Fast Fourier Transform on CPU and GPU

Chia-Lun Tsai chialun2@illinois.edu
Computer Science · University of Illinois

ILLINOIS

Problem Statement

The *discrete Fourier transform (DFT)* transforms a sequence of N complex numbers $\{x_n\} := x_0, x_1, ..., x_{N-1}$ into another sequence of complex numbers, $\{X_k\} := X_0, X_1, ..., X_{N-1}$, which is defined by: [1]

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi \frac{k}{N} n}$$

. For the naive calculation, it has time complexity $\Theta(N^2)$.

Fast Fourier transforms (FFT) are $O(NlogN)$ algorithms to compute the discrete Fourier transform [2]. The most important FFT is known as the **Cooley-Tukey** algorithm, which uses divide-and-conquer strategy to compute discrete Fourier Transform. First we compute the DFTs of the even-indexed inputs and of the odd-indexed inputs, and then combine the two results to produce the DFT of the whole sequence. This idea can then be performed recursively to reduce the overall runtime to $O(NlogN)$. Let E_k denote the DFT of even-indexed terms, and O_k denote the DFT of odd-indexed terms. After some steps, the final DFT can be computed as:

$$X_k = E_k + e^{-2\pi i \frac{k}{N}} O_k$$
$$X_{k+N/2} = E_k - e^{-2\pi i \frac{k}{N}} O_k$$

for $k = 0, 1, \dots, N/2 - 1$. This is called **Radix-2 DIT**. The term $e^{-2\pi i \frac{k}{N}}$ is commonly referred to as the **twiddle factor** W_N^k, which represents the complex exponential multiplier used to combine the even and odd DFTs. These twiddle factors play a crucial role in enabling the recursive structure and efficiency of the FFT algorithm. [3]

The characteristic of Cooley-Tukey algorithm is that we need to do bit reversal to sort the result of FFT in a natural order. In the **Stockham** algorithm, even though there is no bit reversal, the result of FFT is sorted in a natural order. The method is for each stage, the output is written directly to the corresponding position in the input for the next stage. This algorithm removes the overhead of bit reversal, which accounts for 15% of total computation time based on profiling results.

CPU Approach

For Stockham algorithm, first, the outer loop loops for `stride`, which means the distance between the FFT pair. Inside the outer loop we have `block_size`, which means the size of can use same twiddle factor `w` . For each block, we compute the angle as $\theta = \frac{2\pi \cdot \text{block\_index}}{N}$, and the corresponding twiddle factor is $w = (\cos\theta, -\sin\theta)$. Although it seems that each block performs a local size-2 FFT, the carefully computed `w` reveals that we are actually executing a size-N/stride FFT. The local size-2 FFT always outputs at indices $\text{block\_idx} \cdot \text{stride} + \text{block\_offset}$ and $\text{block\_idx} \cdot \text{stride} + \text{block\_offset} + \frac{N}{2}$.

GPU Approach

For the Stockham algorithm, the Radix-2 implementation is similar to the CPU version. We load all the data to shared memory, and we perform operations on it to reduce the overhead of read/write to global memory.

There is no limitation that we can only choose 2 as Radix; we can choose any number instead. There are two reasons I choose Radix-4: first, I aimed to deal with input size with maximum 4096, and a CUDA block can have 1024 threads, which means for each thread I need to deal with 4 elements. The previous Radix-2 method can only support maximum 2048 elements. Second, the time complexity would reduce from $O(Nlog_2N)$ to $O(Nlog_4N)$, which accelerates the computation.

The VkFFT paper [4] mentioned that we should run computations on registers, and use shared memory as communication buffer on GPUs. I implemented like this on Radix-2 and Radix-4 Stockham algorithms.

To support different input sizes, I implemented two kernels for the mix-Radix algorithm. For inputs that are powers of 4, I use `FFT_N_4` kernel, meaning the input is N and the radix is 4. The other kernel is `FFT_N_4_last_2`, which is for the inputs that are powers of 2 but not powers of 4. In this kernel, we perform one more radix-2 step at the end.

Results

My implementation is available on Github [5]. I implemented Radix-2 Stockham and mix-Radix Stockham with Radix 2 and 4. The best performance for Radix-2 occurs at size 128, which takes 120% of the baseline time. Although the worst case for mix-Radix still takes almost twice the baseline time, the performance at sizes 1024 and 2048 slightly outperforms the baseline.

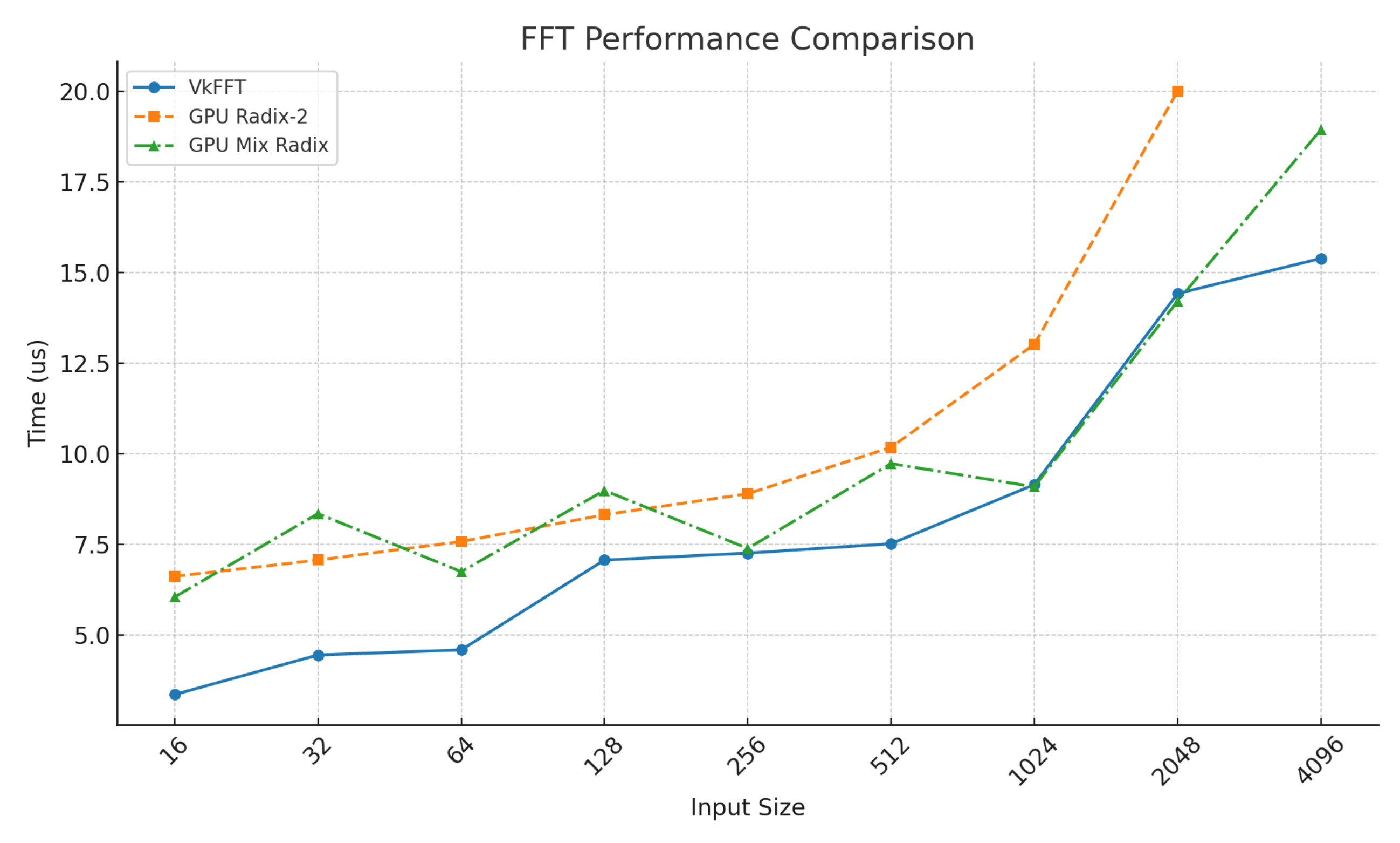

Figure 1: FFT Performance Comparison

References

- Wikipedia, "Discrete fourier transform," 2025, https://en.wikipedia.org/wiki/Discrete_Fourier_transform.
- M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
- Wikipedia, "Cooley-tukey fft algorithm," 2025, https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm.
- D. Tolmachev, "Vkfft: A performant, cross-platform and open-source gpu fft library," *IEEE Access*, vol. 11, pp. 12 039–12 056, 2023.
- C.-L. Tsai, "Fft: Fast fourier transform implementations on cpu and gpu," https://github.com/soulrrrrr/FFT, 2025.