Fast Fourier Transform on CPU and GPU

Chia-Lun Tsai
chialun2@illinois.edu
University of Illinois Urbana-Champaign, Urbana, Illinois, USA

1 Introduction

In this project, I wrote a FFT calculation library, targeting CPU and Nvidia GPU, and attempted to get similar performance on heavily optimized libraries such as FFTW [1] and VkFFT [8]. I started from naive implementation, and gradually speed up the program in multiple ways (eg. single thread CPU -> multi thread CPU -> GPU -> better algorithm, kernel fusion, memory coalescing, etc.) and show the speed up ratio comparing to naive implementation. For each step, I used profiling results to decide what should I improve. Currently, the performance on GPUs are usually memory-bound. FFT is one of the workload that requires extensive computation, and can use the computing power very well. Moreover, FFT is useful in multiple areas, especially scientific computing and signal processing. Last but not least, I am taking Parallel Programming course this semester, it is a great project for me to use the concepts and CUDA programming skills on building it. My implementation is available on Github [7].

2 FFT overview

2.1 Discrete Fourier Transform

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: [10]

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

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

2.2 Fast Fourier Transform

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 thoughts to solve discrete Fourier Transform.

2.2.1 Cooley-Tukey Algorithm. Assuming we have input length N, which is power of 2. And the DFT is defined by the formula:

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

where k is an integer ranging from 0 to $N-1$.

First we compute the DFTs of the even-indexed inputs and of the odd-indexed inputs, and then combines those two results to produce the DFT of the whole sequence. This idea can then be performed recursively to reduce the overall runtime to $O(NlogN)$.

We can separate the even and odd indexed terms as:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-2\pi i \frac{k}{N}(2m)} \quad + \quad \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-2\pi i \frac{k}{N}(2m+1)}$$

Let E_k denote the DFT of even-indexed terms, and O_k denote the DFT of odd-indexed terms. After some steps, then the final DFT is 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 $k = 0, 1, \ldots, N/2-1$. This method 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. [9]

One thing to notice is that to allow the output to be natural order, we need to do **bit reversal** for the input.

Consider the last stage of a radix-2 DIT FFT algorithm like the one presented above, where the output is written *in-place* over the input. When E_k and O_k are combined using a size-2 DFT, these two values are overwritten by the outputs. However, the two output values are placed in the first and second halves of the output array, corresponding to the *most significant bit* b_4 (assuming $N = 32$). In contrast, the inputs E_k and O_k are interleaved as even and odd elements, corresponding to the *least significant bit* b_0.

Thus, to ensure the output values are written to the correct locations, the index bits must be permuted such that b_0 takes the place of b_4, making the bit order $b_0 b_4 b_3 b_2 b_1$. In the next recursive stage, the remaining 4 bits become $b_1 b_4 b_3 b_2$, and so on.

If we include all recursive stages of a radix-2 DIT algorithm, we find that all bits in the index must be reversed. Therefore, we must pre-process the input (or equivalently, post-process the output) using a bit reversal permutation to obtain an in-order output.

2.2.2 Stockham Algorithm. The characteristic of Cooley-Tukey algorithm is the bit reverse part. We need to do bit reverse to sort the result of FFT in a natural order. This is an important point that is different from the Stockham algorithm. In the Stockham algorithm, even if there is no bit reverse, the result of FFT is sorted in a natural order. The method is for each stage, we put the output to the corresponding place of next stage input. This algorithm removes the overhead of bit reverse, which consumes about 15% of total computation time based on the profiling result. [4]

3 Implementation on CPU

3.1 Naive implementation

For naive implementation, I followed the formula of DFT and implemented an $\Theta(N^2)$ algorithm.

3.2 Cooley-Tukey algorithm

For the Cooley-Tukey algorithm, first I performed bit reversal on the input vector. Then I implemented the algorithm iteratively. In the first loop, we do size-2 FFT, then use previous results to perform

size-4 FFT, and so on. We group operations with the same twiddle factor `w` together to reduce the operations of multiplying `w`.

3.3 Stockham algorithm

For the Stockham algorithm, the implementation became more difficult. After my observation of the permutation rule, I decided to implement the algorithm as follows. First, the outer loop loops for `stride`, which means the distance between the FFT pair. For example, for $N = 8$ and stride = 4 (first iteration), we are going to perform size-2 FFT on $(0, 4)$, $(1, 5)$, $(2, 6)$, $(3, 7)$. Inside the outer loop we define `block_size` as the size of elements using the same twiddle factor `w`, which avoids recomputing `w` at each step. 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. For the permutation, the local size-2 FFT is always output at indices block_idx·stride+block_offset and block_idx·stride+block_offset+$\frac{N}{2}$. Besides of iterative implementation, I implemented a recursive version, where the performance is similar. I also used a self-defined complex type instead of `std::complex` to remove the multiplication overhead of `std::complex` [5].

4 Implementation on GPU

For GPU, I will focus on the FFT size of power of 2 and smaller than 4096, which can be fit in a CUDA block. To support multiple input sizes, we need to check the input size using conditional statements. However, on GPUs, if branch divergence occurs within a warp, both branches are executed and the thread will keep the correct result. This adds overhead to our program. Popular GPU FFT libraries such as VkFFT [8] uses **plan** to determine the best configuration for the current input size. This plan generates the optimized kernel code and then executes it. In my implementation, since I am currently working with inputs of powers of 2, I will not use this plan method, but use two kernels instead. I will explain the reason behind this design.

4.1 Cooley-Tukey Algorithm

As we already know that the naive implementation is slow, we start with Cooley-Tukey Algorithm on GPU. The implementation is quite like on CPU, the only difference is that each CUDA thread only deal with a set of operations (size-2). Since the performance is too slow, I directly move on to Stockham algorithm instead.

4.2 Stockham Algorithm

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

There is no limitation that we can only choose 2 as Radix; we can choose any number instead. This is the crux of Cooley-Tukey divide and conquer method, and it applies to Stockham as well. There are two reasons I choose to implement Radix-4. First, I am aiming 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.

Although I initially thought that Radix-4 would be difficult to implement, after studying some materials [3] [4], I found that the code logic is similar to Radix-2, so I was able to implement it with little effort. The main difference is that, in each step, we need to multiply three twiddle factors instead of one.

The library VkFFT [6] suggested that computations should be performed using registers, with shared memory used as a communication buffer on GPUs. I followed this approach in my implementations of the Radix-2 and Radix-4 Stockham algorithms.

4.3 Mix-Radix Algorithm

To serve different length of inputs, we need to use mix-radix algorithm. The basic idea is that for each step, we can choose the radix we want. For example, for size-30 FFT, we can choose our radix such as $(2, 3, 5)$, $(3, 5, 2)$, etc.

In our case, I implemented two kernels for the mix-radix algorithm. For inputs 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 inputs are powers of 2 but not powers of 4. In this kernel, we perform one more radix-2 step at the last step.

5 Evaluation on CPU

5.1 Environment and Metrics

We conduct our experiments on the Porter machine, which is equipped with two Haswell-E 8-core Xeon processors and 64 GiB of DDR4 RAM, and two Nvidia Titan X GPU. All performance measurements are based on single-core execution.

Each program is run 10 times, and the geometric mean of each metric is reported as the final result. To estimate FLOPS, we assume that an N-point FFT requires approximately $5N \log_2 N$ floating-point operations. The achieved FLOPS is then computed by dividing this estimate by the measured runtime.

5.2 Baseline: FFTW [1]

FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, i.e. the discrete cosine/sine transforms or DCT/DST). It is one of the fastest FFT libraries running on CPU.

We use powers of two from 16 (2^4) to 262144 (2^{18}) as the test case input sizes. The inputs are all float type (32 bits).

To measure the execution time of the FFT, we use the C++ `std::chrono::high_resolution_clock` API. Specifically, the timing is measured around the `fftwf_execute` function call as follows:

```
auto start = std::chrono::high_resolution_clock::now();
fftwf_execute(plan);
auto end = std::chrono::high_resolution_clock::now();
```

The elapsed time is then computed using the duration between `start` and `end`. For FLOPS, Since FFTW uses an optimized implementation of the Cooley-Tukey algorithm, we use $5 \cdot N \cdot \log_2 N$ as a rough estimate of the total operation count to normalize performance. The result is shown in Table 1.

Table 1: FFTW Performance

N	Time (s)	MFLOPS
16	0.000001	478.32
32	0.000001	1151.29
64	0.000001	2033.47
128	0.000001	3135.52
256	0.000002	4329.43
512	0.000007	3411.93
1024	0.000015	3392.19
2048	0.000019	6039.17
4096	0.000025	9686.35
8192	0.000063	8424.44
16384	0.000109	10527.41
32768	0.000237	10350.11
65536	0.000521	10064.30
131072	0.001090	10224.87
262144	0.002112	11168.87

5.3 My Implementation

For my $O(NlogN)$ CPU implementation, I started with Cooley-Tukey algorithm. For the $N = 262144$ case, the performance is only $\sim$ 10% of the FFTW library. Then I started to apply optimizations on the code. First I investigate the assembly code of FFTW and my code, found that FFTW uses lots of avx2 commands, but my code doesn't. By adding `-march=native` flag, the performance become 1431 MFLOPS ($\sim$ 13% FFTW). Then I use `perf` to find the bottleneck, and find `bit_reverse` takes about 15% of time, so I started to implement the Stockham algorithm. My first Stockham algorithm achieves 3689.41 MFLOPS ($\sim$ 33% FFTW). After moving to Stockham algorithm, I used `perf` again, and discovered that complex multiplication spent half of the time. Hence I implemented a simple complex type only for addition, subtraction, and multiplication. This is my final implementation and it achieves 4681.23 MFLOPS ($\sim$ 42% FFTW). The full result is shown in Table 2. After discussion with professor, I decided to move on to GPUs.

Table 2: Custom Stockham Performance

N	Time (s)	MFLOPS
16	0.000014	22.10
32	0.000014	57.45
64	0.000015	128.09
128	0.000022	200.73
256	0.000026	393.11
512	0.000029	797.76
1024	0.000033	1544.98
2048	0.000047	2406.33
4096	0.000086	2868.44
8192	0.000164	3254.14
16384	0.000318	3609.30
32768	0.000678	3626.12
65536	0.001347	3890.84
131072	0.002581	4316.80
262144	0.005040	4681.23

6 Evaluation on GPU

6.1 Environment and Metrics

We conducted our experiments on the same Porter machine, which is equipped with two Haswell-E 8-core Xeon processors and 64 GiB of DDR4 RAM, and two Nvidia Titan X GPUs. The work is performed on one GPU.

We use the profiling result from `ncu` as the time metric. The `ncu-rep` file contains the kernel execution time.

6.2 Baseline: VkFFT [8]

VkFFT is an efficient GPU-accelerated multidimensional Fast Fourier Transform library for Vulkan/CUDA/HIP/OpenCL/Level Zero/Metal projects. VkFFT aims to provide the community with an open-source alternative to Nvidia's cuFFT library while achieving better performance. VkFFT is written in C language and supports Vulkan, CUDA, HIP, OpenCL, Level Zero and Metal as backends. It is one of the fastest FFT GPU libraries.

When performing FFT using VkFFT, the code generation algorithm defines the structure of the generated kernels, algorithms used, number of threads, registers and shared memory, then generate the kernel code. We can see in Table 3, even though we are performing in a CUDA block, the threads per block are different, which affects elements per thread and registers used.

Table 3: VkFFT Performance

Size	Threads	Elements/Thread	Registers	Time (us)
16	4	4	28	3.36
32	4	8	48	4.45
64	8	8	48	4.59
128	16	8	56	7.07
256	32	8	56	7.26
512	64	8	48	7.52
1024	128	8	56	9.15
2048	128	16	80	14.42
4096	256	16	80	15.39

6.3 My Implementation

First I started with Cooley-Tukey algorithm. Observing that the performance is too slow due to the overhead of bit reversal, I decided to switch to the Stockham algorithm. Table 4 shows the performance of Radix-2 Stockham algorithm. Since a CUDA block can only contain 1024 threads, the maximum size is 2048. The best performance occurs at size 128, which takes 120% of the baseline time, while the worst is at size 16, taking 212% of the baseline time.

To improve performance and support up to 4096 threads, I implemented a mixed-radix Stockham algorithm using Radix-2 and Radix-4. When the input size is a power of 4, the kernel uses the Radix-4 algorithm exclusively. Otherwise, it applies Radix-4 for all but the final stage, which uses Radix-2. Table 5 shows the performance results.

Figure 1 presents a comparison with the baseline. Although the worst case at size 32 still takes nearly twice the baseline time, the performance at sizes 1024 and 2048 slightly surpasses the baseline.

The register usage varies because some configurations require a Radix-2 stage at the end, while others do not.

Table 4: FFT GPU Radix-2 Performance

Size	Threads	Elements/Thread	Registers	Time (us)
16	8	2	32	6.62
32	16	2	32	7.07
64	32	2	32	7.58
128	64	2	32	8.32
256	128	2	32	8.90
512	256	2	32	10.18
1024	512	2	32	13.02
2048	1024	2	32	20.00
4096	N/A	N/A	N/A	N/A

Table 5: FFT GPU Mix Radix Performance

Size	Threads	Elements/Thread	Registers	Time (us)
16	4	4	46	6.05
32	8	4	40	8.35
64	16	4	46	6.75
128	32	4	40	8.99
256	64	4	46	7.39
512	128	4	40	9.73
1024	256	4	46	9.09
2048	512	4	40	14.21
4096	1024	4	46	18.94

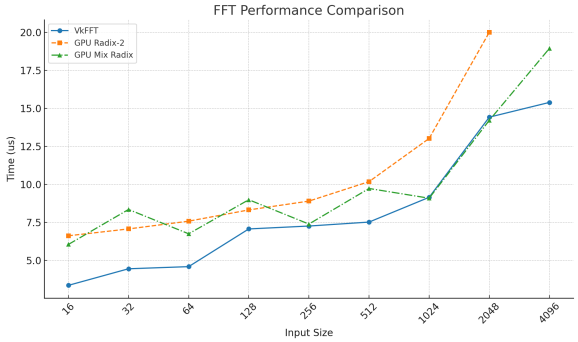

Figure 1: FFT Performance Comparison

7 Future Work on GPU

- For size 4096, the ncu profiling report indicates excessive register usage, which limits performance. While VkFFT uses only 28 registers for 4 elements per thread, our implementation uses 40–46. Reducing register usage is a promising direction for optimization.
- According to the VkFFT paper [6], the radix kernel supports sizes 4, 6, 8, 9, 10, 12, 14, 15, 16, and 32. Exploring larger radix sizes can improve hardware utilization and enhance performance.
- Sizes larger than 4096 are also worth investigating. In these cases, optimizing global memory access patterns becomes increasingly critical.
- Shared memory banking can increase the shared memory throughput. In VkFFT implementation, it almost has zero bank conflict; I tried to implement it but it seems the overhead of indexing is longer than the reduced time, so I didn't implement it in my final code.

8 Conclusion

Through this work, I applied the knowledge gained from the Parallel Programming course to this project, achieving comparable results in specific cases to one of the most optimized GPU FFT libraries. This also highlights that there is still much for me to explore to reveal the potential of GPUs.

References

[1] FFTW. 2025. FFTW. https://www.fftw.org.

[2] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[3] NTU. 2025. NTU FFT. https://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html.

[4] OTFFT. 2025. Introduction to the Stockham FFT. http://wwwa.pikara.ne.jp/okojisan/otfft-en/stockham1.html.

[5] Reddit. 2025. Why is C++ std::complex significant slower in some cases than user-defined complex? https://www.reddit.com/r/Compilers/comments/14wtxss/why_is_c_stdcomplex_significant_slower_in_some/.

[6] Dmitrii Tolmachev. 2023. VkFFT: A Performant, Cross-Platform and Open-Source GPU FFT Library. *IEEE Access* 11 (2023), 12039–12056. https://doi.org/10.1109/ACCESS.2023.3242240

[7] Chia-Lun Tsai. 2025. FFT: Fast Fourier Transform Implementations on CPU and GPU. https://github.com/soulrrrrr/FFT.

[8] VkFFT. 2025. VkFFT. https://github.com/DTolm/VkFFT.

[9] Wikipedia. 2025. Cooley-Tukey FFT Algorithm. https://en.wikipedia.org/wiki/CooleyâĂŞTukey_FFT_algorithm.

[10] Wikipedia. 2025. Discrete Fourier Transform. https://en.wikipedia.org/wiki/Discrete_Fourier_transform.