Polynomial Multiplication with Parallelized FFT writeup

Alex Pan

April 2024

Everything before section 7 (hypertriangle) isn't useful for raising multivariate polynomials to powers.

This is because FFT in k-dimensions has a time complexity of $O(n^k \log n)$. FFT in 1 dimension is $O(n \log n)$, and in k dimensions, the 1-D FFT needs to be repeated roughly n^{k-1} times, making the total complexity $O(n^k \log n)$. So, even with the number of applications FFT has, it isn't the best choice when raising higher-dimensional polynomials to powers.

1 Overview

Using the Fast Fourier Transform (FFT), we can multiply two univariate polynomials of in $O(n \log(n))$ time, compared to the trivial distributive algorithm of $O(n^2)$. In addition, this process is fully parallelizable, making its computation extremely quick using GPUs. In order to understand the algorithm, we need to understand a few steps:

- Polynomial Interpolation
- DFT and FFT
- Parallelization

2 Polynomial Interpolation

Given a set of $n+1$ points $(x_0, y_0), (x_1, y_1)...(x_n, y_n)$, with all x_i being distinct, there exists a unique polynomial of degree *at most* n that passes through all $n+1$ points.

To verify this, we start by writing the polynomial $p(x)$ as $a_0 + a_1x^1 + a_2x^2 + ... + a_nx^n$, we can write this as a matrix product:

$$\begin{bmatrix} 1 & x & x^2 & \dots & x^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = p(x) = y$$

Doing this for all x_i, we get the *Vandermonde Matrix* and the equation:

$$\overbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{bmatrix}}^{V} \overbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}}^{\vec{a}} = \begin{bmatrix} p(x_0) \\ p(x_1) \\ p(x_2) \\ \vdots \\ p(x_n) \end{bmatrix} = \overbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}^{\vec{y}}$$

To show that $\vec{a}$ is exists and is unique, we need to show that the Vandermonde matrix is bijective. Because it is square, we just need to verify that the matrix is invertible. From Wikipedia, the determinant of this matrix is nonzero if and only if all x_i are distinct, which we specified earlier. So, the polynomial represented by $\vec{a}$ exists and is unique, with $\vec{a} = V^{-1}\vec{y}$. (The non-linear algebra proofs are a lot more interesting)

An important comment to make is that the degree of the interpolated polynomial doesn't have to be n; the degree d can be any value from $1 \leq d \leq n$ this is the case when $a_{d+1} \dots a_n = 0$. This means that given a set of data points, the interpolated polynomial will be of the lowest degree possible.

If we evaluate two polynomials $p(x)$ and $q(x)$ at a point x_0, then multiply their outputs together, we get the same value as if we multiplied the two polynomials first, then evaluated them at x_0. This forms the motivation for using polynomial interpolation to multiply polynomials.

The algorithm for multiplying polynomials $p(x)$ and $q(x)$, with coefficients $\vec{a}$ and $\vec{b}$ using interpolation is as follows:

1. Determine degree k of product: This is just the degrees of $p_1(x)$ and $p_2(x)$ added together: $n + m = k$
2. Select arbitrary $k+1$ distinct values $x_0, x_1, \dots, x_k$. Evaluate the polynomials at all of the values using the $(k+1) \times (k+1)$ Vandermonde matrix: $V\vec{a} = \vec{y}$, $V\vec{b} = \vec{z}$
3. Perform component-wise multiplication on $\vec{y}$ and $\vec{z}$:
$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} \circ \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ \vdots \\ z_k \end{bmatrix} = \begin{bmatrix} y_0 z_0 \\ y_1 z_1 \\ y_2 z_2 \\ \vdots \\ y_k z_k \end{bmatrix}$$
4. Interpolate $\vec{y} \circ \vec{z}$ - find the unique polynomial that passes through all of the chosen points - compute $V^{-1}(\vec{y} \circ \vec{z})$

This algorithm works, but runs in $O(n^3)$ time because finding the inverse of a matrix is $O(n^3)$, making it significantly slower than trivial multiplication. DFT fixes the finding the inverse part.

3 DFT and FFT

3.1 DFT

The Discrete Fourier Transform is a function maps a vector containing elements $[a_0, a_1, ...a_n]$ with $a_i \in \mathbb{C}$ to a vector containing elements $[\hat{a}_0, \hat{a}_1, ...\hat{a}_n]$ with the following rule:

$$\hat{a}_k = \sum_{j=0}^{n-1} x_j \cdot e^{-2\pi i \frac{k}{N} j}$$

In the context of polynomial multiplication, DFT just evaluates a polynomial of degree n at the n complex roots of unity. So, with $\omega = e^{\frac{2\pi i}{n}}$ the Vandermonde matrix looks like this:

$$\overbrace{\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix}}^{V} \overbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}}^{\vec{a}} = \overbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}^{\vec{y}}$$

DFT is doing the same thing as the first step in the above algorithm, except it specifies the arbitrary points to be the k roots of unity.

This is only useful because the inverse Vandermonde matrix V^{-1} is already known - we don't need to calculate it:

$$V^{-1} = \frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-1(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

To verify: Checking entries on the diagonal, multiplying ω by its inverse ω^{-1} gives 1, and adding these up n times and dividing by n gives 1's on the diagonal. Checking entries not on the diagonal and in the first row or column, we see that adding the roots of unity just gives 0. I don't feel like proving the other entries.

With this, we no longer have to compute the inverse of the Vandermonde matrix in $O(n^3)$ time, and our algorithm has been shortened to $O(n^2)$, the complexity of multiplying a matrix by a vector. However, since we're doing so many extra steps, this algorithm is still slower, and requires more memory than the trivial algorithm.

3.2 FFT

Fast Fourier Transform in the context of Discrete Fourier Transforms refers to a fast way to compute $V\vec{a} = \vec{y}$. Given a polynomial $p(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_{n-1}x^{n-1}$, and assuming that n is a power of two, we can rewrite it as:

$$p(x) = p_1(x^2) + x * p_2(x^2), \text{ where}$$
$$p_1(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{\frac{n}{2}-1}$$
$$p_2(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{\frac{n}{2}-1}$$

This decomposition perfect for evaluating a polynomial at a complex root of unity. To see this, we look at the evaluation of a polynomial at the $n = 8$ roots of unity using the decomposition above (this looks horrible but I'll live with it for a bit):

$$\begin{bmatrix} p(\omega^0) \\ p(\omega^1) \\ p(\omega^2) \\ p(\omega^3) \\ p(\omega^4) \\ p(\omega^5) \\ p(\omega^6) \\ p(\omega^7) \end{bmatrix} = \begin{bmatrix} p_1(\omega^0) \\ p_1(\omega^2) \\ p_1(\omega^4) \\ p_1(\omega^6) \\ p_1(\omega^8) \\ p_1(\omega^{10}) \\ p_1(\omega^{12}) \\ p_1(\omega^{14}) \end{bmatrix} + \begin{bmatrix} \omega^0 p_2(\omega^0) \\ \omega^1 p_2(\omega^2) \\ \omega^2 p_2(\omega^4) \\ \omega^3 p_2(\omega^6) \\ \omega^4 p_2(\omega^8) \\ \omega^5 p_2(\omega^{10}) \\ \omega^6 p_2(\omega^{12}) \\ \omega^7 p_2(\omega^{14}) \end{bmatrix}$$

Note that with one recursive step, we only have to evaluate p_1 p_2, both with $n/2 = 4$ terms, at $n/2 = 4$ roots of unity. This is because when we square the roots of unity $\omega^0, \omega^1, \omega^2, ..., \omega^n$, we get $\omega^0, \omega^2, \omega^4, ..., \omega^{2n}$, the roots of unity generated by ω^2, of which there are $n/2$ of. So, with each recursive decomposition, problem becomes evaluating two polynomials with half the size of the original polynomial at half of the roots of unity. Furthermore, letting ω_n be the nth root of unity, we see that $\omega_n^{n/2} = -\omega_n$, meaning we really only need to evaluate half of the second vector, and just need to negate the other half.

Note that letting n be a different power $n = m^k$ gives access to a radix-m version of the recursive algorithm.

However, since we want to parallelize this, and not all NVIDIA GPUs support recursion yet, we need a way to build this algorithm bottom-up.

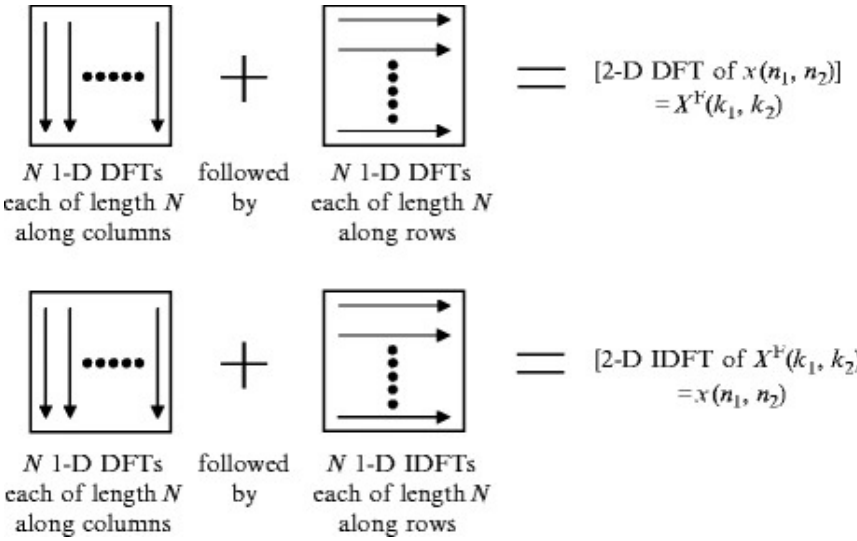
write cooley-tukey

Now our algorithm is $O(n \log_2 n)$, asymptotically much better than our trivial $O(n^2)$ algorithm.

4 Multinomial Multiplication

The multidimensional version of this algorithm follows much of the same steps as the 1-dimensional algorithm. Given a polynomial of k variables, we can represent its coefficients in a k-dimensional matrix/tensor X, We "DFT" X, multiply the corresponding coefficients of X_1 and X_2, and "IDFT" the result.

We need to define what the DFT in k dimensions is first. A diagram from *Fast Fourier Transform - Algorithms and Applications* explains how the DFT works in 2 dimensions:



So, in order to send a matrix/tensor to its frequency domain representation, we iteratively DFT each vector that makes up the matrix/tensor.

If the matrix is $n \times n$, then we need to apply $2n$ DFTs, each of complexity $O(n \log n)$, which already leads to a complexity of $O(n^2 \log n)$, already more than the trivial algorithm. The complexity of DFTing a $n \times n \times n$ tensor becomes $O(n^3 \log n)$, and generally, DFTing a k-dimensional tensor, each dimension with size n, will have complexity $O(n^k logn)$.

Because the polynomials we are interested in multiplying are homogeneous, there is a small optimization we can make. The tensor representation of these polynomials will only have entries on their "opposite diagonal," or in the case of $dim > 2$, the entries will be on their "opposite diagonal slice" (I have no idea what vocabulary I should be using here). In the case of 2d, multiplying a matrix by an "opposite diagonal" matrix simply rotates the matrix and scales the entries, and in the case of $dim > 2$, multiplying the tensor by an "opposite diagonal slice" tensor simply permutes the dimensions, depending on which dimension we're applying DFT to first, and scales the entries by values in the "opposite diagonal slice". This computation is $O(n^k)$, but extremely highly parallelizable, and I think it can just be embedded into a future DFT computation. This reduces the dimensions we have to DFT by one dimension, making our complexity $O(n^{k-1} logn)$.

5 Parallelization

6 NTT

(This stuff I don't fully understand, so I'm mostly just writing this for my own understanding)

Because the end goal is to multiply very high degree polynomials in $\mathbb{F}_p[x]$, (polynomials with coefficients that are elements $\mathbb{F}_p$, the field of integers mod p), the complex DFT starts to lose viability because of very large floats that lose precision, and there is no trivial way to embed mod p into $\mathbb{C}$

This is where the DFT over a ring comes in, which generalizes the complex root of unity method to all rings. The Vandermonde matrix is the same, except we need to replace $\omega_n = e^{\frac{2\pi i}{n}}$ with α, where α is a principal nth root of unity (n is the length of the vector we are transforming). In the case of fields, which $\mathbb{F}_p$ is, the principal nth root of unity just needs to satisfy $\alpha^k \neq 1$ for $1 \leq k < n$, where the exponentiation is repeated multiplication. For the inverse DFT, if n doesn't have a multiplicative inverse in F, then the transformation is not invertible. Also, if a principal nth root of unity doesn't exist, then the transformation won't work either.

To do a quick example, let's say we want to multiply $1+2x$ and $3+4x \mod 5$. Our expected result is $3+3x^2$, corresponding to a vector of $[3, 0, 3, 0]$. Luckily, 2 is a nth principal root of unity in $\mathbb{F}$: $2^1 = 2, 2^2 = 4, 2^3 = 3$. So:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 3 \\ 1 & 4 & 1 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 4 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 3 \\ 1 & 4 & 1 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 4 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 3 \\ 0 \\ 4 \\ 2 \end{bmatrix} \circ \begin{bmatrix} 2 \\ 1 \\ 4 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$4 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3 & 4 & 2 \\ 1 & 4 & 1 & 4 \\ 1 & 2 & 4 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

So, we do indeed get the right output as our vector. However, we won't necessarily be so lucky to have an nth principal root of unity, and we absolutely won't have a principal root of unity when the degree of the polynomial is greater than the order of the field. So, we instead need to look at $GF(p^n)$

7 Pascal's Hypertriangle (Pascal's Simplex)

Just as Pascal's Triangle is referenced to compute the coefficients of each term in a binomial expansion, Pascal's simplex can be used to compute the coefficients of each term in a multinomial expansion.

Say we want to compute $(a_0+a_1+...+a_k)^n$. The coefficient in front of the term $a_0^{d_0}a_1^{d_1}...a_k^{d_k}$ can be computed by $\frac{n!}{d_0!d_1!...d_k!}$. For example, if we wanted to expand $(a+b+c)^3$, then the coefficient in front of a^2b would be $\frac{3!}{2!1!0!} = 2$. Note that the sum of all of the degrees of the result equals n, or $\sum_{i=0}^{k} d_i = n$

Using stars and bars (distributing n degrees amongst k variables), we find that we have to compute $\binom{n+k-1}{k-1}$ coefficients.

The intuitive way to apply this is to compute all of the coefficients, and given a multinomial, map every $\binom{n+k-1}{k-1}$ term to the coefficient in front of it, and add all of them up.

Easy Multinomial Expansion Algorithm (our current one)

Given a multinomial and a power: $(a_1 + a_2 + ... + a_k)^n$, where a_i represents a term $c_i x^p y^q z^r ...$

1. Generate all $\binom{n+k-1}{k-1}$ possible terms of the expansion, store into array termPowers. termPowers will be a 2d array , with elements looking like $(d_1, d_2, ..., d_k)$. Initialize result array with same length as termPowers.
2. Iterate through every index of termPowers. Compute the multinomial coefficient, which will be $\frac{n!}{d_1!d_2!...d_k!}$. Raise c_i^d for each index i in $(d_1, d_2, ..., d_k)$, and multiply c by the multinomial coefficient. Also compute the resulting degrees of the variables, and send to result

3. done

GPU memory required: $4(\binom{n+k-1}{k-1} * (k + 1 + numvars + 1)) + 4(k * (numvars + 1))$ bytes

n is the degree we're raising to, k is the number of starting terms.

4 comes from integers being stored as Int32's, the binomial coefficient comes from the ending number of terms, k comes from each k-integer composition, 1 comes from multinomial coefficients, and the other 1 comes from the ending coefficients.