# Real root isolation for univariate polynomials on GPUs and multicores

Alex Temperville [1] Marc Moreno Maza[2]

[1]University of Lille 1
[2]University of Western Ontario

June 22th 2012

# University of Western Ontario

**Direction of the research activities of Marc Moreno Maza's team :**

- 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.

# Main purpose of the Lab

The laboratory works in close cooperation with *Maplesoft*.

**Current main purpose of the laboratory :**

- Provide Maple's end-users with symbolic computation tools.
- Take advantage of hardware acceleration technologies, using GPUs and multicores, using the best computer ressources.
- Develop the library *cumodp* which will be integrated into Maple so as to provide fast arithmetic operations over prime fields.

# Purpose of my internship

- contribute to the library *cumodp*
- develop code for exact calculation of the real roots of univariate polynomials
- use mathematical tools : *Descartes' rule of signs*, *Fast Fourier transform*, computing by *homomorphic images*...
- realize algorithms using GPUs

# Descartes' rule of sign

## Descartes' rule of signs (DRS)

*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

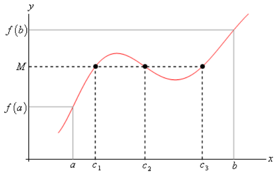
*If we consider the previous rule of signs and count the roots with their multiplicities, then the number of positive real roots of* $P$ *has the same parity as* $c$*.*

This rule can also be used to determine the number of negative real roots of polynomials.

# Intermediate values theorem

## Intermediate values theorem (IVT)

*If $f$ is a real-valued continuous function on the interval $[a, b]$ and $M$ is a number between $f(a)$ and $f(b)$, then there exists $c \in [a, b]$ such that we have $f(c) = M$.*




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

# Horner's method

- We want to evaluate the following polynomial :

$$P(X) = \sum_{i=0}^{n} a_i \, X^i = a_0 + a_1 \, X + a_2 \, X^2 + \cdots + a_n \, X^n$$

A better way to evaluate it is to use **Horner's method**, that is, represents $P$ in the following form :

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

- *costs of operations :* $\Theta(n)$ for *Horner's method*, $\Theta(n^2)$ for naive method.

# Example of real root search

Let's consider $P(X) = X^3 + 3X^2 - X - 2$ : $c = 1$ so $P$ has 1 positive root.
$P(-X) = -X^3 + 3X^2 + X - 2$ : $c = 2$ so $P$ has 2 or 0 negative roots.

$P(-1) = 1$, $\lim_{X \to -\infty} P(X) = -\infty \Rightarrow \exists r_1 \in ]-\infty; -1[ \ \mid P(r_1) = 0$
$\Rightarrow P$ has 2 negative roots.

Let's use the *IVT* for $M = 0$ :
The following evaluations of $P$ can be done using the *Horner's method*.
As $P(-1) = 1$ and $P(-2) = -2$, then $r_1 \in ]-2, -1[$.
As $P(0) = -2$ and $P(-1) = 1$, then $r_2 \in ]-1, 0[$.
As $P(0) = -2$ and $P(1) = 1$, then $r_3 \in ]0, 1[$.

# Vincent-Collins-Akritas' algorithm

The **Vincent-Collins-Akritas' algorithm** (*VCA*) computes a list of disjoint intervals with rational endpoints for a polynomial $P$ such that :

- each real root of $P$ belongs to a single interval, and
- each interval contains only one real root of $P$.

**Algorithm 1**: RealRoots(p)

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

```
begin
    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;
    if x | p then m := 1 else m := 0;
    p1 := p(2^k x);
    p2 := p1(−x);
    m' := RootsInZeroOne(p1);
    m := m + RootsInZeroOne(p2);
    return m + m';
end
```

**Algorithm 2**: RootsInZeroOne(p)

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

```
begin
    p1 := x^d p(1/x);
    p2 := p1(x + 1); //Taylor shift
    Let v be the number of sign variations of the coefficients of p2;
    if v ≤ 1 then return v;
    p1 := 2^d p(x/2);
    p2 := p1(x + 1); //Taylor shift
    if x | p2 then m := 1 else m := 0;
    m' := RootsInZeroOne(p1);
    m := m + RootsInZeroOne(p2);
    return m + m';
end
```

The algorithm 2 is called several times and inside it, *Taylor shift by* 1 also
$\Longrightarrow$: *Taylor shift by* 1 needs to be optimized at the lowest cost possible.

# Modular arithmetic

**A current problem :** expressions in the coefficients swell when computing with polynomial or matrices over a field ($\mathbb{Z}$ e.g.) $\Rightarrow$ performance bottleneck for computer algebra.

**Two solutions :**

- use highly optimized multiprecision libraries (e.g. Gmp), and
- compute by homomorphic images.

**Ways to compute by homomorphic images :**

- use the *Chinese Remainder Theorem*, or
- use the *Hensel's Lemma*.

# Chinese Remainder Theorem (1)

## Chinese Remainder Theorem $1^{st}$ version (CRT1)

*Consider $m_1, m_2, \ldots, m_r$ a sequence of $r$ positive integers which are pairwise coprime. 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 $(S)$ has a unique solution modulo $M = m_1 \times m_2 \times \cdots \times m_r$ :*

$$x = a_1 \times M_1 \times y_1 + a_2 \times M_2 \times y_2 + \cdots + a_r \times M_r \times y_r$$

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

# Chinese Remainder Theorem (2)

### 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 absolutely bounded by $B \in \mathbb{N}$, then the coefficients of $g = f(x + a) \in \mathbb{Z}[x]$ are absolutely bounded by $B(|a| + 1)^n$.*

Our future results will be given in $\mathbb{Z}[x]/M\mathbb{Z}$. Thanks to lemma, if $M$ is sufficiently big, we could consider our results in $\mathbb{Z}[x]$. The algebraic form of the *Chinese Remainder Theorem* will be also used for *homomorphic images* :

### 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 :*

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

# Taylor shift by $a$

## Definition

The **Taylor shift by $a$** of a polynomial $P \in R[x]$, with $R$ a field (e.g., $\mathbb{Z}$) consists of 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 $g_0,\dots,g_n \in \mathbb{Z}$ 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 are several methods to compute this, the classical ones deal with the *Horner's method*. There are also asymptotically fast methods, and in particular *Divide & Conquer method (D&C)* which we will use.

# Divide & Conquer method (D&C)

**Definition**

The size of a polynomial of degree $d$ is $d+1$.

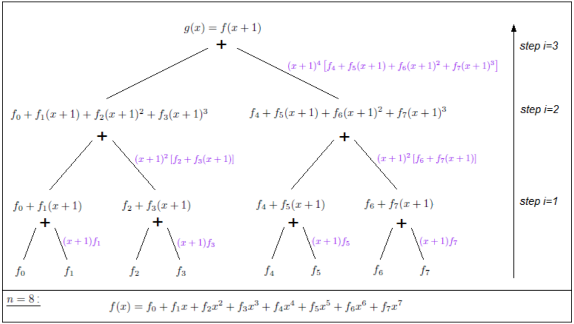
size of the polynomials considered : $n = 2^e$ (so degree : $d = 2^e - 1$)

**The Divide & Conquer method consists of :**

1. split the polynomial as : $P(x) = P^{(0)}(x+1) + (x+1)^{n/2} \times P^{(1)}(x+1)$
2. evaluate $P^{(0)}(x+1)$ and $P^{(1)}(x+1)$ recursively
3. compute a product when $P^{(1)}(x+1)$ is evaluated
4. compute a sum when $P^{(0)}(x+1)$ and the product are evaluated

These are the four main things to implement to realize the *Taylor shift by* 1.

# D&C tree for $n = 8 = 2^3$




D&C computing tree

In parallel, we consider the tree by levels (from its base) and not recursively.

.

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

For multiplication, we need to compute the polynomials $(x+1)^{2^i}$ for $i \in [\![0, e-1]\!]$.

## Consequence of the binomial theorem

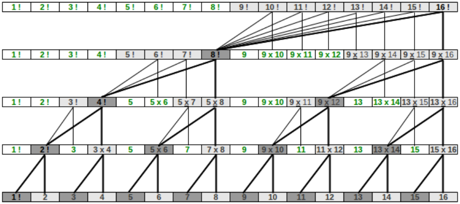
*According to the binomial theorem, we have in general :*

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

**Consequence :** we need to compute the sequence $(i!)_{0 \leq i \leq n}$.

# Factorial sequence

We compute only the sequence $(i!)_{1 \leq i \leq n}$ (power of 2).
*Notation :* $a \underline{\times} b = \prod_{k=a}^{b} k$ (e.g. $9 \underline{\times} 13 = 9 \times 10 \times 11 \times 12 \times 13$).




**Mapping of the computation of the sequence $(i!)_{1 \leq i \leq 16}$**

$n/2$ threads do products using a "pillar" factor (in the darkest boxes).
**Work :** $\Theta(n\, log(n))$

# Store the $(x+1)^{2^i}$s

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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*

**Array *Monomial_shift_device* for $n = 16 = 2^4$ :**

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

**At step $i$, $local\_n = 2^i$ :**

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

# Divide & Conquer method

Recall how we can realize the *D&C method*.

**The Divide & Conquer method consists of :**

1. split the polynomial as : $P(x) = P^{(0)}(x+1) + (x+1)^{n/2} \times P^{(1)}(x+1)$
2. evaluate $P^{(0)}(x+1)$ and $P^{(1)}(x+1)$ recursively
3. compute a product when $P^{(1)}(x+1)$ is evaluated
4. compute a sum when $P^{(0)}(x+1)$ and the product are evaluated

These are the four main things to implement to realize the *Taylor shift by* 1. We will only focus on the multiplication, which is the most hard and tricky operation to realize our code.

# Multiplication : concept

- **Small sizes ($n \leq 512$) :**
  We can use a procedure called *list_Plain_Mul* which computes a list of pairwise products of polynomials of <u>same size</u>.
- **Big sizes ($n > 512$) :**
  We can use *FFT* whic several procedures which compute a list pairwise products of polynomials of <u>same size</u>.

**Polynomials considered at step $i$ :**
Polynomials $P^{(1)}$ (of size $2^i$) and $(x+1)^{2^i}$ (of size $2^i + 1$).

# Multiplication : a challenge

**Problems :**

- Polynomials considered must be of the same size (rather $2^i$) so as to use the procedures of the lab.
- Size of the product of two such polynomials is not a power of 2 (but $2^{i+1} - 1$).

**Solutions :**

- Modify procedure *list_Plain_Mul* and adapt it for my case.
- Consider another product with polynomials of same sizes.

# Multiplication : decomposition

We can decompose the product desired as follows :

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

# Multiplication : technic

## Taylor shift multiplication concept

*Using the following formula :*

$$P^{(1)}(X) \times (X+1)^{2^i} = X \cdot \left( P^{(1)}(X) \times \frac{(X+1)^{2^i} - 1}{X} \right) + P^{(1)}(X)$$

*The multiplication desired amounts to :*

1. *multiplying* $P^{(1)}(X)$ *by* $[(X+1)^{2^i} - 1]/X$ *of sizes* $2^i$,
2. *doing a right shift (multiplication by* $X$*), and*
3. *semi-adding the result of the two first steps with* $P^{(1)}(X)$.

We will just detail how to do the multiplication for the two cases of polynomial sizes.

# Multiplication : arrays considered (artificial ex.)

If we consider polynomials at step 1 :

*Polynomial_shift_device*[0] =

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Then for the two multiplication techniques used, we consider :

- **"Small sizes" ($n \leq 512$) :**

*Mgpu* =

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 4 | 2 | 1 | 5 | 6 | 2 | 1 |

- **"Big sizes" ($n > 512$) :**

*fft_device* =

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 0 | 0 | 2 | 1 | 0 | 0 | 5 | 6 | 0 | 0 | 2 | 1 | 0 | 0 |

# Multiplication according to the size of the polynomials

- **"Small sizes" ($n \leq 512$) :**
  We use *Mgpu* and multiply directly pairwise polynomials inside, then do a right shift. Product size is still a power of 2. So *list_Plain_Mul_and_right_shift* is used to do :
  $X \cdot \left( P^{(1)}(X) \times [(X+1)^{2^i} - 1]/X \right)$.

- **"Big sizes" ($n > 512$) :**
  We transform *Mgpu* in *fft_device* and then use *FFT* for the multiplication $P^{(1)}(X) \times [(X+1)^{2^i} - 1]/X$.

The *FFT* will be explained in the following section.

# First definition

### Definition

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

- $\omega$ is a *$n$-th root of unity* if $\omega^n = 1$.
- $\omega$ is a *primitive $n$-th root of unity* if :
  (1) $\omega^n = 1$.
  (2) $\omega$ is a unit in $R$.
  (3) $\forall t$ prime s.t. $t|n$, $\omega^{n/t} - 1$ is neither 0 nor a 0 divisor.

We now take $\omega \in R$ to be a primitive $n$-th root of unity.

# Discrete Fourier transform (DFT)

## 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 $\omega$ is called the **Discrete Fourier Transform (DFT)**.

## Proposition (consequence of the Lagrange's theorem)

*The $R$-linear map $DFT_\omega$ is an isomorphism.*

Then we can represent a polynomial $f$ by the $DFT$ representation with $\omega$ determined in our code.

# Convolution

## Definition

The **convolution** w.r.t. $n$ of $f = \sum_{0 \le i < n} f_i \, x^i$ and $g = \sum_{0 \le i < n} g_i \, x^i$ in $R[x]$ is $h = f * g = \sum_{0 \le k < n} h_k \, x^k$ s.t.

$$\forall k \in [\![0, n-1]\!],\ h_k = \sum_{i+j \equiv k \mod n} f_i \, g_j .$$

One can prove that $fg = f * g \mod (x^n - 1)$.

## Lemma (DFT product amounts to a scalar product)

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

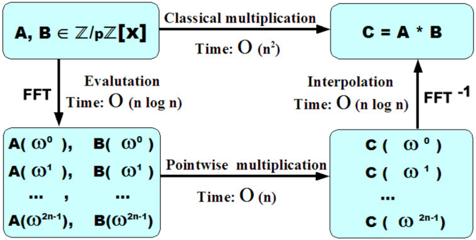
$$DFT_\omega(f * g) = DFT_\omega(f) DFT_\omega(g).$$

# FFT Maping

## Definition

The **Fast Fourier Transform (FFT)** is an efficient algorithm to compute *DFT* and its inverse.

We use the Cooley-Tukey algorithm following a *D&C* strategy.



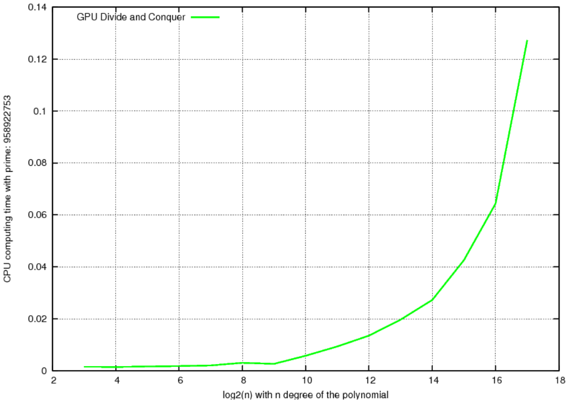

**FFT-based univariate polynomial multiplication over** $\mathbb{Z}/p\mathbb{Z}$

# Taylor shift by $1$ execution times

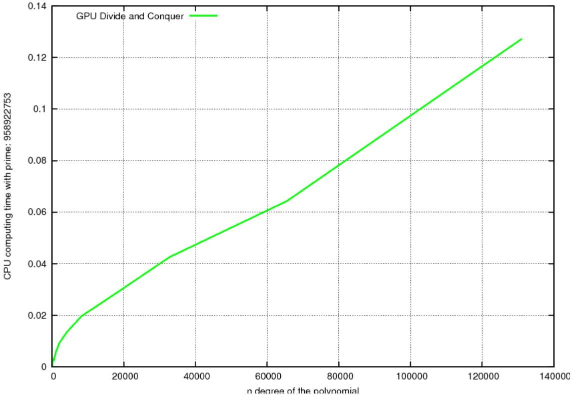
## Results are for polynomials of sizes $n = 2^e$ :

| 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 |

# Execution time of the GPU in function of $e$

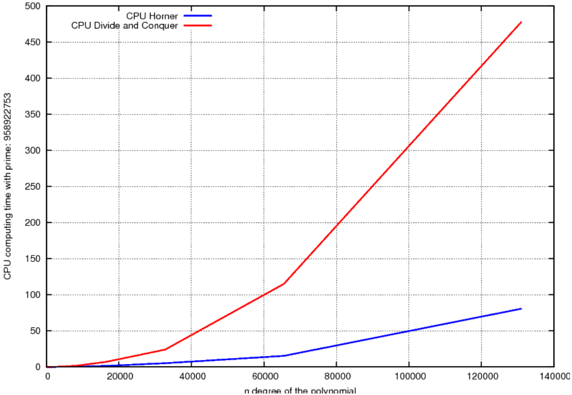


# Execution time of the GPU in function of $n$

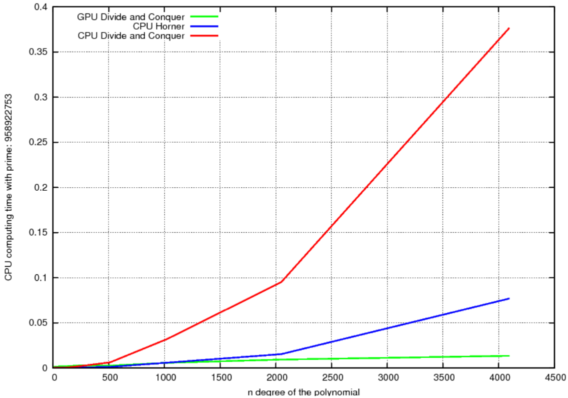



This execution time is approximatively linear.

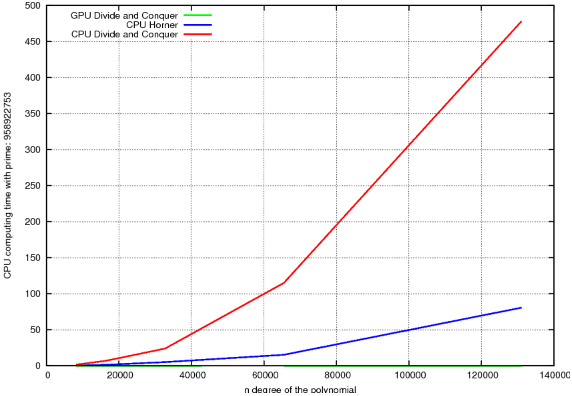
# Execution times of the CPU in function of $n$



# Execution times in function of $n$ for small degrees



# Execution times in function of $n$ for big degrees




We clearly improve performances using GPUs.

**Remark :** the behaviour of the CPU times is the same for the different sizes.

# Possible improvements & FFT primes

- We need to improve the parallel computation of the sequence $(i!)_{1 \leq i \leq n}$, and
- We can reduce the size of the array storing the elements of $(x+1)^{2^i}$ as these polynomials are symmetric.

The *Taylor shift* modulo $p$ is needed several times to get the *Taylor shift* in $\mathbb{Z}$. Two questions arises :

1. What prime numbers must we use ?
2. How many such primes numbers must we use ?

Primes numbers of the form $p = M \times 2^j + 1$ yield the best performance for *FFT*, with $p > n$ and $M < 2^j$ odd integer.

# Combination with the Chinese Remainder thm.

We need to Taylor shift by a prime number $(m_i)_{1 \leq i \leq s}$ $s$ times.
For each coefficient in $\mathbb{Z}$, we obtain a vector $\mathbf{x} = (x_1, \ldots, x_s)$.

**Objective :** using the *CRT2*, compute the image $a$ of $\mathbf{x}$ by

$$\mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_s\mathbb{Z} \cong \mathbb{Z}/m_1 \cdots m_s\mathbb{Z}.$$

**Representation :** we can represent $a$ by $\mathbf{b} = (b_1, \ldots, b_s)$ s.t.

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

Then we will use the conversion of modular numbers to their **mixed radix representation by a matrix formula** to compute $\mathbf{b}$.

# Definition (1)

Let us consider $m_1, m_2, \dots, m_s$ distinct prime numbers.

**Definition ($(m_{i,j})$ and $(n_{i,j})$)**

We define the sequences $(m_{i,j})_{1 \leq i < j \leq s}$ and $(n_{i,j})_{1 \leq i < j \leq s}$ such that :

$$\begin{cases} m_{i,j} \times m_i \equiv 1 \mod m_j \mid 0 \leq m_{i,j} < m_j \\ n_{i,j} = m_j - m_{i,j} \end{cases}$$

# Definition (2)

## Definition (matrix $A$)

We define the matrix $(A_k)_{1\leq k\leq s-1}$ as the following :

$$A_k = \left(\begin{array}{c|c} I_{k-1} & 0 \\ \hline 0 & B_k \end{array}\right) \text{ with}$$

$$B_k = \begin{pmatrix} 1 & n_{k,k+1} & n_{k,k+2} & \dots & n_{k,s} \\ 0 & m_{k,k+1} & 0 & \dots & 0 \\ \vdots & \ddots & m_{k,k+2} & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & m_{k,s} \end{pmatrix}$$

# Mixed radix representation by a matrix formula

## Theorem

$$\boldsymbol{b} = (\ldots(((\boldsymbol{x}A_1)\,A_2)\,A_3)\ldots)\;\; A_{s-1}$$

## Definition

As $A_k$ is sparse, we don't really need to multiply our results by a matrix. Thus, we will consider sequences $(L_k)_{1\leq k\leq s-1}$ and $(D_k)_{1\leq k\leq s-1}$ respectively the first row of $A_k$ and the diagonal of $A_k$ such that :

$$(L_k) = (n_{k,j} \mid k+1 \leq j \leq s),$$

$$(D_k) = (m_{k,j} \mid k+1 \leq j \leq s).$$

This gives the following algorithm (with $d = n - 1 = 2^e - 1$) :

# Algorithm for the mixed radix representation

**Input :** $\mathbf{X}[0..d][1..s]$, $s$, $(m_i)_{1\leq i\leq s}$, $(L_k)_{1\leq k\leq s-1}$, $(D_k)_{1\leq k\leq s-1}$

$\mathbf{Y} := \mathbf{X}$

for $k = 1..s-1$ do

  for $i = 0..d$ do

    for $j = k+1..s$ do

      $Y_{i,j} := \left[\left(Y_{i,j}L_{k,j} \mod m_j\right) + \left(Y_{i,j}D_{k,j} \mod m_j\right)\right] \mod m_j$

    end do

  end do

end do

**Output :** $\mathbf{Y}[0..d][1..s]$

This algorithm can be parallelized but the different loops in $k$ need less and less computations, parallelization must me done with reflection.