Multiplication of Large Integer Polynomials using Approximate FFTs

1. Introduction ✨

This repo contains two short Python/Sage proof of concept implementations (about 50 lines of code each) that enable **fast multiplication of two polynomials** of **large degree** (up to $N = 8192$) and **large integer coefficients** (up to 800 bit integers). The idea is to use available implementations of Fast Fourier Transform (FFT) that are already optimized for floating-point numbers. Even though the floating-point FFTs produce only *approximate results*, they can be used to compute *exact results* in the case of arbitrary-precision polynomial multiplication.

The main tool that is leveraged to achieve this is scipy's (https://pypi.org/project/scipy/) approximate Fast Fourier Transform (FFT) that makes use of complex numbers. The general idea is to break the two polynomials we want to multiply into small-coefficient polynomials of the same degree. `scipy` can fast multiply these small-coefficient polynomials (FFT-based multiplication takes $O(N \log N)$ time), using floating-point arithmetic, such that we get a correct result when the floating-point errors are shaved off. In the end, all these partial computations are used to construct the integer polynomial that is the product of the two initial polynomials. To break the coefficients, two methods are used, each with a separate implementation:

a. Base-decomposition
bd_pmul.py (https://github.com/rtitiu/pmul-approx-ffts/blob/main/bd_pmul.py)

b. Chinese Remainder Theorem (CRT)
crt_pmul.sage (https://github.com/rtitiu/pmul-approx-ffts/blob/main/crt_pmul.sage)

More details on how these methods are used are found in the sections below. The b) implementation follows the ideas from here (https://people.eecs.berkeley.edu/~fateman/papers/fftvsothers.pdf), which is among the few sources I found online on this approach on polynomial multiplication.

Other approches on large degree arbitrary-precision integer polynomial multiplication

- The Fast Library for Number Theory (FLINT) (https://flintlib.org/) implementation offers state of the art arbitrary-precision integer polynomial multiplication. The **Kronecker Substitution (KS)** algorithm is used to to multiply polynomials with large coefficients and large degrees (a nice description of it can be found here (https://arxiv.org/pdf/0712.4046.pdf) and here (https://web.maths.unsw.edu.au/~davidharvey/talks/kronecker-talk.pdf). All the FLINT numbers from the comparison tables below are generated using the KS implementation _fmpz_poly_mul_SS (https://flintlib.org/doc/fmpz_poly.html), which makes use of the arbitrary-precision integer multiplication algorithms from the GMP library (https://gmplib.org/).

- The NTL library (https://libntl.org/) gives state of the art modular polynomial multiplication (i.e. multiplication in $\mathbb{Z}_p[X]$, for large modulus p) via ZZpX module (https://libntl.org/doc/ZZ_pX.cpp.html) (more details here (https://www.sciencedirect.com/science/article/pii/S0747717185710553)). To multiply polynoamials over $\mathbb{Z}[X]$ a sufficiently large modulus can be used $p = p_1 \cdots p_k$, with machine-sized primes p_i, and use the **CRT** to reduce polynomial multiplication in $\mathbb{Z}_p[X]$ to the one in $\mathbb{Z}_{p_i}[X]$. This is very similar to what `crt_polymul` from this repo does. The difference is that NTL applies the **Number Theoretic Transform (NTT)** in each field $\mathbb{Z}_{p_i}$ (this is possbile because the primes p_i can be chosen in such a way) giving *exact results*, whereas `crt_polymul` uses floating-point FFT, with *approximate results*.

> ➡️ NTL is faster than FLINT for modular polynomial multiplication (benchmarks (https://libntl.org/benchmarks.pdf)).

2. Comparison Tables 📊

The goal of the comparison is to give an idea on how practical this implementation is. It is difficult to reach some definitive conclusion based on the tables below. One reason is that we have used Python, Sage and C code to generate de data. Despite this, I still think the data from the tables below is interesting and that the comparison has some value.

Besides the **`bd_polymul`** (python + numpy + scipy) and **`crt_polymul`** (sage + numpy + scipy) implementations from this repo, I included **`numpy.polymul`** and the highly-optimized C implementation for polynomial multiplication function from FLINT (https://flintlib.org/)(**F**ast **Li**brary for **N**umber **T**heory) library.

> 🐌 numpy.polymul (https://numpy.org/doc/stable/reference/generated/numpy.polymul.html) is rather slow, especially for large degree polynomials. This is because it does schoolbook polynomial multiplication, which takes $O(N^2)$ time, where N is the degree of the polynomials.

In some sense I chose to do the comparisons against the extremes (against a really slow or a really fast implementation).

> Below there are some numbers generated on a Ubuntu virtual machine with one core, on a personal laptop using **`sage9.5`**, **`python3.10.12`** (both using **`numpy1.26.3`** and **`scipy1.11.4`**) and **`flint3.0.0`** (we actually used the **`python-flint`** (https://pypi.org/project/python-flint/) wrapper).

To generate the tables below, we multiplied two polynomials with integer coefficients, sampled uniformly from $[-B_{\texttt{coef}}, B_{\texttt{coef}})$, and degree at most $N-1$.

$\log_2(B_{\texttt{coef}})$	`numpy.polymul`	`crt_polymul`	`bd_polymul`	`flint`
100	0.075s	0.010s	0.004s	0.001s
200	0.105s	0.017s	0.009s	0.002s
400	0.216s	0.039s	0.026s	0.003s
800	0.607s	0.095s	0.087s	0.005s

⏳ *Time comparison (seconds) on a single core for* $N = 1024$

$\log_2(B_{\texttt{coef}})$	`numpy.polymul`	`crt_polymul`	`bd_polymul`	`flint`
100	0.305s	0.019s	0.007s	0.003s
200	0.432s	0.036s	0.018s	0.005s
400	0.891s	0.070s	0.048s	0.008s
800	2.460s	0.191s	0.160s	0.010s

⏳ *Time comparison (seconds) on a single core for* $N = 2048$

$\log_2(B_{\texttt{coef}})$	`numpy.polymul`	`crt_polymul`	`bd_polymul`	`flint`
100	1.209s	0.042s	0.016s	0.006s
200	1.728s	0.067s	0.038s	0.010s
400	3.509s	0.148s	0.108s	0.018s
800	9.984s	0.395s	0.345s	0.032s

⌛ *Time comparison (seconds) on a single core for* $N = 4096$

$\log_2(B_{\texttt{coef}})$	`numpy.polymul`	`crt_polymul`	`bd_polymul`	`flint`
100	4.151s	0.078s	0.034s	0.013s
200	6.410s	0.149s	0.084s	0.020s
400	14.778s	0.323s	0.238s	0.035s
800	39.964s	0.831s	0.808s	0.068s

⌛ *Time comparison (seconds) on a single core for* $N = 8192$

3. Implementation Details🔍

Given two polynomials $f = a_0 + a_1X + \cdots + a_{N-1}X^{N-1} \in \mathbb{Z}[X]$ and $g = b_0 + b_1X + \cdots + b_{N-1}X^{N-1} \in \mathbb{Z}[X]$, of the same degree $N - 1$, and all coefficients bounded by $a_i, b_i \in [-B_{\texttt{coef}}, B_{\texttt{coef}})$, *the goal* is to compute

$$f \cdot g := c_0 + c_1 X + \cdots + c_{2N-2} X^{2N-2}, \text{ with } c_k = \sum_{i+j=k} a_i b_j.$$

We can assume that both polynomials have the same degree without any loss of generality. If both f and g have coefficients in $[-B, B)$, for some small enough power of two $B = 2^b$ (for eg. $B = 2^{18}$), then we can approximate $f \cdot g$ using scipy.fft.fft (https://docs.scipy.org/doc/scipy/reference/generated/scipy.fft.fft.html) (we actually use scipy.fft.rfft (https://docs.scipy.org/doc/scipy/reference/generated/scipy.fft.rfft.html) as an optimization) and round off the floating-point error to get the correct result. We quickly explain below how this is done, and then we show how to extend this in the case for large coefficients. Let's denote $\texttt{FFT}_n(f) := \texttt{scipy.fft.fft}(f, n)$ and $\texttt{FFT}_n^{-1}(\tilde{f}) := \texttt{scipy.fft.ifft}(\tilde{f}, n)$.

Quick FFT reminder 🧠

Let $\xi = \cos\left(\frac{2\pi}{n}\right) + i \cdot \sin\left(\frac{2\pi}{n}\right) \in \mathbb{C}$ be the $n-$th primitive root of unity, where n is a power of two. Let $f \in \mathbb{C}[X]$ (in particular, f can be a a polynomial with integer coefficients) be a polynomial of degree $n-1$.

Then the FFT maps this polynomial to to a complex vector:

$$\mathbb{C}[X] \ni f \xrightarrow{\texttt{FFT}_n} \tilde{f} := [f(1), f(\xi), f(\xi^2), \ldots, f(\xi^{n-1})] \in \mathbb{C}^n$$

❗ This map is **linear**: $(\alpha f + \beta g) \xrightarrow{\texttt{FFT}_n} \alpha \tilde{f} + \beta \tilde{g}$, for any scalars $\alpha, \beta \in \mathbb{C}$.

Moreover, because the degree of f is at most $n-1$, the complex values $\tilde{f} \in \mathbb{C}^n$ uniquely determine the polynomial f. For instance, one can recover the coefficients of f by polynomial interpolation (https://en.wikipedia.org/wiki/Polynomial_interpolation). Therefore it makes sense to talk about the inverse transformation:

$$\mathbb{C}[X] \ni f \xleftarrow{\texttt{FFT}_n^{-1}} \tilde{f} = [y_0, y_1, y_2, \ldots, y_{n-1})] \in \mathbb{C}^n$$

whick takes a complex vector $\tilde{f} \in \mathbb{C}^n$ and maps it to the unique polynomial f with degree $\leq n-1$ such that $f(\xi^i) = y_i$, for all $0 \leq i \leq n-1$. Notice that $\texttt{FFT}_n^{-1}(\texttt{FFT}_n(f)) = f$, for any $f \in \mathbb{C}[X]$.

> ‼️ There are efficient algorithms such as Cooley–Tukey FFT (https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm) that run in $O(n \log n)$ time, that compute $\texttt{FFT}_n^{-1}, \texttt{FFT}_n$. These algorithms enable polynomial multiplication in $O(n \log n)$ as: $f \cdot g = \texttt{FFT}_{2n}^{-1}(\texttt{FFT}_{2n}(f) \odot \texttt{FFT}_{2n}(g))$, where $\odot$ denotes the component-wise multiplication of vectors.

Now, getting back to the problem, we choose B as large as possible s.t. the operation $\texttt{round}\left(\texttt{FFT}_{2N}^{-1}(\texttt{FFT}_{2N}(f) \odot \texttt{FFT}_{2N}(g))\right)$, using the FFTs provided by `scipy`, correctly recovers $f \cdot g$. Here is a table with some working values for B, that are derived experimentally.

N	8192	4096	2048	1024
$\log_2(B)$	18	19	20	20

The values are generated in the following way:
For each N, we start with the value for B from here (https://people.eecs.berkeley.edu/~fateman/papers/fftvsothers.pdf). Then, following a similar methodology, I generate two random polynomials $f, g \in \mathbb{Z}[X]$ of degree $\leq N-1$ with coefficients in $[0, B)$ and check (using **FLINT**) that the correct values of $f \cdot g$ and f^2 are obtained (in my experiments I checked the correctness of around 2×10^5 pairs of such multiplications). If all the multiplications are correct, then I double the value of $B := 2B$ and start again.

⚠️ The valus for B obtained in my experiments are slightly more optimistic compared to the values computed here (https://people.eecs.berkeley.edu/~fateman/papers/fftvsothers.pdf). This may be due to possible different FFT implementations used to compute the FFTs or because the lack of sufficiently many tests ($\approx 10^5$ tests for each B may not be enough). Unless we use a theoretical bound that guarantees correctness for the values of B (similar to this (https://www.ams.org/journals/mcom/2003-72-241/S0025-5718-02-01419-9/S0025-5718-02-01419-9.pdf)) we need to do mores tests for higher rates of confidence.

In my experiments I noticed that $\texttt{round}\left(\texttt{FFT}_{2N}^{-1}\left(\texttt{FFT}_{2N}(f) \odot \texttt{FFT}_{2N}(g)\right)\right)$ correctly recovers the product when the polynomials are sampled with uniform coefficients in $[-B, B)$, with B chosen from the table above. It might be that when the coefficients are uniform centered in zero, the floating-point errors from the FFT computation may be smaller compared to the errors corresponding to polynomials with uniform coefficients in $[0, B)$.

Next, we show how to use this fast procedure to multiply polynomials with large coefficients.

a). Base-decomposition polynomial multiplication

For a polynomial $f(X) = \sum_{i=0}^{N-1} a_i X^i$ and an integer power of two base $B = 2^b$, we decompose each coefficient $a_i = \texttt{sign}(a_i) \cdot \sum_{j=0}^{k-1} a_{ij} \cdot B^j$, with each $a_{ij} \in [0, B)$ and $k = \lceil \log_B(B_{\texttt{coef}}) \rceil$. If we denote $f_j(X) := \sum_{i=0}^{N-1} \texttt{sign}(a_i) \cdot a_{ij} X^i$ we write:

$$f = f_0 + f_1 B + f_2 B^2 + \cdots + f_{k-1} B^{k-1}$$

✅ Each polynomial f_i has smaller coefficients, namely they all live in $[-B, B)$.

Given another polynomial g of degree $N-1$ and similar coefficient sizes, we can write $g = g_0 + g_1 B + g_2 B^2 + \cdots + g_{k-1} B^{k-1}$, for the same value of k. Again, the coefficients of the polynomials g_i live in $[-B, B)$. Now we can write the product as:

$$f \cdot g = \sum_{\ell=0}^{2k-2} \left(\sum_{i+j=\ell} f_i \cdot g_j \right) \cdot B^\ell$$

This allows the computation of the product as:

$$f \cdot g = \sum_{\ell=0}^{2k-2} \sum_{i+j=\ell} \texttt{round}\left(\texttt{FFT}_{2N}^{-1}\left(\texttt{FFT}_{2N}(f_i) \odot \texttt{FFT}_{2N}(g_j)\right)\right) \cdot B^\ell$$

> ✅The formula can be easily adapted to the case when f and g don't share the same value of k, i.e. the size of their coefficients significantly differ.

In theory, the most costly operations in the above computation are the FFT computations. In total there are $2 \cdot k$ calls to $\texttt{FFT}_{2N}$ (namely $\{\texttt{FFT}_{2N}(f_i), \texttt{FFT}_{2N}(g_i)\}_{i=0}^{k-1}$) and k^2 calls to $\texttt{FFT}_{2N}^{-1}$ (namely $\{\texttt{FFT}_{2N}^{-1}(\tilde{f}_i \odot \tilde{g}_j)\}_{0 \le i,j \le k-1}$), where $k = \lceil \log_B(B_{\texttt{coef}}) \rceil$. In practice, the table below shows the distribution of the computational time of a single call to the function **`bd_polymul`** in the current implementation (https://github.com/rtitiu/pmul-approx-ffts/blob/main/bd_pmul.py).

N	**base-decomposition**	**FFTs**	**reconstruction**
1024	15%	65%	20%
2048	17%	57%	26%
4096	14%	56%	30%
8192	14%	57%	29%

Distribution of the computational time in **`bd_polymul`** *as a percentage of the total running time when* $B_{\texttt{coef}} = 2^{400}$.

The table above points to where further optmizations could be made. To reduce the number of

FFT calls, we can do a slightly more aggresive implementation, as presented below.

A more aggressive approach 🙈

In this version we take advantage of the linear property of the direct FFT to do the same computation but with fewer calls to the function $\mathtt{FFT}_{2N}^{-1}$. For this variant the product is computed as:

$$f \cdot g = \sum_{\ell=0}^{2k-2} \mathtt{round}\left(\mathtt{FFT}_{2N}^{-1}\left(\sum_{i+j=\ell} \mathtt{FFT}_{2N}(f_i) \odot \mathtt{FFT}_{2N}(g_j)\right)\right) \cdot B^{\ell}$$

> ✅ Fewer calls to $\mathtt{FFT}_{2N}^{-1}$ and roundings: only $(2k-1)$ compared to the previous k^2, where $k \approx \log_B(B_{\mathtt{coef}})$. Consequently, the total FFT time decreases.

> ❗ The floating-point errors are potentially larger because we round less often and the errors add up. This means that that for correctness we might need to use a smaller B. In practice, for polynomials with uniform coefficients centered in zero $[-B, B)$ seems to work for the same values of B. This implementation is not included in the repo. More testing is needed.

---

b). CRT-based polynomial multiplication

Let $B = 2^b$ as in the table above. In order to make use of the Chinese Remainder Theorem (CRT), for each pair of N and B we generate k primes $q_1, q_2, \ldots, q_k$ of similar sizes $\log_2(q_i) \approx \log_2(B)$ but $q_i \leq B$ for all $0 \leq i \leq k$, such that $q_1 \cdot q_2 \cdot \cdots \cdot q_k > 2 \cdot N \cdot B_{\mathtt{coef}}^2$. Therefore $k \approx 2 \cdot \log_B(B_{\mathtt{coef}}) + \log_B(N)$.

Quick CRT reminder 🧠

If we denote $q := q_1 q_2 \cdots q_k$ and $\mathbb{Z}_q$ the integers $\mod q$, then we have the map that takes an integer a to the set of remainders modulo the prime factors of q.

$$\mathbb{Z}_q \ni a \xrightarrow{\texttt{CRT}} (a \bmod q_i)_{1 \le i \le k} \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$$

> ❗ This map preserves both **addition** (➕) and **multiplication** (❌)

Moreover, given a vector $(a_1, a_2, \ldots, a_k) \in \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_k}$ we can uniquely compute the integer $a \in \mathbb{Z}_q$ that maps to this vector as:
$a := \left(\sum_{i=1}^k a_i \cdot \tilde{q}_i \cdot \frac{q}{q_i}\right) \bmod q$, where $\tilde{q}_i := \left(\frac{q}{q_i}\right)^{-1} \bmod q_i$, giving the inverse map

$$\mathbb{Z}_q \ni a \xleftarrow{\texttt{CRT}^{-1}} (a_1, a_2, \ldots, a_k) \in \mathbb{Z}_{q_1} \times \mathbb{Z}_{q_2} \times \cdots \times \mathbb{Z}_{q_k}$$

, such that $\texttt{CRT}^{-1}(\texttt{CRT}(a)) = a$, for any $a \in \mathbb{Z}_q$.

> ‼️ In the actual implementation, the inverse map is computed using Garner's algorithm (https://redirect.cs.umbc.edu/~lomonaco/s08/441/handouts/GarnerAlg.pdf) that is already available in Sage (https://doc.sagemath.org/html/en/reference/rings_standard/sage/arith/multi_modular.html). This speeds up the computation in the case where the primes never change, because precomputations can be made.

The above map (and its inverse) naturally extends to polynomials. The map is applied on each coefficient. Moreover, both maps preserve addition and multiplication of polynomials.

To break the coefficients of the polynomials f and g into smaller chunks we compute the CRT components $(f_1, f_2, \ldots, f_k) = \texttt{CRT}(f)$ and $(g_1, g_2, \ldots, g_k) = \texttt{CRT}(g)$ i.e. $f_i := f \bmod q_i$ and $g_i := g \bmod q_i$ for all $1 \le i \le k$. Notice that f_i, g_i have coefficients in $[0, q_i) \subset [0, B)$. So we can

correctly multiply $f_i \cdot g_i \bmod q_i := \texttt{round}\left(\texttt{FFT}_{2N}^{-1}(\texttt{FFT}_{2N}(f_i) \odot \texttt{FFT}_{2N}(g_i))\right) \bmod q_i$. The reduction $\bmod q_i$ is needed because the sage implementation of Garner's algorithm (https://doc.sagemath.org/html/en/reference/rings_standard/sage/arith/multi_modular.html) works only when the primes (hence the CRT components) can be represented on 32 bits.
Next the inverse CRT transformation is applied to get the result:

$$h := \texttt{CRT}^{-1}\left((f_1 \cdot g_1 \bmod q_1, f_2 \cdot g_2 \bmod q_2, \ldots, f_k \cdot g_k \bmod q_k)\right)$$

By CRT, the resulting polynomial h is equal to $f \cdot g \bmod q$. Because we use this particular sage function for CRT, h has coefficients represented in $[-q/2, q/2)$. Since each coefficient of $f \cdot g$ can be bounded by $N \cdot B_{\texttt{coef}}^2$ in absolute value, combined with the bound $q > 2N \cdot B_{\texttt{coef}}^2$, the result must be equal to the product of polynomials computed over $\mathbb{Z}$.

The procedure above makes $2k$ calls to $\texttt{FFT}_{2N}$ and k calls to $\texttt{FFT}_{2N}^{-1}$, where $k \approx 2 \cdot \log_B(B_{\texttt{coef}})$. In comparison with **`bd_polymul`**, this algorithm makes fewer FFT calls:
$6 \log_B(B_{\texttt{coef}}) \ll (\log_B(B_{\texttt{coef}}))^2 + 2 \cdot \log_B(B_{\texttt{coef}})$, but the downside is that the the CRT computations seem to be expensive.

N	$\texttt{CRT}$	**FFTs**	$\texttt{CRT}^{-1}$
1024	44%	23%	32%
2048	42%	23%	34%
4096	46%	23%	$31s\%$
8192	46%	24%	30%

Distribution of the computational time in **`crt_polymul`** *as a percentage of the total running time when $B_{\texttt{coef}} = 2^{400}$.*