Master degree in Mathematical Engineering (Advanced Scientific Computing) of Lille 1

Universities of Lille 1 and Western Ontario

internship report :

Real root isolation for univariate polynomials on GPUs and multicores

Author :
Alexandre Temperville

Supervisor :
Dr. Marc Moreno Maza

April 16th - August 16th 2012
Presentation June 22th 2012

Acknowledgements

First of all, I thank Marc Moreno Maza to supervise me all these months and also for his welcome, his help, his explanations and all the comments he gave me to improve my work and this paper. I want to thank also all the team of the lab, Chango Chen, Rong Xiao, Anis Sardar Haque and Paul Vrdim for their advices concerning anything, whether it was for the work or for the city, their help and their welcome.

I also thank all the professors of the Master Degree of Computer Sciences at Lille 1 for their patience with my questions, their interesting courses, and in particular Chistophe Besse and Nouredine Melab who are the professors in charge of this Master at Lille 1 for letting me discover all this year High Performance Computing and for their involving in this Master allowing me to be put forth my best efforts. I thank also François Lemaire and Adrien Poteaux who coordinates relations between Lille 1 and UWO and for lending one's support to me.

I thank also my roomates, Matthew and Raj, who give me a nice stay and also for their kindness, their humour and their friendship. Life in Canada without them would not be the same.

To finish I want to thank my family and my friends sending me emails, giving me shouts and news, in particular my little brother Jérôme to help me in english. It was really encouraging from the part of everyone, so thank you everybody.

Contents

1 Internship context

1.1 University of Western Ontario of London (UWO)

The University of Western Ontario, located in the city of London and generally called UWO, is among the top 10 Canadian universities. Renowned for the quality of its students' life, this university is also worldly ranked between 150 and 300, from various sources.

1.2 Departments of Computer Science and Applied Mathematics

These departments are located in the building called Middlesex College we can see in the presentation page.

1.3 My supervisor, professor Marc Moreno Maza

Marc Moreno Maza is an associate professor in the department of Computer Sciences and in the department of Applied Mathematics at the University of Western Ontario. He is also a Principal Scientist in the Ontario Research Centre for Computer Algebra (ORCCA).

Marc Moreno Maza's research activities have four directions :

- Study theoretical aspects of systems of polynomial equations and try to answer the question "what is the best form for the set of solutions ?"
- Study algorithmic answers to the question "how can we compute this form of the set of solutions at the lowest cost ?"
- Study implementation techniques for algorithms to make the best use of today's computers.
- Apply it to unsolved problems when the prototype solver is ready.

1.4 Maple

Conforming to the research activities of my supervisor and of the departments of Computer Science and Applied Mathematics, we work in close cooperation with the software company *Maplesoft*, which is developing and distributing the computer algebra system MAPLE. Our main purpose is to provide MAPLE's end-users with symbolic computation tools that make best use of computer resources and take advantage of hardware acceleration technologies, in particular graphics processing units (GPUs) and multicores. Marc Moreno Maza's team is currently working on a library called *cumodp* which will be integrated into MAPLE so as to do fast arithmetic operations over prime fields (that is, modulo a prime number).

1.5 Purpose of my internship

I participate to the elaboration of the library *cumodp*. My objective is to develop code for the exact calculation of the real roots of univariate polynomials. Stating this problem is very easy. However, as one dives into the details, one realizes that there are lots of challenges in order to reach highly efficient algorithmic and software solutions.

The first challenge is that of representation. Traditionally, scientific software provide numerical approximations to the roots (real or complex) of a univariate polynomial which coefficients might themselves be known inaccurately. Nevertheless, in many applications polynomial systems result from a mathematical model and their coefficients are known exactly. In this case, it is desirable to obtain closed form formulas for the roots of such polynomials, like $x = \frac{1 \pm \sqrt{5}}{2}$. It is well know, however, from Galois Theory, that the roots of univariate polynomials of degree higher than 4 cannot be expressed by radicals.

In this context, computing the real roots of such a polynomial $f(x) \in \mathbb{R}[x]$ means determining pairwise disjoint intervals with rational end points and an effective one-to-one map between those intervals and the real roots of $f(x)$.

Algorithms realizing this task are highly demanding in computer resources. In addition, the most efficient such algorithms combine different mathematical tools such as Descartes' rule of signs, Fast Fourier Transforms (FFTs), continued fractions, computing by homomorphic images, etc. Therefore, one of my first tasks when I arrived in London, was to learn these techniques which are at the core of the problem which is proposed to me.

2 Key points of the research of real roots of a univariate polynomial

2.1 Descartes' rule of signs

In his 1637 treaty called "La Géométrie", René Descartes expressed a rule allowing to estimate the number c of positive real roots of a univariate polynomial, which we commonly call the Descartes's rule of signs. Later, in 1828, Carl Friedrich Gauss proved that if we count the roots with their multiplicities, then the number of positive roots has the same parity as c. Usually, Descartes' rule of signs refers also to the enhancement of Gauss.

Descartes' rule of signs (DRS). *Let us consider a univariate polynomial $P \in \mathbb{R}[X]$ and the sequence (a_n) of its non-zero coefficients. Let c be the number of sign changes of the sequence (a_n). Then the number of positive roots of P is at most c.*

Gauss' property. *If we consider the previous rule of signs and count the roots with their multiplicities, then the number of positive real roots of P has the same parity than c.*

Consequence : We can also find the number of negative real roots applying Descartes' rule of signs to $Q(X) = P(-X)$.

This easy rule is the pillar of the reasoning in our work. Indeed, deciding where the real roots of a polynomial are located reduces to the problem of counting the number of real roots of a polynomial in an interval. The example of Section 2.3 illustrates this process. The following basic theorem plays an essential role in the proof of Descartes' rule of signs.

Theorem of the intermediate values (TIV). *If f is a real-valued continuous function on the interval $[a, b]$ and u is a number between $f(a)$ and $f(b)$, then there exists $c \in [a, b]$ such tha we have $f(c) = u$.*

In particular if $u = 0$, then there exists $c \in [a, b]$ such that $f(c) = 0$ holds.

2.2 Horner's method

This well-known high school tricks is used for evaluating a polynomial efficiently. Instead of considering $P(X) = \sum_{i=0}^{n} a_i X^i = a_0 + a_1 X + a_2 X^2 + ... + a_n X^n$, we write P as

$$P(X) = a_0 + X\ (a_1 + X\ (a_2 + X\ (\cdots (a_{n-1} + (a_n X) \cdots)))).$$

This allows one to reduce the number of coefficient operation necessary to evaluate $P(X)$ at a point from $\Theta(n^2)$ to $\Theta(n)$.

Example : Let us consider $P(X) = 3X^3 - 5X^2 - 3X + 2$. We want to calculate $P(4)$ by hand.
Naive method : $P(4) = 3 \times 4^3 - 5 \times 4^2 - 3 \times 4 + 2$
So $P(4) = 3 \times 4^3 - 5 \times 16 - 12 + 2 = 3 \times 16 \times 4 - 80 - 10 = 3 \times 64 - 90 = 192 - 90 = 102$
Horner's method : $P(X) = 2 + X\ (-3 + X\ (-5 + 3X))$
So : $P(4) = 2 + 4\ (-3 + 4\ (-5 + 3\,4)) = 2 + 4\ (-3 + 4 \times 7) = 2 + 4 \times 25 = 102$

2.3 Example

Let us consider $P(X) = X^3 + 3X^2 - X - 2$.
According to the Descartes' rule of signs, $c = 1$ so P has 1 positive root.
Let us consider $Q(X) = P(-X) = -X^3 + 3X^2 + X - 2$, here $c = 2$ so P has either 2 either 0 negative roots. We have $P(-1) = 1$ and $\lim_{X \to -\infty} P(X) = -\infty$ so there exists $r_1 \in]-\infty; -1[\ |\ P(r_1) = 0$, then we deduce that there are exactly 2 negative roots.
Using the TIV for $u = 0$, we can refine these intervals, calculating P at some real numbers :
As $P(-1) = 1$ and $P(-2) = -2$, then $r_1 \in]-2, -1[$.
As $P(0) = -2$ and $P(-1) = 1$, then the second negative root $r_2 \in]-1; 0[$.
As $P(0) = -2$ and $P(1) = 1$, then the positive root is $r_3 \in]0, 1[$.

2.4 Vincent Collins Akritas Algorithm

To isolate the real roots of a polynomial P, our objective is to use the Vincent-Collins-Akritas (VCA) Algorithm which computes a list of disjoint intervals with rational endpoints such that each real root of P belongs to a single interval and each interval contains only one real root of P. As mentioned above, the problem of isolating the real roots of P and that of counting its real roots are essentially the same. Therefore, if an algorithm solves one of these two problems, then is solves the other. The following Algorithm 1 (taken from [1]) uses Algorithm 2 and shows how we can reduce the search for the roots in $\mathbb{R}$ to the search of the roots in $]0, 1[$.

Algorithm 1: RealRoots(p)

Input: a univariate squarefree polynomial p of degree d
Output: the number of real roots of p

```
1 begin
2   Let k ≥ 0 be an integer such that the absolute value of all the real roots of p is less than or equal to 2^k;
3   if x | p then m := 1 else m := 0;
4   p1 := p(2^k x);
5   p2 := p1(−x);
6   m' := RootsInZeroOne(p1);
7   m := m + RootsInZeroOne(p2);
8   return m + m';
9 end
```

Algorithm 2: RootsInZeroOne(p)

Input: a univariate squarefree polynomial p of degree d
Output: the number of real roots of p in $(0, 1)$

```
1  begin
2    p1 := x^d p(1/x);
3    p2 := p1(x + 1); //Taylor shift
4    Let v be the number of sign variations of the coefficients of p2;
5    if v ≤ 1 then return v;
6    p1 := 2^d p(x/2);
7    p2 := p1(x + 1); //Taylor shift
8    if x | p2 then m := 1 else m := 0;
9    m' := RootsInZeroOne(p1);
10   m := m + RootsInZeroOne(p2);
11   return m + m';
12 end
```

Algorithm 1 calls several times the procedure RootsInZeroOne which, itself, performs several times a Taylor shift by 1 (usually called Taylor shift). The Taylor shift by $a \in \mathbb{R}$ of a polynomial P consists of computing the coefficients of the polynomial $P(x + a)$ in the monomial basis. We will see later what there are differents ways to do this Taylor shift by 1. Even if evaluating $P(x + 1)$ seems an operation mathematically trivial, non-triavial algorithms have been developed to perform this operation efficiently on polynomials of large degrees.

Since the dominant computational costs of the VCA Algorithm comes from the Taylor shift by 1, it is natural to optimize this operation, in particular, in terms of parallelism and data locality. Of ourse Algorithms 1 and 2, with their divide and conquer scheme, seem to provide additional opportunities for concurrent execution. However, the work load in the recursive calls of Algorithms 1 and 2 is generally largely unbalanced, thus, leading to very little parallelism in practice.

2.5 Modular arithmetic

Computing with polynomials or matrices over $\mathbb{Z}$ (and thus $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) one generally observes expression swell in the coefficients. This phenomenom can be a severe performance bottleneck for computer algebra software. There are essentially two ways to deal with that. One solution is to use highly optimized multi-precision libraries, such as GMP, for computing in $\mathbb{Z}$ and $\mathbb{Q}$. Another approach consists in computing by homomorphic images. One popular way to do this is via (one of the variants of) the *Chinese Remainder Algorithm*, the other one is to Hensel's Lemma. In our work, we rely on the former, see Section 2.6.

Therefore, we replace computations (for instance Taylor shift by 1) over $\mathbb{Z}$ by computations over prime fields of the form $\mathbb{Z}/p\mathbb{Z}$ where p has machine word size. Then it becomes essential to perform efficiently arithmetic operations in $\mathbb{Z}/p\mathbb{Z}$.

In the C code below, used in my implementation, *sfixn* represents an integer according to the architecture and operating system of the target computer. For Linux on Intel 64-bit, *sfixn* is an *int*) and $BASE\_1 = 31$.

add_mod

It corresponds to the addition modulo a number, using binary operations.

```
__device__ __host__ __inline__
sfixn add_mod(sfixn a, sfixn b, sfixn p)
{
    sfixn r = a + b;
    r -= p;
    r += (r >> BASE_1) & p;
    return r;
}
```

mul_mod

It corresponds to the multiplication modulo a number, using binary operations also, but contrary to what we can expect, we use floting point numbers and the euclidean division.

```
__device__ __host__ __inline__
sfixn mul_mod(sfixn a, sfixn b, sfixn n, double ninv)
{
    sfixn q  = (sfixn) ((((double) a) * ((double) b)) * ninv2);
    sfixn res = a * b - q * n;
    res += (res >> BASE_1) & n;
    res -= n;
    res += (res >> BASE_1) & n;
    return res;
}
```

The reason of the use of this *mul_mod* procedure is detailed in [8] pages 78-79, it takes advantage of hardware floating point arithmetic. Double precision floating point numbers are encoded on 64 bits and make this technique work correctly for primes p up to 30 bits. This technique comes from euclidean division. Let us explain this, to obtain $res = a \times b \mod p$ with p a prime number and $res < p$, then we have to divide$a \times b$ by p to obtain the quotient q and then the remainder. This is equivalent to multiply $a \times b$ with $pinv$ and do an integer cast after so $q = (int)\ a \times b \times pinv$.
Let us consider $prod = a \times b$. Let us recall also that the euclidean division of a integer $prod$ by another p is of the form : $prod = quotient * p + remainder$ with $q = quotient$ and $res = remainder < p$.

Then : $res = a \times b - q \times p$. After, there are just some binary operations to have clearly the good number we wish.

inv_mod

To compute the inverse of an integer a modulo another integer p, we must first check if this possible. Indeed, this is possible only if a and m are relatively primes. If p is a prime number and $1 \leq a < p$, this is always the case. That's why we will only deal with prime numbers in our code as we will need to compute

inverse of numbers. To do that, we will use the extended euclidean algorithm which consists on finding the integers u and v such that $a \times u + b \times v = GCD(a, b)$ for two integers a and b given. In particular, if we take $b = p$ a prime number then we have $a \times u \equiv 1 \mod p$ so u will be the inverse we are looking for. *egcd* computes this u used in the function *inv_mod*.

```
__device__ __host__ __inline__
void egcd(sfixn x, sfixn y, sfixn *ao, sfixn *bo, sfixn *vo)
{
    sfixn t, A, B, C, D, u, v, q;

    u = y; v = x;
    A = 1; B = 0;
    C = 0; D = 1;

    do {
        q = u / v;
        t = u;
        u = v;
        v = t - q * v;
        t = A;
        A = B;
        B = t - q * B;
        t = C;
        C = D;
        D = t - q * D;
    } while (v != 0);

    *ao = A;
    *bo = C;
    *vo = u;
}

__device__ __host__ __inline__
sfixn inv_mod(sfixn n, sfixn p)
{
    sfixn a, b, v;
    egcd(n, p, &a, &b, &v);
    if (b < 0) b += p;
    return b % p;
}
```

quo_mod

Using the previous modular instructions, this one gives the quotient modulo a prime number of two integers.

```
__device__ __host__ __inline__
sfixn quo_mod(sfixn a, sfixn b, sfixn n, double ninv)
{
    return mul_mod(a, inv_mod(b, n), n, ninv);
}
```

2.6 Chinese Remainder Theorem

As we are working modulo a prime number in our work, it is not sufficient to give the answer in $\mathbb{Z}$. We will need to run our code several times with different values of p and then recombine the results to get the real solution on $\mathbb{Z}$. In this section we recall the Chinese Remainder theorem and explain how we can use it, the implementation in the code will be explained later.

Chinese Remainder theorem 1^{st} version (CRT1). *Let us consider m_1, m_2, ..., m_r a sequence of r positive integers which are pairwise coprimes. Let us consider also a sequence (a_i) of integers and the following system (S) of congruence equations :*

$$(S) : \begin{cases} x \equiv a_1 \mod m_1 \\ x \equiv a_2 \mod m_2 \\ \quad \vdots \\ x \equiv a_r \mod m_r \end{cases}$$

Then (E) has a unique solution modulo $M = m_1 \times m_2 \times \cdots \times m_r$:

$$x = \sum_{i=1}^{r} a_i \times M_i \times y_i = a_1 \times M_1 \times y_1 + a_2 \times M_2 \times y_2 + \cdots + a_r \times M_r \times y_r$$

with $\forall i \in [\![1, r]\!]$, $M_i = \dfrac{M}{m_i}$ and $y_i \times M_i \equiv 1 \mod m_i$.

Running the code of the Taylor shift by 1 we implement a lot of times for differents prime number m_i, we will obtain all the coefficients a_i and then we will just need to get the integers M_i and y_i to have $x \mod M$. We can so use this theorem to solve our problem and find the different coefficients of our polynomial 'shifted'. One may notice that in this case, we will obtain the solution in $\mathbb{Z}[x]/M\mathbb{Z}$ and not in $\mathbb{Z}$ as we want but according to the following lemma coming from [4], if M is sufficiently big then we have our solution in $\mathbb{Z}[x]$:

Lemma. *Let $f \in \mathbb{Z}[x]$ be nonzero of degree $n \in \mathbb{N}$ and $a \in \mathbb{Z}$. If the coefficients of f are bounded in absolute value by $B \in \mathbb{N}$, then the coefficients of $g = f(x+a) \in \mathbb{Z}[x]$ are absolutely bounded by $B(|a|+1)^n$.*

The Chinese Remainder Theorem is also known i its algebraic form :

Chinese Remainder theorem 2^{nd} version (CRT2). *Let us consider m_1, m_2, ..., m_r a sequence of r positive integers which are pairwise coprimes and $M = m_1 \times m_2 \times \cdots \times m_r$. Then :*

$$\mathbb{Z}/M\mathbb{Z} \cong \mathbb{Z}/m_1\mathbb{Z} \times \mathbb{Z}/m_2\mathbb{Z} \times \cdots \times \mathbb{Z}/m_r\mathbb{Z}.$$

3 Taylor shift

The Taylor shift by a of a polynomial P consists on evaluating the coefficients of $P(x+a)$. So, for a polynomial $P = \sum_{0 \leq i \leq n} f_i \, x^i \in \mathbb{Z}[x]$ and $a \in \mathbb{Z}$, we want to compute the coefficients $g_0, \ldots, g_n \in \mathbb{Z}$ of the Taylor expansion :

$$Q(x) = \sum_{0 \leq k \leq n} g_k \, x^k = P(x+a) = \sum_{0 \leq i \leq n} f_i \, (x+a)^i$$

There exists several classical polynomial arithmetic methods and also asymptotically fast methods described in [4]. Among the three classical polynomial arithmetic methods explained in [4], all these methods amount to deal with the Horner's method to do a Taylor shift by 1. We have already explained in what consists this method in the part Horner's rule. I implement the Horner's method in C++ code, see appendix for this code.

Concerning the asymptotically fast methods, if we take a look at the different execution times, we remark that the divide and conquer (D & C) method is the fastest one. This is the method we favour for parallel programming. I implement also a melt of the convolution method and the D & C method in C++ code, just to have a preview of the result and compare it with the Horner's method and the D & C method in parallel. I don't develop so much this method for the reason I had implement the Horner's method and it was sufficient to compare with the parallel method. In our work we mainly want to do a Taylor shift by 1.

3.1 Divide & Conquer method (D & C)

This method consists to split a polynomial P of degree $n = 2^e - 1$ (so this polynomial as 2^e coefficients) in other polynomials we split also. We will call *size of a polynomial* the number of coefficients of this polynomial, so we consider here polynomials of size a power of 2. We split the polynomial considered as the following (for a Taylor shift by 1) :

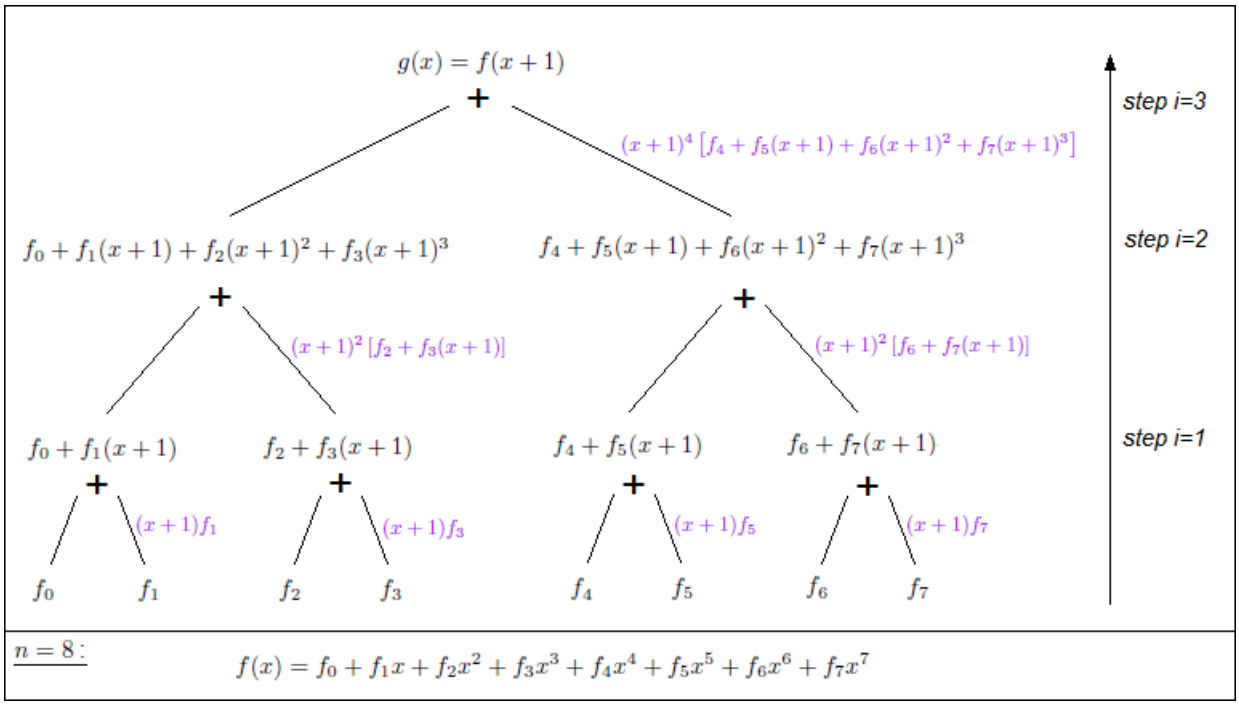
At the beginning, we split P like this :

$$P(x) = P^{(0)}(x+1) + (x+1)^{\frac{n+1}{2}} \times P^{(1)}(x+1)$$

Then, we evaluate $P^{(0)}(x+1)$ and $P^{(1)}(x+1)$ recursively. So we have finally to consider 3 things, the computations of all the $(x+1)^{2^i}$ for $i \in [\![0, e-1]\!]$, a multiplication of one of these $(x+1)^{2^i}$ with the evaluation of a $P^{(j)}(x+1)$ and then an addition. One may notice that :

- We just need all the taylor shift by 1 of the monomial x^{2^i} for $i \in [\![0, e-1]\!]$, namely $(x+1)^{2^i}$.
- At each step, we split polynomials of size 2^d in two polynomials of size 2^{d-1}, so at each step of the recursion, we approach a size which will be more and more easier to compute (and the size is always a power of 2).
- Contrary to the polynomial $P^{(j)}$ we consider, $(x+1)^{2^i}$ is not a power of 2 (except for $i = 0, 1$). This will be a problem we will need to solve later, and we will see why and what's our strategy.
- At the next step of this recursion, we call the coefficients of $P(x)$.

We can represent how to do this recursion as the following tree. At the beginning we want to split the polynomial $P(x+1)$ not evaluated in two polynomials, which need to be splitted in turn, and so on. So the recursion begin at the top of this tree.



The first problem we can encounter is the fact it is not simple to make a recursive algorithm in parallel. That's why finally we will consider the tree since its base, i.e. from the coefficients of the input polynomial coefficients. In the C++ code, I did the recursive method, and it was easier to write a method from the base in parallel. In each branch, we see if we need to multiplicate the polynomial evaluated at the base by a monomial shift or not. We see that for each step, the parallelism can be obtained easily. Now the difficulty is to work step by step, consider the good sizes of the arrays, the number of polynomials, the position of the coefficients we're modifying, how to do that the fastest possible.

3.2 Compute the $(x+1)^{2^i}$

Idea

To compute the serie of $(x+1)^{2^i}$, we have different ways to proceed. But all the classic ways are not necessary the best to compute all the coefficients.

If we need all the $(x+1)^j$ for $j \in [\![1, n]\!]$, it will be better to use the formula $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$ to compute each element and proceed to a divide and conquer method to compute in parallel the coefficients. But, we just need the coefficients of $(x+1)^{2^i}$. Then, if we proceed with a divide and conquer on all the elements of $(x+1)^j$ and just keep the $(x+1)^{2^i}$, we can have a lot of computations, in particular for i big, which won't be stored and just be needed to go to the next power of $(x+1)$. That's not what we want, so we will use the formula of the binomial coefficients with the factorial sequence to get directly the coefficient of each $(x+1)^{2^i}$.

With the method I will explain in this part, I think we loose a little time at the beginning, computing yet the factorial sequence in parallel. But after, the computations are fast and nothing is superfluous.

So, to compute the Newton's binomial coefficients, we want to use the following formula :

$$\forall n \in \mathbb{N},\ (x+1)^n = \sum_{k=0}^{n} \binom{n}{k} x^k \text{ with } \binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

We need so to compute all the elements $i!\, \forall i \in [\![0, n]\!]$.

The sequence of factorials

To use the previous formula to get all the binomials' coefficients, we need the values of each $i!\mod p$ for $i \in [\![0, n]\!]$.

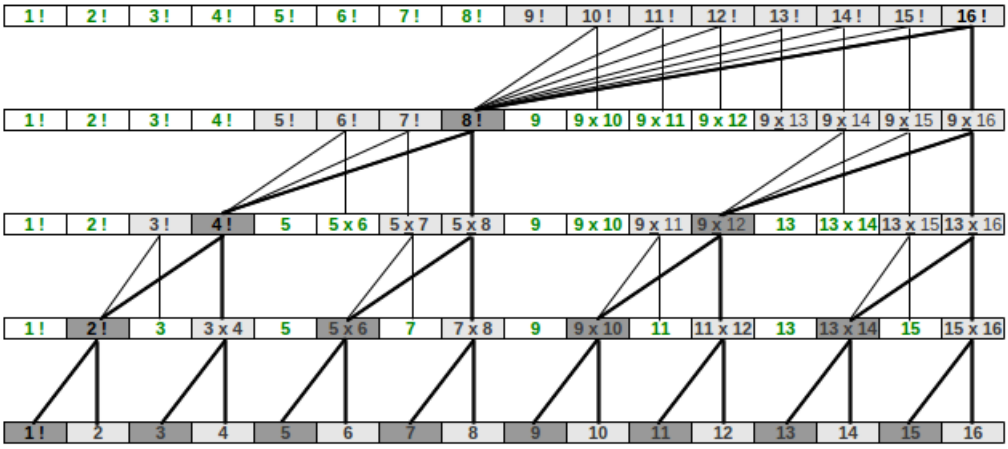
One may notice that to obtain $i!$, we need $(i-1)!$. So if we consider the computation in serie of all the $i!$, it may be very long to obtain $n!$ for n big.

First of all, we need an array of size $n+1$, called *Factorial_ device* in my code as we also need 0!. But we don't really need to compute $0! = 1$ so we'll consider the following computations on *Factorial_ device + 1* thus we just compute $i!$ for $i \in [\![1, n]\!]$. Now we consider computations of size $n = 2^e$ with $e \in \mathbb{N}^*$. As n is a power of 2, I was looking for a code to compute pairwise elements.

Mapping for the computation of the factorial sequence

To understand the code, see the following picture from the bottom which is the first step : the initialization. First we initialize the array *Factorial_ device + 1* letting the even elements as their position and we do multiplications on the odd elements.

Then, in each step, we consider the array per parts. We can see a part in each step as the following : we multiply the element of a position by a same number for a part, these 'same numbers' are what I call "pillar factors", represented by the darkest boxes. We can see that step after step, we compute several $i!$ in parallel and finally get every $i!$ in $\log_2(n)$ steps.



Notation : **x** is the product of all the integers between the two integers of the product

Example : 9 **x** 13 = 9 x 10 x 11 x 12 x 13

The straight lines represent the multiplications done from a lower level to the upper level and where the product is stored.

Colored boxes are the boxes modified at each step (or level). The darkest boxes show the elements used several times in multiplications.

See in the **Appendix** the file *Taylor_ shift_ kernel.cu* and in particular the procedures *identity_ GPU*, *create_ factorial_ step0_ GPU* and *create_ factorial_ stepi_ GPU* which correspond to the different steps of the computation in parallel of the sequence of factorials. First we inialise the $n+1$ positions in the array *Factorial_ device* with their position or 1 for *Factorial_ device[0]* with *identity_ GPU* and then the two

others procedure modified $n/2$ positions of the array at each step. The procedure *create_factorial_stepi_GPU* has only one problem, if we take a look at the previous graphic we can see that in each step, a lot of threads will need the same "pillar factor", so many threads will take the value of a same position in the array at the same time, there is an overlap reducing performances of the code. This part needs to be worked to find another way to do the computation of the factorial elements to improve a little more the efficiency of my code.

The array of the $(x+1)^{2^i}$

To do the Taylor shift by 1, we need to compute all the $(x+1)^{2^i}$ as we said before. As now we have the sequence of the factorials, we can compute in parallel each coefficient of the $(x+1)^{2^i}$ but do we we really need all of them ?

If we look at the following Pascal triangle until the binomial coefficients of $(x+1)^8$, naively we just need to use the coefficients colored in yellow. But, a lot of them are 1, we can avoid to have too much ones in the array. Moreover, we don't really need $(x+1)^0$. We want also to store these coefficients in an array of 1 dimension so successively. If we store the coefficients naively, we will store the following sequence : $1, 1, 1, 1, 2, 1, 1, 4, 6, 4, 1, 1, 8, 28, 56, 70, 56, 28, 8, 1$. We have to keep in mind also that when we will need to use $(x+1)^{2^i}$, we will need to know the position of this $(x+1)^{2^i}$ in our array.

1																
1	1															
1	2	1														
1	3	3	1													
1	4	6	4	1												
1	5	10	10	5	1											
1	6	15	20	15	6	1										
1	7	21	35	35	21	7	1									
1	8	28	56	70	56	28	8	1								
1	9	36	84	126	126	84	36	9	1							
1	10	45	120	210	252	210	120	45	10	1						
1	11	55	165	330	462	462	330	165	55	11	1					
1	12	66	220	495	792	924	792	495	220	66	12	1				
1	13	78	286	715	...	...	...	...	715	286	78	13	1			
1	14	91	364	...	...	...	...	...	...	...	364	91	14	1		
1	15	105	455	...	...	...	...	...	...	...	...	455	105	15	1	
1	16	120	560	...	...	...	...	...	...	...	...	...	560	120	16	1

Pascal triangle

Now, if we avoid all the ones in the first column of the Pascal triangle (except the first for $(x+1)^0$, even though it is useless), the sequence to store becomes : $1, 1, 2, 1, 4, 6, 4, 1, 8, 28, 56, 70, 56, 28, 8, 1$. This sequence is better as the coefficients of $(x+1)^{2^i}$ are easily found at the position $2^i - 1$ (except for the first but keep in mind it is useless). Thus, most of the ones in this sequences can be used to represent two consecutives $(x+1)^{2^i}$.

Moreover, for a polynomial of degree n, this array is of size... $n!$ So like this, this array will be very practical to use. For $n = 16 = 2^4$, we will use the following array :

1	1	2	1	4	6	4	1	8	28	56	70	56	28	8	1

This array is called *Monomial_ shift_ device* in the code and created by the procedure *develop_ x_ shift*. Each thread need first of all to know what $(x+1)^{2^i}$ it deals with and after the coefficient it needs to compute with the function *create_ binomial2_ GPU*.

One may notice that if we think more on how to reduce computations, the Pascal triangle is symmetric. So approximatively the half of the coefficients don't really need to be stored as I do for the moment. For example, in the case $n = 16$ I have taken, we could just store $1, 2, 4, 6, 8, 28, 56, 70$ which will be an array of size $n/2$ and after find a way to use this smaller array correctly to use the coefficient. It is possible and it is a way of reflection I keep in mind when we will try to improve our code when it will be completely finished.

4 First steps : Polynomials of size 2 to 512

4.1 The beginning

First of all, we need to have in input the different coefficients of the polynomial we want to be shifted by 1. I made a code called *aleaPol.cpp* to create random polynomials of the size wanted. This code stores the coefficients created in a file line per line. So the input is a file we need to read in sequence and then store its data in an array to begin our program. To do that has obviously a cost in time which increases with the size of the polynomial. The way to store efficiently the polynomials is also a question we must ask.

At the beginning of the Taylor shift, after the array *Monomial_ shift* was built, we need to copy the array *Polynomial* for the device and then do the first step, which is very simple, this is done by the procedure *init_ polynomial_ shift_ GPU*.

Since the second step, computations become less obvious.

4.2 Next steps, how we proceed

Let us recall what we have to do. In each step of the previous tree we saw before. In each step, the half of the branches need to be multiplicated by $(x+1)^{2^i}$, i being the current step. Then the result must be added with the previous polynomials of each branch and all of this in parallel in each part of the tree. First of all, we need to determinate which polynomials we have to multiplicate between us. This is what we do with the procedure *transfert_ array_ GPU*. This procedure stores in the array *Mgpu* the polynomials we need to be multiplicated, at a little difference we will explain with the following procedure.

4.3 The plain multiplication & the right shift

Now I want to multiplicate in parallel all the polynomials I have stored in *Mgpu*. A member of the laboratory, Anis Sardar Haque has implemented a very efficient multiplication called *listPlainMulGpu* of a list of polynomials I wanted to use for its efficiency. I need to be careful with this because the use I want to do with it is different from the use done by other codes written by the laboratory. So, I face to different problems.

Before seeing these problems, let us explain how works exactly *listPlainMulGpu* :

listPlainMulGpu do pairwise products of polynomials of the same size in a list containing these polynomial. For example, an array may contain the coefficients of eight polynomials called P_i for $i \in [\![0,7]\!]$ with *listPlainMulGpu*, we can multiply in parallel four products : the products $P_0 \times P_1$, $P_2 \times P_3$, $P_4 \times P_5$ and $P_6 \times P_7$. To do that, we need in particular in parameter the array containing the polynomials to multiply (*Mgpu1*), the array which will contain the products (*Mgpu2*), the size of the polynomials *length_poly*, the number of polynomials called *poly_ on_ layer* (for our example $poly\_on\_layer = 8$), the number of threads used for on multiplication, the number of multiplications in a thread block, and obviously p and its inverse for computations modulo p. If *Mgpu1* contains k polynomials of size m then *Mgpu1* is of size $k \times m$) and *Mgpu2* contains $k/2$ polynomials of size $2m-1$ so *Mgpu2* is of size $\frac{k}{2} \times (2m-1) \neq k \times m$.

So, for example after the plain multiplication, the following array *Mgpu1* of size

$$poly\_on\_layer \times lenght\_poly = 8 \times 128 = 1024 :$$

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7

becomes the following array *Mgpu2* of size

$$\frac{poly\_on\_layer}{2} \times (2 \times lenght\_poly - 1) = 4 \times 255 = 1020 \ :$$

$P_0 \times P_1$	$P_2 \times P_3$	$P_4 \times P_5$	$P_6 \times P_7$

Now, let us explain what are the problem we face.

First of all, the multiplications we want to do are not between polynomials of the same size. If we look at the tree of computations, we can see that we have to multiply a polynomial $P(X) = \sum_{i=0}^{n-1} a_i X^i$ of size $n = 2^k$ by $(X+1)^n$, we obtain a polynomial of size 2^{k+1} but $P(X)$ and $(X+1)^n$ have for size respectively n and $n+1$. So, to use the multiplication, either we can put zeros on polynomials to have the same sizes (but we will have a lot of multiplications by zeros which will be useless), or we think another way to use the multiplication.

Secondly, the initial code of the plain multiplication considered the output array has a modified size as I show that in the previous arrays (size of 1024 for the first and 1020 for the second). My objective is to keep the same size as the product of my polynomials are different and allow me to keep at each step polynomial sizes as a power of 2.

So I clearly need to modify this procedure a little to adapt it for what I want to do with. My idea was the following :

I decompose the multiplication in another multiplication, a right shift and an addition. As we need the Newton's binomial coefficients, the way to store them I used before was a power of 2. Consider $local\_n = 2^k$:

$$\forall j \in [\![0, local\_n - 1]\!],\ Monomial\_shift[local\_n + j] = \binom{local\_n}{j+1}$$

so at *Monomial_ shift + local_ n*, we store the coefficients of $[(X+1)^{local\_n} - 1]/X$, this polynomial is of size $local\_n$ and thus I want to use this polynomial instead of $(X+1)^m$ for the multiplication.

To understand what we can do, imagine we have to multiply locally a polynomial $Q(X)$ of size $m = 2^k$ by $(X+1)^m$, let us decompose $Q(X) \times (X+1)^m$ and see how to proceed according to the following to compute $Q(X) \times (X+1)^m$:

$$\begin{aligned}
Q(X) \times (X+1)^m &= \left(\sum_{i=0}^{2^k - 1} a_i X^i \right) \times (X+1)^{2^k} \\
&= Q(X) \times [(X+1)^m - 1 + 1] \\
&= Q(X) \times [(X+1)^m - 1] + Q(X) \\
&= Q(X) \times X \times \frac{(X+1)^m - 1}{X} + Q(X) \\
&= X \times \left(Q(X) \times \frac{(X+1)^m - 1}{X} \right) + Q(X)
\end{aligned}$$

So let us keep in mind the formula obtained :

$$Q(X) \times (X+1)^m = X \times \left(Q(X) \times \frac{(X+1)^m - 1}{X} \right) + Q(X)$$

This formula is very interesting because it allows to solve the problems I explained before. Let us explain why. The polynomials $Q(X)$ and $[(X+1)^m - 1]/X$ are of the same size m so can be computed with the plain multiplication described before. Then, as it is not what we really want, we nedd to multiply the result by X which corresponds for our arrays to a right shift of all the coefficients computed, then I won't need to decrease of 1 the size of each product if I do the right shift within this procedure. And then, we just need to add the previous value of Q to get the result we wish. I called this procedure modified *listPlainMulGpu_ and_ right_ shift*. Just see now a concrete example.

Example : We want to compute $(3+4x) \times (x+1)^2$.
Then we store the following coefficients in the array *Mgpu1* : $[3, 4, 2, 1]$.
If we write what happens, this is :

$$\begin{aligned}
(3+4x) \times (x+1)^2 &= x\left[(3+4x) \times (2+x)\right] + (3+4x) && \text{decomposition to simplify the problem} \\
&= x\left(6 + 11x + 4x^2 + 0x^3\right) + (3+4x) && \text{plain multiplication done} \\
&= \left(0 + 6x + 11x^2 + 4x^3\right) + (3+4x) && \text{right shift done} \\
&= 3 + 10x + 11x^2 + 4x^3 && \text{addition done}
\end{aligned}$$

listPlainMulGpu_ and_ right_ shift do the product giving $6+11x+4x^2$ and store it like this $[0, 6, 11, 4]$ so we have done the right shift for the multiplication by x (I write useless zeros in the calculations to show they correspond to a position in the array *Mgpu2*), then we just need to sum with the polynomial, we notice that we just need to sum the half of the coefficients, to do that, I will use the procedure called *semi_ add* I explain in the following part.

4.4 Partial additions

As we just saw in the previous part, at the end, we need to add the missing parts $Q(X)$ to obtain exactly the products we want. If we look precisely at these additions, we don't really need to add all the positions of the arrays. Indeed, if we look at the previous example, just the half of the coefficients of the polynomial where added to $Q(X)$ as the size of the $Q(X)$ is the half of the sizes of the polynomials obtained with the procedure *listPlainMulGpu_ and_ right_ shift*. Now we have really multiplied the half of the branches of the current step, we have to add the polynomials computed in these branches with the polynomials of the branches where there were not any multiplication to do.

So, to sum up, semi_add adds the elements $Q(X)$ which were missing to do correctly $Q(X) \times (X+1)^{2^i}$ and then adds in some way $P^{(0)}$ with $P^{(1)} \times (X+1)^{2^i}$ (with $Q = P^{(1)}$). This ends a loop. For the next loop, we do the same with polynomial sizes increased by 2 and a number of polynomials considered to be multiplicated divided by 2. This corresponds respectively to $local\_n* = 2$ and $polyOnLayerCurrent/ = 2$.

4.5 The arrays

Some arrays are used in all what I have described before. But let us explained exactly the choice of these arrays and what each array does exactly.

At the first step, the array *Polynomial_ device* contains all the coefficients defining the polynomial we want to be shifted. Inside this array, the coefficients are store in the increasing order of the power of x.

We do then the first step of the tree in the array *Polynomial_ shift_ device[0]* which must contain at the end of the loop the polynomials for the next step.

Since the second step, the array *Mgpu* contains exclusively all the polynomials which need to be multiplicated pairwise. For example, at the step 2, let us consider the array :

Polynomial_shift_device[0] =

1	2	3	4	5	6	7	8

Then, some polynomials of size 2 inside need to be multiplicated by $(x+1)^2$ so we store in *Mgpu* :

Mgpu =

3	4	2	1	5	6	2	1

We store then the result of *listPlainMulGpu_ and_ right_ shift* in *Polynomial_ shift_ device[1]*. To complete this array, we add the missing part to have exactly $Q(X) \times (X+1)^{2^i}$ so we add parts of *Polynomial_ shift_ device[0]* with *Mgpu*, and then to finish we complete *Polynomial_ shift_ device[1]*, ready for the next step.

Before to comment the other step, one can see that at each step since now, we just need the previous *Polynomial_ shift_ device[i-1]* to compute *Polynomial_ shift_ device[i]* (and a new *Mgpu*). We don't need to keep all these arrays, but just the previous one at each step so to avoid useless arrays and costly cudaMalloc, I modified my code to just use *Polynomial_ shift_ device[i%2]* and *Polynomial_ shift_ device[(i+1)%2]* so that finally in each step we invert *Polynomial_ shift_ device[0]* and *Polynomial_ shift_ device[1]*. Thus, whatever is the size of the polynomial we consider at the beginning, we have the same number of arrays in the code, at least before the step 10.

Now, we come in a new part of the code. The plain multiplication implemented by the laboratory is very efficient for multiplicate polynomials of degrees at most the number of threads in a thread block, so for polynomials of degrees 512 for my machine. This multiplication can't be done for polynomials of degree size more than 512, at least, it can't be sufficiently efficient. Then we need to proceed differently. For bigger degree, we need to use FFT. The Section 5 explain what it is exactly and how we use it.

5 Fast Fourier Transform (FFT)

In this section, we will describe what is the Discrete Fourier Transform (DFT), the Fast Fourier Transform (FFT) and how we use it in our code. First of all, let's consider in all this section two univariate polynomials $f, g \in R[x]$ of degree less than an integer n, where R is a commutative ring with units. We want to multiply these two polynomials. To not enter in too much details as we just need to apply the theory, we will consider generally rings such that $R = \mathbb{Z}/m\mathbb{Z}$ with $m \in \mathbb{N}^*$.

The product $f \times g$ is of degree less than $2n-1$. We assume we are given a subset $\mathbf{E}$ of R such that $card(\mathbf{E}) = 2n-1$ and that $\forall r \in \mathbf{E}$, we know $f(r)$ and $g(r)$. The values $f(r)g(r)$ can define the polynomial fg completely thanks to the Lagrange interpolation as we have sufficiently values. The cost of defining fg like this is $2n-1$. To build the coefficients directly of fg has a cost in $\Theta(n^2)$. We want to avoid this high cost using another way to do multiplication : this is what FFT will do. the underlying idea of the fast polynomial multiplication based on the Discrete Fourier Transform is the following : assume that there exists a subset P of R with $2n-1$ elements such that :

- evaluating f and g at every $r \in P$ can be done at nearly linear time cost, such as $\Theta(n\log(n))$,
- interpolating $f(r)g(r)$ for $r \in P$ can be done at nearly linear time cost.

Then the multiplication of f by g, represented by their coefficients, can be done in $\Theta(n\log(n))$.

5.1 Discrete Fourier Transform

Definition. *Let n be a positive integer and $\omega \in R$.*

- *ω is a n-th root of unity if $\omega^n = 1$.*
- *ω is a primitive n-th root of unity if :*
  (1) $\omega^n = 1$.
  (2) ω is a unit in R.
  (3) for every prime divisor t of n the element $\omega^{n/t} - 1$ is neither zero nor a zero divisor.

Let n be a positive integer and $w \in R$ be a primitive n-th root of unity. In what follows we identify every univariate polynomial

$$f = \sum_{i=0}^{n-1} f_i\, x^i \in R[x]$$

of degree less than n with its coefficient vector $(f_0, \ldots, f_{n-1}) \in R^n$.

Definition. *The R-linear map*

$$DFT_\omega : \begin{cases} R^n & \to R^n \\ f & \mapsto (f(1), f(\omega), f(\omega^2), \ldots, f(\omega^{n-1})) \end{cases}$$

which evaluates a polynomial at the powers of ω is called the Discrete Fourier Transform (DFT).

Proposition. *The R-linear map DFT_ω is an isomorphism.*

Then we can represent a polynomial f by the DFT representation with ω we will determine in our code.

5.2 Convolution of polynomials

Let n be a positive integer and $\omega \in R$ be a primitive n-th root of unity.

Definition. *The convolution with respect to n of the polynomials $f = \sum_{0 \leq i < n} f_i \, x^i$ and $g = \sum_{0 \leq i < n} g_i \, x^i$ in $R[x]$ is the polynomial*

$$h = \sum_{0 \leq k < n} h_k \, x^k$$

such that for every $k \in [\![0, n-1]\!]$ the coefficient h_k is given by

$$h_k = \sum_{i+j \equiv k \mod n} f_i \, g_j$$

The polynomial h is also denoted by $f *_n g$ or simply by $f * g$ if not ambiguous.

The convolution $f * g$ (of size n) and the product $p = fg$ (of size $2n-1$) are different. Let us try to find a relation between these polynomials. We have :

$$p = \sum_{k=0}^{2n-2} p_k \, x^k$$

where for every $k \in [\![0, 2n-2]\!]$ the coefficient p_k is given by

$$p_k = \sum_{i+j=k} f_i \, g_j$$

We can rearrange the polynomial p as follows :

$$\begin{aligned} p &= \sum_{0 \leq k < n} \left(p_k \, x^k\right) + x^n \sum_{0 \leq k < n-1} \left(p_{k+n} \, x^k\right) \\ &= \sum_{0 \leq k < n} \left(p_k + p_{k+n} \left(x^n - 1 + 1\right)\right) x^k \\ &= \sum_{0 \leq k < n} \left(\left(p_k + p_{k+n}\right) x^k\right) + \left(x^n - 1\right) \sum_{0 \leq k < n} \left(\left(p_k + p_{k+n}\right) x^k\right) \\ &\equiv f * g \mod \left(x^n - 1\right) \end{aligned}$$

Lemma. *For $f, g \in R[x]$ univariate polynomials of degree less than n we have*

$$DFT_\omega(f * g) = DFT_\omega(f) DFT_\omega(g)$$

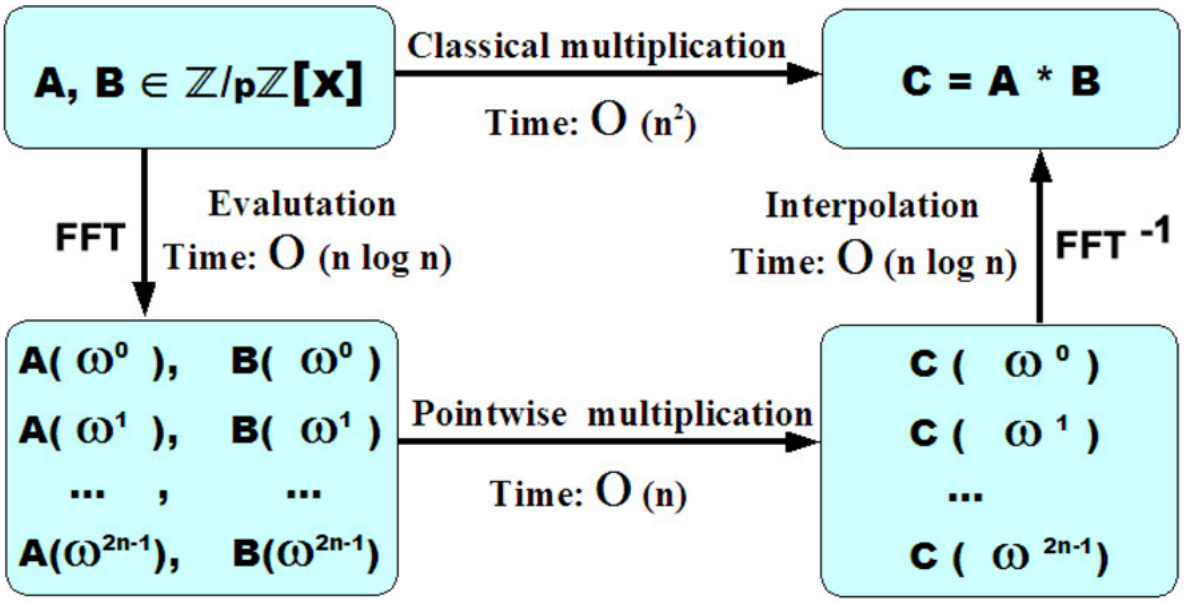
where the product of the vectors $DFT_\omega(f)$ and $DFT_\omega(g)$ is computed componentwise.

This lemma allows us to understand that to compute a convolution product is the same as computing a scalar product.

5.3 The Fast Fourier Transform

The Fast Fourier Transform computes the DFT quickly and its inverse. This important algorithm for computer science was (re)-discovered by Cooley and Tukey in 1965.
Let n be a positive even integer, $\omega \in R$ be a primitive n-th root of unity and $f = \sum_{0 \leq i < n} f_i \, x^i$. In order to evaluate f at $1, \omega, \omega^2, \ldots, \omega^{n-1}$, the Cooley-Tukey algorithm follows a divide and conquer strategy. In [7], Marc Moreno Maza details how this algorithm is done and how the cost of the classical multiplication ($\Theta(n^2)$) can be decreased if we do the multiplication using FFT ($\Theta(n \log(n))$).

The picture below shows how we will proceed to multiplicate two polynomials using FFT :



FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$

6 Polynomials of high degrees (> 512)

As we saw in Section 4, we cannot use the Plain Multiplication for polynomials of size more than 512. And step by step, we increase the size of the polynomials considered we want to multiply. We need to think another way to do a fast multiplication. The previous section explains the idea we want to use for this new case.

Since the step when we need to multiply polynomials of size more than 512 (which is the limit size of the number of threads per thread block), we will proceed to the FFT. The procedures and functions I will use for FFT (*primitive_ root*, *list_ stockham_ dev* and *list_ pointwise_ mul*) were implemented by a Wei Pan, a previous PhD student in the laboratory. The thesis he made put in evidence that some prime numbers improve performances of FFT, see [8] for more details. We will talk about these prime numbers in the part "Next works".

6.1 The array for FFT operations

Even though we need to think the multiplication differently, the other procedures we have done for the smallest size of the polynomials considered are still used. Indeed, first of all, we use again *transfert_ array_ GPU* to put the polynomials we want to be computed in the array *Mgpu*.

But now, this array is not sufficient to be used for FFT. Indeed, we saw before that to compute the product of two polynomials f and g of size n using FFT, we need to know the values of $f(w^i)$ and $g(w^i)$ for $i \in [\![0, 2n-2]\!]$. We first need to convert *Mgpu* in another array twice bigger than it, this array is called *fft_ device*. This array is the same than *Mgpu* but contains some zeros between all the polynomials of this array. *fft_ device* is created thanks to the procedure *transfert_ array_ fft_ GPU*.

6.2 Multiplication using FFT

Now, we need to do the multiplication like explained in the previous section, but first of all, we need to define the value of ω (w in my code) which has to be a 2^{i+1}-th root of unity in $\mathbb{Z}/p\mathbb{Z}$ at the step i, this is done by the function *primitive_ root*.

We need then to evaluate with the procedure *list_ stockham_ dev* thanks to this ω and its powers the polynomials we have stored in *transfert_ array_ fft_ GPU*. Then the positions of the array *fft_ device* where there were zeros contain now the evaluation of the polynomials for some values of ω^j. We follow exactly the scheme at the end of the previous section concerning FFT, we need then to do the pointwise multiplication which is just a multiplication coefficient per coefficient of the pairwise polynomials considered, done by *list_ pointwise_ mul*.

Then, we need to transform again our polynomial doing the inverse operation of the FFT we have done. To do that we need this time to consider another ω which is the invert of the one we use for the FFT.

We don't really obtain the products we want, but we have them up to a multiplicative factor, so we just need to divide all the coefficients of the polynomials obtained by ω being the inverse of $2^{i+1} \mod p$. The reason is explained in [7]. Moreover, we have a array twice bigger than we want, the parts of the array where there were zeros before the FFT are know values which are useless. So we can come back again to an array of the size of the input polynomial.

To finish the step, we do a *semi_ add* as we did for small degree.

7 Next works

7.1 Critical look and improvements

Some procedure and functions I have implemented can be improved.

I think notably to the factorial procedure which has the default that in each step, some threads are calling the same elements of an array. This is not disturbing as it has a work of $\Theta(n\log(n))$ and this is called just one at the beginning. It is also possible that the way to compute the $(x+1)^{2^i}$ I had chosen is not the best, and if there exists a better way to compute this, maybe it doesn't need to compute the factorial sequence.

I can probably use more the shared memory and reduce the global work for example when I store several times the polynomials $(x+1)^{2^i}$ and I would like to take a look at the use of the global memory.

7.2 Prime numbers to consider

The Taylor_shift procedure I have implemented is done modulo a prime number p. According to the Section 2.6, we need to do this Taylor shift several times with different primes numbers and then use the Chinese Remainder Theorem to have the Taylor shift by 1 of the input polynomial in $\mathbb{Z}$.

Two questions arises :

(1) What prime numbers p must we use ?
(2) How many such primes numbers must we use ?

The works of Wei Pan (see [8]) put in evidence that primes numbers of the form $p = M \times 2^i + 1$ bring best performances for FFT than other prime number with i and M integers such that M is odd and $M < 2^i$. p must be also sufficiently greater than the degree of the incoming polynomial, otherwise the monomials of big degrees which compose the input polynomial won't be taken into account.

To find these prime numbers, we can run a MAPLE code which will be more adapted to deal with prime number with its pre-existing functions. Here is a prototype of such a code :

```
A:= Array(1..500):
p:= 962592769:
pow := 2^26:
pow2 := 2^19:
i:= 0:

  while ((i<400) and (p>pow)) do
    if ((p-1 mod pow2) = 0) then
      A[i] := p:
      i := i+1:
    end if:
    p := prevprime(p):
  end do:
```

This code can give a list of prime numbers, we have such a list but it does not contain sufficiently primes numbers for the moment. Now, let us explain how many such prime numbers we need. According to the calculation we just made, we can need approximatively 300 prime numbers for polynomials of degree 10000, so we need to create a table containing the prime numbers we want to use. The Taylor shift we implement will be called several times for these different prime numbers, and with an increasing size of the polynomial, we will need more or less calls of this procedure.

7.3 Combining several computations of the Taylor_shift procedure

Now the problem we are actually discussing is how to recombine the output solutions modulo prime numbers efficiently using the Chinese Remainder Theorem. In input, we will have $m_1, \ldots, m_s$ prime numbers and the s polynomials of size d shifted by 1 stored in an array $X[1:s][1:d]$ such that the s first positions of this array contain the first coefficient of each polynomial, then the second coefficient of each polynomial...
Let us consider $\mathbf{x} = (x_1, \ldots, x_s)$.

The objective is to compute the image a of $\mathbf{x}$ by $\mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_s\mathbb{Z} \cong \mathbb{Z}/m_1 \ldots m_s\mathbb{Z}$. According to [9], we can represent a by $\mathbf{b} = (b_1, \ldots, b_s)$ such that

$$a = b_1 + b_2 m_1 + b_3 m_1 m_2 + \cdots + b_s m_1 m_2 \ldots m_{s-1}$$

The idea of [9] is to compute a step by step using a mixed representation by a matrix formula. My work until this report ends here, but I'll explain how to use this radix representation when I will defend my internship.

8 Benchmarks

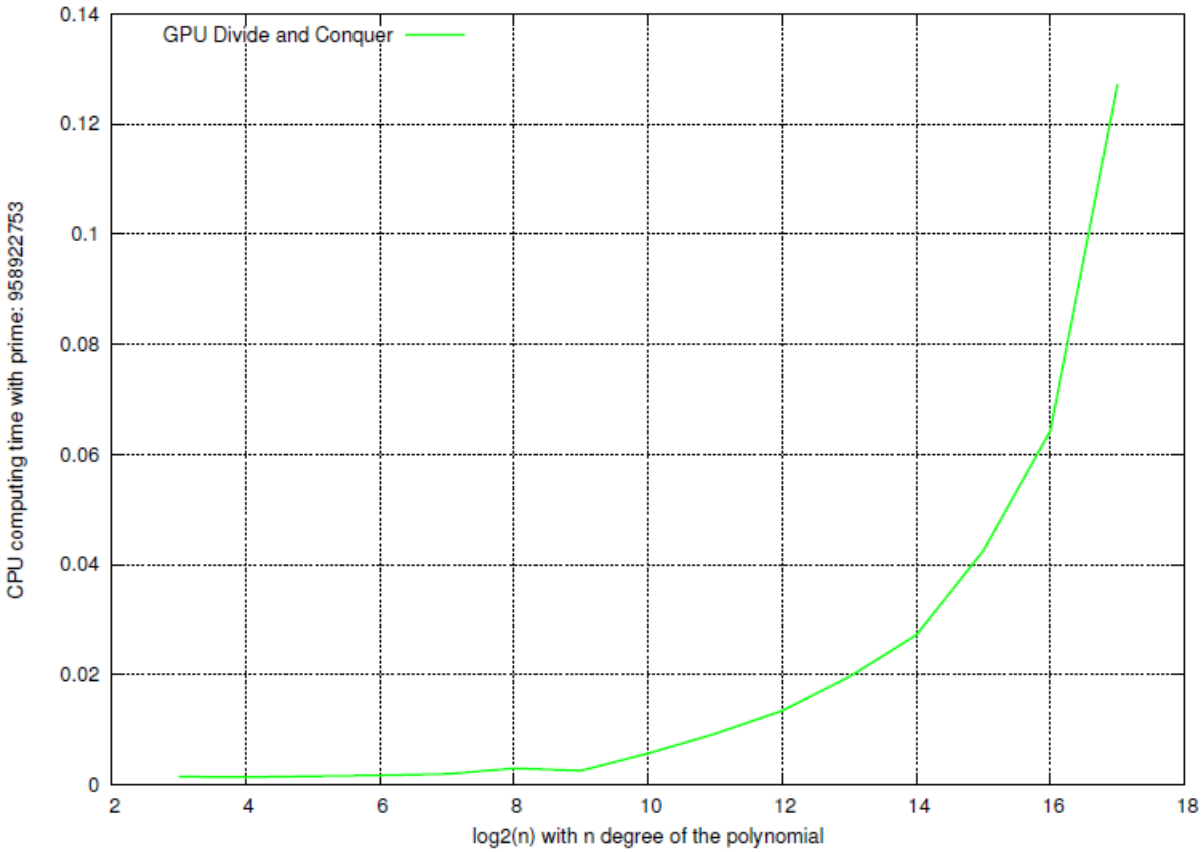
First of all, to be sure that the Taylor shift by 1 was done correctly, I needed some comparison, like using Maple, or other codes, in particular in C++. I used mainly the comparison of my CUDA code with the execution of the Horner's method in C++, which is the fastest compared to the Divide and Conquer method in C++ and the execution on Maple.

In the following graphics, MAPLE is not represented because the execution time increases by a factor of 4 each time we multiply by 2 the number of coefficients of the polynomial we want to be shifted by 1. Nevertheless, I show the execution time of the Maple script I used in the array of the execution times.

Here are the differents results obtained for different sizes of random polynomials. Let us recall that these polynomials are of degree $n = 2^e - 1$:

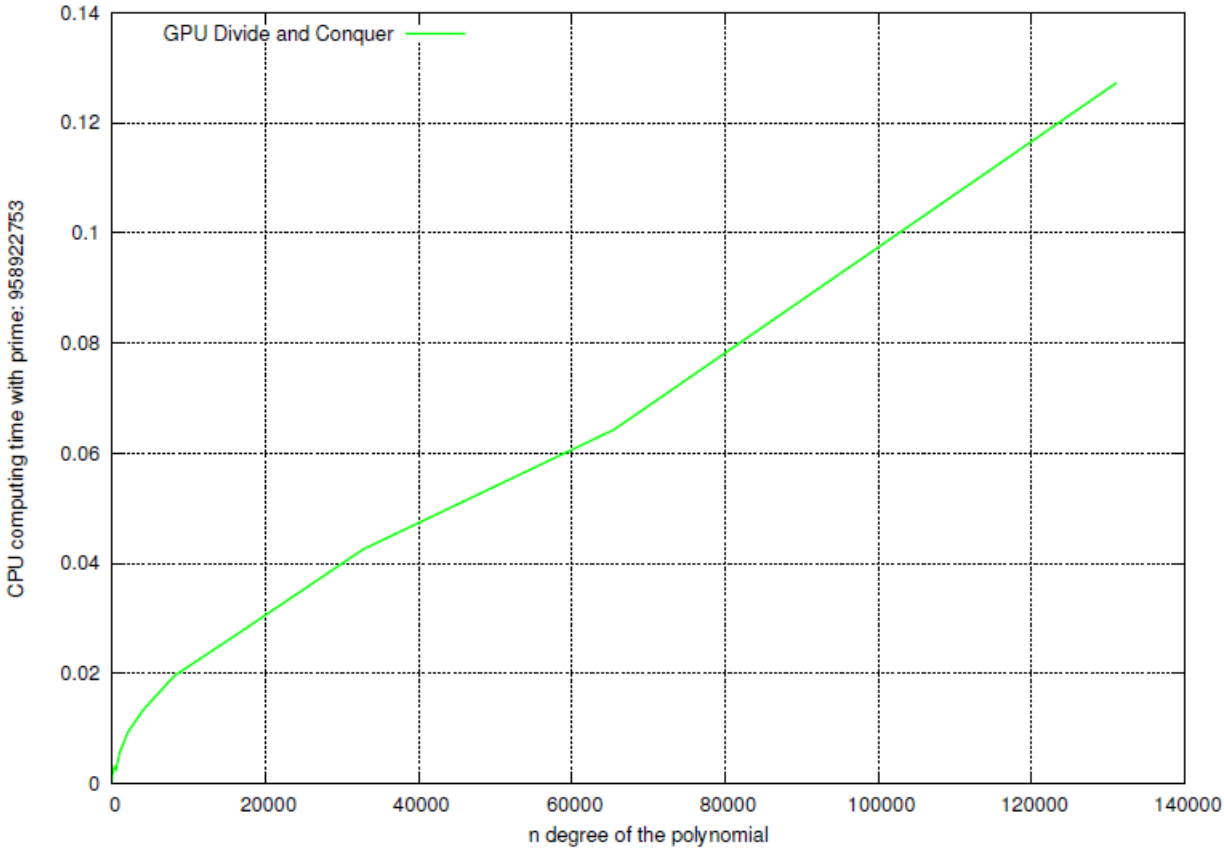
Execution time in seconds					
e	n	GPU	CPU : HOR	CPU : DNC	Maple 16
3	8	0.001518	0.000128	0.000141	<0.001
4	16	0.001432	0.000186	0.000172	<0.001
5	32	0.001590	0.000167	0.000191	<0.001
6	64	0.001773	0.000192	0.000294	0.008
7	128	0.002016	0.000261	0.000628	0.024
8	256	0.003036	0.000593	0.002331	0.084
9	512	0.002624	0.001278	0.006304	0.320
10	1024	0.005756	0.005940	0.032073	1.400
11	2048	0.009317	0.015312	0.095027	5.640
12	4096	0.013475	0.076866	0.376543	24.478
13	8192	0.019674	0.324029	1.498890	104.438
14	16384	0.027229	1.282708	6.861433	437.848
15	32768	0.042561	5.110919	23.907799	1781.427
16	65536	0.064306	15.184347	114.988129	7407.063
17	131072	0.127214	80.625801	477.934692	>10000

To compare more in details, take a look at the differents pictures we can have using gnuplot. Gnuplot scripts can be seen in the appendix. The execution times of MAPLE 16 don't appear as they increase too much compared to my other results. I show different graphics, considering e in x-axis then n, as my results are for polynomials of size 2^e. The results with n in x-axis are more eloquent concerning the fact that the execution time becomes linears when n increases by a factor of 2.



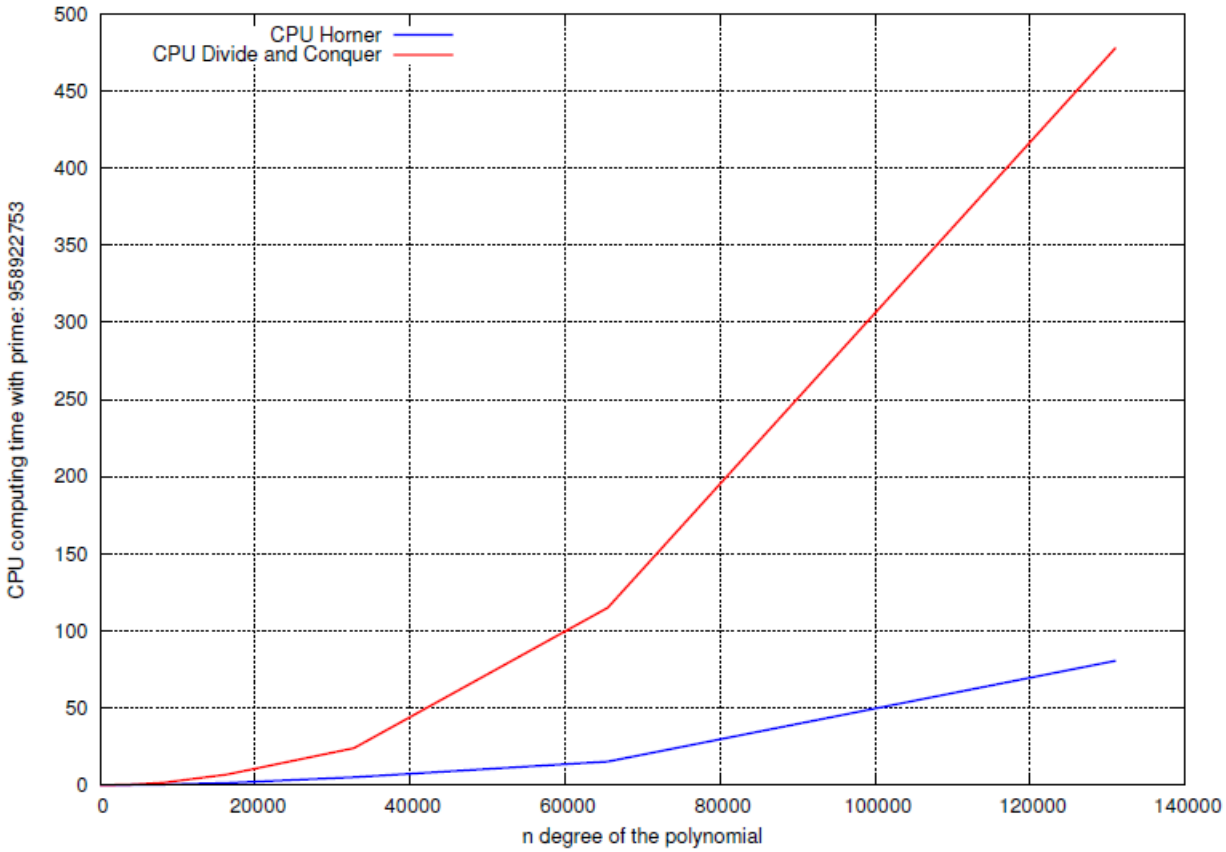
Execution time of the GPU in function of e

The following graphic is more eloquent concerning the linear increasing of the execution time of the GPU depending on the power of 2 the size of the polynomial is considered. Nevertheless, the execution is not really linear. Indeed, the cuda code contains also a part executed in serie, like when we read the polynomial in a file and also when we store the result in a file.



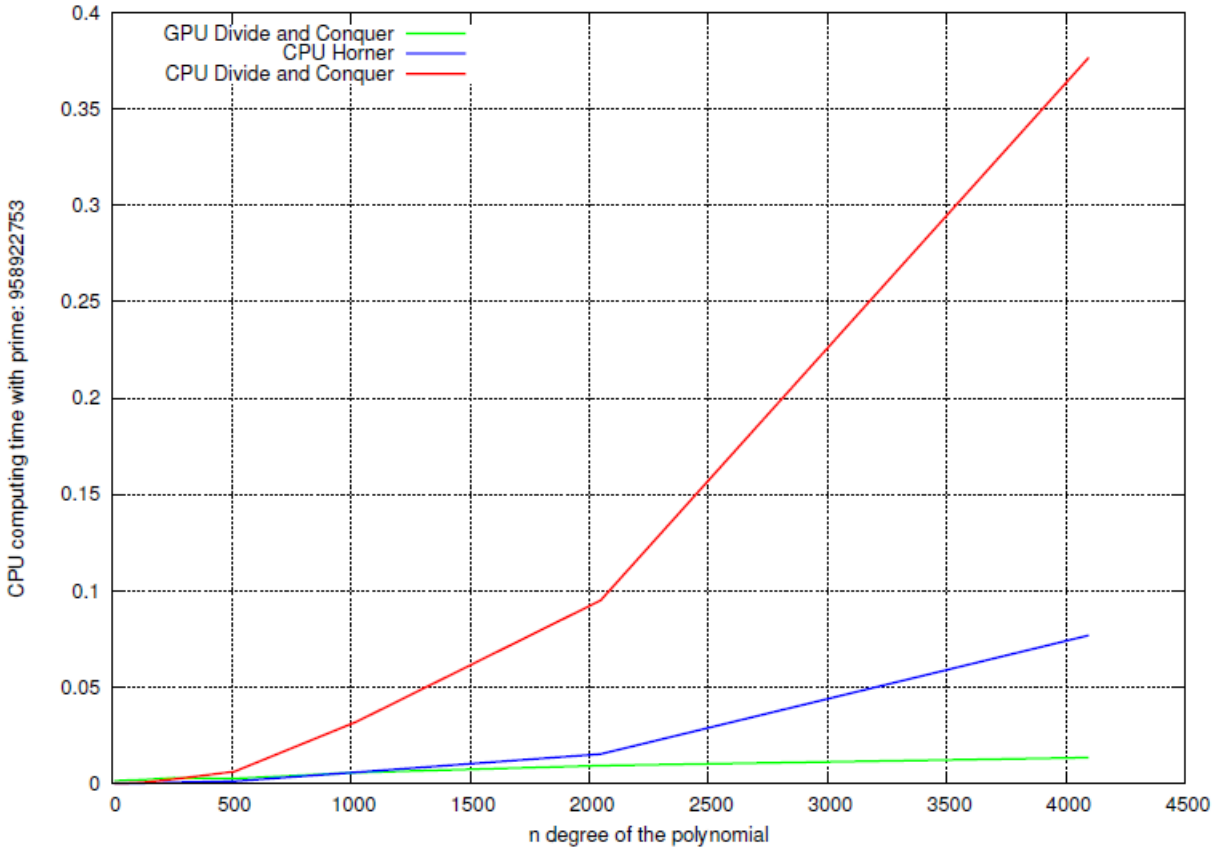
Execution time of the GPU in function of n

Now let's take a look at the execution times for the Horner's method and for the Divide and Conquer method in serie, so with the CPU only. The two methods are not linear and increase by a factor of 4 according to the previous array.



Execution time of the CPU in function of n

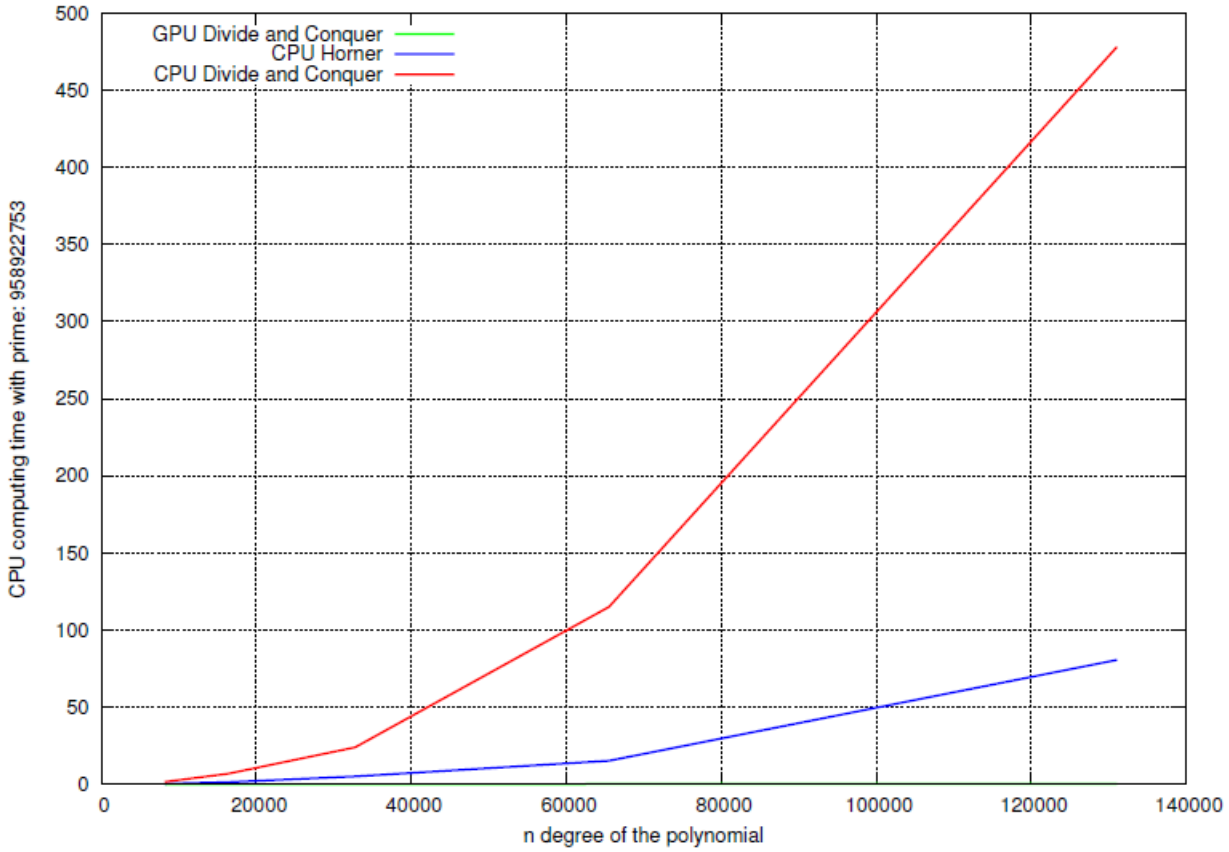
We clearly see that the Horner's method in serie is more efficient than the Divide and Conquer method in serie. To see exactly what happen for small degrees and big degree for the three methods, namely the GPU DNC method, the CPU DNC method and the CPU Horner's method (when I say GPU I mean 'in parallel', and when I say CPU I mean 'in serie'). Let's consider the two following graphics.



Execution times in function of n for small degrees

For small degrees, we see that the execution time of the GPU code is approximatively the same and worst than the execution times of the CPU execution times. The reasons are very simple, for small degrees, we need the same number of arrays in the CUDA code than for the big degrees, even though obviously they are not at the same size than for big degrees. Allocate memory for the GPU takes a lot of time at the scale of the execution times for small degrees, so finally there are a lot of communications just for 'small' computations. To conclude with this, for small degrees, it will be better to do the Taylor shift by 1 in serie. There is also the fact that we don't compute the same way the Taylor shift in the three methods, even though for the divide and conquer in serie, the only difference is the way we do the multiplication (and obviously the fact it is in serie).

Let's see the comparison for the next degrees :



Execution times in function of n for big degrees

For big degrees, using the GPU gives best performances with a execution time of approximatively 0.1 second for a polynomial of size 2^{17}. We see clearly a huge gap between the execution times of the different methods implemented, and remember that I didn't put the execution time using Maple also. We see that to parallelize the Taylor shift, is a real challenge, notably for MAPLE, which has finally the worst execution time for big degrees.

9 Conclusion

To realize something which seems simple at the first look becomes tricky when you want to parallelize it. You face problems and you must elaborate solutions using your knowledge, your own researchs and using the researchs of scientifics. This internship becomes me aware of the utility to be informed of the works and papers submitted by scientifics around the world.

To realize the Taylor shift by 1 as fast as possible is a part of a huge and difficult work. When this will be done thanks to the work on the mix radix representation, we will be able to continue towards the isolation of the real roots of a univariate polynomial, but we don't forget that we will have others step like to consider rational then real polynomials, and so considering others works, other algorithms like the VCA algorithm which is the algorithm we want to use.

When you call several times the same procedure, even if the execution time of this procedure is very small, then you can have a very high execution time. For example, to run a Taylor shift with my code for a polynomial of size 2^{17} modulo a prime number cost 0.1s approximatively. To obtain the Taylor shift in $\mathbb{Z}$ need to call this for more than 300 prime numbers, then finally we will have a cost of more than 30s for getting a Taylor shift in $\mathbb{Z}$. In the VCA algorithm, you use a lot this taylor shift, then you multiply your 30s again by a factor. Finally, even if you improve a little the procedures which are called several times, as the *mul_mod* procedure in my taylor shift code, you will have a huge gain of performances in you main code, that's why we do the maximum to parallelize everything can be parallelized in the best possible way.

10 Appendix

10.1 Divide and Conquer Cuda code

taylor_shift.cu

```
#include "taylor_shift_conf.h"
#include "inlines.h"
#include "taylor_shift_cpu.h"
#include "taylor_shift_kernel.h"
#include "taylor_shift.h"
#include "taylor_shift_fft.h"
#include "list_pointwise_mul.h"
#include "list_stockham.h"

/* Important to notice :

  n  : number of coefficients of the polynomial considered
 n-1 : degree of the polynomial considered
  p  : prime number, it must be greater than n
*/


// Taylor_shift procedure
void taylor_shift_GPU(sfixn n, sfixn e, char *file, sfixn p, double pinv)
{

  // declaration of variables
  sfixn i, nb_blocks, local_n;
  sfixn *Factorial_device;
  sfixn *Polynomial, *Polynomial_device;
  sfixn *Monomial_shift_device;
  sfixn *temp;
  sfixn *Mgpu;
  sfixn *Polynomial_shift_device[2];
  float cpu_time, gpu_time, outerTime;
  cudaEvent_t start, stop;     /* Initial and final time */

  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);

  // beginning parameters
  local_n = 2;
  stock_file_in_array(file, n, Polynomial);

//  printf("  * pinv = %0.20lf\n", pinv);

  // TIME
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&cpu_time, start, stop);
  cudaEventDestroy(stop);
```

```
cpu_time /= 1000.0;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

// Create the array Factorial
cudaMalloc( (void **) &Factorial_device, (n+1) * sizeof(sfixn) );
nb_blocks = number_of_blocks(n+1);
cudaThreadSynchronize();
identity_GPU<<<nb_blocks, NB_THREADS>>>(Factorial_device, n+1);
cudaThreadSynchronize();
nb_blocks = number_of_blocks(n/2);
create_factorial_step0_GPU<<<nb_blocks, NB_THREADS>>>(Factorial_device+1,\
n, e, p, pinv);
cudaThreadSynchronize();
sfixn L = 1;
for (i=1; i<e; i++)
{
  L *= 2;
  create_factorial_stepi_GPU<<<nb_blocks, NB_THREADS>>>(Factorial_device+1,\
  n, e, p, pinv, L);
  cudaThreadSynchronize();
}

// Create the array of the (x+1)^i
cudaMalloc( (void **) &Monomial_shift_device , n * sizeof(sfixn) ); // n+1
cudaThreadSynchronize();
nb_blocks = number_of_blocks(n);
develop_xshift_GPU<<<nb_blocks, NB_THREADS>>>(Monomial_shift_device, n, \
Factorial_device, p, pinv);
cudaThreadSynchronize();
cudaFree(Factorial_device);

/* ************************************************************

                  1st step : initialization

   ************************************************************ */

cudaMalloc( (void **) &Polynomial_device, n * sizeof(sfixn) );
cudaMemcpy( Polynomial_device, Polynomial, n*sizeof(sfixn), \
cudaMemcpyHostToDevice );
free(Polynomial);
cudaMalloc( (void **) &Polynomial_shift_device[0], n * sizeof(sfixn) );
cudaThreadSynchronize();

// initialize polynomial_shift
nb_blocks = number_of_blocks(n/2);
init_polynomial_shift_GPU<<<nb_blocks, NB_THREADS>>>(Polynomial_device, \
Polynomial_shift_device[0], n, p);
cudaThreadSynchronize();

/* ************************************************************
```

```
                        next steps (i<10)

   ********************************************************** */

sfixn polyOnLayerCurrent = n/2;
sfixn mulInThreadBlock;
cudaMalloc((void **)&Mgpu, n * sizeof(sfixn));
sfixn I = 9;
if (e < 9)
  I = e;
cudaMalloc( (void **) &Polynomial_shift_device[1], n * sizeof(sfixn) );
cudaThreadSynchronize();

// LOOP
for (i=1; i<I; i++)
{
  // transfer the polynomials which will be computed
  nb_blocks = number_of_blocks(n);
  transfert_array_GPU<<<nb_blocks, NB_THREADS>>>(Mgpu, \
  Polynomial_shift_device[(i+1)%2], Monomial_shift_device, n, local_n, p, pinv);
  cudaThreadSynchronize();

  // Compute the product of the polynomials in Mgpu ('P2 * Bin' with Bin
  // the array of binomials) and store them in Polynomial_shift_device[i%2]
  // shifted at the right for the multiplication by x so do
  // [( (x+1)^i - 1 ) / x ] * P2(x+1), then multiply it by x so we have
  //[(x+1)^i - 1] * P2(x+1)
  mulInThreadBlock = (sfixn) floor((double) NB_THREADS / (double) (2*local_n));
  nb_blocks = (sfixn) ceil(((double) polyOnLayerCurrent/(double)mulInThreadBlock)*0.5);
  listPlainMulGpu_and_right_shift_GPU<<<nb_blocks, NB_THREADS>>>(Mgpu,   \
  Polynomial_shift_device[i%2], local_n, polyOnLayerCurrent, 2*local_n, \
  mulInThreadBlock, p, pinv);
  cudaThreadSynchronize();

  // add [(x+1)^i - 1] * P2(x+1) with P2(x+1) then we get (x+1)^i * P2(x+1) \
     then do P1(x+1) + (x+1)^i*P2(x+1)
  nb_blocks = number_of_blocks(n/2);
  semi_add_GPU<<<nb_blocks, NB_THREADS>>>(Polynomial_shift_device[i%2], Mgpu, \
  Polynomial_shift_device[(i+1)%2], n, local_n, p);
  cudaThreadSynchronize();

  // for the next step
  polyOnLayerCurrent /= 2;
  local_n *= 2;
}

/* ********************************************************

                      next steps : FFT (i >= 10)

   ********************************************************** */
```

```
sfixn J = e;
if (e < 9)
  J = 9;
sfixn w;
sfixn *fft_device;
cudaMalloc( (void **) &fft_device, 2 * n * sizeof(sfixn) );
cudaThreadSynchronize();

// LOOP
for (i=9; i<J; i++)
{
  // transfer the polynomials which will be FFTed and Mgpu
  nb_blocks = number_of_blocks(n);
  transfert_array_GPU<<<nb_blocks, NB_THREADS>>>(Mgpu, \
  Polynomial_shift_device[(i+1)%2], Monomial_shift_device, n, local_n, p, pinv);
  cudaThreadSynchronize();
  nb_blocks = number_of_blocks(2*n);
  transfert_array_fft_GPU<<<nb_blocks, NB_THREADS>>>(fft_device, Mgpu, n, \
  local_n);
  cudaThreadSynchronize();

  // Convert the polynomials in the FFT world
  w = primitive_root(i+1, p);
  list_stockham_dev(fft_device, polyOnLayerCurrent, i+1, w, p);
  cudaThreadSynchronize();

  // same operation than for ListPlainMul but in the FFT world
  nb_blocks = number_of_blocks(2*n);
  list_pointwise_mul<<<nb_blocks, NB_THREADS>>>(fft_device, 2*local_n, p, pinv, \
  2*n);
  cudaThreadSynchronize();

  // return to the real world
  w = inv_mod(w, p);
  list_stockham_dev(fft_device, polyOnLayerCurrent, i+1, w, p);
  cudaThreadSynchronize();

  // adjust the real coefficients : we need to multiplicate by the following w
  // to have to correct size
  w = inv_mod(2*local_n, p);
  nb_blocks = number_of_blocks(n);
  mult_adjust_GPU<<<nb_blocks, NB_THREADS>>>(Polynomial_shift_device[i%2], \
  fft_device, n, local_n, w, p, pinv);
  cudaThreadSynchronize();

  // semi_add
  nb_blocks = number_of_blocks(n/2);
  semi_add_GPU<<<nb_blocks, NB_THREADS>>>(Polynomial_shift_device[i%2], Mgpu, \
  Polynomial_shift_device[(i+1)%2], n, local_n, p);
  cudaThreadSynchronize();

  // for the next steps
  polyOnLayerCurrent /= 2;
```

```
    local_n *= 2;
  }

  /* ************************************************************

                        end : results

     ************************************************************ */

  // Copy the last array containing the Taylor shift by 1 of the input polynomial
  temp = (sfixn*) malloc(n * sizeof(sfixn));
  cudaMemcpy( temp, Polynomial_shift_device[(e-1)%2], n*sizeof(sfixn), \
  cudaMemcpyDeviceToHost );
  cudaThreadSynchronize();

  // TIME
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&gpu_time, start, stop);
  cudaEventDestroy(stop);
  gpu_time /= 1000.0;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);

  // stockes the array of Newton's coefficients in a file
  char name_file[100];
  sprintf(name_file, "Pol%d.shiftGPU_%d.dat\0", e, p);
  stock_array_in_file(name_file, temp, n);

  // deallocation of the last arrays
  free(temp);
  cudaFree(Monomial_shift_device);
  cudaFree(Mgpu);
  cudaFree(fft_device);
  for (i=0; i<2; i++)
    cudaFree(Polynomial_shift_device[i]);

  // TIME
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&outerTime, start, stop);
  cudaEventDestroy(stop);
  outerTime /= 1000.0;
  cpu_time += outerTime;
}
```

taylor_shift_cpu.cu

```
#include "taylor_shift_conf.h"
#include "taylor_shift_cpu.h"

// error message if there is a lack of arguments
```

```
// to make the program
void error_message(sfixn m)
{
  if (m < 3)
    {
      printf("********** ERROR, not enough arguments ! \
      **********\nThe program works with the following parameters:\n\n");
      printf("1st parameter : file containing coefficients of the \
      polynomial you want to consider.\n");
      printf("2nd parameter : prime number p.\n");

      exit(1);
    }
}


// computes the nomber of blocks
sfixn number_of_blocks(sfixn n)
{
  sfixn res;
  res = n/NB_THREADS;
  if ( n % NB_THREADS != 0)
    res++;
  return res;
}


// stocks a file in an array
void stock_file_in_array(char* filename, sfixn n, sfixn* & a)
{
  ifstream data_file;
  sfixn i;
  data_file.open(filename);

  if (! data_file.is_open())
    {
      printf("\n Error while reading the file %s. Please check \
      if it exists !\n", filename);
      exit(1);
    }

  a = (sfixn*) malloc (n*sizeof(sfixn));

  for (i=0; i<n; i++)
    data_file >> a[i];

  data_file.close();
}


// stockes the array of Newton's coefficients in a file
void stock_array_in_file(const char *name_file, sfixn *T, sfixn size)
{
```

```
  sfixn i;
  FILE* file = NULL;

  file = fopen(name_file, "w+");
  if (file == NULL)
  {
    printf("error when opening the file !\n");
    exit(1);
  }

  // writting the file
  fprintf(file, "%d", T[0]);
  for (i=1; i<size; i++)
    fprintf(file, "\n%d", T[i]);
  fclose(file);
}


// computes the number of lines of a file
sfixn size_file(char* filename)
{
  sfixn size = 0;
  ifstream in(filename);
  std::string line;

  while(std::getline(in, line))
    size++;
  in.close();

  return size;
}


// display of an array
void display_array(sfixn *T, sfixn size)
{
  sfixn k;
  printf("[ ");
  for (k=0; k<size; k++)
    printf("%d ", T[k]);
  printf("] \n");
}


// addition of two arrays
void add_arrays(sfixn *res, sfixn *T1, sfixn *T2, sfixn size, sfixn p)
{
  sfixn i;
  for (i=0; i<size; i++)
    res[i] = (T1[i] + T2[i]) % p;
}
```

```
// Horner's method to compute g(x) = f(x+1) (equivalent to Shaw &
//Traub's method for a=1)
void horner_shift_CPU(sfixn *Polynomial, sfixn *Polynomial_shift, \
                      sfixn n, sfixn p)
{
  sfixn i;
  sfixn *temp;
  temp = (sfixn*) calloc (n, sizeof(sfixn));

  Polynomial_shift[0] = Polynomial[n-1];

  for (i=1; i<n; i++)
  {
    memcpy(temp+1, Polynomial_shift, i*sizeof(sfixn));
    add_arrays(Polynomial_shift, Polynomial_shift, temp, n, p);
    Polynomial_shift[0] = (Polynomial_shift[0] + Polynomial[n-1-i]) % p;
  }

  free(temp);
}
```

taylor_shift_kernel.cu

```
#include "taylor_shift_conf.h"
#include "taylor_shift_kernel.h"
#include "inlines.h"

// fast multiplication of two polynomials, created by
// Sardar Haque, I modify just a line to use it in my code
__global__ void listPlainMulGpu_and_right_shift_GPU(sfixn *Mgpu1,\
               sfixn *Mgpu2 , sfixn length_poly, sfixn poly_on_layer,\
               sfixn threadsForAmul, sfixn mulInThreadBlock, sfixn p,\
               double pinv)
{

__shared__ sfixn sM[2*Tmul];
/*
sM is the shared memory where the all the coefficients and intermediate
multiplications results are stored. For each multiplication it reserve
4*length_poly -1 spaces. mulID is the multiplication ID. It refers to
the poly in Mgpu2 on which it will work. mulID must be less than
(poly_on_layer/2).
*/
  sfixn mulID= ((threadIdx.x/threadsForAmul) + blockIdx.x*mulInThreadBlock);

  if (mulID < (poly_on_layer/2) && threadIdx.x < threadsForAmul*mulInThreadBlock)
  {
  /*
  The next 10 lines of code copy the polynomials in Mgpu1 from global memory to
  shared memory.
  Each thread is responsible of copying one coefficient.
  A thread will copy a coefficient from
  Mgpu1[( mulID* length_poly*2)...( mulID* length_poly*2) + length_poly*2 -1].
```

```
j+u gives the right index of the coefficient in Mgpu1.

In sM, the coefficients are stored at the lower part.
t will find the right (4*length_poly-1) spaced slot for it.
s gives the start index of its right slot.
s+u gives right position for the index.
*/

  sfixn j = ( mulID* length_poly*2);
  sfixn q = ( mulID*(2*length_poly));  // modified, clean the -1
  sfixn t = (threadIdx.x/threadsForAmul);
  sfixn u = threadIdx.x % threadsForAmul;

  sfixn s = t*(4*length_poly-1);
  sfixn k = s + length_poly;
  sfixn l = k + length_poly;
  sfixn c = l+u;
  sfixn a, b, i;

  sM[s+u] = Mgpu1[j + u];
  __syncthreads();

  if(u != (2*length_poly-1) )
  {
/*
In the multiplication space, the half of the leading coefficients
are computed differently than the last half. Here the computation of
first half are shown. the last half is shown in else statement.
In both cases sM[c] is the cofficient on which this thread will work on.
sM[a] is the coefficient of one poly.
sM[b] is the coefficient of the other poly.
*/
    if(u < length_poly)
    {
      a = s;
      b = k + u;
      sM[c] =  mul_mod(sM[a],sM[b],p,pinv);
      ++a; --b;

      for(i = 0; i < u; ++i, ++a, --b)
        sM[c] =  add_mod(mul_mod(sM[a],sM[b],p,pinv),sM[c] ,p);
        Mgpu2[q+u+1] = sM[c]; //+1 added
    }

    else
    {
      b = l - 1;
      a = (u - length_poly) + 1 + s;
      sM[c] =  mul_mod(sM[a],sM[b],p,pinv);
      ++a; --b;

      sfixn tempU = u;
      u = (2*length_poly-2) - u;
```

```
        for(i = 0; i < u; ++i, ++a, --b)
          sM[c] =  add_mod(mul_mod(sM[a],sM[b],p,pinv),sM[c] ,p);
        Mgpu2[q+tempU+1] = sM[c];   //+1 added
    }
  }

  else
    Mgpu2[q] = 0; // added for put 0 at position

  }
}

// create array identity (initialization of the array Fact)
__global__ void identity_GPU(sfixn *T, sfixn n)
{
  sfixn k = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn boolean = (sfixn) (k == 0);

  if (k < n+1)
    T[k] = k + boolean;
}

// create all the elements of Factorial (%p)
__global__ void create_factorial_GPU(sfixn *Fact, sfixn n, sfixn e,\
                                     sfixn p, double pinv)
// warning : n+1 is the size of Fact but we will just full the n
//last element, not the first one
{
  sfixn k = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn i, j, part, pos, base;
  sfixn L = 1;
  sfixn B = 2;

  if (k < n/2)
  {
    // step 1
    Fact[2*k+1] = mul_mod(Fact[2*k], Fact[2*k+1], p, pinv);
    __syncthreads();

    // next steps
    for (i=1; i<e; i++)
    {
      B *= 2;
      L *= 2;
      part = k / L;
      pos = k % L;
      __syncthreads();
      j = L + part*B + pos;
      __syncthreads();
      base = Fact[L + part*B - 1];
      __syncthreads();
      Fact[j] = mul_mod(base, Fact[j], p, pinv);
```

```
      __syncthreads();
    }
  }
}

__global__ void create_factorial_step0_GPU(sfixn *Fact, sfixn n, \
                                           sfixn e, sfixn p, double pinv)
// warning : n+1 is the size of Fact but we will just full the n
//last element, not the first one
{
  sfixn k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k < n/2)
  {
    // step 1
    Fact[2*k+1] = mul_mod(Fact[2*k], Fact[2*k+1], p, pinv);
  }
}

__global__ void create_factorial_stepi_GPU(sfixn *Fact, sfixn n, \
                                 sfixn e, sfixn p, double pinv, sfixn L)
// warning : n+1 is the size of Fact but we will just full the n last
// element, not the first one
{
  sfixn k = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn j, part, pos, base;
  sfixn B = 2 * L;

  if (k < n/2)
  {

    // next steps
      part = k / L;
      pos = k % L;
      j = L + part*B + pos;
      base = Fact[L + part*B - 1];
      Fact[j] = mul_mod(base, Fact[j], p, pinv);
  }
}

// create an array of the inverse numbers in Z/pZ
__global__ void inverse_p_GPU(sfixn *T, sfixn p, double pinv)
{
  sfixn i;
  sfixn k = blockIdx.x * blockDim.x + threadIdx.x;

  if (k < p)
  {
    if (k > 1)
      for (i=2; i<p; i++)
        {
          if (mul_mod(k, i, p, pinv) == 1)
          {
```

```
            T[k] = i;
            i = p;      // to stop the loop
          }
        }

    else if (k == 1)
      T[1] = 1;
    else // (k == 0)
      T[0] = 0;
  }
}

// create the inverse of a number in Z/pZ
__device__ sfixn inverse_GPU(sfixn k, sfixn p, double pinv)
{
  sfixn i, res;

  if (k > 1)
    for (i=2; i<p; i++)
      {
        if (mul_mod(k, i, p, pinv) == 1)
        {
          res = i;
          i = p;      // to stop the loop
        }
      }

  else if (k == 1)
    res = 1;
  else // (k == 0)
    res = 0;

  return res;
}

// creates an array of the Newton's Binomials until n modulo p
// (! size of the array = n+1)
__device__ sfixn create_binomial_GPU(sfixn *Factorial, sfixn *Inverse_p,\
                                     sfixn n, sfixn p, double pinv, sfixn id)
{
  sfixn l = n - id;
  sfixn temp = mul_mod(Factorial[id], Factorial[l], p, pinv);

  return mul_mod(Factorial[n], Inverse_p[temp], p, pinv);
}

// create the Newton's Binomial coefficient "n choose id" modulo p
// return "n choose id" = n! / [id!(n-id)!] mod p
__device__ sfixn create_binomial2_GPU(sfixn *Factorial, sfixn n, sfixn p, \
                                      double pinv, sfixn id)
{
  sfixn l = n - id;
  sfixn prod = mul_mod(Factorial[id], Factorial[l], p, pinv);
```

```
  return quo_mod(Factorial[n], prod, p, pinv);
}

// create the array of the coefficients of (x+1)^k for k in (1,2^(e-1))
__global__ void develop_xshift_GPU(sfixn *T, sfixn n, sfixn *Factorial, \
                                   sfixn p, double pinv)
{
  sfixn k = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn m;
  sfixn pow2 = 1;

  if (k < n)
  {
//     if (k > 1)
    {
      m = (k+1)/2; //k/2

      while (m != 0)
      {
        m /= 2;
        pow2 *= 2;
      }

      T[k] = create_binomial2_GPU(Factorial, pow2, p, pinv, k+1 - pow2);
    }
  }
}

// create the product of two arrays representing polynomials
__device__ void conv_prod_GPU(sfixn *res, sfixn *T1, sfixn *T2, sfixn m,\
                              sfixn p, sfixn local_n)
{
  sfixn i, j;
  sfixn K = blockIdx.x * blockDim.x + threadIdx.x;

  if (K < m)
  {
    for (j=0; j<K; j++)
    {
      i = K - j;            // K = i+j
// if i < local_n + 1 then T1[i] != 0, else T1[i] = 0 so useless computations
      if ((i < local_n+1) && (j < local_n))
        res[K] = (res[K] + T1[i]*T2[j]) % p;
    }

    for (j=K+1; j<m; j++)
    {
      i = K + m - j;
      if ((i < local_n+1) && (j < local_n))
        res[K] = (res[K] + T1[i]*T2[j]) % p;
    }
  }
```

```
}

// addition of two arrays
__global__ void add_arrays_GPU(sfixn *res, sfixn *T1, sfixn *T2, sfixn size,\
                               sfixn p)
{
  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i<size)
    res[i] = add_mod(T1[i], T2[i], p);
}

// creates an array of zeros
__global__ void Zeros_GPU(sfixn *T, sfixn n)
{
  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < n)
    T[i] = 0;
}

// initialize Polynomial_shift
__global__ void init_polynomial_shift_GPU(sfixn *Polynomial, \
                    sfixn *Polynomial_shift, sfixn n, sfixn p)
{
  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn j = 2*i;
  if (i < n/2)
  {
//     if (i % 2 == 0)
       Polynomial_shift[j] = add_mod(Polynomial[j], Polynomial[j+1], p);
//     else // (i % 2 == 1)
       Polynomial_shift[j+1] = Polynomial[j+1];
  }

  /* EXAMPLE for n=8 :
     after this procedure :
     Polynomial_shift = [f0+f1, f1, f2+f3, f3, f4+f5, f5, f6+f7, f7] */
}

// transfer at each step the polynomials which need to be multiplicated
__global__ void transfert_array_GPU(sfixn *Mgpu, sfixn *Polynomial_shift,\
      sfixn *Monomial_shift, sfixn n, sfixn local_n, sfixn p, double pinv)
{
  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;
//  sfixn B = 2*local_n;
  sfixn pos, part, bool1, bool2;

//  __shared__ sfixn sM[NB_THREADS];

  /*                    EXAMPLE

  ____________________________________________________________
```

```
    ARRAY Polynomial_shift_device[i-1] considered

     _________ _________ _________ _________
    |         |         |         |         |
    |         |    X    |         |    Y    |
    |_________|_________|_________|_________|
      part=0    part=1    part=2    part=3

    local_n = size of a part
```

```
    ARRAY Mgpu[i] considered

     ___________________ ___________________
    |                   |                   |
    |   X      (x+1)^m  |   Y      (x+1)^m  |
    |___________________|___________________|
          PART=0              PART=1

    B = 2 * local_n = size of a PART
    m = local_n
```

We want to fill the array Mgpu[i] like this : the polynomials which need to be multiplicated by (x+1)^m are of odd part and we store them at the beginning of each PART of Mgpu[i]. The end of each part doesn't really contain (x+1)^m as we need arrays to be multiplicated, so we avoid the multiplication by 1. Thus the end of each PART contains exactly :

```
 [(x+1)^m - 1] / x = m + ... + x^(m-1)     {m elements}          */

  if (i < n)
  {
    part = i / local_n;
    pos  = i % local_n;       // i = part * local_n + pos
    bool2 = part % 2;  // = 0 or 1
    bool1 = 1 - bool2; // = 1 or 0, bool1 and bool2 are contraries

    Mgpu[i] = bool1 * Polynomial_shift[local_n+i] + \
                      bool2 * Monomial_shift[local_n+pos];
  }
}

__global__ void right_shift_GPU(sfixn *T, sfixn n)
{
  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn a;

  if (i < n)
  {
    a = T[i];
    __syncthreads();
```

```
    if (i < n-1)
      T[i+1] = a;
    else
      T[0] = 0;
  }
}

// add parts of three arrays between them
__global__ void semi_add_GPU(sfixn *NewPol, sfixn *PrevPol1, \
            sfixn *PrevPol2, sfixn n, sfixn local_n, sfixn p)
{
  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn part = i / local_n;
  sfixn pos = i % local_n;
  sfixn j = 2 * local_n * part + pos;
  sfixn res;

  if (i < n/2)
  {
      res = add_mod(PrevPol1[j], PrevPol2[j], p);
      NewPol[j] = add_mod(NewPol[j], res, p);
  }
}
```

taylor_shift_fft.cu

```
#include "taylor_shift_conf.h"
#include "taylor_shift_cpu.h"
#include "taylor_shift_kernel.h"
#include "taylor_shift.h"
#include "taylor_shift_fft.h"
#include "inlines.h"

__global__ void mult_adjust_GPU(sfixn *Polynomial_shift, sfixn *fft,
                sfixn n, sfixn local_n, sfixn winv, sfixn p, double pinv)
{
  /*                    EXAMPLE

  ____________________________________________________________

    ARRAY fft considered

     _______________ _______________ _______________ _______________
    |               |               |               |               |
    |  real coeffs  |    useless    |  real coeffs  |    useless    |
    |_______________|_______________|_______________|_______________|
        PART=0                        PART=2


    B = 2 * local_n =  size of a PART

  ____________________________________________________________

    ARRAY Polynomial_shift_device considered
```

```
        _____________________________
       |             |               |
       |  real coeffs  |  real coeffs  |
       |_____________|_____________|
            part=0          part=1

     B = 2 * local_n = size of a part
                                                                  */

  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn B = 2*local_n;

  if (i < n)
    {
      sfixn part = i / B;
      sfixn pos  = i % B;
      sfixn bool1 = (sfixn) (pos != 0);

      Polynomial_shift[i] = bool1 * mul_mod(winv, fft[2*B*part + pos-1], \
                                            p, pinv);
    }
}

// transfer at each step the polynomials which need to be multiplicated
__global__ void transfert_array_fft_GPU(sfixn *fft, sfixn *Mgpu, sfixn n, \
                                        sfixn local_n)
{
  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;
//  sfixn B = 2*local_n;
  sfixn part, pos, bool1, bool2;
  part = i / local_n;
  pos  = i % local_n;
  bool2 = part % 2;
  bool1 = 1 - bool2;

  if (i<2*n)
  {
//     if (part % 2 == 0)
      fft[i] = bool1 * Mgpu[(part/2)*local_n + pos];
//     else
//       fft[i] = 0;
  }
}

__global__ void full_monomial(sfixn *Mgpu, sfixn *Monomial_shift, \
sfixn n, sfixn local_n)
{
  sfixn i = blockIdx.x * blockDim.x + threadIdx.x;
  sfixn part = i / local_n;
  sfixn pos  = i % local_n;

  if (i < n)
  {
```

```
    if (part % 2 == 1)
      Mgpu[i] = Monomial_shift[pos];
  }
}
```

taylor_shift_conf.h

```
#ifndef _TAYLOR_SHIFT_CONF_H_
#define _TAYLOR_SHIFT_CONF_H_

// Libraries :
# include <stdlib.h>
# include <stdio.h>
# include <string.h>
# include <time.h>
# include <ctime>
# include <math.h>
# include <unistd.h>
# include <iostream>
# include <fstream>
using namespace std;


// Number of threads per block (size of a block) :
#define NB_THREADS 512

//#define MAX_LEVEL 25

typedef int sfixn;

const sfixn Tmul = 512;
//const int BASE_1 = 31;


// Debuging flags
//#define DEBUG 0

#endif // _TAYLOR_SHIFT_CONF_H_
```

File calling these procedures : testGPU.cu

This code calls the procedure taylor_shift included in the file *taylor_shift.cu* :

```
#include "taylor_shift.h"
#include "taylor_shift_cpu.h"
#include "taylor_shift_conf.h"
#include "taylor_shift_kernel.h"

int main(int argc, char* argv[])
{
  // temporal data
  float total_time;
  cudaEvent_t start, stop;      /* Initial and final time */

  // TIME
```

```
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);

  // declaration of variables
  int n, e, p;
  double pinv;
  char name_file[100];

  error_message(argc);
  p = atoi(argv[2]);
  pinv = (double) 1/p;
  n = size_file(argv[1]);
  e = (int) log2((double) n);

  sprintf(name_file, "Pol%d.shiftGPU_%d.dat\0", e, p);

  taylor_shift_GPU(n, e, argv[1], p, pinv);

  // TIME
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&total_time, start, stop);
  cudaEventDestroy(stop);
  total_time /= 1000.0;
  printf("%d %.6f ", e, total_time);

  return 0;
}
```

10.2 Horner's method C++ code

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include <iostream>
#include <fstream>
using namespace std;

/* Important to notice :

  n  : number of coefficients of the polynomial considered
 n-1 : degree of the polynomial considered
  p  : prime number, it must be greater than n
*/

// error message if there is a lack of arguments to make the program
void error_message(int m)
{
```

```
  if (m < 3)
    {
      printf("********** ERROR, not enough arguments ! **********\n\
              The program works with the following parameters:\n\n");
      printf("1st parameter : file containing coefficients of the \
              polynomial you want to consider.\n");
      printf("2nd parameter : prime number p.\n");

      exit(1);
    }
}

// function modulo (faster than using %p)
int double_mul_mod(int a, int b, int p, double pinv)
{
  int q = (int) ((((double) a) * ((double) b)) * pinv);
  int res = a * b - q * p;

  return (res < 0) ? (-res) : res;
}

// creates an array of the sequence of the factorials until n
// modulo p (! size of the array = n+1)
void create_factorial(int *Factorial, int n, int p, double pinv)
{
  int k;
  Factorial[0] = 1;
  Factorial[1] = 1;
  for (k=2; k<n+1; k++)
    Factorial[k] = double_mul_mod(k, Factorial[k-1], p, pinv);
}

// creates an array of the Newton's Binomials until n
// modulo p (! size of the array = n+1)
void create_binomial_CPU(int *Binomial, int *Factorial, \
                         int *Inverse_p, int n, int p, double pinv)
{
  int k, l;
  int temp;
  for (k=0; k<n+1; k++) // we create together two parts of the array Binomial
  {
    l = n-k;
    if (k>l)            // and finally this loop has just n/2 steps
      break;
    temp = double_mul_mod(Factorial[k], Factorial[l], p, pinv);
    Binomial[k] = double_mul_mod(Factorial[n], Inverse_p[temp], p, pinv);
    Binomial[l] = Binomial[k];
  }
}

// stocks a file in an array
void stock_file_in_array(char* filename, int n, int* & a)
{
```

```
  ifstream data_file;
  int i;
  data_file.open(filename);

  if (! data_file.is_open())
    {
      printf("\n Error while reading the file %s. Please check \
                 if it exists !\n", filename);
      exit(1);
    }

  a = (int*) malloc (n*sizeof(int));

  for (i=0; i<n; i++)
    data_file >> a[i];

  data_file.close();
}

// stockes the array of Newton's coefficients in a file
void stock_array_in_file(const char *name_file, int *T, int size)
{
  int i;
  FILE* file = NULL;

  file = fopen(name_file, "w+");
  if (file == NULL)
  {
    printf("error when opening the file !\n");
    exit(1);
  }

  // writting the file
  fprintf(file, "%d", T[0]);
  for (i=1; i<size; i++)
    fprintf(file, "\n%d", T[i]);
  fclose(file);
}

// computes the number of lines of a file
int size_file(char* filename)
{
  int size = 0;
  ifstream in(filename);
  std::string line;

  while(std::getline(in, line))
    size++;
  in.close();

  return size;
}
```

```
// display of an array
void display_array(int *T, int size)
{
  int k;
  printf("[ ");
  for (k=0; k<size; k++)
    printf("%d ", T[k]);
  printf("] \n");
}

// create an array of the inverse numbers in Z/pZ
void inverse_p(int *T, int p, double pinv)
{
  int i,j;
  T[0] = 0;
  T[1] = 1;
  for (i=2; i<p; i++)
    for (j=2; j<p; j++)
      if (double_mul_mod(i, j, p, pinv) == 1)
      {
        T[i] = j;
        T[j] = i;
        break;
      }
}

// create the array of the coefficients of (x+1)^k for several k
void develop_xshift(int *T, int e, int *Factorial, int *Inverse_p,\
                    int p, double pinv, int n)
{
  int i, j;
  int *bin;
  int power2_i = 2;
  T[0] = 1;               // for (x+1)^0
  T[1] = 1;               // for (x+1)^1
  T[n] = 1;

  for (i=2; i<e+1; i++)  // for (x+1)^i with i in (2,e)
  {
    bin = (int*) malloc((power2_i+1)*sizeof(int));
    create_binomial_CPU(bin, Factorial, Inverse_p, power2_i, p, pinv);
    for (j=0; j<power2_i; j++)
      T[power2_i+j] = bin[j];
    free(bin);
    power2_i *= 2;
  }
}

// create the product of two arrays representing polynomials
void conv_prod(int *res, int *T1, int *T2, int m, int p)
{
  int i, j, k;
```

```
  for (i=0; i<m; i++)
    for (j=0; j<m; j++)
    {
      k = (i+j) % m;
      res[k] = (res[k] + T1[i]*T2[j]) % p;
  }
}

// addition of two arrays
void add_arrays(int *res, int *T1, int *T2, int size, int p)
{
  int i;
  for (i=0; i<size; i++)
    res[i] = (T1[i] + T2[i]) % p;
}

// creates Polynomial_shift(x) = Polynomial(x+1)
void create_polynomial_shift_CPU(int *Polynomial, int *T, \
            int *Monomial_shift, int n, int p, double pinv, int local_n)
{
  int i, j;
  int *Temp1, *Temp2, *Temp3, *res;

  if (local_n != 1)
  {
    create_polynomial_shift_CPU(Polynomial, T, Monomial_shift, n, p, \
                                pinv, local_n/2);

    if (local_n != n)
    {
      Temp1 = (int*) calloc(2*local_n, sizeof(int));
      Temp2 = (int*) calloc(2*local_n, sizeof(int));
      Temp3 = (int*) calloc(2*local_n, sizeof(int));
      res   = (int*) malloc(2*local_n * sizeof(int));

      memcpy(Temp3, Monomial_shift + local_n, (local_n+1) * sizeof(int));

      for (j=0; j<n; j+=2*local_n)
      {
        memcpy(Temp1, T + j, local_n*sizeof(int));
        memcpy(Temp2, T + local_n + j, local_n*sizeof(int));
        for (i=0; i<2*local_n; i++)
          res[i] = 0;
        conv_prod(res, Temp3, Temp2, 2*local_n, p);
        add_arrays(res, res, Temp1, 2*local_n, p);
        memcpy(T + j, res, 2*local_n*sizeof(int));
      }

      free(Temp1);
      free(Temp2);
      free(Temp3);
      free(res);
    }
```

```
  }

  else
  {
    for (i=0; i<n; i+=2)
    {
      T[i] = Polynomial[i] + Polynomial[i+1];
      T[i+1] = Polynomial[i+1];
    }
  }
}

int add_mod(int a, int b, int p) {
    int r = a + b;
    r -= p;
    r += (r >> 31) & p;
    return r;
}

// Horner's method to compute g(x) = f(x+1) (equivalent to Shaw &
// Traub's method for a=1)
void horner_shift_CPU(int *Polynomial, int *Polynomial_shift, int n, int p)
{
  int i;
  int *temp;

  temp = (int*) calloc (n, sizeof(int));
  Polynomial_shift[0] = Polynomial[n-1];

  for (i=1; i<n; i++)
  {
    memcpy(temp+1, Polynomial_shift, i*sizeof(int));
    add_arrays(Polynomial_shift, Polynomial_shift, temp, n, p);
    Polynomial_shift[0] = add_mod(Polynomial_shift[0], Polynomial[n-1-i], p);
  }

  free(temp);
}

// MAIN
int main(int argc, char* argv[])
{
  // TIME
  cudaEvent_t start, stop;      /* Initial and final time */
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);

  int n, p, e;
  int *Polynomial, *Polynomial_shift2;
  float cpu_time;          /* Total time of the cpu in seconds */

  // beginning parameters
```

```
  error_message(argc);
  p = atoi(argv[2]);
  n = size_file(argv[1]);
  e = (int) log2((double) n);
  stock_file_in_array(argv[1], n, Polynomial);

  Polynomial_shift2 = (int*) calloc(n,sizeof(int));
  horner_shift_CPU(Polynomial, Polynomial_shift2, n, p);

  // save in files the polynomial_shift
  char name_file[100];
  sprintf(name_file, "Pol%d.shiftCPU_%d.dat\0", e, p);
  stock_array_in_file(name_file, Polynomial_shift2, n);

  free(Polynomial);
  free(Polynomial_shift2);

  // TIME
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&cpu_time, start, stop);
  cudaEventDestroy(stop);
  printf("%.6f ", cpu_time);

  return 0;
}
```

10.3 Divide and Conquer C++ code

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include <iostream>
#include <fstream>
using namespace std;

typedef int sfixn;
const int BASE_1 = 31;

/* Important to notice :

  n  : number of coefficients of the polynomial considered
 n-1 : degree of the polynomial considered
  p  : prime number, it must be greater than n
*/

sfixn add_mod(sfixn a, sfixn b, sfixn p) {
    sfixn r = a + b;
    r -= p;
    r += (r >> BASE_1) & p;
```

```
    return r;
}

sfixn mul_mod(sfixn a, sfixn b, sfixn n, double ninv) {

    sfixn q  = (sfixn) ((((double) a) * ((double) b)) * ninv);
    sfixn res = a * b - q * n;
    res += (res >> BASE_1) & n;
    res -= n;
    res += (res >> BASE_1) & n;
    return res;
}

void egcd(sfixn x, sfixn y, sfixn *ao, sfixn *bo, sfixn *vo) {
    sfixn t, A, B, C, D, u, v, q;

    u = y; v = x;
    A = 1; B = 0;
    C = 0; D = 1;

    do {
        q = u / v;
        t = u;
        u = v;
        v = t - q * v;
        t = A;
        A = B;
        B = t - q * B;
        t = C;
        C = D;
        D = t - q * D;
    } while (v != 0);

    *ao = A;
    *bo = C;
    *vo = u;
}

sfixn inv_mod(sfixn n, sfixn p) {
    sfixn a, b, v;
    egcd(n, p, &a, &b, &v);
    if (b < 0) b += p;
    return b % p;
}

sfixn quo_mod(sfixn a, sfixn b, sfixn n, double ninv) {
    return mul_mod(a, inv_mod(b, n), n, ninv);
}

// error message if there is a lack of arguments to make the program
void error_message(int m)
{
  if (m < 3)
```

```
    {
      printf("********** ERROR, not enough arguments ! **********\n\
              The program works with the following parameters:\n\n");
      printf("1st parameter : file containing coefficients of the \
              polynomial you want to consider.\n");
      printf("2nd parameter : prime number p.\n");

      exit(1);
    }
}

// function modulo (faster than using %p)
int double_mul_mod(int a, int b, int p, double pinv)
{
  int q = (int) ((((double) a) * ((double) b)) * pinv);
  int res = a * b - q * p;

  return (res < 0) ? (-res) : res;
}

// creates an array of the sequence of the factorials until n
// modulo p (! size of the array = n+1)
void create_factorial(int *Factorial, int n, int p, double pinv)
{
  int k;
  Factorial[0] = 1;
  Factorial[1] = 1;
  for (k=2; k<n+1; k++)
    Factorial[k] = double_mul_mod(k, Factorial[k-1], p, pinv);
}

// creates an array of the Newton's Binomials until n
// modulo p (! size of the array = n+1)
void create_binomial_CPU(int *Binomial, int *Factorial, \
                         int n, int p, double pinv)
{
  int k, l;
  int temp;
  for (k=0; k<n+1; k++) // we create together two parts of the array Binomial
  {
    l = n-k;
    if (k>l)          // and finally this loop has just n/2 steps
      break;
    temp = mul_mod(Factorial[k], Factorial[l], p, pinv);
    Binomial[k] = quo_mod(Factorial[n], temp, p, pinv);
    Binomial[l] = Binomial[k];
  }
}

// stocks a file in an array
void stock_file_in_array(char* filename, int n, int* & a)
{
  ifstream data_file;
```

```
  int i;
  data_file.open(filename);

  if (! data_file.is_open())
    {
      printf("\n Error while reading the file %s. Please \
                check if it exists !\n", filename);
      exit(1);
    }

  a = (int*) malloc (n*sizeof(int));

  for (i=0; i<n; i++)
    data_file >> a[i];

  data_file.close();
}

// stockes the array of Newton's coefficients in a file
void stock_array_in_file(const char *name_file, int *T, int size)
{
  int i;
  FILE* file = NULL;

  file = fopen(name_file, "w+");
  if (file == NULL)
  {
    printf("error when opening the file !\n");
    exit(1);
  }

  // writting the file
  fprintf(file, "%d", T[0]);
  for (i=1; i<size; i++)
    fprintf(file, "\n%d", T[i]);
  fclose(file);
}

// computes the number of lines of a file
int size_file(char* filename)
{
  int size = 0;
  ifstream in(filename);
  std::string line;

  while(std::getline(in, line))
    size++;
  in.close();

  return size;
}

// display of an array
```

```
void display_array(int *T, int size)
{
  int k;
  printf("[ ");
  for (k=0; k<size; k++)
    printf("%d ", T[k]);
  printf("] \n");
}

// create an array of the inverse numbers in Z/pZ
void inverse_p(int *T, int p, double pinv)
{
  int i,j;
  T[0] = 0;
  T[1] = 1;
  for (i=2; i<p; i++)
    for (j=2; j<p; j++)
      if (mul_mod(i, j, p, pinv) == 1)
      {
        T[i] = j;
        T[j] = i;
        break;
      }
}

// create the array of the coefficients of (x+1)^k for several k
void develop_xshift(int *T, int e, int *Factorial, int p, double pinv, int n)
{
  int i, j;
  int *bin;
  int power2_i = 2;
  T[0] = 1;                // for (x+1)^0
  T[1] = 1;                // for (x+1)^1
  T[n] = 1;

  for (i=2; i<e+1; i++)  // for (x+1)^i with i in (2,e)
  {
    bin = (int*) malloc((power2_i+1)*sizeof(int));
    create_binomial_CPU(bin, Factorial, power2_i, p, pinv);
    for (j=0; j<power2_i; j++)
      T[power2_i+j] = bin[j];
    free(bin);
    power2_i *= 2;
  }
}

// create the product of two arrays representing polynomials
void conv_prod(int *res, int *T1, int *T2, int m, int p, double pinv)
{
  int i, j, k, temp;

  for (i=0; i<m; i++)
    for (j=0; j<m; j++)
```

```
    {
      k = add_mod(i,j,m);
      temp = mul_mod(T1[i], T2[j], p, pinv);
      res[k] = add_mod(res[k], temp, p);
  }
}

// addition of two arrays
void add_arrays(int *res, int *T1, int *T2, int size, int p)
{
  int i;
  for (i=0; i<size; i++)
    res[i] = add_mod(T1[i], T2[i], p);
}

// creates Polynomial_shift(x) = Polynomial(x+1)
void create_polynomial_shift_CPU(int *Polynomial, int *T, \
     int *Monomial_shift, int n, int p, double pinv, int local_n)
{
  int i, j;
  int *Temp1, *Temp2, *Temp3, *res;

  if (local_n != 1)
  {
    create_polynomial_shift_CPU(Polynomial, T, Monomial_shift, n, \
                                p, pinv, local_n/2);

    if (local_n != n)
    {
      Temp1 = (int*) calloc(2*local_n, sizeof(int));
      Temp2 = (int*) calloc(2*local_n, sizeof(int));
      Temp3 = (int*) calloc(2*local_n, sizeof(int));
      res   = (int*) malloc(2*local_n * sizeof(int));

      memcpy(Temp3, Monomial_shift + local_n, (local_n+1) * sizeof(int));

      for (j=0; j<n; j+=2*local_n)
      {
        memcpy(Temp1, T + j, local_n*sizeof(int));
        memcpy(Temp2, T + local_n + j, local_n*sizeof(int));
        for (i=0; i<2*local_n; i++)
          res[i] = 0;
        conv_prod(res, Temp3, Temp2, 2*local_n, p, pinv);
        add_arrays(res, res, Temp1, 2*local_n, p);
        memcpy(T + j, res, 2*local_n*sizeof(int));
      }

      free(Temp1);
      free(Temp2);
      free(Temp3);
      free(res);
    }
  }
```

```
  else // (local_n == 1)
  {
    for (i=0; i<n; i+=2)
    {
      T[i] = add_mod(Polynomial[i], Polynomial[i+1], p);
      T[i+1] = Polynomial[i+1];
    }
  }
}

// MAIN
int main(int argc, char* argv[])
{
  float cpu_time;         /* Total time of the cpu in seconds */
  // TIME
  cudaEvent_t start, stop;     /* Initial and final time */
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);

  int n, p, e;
  int *Factorial, *Binomial;
  int *Polynomial, *Polynomial_shift1;
  int *Monomial_shift;
  double pinv;

  // beginning parameters
  error_message(argc);
  p = atoi(argv[2]);
  pinv = (double) 1/p;
  n = size_file(argv[1]);
  e = (int) log2((double) n);
  stock_file_in_array(argv[1], n, Polynomial);

  // allocation of memory
  Factorial         = (int*) malloc((n+1)*sizeof(int));
  Binomial          = (int*) malloc((n+1)*sizeof(int));
  Monomial_shift    = (int*) malloc((n+1)*sizeof(int));
  Polynomial_shift1 = (int*) calloc(n,sizeof(int));

  // instructions
  create_factorial(Factorial, n, p, pinv);
  create_binomial_CPU(Binomial, Factorial, n, p, pinv);
  develop_xshift(Monomial_shift, e, Factorial, p, pinv, n);
  create_polynomial_shift_CPU(Polynomial, Polynomial_shift1, \
                              Monomial_shift, n, p, pinv, n);

  // save in files the polynomial_shift
  char name_file[100];
  sprintf(name_file, "Pol%d.shiftCPU_%d.dat\0", e, p);
  stock_array_in_file(name_file, Polynomial_shift1, n);
```

```
  // free memory
  free(Factorial);
  free(Binomial);
  free(Polynomial);
  free(Polynomial_shift1);
  free(Monomial_shift);

  // TIME
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&cpu_time, start, stop);
  cudaEventDestroy(stop);
  cpu_time /= 1000.0;

  printf("%.6f ", cpu_time);

  return 0;
}
```

10.4 Maple code

The procedure we describe in this part makes the taylor shift by one of a polynomial *pp* of variable x in input modulo *prime* which is a prime number. We can create a random polynomial *pp* with the maple instruction *randpoly*, for example :
pp := randpoly(x, degree $= 2^{10} - 1$*, terms* $= 2^{10}$*) :*

Then we can run the following procedure with *pp* and a prime number we choose (958922753 as for the other examples). Here is the procedure :

```
xplus1 := proc(pp,x,prime)

local p, L, d, i, j, res, st, cputime:
  st := time():
  p := collect(pp, x):
  d := degree(p, x):
  L := Array(1..d+1):

  for i from d to 0 by -1 do
    L[d+1-i] := coeff(p, x, i):
  od:

  for i from 1 to d do
    for j from 2 to d-i+2 do
      L[j] := (L[j-1] + L[j]) mod prime:
    od:
  od:

  res := 0:
  for i from 1 to d+1 do
    res := (res + L[i]*x^(d+1-i)) mod prime:
  od:

  cputime := time() - st;
```

```
  # return cputime:
  return res;
end:
```

References

[1] Changbo Chen, Marc Moreno Maza, and Yuzhen Xie. Cache complexity and multicore implementation for univariate real root isolation. *Journal of Physics : Conferences Series*, 2010.

[2] Pavel Emeliyanenko. *Advanced Parallel Processing Technologies : 8th International Symposium, Efficient Multiplication of Polynomials on Graphics Hardware*. Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2009.

[3] Pavel Emeliyanenko. High-performance polynomial gcd computations on graphics processors. pages 134–149, 2010.

[4] Joachim Von Zur Gathen and Jürgen Gerhard. *Fast algorithms for Taylor shifts and certain difference equations*. Universität-GH Paderborn D-33095 Paderborn, Germany, 1997.

[5] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern Computer Algebra 2nd edition*. Universität-GH Paderborn D-33095 Paderborn, Germany, 2003.

[6] A.B.M. Zunaid Haque. *Multi-threaded real root isolation on multi-core architectures*. PhD thesis, University of Western Ontario, London, Ontario, Canada, 2012.

[7] Marc Moreno Maza. Foundations of computer algebra : Fast polynomial multiplication. 2008.

[8] Wei Pan. *Algorithmic Contributions to the Theory of Regular Chains*. PhD thesis, University of Western Ontario, London, Ontario, Canada, 2011.

[9] J. Schönheim. *Conversion of Modular Numbers to their Mixed Radix representation by a Matrix Formula*. Tel-Aviv University, Ramat-Aviv, Tel-Aviv, Israel, 1967.