Parallel FFT

Ritwik Sen

September 6, 2023

1 Introduction

In this document we seek to provide exposition on the Parallel Split-Radix algorithm to compute the Discrete Fourier Transform of a polynomial. To do so we introduce the relevant algorithms necessary to describe it. It is assumed that the reader is familiar with the recursive algorithm for FFT.

2 In-place iterative form of FFT

First we describe how to "flatten" the recursive algorithm of FFT such that we can describe it as an iteration. To do so we recall that fundamentally, given a description of a polynomial $f \in \mathbb{F}^{<n}[X]$ in terms of its coefficients $f = [a_0, \dots, a_N]$, and an N-th primitive root of unity ω_N we want to compute $T_N(f) = \left[f(\omega_N^0), \dots, f(\omega_N^{N-1})\right]$. We assume that $N = 2^m$ for some $m \in \mathbb{N}$, which is that N is a power of 2.

In computing these evaluations we will exploit the following properties of ω_N,

1. $\omega_N^{N/2} = -1$
2. $\omega_N^2 = \omega_{\frac{N}{2}}$

Now consider the i-th element of the desired output vector, $f(\omega_N^i)$. Clearly we can decimate the output elements into odd and even indices. Suppose $i = 2k$ for $0 \le k < \frac{N}{2}$,

$$\begin{aligned}
f(\omega_N^{2k}) &= \sum_{i=0}^{N-1} a_i \omega_N^{2ki} \\
&= \sum_{i=0}^{N-1} a_i \omega_{\frac{N}{2}}^{ki} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_i \omega_{\frac{N}{2}}^{ki} + \sum_{i=\frac{N}{2}}^{N-1} a_i \omega_{\frac{N}{2}}^{ki} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_i \omega_{\frac{N}{2}}^{ki} + \sum_{i=0}^{\frac{N}{2}-1} a_{\frac{N}{2}+i} \omega_{\frac{N}{2}}^{\frac{N}{2}+ki} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_i \omega_{\frac{N}{2}}^{ki} + \sum_{i=0}^{\frac{N}{2}-1} a_{\frac{N}{2}+i} \omega_{\frac{N}{2}}^{ki} \\
&= \sum_{i=0}^{\frac{N}{2}-1} \left(a_i + a_{\frac{N}{2}+i}\right) \omega_{\frac{N}{2}}^{i}
\end{aligned}$$

And for i odd, which is, when i is of the form $2k+1$ for $0 \le k < \frac{N}{2} - 1$.

$$\begin{aligned}
f(\omega_N^{2k+1}) &= \sum_{i=0}^{N-1} a_i \omega_N^{(2k+1)i} \\
&= \sum_{i=0}^{N-1} \left(a_i \omega_N^{i}\right) \omega_N^{2ki} \\
&= \sum_{i=0}^{\frac{N}{2}-1} \left(a_i \omega_N^{i}\right) \omega_{\frac{N}{2}}^{ki} + \sum_{i=\frac{N}{2}}^{N-1} \left(a_i \omega_N^{i}\right) \omega_{\frac{N}{2}}^{ki} \\
&= \sum_{i=0}^{\frac{N}{2}-1} \left(a_i \omega_N^{i}\right) \omega_{\frac{N}{2}}^{ki} + \sum_{i=0}^{\frac{N}{2}-1} \left(a_{\frac{N}{2}+i} \omega_N^{\frac{N}{2}+i}\right) \omega_{\frac{N}{2}}^{\frac{N}{2}+ki} \\
&= \sum_{i=0}^{\frac{N}{2}-1} \left(\left(a_i - a_{\frac{N}{2}+i}\right) \omega_N^{i}\right) \omega_{\frac{N}{2}}^{i}
\end{aligned}$$

Then we can define polyomials g_0 and g_1 in the following way,

$$\begin{aligned}
g_0(x) &= \sum_{i=0}^{\frac{N}{2}-1} (a_i + a_{\frac{N}{2}+i}) x^i \\
g_1(x) &= \sum_{i=0}^{\frac{N}{2}-1} (a_i - a_{\frac{N}{2}+i}) \omega_N^i x^i
\end{aligned}$$

These polynomials are such that for k such that $0 \leq k < \frac{N}{2}$, $f(\omega_N^{2k}) = g_0(\omega_N^{2k})$ and $f(\omega_N^{2k+1}) = g_1(\omega_N^{2k})$. Thus we can split the problem of computing N evaluations of f to computing $\frac{N}{2}$ evaluations for 2 problems of size $\frac{N}{2}$. Doing so recursively and noticing for each polynomial we require $O(N)$ additions and multiplications, and through $\log(N)$ splits we achieve $O(N\log(N))$ time complexity.

We now give a description of this recursion in psuedocode. We assume that for $0 \leq i < \frac{N}{2}, \omega_N^i$ are all precomputed and stored in an array ω, such that their index corresponds to their power. To transform a polynomial of degree m, where m is a power of 2, less than N, we can use the fact that $\omega_m = \omega_N^{N/m}$, to compute the transform of subproblems by indexing a subarray of ω.

Algorithm 1 Recursive description of Radix-2 DIF FFT

```
function FFT(a)
    ▷ The input is assumed to be an array of size N, describing the coefficients of
      the polynomial to be transformed. ◁
    $size = len(a)$
    $half = size/2$
    $stride = N/size$
    ▷ The stride determines the step with which we index powers of $\omega_N$, in order
      to transform subproblems. ◁
    if $size = 1$ then
        return $a$
    else
        $a_0 = [0; half]$
        $a_1 = [0; half]$
        for $i = 0, \ldots, half - 1$ do
            $a_0[i] = a[i] + a[half + i]$
            $a_1[i] = (a[i] - a[half + i]) * \omega[i * stride]$
        FFT($g_0$)
        FFT($g_1$)
        return $\left(a_0[0], a_1[0], \ldots, a_0[half - 1], a_1[half - 1]\right)$
```

Notice now that at each step the only computation is that of the coefficients of the polynomials corresponding to the odd and even positions of the output. This can be illustrated by a butterfly diagram, called the Gentleman-Sande butterfly illustrated below.

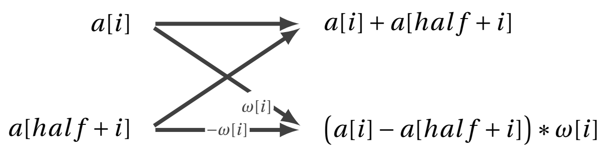

Figure 1: The Gentleman-Sande butterfly

We can thus describe the same procedure in-place in the following way,

Algorithm 2 Recursive in-place description of Radix-2 DIF FFT

```
function FFT(a)
    ▷ The input is assumed to be an array of size N, describing the coefficients of the polynomial to be transformed. ◁
    $size = len(a)$
    $half = size/2$
    $stride = N/size$
    ▷ The stride determines the step with which we index powers of $\omega_N$, in order to transform subproblems. ◁
    if $size = 1$ then
        return $a$
    else
        $temp = a[i]$
        for $i = 0, \dots, half - 1$ do
            $a[i] = temp + a[half + i]$
            $a[half + i] = (temp - a[half + i]) * \omega[i * stride]$
        FFT($a[0 : half - 1]$)
        FFT($a[half : size - 1]$)
        return $a$
```

Interestingly, while this gives our evaluations of f at different powers of ω_N, the resulting output is bit-reversed. We now give a formal proof of these claims.

Proposition 2.1. *Algorithm 2 computes and assigns to $a[i_n \dots i_0]$, $f(\omega_N^{i_n \dots i_0})$, where $i_n \dots i_0$ is the binary representation of the integer defining the index.*

Proof. For input a, the coefficients of the polynomials are assigned to the first half if they correspond to even evaluations or second half if they correspond to odd evaluations. Since we assume that N is a power of 2, the first bit of integers less than $\frac{N}{2}$ must be 0 and those greater than or equal to $\frac{N}{2}$ must be 1. Hence viewing indices in their binary representation, the coefficients of the polynomial for evaluations of the form $f(\omega^{i_n \dots 0})$, are assigned to $a[0 i_n \dots i_1]$ and $f(\omega^{i_n \dots 1})$ are assigned to $a[1 i_n \dots i_1]$. If this is true at every recursive call, it is implied that at the end of the algorithm, $a[i_0 \dots i_n] = f(\omega^{i_n \dots i_0})$. □

We now describe how we parallelise this algorithm in the proceeding section.

3 Parallel Radix-2 DIF FFT

We use Rayon to parallelise our recursion. While the number of branches less than the number of threads, we use Rayon's parallel iterators at each recursive step in order to utilise all logical threads while the problem size is large, we spawn 2 worker threads at every recursive split with a "join". Once the number of branches is equal to the number of threads, the algorithm continues sequentially in each branch, we call joins on the subsequent branches because rayon functions using potential parallelism, where if there are no idle CPUs, both worker threads at a join are computed by the same CPU. Further, using work-stealing, if any CPU completes it's threads and is idle, it will "steal" tasks from other threads.

Thus, we rely on rayon's recursive splitting of tasks to distribute work between available CPUs while there are more branches than threads, after which every branch continues sequentially when divided equally between threads.

Algorithm 3 Parallel FFT with rayon

```
function PAR-FFT(poly, omega, step, branches, threads)
    ▷ The input is assumed to be an array of size N, describing the coefficients of
      the polynomial to be transformed. ◁
    $size = len(poly)$
    $half = size/2$
    ▷ The stride determines the step with which we index powers of $\omega_N$, in order
      to transform subproblems. ◁
    if size > 1 then
        if branches<threads then
            for $i = 0, \ldots, half - 1$, rayon :: par − iter do
                $temp0 = a[i] + a[half + i]$
                $temp1 = (a[i] - a[half + i]) * \omega[i * step]$
                $a[i] = temp0$
                $a[half + i] = temp1$
        else
            for $i = 0, \ldots, half - 1$ do
                $temp0 = a[i] + a[half + i]$
                $temp1 = (a[i] - a[half + i]) * \omega[i * step]$
                $a[i] = temp0$
                $a[half + i] = temp1$
        rayon::join(
            par-fft(poly[..halfsize], omega, step*2,branches*2, threads),
            par-fft(poly[halfsize..], omega, step*2,branches*2, threads),
        )
```

Notice the only difference in the above algorithm with respect to the sequential description is the use of rayon's parallel iterator when there are less branches than threads and the invoking of rayon's join function in order to spawn the 2 subproblems at each step to be solved potentially parallelly. This means that once branches exceed available cpus, calling join is equivalent to calling further recursive calls sequentially within the current thread with low-overhead as claimed by rayon.

Use of Parallel Iterators in the first stages.

Rayon's parallel iterator, effectively recursively calls joins on an iterator object, such that the iterator is split and the latter half is stolen by idle CPUs until the computation of the iterator is is split into even branches among CPUs. Further even if the load is not balanced, once CPUs finish their task queue, they can steal tasks from the other task queues. When the CPU that owns that task queue reaches the stolen task, it skips it and computes the next task.

As described in the previous section, the output of the above algorithm returns the desired evaluations in bit reversed order with the same reasoning as the sequential case. Thus we wrap the function call in another that passes the default parameters and permutes the result in parallel.

```
function PAR-EVAL-POLY(poly, omega, threads)
    PAR-FFT(poly,omega,1, branches,threads)
    PAR-PERMUTE(poly)
```

The worst-case run-time for parallel radix-2 FFT is $O(\frac{N}{p}log(\frac{N}{p}))$, where N is the size of the input and p is the number of processors.