Putting Fürer Algorithm into Practice

Svyatoslav Covanov, supervised by Marc Moreno Mazza, ORCCA Lab

August 22, 2014

1 Introduction

The general context

The multiplication of polynomials is a primitive widely used in computer algebra. It appears to contribute to a large panel of mathematical functions. Consequently, the optimization of this primitive is critical in cryptography for instance.

Fast Fourier Transform (FFT) is commonly used in order to multiply polynomials. This is why an improvement of the Fast Fourier Transform algorithm is important. The multiplication of polynomials is connected to the multiplications of big integers. For almost 35 years, the best asymptotical complexity of the best known algorithm to multiply big integers of size n was $O(n \log n \log\log n)$. The algorithm in itself was due to Schönhage-Strassen [Knu97]. In 2007, Martin Fürer published a paper [Für07] achieving a complexity of $O(n \log n 2^{O(\log^* n)})$. We still don't know if we could reach the theoretical bound $O(n \log n \log^* n)$ or even $O(n \log n)$. A more practical problem is to be able to make the Fürer's algorithm as efficient as the current implementations of Schnhage-Strassen algorithm.

The research problem

The goal of the internship was to improve the theoritical complexity of the multiplication of polynomials used in the BPAS (Basic Polynomial Algebra Subprograms) Library [BPA], developped by Changbo Chen, Farnam Mansouri, Marc Moreno Maza, Ning Xie and Yuzhen Xie, which is fast in practice. This goal lead me to improve the one dimensional FFT which was used as a black box. In order to do so, I improved in particular the arithmetic involved in this one dimensional FFT and I helped to make it cache-friendly.

I worked on a similar subject during an internship starting from April 2013 to July 2013 in LORIA, in Nancy. I was working on the practical implementation of the Fürer algorithm in language C. I managed to improve to asymptotical complexity of the original algorithm of Fürer. This is why I thought I could bring an original contribution to the subject.

Your contribution

The main cost of the arithmetic used in the FFT algorithm is due to the multiplication over a finite field $\mathbb{Z}/p\mathbb{Z}$. I improved the arithmetical cost of this multiplication, which is using the Montgomery's trick.

The implementation of my improvement to the Fürer's algorithm required to determine how to choose some finite field $\mathbb{Z}/p\mathbb{Z}$, which representation to choose and how to

implement it efficiently. One of my contribution was to compare different approaches and allows to understand the best strategy.

Another contribution is related to the programming of the FFT in itself. I wrote self-generating code allowing to write an optimized FFT (using methods like loop-unrolling and optimizing the pipeline) in any radix.

Arguments supporting its validity

Experimentally, I observed that the optimization brought to the implementation of the one dimensional FFT accelerated the previous implementation by a factor 4. In terms of proof, the new implementation has the advantage to be more cache-friendly and reduces the number of cache-misses, which has been observed with a tool like perf.

The arithmetic related to the improvement brought to the Fürer's algorithm has been compared to the nave approach and is better in theory. The practical aspect of this new arithmetic depends essentially on the vectorization instructions and bit manipulation.

The improvement of the Montgomery's trick relied on the reduction of the number of cycles involved, which has been verified by the experiments. It seems to depend on the architecture of the machine used, and in particular on pipeline.

Summary and future work

The following would involve an implementation of the improvement of Frer's algorithm using FPGA technology, or a difference choice of finite field for the specific purpose of of multiplication of big integers.

My improvement to Montgomery's trick leads to a set of theoretical improvements which involve the use of vectorization and should be tested in the future. So far, it has been used for one machine word. It would be interesting to determine how to adapt it for two machine words.

The self-generating code allows to specialize all algorithms involved in FFT for some primes. Loop-unrolling and pipelining has been widely used but it is interesting to explain why some optimizations work and why other don't. It is also interesting to determine the optimal primes for a binary machine. We know that Proth numbers are very adapted to the use of Montgomery's trick. Can we choose an even more restrictive class of primes such that the arithmetical cost of the FFT is reduced?

Note and Acknowledgements

This report is intended to be used in order to publish an article. This is why it is written in English.

I wish to thank my supervisor Marc Moreno Maza. Working with him was very entertaining and contributed a lot to my motivation during this internship. He has an impressive skill to get the best from people working with him.

I am also grateful to Yuzhen Xie for her help in my implementation and her experience of architecture optimizations. I learned a lot about high performance and low level implementation during this internship, which will be undoubtedly useful for my Ph.D. and the rest of my carrier in research in computer science.

2 Background: FFT and polynomial multiplication

This section presents the basics needed to understand the state of the art involved in polynomial multiplication.

2.1 Radix 2 Fast Fourier Transform

In order to multiply two polynomials, the evaluation-interpretation paradigm seems to allow us to reach the best asymptotic bounds for the estimation of algebraic complexity. In other terms, to multiply polynomials A and B fast, we have to evaluate those polynomials at some points $\omega_0 \cdots \omega_K$, and retrieve the result $C = A \cdot B$ from those evaluations, which corresponds to the interpolation.

The FFT method is used to evaluate a univariate polynomial A, of degree less than N, over a ring R containing a primitive N-th root of unity ω, at the N points $1, \omega, \omega^2, \cdots \omega^{N-1}$. Several algorithms realize this method; they usually follow a divide-and-conquer strategy. This is the case for the most popular one sketched in Algorithm 1, which assumes that N is a power of 2. This algorithm is usually refered as the *radix 2 FFT* since it is a 2-way divide-and-conquer procedure.

Algorithm 1 Radix 2 Fast Fourier Transform in R

procedure $\mathrm{FFT}_{\text{radix 2}}((A_1 A_0 ... A_{N-1}), \omega)$
  if $N = 1$ **then**
    return A_0
  else
    $\tilde{A}[0] = \mathrm{FFT}_{\text{radix 2}}((A_0, A_2 ... A_{N-2}), \omega^2)$ ▷ We put apart even and odd parts
    $\tilde{A}[1] = \mathrm{FFT}_{\text{radix 2}}((A_1, A_3 ... _{N-1}), \omega^2)$
    $\Omega = (1, \omega, \cdots, \omega^{N/2-1})$
    $R[0] = \tilde{A}[0] + \Omega \cdot \tilde{A}[1]$
    $R[1] = \tilde{A}[0] + \omega^{N/2} \Omega \cdot \tilde{A}[1]$
    return $R[0]|R[1]$ ▷ "—" is the concatenation of 2 vectors
  end if
end procedure

The complexity analysis of Algorithm 1 gives a bound of the form $O(N \log_2 N)$, when counting the number of arithmetic operations in R.

Let us estimate now the cache complexity of Algorithm 1 for an ideal cache with Z words and L words per cache line. We assume that each coefficient of R fits within a machine word and that the array storing the coefficients of a polynomial consist of consecutive memory words. If $Q(N)$ denotes the number of cache misses incurred by Algorithm 1, then, neglecting misalignments, we have for some $0 < \alpha < 1$,

$$Q(N) = \begin{cases} N/L & \text{if} \quad N < \alpha Z \quad \text{(base case)} \\ 2Q(N/2) + N/L & \text{if} \quad N \geq \alpha Z \quad \text{(recurrence)} \end{cases} \tag{1}$$

Unfolding k times the recurrence relation (1) yields

$$Q(N) = 2^k \, Q(N/2^k) + kN/L.$$

Assuming $N \geq \alpha Z$ and choosing k such that $N/2^k = \alpha Z$, that is, $2^k = \frac{N}{\alpha Z}$, or

equivalently $N/L = 2^k \alpha Z/L$, we obtain

$$\begin{aligned} Q(N) &\leq 2^k \alpha Z/L + kN/L \\ &= N/L + kN/L \\ &= (k+1)N/L \\ &\leq (\log_2(\tfrac{N}{\alpha Z}) + 1)N/L. \end{aligned}$$

Therefore we have $Q(N) \in O(N/L\,(\log_2(N) - \log_2(\alpha Z)))$. This result is known to be non-optimal, following the work of Hong Jia-Wei and H.T. Kung in their landmark paper *I/O complexity: The red-blue pebble game* in the proceedings of STOC '81 [HK81].

2.2 Fürer's algorithm

Fürer's algorithm [Für09] uses a different divide-and-conquer pattern than the one of Algorithm 1. This modification is motivated by the assumption that for some integer K dividing N there exists a K-th root of unity Ω such that the multiplication of an arbitrary element of R by Ω is "computationally much cheaper" than the the multiplication of two arbitrary elements of R. For instance, if R is the field of complex numbers, we have a natural 4-th root of unity which is $\imath$. The multiplication by a power of i is very cheap on a binary machine, since it requires only shift and change of sign.

To take advantage of such primitive roots of unity, Fürer's algorithm divides the computation according to the Cooley-Tukey factorization. If the integer N writes $J \cdot K$, we split A into K parts of size J, call the FFT on those parts, multiply by some twiddle factors the resulting coefficients, and finally call he FFT on J parts of size K.

Let ω be an N-th primitive root of unity. Denote by a_k the degree k coefficient of A and by b_k the value $A(\omega^k)$, for $0 \leq k < N$. Then, for $0 \leq j' < K$ and $0 \leq j < J$, the Cooley-Tukey factorization writes:

$$\begin{aligned} b_{j'J+j} &= \sum_{k=0}^{K-1} \sum_{k'=0}^{J-1} \omega^{(j'J+j)(k'K+k)} a_{k'K+k} \\ &= \sum_{k=0}^{K-1} \omega^{Jj'k} \underbrace{\underbrace{\omega^{jk} \underbrace{\sum_{k'=0}^{J-1} \omega^{(Kjk')} a_{k'K+k}}_{\text{inner transforms}}}_{\text{coefficients of the outer transforms}}}_{\text{outer transforms}} \end{aligned}$$

Assuming that N is a power of K, the above formula leads to Algorithm 2.

Algorithm 2 Radix K Fast Fourier Transform in R

procedure $\mathrm{FFT}_{\text{radix } K}((\alpha_0\alpha_1...\alpha_{N-1}), \omega, N = J \cdot K, \Omega = \omega^{N/K})$
  for $0 \leq k < K-1$ **do** ▷ Outer transforms
    for $0 \leq k' < J-1$ **do**
      $\gamma[k][k'] = \alpha_{k'K+k}$
    end for
    $c[k] = \mathrm{FFT}_{\text{radix } K}(\gamma[k], \omega^K, J, \Omega)$
  end for
  for $0 \leq j < J-1$ **do** ▷ Inner transforms
    for $0 \leq k < K-1$ **do**
      $\delta[j][k] = c[k][j] * \omega^{jk}$ ▷ Computation of coefficients
    end for
    $d[j] = \mathrm{FFT}_{\text{radix } 2}(\delta[j], \omega^J, K, \Omega)$
    for $0 \leq j' < K-1$ **do**
      $\beta_{j'J+j} = d[j][j']$
    end for
  end for
  return $b = (\beta_0, ..., \beta_{N-1})$
end procedure

In Algorithm 2, the procedure $\mathrm{FFT}_{\text{radix2}}$ refers to Algorithm 1 or to an iterative version of it. The inner transforms computed there use a K-th root of unity Ω. Assuming that multiplying an arbitrary element of R by Ω is cheap enough, an analysis of the above algorithm yields a bound of the form $O(N \log_K N)$ arithmetic operations in R. In the sequel of this section, we shall review this analysis as well as related complexity estimates.

2.3 First arithmetic cost estimate for Algorithm 2

If we denote by $\mathsf{T}_R(N)$ the number of arithmetic operations in R performed by Algorithm 2, we clearly have:

$$\mathsf{T}_R(N) \;=\; K\,\mathsf{T}_R(N/K) \;+\; N \;+\; N/K\,\mathsf{T}_R(K). \tag{2}$$

Assume $N = K^{e+1}$, that is $e = \log_K(N) - 1$. Unfolding e times Relation (2) yields

$$\begin{aligned}
\mathsf{T}_R(N) &= K^e\mathsf{T}_R(K) \;+\; (N \;+\; N/K\,\mathsf{T}_R(K))e \\
&= (K^e \;+\; (N \;+\; N/K)e)\;\mathsf{T}_R(K) \\
&= (N/K \;+\; N(1/K \;+\; 1/K^2)e)\;\mathsf{T}_R(K) \\
&= e\;N/K\;(\tfrac{1}{e} \;+\; 1 \;+\; 1/K)\;\mathsf{T}_R(K) \\
&\leq \log_K(N)\;N/K\;\mathsf{T}_R(K).
\end{aligned} \tag{3}$$

If we take $\mathsf{T}_R(K) \in O(K \log_2(K))$, we retrieve the usual result $\mathsf{T}_R(N) \in O(N\log_2(N))$. In order to obtain a finer result, we switch to bit complexity in the next section. The intention is create opportunities to take into account a lower asymptotic upper bound for $\mathsf{T}_R(K)$.

2.4 Bit operation cost estimate for Algorithm 2

Let n be the total number of bits for writing the coefficient vector representing A. If we denote by $\mathsf{T}_{\text{bit}}(n)$ the number of bit operations performed by the above algorithm

and assuming that each element of R is encoded by n/N bits, then we have:

$$\mathsf{T}_{\text{bit}}(n) \;=\; K\,\mathsf{T}_{\text{bit}}(J\,n/N) \;+\; N\,\mathsf{M}_R(1) \;+\; J\,\mathsf{T}_{\text{bit}}(K\,n/N), \tag{4}$$

where, for any non-negative integer d, multiplying two polynomials of $R[x]$ of degree less than d amounts at most to $\mathsf{M}_R(d)$ bit operations. For convenience and based on our assumption that our each element of R is encoded by n/N bits, we define

$$\mathsf{M}_{\text{bit}}(d\,n/N) \;:=\; \mathsf{M}_R(d).$$

Using Bluestein's FFT algorithm, which re-expresses the DFT as a convolution, thus as polynomial multiplication, we can assume w.l.o.g.

$$\mathsf{T}_{\text{bit}}(K\,n/N) \;\in\; O(\mathsf{M}_{\text{bit}}(K\,n/N)).$$

Hence we deduce:

$$\mathsf{T}_{\text{bit}}(n) \;=\; K\,\mathsf{T}_{\text{bit}}(J\,n/N) \;+\; N\,\mathsf{M}_{\text{bit}}(n/N) \;+\; J\;O(\mathsf{M}_{\text{bit}}(K\,n/N)). \tag{5}$$

That is:

$$\mathsf{T}_{\text{bit}}(n) \;=\; K\,\mathsf{T}_{\text{bit}}(n/K) \;+\; N\,\mathsf{M}_{\text{bit}}(n/N) \;+\; N/K\;O(\mathsf{M}_{\text{bit}}(K\,n/N)). \tag{6}$$

Writing $b := n/N$, the above equality becomes:

$$\mathsf{T}_{\text{bit}}(n) \;=\; K\,\mathsf{T}_{\text{bit}}(N/K\,b) \;+\; N\,\mathsf{M}_{\text{bit}}(b) \;+\; N/K\;O(\mathsf{M}_{\text{bit}}(K\,b)). \tag{7}$$

Observing that $\mathsf{M}_{\text{bit}}(d) \in O(\mathsf{T}_{\text{bit}}(d))$ (by means of three FFTs and one point-wise multiplication) we have

$$\mathsf{T}_{\text{bit}}(n) \;=\; K\,\mathsf{T}_{\text{bit}}(N/K\,b) \;+\; N\,\mathsf{M}_{\text{bit}}(b) \;+\; N/K\;O(\mathsf{T}_{\text{bit}}(K\,b)). \tag{8}$$

Unfolding $\log_K(N)$ times leads to

$$\mathsf{T}_{\text{bit}}(n) \;=\; (N\,\mathsf{T}_{\text{bit}}(b) \;+\; N/K\;O(\mathsf{T}_{\text{bit}}(K\,b)))\,\log_K(N) \tag{9}$$

Assuming $b = K = \log_2(n)$, this latter relation leads to $\mathsf{T}_{\text{bit}}(n) \in O(n\;\log n\;2^{O(\log^* n)})$, where $\log^* n$ represents the iterated logarithm operation, defined as follows

$$\log^* n := \begin{cases} 0 & \text{if } n \le 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases} \tag{10}$$

This finally proves that, under the assumption $b = K = \log_2(n)$, Algorithm 2 can be cast into the Fürer complexity class.

2.5 Cache complexity estimate for Algorithm 2

Let us estimate now the cache complexity of Algorithm 2 for an ideal cache with Z words and L words per cache line. As before, we assume that each coefficient of R fits within a machine word. If $Q(N)$ denotes the number of cache misses incurred by Algorithm 2, then, neglecting misalignments, we have for some $0 < \alpha < 1$,

$$Q(N) = \begin{cases} N/L & \text{if} \quad N < \alpha Z \quad \text{(base case)} \\ KQ(N/K) + N/L + N/KQ(K) & \text{if} \quad N \ge \alpha Z \quad \text{(recurrence)} \end{cases} \tag{11}$$

We shall assume that $K < \alpha Z$ holds. Hence, we have $Q(K) \leq K/L$. Thus, for $N \geq \alpha Z$, Relation (11) leads to:

$$\begin{array}{rcl} Q(N) & = & KQ(N/K) \ + \ 2N/L \\ & \leq & K^e \, Q(N/K^e) \ + \ 2\,e\,N/L \\ & \leq & K^e \, \frac{\alpha Z}{L} \ + \ 2\,e\,N/L \\ & = & N/L \, (1 \ + \ 2\,e) \\ & \leq & N/L \ 3\,e. \end{array} \tag{12}$$

where e is chosen such that $N/K^e = \alpha Z$, that is, $K^e = \frac{N}{\alpha Z}$ or equivalently $N/L = K^e \alpha Z/L$. Therefore, we have $Q(N) \in O(N/L \, (\log_K(N) - \log_K(\alpha Z)))$. In particular, for $K \simeq \alpha Z$ and since we have

$$Q(N) \in O(N/L \log_{\alpha Z}(N). \tag{13}$$

According to the paper *I/O complexity: The red-blue pebble game*, this bound would be optimal for $\alpha = 1$. In practice α is likely to $1/8$ or $1/16$ and Z is likely to be between 1024 and 8192 for an L1 cache. Hence, the above estimate of $Q(N)$ suggests to choose K between 64 and 1024. In fact, in practice, we have experimented K between 8 and 16. The reason is that optimizing register usage (minimizing register spilling) is also another factor of performance and, to some sense, registers can be seen another level cache. As an example, the X86-64 processors that we have been using have 16 GPRs/data+address registers and 16/32 FP registers.

2.6 Let us speculate a bit

Let us return to Relation (8) and assume that, in Algorithm 2, we could optimize the function call

$$\mathrm{FFT}_{\text{radix 2}}(\delta[j], \omega^J, K, \Omega)$$

so that multiplication by a power of Ω could be hard-coded. More precisely, let us assume

$$\mathsf{T}_{\text{bit}}(K\,b) \ \leq \ K \log_2(K) \ C_R, \tag{14}$$

where C_R is an upper bound for the number of bit operations for multiplying an arbitrary element of R by a constant of R. Then, we can choose C_R such that Relation (8) implies

$$\mathsf{T}_{\text{bit}}(n) \ \leq \ K \, \mathsf{T}_{\text{bit}}(N/K\,b) \ + \ N \, \mathsf{M}_{\text{bit}}(b) \ + \ N \log_2(K) \ C_R, \tag{15}$$

Recall that K is meant to be "small" and that $\mathsf{M}_{\text{bit}}(b)$ is the number of bit operations for multiplying two arbitrary elements of R. Therefore, it is conceivable that, for b large enough, we have

$$\mathsf{M}_{\text{bit}}(b) \ \geq \ \log_2(K) \ C_R. \tag{16}$$

Then, Relation (15) leads to

$$\mathsf{T}_{\text{bit}}(n) \ \in \ O(N \ \log_K(N)). \tag{17}$$

To some sense, the above remark is the key idea of Fürer's algorithm. However, this remark also suggests that, to make Fürer's algorithm practically better than the classical radix 2 FFT algorithm, one should be able to make b large enough, that is, working with prime fields where the characteristic is of several machine words size. We have verified experimentally that, indeed, machine word size prime fields satisfying like Relations (14) do not lead to an implementation of Fürer's algorithm which can compete with the classical radix 2 FFT algorithm using the same prime field.

3 Main Results

3.1 Improved modular arithmetic

Let us explain how the improvement to Fürer's algorithm works. The idea is to use a ring R of the form $\mathbb{Z}/p\mathbb{Z}$. The prime should be of the form $x^K + 1$ where x is even. If we decompose the elements of R in the x radix, we have a natural cheap K-th root of unity in R which is x : the multiplication by a power of x of an element decomposed in radix x is equivalent to some shifts, some change of sign, and some additions.

To include it inside the BPAS library, primes like $(255 \cdot 256)^4 + 1$ have been considered. This number can be rewritten like $(2^{16} - 2^8)^4 + 1$. Since the radix is sparse, we can complete the multiplication of two elements of R represented in radix r and avoid the cost of the divisions involved due to the fact that the result of the multiplication has to be well represented in radix r. The details of this multiplication are given in **section 5**.

The sparsity of the radix allows to compute the operations modulo and division faster. The idea is to reduce the cost of working in a new radix, such that the main cost of the multiplication modulo $p = r^K + 1$ will be concentrated in the first multiplication.

3.2 FFT code generator for Fürer class complexity algorithm

Relying on the last paper [HvdHL14] related to the multiplication of big integers, an algorithm entering in the Frer's class of complexity has been implemented. The idea is to use, like suggested in the original Frer's paper, an other radix P such that $P = O(\log_2 N)$ where N is the size of the input and P a power of two.

The Bluestein's chirp transform explains why we get a complexity as in Frer's algorithm. This transform allows, given a vector of size N over $\mathbb{Z}/p\mathbb{Z}$ to exchange the computation of its FFT using Cooley-Tukey for instance with the computation of a multiplication of two polynomials of degree N over $\mathbb{Z}/p\mathbb{Z}$.

We use the idea of Bluestein to justify that the implementation of the algorithm is optimal with a theoretical point of view in terms of algebraic complexity but also in terms of cache complexity. The implementation uses a python code in order to generate a code adapted to a given architecture, depending on some parameter like the size of the Cache L1, or the number of registers.

4 Computing modulo Proth prime numbers: Montgomery's trick

4.1 The original version

The Montgomery trick [Mon85] is an improved way of performing the modular multiplication $a \cdot b \mod p$, given a, b, p in $\mathbb{Z}$, where $p > 2$ is a prime. The nave way proceeds by computing the product $c = a \cdot b$ in $\mathbb{Z}$, followed by computing the remainder $c \mod p$. On common architectures, this remainder operation is usually very slow, compared to that of multiplication. For instance, on some `Intel` architectures, the multiplication will require 3-10 cycles, whereas the division will require 70-80 cycles [Int].

Morally, the Montgomery's trick allows to compute $c \mod p$ using 3 multiplications. Let $R = 2^{\lceil \log_2 |p| \rceil}$. The Montgomery Multiplication is the following map:

$$* \ : \ \begin{cases} \mathbb{F}_p \times \mathbb{F}_p & \longrightarrow & \mathbb{F}_p \times \mathbb{F}_p \\ (u, v) & \mapsto & (u \cdot v)/R \mod p \end{cases}$$

We compute this Montgomery multiplication in this way:

1. We precompute p' such that $R \cdot R^{-1} - p \cdot p' = 1$ and $p' < R$
2. We compute $x = u \cdot v$
3. We compute $y = x \mod R$
4. We compute $z = y \cdot p' \mod R$
5. The result is $(x + z \cdot p)/R$

Let us show why this gives us the expected result. There exist $k, k' \in \mathbb{Z}$ such that we have:

$$x + z \cdot p = y + k \cdot R + (y \cdot p' - k' \cdot R) \cdot p = k \cdot R + y \cdot (1 + p \cdot p') - k'p \cdot R$$

Then $x + z \cdot p$ is of the form $q \cdot R$, which means that $x/R = q \mod p$.

Since $0 \leq x < p^2$ and $z < R$, we have $x + z \cdot p < 2p \cdot R$, which means that the result is in the interval $[0; 2p[$.

4.2 Xin Li's version

Let us pick a prime p such that we have

$$p - 1 = c2^n \text{ and } l \leq 2n,$$

where $\ell := \lceil \log_2 p \rceil \leq b$ and b is the number of bits of a machine word.

Let $R = 2^\ell$ and $0 \leq x \leq (p-1)^2$. We compute $\frac{x}{R} \mod p$ with the following instructions:

1. We compute $x = u \cdot v$
2. We compute y and k such that $y = x + k \cdot R$ and $0 \leq y < R$
3. We compute z and k' such that $z = y \cdot c2^n + k' \cdot R$ and $0 \leq z < R$
4. The result is $-k + k' + (z \cdot c2^n)/R$

Since $c2^n = -1 \mod p$, the following equations hold modulo p:

$$\frac{x}{R} = -k + \frac{y}{R} = -k + k' - \frac{z}{R} = -k + k' + \frac{z \cdot c2^n}{R}$$

The advantage of this version is that instead of doing 3 long multiplications, we are computing 2 long multiplications (i.e. in double precision) and 1 short multiplication (i.e. in single precision), which on common architectures is very fast. The short multiplication is the computation of $z \cdot c2^n$. Indeed, we have $z = y \cdot c2^n \mod R$, which means that the n first bits of z are all zeros. The non zeros bits are stored on a chunk of size $\ell - n$, which is the size of c. Since $l \leq 2n$, $c < 2^n$ and $c \cdot z/2^n < 2^\ell < 2^b$.

4.3 New improvements

The main issue with the last version of Xin Li's multiplication is the final interval. Indeed, the result computed is in the interval $[-(p-1); 2(p-1)]$, instead of $[0, 2p-1]$ like in the original version. Experimentally, the normalization of the result in the interval $[0, p[$ appeared recently to be very slow. This suggested us to revisit the original vversiob of Montgomery multiplication.

Reusing the notations of Section 4.1, we observe that in the modular product $z = y \cdot p' \mod R$, only the "lower part" of $y \cdot p$, hence this latter product can be computed using single precision, since R fits on a machine word.

The assembly routine implemented is given in **Appendix D**.

The first multiplication is the multiplication $u \cdot v$. The second "imul" instruction is the multiplication by INV_PRIME, which contains the value of p' described before.

This multiplication is followed by the third long multiplication "mulq". The result is contained in the **rdx** register, in the interval $[0; 2p[$. We want that the correct residue in $[0; p[$. This is why we end the routine by the rearrangement in the correct interval. To proceed, we subtract p to **rdx** and store the result in **rax**. The operation "sar" fills the register with the sign bit. In other terms, if **rax** is negative, then the "sar" instruction returns the machine word all ones. Otherwise, it returns zero. The next steps consists in computing the logical "and" with the value of p and returning the correct residue.

We can, moreover, choose primes such that $\lceil \log_2 c \rceil + \lceil \log_2 R \rceil < 64$. This way, even the last multiplication will be a short multiplication. Indeed, we will have $f < R$ and $p = c2^n + 1$, which means that if we compute $f \cdot p$, we can decompose this multiplication like this : $f \cdot c2^n + f$, and the multiplication of f by c fits on a machine word. In conclusion, we will have to compute a short multiplication, a shift by n and an addition with f.

There is an opportunity for vectorization on the second multiplication by p'. We observe that p' will be of the form $c'2^n - 1$. Indeed, writing $p' = c'2^{n'} - 1$ and using the fact that $pp' - 1$ is a multiple of $R = 2^\ell$ with $\ell \geq n$. leads to $n' \geq n$. Then, , let us decompose $p' \cdot y \mod R$ in this way :

$$p' \cdot y \equiv c'2^n y - y \mod R$$

We just need to compute $c'2^n y \mod R = c'y \mod R/2^n$. This is why getting the $\ell - n = \lceil \log_2 c \rceil$ first bits of $c' \cdot y$ is enough to be able to recompose $y \cdot p' \mod R$.

5 Computing modulo sparse radix prime numbers

The problem of this section is to determine how to compute in a ring $R := \mathbb{Z}/p\mathbb{Z}$ with the following hypotheses:

1. p can be decomposed like this: $p = r^K + 1$,
2. p fits on a machine word,
3. r is sparse (to be specified shortly after).

Let M represent the size of a machine word and assume that K divides M.

r is sparse means that, ideally, r is of the form $2^{M/K} \pm 2^u$ where u is the smallest we can find. The idea is to representing every x of R as a polynomial in $\mathbb{Z}[r]$. The coefficients can be stored on K slots of $\log_2 r$ bits on a machine word.

We will show how to compute an addition modulo a sparse radix prime number. Let x and y be two elements of $\mathbb{Z}/p\mathbb{Z}$. We represent them as $\sum_i x_i r^i$ and $\sum_i y_i r^i$.

Ideally, we want to compute an addition using only the smallest number of operations we can achieve on a binary machine, giving some instructions such as arithmetic operations (add, sub) and logical operations (and, or, xor, shift). Let us take the number $r = 2^{16} - 2^8$ and the prime $p = r^4 + 1 = 18160198153666560001$ as an example.

On 64-bits, the coefficients x_i and y_i will fit on 16-bits. Each of those coefficients fit are less than r. The nave algorithm to add x and y would be to compute simply the addition, and handling the carries for each coefficient. But it would require $O(K)$ operations.

Let $a = \sum_i (2^{16} - r) \cdot 2^{16i}$. Consider $x' = x + a$. If we observe what happens on each slot of 16 bits, there is no overlap with the next slot after the addition, since we have

$$x_i + 2^{16} - r < 2^{16}.$$

Now, let us consider $z' = x' + y$. We have

$$x_i + y_i \geq r \quad \iff \quad x'_i + y_i >= 2^{16}.$$

We conclude that we have an overlap with the slot $i+1$ if and only if $x_i + y_i \geq r$. Moreover, if $x_i + y_i \geq r$, then the low 16-bits of $x'_i + y_i$ represent exactly $x_i + y_i \mod r$.

The next step is to detect the slots where a carry has been propagated. In order to do it, we just have to look at the first bit of each slot of z' and compare with $\mathbf{xor}(x',y)$. Indeed, $\mathbf{xor}(x',y)$ gives a "carry-less addition". Then, we just need to use some mask to retrieve the slots for which the first bit of the next one has not changed, do a shift and a subtraction.

Algorithm 3 summarizes this addition process:

Algorithm 3 Addition in R

1: **procedure** ADD($x = (x_0 x_1 ... x_{K-1})$, $y = (y_0 y_1 ... y_{K-1})$)
2: $x' = x + a$ ▷ $a = \sum_i (2^{16} - r) \cdot 2^{16i}$
3: $u = xor(x', y)$
4: $v = x' + y$
5: $carries = xor(u, v)$
6: $carries = and(not(carries), Ones)$ ▷ $Ones = \sum_i 2^{16}$
7: $carries = rshift(carries, 16)$
8: **return** $v - carries$
9: **end procedure**

In the algorithm below, we skipped an important point, which is related to the eventual carry due to the $(K-1)$-th slot. Indeed, if $x_{K-1} + y_{K-1} \geq r$, then we have $x + y = r^K + s$, with $0 \leq s < r^K$. Since $r^K = -1 \mod p$, we have to propagate the carry (namely -1) to the first coefficient.

But we have to be careful. Indeed, in our example, $2^{16} - r = 2^8$. If the first coefficient is equal to 2^8 or 0, then propagating the last carry directly could give an incorrect result. If the first coefficient is equal to 2^8, it means $y_0 = x_0 = 0$, which implies that there is no carry propagated on the next slot. According to the previous algorithm, we will have to remove 2^8, but since we propagated the "last carry", we would have $2^8 - 1 - 2^8$, and this implies that the representation will not be correct. We have a similar phenomenon if $u_0 = 0$.

This is why we have to handle those particular cases by comparing v_0 to 2^8 and 0. Since the probability that v_0 is equal to one those is very low, the impact of branch misprediction is very low.

Let us show how to compute the multiplication of some x by a power of r. Indeed, this operation has to be as fast as possible, since the complexity analysis of Frer's improved algorithm relies on this property. We will take as an example the multiplication by r. The multiplication by other powers of r can be easily deduced from this example.

The multiplication by r gives us :

$$x \cdot r = \sum_i x_i r^{i+1} = \sum_{i=0}^{K-2} x_i r^{i+1} + x_{K-1} r^K = \sum_{i=0}^{K-2} x_i r^{i+1} - x_{K-1}$$

In other terms, the multiplication by r can be done with a "shift" and a subtraction in r-radix. But since the subtraction is very specific, we don't have to compute a full

subtraction (like the full addition we saw below). Let j be the smallest i in $\sum_{i=0}^{K-2} x_i r^{i+1}$ such that $x_i > 0$. Observe that we clearly have in $\mathbb{Z}$:

$$0 = r + (r-1) \cdot r + \cdots + (r-1) \cdot r^{j-1} - r^j$$

Define $z = \sum_{i=0}^{j} z_i r^j$ with $z_0 := r, z_1 := \cdots := z_{j-1} := r-1$ and $z_j = -1$. Now we compute $y := z + \sum_{i=0}^{K-2} x_i r^{i+1}$ followed by the usual subtraction for $y - x_{K-1}$.

Given x and y, computing the multiplication of those elements represented in radix r requires to pad zeros between coefficients, in order to use Kronecker Substitution. For instance, if $x = \sum_i x_i \cdot r^i$, each x_i will be followed by $\lceil \log_2 r \rceil + \lceil \log_2 K \rceil$ zeros.

Once we have computed $z = x \cdot y$, we have to compute the correct representation of z in radix r. We can using the fact that r is sparse in order to compute several coefficients at once.

Suppose $r = 2^{16} - 2^8$. If we consider the first coefficient z_0 of z, then we can cut z_0 into two parts: $z_0 = a + 2^{16}b$, with $a < 2^16$. We want to represent z_0 like this: $z_0 = a' + r \cdot b'$ with $a' < r$. Since $2^{16} = r + 2^8$, we have $z_0 = (a + 2^8 b) + b \cdot r$. Suppose we can represent $(a + 2^8 b)$ like $c + r \cdot d$, with $c < r$, then $z_0 = c + r \cdot (d + b)$. This means that we can apply recursively the previous routine on the successive remainders we get.

Here is the pseudo-code we apply :

```
while input >= r
    high = input & (2^32-2^16)
    low = input & (2^16-1)
    high >>=8
    low = low+high
    input = low
```

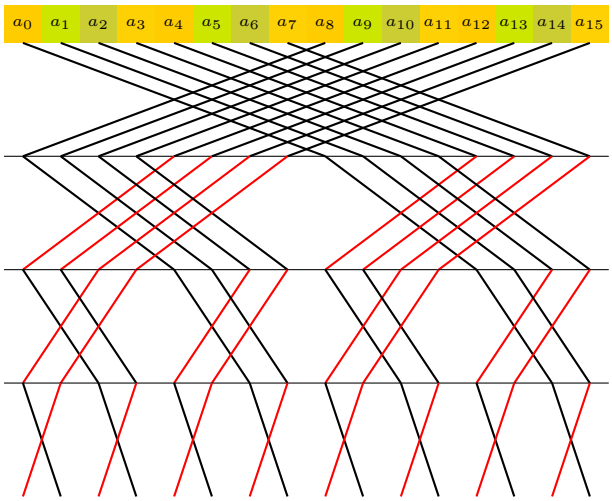
The first line in the loop body allows to retrieve the high part of the input, and the second retrieves the low part. We apply this routine until *low* fits on 16 bits. We then might have to subtract r to the final remainder.

6 Generating 1D FFT code

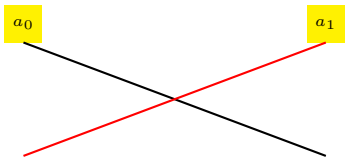
6.1 Two strategies

Two strategies have been explored in order to implement the Fast Fourier Transform. The first strategy consisted to implement the Radix-two Cooley-Tukey Number Theoretic transform.

The radix-two Cooley-Tukey transform relies on the factorization of the size of the input vector. We will admit that this size N is a power of two 2^k. Then the Cooley-Tukey algorithm appears to be the algorithm describing the usual FFT, which consist in separating odd and even part of the input vector, and apply recursively the algorithm on the two parts. Since we divide by 2 the size of the input at each level, we will have k level of recursions.

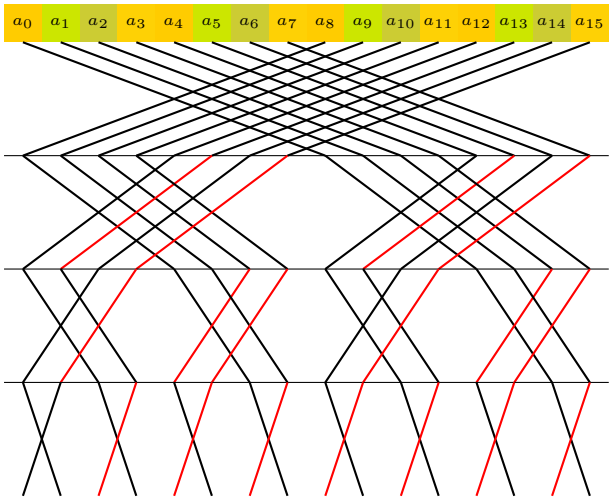


On Figure above, we have the butterfly graph for a 16-point FFT, using the radix-2 Cooley-Tukey algorithm. The input is the array $[a_0\ a_1 \cdots\ a_{15}]$. A butterfly graph describes the operations performed on each level of recursion. The basic operation is the following :



The red line illustrates the fact that we multiply a_1 by a root of unity. In other terms, according to the whole Butterfly graph for 16 points, we have 24 multiplications for the whole algorithm. Indeed, the first level does not contain any multiplication because those Butterflies correspond to multipliation by 1, which is clearly cheap.

Actually, we can further refine, since some of the multiplications on lower levels of FFT are multiplications by 1 as well:



In conclusion, we have $4+6+7=17$ multiplications in the classical algorithm, and 4 levels of recursion.

Let us consider $2^j = O(k)$ the smallest power of two greater than k. We factorize the size N like $N = 2^j \cdot 2^{k-j}$. We cut the input vector of the FFT in 2^j pieces of

size 2^{k-j} on which we apply recursively the algorithm. Then we multiply by twiddle factors, and we compute 2^{k-j} transforms of size 2^j. Hence, the transforms of size 2^j should be regarded as the *base case*.

Here is the formula associated to this decomposition, if the input vector is a and the output is b :

$$\begin{aligned} b_{i'2^{k-j}+i} &= \sum_{l=0}^{2^j-1} \sum_{l'=0}^{2^{k-j}-1} \omega^{(i'2^{k-j}+i)(l'2^j+l)} a_{l'2^j+l} \\ &= \underbrace{\sum_{l=0}^{2^j-1} \omega^{2^{l-j}j'l} \quad \underbrace{\omega^{il} \underbrace{\sum_{l'=0}^{2^{k-j}-1} \omega^{(2^j il')} a_{l'2^j+l}}_{\text{inner transforms}}}_{\text{coefficients of the outer transforms}}}_{\text{outer transforms}} \end{aligned}$$

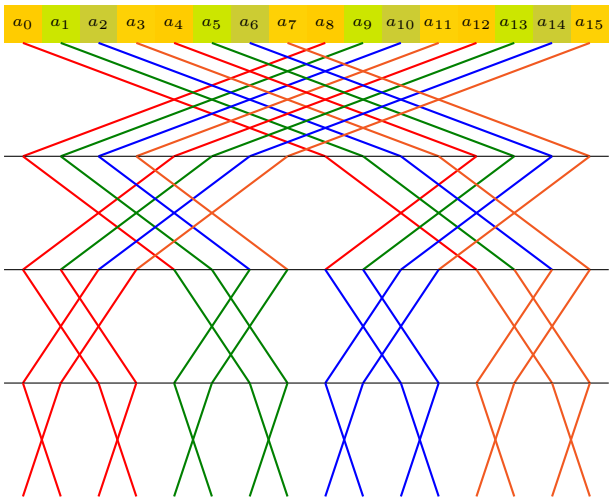
In the above a and b have size N; j and k are as above: $0 \leq i < 2^{k-j}$ and $0 \leq i' < 2^j$.

We can rewrite the algorithm seen in the background this way :

Second Strategy FFT

procedure FFT($(\alpha_0\alpha_1...\alpha_{N-1})$, ω, $N = 2^{k-j} \cdot 2^j$, $\Omega = \omega^{2^{k-j}}$)
  for $0 \leq l < 2^j - 1$ **do** ▷ Inner transforms
    for $0 \leq l' < 2^{k-j} - 1$ **do**
      $\gamma[l][l'] = \alpha_{l'2^j+l}$
    end for
    $c[l] = FFT(\gamma[l], \omega^K, 2^{k-j}, \Omega)$
  end for
  for $0 \leq i < 2^{k-j} - 1$ **do** ▷ Outer transforms
    for $0 \leq l < 2^j - 1$ **do**
      $\delta[i][l] = c[l][i] * \omega^{il}$ ▷ Computation of coefficients
    end for
    $d[i] = FFT(\delta[i], \omega^{2^{k-j}}, 2^j, \Omega)$
    for $0 \leq i' < 2^j - 1$ **do**
      $\beta_{i'2^{k-j}+i} = d[i][i']$
    end for
  end for
  return $b = (\beta_0, ..., \beta_{N-1})$
end procedure

In the last strategy, we will have $\log_k N$ levels of recursion. At each level, we multiply by some twiddle factors and we compute small-size FFT. Somehow, $\log_2 N - \log_k N = \left(1 - \frac{1}{\log_2 k}\right) \log_2 N$ level of recursions in the first strategy have been exchanged against small-size FFT, reusing the same roots of unity, which is better on architectures preferring repetitive tasks.



In the picture below, we represent on 2 level of recursions the small 4-points butterflies by the same color. In other terms, We can see 2 level of recursions, on which we compute each time 4 small butterflies. Each butterfly is a 4-point FFT, so for each of them, we have 1 multiplication according to the previous analysis. The product by twiddle factors happens in the middle of the previous picture. 1/4-th of those twiddle factors are multiplication by one. Among the other twiddle factors, 3 of them will be multiplication by one. This means we have $4 \cdot 3 - 3 = 9$ multiplications. We have $4 + 4$ 4-points butterflies, which means we have 17 multiplications for the whole algorithm.

6.2 Complexity Analysis

Let us perform the complexity analysis of the previous algorithms. We will compute the number of multiplications for each version.

For the first version, let us say we have an input vector of size $N = 2^k$. We will have then k level of recursions. On the first level of recursion, there will not be any multiplication. On the second level of recursion, we will have $N/2$ multiplications, among which half will be multiplications by one : we will have $N/4$ real multiplications. On the third level of recursion, 3/4-th of the $N/2$ multiplications will be real ones.

Here is the formula we can deduce, if $M(N)$ is the number of multiplications :

$$\begin{aligned} M(N) &= \frac{N}{2}\left(\frac{1}{2} + \frac{3}{4} + \frac{7}{8} + \cdots + \frac{N-1}{N}\right) \\ &= \frac{N}{2}\left(k - 1 - \frac{1}{2} - \frac{1}{4} - \frac{1}{8} - \cdots - \frac{1}{N}\right) \\ &= \frac{N}{2}\left(k - 1 - 1 + \frac{1}{2^{k-1}}\right) \\ &= \frac{N}{2}\left(k - 2 + \frac{2}{N}\right) \\ &= \frac{N}{2}k + 1 - N \end{aligned}$$

Now let us analyse the number of multiplications for the algorithm factorizing N like $2^{k-j} \cdot 2^j$. We have $\frac{k}{j}$ level of recursions. On each level of recursion, we have 2^{k-j} butterflies of size 2^j. According to the previous analysis, those butterflies require $j2^{j-1} + 1 - 2^j$ multiplications. Between each level of recursions, we have multiplications by twiddle factors. At each level of multiplication by twiddle factors, we will have one chunk of size $N/2^j$ of multiplications by one.

For the other chunks, we have to remove the first multiplication which is a multiplication by one. After the first level of the FFT butterfly, we will have $\frac{N}{2^{2j}}$ sets in which we have 2^j chunks for the twiddle factors. The first chunk will contain only multiplications by one as we have seen it before. The other $2^j - 1$ chunks will contain

one multiplication by one (the first). After the second level in the Butterfly, we will have $\frac{N}{2^{3j}}$ sets respecting the same properties we just have seen. We can keep going and conclude that we have to remove $\frac{N}{2^{2j}}\left(2^j-1\right)\right) + \frac{N}{2^{3j}}\left(2^j-1\right)\right) + \cdots + 1\cdot\left(2^j-1\right)$ multiplications by one. In other words, the whole cost is :

$$\begin{aligned} M(N) &= 2^{k-j}\frac{k}{j}\left(j2^{j-1}+1-2^j\right) + \left(\frac{k}{j}-1\right)\cdot\left(1-\frac{1}{2^j}\right)N \\ &\quad -\frac{N}{2^{2j}}\left(2^j-1\right)\right) - \frac{N}{2^{3j}}\left(2^j-1\right)\right) - \cdots - 1\cdot\left(2^j-1\right) \\ &= \frac{Nk}{2} + \frac{N}{2^j} - N - \frac{N}{2^j} + 1 \\ &= \frac{Nk}{2} + 1 - N \end{aligned}$$

If the last algorithm is implemented in a naive way, it should be as fast as the previous algorithm. But since the multiplications in the small butterflies are supposed to be cheaper, we can assume that the leading term $\frac{Nk}{2}$ grows more slowly than in the first analysis.

We can describe the binary complexity for each algorithm. We will assume we work over $\mathbb{Z}/p\mathbb{Z}$. b will describe the number of bits of p. We will show in this section that we fall in the Frer class for the second strategy, relying on the Bluestein's chirp transform.

Let us design by $M(k)$ the bit complexity for multiplying two integers of k-bits. Then, for the first strategy, we can write the (multiplicative) bit complexity $C(N,b)$ like this:

$$C(N,b) = \left(\frac{N}{2}k + 1 - N\right)\cdot M(b)$$

Indeed, we have computed the number of multiplications in the first strategy and those multiplications are made over $\mathbb{Z}/p\mathbb{Z}$.

In the second strategy, we can use the Bluestein's chirp transform to analyse the complexity. The Bluestein's chirp transform converts the computation of a P-points FFT in a multiplication of polynomials of degree P in variable x modulo x^P-1. This way, the binary complexity of a P-points FFT over $\mathbb{Z}/p\mathbb{Z}$ is at most $M(P\cdot b)$. We will apply this idea to the small butterflies.

The complexity can be rewritten like this :

$$\begin{aligned} C(N,b) &= M(2^j\cdot b)2^{k-j} + \left(\left(\frac{k}{j}-1\right)\cdot\left(1-\frac{1}{2^j}\right)N - \frac{N}{2^j} + 1\right)\cdot M(b) \\ &= 2^{k-j}2^j b(j+\log b)2^{O(\log^*(2^j b))} + \left(\left(\frac{k}{j}-1-\frac{k}{j2^j}\right)\cdot N + 1\right) b\log b 2^{O(\log^*(b))} \\ &= Nb(j+\log b)2^{O(\log^*(2^j b))} + \left(\left(\frac{k}{j}-1-\frac{k}{j2^j}\right)\cdot N + 1\right) b\log b 2^{O(\log^*(b))} \end{aligned}$$

We are free to choose j and p; and we do so. If $2^j = k$ and $b = O(k)$, the leading term in the first case will be $\frac{Nk^2}{2}\log k 2^{O(\log^* k)}$ and in the second case, it will be $O(\frac{Nk^2}{2}2^{O(\log^* k)})$, which is better. We introduced here the complexity of Frer's algorithm to estimate the value of $M(k)$. To conclude, since we have an input of $n = Nb = O(Nk)$ bits, we end with a complexity of the form $O(n\log n 2^{O(\log^* k)})$.

Let us explain now the principle of Bluestein's chirp transform.

The problem is to compute the FFT of P points x_k. We assume we work in $R = \mathbb{Z}/p\mathbb{Z}$ containing a $2P$-th root of unity ω.

We want to compute $\hat{x_k}$ where :

$$\hat{x_k} = \sum_{i=0}^{P-1} x_i \cdot \omega^{2ik}$$

We observe that we can express this sum as a convolution product :

$$\hat{x_k} = \omega^{-k^2} \sum_{i=0}^{P-1} \left(x_i \omega^{-i^2} \right) \omega^{(k-i)^2}$$

If we consider the sequences $a_i = x_i \omega^{-i^2}$ and $b_i = \omega^{i^2}$, the $\hat{x_k}$ are obtained through the following convolution product :

$$\hat{x_k} = \omega^{-k^2} \sum_{i=0}^{P-1} a_i b_{k-i}$$

In conclusion, we can get the vector $X = (x_0, x_1, \cdots, x_{P-1})$ doing the multiplication of the polynomials $\sum_{i=0}^{P-1} a_i x^i$ and $\sum_{i=0}^{P-1} b_i x^i$ modulo $x^P - 1$ and by multiplying the coefficients $\tilde{x_i}$ obtained by ω^{-i^2}.

6.3 Implementation

Both strategy have been implemented using Python code generating C code. The automatic generation of code is meant to be adaptive to different architectures, for which for instance the cache size varies, or the number of registers.

Indeed, in order to make the Fast Fourier Transform cache-friendly, the approach consisting in using a recursive algorithm, followed by the iterative one once the size of the input vector fits in a cache has been explored. This is why depending on the size of the cache for a given architecture, the implementation should adapt itself.

Let us describe the implementation for the first strategy. The recursive function calls itself twice, on the even and the odd parts of the input vector. Once this is done, we compute the small 2-points FFT, by unrolling a **for-loop**.

Algorithm 4 Multiplication by twiddle factors

for $0 \leq i \leq n/2$,$i = i + 8$ **do**
$A[i + n/2] = A[i + n/2] * \omega^{i}$
$A[i + 1 + n/2] = A[i + 1 + n/2] * \omega^{i+1}$
$A[i + 3 + n/2] = A[i + 2 + n/2] * \omega^{i+2}$
$A[i + 3 + n/2] = A[i + 3 + n/2] * \omega^{i+3}$
$A[i + 4 + n/2] = A[i + 4 + n/2] * \omega^{i+4}$
$A[i + 5 + n/2] = A[i + 5 + n/2] * \omega^{i+5}$
$A[i + 6 + n/2] = A[i + 6 + n/2] * \omega^{i+6}$
$A[i + 7 + n/2] = A[i + 7 + n/2] * \omega^{i+7}$
$u = A[i] + A[i + n/2]$
$v = A[i] - A[i + n/2]$
$A[i] = u$
$A[i + n/2] = v$
$\dots$
$u = A[i + 7] + A[i + 7 + n/2]$
$v = A[i + 7] - A[i + 7 + n/2]$
$A[i + 7] = u$
$A[i + 7 + n/2] = v$
end for

As we can see in Algorithm 4, we compute in a row 8 multiplications (can be adapted depending on the number of registers) and we compute the additions and the subtractions immediately after, so that we reuse the content of the registers. We have to compute some multiplications in a row in order to use efficiently the pipeline. Indeed, since those multiplications are independent, we can assume that at the end of the multiplications, the first one is finished.

The iterative part is made using blocks. Indeed, instead of computing only the first 2-points FFT for the first level, we compute directly 16-points FFT. This way, we use efficiently the registers. The multiplications are done successively, in order to use the pipeline. 16 is the parameter usually used, but it can be adapted, depending on the architecture, through the python code.

For the next levels, we also use a block strategy, but the root of unity used inside those blocks has an increasing order, depending on the level.

In the second strategy, the iterative part is straight-forward. It is enough to compute the blocks like it has been described previously. We have to compute the twiddle factors between each level in the butterfly.

For the recursive part, we have to reuse the way the first strategy that has been implemented, but in an iterative way for the part computing the small butterflies. More details are given in **Appendix B**.

7 Experimentations

7.1 One dimensional FFT

We will show here the influence of the different parameters of the implementation. In the first time, we can compare the naive implementation of the FFT (which was iterative) with the first strategy described in the previous section. We will show the differences in terms of cycles and cache-misses. The threshold used has the value 1024 : it corresponds to the size after which we call the iterative FFT.

The parameter N is the size of the input vector given to the FFT.

N	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
Old FFT	4.08s	8.35s	16.93s	36.67s	74.74s	152.65s
1st Strat.	1.27s	2.69s	5.70s	12.00s	24.80s	53.23s

As we can see above, we win a factor greater than 3 by adopting a more cache-friendly strategy. These timings have been made on the following machine : AMD Opteron(tm) Processor 6168. Let us compare the number of cycles and the cache-misses :

N	**old cache-misses**	**new cache-misses**	**old cycles**	**new cycles**
2^{23}	$264,859$	$240,404$	$9,024,307,443$	$3,907,597,584$
2^{24}	$455,035$	$437,628$	$18,393,738,625$	$8,099,174,053$
2^{25}	$842,998$	$836,671$	$37,211,499,103$	$16,783,017,148$
2^{26}	$1,727,444$	$1,626,975$	$79,700,922,054$	$34,693,748,735$
2^{27}	$3,489,699$	$3,245,469$	$162,189,284,924$	$70,432,997,418$
2^{28}	$7,171,002$	$6,616,190$	$330,219,164,322$	$148,290,744,335$

We ran the implementation on another machine : Intel(R) Xeon(R) CPU X5650 @ 2.67GHz. The timings on the Intel Machine are given in **Appendix C**.

Let us now compare the two strategies described before. By default, for the second strategy, we use butterflies of size 8, which means we are compute a radix-8 FFT.

N	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
1st Strat.	1.27s	2.69s	5.70s	12.00s	24.80s	53.23s
2nd Strat.	1.36s	2.90s	6.22s	13.06s	27.61s	58.24s

We observe that the second strategy is not as fast in practice as the first one. Indeed, it is slightly slower by 10 percent.

Let us compare the cache-misses and the cycles :

N	**1st cache-misses**	**2nd cache-misses**	**1st cycles**	**2nd cycles**
2^{23}	$240,404$	$433,467$	$3,907,597,584$	$3,898,771,724$
2^{24}	$437,628$	$778,403$	$8,099,174,053$	$8,097,450,265$
2^{25}	$836,671$	$1,473,226$	$16,783,017,148$	$16,833,741,258$
2^{26}	$1,626,975$	$2,954,530$	$34,693,748,735$	$35,257,722,765$
2^{27}	$3,245,469$	$5,890,483$	$70,432,997,418$	$73,057,301,199$
2^{28}	$6,616,190$	$11,855,501$	$148,290,744,335$	$150,736,731,538$

We deduce from the previous array that the number of cache-misses has been almost doubled in the second strategy.

Let us observe the influence of the value of **HTHRESHOLD** on the first strategy's timings. These timings are made on the AMD architecture.

N	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
HTHRESHOLD=512	1.29s	2.56s	5.87s	12.34s	25.06s	54.74s
HTHRESHOLD=1024	1.27s	2.69s	5.70s	12.00s	24.80s	53.23s
HTHRESHOLD=2048	1.34s	2.86s	6.07s	12.87s	26.35s	56.53s

It seems according to what we can see above that the optimal value of the **HTHRESHOLD** on AMD machines is 1024. Let us compare the cache-misses :

N	HTHRESHOLD=512	HTHRESHOLD=1024	HTHRESHOLD=2048
2^{23}	$807,951$	$240,404$	$368,185$
2^{24}	$1,545,191$	$437,628$	$649,700$
2^{25}	$3,026,848$	$836,671$	$1,243,836$
2^{26}	$6,096,304$	$1,626,975$	$2,518,562$
2^{27}	$12,035,181$	$3,245,469$	$4,966,387$
2^{28}	$24,147,545$	$6,616,190$	$9,803,546$

We observe that the number of cache-misses is optimal in the case **HTHRESHOLD=1024**.

To conclude, we implemented the improvement to the Montgomery multiplication. It seems that this idea does not give better timings, compared to the original Montgomery multiplication. We give here the running time on the AMD architecture :

N	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
2nd Strat.	1.36s	2.90s	6.22s	13.06s	27.61s	58.24s
New Montg. Mult.	1.54s	3.43s	7.36s	15.12s	32.48s	68.89s

8 Discussion

It appears from the experimentations that the improvement to the Montgomery's trick has not been successful. Moreover, the implementation of the sparse radix arithmetic does not give satisfying timings, when we are working on machine words. This is why the next step consists in trying to increase the size of coefficients of our polynomials. Indeed, we may try to work with several machine words. This could make the additional cost associated to the linear operations like addition or subtraction in sparse radix arithmetic negligible, compared to the gain linked to the cheap roots of unity.

Another lead could be to try to implement the new arithmetic using FPGA. Indeed, the loss related to the computation of carries after a multiplication of numbers represented in an usual radix could be compensated if the hardware allows to use some new instructions and a new representation of numbers.

Some experimental results need an explanation. Indeed, whilst the theory suggest that the second strategy implemented should be better than the first one in terms of algebraic and cache complexity, the experiments suggest that the first strategy is the best. Is it due to the eventual more efficient use of registers ? At each level of recursion involved in the implementation of the one dimensional FFT, we have to shuffle the input array. From a cache-complexity point of view, the combination of several layers of shuffle should be more efficient. In practice, it appears to be worse.

Appendix A. References

References

[BPA] Bpas library. `http://www.bpaslib.org`.

[Für07] Martin Fürer. Faster integer multiplication. In David S. Johnson and Uriel Feige, editors, *STOC*, pages 57–66. ACM, 2007.

[Für09] Martin Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39(3):979–1005, 2009.

[HK81] Jia-Wei Hong and H. T. Kung. I/o complexity: The red-blue pebble game. In *STOC*, pages 326–333. ACM, 1981.

[HvdHL14] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Even faster integer multiplication. *CoRR*, abs/1407.3360, 2014.

[Int] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[Knu97] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

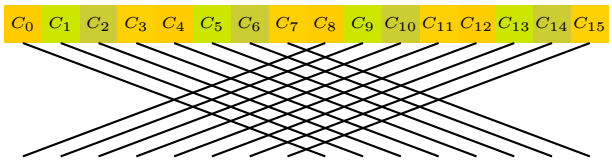
[Mon85] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

Appendix B. Implementation of the recursive FFT

We will describe here how we implemented the multiplication by twiddle factors and the computation of the small butterflies in the second strategy of **subsection 6.3**.

If we work in 16-radix FFT, and we have an input vector of size N, we will at first cut this vector in 16 chunks of size $N/16$. Let us call these chunks $(C_i)_{i\in[0,;15]}$.

We compute the multiplication by twiddle factors for each of those chunks. For $c_{ij} \in C_i$, we multiply c_{ij} by ω^{ij}. Once we have multiplied the chunks C_0 and C_8, we can add and subtract the elements of those chunks.



We can see the algorithm computing those butterflies here :

Multiplication by twiddle factors

procedure BUTTERFLY(C_0,C_8)
  for $0 \leq i \leq N/8$,$i = i + 8$ **do**
    $C_8[i] = C_8[i] * \omega^{8 \cdot i}$
    $C_8[i+1] = C_8[i+1] * \omega^{8 \cdot (i+1)}$
    $C_8[i+2] = C_8[i+2] * \omega^{8 \cdot (i+2)}$
    $C_8[i+3] = C_8[i+3] * \omega^{8 \cdot (i+3)}$
    $C_8[i+4] = C_8[i+4] * \omega^{8 \cdot (i+4)}$
    $C_8[i+5] = C_8[i+5] * \omega^{8 \cdot (i+5)}$
    $C_8[i+6] = C_8[i+6] * \omega^{8 \cdot (i+6)}$
    $C_8[i+7] = C_8[i+7] * \omega^{8 \cdot (i+7)}$
    $u = C_0[i] + C_8[i]$
    $v = C_0[i] - C_8[i]$
    $C_0[i] = u$
    $C_8[i] = v$
    ...
    $u = C_0[i+7] + C_8[i+7]$
    $v = C_0[i+7] - C_8[i+7]$
    $C_0[i+7] = u$
    $C_8[i+7] = v$
  end for
end procedure

We apply the algorithm **Butterfly** to the pairs (C_i, C_{i+8}) for $i \in [0; 8[$. We then apply the same strategy but using roots of unity of lower order, in order to compute the 16-points FFT.

Concretely, we apply for decreasing values of u the following algorithm :

Multiplication by twiddle factors

procedure BUTTERFLY(C_k,C_{k+u},u)
  for $0 \leq i \leq N/8$,$i = i + 8$ **do**
    $v = k \mod u$
    $C_{k+u}[i] = C_{k+u}[i] * \omega^{\frac{uvN}{16}}$
    $C_{k+u}[i+1] = C_{k+u}[i+1] * \omega^{\frac{uvN}{16}}$
    $C_{k+u}[i+2] = C_{k+u}[i+2] * \omega^{\frac{uvN}{16}}$
    $C_{k+u}[i+3] = C_{k+u}[i+3] * \omega^{\frac{uvN}{16}}$
    $C_{k+u}[i+4] = C_{k+u}[i+4] * \omega^{\frac{uvN}{16}}$
    $C_{k+u}[i+5] = C_{k+u}[i+5] * \omega^{\frac{uvN}{16}}$
    $C_{k+u}[i+6] = C_{k+u}[i+6] * \omega^{\frac{uvN}{16}}$
    $C_{k+u}[i+7] = C_{k+u}[i+7] * \omega^{\frac{uvN}{16}}$
    $u = C_k[i] + C_{k+u}[i]$
    $v = C_k[i] - C_{k+u}[i]$
    $C_k[i] = u$
    $C_{k+u}[i] = v$
    $\cdots$
    $u = C_k[i+7] + C_{k+u}[i+7]$
    $v = C_k[i+7] - C_{k+u}[i+7]$
    $C_k[i+7] = u$
    $C_{k+u}[i+7] = v$
  end for
end procedure

We compute the previous algorithm by increasing k first, then by decreasing u. At first, k lives in $0, 1, 2, 3, 8, 9, 10, 11$, then in $0, 1, 4, 5, 8, 9, 12, 13$ and finally in $0, 2, 4, \cdots, 14$. u decreases by a factor 2, starting from 8, until reaching 2.

Computation of small Butterflies

for $3 \geq u \geq 1$ **do**
  for $0 \leq j < 16/2^u$ **do**
    for $0 \leq i < 2^{u-1}$ **do**
      Butterfly($C_{i+j\cdot 2^u}$,$C_{i+2^{u-1}+j\cdot 2^u}$,2^u)
    end for
  end for
end for

Appendix C. Additional timings on Intel(R) Xeon(R) CPU X5650 @ 2.67GHz

Let us compare the first strategy with the old implemented of the one dimensional FFT in the BPAS Library on the Intel architecture :

N	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
Old FFT	2.86s	6.04s	12.45s	25.83s	53.15s	108.96s
1st Strat.	0.67s	1.42s	3.02s	6.38s	13.46s	28.32s

The main observation here is that we just divided by 2 the first timings we had. If we focus on the amount of cache-misses and cycles, we have more interesting statistics

:

N	old cache-misses	new cache-misses	old cycles	new cycles
2^{23}	29,395,618	15,665,846	9,800,035,659	3,326,946,112
2^{24}	57,616,488	32,216,102	20,568,943,112	6,955,646,130
2^{25}	115,551,894	66,554,828	42,265,440,870	14,334,505,067
2^{26}	231,737,115	136,596,768	87,372,621,630	29,705,443,156
2^{27}	465,328,833	281,304,231	179,345,750,087	61,548,074,907
2^{28}	932,169,367	578,830,627	366,803,917,315	127,356,990,728

We notice that the number of cache-misses has decreased by a factor 2 for the Intel machine.

Let us see what happens on the Intel architecture when we compare the first and the second strategy implemented :

N	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
1st Strat.	0.67s	1.42s	3.02s	6.38s	13.46s	28.32s
2nd Strat.	0.71s	1.52s	3.23s	6.83s	14.43s	30.38s

N	1st cache-misses	2nd cache-misses	1st cycles	2nd cycles
2^{23}	15,665,846	12,466,989	3,326,946,112	3,306,680,210
2^{24}	32,216,102	23,933,208	6,955,646,130	6,850,091,891
2^{25}	66,554,828	46,836,535	14,334,505,067	14,214,290,327
2^{26}	136,596,768	112,604,760	29,705,443,156	29,812,712,012
2^{27}	281,304,231	218,119,524	61,548,074,907	61,681,553,133
2^{28}	578,830,627	426,335,857	127,356,990,728	127,411,046,347

It seems that on the Intel machine, the amount of cache-misses has decreased. But the timings are still slower for the second strategy.

Appendix D. Assembly routine for the Montgomery Multiplication

We have below the X86 assembly routine written for the Montgomery Multiplication in the `BPAS` library :

```
inline sfixn MontMulModSpe_OPT3_AS_GENE_INLINE(sfixn u, sfixn v){
        asm("mulq %2\n\t"
                "movq %%rax,%%rsi\n\t"
                "movq %%rdx,%%rdi\n\t"
                "imulq %3,%%rax\n\t"
                "mulq %4\n\t"
                "add %%rsi,%%rax\n\t"
                "adc %%rdi,%%rdx\n\t"
                "subq %4,%%rdx\n\t"
                "mov %%rdx,%%rax\n\t"
                "sar $63,%%rax\n\t"
                "andq %4,%%rax\n\t"
```

```
                "addq %%rax,%%rdx\n\t"
                : "=d" (u)
                : "a" (u), "rm" (v), "b" (INV_PRIME), "c" (MY_PRIME)
                :"rsi","rdi");
        return u;
}
```