Polynomial Multiplication with Parallelized FFT writeup

Alex Pan

April 2024

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 $x_0, x_1, \ldots, x_k$. 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:

$$\overbrace{\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega_0^2 & \ldots & \omega_0^{n-1} \\ 1 & \omega^2 & \omega_0^4 & \ldots & \omega_0^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega_0^{2(n-1)} & \ldots & \omega_0^{(n-1)(n-1)} \end{bmatrix}}^{V} \overbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}}^{\vec{a}} = \overbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}^{\vec{y}}$$

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:

$$V^{-1} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega^{-1} & \omega_0^{-2} & \ldots & \omega_0^{-(n-1)} \\ 1 & \omega^{-2} & \omega_0^{-4} & \ldots & \omega_0^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-1(n-1)} & \omega_0^{-2(n-1)} & \ldots & \omega_0^{-(n-1)(n-1)} \end{bmatrix}$$

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:

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

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):

$$\begin{bmatrix} p(\ominus) \\ p(\oslash) \\ p(\odot) \\ p(\otimes) \\ p(\ominus) \\ p(\oslash) \\ p(\odot) \\ p(\otimes) \end{bmatrix} = \begin{bmatrix} p_1(\ominus) \\ p_1(\odot) \\ p_1(\ominus) \\ p_1(\odot) \\ p_1(\ominus) \\ p_1(\odot) \\ p_1(\ominus) \\ p_1(\odot) \end{bmatrix} + \begin{bmatrix} \ominus p_2(\ominus) \\ \oslash p_2(\odot) \\ \odot p_2(\ominus) \\ \otimes p_2(\odot) \\ \ominus p_2(\ominus) \\ \oslash p_2(\odot) \\ \odot p_2(\ominus) \\ \otimes p_2(\odot) \end{bmatrix}$$

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.

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