Spring 2019: Advanced Topics in Numerical Analysis High Performance Computing Implementation of Fast Fourier Transform (FFT)

Yongyan Rao, yr780@nyu.edu
(Dated: May 14, 2019)

I. 0. INTRODUCTION

The Fourier transform of a function $f : \mathbb{R} \to \mathbb{C}$ is defined as

$$\hat{f}(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x k} dx. \tag{1}$$

And the inverse Fourier transform is defined as

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(k) e^{2\pi i x k} dk. \tag{2}$$

The discrete Fourier transform (DFT) of a sequence of N complex numbers $\{x_0, x_1, \cdots, x_{N-1}\}$ is defined as

$$y_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, k = 0, 1, \cdots, N-1. \tag{3}$$

From the definition, we can find that a straightforward implementation of DFT is of complexity of $\mathcal{O}(N^2)$. By observing and preserving the underlying symmetry of DFT, a fast Fourier transform (FFT) algorithm is able to reduce the complexity to $\mathcal{O}(N \log N)$, without losing any accuracy.

The most commonly used FFT algorithm is the Cooley-Tukey algorithm, which uses the divide and conquer paradigm that recursively factorizes N and applies the transform on a lower scale. To simplify the discussion, we only consider the radix-2 FFT, which means N is a power of 2, i.e., $N = 2^l, l \in \mathbb{N}$. The radix-2 FFT can be derived as follows.

$$\begin{aligned}
y_k \equiv y_{k,0,1} &= \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} = \sum_{\text{even } n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} + \sum_{\text{odd } n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-\frac{2\pi i}{N} k(2n)} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-\frac{2\pi i}{N} k(2n+1)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-\frac{2\pi i}{N/2} kn} + e^{-\frac{2\pi i}{N} k} \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-\frac{2\pi i}{N/2} kn} \\
&= y_{k,0,2} + e^{-\frac{2\pi i}{N} k} y_{k,1,2},
\end{aligned}$$

where $y_{k,i,j}$ represents the k-th term of the transform of x_n, using summation beginning from i-th term with stride j. The recurrence relation of y can be written as

$$\begin{aligned}
y_{k,0,1} &= y_{k,0,2} + e^{-\frac{2\pi i}{N} k} y_{k,1,2} \\
&= y_{k,0,4} + e^{-\frac{2\pi i}{N} k} (y_{k,1,4} + y_{k,2,4}) + e^{-2\frac{2\pi i}{N} k} y_{k,3,4} = \cdots .
\end{aligned}$$

Therefore, it is straightforward to implement the FFT algorithm in terms of recursion. However, since it is easier to parallelize iteration than recursion in OpenMP's paradigm, we use the iterative Cooley-Tukey algorithm in the discussion as follows.

Listing 1: Iterative Cooley-Tukey algorithm for FFT

```
algorithm fft
  //input: an array x of complex numbers of length $n$, with $n$ is a power of 2.
  //output: an array y of complex numbers of length $n$, which is the FFT of x.

```

```
5     y = bit-reverse(x);
6
7     for(m = 2; m <= n; m *= 2)
8       theta = 2 * pi/m;
9       omega_m = exp(i * theta); //Euler's formula exp(i * theta) = cos(theta) + i * sin(theta)
10      for(k = 0; k < n; k += m)
11        omega = 1;
12        for(j = 0; j < m/2; ++j)
13          t = omega * y[k + j + m/2];
14          u = y[k + j];
15          y[k + j] = u + t;
16          y[k + j + m/2] = u - t;
17          omega = omega * omega_m;
18
19    return y;
```

A. Note on the Bit Reverse function

In Listing 1, we still need to further discuss the implementation details about the bit reverse function on line 5. The bit reverse is defined as reversing the bits in the binary representation of an unsigned integer. Therefore, after the bit reverse, the most significant bit becomes the least significant bit, and vice versa, the least significant bit becomes the most significant bit. For example, the bit reserve of an 8-bit unsigned integer $00000100_2 (= 4_{10})$ is $00100000_2 (= 32_{10})$. It is clear that for a n-bit unsigned integer x, calculating its bit reverse is of complex of $\log n$. If the number n is fixed in the problem, the bit reverse function can be implemented as a lookup table. Otherwise, it needs to be implemented explicitly in the program. It is good to observe that the bit reversing of the sequence $0, 1, \cdots, 2^n - 1$ is a permutation of the sequence, which implies that the bit reverse function can be fully parallelized. The coded bit reverse function is modified from [1].

The rest of the paper is organized as follows. Section II discusses the sequential implementation of the FFT. Section III discusses the multithreading solution implemented using OpenMP. Section IV presents a naive cuda implementation of the FFT. And Section V concludes the paper by summarizing the findings of the study.

II. 1. SEQUENTIAL IMPLEMENTATION OF FFT

We first implemented the sequential version of FFT, with $\mathcal{O}(n \log n)$ complexity. There are several points worth noticing as follows.

1. A fast implementation of sine and cosine functions is used, which covers the domain of $[-\pi, \pi]$. It is sufficient in terms of the use of FFT.

2. When the problem size is relatively large, there exists numerical discrepancy between the results of the implementation and the GNU Scientific Library (GSL). For example, when the problem size is 67,108,864, there are 48 such instances that the relative error of either the real or the imaginary part is greater than 10^{-3}, and it always happens at the points where the absolute value of the result is small. It is very rare when the problem size is less than this scale. Furthermore, using sine and cosine functions from `C` language's `math.c` library does not remedy the discrepancy significantly.

III. 2. OPENMP IMPLEMENTATION OF FFT

Although the algorithm above has been adapted to the form of iteration, it still keeps the trace of recursion, which means not all the iterations are independent. It is clear that the bit-reverse-copy function, line 5, has independent iterations, and the loop on line 10 has independent iterations. We can use OpenMP to parallel these portions of FFT. The program was compiled and run on cuda2.cims, which has two (2) 20-core CPUs. Since the OpenMP code does not explicitly indicate the CPU usage, the program would run on one of the CPUs, which means it could utilize 20 cores. The followings are the time measures, FIG. 1, and the speedup of the OpenMP implementation, FIG. 2.

The running time of the OpenMP implementation becomes shorter than both sequential and GSL implementations when the problem size is over the order of 10^5. This is because OpenMP needs to spawn and join threads, which involves system calls, and problem with relatively small size is not able to compensate such overhead. The speedup measured is never greater than 10, given the program is run on a 20-core machine. The phenomenon is understandable in terms of Amdahl's law. The outermost and innermost loops, line 7 and line 12, respectively, in the algorithm are iteration dependent, which cannot be parallelized. They serialize a great portion of the program.

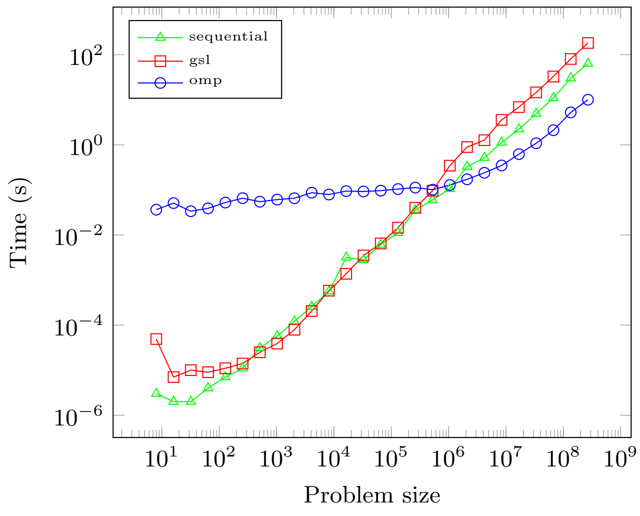

FIG. 1: Running time of FFT implementations

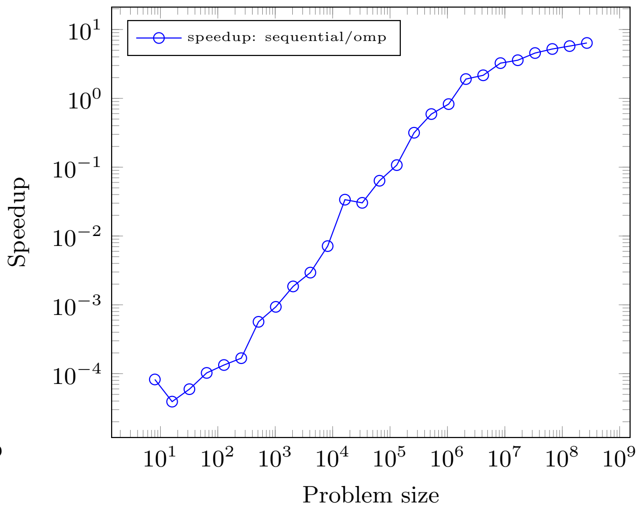

FIG. 2: Speedup of OpenMP FFT

One could also study the scalability of the OpenMP implementation. A program is weakly scalable, if when the number of processes/threads increases x times, the program completes an x times larger problem within the same amount of time. Therefore, from FIG. 3, we can conclude that the OpenMP implementation of FFT is not weakly scalable.

A program is strongly scalable, if when the number of processes/threads increases x times, the program completes a same-size problem x faster, which means the speedup of the program is proportional to the number of processes/threads. Therefore, from FIG. 4, we can conclude that the OpenMP implementation of FFT is not strongly scalable.

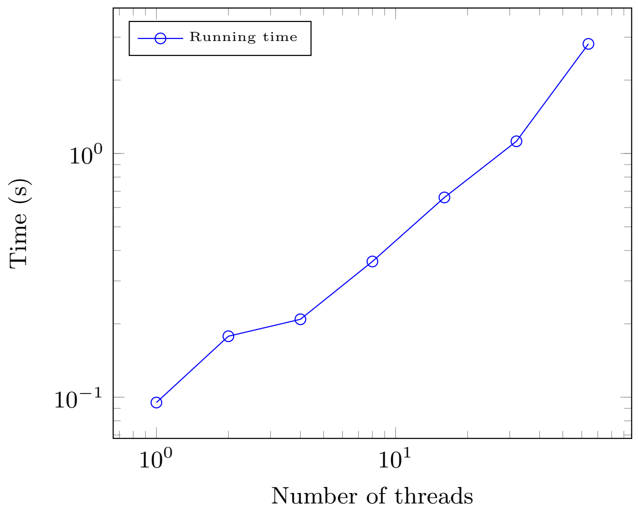

FIG. 3: OpenMP FFT weak scaling study

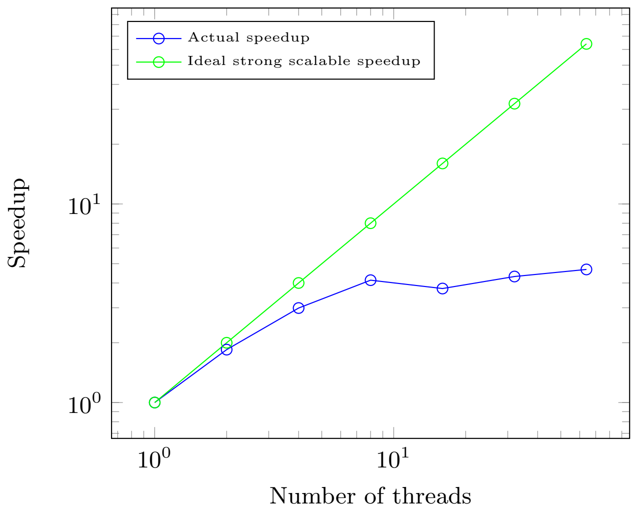

FIG. 4: OpenMP FFT strong scaling study

IV. 3. CUDA IMPLEMENTATION OF FFT

A straightforward implementation of cuda FFT is coded, and tested on cuda2.cims. There are several points worth mentioning as follows.

1. In the implementation, memory copy happens twice. One is copying input from host to device before the first kernel launch, and the other is copying output from device to host after the last kernel run.

2. Due to the distribution of the parallelizable regions, the bit reverse function in line 5 and the loop in line 10, two (2) kernels are implemented, and two (2) kernel launches happen in the program execution. The first kernel is used as the bit reverse function, which only needs to be launched once, i.e., the first kernel launch. The second kernel is used for the parallel computing of the loop of line 10.

3. Because the line 10 loop is nested in the outermost loop of line 7, the second kernel needs to be launched multiple times, $\log n$, where n is the problem size.

4. The program was complied and run on cuda2.cims, and we noticed that to run properly, the greatest problem size the program accepted was 268,435,456. There exist four (4) `double` arrays in device memory during a kernel run, with two (2) input arrays and two (2) output arrays. Two components for either the input or the output represent the real part and the imaginary part of a complex number, respectively. Therefore, for a problem of size n, it needs $8 \times 4 \times n$ byte device memory, which means a problem size 268,435,456 corresponds to 8.6×10^9 byte device memory. The device memory was measured during the program run, which showed that the device's total memory was 11.5×10^9 byte and free memory was 9.2×10^9 byte. Therefore, the phenomenon was due to device memory limit.

5. The performance of the cuda implementation was measured and presented in FIG. 5 and FIG. 6. We could notice that from the measurement that the cuda implementation always took longer time than the original sequential implementation. The best relative performance of the cuda implementation was only about $\frac{3}{4}$ of the performance of the sequential implementation.

6. It is not a very great surprise for the cuda implementation to have such poor performance. As discussed above, the iterative Cooley-Tukey algorithm is not fully parallelizable. This fact has the following implications.
   (a) The second kernel needs to be launched multiple times, $\log n$. Furthermore, in the first iterations, when m is small, the kernel has less operations to execute in each launch, which makes it more memory bounded.
   (b) The memory access pattern for the loop in line 10 makes it difficult to fully exploit the coalesced memory access. This is because when m is large, the threads next to each other are actually trying to access memory locations that are with m locations apart from each other, which means combining multiple memory accesses into a single transaction is impossible in the situation.
   (c) The naive cuda implementation of FFT does not exploit the use of shared memory. And instead, data is directly read from and written to device main memory. To make use of the lower latency shared memory, the original Cooley-Tukey algorithm needs to be modified to suit the GPU architecture.

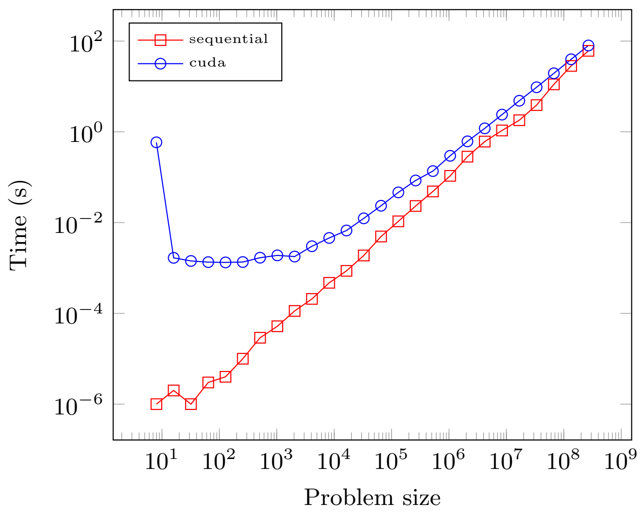

FIG. 5: Running time of FFT implementations

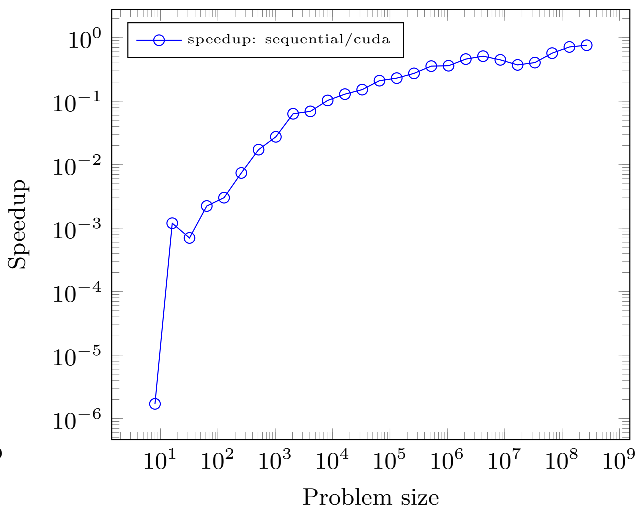

FIG. 6: Speedup of cuda FFT

V. 4. CONCLUSION

From the experimentations conducted above one could conclude that

1. The self-implemented sequential FFT algorithm generally has very similar performance to the GSL FFT function, and for large problem size, the performance of self-implementation even exceeds GSL.
2. The OpenMP implementation of FFT shows significant advantage when problem size is greater than 10^5. However, because of the existence of the sequential portion in the algorithm, the multithreaded FFT algorithm is not either weakly scalable or strongly scalable.
3. The naive cuda implementation of FFT does not provide reasonable speedup due to multiple factors, including device memory limit, and inefficient device memory access. cuda provides optimized cuFFT API, which is the cuda Fast Fourier Transform library.

[1] Bit reverse, `http://www.katjaas.nl/bitreversal/bitreversal.html`.
[2] Jakub Kurzak, David A. Bader, Jack Dongarra, *Scientific Computing with Multicore and Accelerators*, Chapman & Hall/CRC, 2010.
[3] FFT of GNU GSL, `https://www.gnu.org/software/gsl/doc/html/fft.html`.