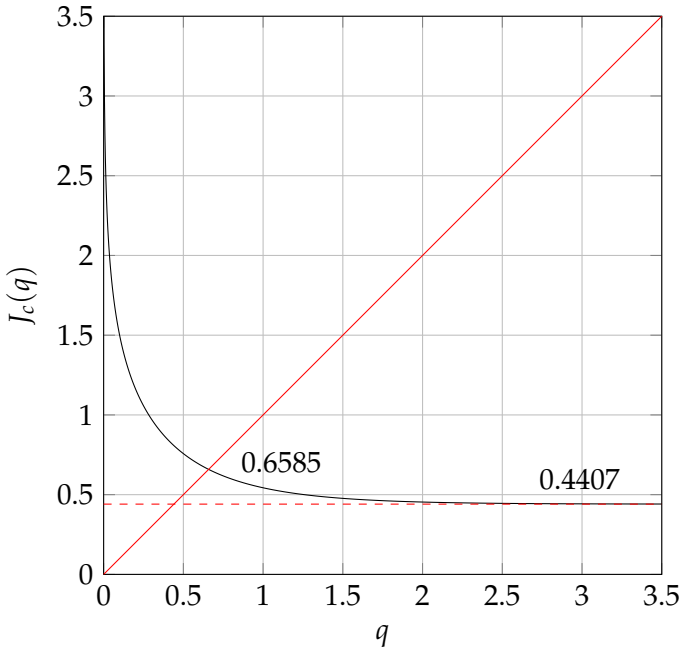


FIGURE 6.4: The critical curve $J_c(q)$

Introducing the following notation on the hexagonal lattice

$$J_e \equiv \begin{cases} J \text{ if } e = (x, x^{\uparrow}) \text{ or } e = (x, x^{\rightarrow}) \\ q \text{ otherwise} \end{cases} \tag{6.2.8}$$

the critical curve $J_c(q)$ for the Hamiltonian (6.2.4) is the unique solution for $J, q > 0$ of the equation

$$\sum_{\gamma \in \mathcal{E}_0(G)} \prod_{e \in \gamma} \tanh J_e = \sum_{\gamma \in \mathcal{E}_1(G)} \prod_{e \in \gamma} \tanh J_e \tag{6.2.9}$$

where $\mathcal{E}_0(G)$ is the set of even subgraphs of $G^{\hexagon}$ winding an even number of times around each direction of the torus, and $\mathcal{E}_1(G) = \mathcal{E}(G) \setminus \mathcal{E}_0(G)$ [ADS+19a][CDC13]. The explicit form of the equation (6.2.9) is

$$1 = 2 \tanh J \tanh q + \tanh^2 J \tag{6.2.10}$$

The solution of equation (6.2.10) with respect to the J

$$J_c(q) = \tanh^{-1} \left(\sqrt{\tanh^2 q + 1} - \tanh q \right) \tag{6.2.11}$$

is plotted in Figure 6.4. We observe that

$$\lim_{q \to \infty} J_c(q) = \tanh^{-1}(\sqrt{2} - 1) = 0.4406867 \tag{6.2.12}$$

is the critical value of β for the Ising model on the lattice square, while on the point $J_c(q) = q$ the equation (6.2.11) gives the critical value for the Ising model on the hexagonal lattice $J = q = 0.6585$. If $q \to 0, J \to \infty$ there are no phase transitions as in the unidimensional Ising model. Following [ADS+19b] we let the system evolve as a Markov chain where the spins in Λ_1 and in Λ_2 are alternatively updated with a

probability proportional to the exponential of the Hamiltonian of the target configuration in $\mathcal{X} \times \mathcal{X}$.
More precisely, using the notation

$$\begin{aligned} \overrightarrow{h_x}(\sigma) &= J(\sigma_{x^\uparrow} + \sigma_{x^\rightarrow}) + q\sigma_x \\ \overleftarrow{h_x}(\sigma) &= J(\sigma_{x^\downarrow} + \sigma_{x^\leftarrow}) + q\sigma_x \end{aligned} \tag{6.2.13}$$

we consider a Markov chain on $\mathcal{X} \times \mathcal{X}$ with transition probabilities given by

$$P((\sigma,\tau),(\sigma,\tau')) = \frac{e^{-H(\sigma,\tau')}}{Z_\sigma} = \frac{e^{-\sum_{u\in\Lambda}\overleftarrow{h_u}(\sigma)\tau'_u}}{Z_\sigma} = \prod_{u\in\Lambda}\frac{e^{\overleftarrow{h_u}(\sigma)\tau'_u}}{2\cosh\overleftarrow{h_u}(\sigma)} \tag{6.2.14}$$

at odd times and

$$P((\sigma,\tau),(\sigma',\tau)) = \frac{e^{-H(\sigma',\tau)}}{Z_\tau} = \frac{e^{-\sum_{u\in\Lambda}\overrightarrow{h_u}(\tau)\sigma'_u}}{Z_\tau} = \prod_{u\in\Lambda}\frac{e^{\overrightarrow{h_u}(\tau)\sigma'_u}}{2\cosh\overrightarrow{h_u}(\tau)} \tag{6.2.15}$$

at even times where $Z_\sigma = \sum_{\eta\in\mathcal{X}} e^{-H(\sigma,\eta)}$ and $Z_\tau = \sum_{\eta\in\mathcal{X}} e^{-H(\eta,\tau)}$.
The factorization in Equation (6.2.14) and (6.2.15) and the mutual dependence of the variables σ and τ makes it quite easy the parallel numerical implementation of this dynamics. In particular, to simulate the evolution of the chain it is possible to sample the value $\zeta \in \{-1,1\}$ of the spin at site u with probability $P(\tau'_u = \zeta|\sigma) = \frac{e^{\zeta\overleftarrow{h_u}(\sigma)}}{2\cosh\overleftarrow{h_u}(\sigma)}$ at odd times and $P(\sigma'_u = \zeta|\tau) = \frac{e^{\zeta\overrightarrow{h_u}(\tau)}}{2\cosh\overrightarrow{h_u}(\tau)}$ at even times independently for all $u \in \Lambda$.
In this framework, the shaken dynamics introduced in [ADS+19b] is obtained by looking at the evolution of the spin configuration in Λ_1. In other words, the shaken dynamics is the marginal of the alternate dynamics defined by Equations (6.2.14) and (6.2.15) and the shaken transition probabilities are

$$P^{\square}(\sigma,\sigma') = \sum_\tau \frac{e^{-H(\sigma,\tau)}}{Z_\tau}\frac{e^{-H(\sigma',\tau)}}{Z_\tau}$$

In [ADS+19b] it has been proven that the equilibrium measure of this dynamics is

$$\pi_s(\sigma) = \frac{Z_\sigma}{Z}$$

and $Z = \sum_\sigma Z_\sigma$.
In the remainder of the paper, we use the wording *shaken dynamics* when we are interested in the evolution on the sub–lattice Λ_1 whereas we call the dynamics on the hexagonal lattice subject to the transition probabilities (6.2.14) and (6.2.15) the *alternate parallel dynamics* (on the hexagonal lattice).

6.3 Simulation results

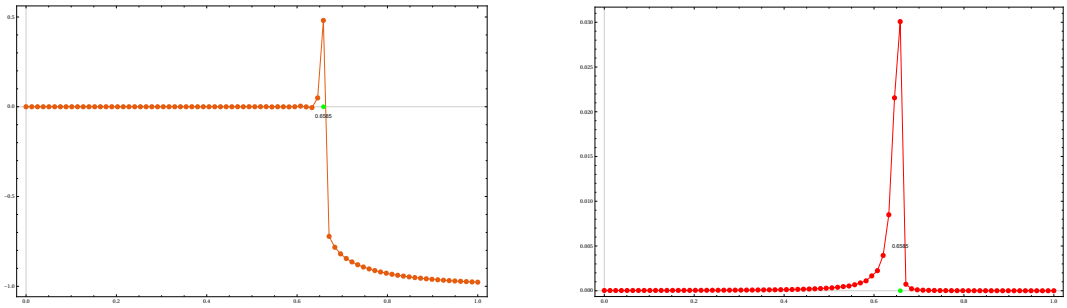
6.3.1 Numerical estimation of critical curve

As stated before, the critical curve (6.2.11) is the function that separates the ordered and the disordered phases. Above this line the values of the spins tend to be highly correlated whereas on the opposite side the value assumed by each spin is weakly

dependent on the values taken by other spins. To determine whether the system is in the ordered or disordered phase we compute, over a large number of iterations, the average and the variance of the magnetization on one of the two layer Λ_i is where the magnetization m is defined as

$$m = \frac{1}{|\Lambda_i|} \sum_{x \in \Lambda_i} \sigma_x \tag{6.3.1}$$

By Theorem 2.1 in [ADS+19a] $\pi_s(m) = \pi_2(m)$, that is, the average magnetization (in Λ_1) of the shaken dynamics is the same as the average magnetization of the parallel alternate dynamics (on the hexagonal lattice $\Lambda_1 \cup \Lambda_2$).
We take Λ to be a 200×200 torus and simulate the evolution of the shaken dynamics starting from configuration $\sigma_0 = \{-1, -1, \dots -1\}$ for $(J, q) \in \{(0, 2) \times (0, 2)\}$ on a 80×80 grid. We first let the system run for a warm-up time of 300000 steps and then record the average and the variance of the magnetization for 300000 additional steps.
Figure 6.5 shows the average and the variance of the magnetization as a function of J for $q = 0.6585$. It is evident that the average magnetization has a sharp transition around the point $J = 0.6585$ which is the critical value of J for the Ising model on the honeycomb lattice.
Around the same point the variance of the magnetization has a spike whereas it is negligible for values of J far from the critical point.
The results obtained on the whole grid (q, J) are summarized in Figure 6.6.



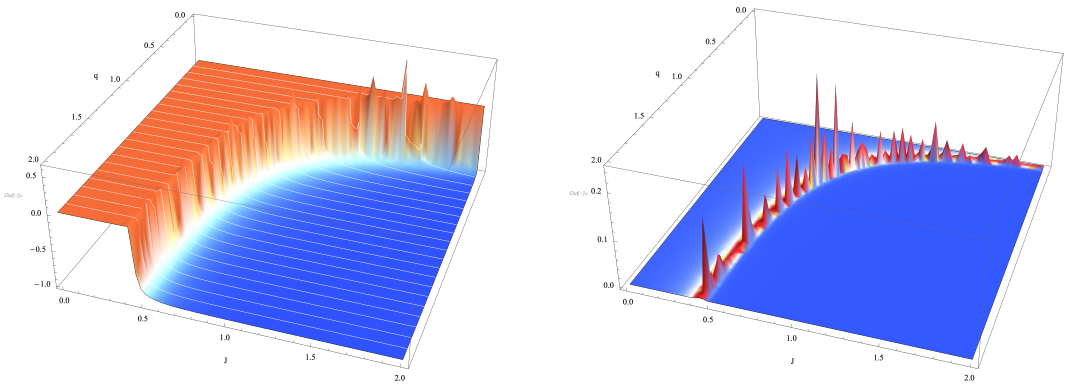
(a) (b)

FIGURE 6.5: Average (a) and variance (b) of the magnetization as a function of J for $q = 0.6585$

It is known that, at equilibrium, the average value of the magnetization fluctuates heavily only close to the critical value of the interactions (see [Rue99] for a reference). Figure 6.7 shows that the variance of the magnetization is significantly different from zero only for points of the (q, J) plane in the vicinity of the pins on the curve (6.2.11). This show that, even for a small lattice, the magnetization fluctuates only close to the critical line and for the whole class of Ising models that can be described tuning the values of J and q.

6.3.2 Coalescence times and perfect sampling

To assess whether the number of steps for which a Markov chain run is large enough for its distribution to be close to the equilibrium distribution, it is convenient to look at its *mixing time*. For a Markov chain $(X_n)_{n \in \mathbb{N}}$ with state space $\mathcal{X}$ and stationary



(a) (b)

FIGURE 6.6: Average (a) and variance (b) of the magnetization on the whole (q, J) grid

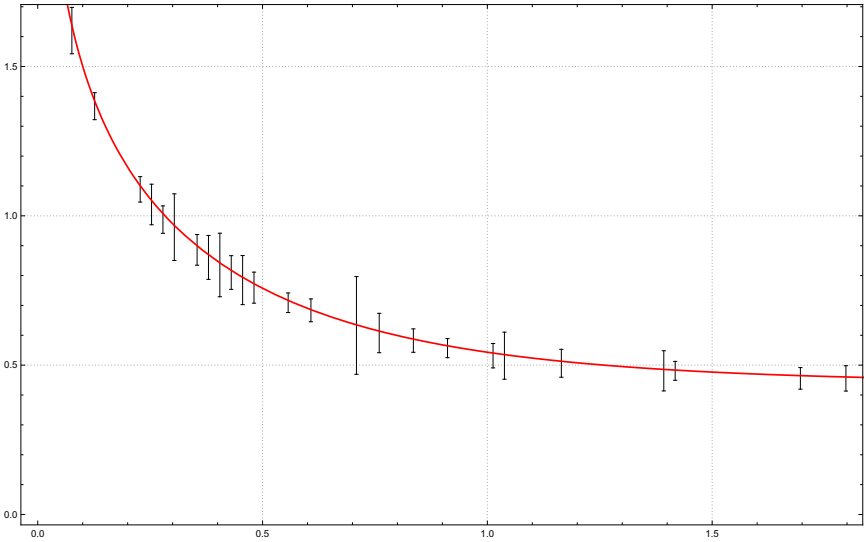


FIGURE 6.7: The bars are centered at those points in the (q, J) plane for which the variance of the magnetization is sufficiently large (≥ 0.03). The length of the bars is proportional to the variance of the magnetization.

distribution π, the mixing time is defined as

$$T_{\text{mix}} = T_{\text{mix}}(\varepsilon) = \min\{n > 0 : \|\mu_\sigma^n - \pi\|_{\text{TV}} < \varepsilon \, \forall \, \sigma \in \mathcal{X}\}$$

where μ_σ^n is the distribution of X_n conditioned on $X_0 = \sigma$, $\|\mu - \nu\|_{\text{TV}}$ denotes the total variation distance between the probability measures μ and ν and ε is some "small" number (for instance e^{-1}). For a reference on mixing times see, for instance, [LP17]. Determining useful bounds for the mixing time of a Markov chain is, in general, a quite challenging task. However, indication on the mixing time of a Markov chain can be gathered looking at the *coalescence times* (see [HÖ00] for a reference).
Consider two Markov chains $(X_n)_{n\in\mathbb{N}}$ and $(Y_n)_{n\in\mathbb{N}}$ living on the same state space $\mathcal{X}$ and consider the coupling $(Z_n)_{n\in\mathbb{N}} = (X_n, Y_n)$ obtained by letting X_n and Y_n to evolve with the same update function and the same sequence of random numbers (for an introduction on the coupling method see [Lin12]). Further assume that the update function is chosen is a way such that $P_Z(X_n = Y_n) \to 1$ as $n \to \infty$.

We define the coalescence time T between X_n and Y_n as $T = \min\{n \in \mathbb{N} : X_n = Y_n\}$. Note that, since X_n and Y_n evolve with the same update function and the same sequence of random numbers $X_n = Y_n$ for all $n > T$. This definition extends naturally to a collection of K chains X_n^k with $k \in 1 \dots K$.
The mixing time of the chain $(X_n)_{n\in\mathbb{N}}$ is estimated by the coalescence time of the chains $(X_n^k)_{n\in\mathbb{N}}$ for $k = 1 \dots |\mathcal{X}|$, all defined on the state space $\mathcal{X}$, where chain X_n^k has initial distribution concentrated on state k.
To effectively determine the coalescence time of the shaken dynamics, however, it is not necessary to run $2^{|\Lambda|}$ copies of the Markov chains, but it is possible to use the so called sandwiching technique since the shaken dynamics preserves the *partial ordering* between configurations[1] . In other words, it can be directly checked that if $X_0^k \leq X_0^l$ then $X_n^k \leq X_n^l$ for all $n > 0$ (see, again, [Hö0], for a reference). To determine the coalescence time it is therefore sufficient to look at the coalescence times of two chains starting, respectively, from $\sigma^{\text{top}} = \{1, 1, \dots, 1\}$ (the largest possible configuration) and $\sigma^{\text{bot}} = \{-1, -1, \dots, -1\}$ (the smallest possible one).
Further note that, leveraging on coupling between Markov chains it is possible to perform an unbiased sampling from the equilibrium distribution of a Markov chain using the Propp–Wilson algorithm, introduced in [PW96], which requires two copies of the Markov chain to be run with the same update function and the same sequence of random numbers.
We studied the coalescence times of the shaken dynamics. The simulations were run taking Λ to be a 32×32 square lattice. This means that the induced hexagonal lattice $\Lambda_1 \cup \Lambda_2$ has $32 \times 32 \times 2$ points.
We computed the average coalescence time for values of J and q close to the critical line $J_c(q)$. The results obtained are summarized in Figure 6.8.

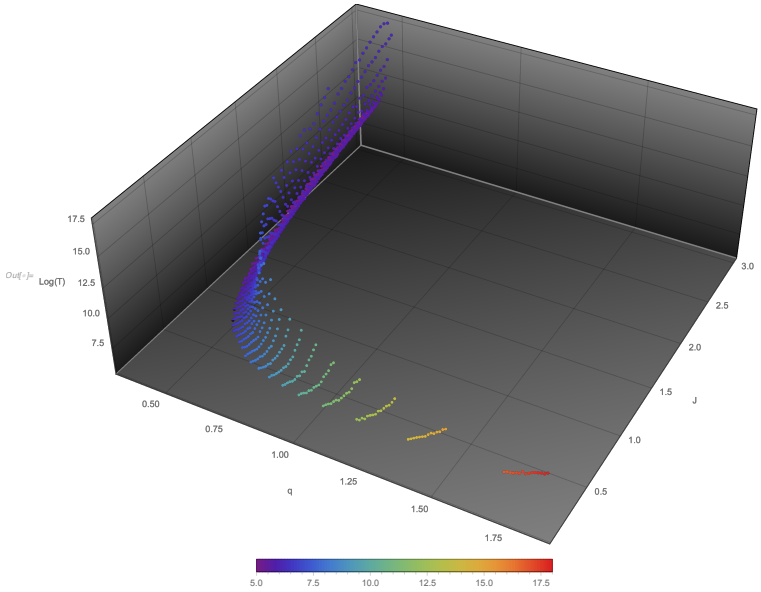


FIGURE 6.8: Logarithm of the average coalescence time for values of J and q close to the critical curve

For $J = q$, the shaken dynamics is the marginal of the alternate dynamics on the isotropic hexagonal lattice. More properly, pairs of configurations (σ, τ) with τ the configuration obtained from σ by performing the first half step of the shaken dynamics can be regarded as spin configurations on the honeycomb lattice. The equilibrium

[1] $\sigma \geq \eta$ if, for all $u \in \Lambda$, $\{\eta_u = +1\} \Rightarrow \{\sigma_u = +1\}$

distribution of these pairs is the Gibbs measure of the Ising model on the isotropic hexagonal lattice (see Theorem 2.1 in [ADS+19b]). Therefore it makes sense to compare the mixing time of the shaken dynamics with the mixing time of a single spin flip dynamics defined on the hexagonal lattice and whose stationary distribution is the Gibbs measure. As a reference we take the heat bath dynamics defined as follows:

$$P(\sigma,\sigma') = \begin{cases} \frac{1}{|\Lambda|}\frac{e^{h_x(\sigma)\sigma_i}}{2\cosh(e^{h_x(\sigma)})} & \text{if } \sigma' = \sigma^x \\ 1 - \sum_{x\in\Lambda} P(\sigma,\sigma') & \text{if } \sigma = \sigma' \\ 0 & \text{otherwise} \end{cases}$$

where σ^x is the configuration obtained from σ by flipping the spin at site u and $h_u(\sigma) = \sum_{y\sim x} J\sigma_y$. Also the heat bath dynamics preserves the partial ordering between configurations and, hence, also in this case it is sufficient to simulate the evolution of two chains one starting from all spins set to $+1$ and one starting from all spins set to -1.
Note that it is possible to argue that the parallel alternate dynamics studied here is a parallel version of the single spin flip heat bath described above.
The results obtained, for several values of J (and, consequently, q) are presented in Figure 6.9. Note that for the single spin flip dynamics the value shown in the chart is the number of steps divided by $2|\Lambda|$ so that, for both algorithms, we are comparing the total number of "attempted spin flips".
It appears that the parallel alternate dynamics is faster than the single spin flip one even if the single spin flip one is "renormalized" with the volume of the box as described above.

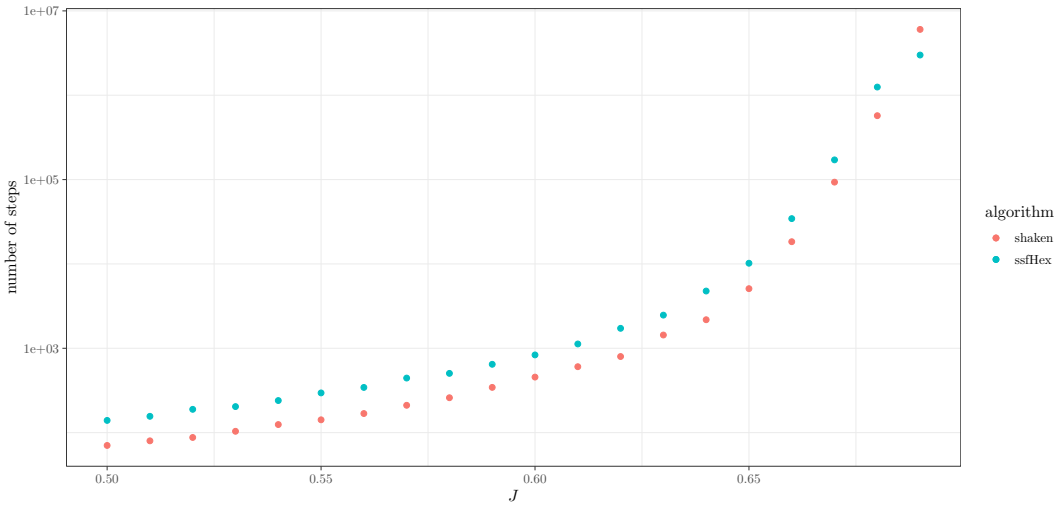


FIGURE 6.9: Sample average of the coalescence time for $J = q$ (hexagonal lattice)

In [ADS+19b], Theorem 2.3 it has been shown that, for large values of q, the equilibrium distribution of the shaken dynamics approaches the Gibbs measure for the Ising model on the square lattice. More precisely it has been proven that, if

$$\lim_{|\Lambda|\to\infty} e^{-2q}|\Lambda| = 0,$$

then, for J sufficiently large,

$$\lim_{|\Lambda|\to\infty} \|\pi_s - \pi_G\|_{\mathrm{TV}} = 0,$$

where π_G is the Gibbs measure for the Ising model on the square lattice. Therefore it makes sense to evaluate numerically the goodness of this approximation as q increases. To this purpose we consider two observable: the magnetization m and the energy $H(\sigma)$. For both observable we compare their sample mean and sample standard deviation over samples drawn from the equilibrium distribution of the shaken dynamics with the sample mean and the sample standard deviation of two other reference dynamics having the Gibbs measure as stationary distribution. One of the two reference dynamics taken into account is, again, the heat bath dynamics. The other dynamics is a parallel version of the heat bath dynamics that updates, alternatively, the spins on the odd and the even sites of the lattice. The latter is the equivalent for the square lattice of the alternate parallel dynamics on the hexagonal lattice defined by equations (6.2.14) and (6.2.15). Theorem 2.2 in [ADS+19b] states that the equilibrium measure of this dynamics is, indeed, the Gibbs measure on the square lattice. For all these dynamics, samples are drawn using the Propp-Wilson algorithm introduced above. Several values of J close to the critical value for the Ising model on the square lattice and the results obtained are summarized in Figures 6.10, 6.11, 6.12 and 6.13.
The data suggests that, for $q \geq 2.5$ the approximation provided by the shaken dynamics is quite good.

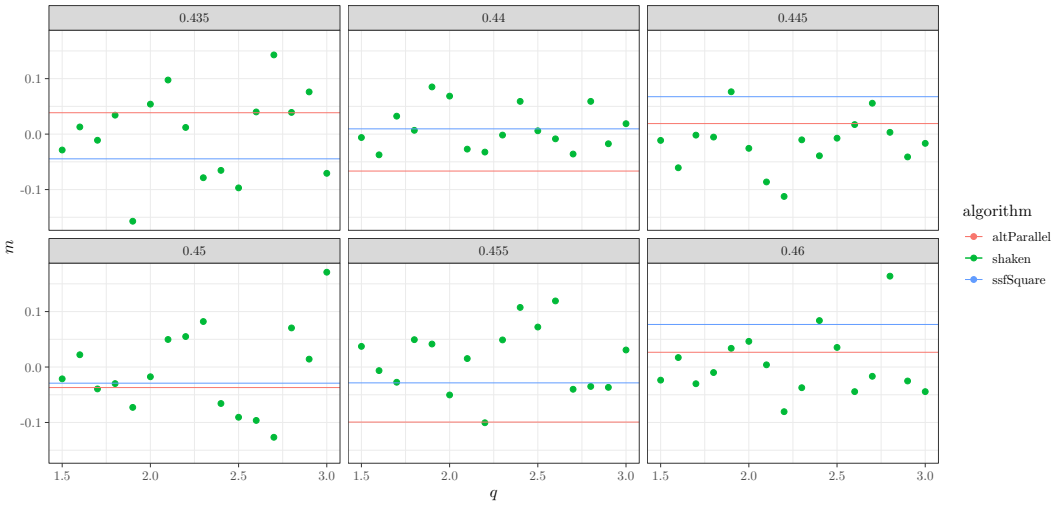


FIGURE 6.10: Sample average of the magnetization for several values of J

On the other hand, we also estimated the time required to approach the equilibrium distribution by comparing the coalesce time of the shaken dynamics with those of the two other reference dynamics. Also in this case the number of steps required by the single spin flip dynamics is renormalized with the volume of the box Λ. The result obtained are summarized in Figure 6.14. It is apparent that, though more flexible, the shaken dynamics becomes slower than "specialized" algorithms as the accuracy of the approximation increases.

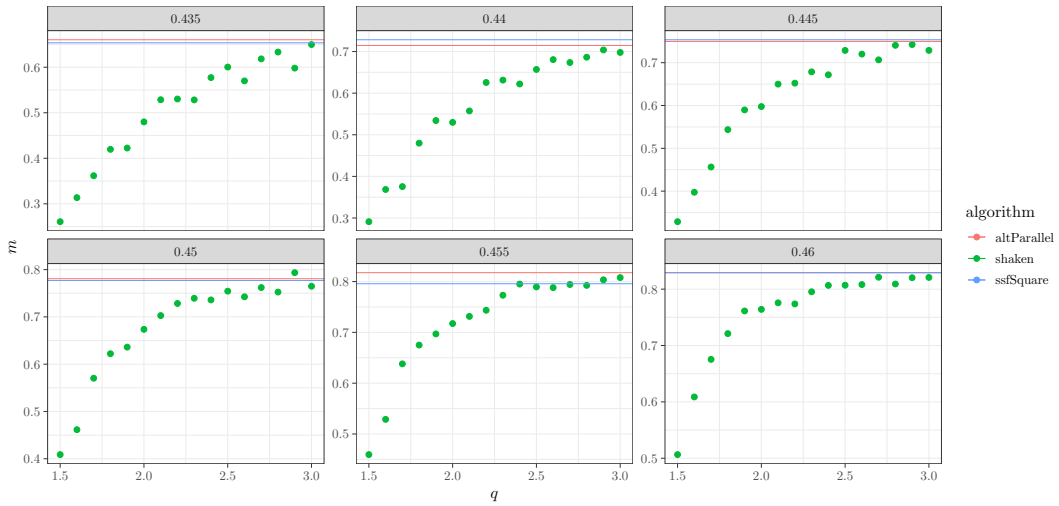


FIGURE 6.11: Sample standard deviation of the magnetization for several values of J

Parts (b) of Figures 6.15, 6.16 and 6.17 show configurations drawn from the equilibrium distribution of the alternate parallel on the hexagonal whereas parts (a) show the corresponding sub-configurations on the sublattice Λ_1. These sub-configurations are, therefore, drawn from the equilibrium distribution of the shaken dynamics. In Figure 6.15 it is possible to observe that the spins linked by a q-edge have almost always the same value. This is in good accordance with the fact that stationary measure of the shaken dynamics is close to the Gibbs measure for the Ising model on the square lattice. On the other side, Figure 6.17 is consistent with the fact that for q very small the equilibrium measure of the shaken dynamics tends to that of a collection of weakly dependent unidimensional Ising models.

6.3.3 Correlations

Theorem 2.4 in [ADS+19a] establishes that, if q is sufficiently small, $\pi(\sigma_{0,0}, \sigma_{\ell,\ell}) < \pi(\sigma_{0,\ell}, \sigma_{\ell,0})$ where π is the equilibrium measure of the shaken dynamics and σ is, therefore, a spin configuration living on Λ_1. In words, the theorem states that the SW-NE correlations are weaker than the NW-SE ones if the *self interaction* is weak. On the other hand, we expect that the SW-NE and the NW-SE correlations tend to be similar for large values of q, that is for those values of the pair (q, J) for which the equilibrium distribution of the shaken dynamics approaches the Gibbs measure of the Ising model on the square lattice.
We study the SW-NE and the NW-SE correlations as ℓ varies with Λ a 32×32 square box. The results are shown in Table 6.1.

All pairs (q, J) taken into account correspond to points of the q, J plane close to the critical curve $J_c(q)$. It is possible to observe that, as q decreases, the SW-NE correlations become, indeed, smaller than the NW-SE ones, whereas, for q large the two are quite similar. Further, if the pair (q, J) is below the critical curve the correlations decay quite rapidly. On the other hand, if (q, J) is above J_c the correlations are significant also for larger values of ℓ.

q	J	supercritical	direction	ℓ				
				1	2	4	8	16
0.05	1.7		NW-SE	0.821	0.765	0.7	0.481	0.425
			SW-NE	0.313	-0.063	-0.051	0.195	0.454
0.05	1.855	✓	NW-SE	0.916	0.852	0.767	0.704	0.726
			SW-NE	0.618	0.316	0.124	0.081	0.739
0.2	1.05		NW-SE	0.566	0.463	0.444	0.38	-0.016
			SW-NE	0.4	0.203	0.002	-0.037	0.041
0.2	1.175	✓	NW-SE	0.84	0.7	0.624	0.584	0.54
			SW-NE	0.54	0.54	0.52	0.5	0.685
0.4	0.82		NW-SE	0.65	0.507	0.462	0.356	0.426
			SW-NE	0.398	0.279	0.119	0.103	0.218
0.4	0.86	✓	NW-SE	0.68	0.644	0.541	0.679	0.538
			SW-NE	0.6	0.431	0.59	0.485	0.566
0.6	0.67		NW-SE	0.74	0.646	0.4	0.378	0.167
			SW-NE	0.622	0.401	0.36	0.321	0.283
0.6	0.7	✓	NW-SE	0.855	0.772	0.763	0.732	0.664
			SW-NE	0.654	0.677	0.61	0.593	0.578
0.65	0.65		NW-SE	0.701	0.472	0.477	0.503	0.475
			SW-NE	0.56	0.501	0.4	0.279	0.481
0.663	0.663	✓	NW-SE	0.749	0.646	0.6	0.544	0.477
			SW-NE	0.65	0.52	0.442	0.578	0.642
0.8	0.58		NW-SE	0.68	0.627	0.281	0.243	0.245
			SW-NE	0.609	0.522	0.307	0.444	0.433
0.8	0.61	✓	NW-SE	0.66	0.661	0.62	0.581	0.52
			SW-NE	0.74	0.52	0.581	0.52	0.524
1.0	0.52		NW-SE	0.581	0.56	0.258	-0.019	0.103
			SW-NE	0.541	0.299	0.341	0.221	0.04
1.0	0.55	✓	NW-SE	0.602	0.606	0.398	0.441	0.599
			SW-NE	0.58	0.58	0.561	0.54	0.532
2.5	0.43		NW-SE	0.462	0.456	0.27	0.194	0.164
			SW-NE	0.541	0.42	0.221	0.26	0.201
2.5	0.46	✓	NW-SE	0.658	0.701	0.74	0.701	0.699
			SW-NE	0.761	0.739	0.654	0.538	0.654

TABLE 6.1: Spin-spin correlations. The check-mark ✓in the *supercritical* column identifies pairs (q, J) above the critical curve J_c

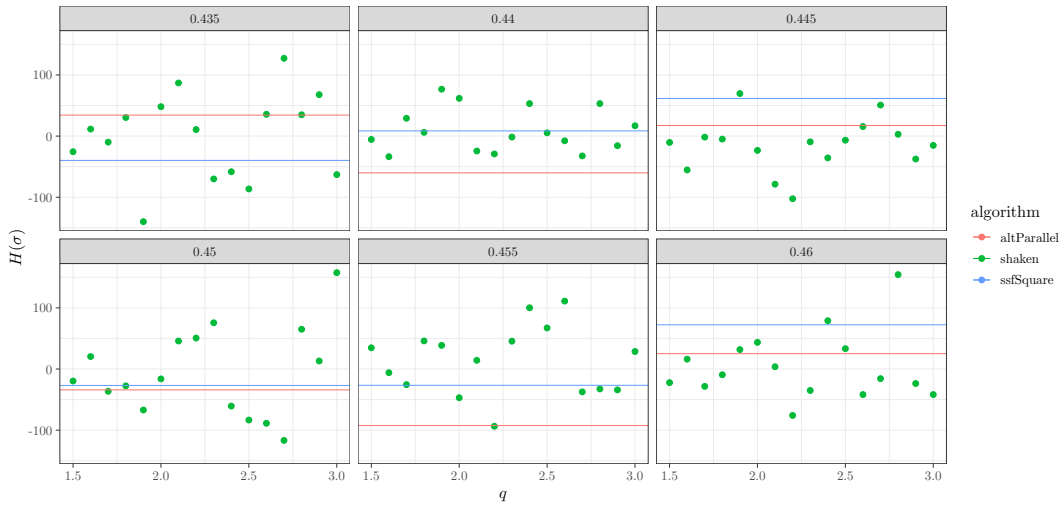


FIGURE 6.12: Sample average of the energy $H(\sigma)$ for several values of J

6.4 Implementation details

To approximate numerically the critical curve $J_c(q)$, we take samples for different values of J and q. The code used for the simulation written in Julia [BEKS14] and simulations are performed through 80 thread processors running, in parallel, the simulation on 80 couples of values (q, J) in the range of $(q, J) \in (0,2) \times (0,2)$. The Hamiltonian is defined on a square 200×200 lattice. Statistics are collected over 300,000 iterations. Figure 6.6 shows that the chosen simulation parameter is good enough to approximate the critical curve.
The elementary step of the shaken dynamics described in the previous section has been simulated by the Algorithm 12. A spin configuration is updated via a sequence of two similar half steps. The computation of the vector of local fields h that drives the transition probabilities of each spin is alternatively carried out using the functions `collectUR` and `collectDL` which determine the `up-right` and `down-left` contribution as in Equation (6.2.13).

Algorithm 10 collectUR

Input x_σ, J, q
Output f
1: $f \leftarrow J(\sigma_{x^\uparrow} + \sigma_{x^\rightarrow}) + q\sigma_x$
2: Return f

Algorithm 11 collectDL

Input x_σ, J, q
Output f
1: $f \leftarrow J(\sigma_{x^\downarrow} + \sigma_{x^\leftarrow}) + q\sigma_x$
2: Return f

The algorithm 12 is the complete update in two steps of the shaken dynamics, which is more general than the one used in [LS13]. The choice of collecting the statistics over 300,000 time steps (after a warm up time of 300,000 additional time steps)

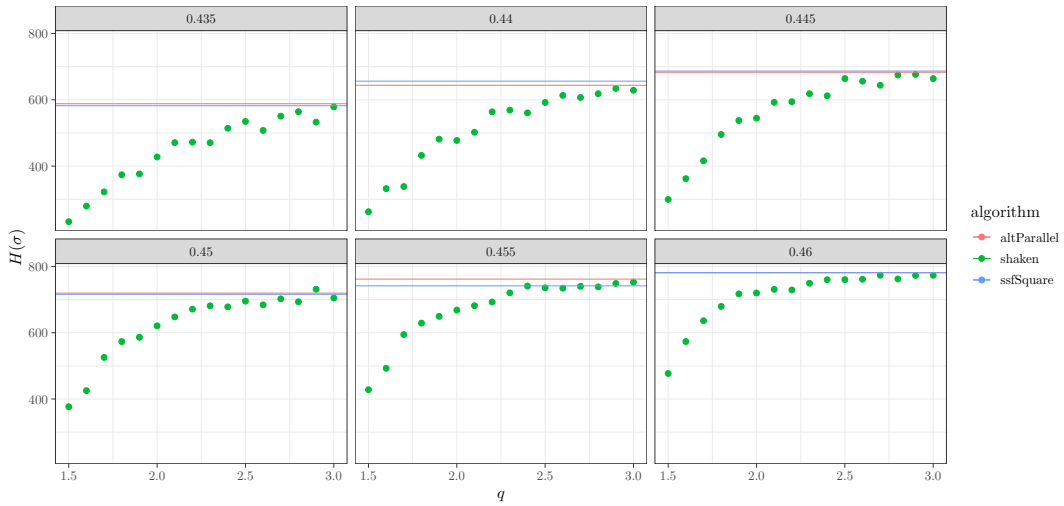


FIGURE 6.13: Sample standard deviation of the energy $H(\sigma)$ for several values of J

Algorithm 12 Shaken dynamics

Input initial spin configuration σ
Output updated spin configuration τ

for each x_σ **do**
  $h \leftarrow$ **collectUR**(x_σ, J, q)
  $p \leftarrow exp(h)/2\cosh(h)$
  if $rand() < p$ **then**
    $x_\tau \leftarrow 1$
  else
    $x_\tau \leftarrow -1$
  end if
end for
$\sigma \leftarrow \tau$
for each x_σ **do**
  $h \leftarrow$ **collectDL**(x_σ, J, q)
  $p \leftarrow exp(h)/2\cosh(h)$
  if $rand() < p$ **then**
    $x_\tau \leftarrow 1$
  else
    $x_\tau \leftarrow -1$
  end if
end for

turned out to be good enough, and the results show, unmistakably,the separation of the two phases (ordered and disordered).
We implemented the algorithm 12 in two parallel ways. A **CUDA** [2] implementation of a parallel heat bath for large dimension Lattice spin, and a **Julia** [BEKS14] implementation on a single CPU to be used on a multiprocessor systems (trivial parallel on a multi data input). Both have been optimized to handle our problem, and used

[2]Compute Unified Device Architecture, parallel platform and programming model to make use of the Graphic Processing Units general purpose computing simple and elegant.

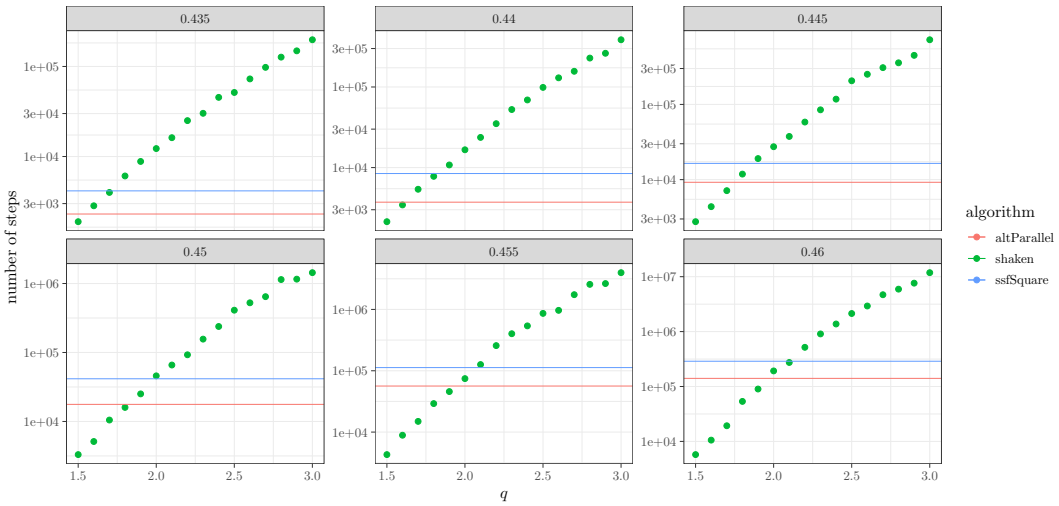
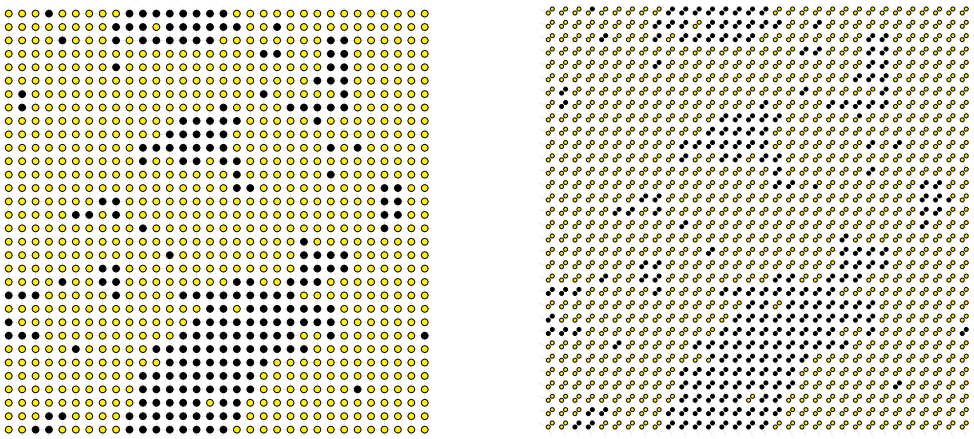


FIGURE 6.14: Sample average of the coalescence time (number of steps) for several values of J



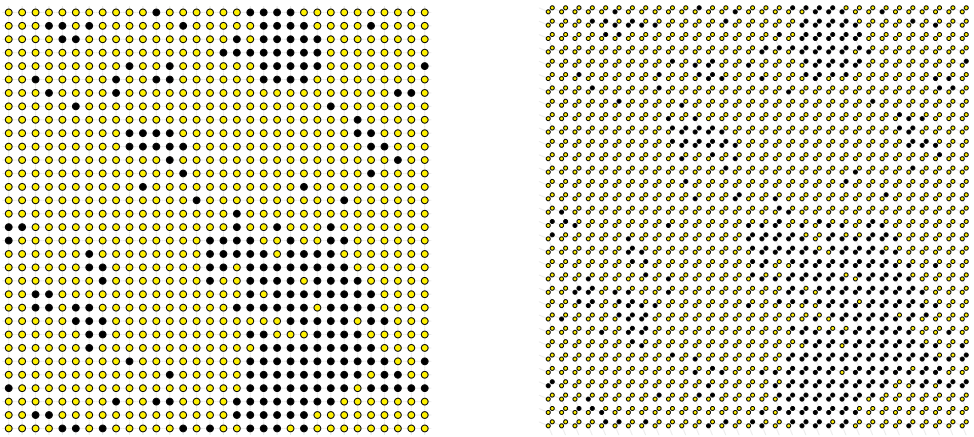
(a) (b)

FIGURE 6.15: $J = 0.44$, $q = 3.0$

to simulate the shaken dynamics of the PCA, a quasi-similar behavior was observed during our experiment.

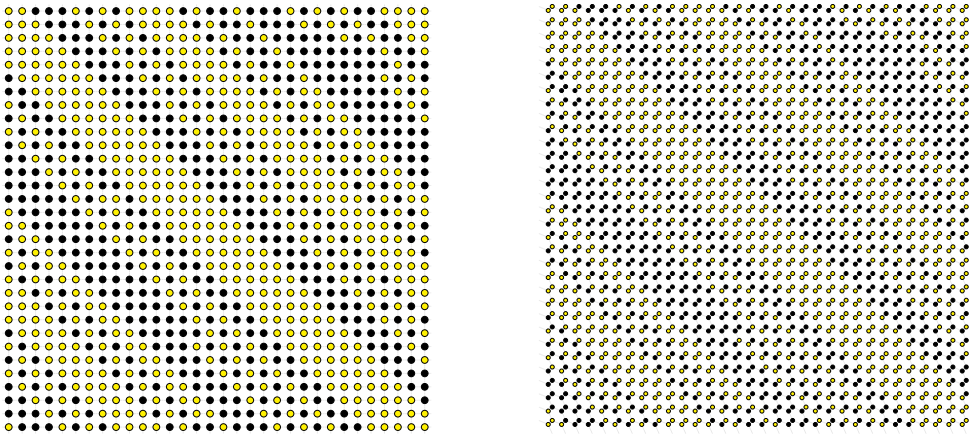
Parallel single-GPU code

The general heat bath procedure has been implemented on SIMD (Single Instruction Multiple Data) system. To optimize the code exploiting the CUDA memory architecture we implemented three kernels for the functions `collectUR`, `collectDL` and for updating the configuration. We used the default random generator from curand library.
The `collect` function computes the transition probabilities in the given direction. Each thread handles one spin on the lattice field. The principal use of the global memory is the four square spin lattice, the two configuration sigma (σ) and tau (τ), the fields which handles the Hamiltonian computation and the random-unit contains random uniform variables.
In our implementation all the operation are performed on register cache memory.



(a) (b)

FIGURE 6.16: $J = 0.6585$, $q = 0.6585$



(a) (b)

FIGURE 6.17: $J = 2.0$, $q = 0.03$

We did not use shared memory for the random-unit. The code was written to run on the Nvidia-GPU Tesla P100 using by 16GB video memory, using 4 matrices of dimension $L \times L$, two for the lattice spin field (single byte), and two for the collected fields and for the random uniform (four byte). All the matrices are allocated on the global memory.
For the management of the memory, before allocating the memory of the 4 matrices, the code used approximately 303 MB, leaving 15973.250000 MB. We used $2 * 4 * L * L + 2 * L * L$ bytes, but we can not go beyond 10^5 for this GPU.
The purpose of the CUDA implementation is to work on large dimension which allows us to observe the statistical behavior of the shaken dynamic, also for real time simulation.
Figure (6.18) shows a state of configuration captured at 60-th iteration on a simulation of the shaken dynamics for PCA lattice square with dimension 512×512, under the temperature $J = 0.99$ and the external field $q = 0.5$.

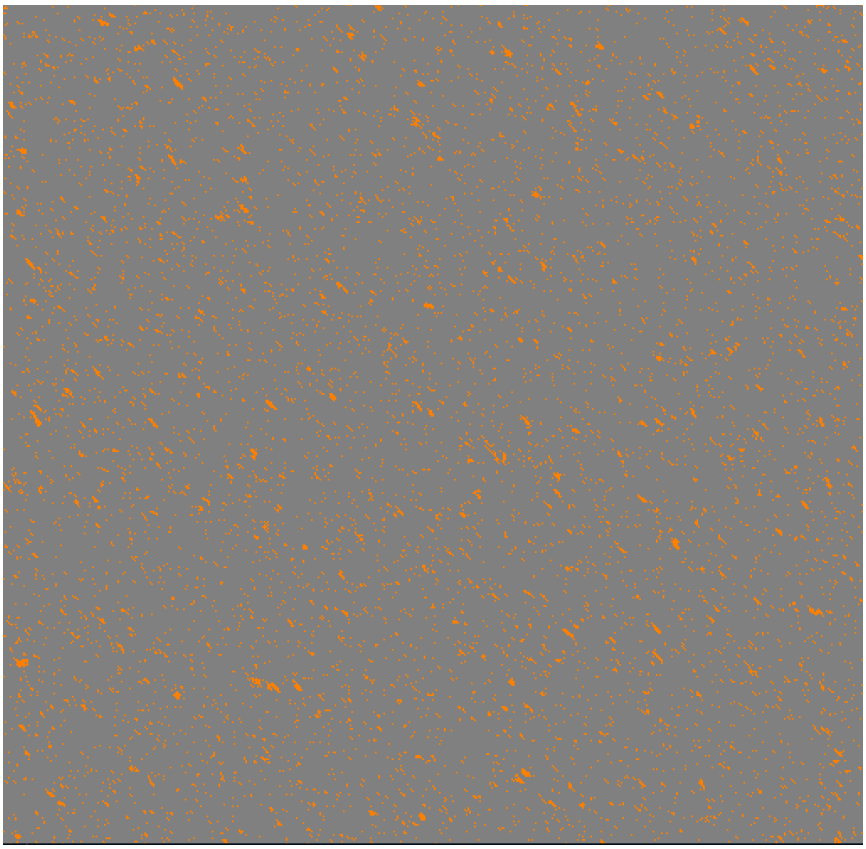


FIGURE 6.18: GPU sample: $L = 512$, $J = 0.99$, $q = 0.5$, iteration$= 60^{th}$

Benchmarking

To measure the performance of our GPU code it is not fair to compare it with the single-CPU implementation from Julia. We have implemented a serial version of the shaken dynamics in a lower level language, a captured sample for a square lattice spins of size 512×512 with $J = 0.99$ and $q = 0.5$ is given in Figure 6.19. For our simple measurement, we set the parameters $J = 0.44$ and $q = 0.66$ and to have more significant value we measure it in milliseconds. We compare the two implementation for different dimension L, hence the size of the square lattice which is L^2. For this benchmark (Figure 6.20), we used an Nvidia graphic card Tesla P-100 vs single core of the CPU Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz. We measure the time for one update execution. As we have mentioned the GPU memory is limited so that we did the experiment under this condition for the size of the square lattice spin.
We observe that the GPU is much faster than the CPU with a factor of 500 as far as the lattice size grows. We can see that the CPU time looks linear while the case for GPU is when the size is more than 2048×2048.

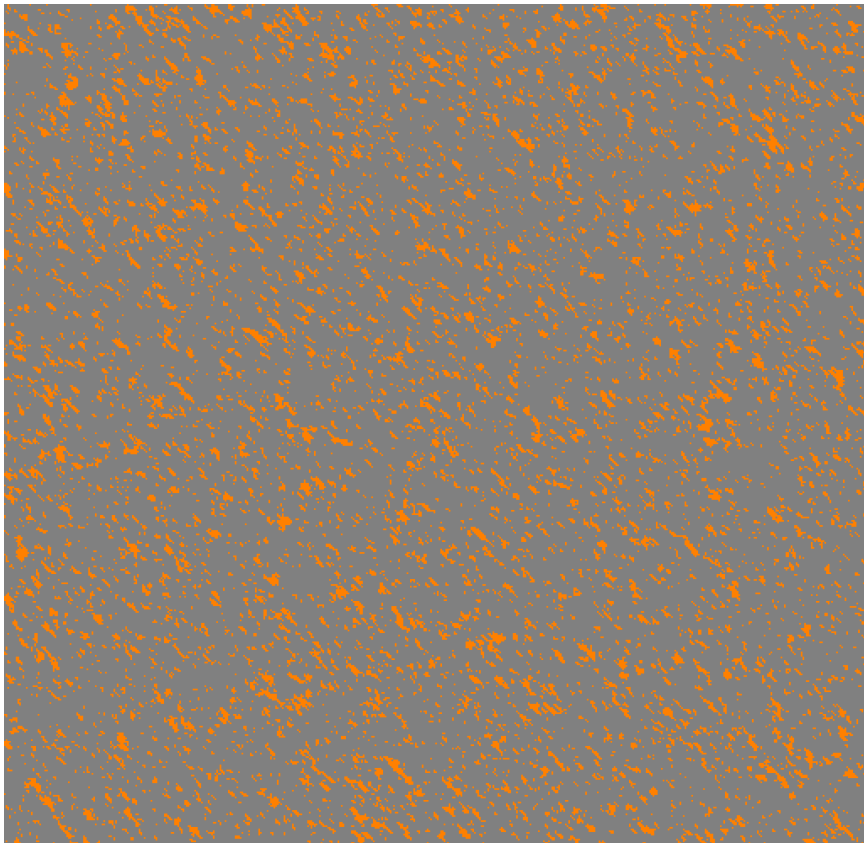


FIGURE 6.19: CPU sample: $L = 512$, $J = 0.99$, $q = 0.5$, iteration$= 60^{th}$

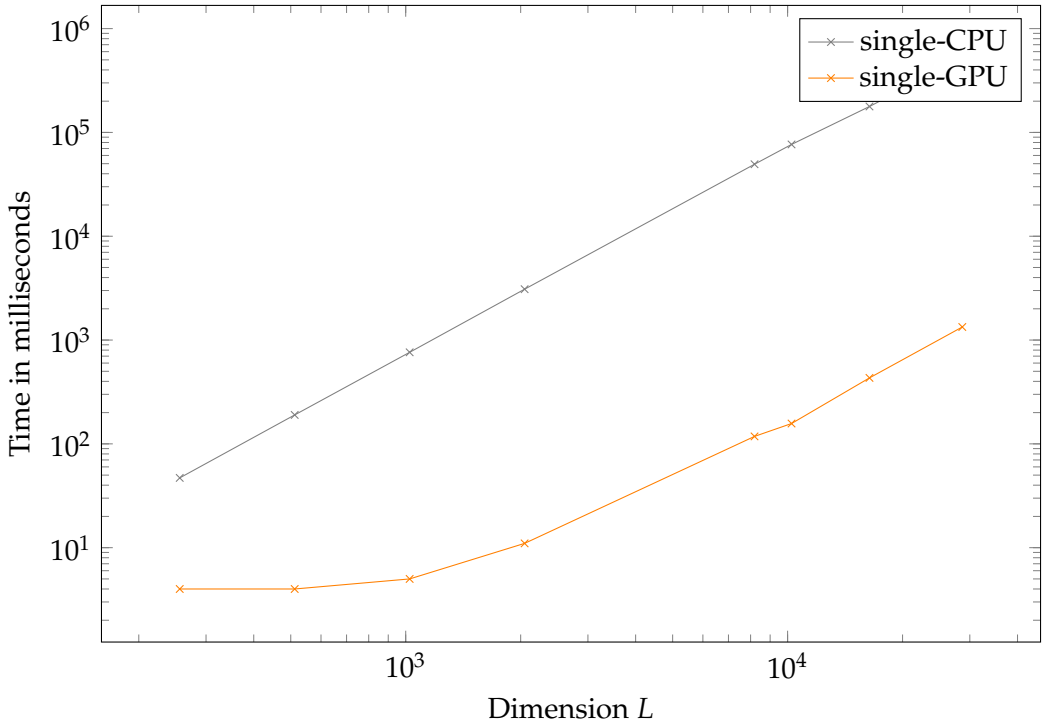


FIGURE 6.20: Running-time in function of the size L.

7. Summary

In the present work, we concentrated particularly on the two dimensional Ising model by means of Probabilistic Cellular Automata (PCA) [DSS12]. This model can be seen as a Markov Chain of a lattice square configuration spins, and studied its dynamics by an algorithmic approach.
We based our experiment on the dynamics and its simulation algorithm suggested in [ADS+19b], a new dynamics called *shaken dynamics*. We discussed the practical perspective of this dynamics especially its parallel implementation. We discuss an implementation of the algorithm on the GPU that can be used to simulate the shaken dynamics in real-time.
We retrieved numerically the critical curve of the PCA dynamics which has been found in [ADS+19a] using the *shaken dynamics*, by the help of the parallel implementation on a multi-core processors. Also we give indications of the convergence of the dynamics to its equilibrium distribution. In summary, the present work is a numerical exploration ot the results of the papers [ADS+19b, ADS+19a].

We have implemented all the codes in Julia [BEKS14]. The codes contains two classes of lattices squares and hexagonal with their methods, serial and parallel. This can be collected into a library for studying the 2D Ising model especially the shaken dynamics.

References

(ADS+19a) Valentina Apollonio, Roberto D'Autilia, Benedetto Scoppola, Elisabetta Scoppola, and Alessio Troiani, *Criticality of measures on 2-d Ising configurations: from square to hexagonal graphs*, arXiv e-prints (2019), arXiv:1906.02546.

(ADS+19b) _____, *Shaken dynamics for the 2d ising model*, arXiv e-prints (2019), arXiv:1904.06257.

(BEKS14) Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah, *Julia: A fresh approach to numerical computing*, CoRR **abs/1411.1607** (2014).

(CDC13) David Cimasoni and Hugo Duminil-Copin, *The critical temperature for the ising model on planar doubly periodic graphs*, Electronic Journal in Probability **18** (2013), no. 44, 1–18 (eng), ID: unige:30547.

(DNT19) Roberto D'Autilia, Louis Nantenaina Andrianaivo, and Alessio Troiani, *Parallel simulation of two–dimensional Ising models using Probabilistic Cellular Automata*, arXiv e-prints (2019), arXiv:1908.07341.

(DPSS15) Paolo Dai Pra, Benedetto Scoppola, and Elisabetta Scoppola, *Fast mixing for the low temperature 2d ising model through irreversible parallel dynamics*, Journal of Statistical Physics **159** (2015), no. 1, 1–20.

(DSS12) Paolo Dai Pra, Benedetto Scoppola, and Elisabetta Scoppola, *Sampling from a gibbs measure with pair interaction by means of pca*, Journal of Statistical Physics **149** (2012), no. 4, 722–737.

(Gal72) G. Gallavotti, *Instabilities and Phase Transitions in the Ising Model. A Review*, RIVISTA DEL NUOVO CIMENTO **2** (1972), 133–169.

(GG84) S. Geman and D. Geman, *Stochastic relaxation, gibbs distributions, and the bayesian restoration of images*, IEEE Transactions on Pattern Analysis and Machine Intelligence **PAMI-6** (1984), no. 6, 721–741.

(GKLM89) Sheldon Goldstein, Roelof Kuik, Joel L. Lebowitz, and Christian Maes, *From pca's to equilibrium systems and back*, Communications in Mathematical Physics **125** (1989), no. 1, 71–79.

(Gla63) Roy J. Glauber, *Time-dependent statistics of the Ising model*, j-J-MATH-PHYS **4** (1963), no. 2, 294–307.

(HÖ0) Olle Häggström, *Finite markov chains and algorithmic applications*, in London Mathematical Society Student Texts, Cambridge University Press, 2000.

(Ins) InsideHPC, *Articles and news on parallel programming and code modernization*, https://insidehpc.com/category/hpc-software/parallel-programming/, Accessed June, 2019.

(Isi25) E. Ising, *Beitrag zur Theorie des Ferromagnetismus*, Zeitschrift fur Physik **31** (1925), 253–258.

(Kot08) Jacques Kotze, *Introduction to Monte Carlo methods for an Ising Model of a Ferromagnet*, arXiv e-prints (2008), arXiv:0803.0217.

(**Lin12**) T. Lindvall, *Lectures on the coupling method*, Dover Books on Mathematics, Dover Publications, 2012.

(**LP17**) D. A. Levin and Y. Peres, *Markov chains and mixing times: Second edition*, American Mathematical Society, 2017.

(**LS13**) C. Lancia and B. Scoppola, *Equilibrium and Non-equilibrium Ising Models by Means of PCA*, Journal of Statistical Physics **153** (2013), 641–653.

(**MRR+53**) Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller, *Equation of state calculations by fast computing machines*, The Journal of Chemical Physics **21** (1953), no. 6, 1087–1092.

(**PW96**) James Gary Propp and David Bruce Wilson, *Exact sampling with coupled markov chains and applications to statistical mechanics*, Random Struct. Algorithms **9** (1996), no. 1-2, 223–252.

(**RC05**) Christian P. Robert and George Casella, *Monte carlo statistical methods (springer texts in statistics)*, Springer-Verlag, Berlin, Heidelberg, 2005.

(**Rue99**) D. Ruelle, *Statistical mechanics: Rigorous results*, World Scientific, 1999.

(**SW87**) Robert H. Swendsen and Jian-Sheng Wang, *Nonuniversal critical dynamics in monte carlo simulations*, Phys. Rev. Lett. **58** (1987), 86–88.

(**van41**) B. L. van der Waerden, *Die lange Reichweite der regelmäßigen Atomanordnung in Mischkristallen*, Zeitschrift fur Physik **118** (1941), 473–488.

(**Wil13**) Nicholas Wilt, *The CUDA handbook: a comprehensive guide to GPU programming*, Addison-Wesley, Upper Saddle River, NJ, 2013 (en).

(**Wol89**) Ulli Wolff, *Collective monte carlo updating for spin systems*, Phys. Rev. Lett. **62** (1989), 361–364.

Appendices

7.A Codes for the shaken dynamics

7.A.1 Julia implementation

The main purpose of the **Julia** [BEKS14] implementation is to provide a library that can be used in statistical mechanics for academic purpose. Since Julia is a high level language that have a comparable running time with low level language, and it facilitate the use of high performance, we would like to exploit this to solve algorithmic problem like the one we faced in this project.
At the moment, the code is under development, it contains the definition of the Lattice square and hexagonal with all the methods that have used doing this project, for instance the coupling from the past [PW96].

7.A.2 Cuda implementation

We have implemented the shaken dynamics in cuda to be used as a library for the above julia project. In our experiment, we have the Nvidia-GPU Tesla P100 occupied by 16GB Video memory. For this particular device, we give some detail on the parameter used for our simulation.

We have presented in the benchmark that we could go between $L = 10^4$ and $L = 10^5$. Indeed, on the global memory we have to allocate 4 matrices of size $L \times L$ such that 2 is for the configuration σ and τ which can be represented by a single byte, the two other is for the fields computation and the random uniform that are represented by a four byte (double precision). In total, we used $2 \times 4 \times L \times L + 2 \times L \times L$ bytes. A simple computation leads to us that we can not go beyond 10^5.
For the occupancy, the two collect kernel functions requires 28 register, the thread-block size was deduced from the Figure 7.A.1 while the update kernel requires 38 registers and the warp-occupancy in function of the thread-block size is presented Figure 7.A.2.

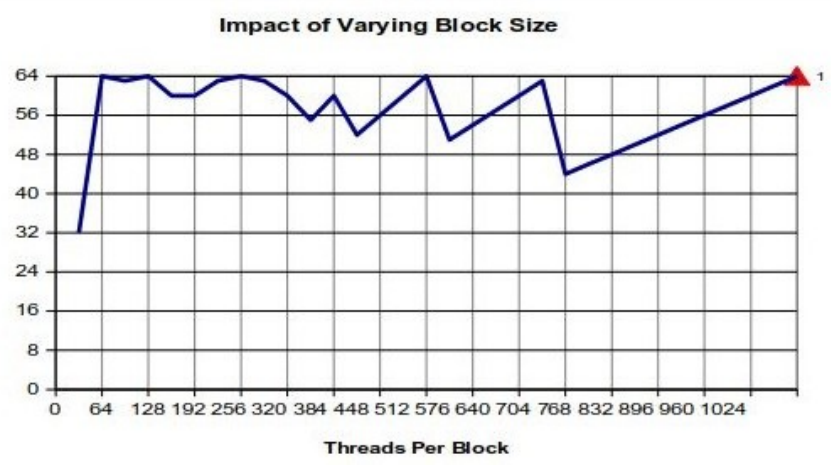


FIGURE 7.A.1: warps occupancy for the `collectUR` and `collectDL`.

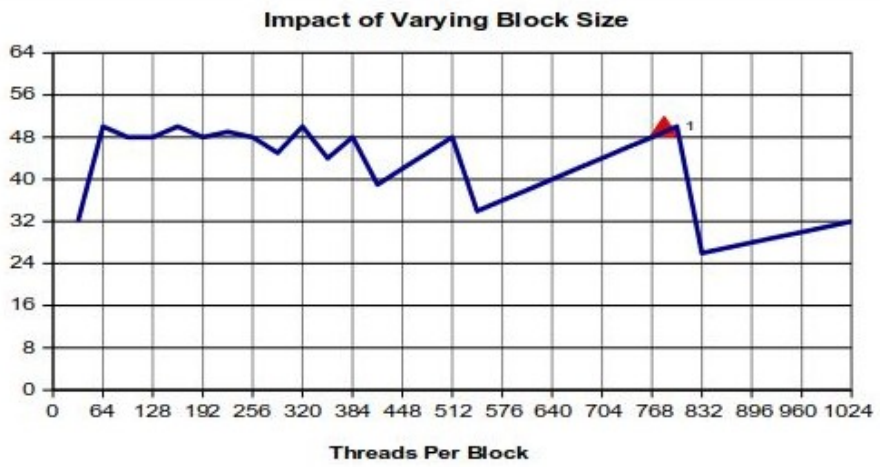


FIGURE 7.A.2: warps occupancy for the *update*.

The three kernels function is given in the following source codes,

```
/*
 ============================================================================
 Name        : kernel.cuh
 Author      : Louis
 Version     :
 Copyright   :
 Description : kernel shaken dynamics
 ============================================================================
 */

 __global__ void collectDL(double *fields,
                           int8_t *sigma,
                           double J,
                           double q,
                           double lambda,
                           int length)
        {

        int x_0 = threadIdx.x + blockDim.x*blockIdx.x;
        int y_0 = threadIdx.y + blockDim.y*blockIdx.y;

        int stride_col = blockDim.x*gridDim.x;
        int stride_row = blockDim.y*gridDim.y;

        for(int y = y_0; y < length; y += stride_row)
                {
                for(int x = x_0; x < length; x += stride_col)
                        {
                        const double reg_c =
                                (double) *(sigma + Index2D(y, x, length));
                        const double reg_d =
                                (double) *(sigma + Index2D(Modn(y+1,length)
                                                ,x,length));
                        const double reg_l =
```

```
                                (double) *(sigma + Index2D(y
                                        ,Modn(x-1,length),length));

                        * (fields + Index2D(y, x, length)) =
                                J*(reg_d + reg_l) + q*(reg_c) + lambda;
                        }
                }
        }

__global__ void collectUR(double *fields,
                        int8_t *sigma,
                        double J,
                        double q,
                        double lambda,
                        int length)
        {

        int x_0 = threadIdx.x + blockDim.x*blockIdx.x;
        int y_0 = threadIdx.y + blockDim.y*blockIdx.y;

        int stride_col = blockDim.x*gridDim.x;
        int stride_row = blockDim.y*gridDim.y;

        for(int y = y_0; y < length; y += stride_row)
                {
                for(int x = x_0; x < length; x += stride_col)
                        {
                        const double reg_c =
                                (double) *(sigma + Index2D(y, x, length));
                        const double reg_u =
                                (double) *(sigma + Index2D(Modn(y - 1,length),
                                        x , length));
                        const double reg_r =
                                (double) *(sigma + Index2D(y,
                                        Modn(x+1,length),length));

                        * (fields + Index2D(y, x, length)) =
                                J*(reg_u + reg_r) + q*(reg_c) + lambda;
                        }
                }
        }
__global__ void update_config(int8_t *tau,
                                double *fields,
                                double *randunit,
                                int length)
        {
        int x_0 = threadIdx.x + blockDim.x*blockIdx.x;
        int y_0 = threadIdx.y + blockDim.y*blockIdx.y;

        int stride_col = blockDim.x*gridDim.x;
        int stride_row = blockDim.y*gridDim.y;
```

```
        for(int y = y_0; y < length; y += stride_row)
                {
                for(int x = x_0; x < length; x += stride_col)
                        {
                        const double reg_field = * (fields +
                                Index2D(y,x,length));
                        * (tau + Index2D(y, x ,length)) =
                                        ( *(randunit + Index2D(y, x ,length))
                                        exp(reg_field)/(2*cosh(reg_field))) ?
1 : -1;
                        }
                }

        }
```

Part III

Artificial Intelligence

8. Introduction

The term Artificial Intelligence (AI) describes the possible "intelligent behavior" of machines, in contrast to the natural intelligence shown by humans and other animals. It was founded as an academic discipline in 1956 [KH19]. Computer science often use this word to mean the teaching of an intelligent agents (training a devices to be able to perform a task by maximizing the change). Machine learning is a subset of the AI mainly based on statistics and probabilistic techniques. This field is interdisciplinary in terms of the techniques and their applications.

On the other hand, *Computer Vision* is a field in computer science which particularly deals with images and video. It utilizes many others academics disciplinary such as, mathematics, physics and engineering. Its main task is to apply several methods such as machine learning algorithms to be able to interpret and understand digital images. This lead to an automatic system which made the computer vision to be one of the principal component of AI.

In this work, we apply the image classification technique on monuments and architectural images. Our goal is to integrate the deep learning technique used in computer vision on urban data. The idea is to provide a facility for the user to gather information from an inaccessible environment. This information can be seen as a set of linked database in which we are able to put them into one system.
A typical method to consume these data is to identify them by an "id" or a "keyword", our idea is to use a pixels images of an object. Using a mobile device, one can takes the picture of the desired object and process the query of all the related information of the object.
The raise of the new technology, especially in mobile device, allows us to perform this task easily. The integration of a new device component that make the process of machine learning algorithm fast and easy to use.
The methods proposed here is the exploitation of the capacity to learn from experience, the model can interpret an input object and assign it to a category. Formally, this corresponds to the construction of a function from a sample of (input, output) pairs. In general, we have multidimensional data and we also assume that the function to construct is a multivariate non-linear function. We use a type of artificial neural network, Convolutional Neural Networks (CNNs) which is largely applied to deal with images input [RW17]. This kind of network are currently applied in many computer vision tasks and performed better than the traditional techniques, for instance image analysis for medicine and biology [HPQ+18]. This technique takes advantage from the fast development of the new technology in many issues. By training the network on modern graphic card dedicated for high performance computing and use the inference model on mobile devices.
This work is a contribution of the candidate on the Shazarch project [ADP19] which is a complete mobile application on iOS [Pal19].
We plan this part as follows: In *Chapter 9*, we introduce a general background in Artificial Neural Networks and Deep Learning that motivate the details steps of the methodology we have processed during this project. And we conclude with a summary and an overview of a working in progress that will be the extension of this task in *Chapter 10*.

9. The project : Image classification

This project is a complete mobile application that has been initiated by the ***Shazarch Project***, it was an architecture initiative which are based on the **Foro Romano**. The main goal is to facilitate the navigation of people (tourists) visiting monuments in the Museum of Roma. Hundreds of monuments are difficult to remembered or recognized, apparently they look similar. The full application contains a model which recognize the monument, then it gives every information about the object ,and also links it into a 3D model that is very useful to visualize the architectural structure of the object [Pal19]. My contribution of this project is the first part, building the model which is used to recognize the monument from its picture.

9.1 Artificial Neural network (ANN)

Inspired from the biological properties of neurons, McCulloch and Pitts [MP43] introduced the *perceptron*. This is considered the first conceptual model of an artificial neural network. As well as the human nervous systems neural networks, the ANN is intended to be a block of perceptron (neurons). A neuron can be seen as cell living in a network of cells, it receives inputs and process it to generates an output. Amit [Ami89] gave a detailed description of this concept (from the biological behavior of the nervous systems, how the neurons transmits signal (information), where the signals are processed and outputted). To summarize, the neurons are represented as *processing unit* (body for the neurons). Several input signals are connected to the processing unit to be processed and outputted (this can be an activate signal or not). This can be represented as in figure 9.1.

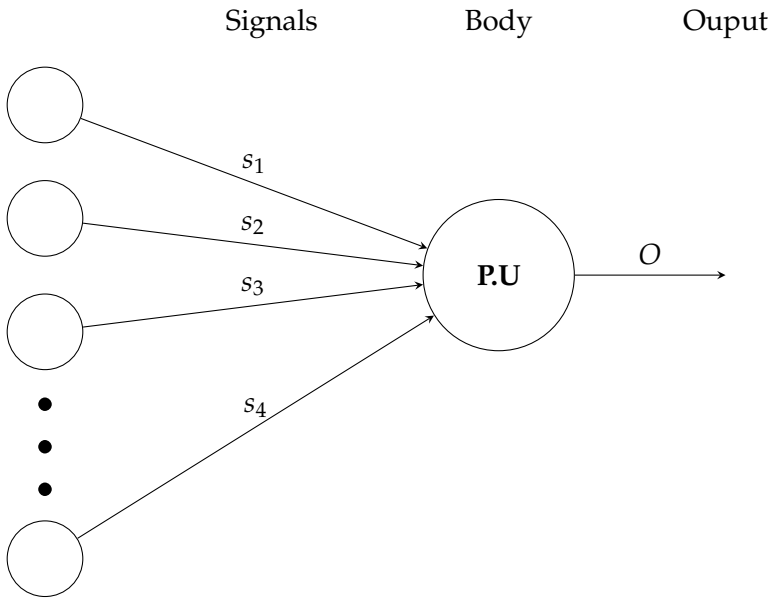


FIGURE 9.1: Perceptron. In the left, the sets of input signals $(s_0, \dots s_n)$ from other perceptron which are connected into the processing unit. On the right, the outputs $O = \{y_0, y_1\}$.

As we presented on figure 9.1, in a neural network the neurons interacts by sending signals. In other words, the signal activates the other neurons. The sets of outputs

O can be seen as a conditional value depending on the result from **P.U**. Formally, let P be the processing application then the **P.U** is a function given by $P(s_1, \ldots, s_4)$. We define T be a *threshold* so the output is given by

$$y = \begin{cases} y_0 & \text{if } output(x) > T \\ y_1 & \text{o/w} \end{cases}$$

A little modification is needed to redefine this function to be an *Activation*. There are several activation used in perceptron (ANNs), for instance the **ReLU**[1], a non linear activation function which is the closest to the biological property of neurones, that means it send an activate signal when it is under the threshold T.

9.1.1 Universal approximation theorem

An artificial neural network is a block of perceptrons, which can be represented as a graph. The edges which connects the inputs signal to be processed into other perceptron (edge) represents the *weight*, We assigned a *bias* each perceptron (node) in which will replace the role of threshold.

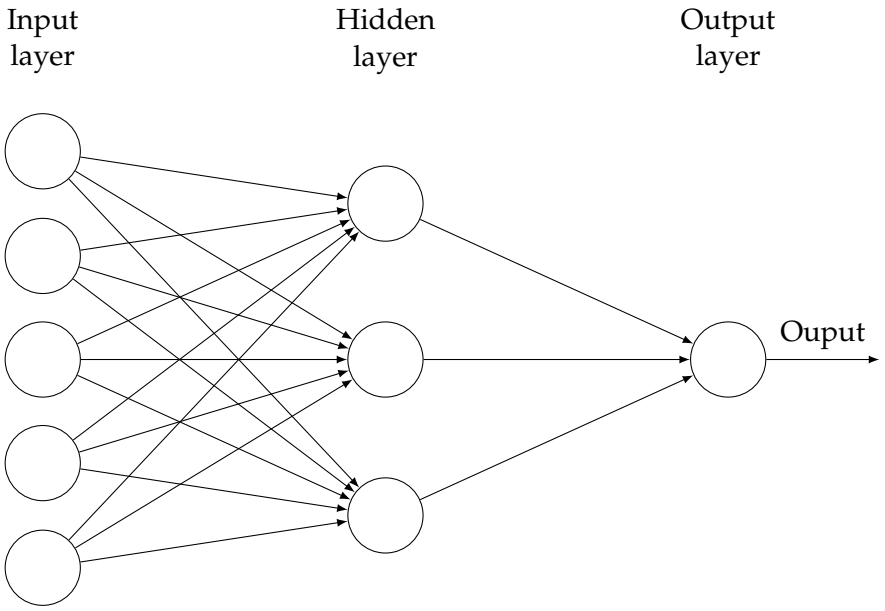


FIGURE 9.2: Artificial neural network.

Figure 9.2 presents a typical ANN with one hidden layer. All the edges possess a weight that we call w those which connect input to the hidden layer and v for hidden to output layer. The input nodes represent the assigned parameters, that means each node takes one coordinate of the input. The nodes in the hidden layers possesses each a bias that we have discussed before.
According to this configuration, the above ANN can be expressed as

$$\text{Output} = \sum_{i=1}^{3} v_i \times \varphi(w_i \times x + b_i)$$

[1]Rectified Linear Unit $f(x) = \max(0, x)$

This can be generalized by considering a $k > 0$ processing units in the hidden layers, so an ANN can be expressed as

$$F(x) = \sum_{i=1}^{k} v_i \times \varphi(w_i \times x + b_i)$$

here φ is the *activation function*. The *activation function* plays an important rule both in theory (for the demonstration) and in practice (fine tuning the ANN).

Let $X = (x_1, \ldots, x_n) \in \mathbb{R}^n$ and $Y = (y_1, \ldots, y_m) \in \mathbb{R}^m$. Let D be a set of data, we want to find a function

$$\begin{array}{rccc} f: & \mathbb{R}^n & \longrightarrow & \mathbb{R}^m \\ & \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} & \longmapsto & \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \end{array}$$

Formally, we can find a compact $\mathbb{K}$ subset of $\mathbb{R}^n$ containing the finite dataset D. The goal is to find a function $f \in C(\mathbb{K})$, here $C(\mathbb{K})$ is the set of continuous function on K.
Let define

$$\sum_n \varphi = \left\{ F : X \longrightarrow \sum_{i=1}^{k} v_i \times \varphi(w_i \times x + b_i)\,,\ w_i \in \mathbb{R}^n\,,\ b_i, y_i \in \mathbb{R}\,,\ k \,\text{finite} \right\}$$

the set of all the possible output of the ANN.

Theorem 9.1.2 (Universal Approximation Theorem). *$\sum_n \varphi$ is uniformally dense in $C(\mathbb{K})$.*

Remarks.

- The activation function φ needs the following properties
  - Continuous, non-constant, bounded, monotonically increasing.
  - $\varphi \in L^{\infty}_{Loc}$.
  - non-polynomial.
- There are many approaches to prove this theorem, which is related to its application on ANN, for instance Cybenko in [Cyb89] and Kurt Hornick in [Hor91].

9.1.3 ANN in application

The training of a ANNS is the process of finding a proper approximation of the given function f by means of an artificial neural network. We use classical calculus to reduce the amount of error $E = X - F(X)$ the difference between the desired output with the ANN predicted output. This technique called *Backpropagation*. It has to be utilized carefully to obtain algorithms which learn in quick and good way.

To have a clear pictures of this procedure, let X, Y and B be a real vector, B represent the bias, and the weight can be expressed as a matrix

$$W = \begin{pmatrix} w_{1,1} & \cdots & w_{1,c} \\ \vdots & \ddots & \\ w_{r,1} & \cdots & w_{r,c} \end{pmatrix}$$

such that $w_{r,c}$ is the weight connecting the node r to c. This representation shows that all the operation in ANN are between tensors. Each layer has its own output, we need to fix one layer say l. The result can be applied for all the other layers. The output on the layer l is given by

$$y_i^l = \varphi\left(\sum_k w_{i,k} \times y_k^{l-1} + b_i\right)$$

here k runs over the number of neurons at layers $l-1$. The goal is to minimize the value

$$G(W,B) = \frac{1}{2N}\sum_{i=1}^{N}(y_i - y_i^l)^2 \tag{9.1.1}$$

where N is the cardinality of the dataset. We can deduce from this the amount of the error ΔW and ΔB affected by W and B respectively.
Once computed $\Delta W, \Delta B$, we can update the value of weight and bias over the ANNs. The terms *backpropagation* means that we always start this process from the last layers and go backward for the update of each previous layer (due to the fact that only the last layer has the desired output from the dataset). Notices that the equation 9.1.1 suggest to compute its value over all the dataset. Mostly this is very large and computationally infeasible. In practice, the update is done on a random batch of the data, this technique is refereed as the **Stochastic Gradient Descent** (SGD). This procedure is repeated until the equation 9.1.1 is minimized, the ANN learns fast if its converges quickly to zero.

9.1.4 Deep neural network

From theorem 9.1.2, a single hidden layer ANN can approximate any continuous function. This kind of ANN is usually referred as **feed forward neural network**. We also notice that the architecture of the feed forward neural network is a very important challenge in Machine learning. The universal approximation theorem guaranties the existence of the the ANN which can approximate but does not provide any methods neither an indication on how they ANN are constructed.

The input signals which connects to all the processing unit in the hidden layer and each unit output the result after applying the activation function. There are several activation function that could be used, for instance *RELU, sigmoid, tanh* but the choice depends on the requirement of the problem. From the hidden layer, the signals are feed to the last layer and again output the result after applying an activation function. For **classification problem** it is preferable to use the *softmax*.

On the other hand, there is no preference to easily decide the number of processing unit in the hidden layer. However, to have a better approximation, there is an exponential approximation bound with respect to the size of input given in [Bar93].

This means that the number of the variables *weight* and *bias* which controls the error function (equation 9.1.1) is very large. To omit this we may break down this one block of perceptron into many layers, this reduces the degree of freedom of the error and helps the neural network to learn correctly. The number of the new hidden layer is referred as the *depth* of the ANN and a such ANNs are called **Deep feed forward neural network**. This was first used in Hiton paper [SH08] and has been proved that it has a good approximation of the desired function. In [Mon13], it is estimated the number of the hidden layers (the depth of the ANN) with respect to the input size although the better choice is still obtained via experimentation. This new architecture is then used for machine learning problem and referred as **Deep Learning** technique.
Deep neural network can be seen as a composition of function, i.e each output layer become the input of the next layers, therefore the backpropagation algorithm can be generalized.

9.1.5 Convolutional neural network

The seep neural network have an extensive applications area in Machine Learning, one of its application is classification, in particular image recognition. The idea is by extracting the features from the three layers RGB (Read, Green, Blue) pixels of an image to predict its description. This technique has been an important subject as it can be extended in many field, for instance the popular modern technique used in image computer vision can be seen as an extension of a such techniques.
The *convolution neural networks* (CNN) is a class of deep neural networks. This method is used in traditional image processing, signal processing, and is based on the mathematical operation of convolution defined by

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(x)g(t-x)dx \tag{9.1.2}$$

where f and g are function well defined on $(-\infty, +\infty)$.
The application of this operation in signal processing is to minimize the noise on a given signal, f represents the signal and g a probability distribution. Equation (9.1.2) can be given in a discrete time.

$$(f * g)[n] = \sum_{m=-\infty}^{+\infty} f[m]g[n-m] \tag{9.1.3}$$

This is often used in practice. For instance in image processing, represents f the pixels images and g is the filter (kernel) to extract some particular features (edge, etc). This can be defined for any dimensional data input f as the case for color RGB images (which is 3D), the operation can be illustrated as in Figure 9.3.
Notice that in the 2D convolution in Figure 9.3, the dimension of the output is reduced. In fact, the dimension can be kept by using a *zero-padded* on the boundary of the input depending on the shape of the kernel. This make sense in the operation as for signal one can assume that it is zero at initial and at the end.
Figure 9.3 can be expressed as follows

$$(I * K)[i, j] = \sum_{r=0}^{k-1} \sum_{c=0}^{k-1} I[i+r, j+c]K[r, c] \tag{9.1.4}$$

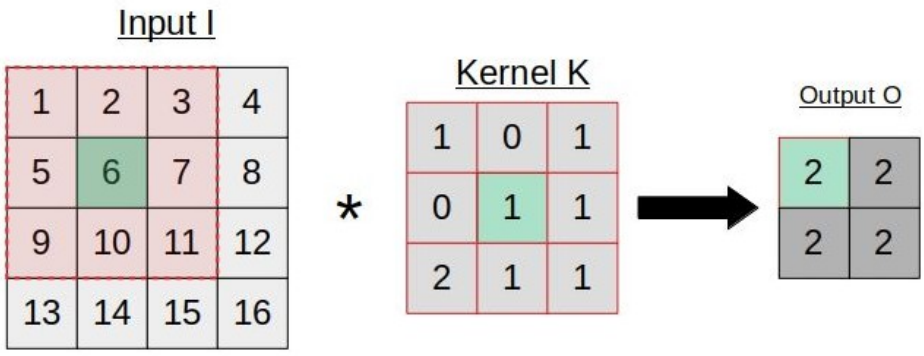


FIGURE 9.3: *2D Convolution operation.*

Inspiring from this operation, the use of discrete convolution in a neural network was suggested in [Lec89]. By looking closely to the structure of the convolutional layer we have the property of *sharing weights*, unlike the feed forward neural network where each neuron has separate weight. This technique reduces the number of parameters to be trained (space and time complexity). In practice, convolutional layer is composed by the convolution operation, followed by a non-linear function ReLU and a pooling operation (down sampling). The ReLU can be seen as a cancellation of negative values over the output where the pooling is to reduce the dimension since one usually uses it right before the fully connected layer.
There are few hyper-parameters that we have to provide to characterize the convolutional layer, to control the shape of the output in function of the input:

- **Depth** : is the number of kernels we want to use in the convolutional layers. One assume that each kernel extracts different features. For example, in image processing it can be seen as an extraction of the edges shared into piece of kernels. Referring to the feed forward neural network this is the activated neurons by the input pixels.
- **Stride** : is the amount of step the kernel slides. In Figure 9.3, we have a stride 1.
- **Zero-padding** : is the number of padded zero to control the output size.

For an input of size W, kernel of size K of stride S and padding P, the dimension of the output can be computed as

$$\frac{W - K + 2P}{S} + 1$$

In image processing, we are dealing with three layers pixels, say of shape $W_i \times H_i \times C_i$. Let K be the kernel of shape $K_w \times K_h \times d$, S and P are the stride and zero-padding. The output shape is given by

$$W_2 \times H_2 \times C_2$$

where

$$\begin{aligned} W_2 &= \frac{W_1 - K_w + 2P}{S} + 1 \\ H_2 &= \frac{H_1 - K_h + 2P}{S} + 1 \\ C_2 &= d \end{aligned}$$

Training a CNNs is tuning the kernel parameters, similar strategy as in a feed-forward networks the *backpropagation* is used.

Depth separable convolution layers [KGC17]

We explore a specific type of convolutional layer that are often used to optimize the operation on the neural network. The use of convolutional layers reduces the space complexity with respect to the feed forward neural networks especially for large dimensional input data. One can observe that the convolution operation is similar to a tensor product, a sliding dot product.
Some applications of deep learning requires less operation, for example the inference used in a mobile device, Inspired by the *spatial separable convolution* technique used in image processing, that is a division of the kernel into two smaller kernels of size. One obtain a computational complexity reduction but only applied when the kernel can be factorized as in the case for color image processing. The *depth separable convolution layers* is used when the input of the convolution has a depth dimension. The operation treats the spatial and depth dimension of the inputs by factorizing the normal convolutional layer into two convolution:

- *depth-wise convolution*: it can be seen as normal convolution but keeps the depth of the input. Each kernel slides over one layer of the input.
- *point-wise convolution*: after the depth-wise convolution, we have a spatial reduction while the depth is kept. We want to obtain the same number of output as in the normal convolution operation (depth d of the kernel), we apply the point-wise 1×1 as much as the number of the kernels.

For instance, we illustrate this operation by considering an input image of shape $224 \times 224 \times 3$. For a kernel of shape $3 \times 3 \times 3 \times 32$, the normal convolution outputs $112 \times 112 \times 32$ for stride 2.
For the depth separable convolution, we use a depth-wise convolution $3 \times 3 \times 1$ within stride 2 to obtain $112 \times 112 \times 3$, for an output which have the same dimension as the normal convolution we use a point-wise convolution of shape $1 \times 1 \times 3 \times 32$.

Remarks.

- The point-wise convolution is only relevant when the input have a third dimension (depth).
- The number of operation is indeed reduced in the depth separable convolution compared to the normal convolution. We can observe that in the above example,
  1. *normal convolution*: We have 32 kernels of size $3 \times 3 \times 3$ that are applied 112×112 times. The total operation is given by

$$32 \times 3 \times 3 \times 3 \times 112 \times 112 = 10838016$$

2. *depth- separable convolution*: We have 3 of $3 \times 3 \times 1$ depth-wise kernels sliding on 112×112. The operation performed is equal to $3 \times 3 \times 3 \times 112 \times 112$. For the point-wise, we have 32 of $1 \times 1 \times 3$ kernels which moves over the 112×112. We have an operation equals to $32 \times 1 \times 1 \times 3 \times 112 \times 112$. The total operation is the sum of these two number, given by
$$(3 \times 3 + 32)(3 \times 112 \times 112) = 1542912$$

The purpose of the CNNs is to reduce the number of trained parameters in the networks. This is often used when the dimension of the input is very large. The general structure of CNNs can be seen as a sequence of convolutional layers and one dense layer. The role of the sequence of convolutional layers is a coordinate transformation of the input of the fully connected layer at the end of the network. This can be illustrated in Figure 9.4.

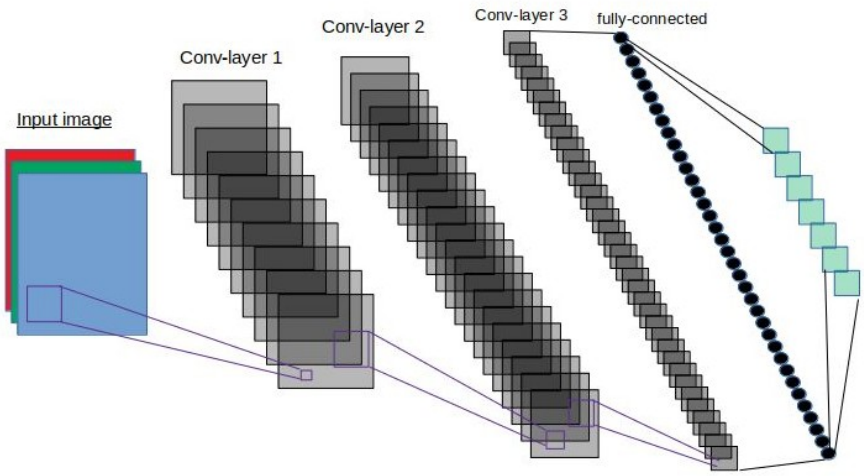


FIGURE 9.4: An general structure of a deep CNNs in image processing.

9.2 Methodology description

In this section, we give a details of the steps and methodology we have followed through this project.

9.2.1 Data preparation

We started by preparing the dataset for the training of the model.

- ***Data collection***
  For a given lists of monuments, we collect the pictures on the fields. We try to get all the possible angles and access of the objects, using mobile phone camera and a normal camera.
  Once the pictures of each monuments are collected, we perform a modification by hand, its aim is to let the model learn correctly on a difficult data (ex: monument with other monuments in the background).

- ***Data augmentation***
  All the pictures we have collected are from the accessible location, per each classes we possessed approximately 15 or 25 pictures. In our experiment, we estimated at least 500 pictures for each monuments are needed to train the model (400 train, 100 validation). Of course, this problem can be handled during the training. For instance Keras [C+15] offers a function that generate random data during the training but we decided to do this separately. Indeed, as the black-box generator performs a random generation so that we are not sure if the data generated is not duplicated and may cause an overfitting. Also as an experiment, we want to understand the type of data augmentation technique which is dedicated for our task. For this purpose, we use the following operation for the data augmentation:
  - Rotation (clockwise and counter-clockwise as we work on a small range of angle).
  - Crop (this is used by given an estimation of the area occupied by the monuments in the pictures)
  - Flip (top-bottom and left-right)
  - Distortion (simple and Gaussian)
  - Zoom
  - Histogram equalization
  - Invert
  - Resize (here 224×224)

  Each of this operation is done randomly.

- ***Data cleaning***
  Before feeding the data to the training step we delete all the duplicated pictures. After this we also need to control the number as we want that each monument have the same amount of data.

9.2.2 Training the model

We have to choose the architecture of the network we are going to use. Our choice is made experimentally. After several trials of the popular existing models we decided to use the MobileNets [HZC+17] architecture. This model is developed for mobile devices due to its fast characteristics and has a good accuracy.
The Mobilenets was developed by Google's team to reduce the size of the trained parameter in the convolutional layer, they introduce the use of depth separable convolution layers. Each of these layers is followed by a batch normalization suggested in [IS15] to normalize the output from the separable convolution with respect the online batch of data, before feeding into the non-linear transfer function $ReLU6$ (see [Kri10]). And the pooling layer is applied only once before the dense layer.
The main purpose of Mobilenets is to give a trade off on the performance (accuracy) and resources (latency), there is a parameter input α called *depth multiplier* that is used to reduced the depth of the depth-wise convolution kernel. According to the need of our model we choose $\alpha = 1.0$. There are three versions of the MobileNets characterized by the shape of the inputs Images, we use the 224 in our project. An overview of this architecture is presented in the Table 9.1

convolution type	kernel shape	input shape
normal	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
depthwise	$3 \times 3 \times 32$	$112 \times 112 \times 32$
pointwise	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
depthwise	$3 \times 3 \times 64$	$112 \times 112 \times 64$
pointwise	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
depthwise	$3 \times 3 \times 128$	$56 \times 56 \times 128$
pointwise	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
depthwise	$3 \times 3 \times 128$	$56 \times 56 \times 128$
pointwise	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
depthwise	$3 \times 3 \times 256$	$28 \times 28 \times 256$
pointwise	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
depthwise	$3 \times 3 \times 256$	$28 \times 28 \times 256$
pointwise	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5 \times$ depthwise pointwise	$3 \times 3 \times 512$ $1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$ $14 \times 14 \times 512$
depthwise	$3 \times 3 \times 512$	$14 \times 14 \times 512$
pointwise	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 1024$
depthwise	$3 \times 3 \times 1024$	$7 \times 7 \times 1024$
pointwise	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
averagePool2D	7×7	7×1024
fully connected	$1024 \times N$	$1 \times 1 \times 1024$

TABLE 9.1: MobileNet-224

9.3 Operations and tools

In this section, we give the description of the operations done and the tools we used.

9.3.1 Data preparation

The main procedure in the data preparation is the augmentation step since it needs more resources than the other steps. We use Augmentor [BRH19], a *Python* library which is easy to use and quite fast for our task. The profiling of the memory and running time is given in Figure 9.5. This profile (Figure 9.5) was taken to generates

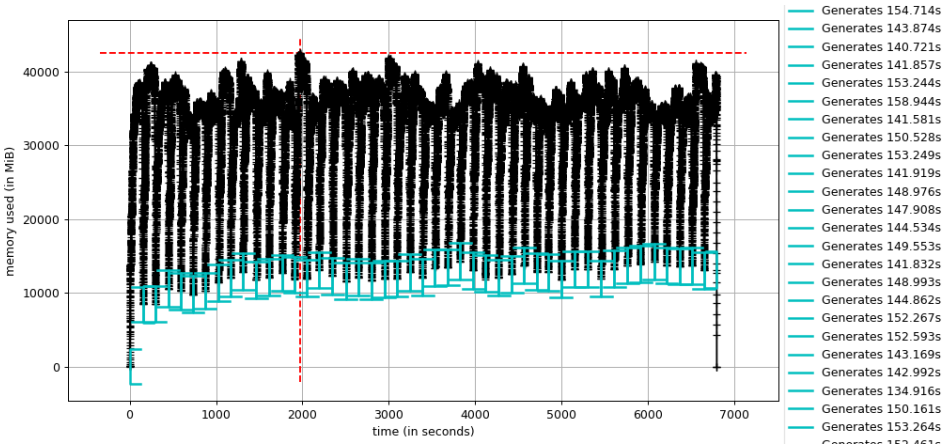


FIGURE 9.5: The memory size occupied by each class running in serial in function of the time.

2000 images per each classes running in serial, each run is processed on a 80 threads CPU. One can observe that each run requires approximately $40GB$ of memory and 160 seconds to finish the task. For this example, we need around 2 hours for a 46 classes. The detection and cleaning of the duplicated images can be done very fast.

9.3.2 Training

In here, we used the known technique called *Transfer Learning*, this is a common technique used in machine learning, which means that instead of training the model from zero-knowledge of the dataset we can start the training with a weighted model. That is we use the weighted MobileNets-1 implementation from Keras [C+15], since the model has been used on the Imagenet[2] database. The reason why this work is that the input dataset have the same features RGB pixels.
We use the network architecture presented in Table 9.1. We initialize the weights of the fully connected layer by random uniform that are regularized by the L_2 function. The categorical *cross entropy* loss function is used for the Stochastic Gradient Descent back-propagation optimizer.
The training is performed on one Nvidia graphic card Tesla P 100 (Table A.1). The time spent depends on the epochs numbers, batch size and number of classes. In our experiment, we warp-up the network for 500 or 1000 epochs for two or more runs, and start the fine-tuning on the weight of the model which have a good performance.

9.4 Convergence and accuracy

The principal task in debugging neural network model is the convergence of the error. There are two main issues, the *underfitting* and *overfitting*. Underfitting is easy to observe since it shows a very week performance on the training data. From the experimental point of view, the cause of this problem is that the model has a difficulty to learn from the training data and a trial with other model is the common solution to overcome this issue. This is one of the reason why we have picked up the MobileNets model.
On the other hand, the overfitting is difficult to detect. The model has very rich performance on the training data but poor accuracy on the test data. To overcome the overfitting, we provide good training data so that the model can learn their conceptual feature. During the training an usual tools is to apply a random dropout (turning off neurons) in the network.

On our dataset, we did a simple comparison by showing graphically the convergence of the loss and accuracy for two weighted networks random uniform and ImageNet. The purpose is to show the performance of the two techniques transfer learning and zero-knowledge.
We can observe that the train accuracy overlap, which means that if we choose the model from its accuracy then both are good. However, the loss of the zero-knowledge model is trapped in a local minimum for both train and validation. This model needs more effort for the fine tuning, time and training data.

In Figure 9.7, we have a convergence of the loss for 1000 epochs. The model was trained for 32 hours on one GPU Tesla P100 (16 GB), the weight was initialized from the ImageNet which has already trained on less classes for a different training data.

[2] http://image-net.org/

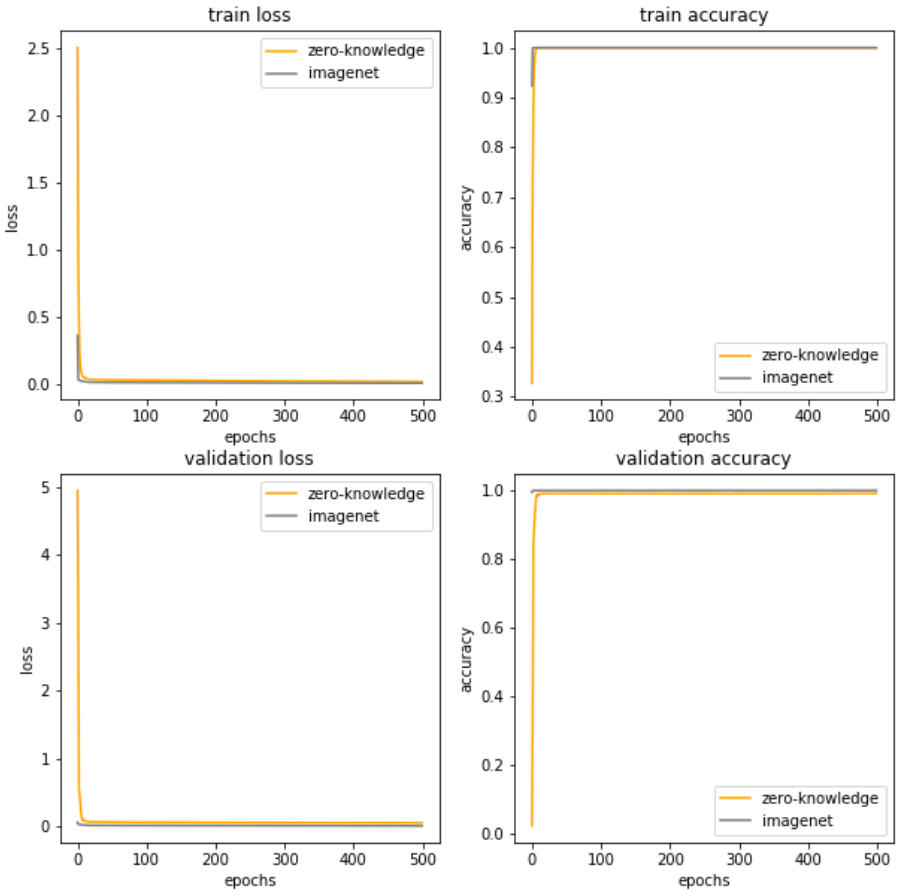


FIGURE 9.6: Comparison between a warm-up 500 epochs of the network initialized by the ImageNet and random weight.

The model performs well on a separate data test with a high accuracy 0.9988086 and lower loss 0.013294586775122288.

For the model initialized by the random uniform weight, 2000 epochs of a fine-tuning took around 63 hours on the same GPU as above. Evaluated on the same data test as above, we obtain a score of 0.98757046 and loss of 0.06327599759058634, we gain a reduction 0.02 of the loss in 2000 epochs. The convergence of the loss function is shown in Figure 9.8. We can observe that the error is converging but very slow.

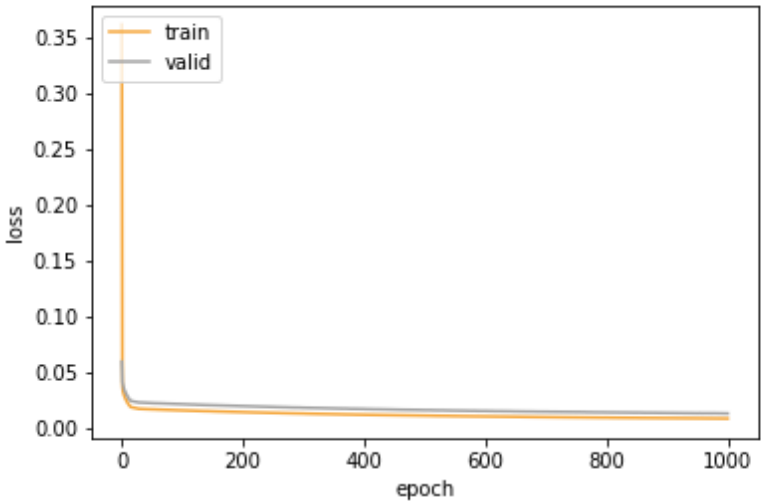


FIGURE 9.7: Convergence of the loos trained on 1000 epochs (ImageNet weight).

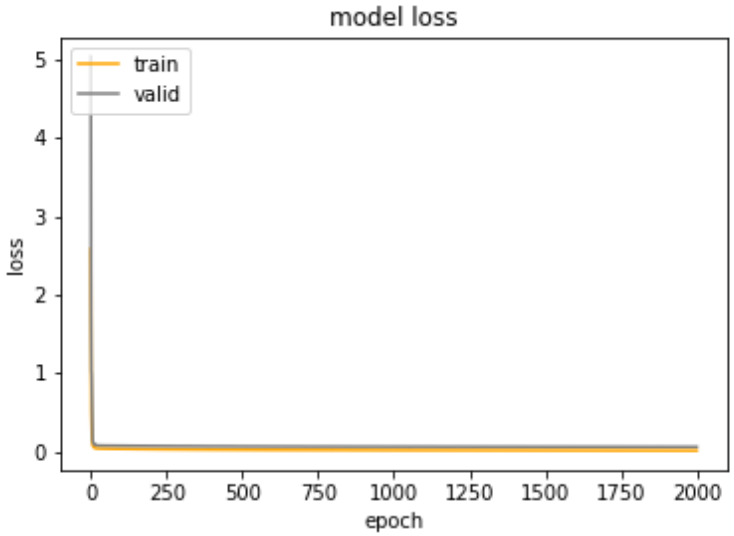


FIGURE 9.8: Convergence of the loos trained on 2000 epochs (Random weight).

10. Summary and work in progress

In this part, we described the detail steps we followed by creating a machine learning model that can be used in mobile platform. The model is trained on architectural images. The main purpose of this part is to give an overview of the experiment we have done for this project.
A simple prototype Android application was developed to test the model. The full application for iOS mobile is described in [Pal19].

This project was initiated for the site Foro Romano, historical monuments in Roma. In this moment, we are working on the historical architecture for Torino. Also we would like to extend the computer vision technique based on deep learning we have used here, such as image segmentation. The model should be flexible in vary type of inputs (pixels images, point clouds 3D, . . .).

References

(ADP19) L. N. Andrianaivo, R. D'Autilia, and V. Palma, *Architecture recognition by means of convolutional neural networks*, ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences **XLII-2/W15** (2019), 77–84.

(Ami89) Daniel J. Amit, *Modeling brain function: The world of attractor neural networks*, Cambridge University Press, 1989.

(Bar93) A. R. Barron, *Universal approximation bounds for superpositions of a sigmoidal function*, IEEE Transactions on Information Theory **39** (1993), no. 3, 930–945.

(BRH19) Marcus D Bloice, Peter M Roth, and Andreas Holzinger, *Biomedical image augmentation using Augmentor*, Bioinformatics (2019).

(C+15) François Chollet et al., *Keras*, https://github.com/fchollet/keras, 2015.

(Cyb89) G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of Control, Signals and Systems **2** (1989), no. 4, 303–314.

(HGDG17) Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick, *Mask R-CNN*, CoRR **abs/1703.06870** (2017).

(Hor91) Kurt Hornik, *Approximation capabilities of multilayer feedforward networks*, Neural Netw. **4** (1991), no. 2, 251–257.

(HPQ+18) Ahmed Hosny, Chintan Parmar, John Quackenbush, Lawrence H. Schwartz, and Hugo J. W. L. Aerts, *Artificial intelligence in radiology*, Nature Reviews Cancer **18** (2018), no. 8, 500–510 (en).

(HZC+17) Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, CoRR **abs/1704.04861** (2017).

(IS15) Sergey Ioffe and Christian Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, Proceedings of the 32nd International Conference on Machine Learning (Lille, France) (Francis Bach and David Blei, eds.), Proceedings of Machine Learning Research, vol. 37, PMLR, 07–09 Jul 2015, pp. 448–456.

(KGC17) Lukasz Kaiser, Aidan N. Gomez, and Francois Chollet, *Depthwise Separable Convolutions for Neural Machine Translation*, arXiv e-prints (2017), arXiv:1706.03059.

(KH19) Andreas Kaplan and Michael Haenlein, *Siri, siri, in my hand: Who's the fairest in the land? on the interpretations, illustrations, and implications of artificial intelligence*, Business Horizons **62** (2019), no. 1, 15 – 25.

(Kri10) Alex Krizhevsky, *Convolutional deep belief networks on cifar-10*, unpublished manuscript, 2010.

(Lec89) Yann Lecun, *Generalization and network design strategies*, Elsevier, 1989 (English (US)).

(**Mon13**) Guido Montufar, *Universal approximation depth and errors of narrow belief networks with discrete units*, Neural Computation (2013).

(**MP43**) Warren S. McCulloch and Walter Pitts, *A logical calculus of the ideas immanent in nervous activity*, The bulletin of mathematical biophysics **5** (1943), no. 4, 115–133.

(**Pal19**) Valerio Palma, *Towards deep learning for architecture: a monument recognition mobile app*, ISPRS - International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences **XLII-2/W9** (2019), 551–556 (en).

(**RW17**) Waseem Rawat and Zenghui Wang, *Deep convolutional neural networks for image classification: A comprehensive review*, Neural Computation **29** (2017), 1–98.

(**SH08**) Ilya Sutskever and Geoffrey E. Hinton, *Deep, narrow sigmoid belief networks are universal approximators*, Neural Computation **20** (2008), no. 11, 2629–2636, PMID: 18533819.

Appendix A. Software tools

A.1 GPU programming

The Graphic Processing Unit (GPU) is a dedicated hardware to accelerate graphic rendering. In 3D-graphic transformation (geometry operation) all the operations are based on floating-point, vector and matrix manipulation. In modern graphic applications (*gaming*) these operations are a huge amount and require to be performed very fast (w.r.t to real time), although they have the property of being able to be processed independently. The GPU is responsible of the processing and the visual output in real time. For this reason it needs to have a very good resolution and framework.
At the same time, the improvement of the algorithm architecture, especially in term of parallel computing evolved very fast. Many scientific computational problems meets their limit of time and memory. The GPU was suggested to handle this problem, for instance, for real time simulations, solving numerical calculation, machine learning and so on. Unlike the CPU (Central Unit Processor), the GPU possesses a huge number of transistors (so many threads). This is of purpose of doing the same operations in parallel. This architecture is referred to the SIMD (Single Instruction Multiple Data) system.
In 2008, with the family *Tesla* Nvidia Graphic Card, the **CUDA** (Compute Unified Device Architecture) has been introduced, a parallel computing platform and programming model that makes using the GPU for general purpose computing simple and elegant[1]. Precisely, this general purpose can be seen as, software to program the GPU and allow an efficient and scalable execution on it. In term of hardware, it exploits the parallelism of the GPU via its number of multiprocessors endowed with cores and memory hierarchy. At a low level programming language, CUDA is an extension of **C** language. The program is written (in *C* with some specification) in on single threads but it executes automatically in bunch of threads in parallel. In CUDA, the keyword *Device* is to refer to the GPU where *Host* for the CPU.

A.1.1 Host and Device

The CUDA-C principle is to allow heterogeneous computing between host and device. In general, the code contains the host instruction that control and access the device. The device instruction consists of a specifics function for the device and a global codes that are basically a set of routing supported by the two sides.
The device can be seen as a co-processor of of the host, CUDA controls each part of the code that are executed on the GPU and CPU, as well as the access of the data. This later is important since both sides have a different storage which are referred respectively the *system memory* (ex DDR) and *video memory* (ex GDDR) for the host and device respectively. The two memory is relied by a PCI bus.
CUDA-C comes with a compiler called NVCC, it is an extension of gcc which can be used to compile a host code. The control of the access of the device and its memory is made automatically once the CUDA code is written carefully by the programmer. This is the principal routine seen in a cuda code, the allocation of memory and the transfer of data between device and host.

[1] https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2

A.1.2 CUDA hardware design

To have a better code in a low level language, coder should understand the design of the hardware. Especially for CUDA hardware model, the study of the design is very important, it has a huge impact on the execution time and the correctness of the result. We give a a rough description of a general concept on CUDA hardware model, further information can be seen in [Wil13].

GPU hardware

a GPU consists of bunch of multiprocessors (referred as SMs), each of SM contains many cores (the stream processors), the total number of cores in a GPU is the number of multiprocessor multiply by the number of stream processors. Each multiprocessor has shared and register memory, the cache (read-only) memory which are the constant and the (traditional graphic) texture memory. Due to the improvement of the new generation of GPU, with the number of transistors increments, the introduction of the L1 improved to L2 cache memory, the width of the address bus that can fit up to 64-bit precision, floating (double) point operation (**fp64**), the warp scheduler. There are features that are supported (cuda program and graphic application) by each GPU, this is mainly depend on the architecture of device, it is referred as *cuda compute capability*. Notices that, of course, more performance consumes more energy power.
For example, the following tabular presents the specification of the Tesla P100[2]

Architecture	**NVIDIA Pascal (Tesla P100)**
Time frame	**2016**
Transistors	**15300 Million**
Compute capability	**6.0**
Multiprocessors	**56**
cores per multiprocessor	**64**
Total cores	**3584**
fp64 cores	**1792**
Clock frequency	**1491 MHz (1.48 GHz)**
Double precision performance	**4.7 TFLOPS**
Video memory (bandwidth)	**16G (720 GB/s)**
Memory technology	**4096-bit HBM2**
Register per multiprocessor	**65536**
Shared memory per multiprocessor	**64 KB**
L2 cache	**4096 KB**

TABLE A.1: GPU Tesla P100.

Memory architecture

In CUDA, there are essential keywords **Grid** which is composed by some number of **Block**, and each **Block** is set of threads. **Warp** is a set of 32 threads, the granularity of the scheduler for issuing threads to the execution units. In CUDA programm, each multiprocessor (hardware) processes batches of blocks (software) in serial, that

[2] https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf

means one SMs may handle many blocks. During execution, all the blocks executed by each multiprocessor called the actives blocks hence the actives threads. The dimension of the block and grid is explicitly given by the programmer, this can be 1D, 2D or 3D. An illustration of a 2D grid of blocks is presented in Figure A.1

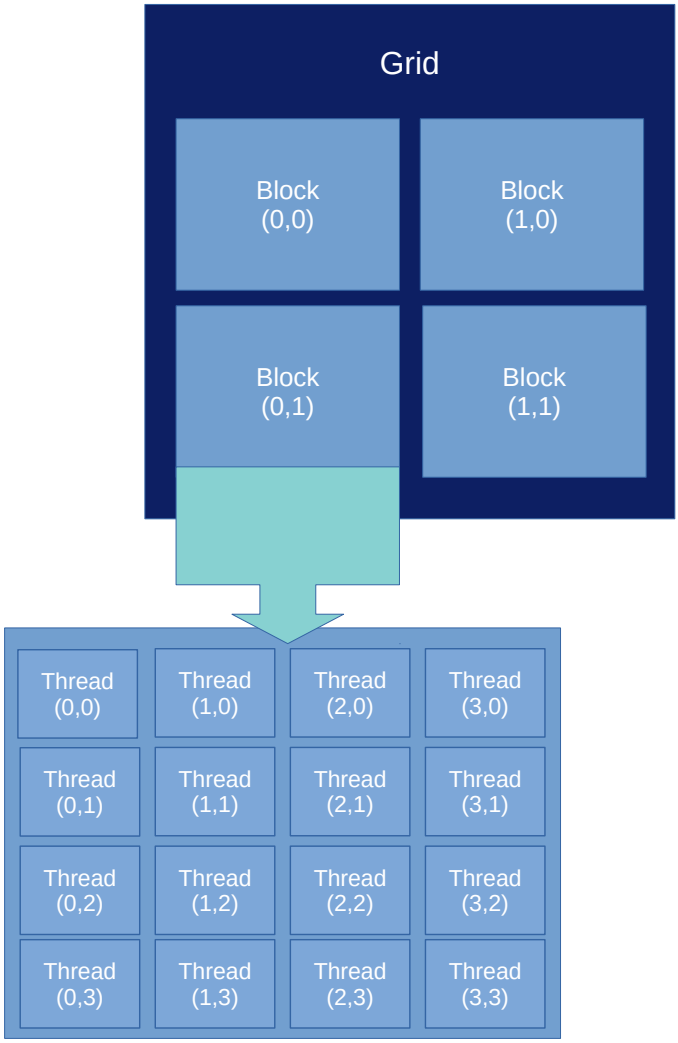


FIGURE A.1: A grid of threads example in 2D (grid and blocks).

The GPU hardware is occupied by a video memory which is referred to the global memory, all multiprocessors have an access on it. The constant memory is a cache memory which can be accessed by all multiprocessors. The texture memory, the traditional cache memory of the graphic card (used to render image). The shared memory is private for each multiprocessor as well as the register memory for each processor. The L1 cache memory is the cache memory used for each multiprocessor in parallel with the shared memory, the L2 cache for all the multiprocessor in parallel with the global memory.
From the hardware design we have two classes of memory all within the graphic card, the on-chip and the attached on the GPU chip. Shared and register memory

are on-chip while the constant, texture within the global memory (un-cached) are outside of the graphic card. The L1 and L2 cached memory are used to accesses global memory so that they are not accessible from the cuda code. This architecture is shown in Figure A.2.

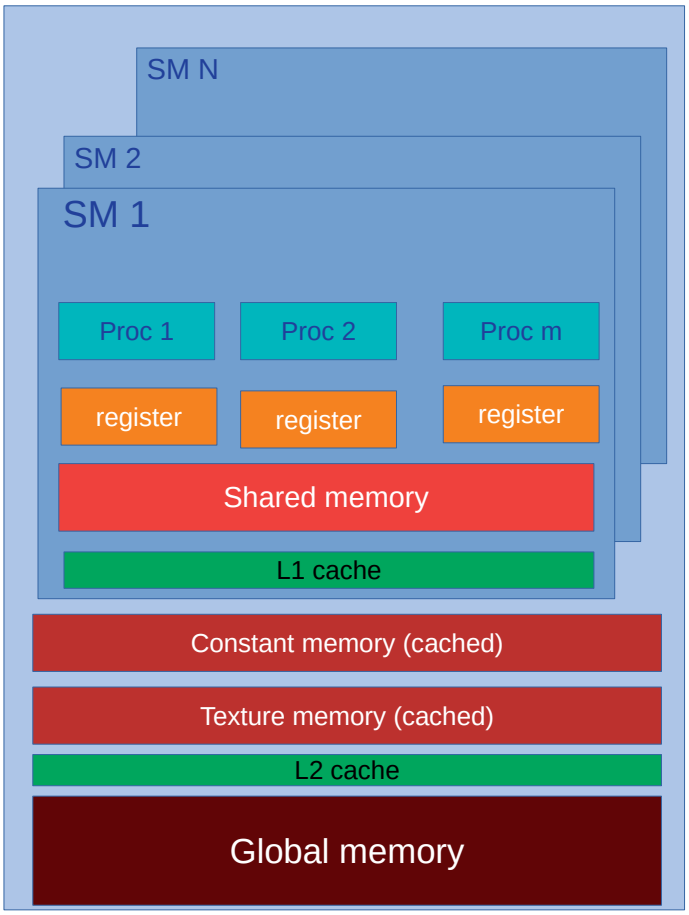


FIGURE A.2: GPU memory architecture.

The global memory is a huge amount of memory, used to be faster than the system memory of CPU but much slower (500 cycles) than the shared memory, L1 and L2 memory are small amount of cache memory used to accesses element on the global memory, their location is different. The texture and constant memory are some small amount of memory on the global memory but cached, the shared memory is a private memory for each Multiprocessor (SM) and the register is private for each processor. The velocity of the memory is with respect to its location, close to processor fast to access.
In the cuda code, one multiprocessor can handle more than one block, the shared memory is divided between those blocks, so one often consider that each block have

its private part of shared memory, as well as the register for thread since one processor may have more than one threads.

A.1.3 Cuda

The function that run on the GPU is referred as kernel, the code is implemented as one process but executed in a block of threads simultaneously. Simultaneously is not always the case, because it runs simultaneously on the active thread. There is a scheduler which schedule the execution of some bunch of threads. In fact, the kernel is launched on the grid, each SM execute the block that belongs to it, could be concurrent or sequential without synchronization. The scheduler execute a number of threads multiple of the dimension of the warp, as we have mentioned, that block may contain more than one warp. One of the reason we use dimension of block as multiple of the dimension of warp is to avoid the waste of threads. The block can be synchronized explicitly from the code. The kernel has further arguments referred as the execution configuration: the dimension of the block, grid and the shared memory allocated dynamically.

Optimization

For each kernel function, the dimension of grid and block is given. This number must be optimized with respect to the features of the device to obtain a better performance. There are two main components that needs to be taken care when optimizing cuda codes, the *memory* usage and thread occupancy. Regarding to the different hierarchies of the memory on the GPU, each has its number of cycles for accessing its address. For the thread occupancy, since each SMs is composed by more than or equal to one warps, each SM launched is equivalent to a multiple of warp-size activated threads. For example, if the dimension of the block-threads is not multiple of the warp-size then the offset threads are wasted, which imply the warp occupancy is weak. This is not the only issue that limit the potential occupancy of a CUDA code, there are the registers memory, shared memory and the block size.