Parallel FFT

Pinnong Li; Yitao Meng

pinnongl; yitaom

The date: May 5, 2022

1 SUMMARY

We implemented paralleled one Dimensional Fast Fourier transform algorithm on both GPU and multi-core CPU platforms, and perform a detailed analysis of both systems' performance characteristics.

2 BACKGROUND

2.1 Motivation

A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. [4]

Fast Fourier transforms are widely used for applications in engineering, music, science, and mathematics. [4] One example is big integer multiplication: with FFT, the time complexity of big integer multiplication can be reduced from O(n^2) to O(nlogn).

Moreover, FFT can be parallelized, which can help accelerate big integer multiplication even more. So we decided to explore different parallel implementations of FFT, including SIMD, multithreading and CUDA.

2.2 Algorithm Specification

2.2.1 Input and Output

1. Input: One dimensional time signal, in a form of a sequence of complex numbers, the complex number indicates the signal value and the index indicates time. Or in other words, an array

of complex number. Required by the algorithm, the length must be power of 2, if it's not, then the user should extend it and pad with zeros.

2. Output: One dimensional frequency domain representation transformed by the algorithm, also as a sequence of complex numbers, which has the same length as input. The index indicates the frequency and value indicates the magnitude.

The input and output are both just an array of complex numbers with a power of 2 length if ignoring the physical meaning of them.

2.2.2 High Level Structure

The whole algorithm could be divided into 3 steps:

1. SHUFFLE: an O(n) operation that shuffles the input array.

2. CALCULATION: O(nlogn) time complexity. Perform $log(n)$ iterations on the shuffled input data to transform it into the output. In each iteration, every element needs to be updated based on itself and another element and one particular complex weight, which is calculated based on its' index and current iteration number. Specifically, in the i-th iteration, the j-th element e[j] and (j + 2^i)-th element e[j + 2^i], where $j \bmod 2^{i+1} < 2^i$ is updated as Figure 1 shows. Overall, the data dependency is as Figure 2 shows.

```
// in i-th iteration
x = e[j];
y = e[j + (1 << i)] * w(i, j mod (1 << i));
// w(i, k) is a weight, which is a complex number
// and can be computed from w(i, k-1) in O(1) time
e[j] = x + y;
e[j + (1 << i)] = x - y;
```

Figure 1: pseudo calculation code

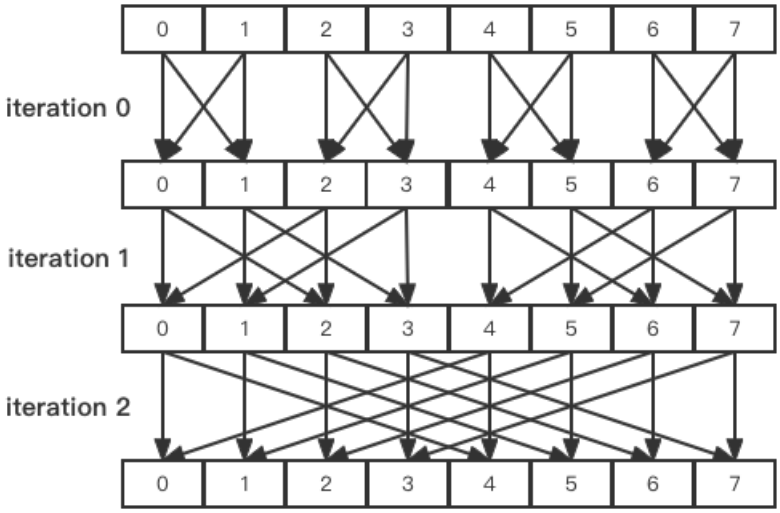

Figure 2: data dependency of data size = 8

3. NORMALIZATION: this may be needed if the transformation is in a reverse direction, takes O(n) to complete. Normalizes the final output.

The most computationally expensive part is the second step: CALCULATION, which is exactly our parallel target. In the following sections, we will introduce our detailed design and implementation of parallel CALCULATION.

3 APPROACH

3.1 Sequential

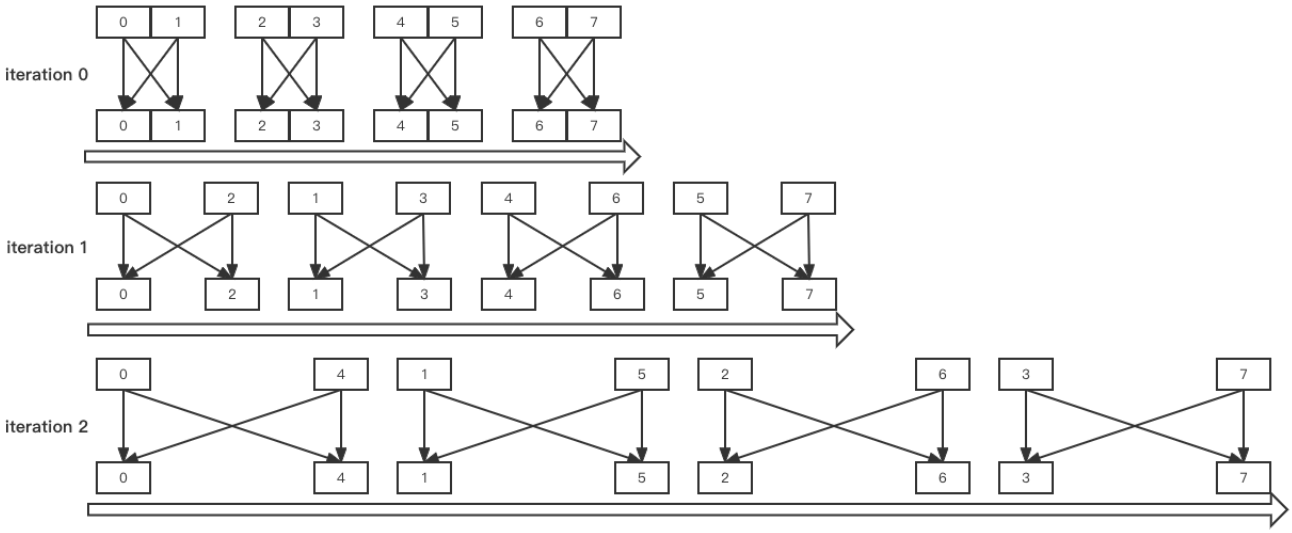

Figure 3: sequential computation of data size = 8

As the Figure 3 demonstrates, within one iteration, sequential implementation executes the updating computation for element pairs from low index to high index and updates the weight w at the same time. This implementation requires no extra space and computation, the executing order maintains a spatial locality to utilize cache. The performance of our sequential implementation could reach to about 2-3 times slower than a popular c++ FFT library, which is called FFTW.

3.2 Target Machine and Parallel Preparation

We parallelized FFT through three main approaches, and targeted multi-core single machine (used GHC machines for testing). The CALCULATION portion(as shown in section 2.2.2) in FFT is highly parallelizable. The modification of input elements can be handled by different CPU threads or Kernel threads. In SIMD, multiple adjacent input elements can be calculated as a group because the operations have limited dependency.

In order to perform parallelization better, we made modification on top of our sequential code. These modifications apply to some or all our parallelization methods in the following subsections.

1. Within the CALCULATION stage, for each input index we need access to its weight parameter. In sequential implementation, the weight parameter is computed from previous weight value, which forms a dependency that cannot be parallelized. In order to get rid of the dependency between weight parameters. we pre-computed all weights in O(n) time and stored in an array, called ws.

2. Also within the CALCULATION stage, we originally have three layers of for loops – iterations, blocks, and index within blocks.

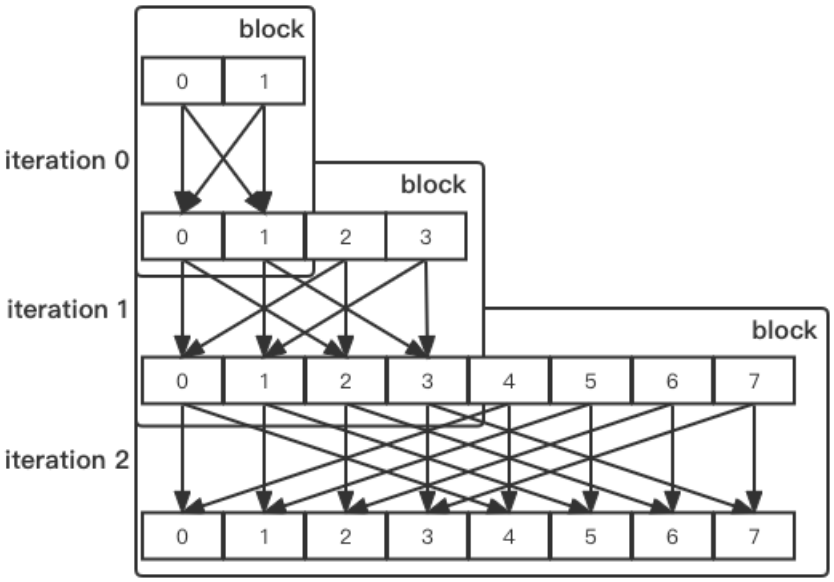

Figure 4: block in different iterations

This is easier to understand, but may produce overhead for thread level parallelism. So we combined the later two for loops. From this point on we refer to the second for loop as the inner for loop.

3. Our sequential algorithm utilizes C++'s complex library to perform CALCULATION stage. However, SIMD and CUDA weren't able to adjust to complex library. So we defined our own struct of complex numbers, and overwrote the addition/multiplication operators.

3.3 CPU Parallel – Thread

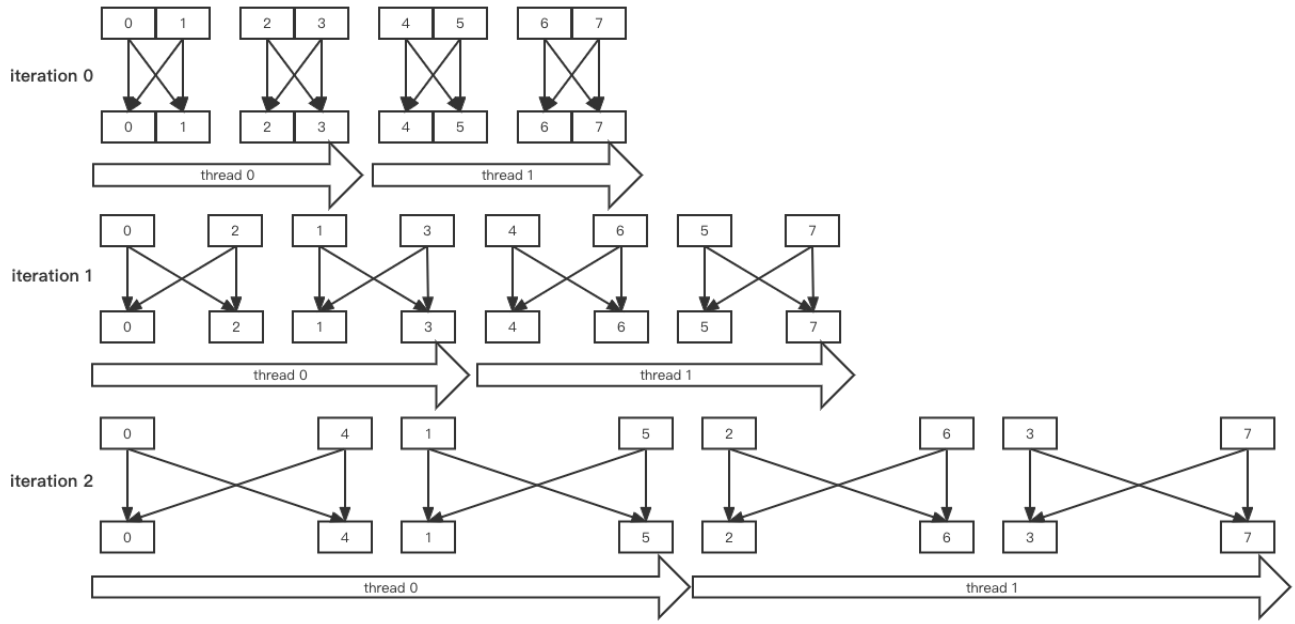

Figure 5: multi thread computation of data size = 8

Thread level parallelization is our first attempt to parallelize FFT algorithm. We tried openMP and implementing our own pthread version. Here are a few design details.

1. Workload distribution: We assign each thread a equally sized index continuous list of element pairs to balance the workload and maintain locality within a thread.

2. openMP: we choose to use static assignment for the inner for loop of CALCULATION stage. This is because the number of operation is similar for each index, and static assignment should be able to generate near balanced workload distribution. Since dynamic has a higher overhead, static assignment is sufficient.

3. pthread implementation: we generated a function that wraps around our CALCULATION stage, and added pthread_barrier_wait to synchronize between iterations.

3.4 CPU Parallel – SIMD

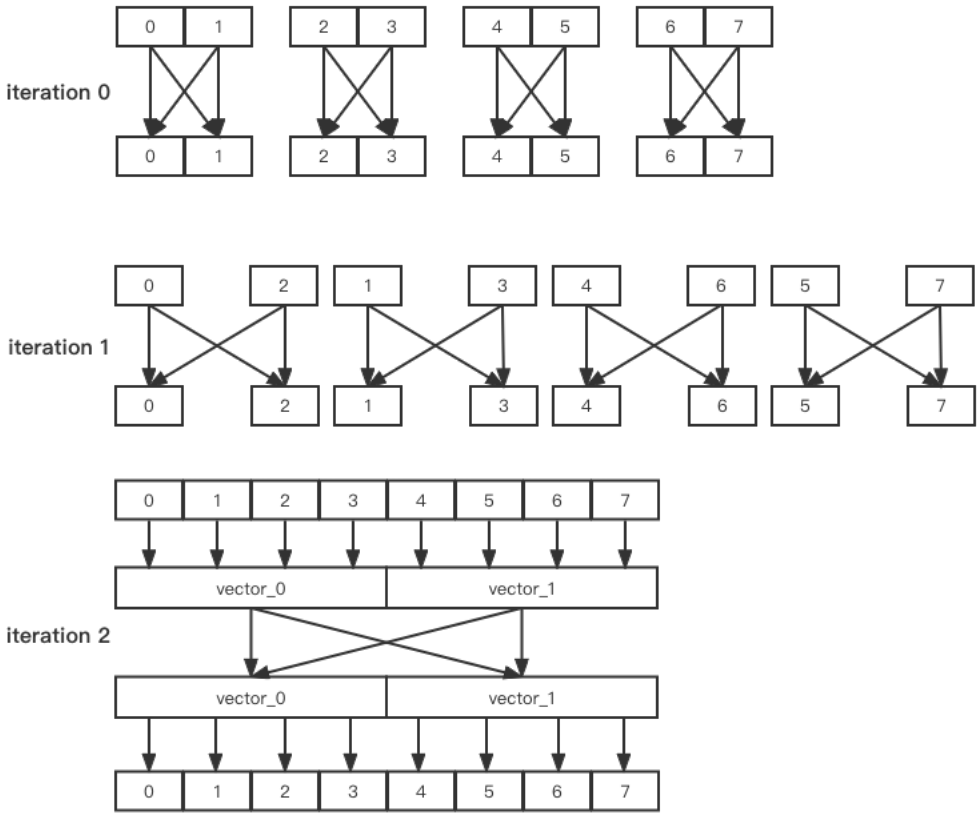

Figure 6: SIMD computation of data size = 8

In order to perform SIMD, we utilized C++ AVX intrinsics, and organize 8 floats into a _mm256 element. We reorganized CALCULATION's code structure for SIMD as following.

1. For the first 3 iterations with block-size smaller than 16, we leave the structure untouched. This is because in the first 3 iterations, the elements within one SIMD vector are executing different instructions, so SIMD cannot help accelerate the computation in this case. And if we shuffle the data to make the instructions within one vector the same, the overhead would however decrease the overall performance.

2. We then create an aligned _mm256 array *out_simd* in heap memory, and load the modified input array into our SIMD array. We also created an aligned _mm256 array *ws_simd* for weight parameters, but since each loop has different access pattern of weight parameters, we need to refill *ws_simd* before each inner for loop.

3. For all following iterations with block-size larger or equal to 16, we first perform an O(block-size) operation to fill *ws_simd*, and then use *out_simd* to perform an O(n) calculation.

3.5 CPU Parallel – SIMD combined with Thread

After we finished testing both SIMD and multi-thread, we applied openMP to our SIMD operation to obtain a better CPU parallelization speedup.

3.6 GPU Parallel

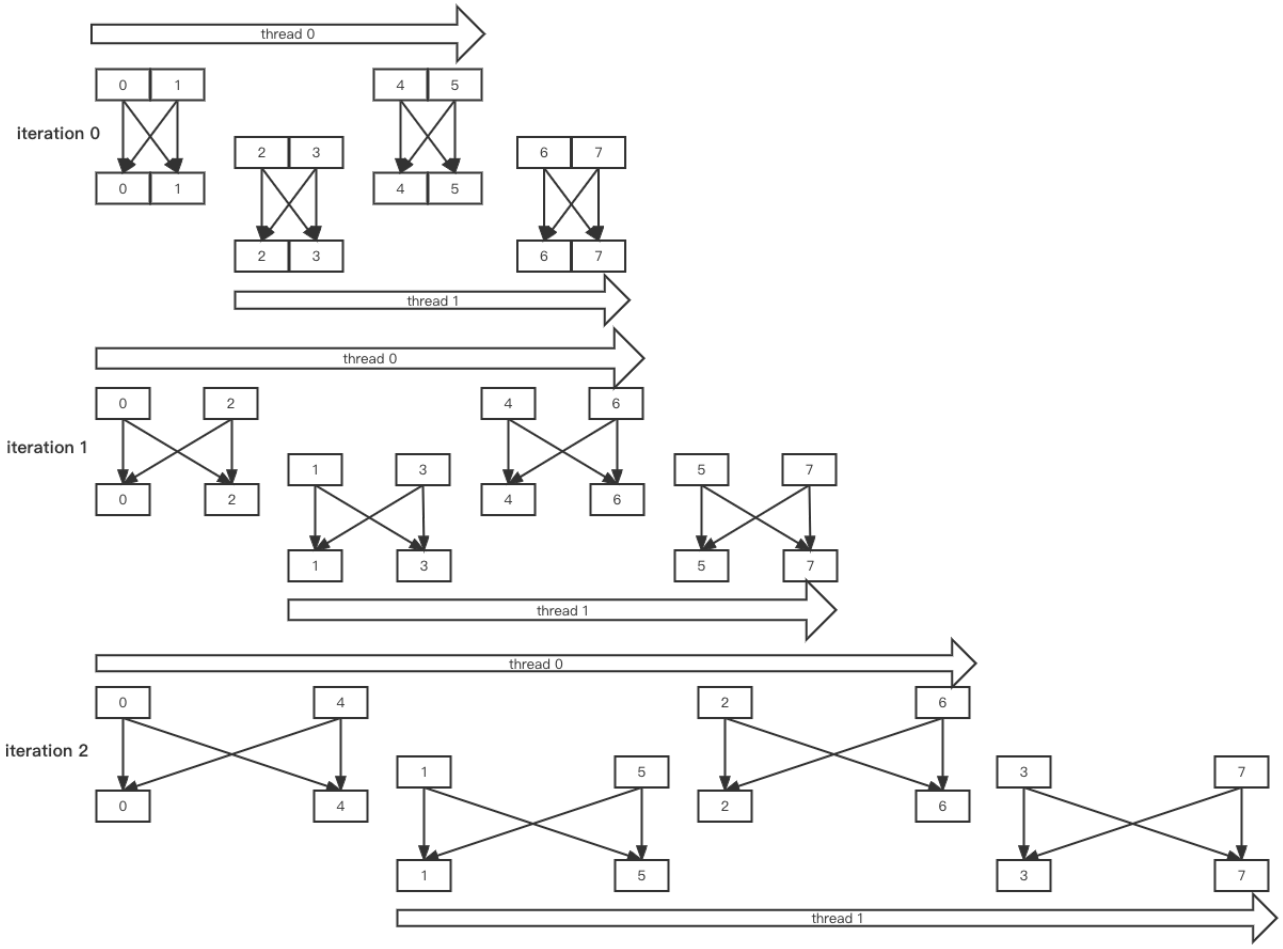

Figure 7: GPU computation of data size = 8

In our GPU parallelization, we parallelize multiple FFT computation, and let each block handle one FFT CALCULATION stage. In our implementation, we assume that different input arrays

for different FFTs are of the same length n. So we truncate all input arrays input one large input array. The following is the implementation details.

1. Within one block, each thread updates one pair of elements in a continuous block of the input data in the current iteration and then update the next continuous block until the end. This is different from the workload assignment in CPU parallel version, we use this interleaved assignment to maintain spatial locality in order to utilize the shared memory of a thread block.

2. At the end of each iteration, a thread level synchronization is applied.

We choose to assign each FFT to a single thread block mainly because we want to reduce the overhead of synchronization. An alternative will be launching multiple blocks for each FFT, synchronize blocks for each iteration, and repeat. Our approach requires less overhead and achieves better overall performance in comparison.

3.7 Code Reference

Our parallalization are done without existing code. Our sequential version's SHUFFLE stage includes 4 lines existing code from [1]. We commented in github to show where these 4 lines are used. In order to understand the FFT algorithm, we used the following resources [2].

4 RESULTS

We separated the preparation work like memory allocation and main FFT execution, and only take main FFT execution into consideration when evaluating performance. All our algorithms are written in C++/CUDA.

4.1 Base line

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).[3] We used FFTW v3.3.10 sequential version as our baseline.

4.2 Analysis

4.2.1 Sequential

input size	fftw(ns)	o3-sequential(ns)
2^14	99209	869359
2^16	3537983	3451621
2^18	2666547	8797387
2^20	23142537	46429074
2^22	123588297	210302527

Figure 8: sequential with o3 compared to baseline

With -O3 enabled, our sequential version reaches near baseline performance (within 2-3 times baseline) as the size of input increases.

4.2.2 CPU Parallel – Thread

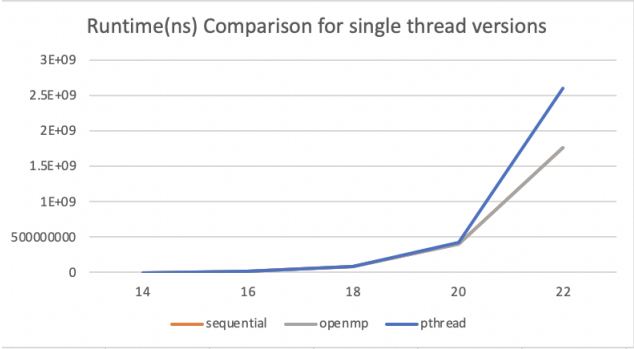

Figure 9: runtime(ns) comparison for sequential/openmp(1)/pthread(1)

In the graph above, we show the runtime comparison of sequential version / openMP with one thread / single pthread for different input size (2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}). All three versions show similar runtime.

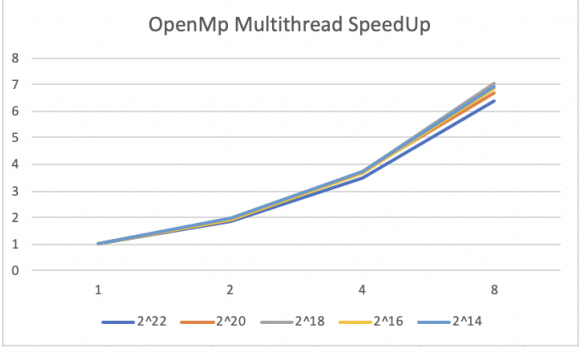

Figure 10: openMP speedup for 1/2/4/8 threads for different input size

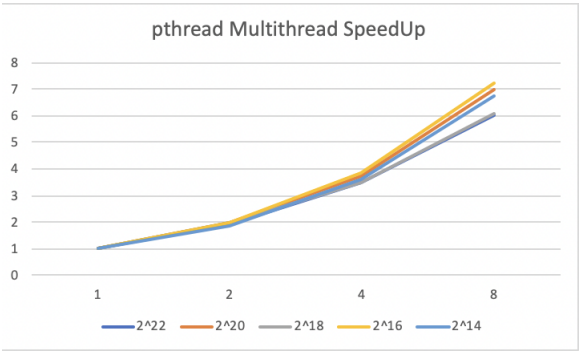

Figure 11: pthread speedup for 1/2/4/8 threads for different input size

In the two graphs above, we show the speedup for openMP and pthread with the increase of thread number. Each line represents a different input size's speedup. We measured input size of 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}. Input size doesn't impact the trend of speedup in general.

As shown in the graph, we are able to reach around 7 speedup (near optimal) for both openMP and pthread without enabling -O3 for C++. The overhead of creating pthread and enabling openMP keeps our implementation from reaching *8 speedup.

Enabling -O3 improves our single thread implementation significantly, but doesn't improve multi-thread version proportionally. So the speedup is not as significant.

4.2.3 CPU Parallel – SIMD

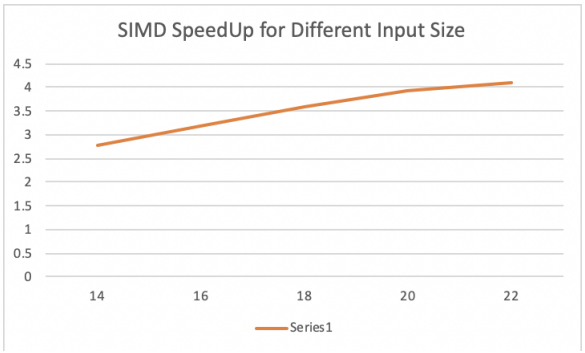

Figure 12: SIMD speedup for different input size

The graph above shows our SIMD implementation speed up compared to the sequential version for different input sizes(2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}).

The speedup is around 3-4. It doesn't reach ideal 8 times speedup for multiple reasons.

(1) SIMD implementation of CALCULATION requires three stages as listed in section 3.4. The first stage is the same as the sequential version, the second stage introduces additional workload, only the third stage experience speedup due to SIMD.

(2) To make things worst, the third stage includes multiple data loads and data stores, which doesn't experience much speedup with SIMD implementation.

To prove our theory, we modified both the sequential version (added an artificial SIMD stage two) and the SIMD version, deleting all addition/multiplication to test a version with only load/store and no computation. There were no visible speedup, proving our assumption – speedup is limited by the loads and stores.

(3) One interesting observation: we timed all three portions separately to profile our speedup, and observed around 10 times speedup in the later portion of stage 3 (exclude rebuilding ws array). By profiling, we found that the SIMD implementation has much less cache miss, which we believe is the main reason for this speed up.

4.2.4 CPU Parallel – SIMD combined with Thread

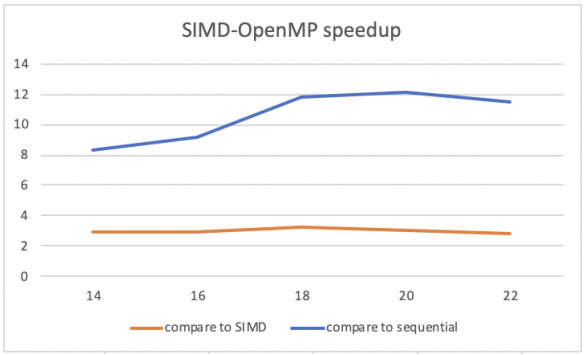

Figure 13: SIMD-openMP speedup for different input size

We combined SIMD implementation with openMP and achieves the speedup shown in the graph above. The speedup compared to SIMD is around 2.5, and the total speedup compared to the sequential version is around 10.

The speedup compared with SIMD doesn't reach optimal because of the portion of work that can be parallalized. SIMD implementation of CALCULATION requires three stages as listed in section 3.4. The first stage and the later portion of the third stage can be paralleled by openMP, but the second stage and the rebuild of ws in the third stage cannot.

4.2.5 GPU Parallel

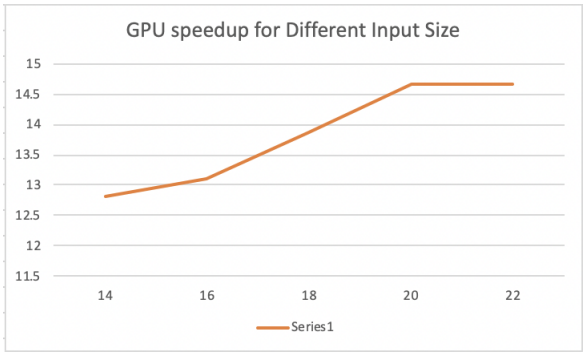

Figure 14: GPU speedup for different input size

The graph above is the speedup of GPU version compared to sequential version for different input sizes.

The general speedup is around 14.5.

```
==28290== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   65.46%  106.45ms         2  53.225ms  48.397ms  58.053ms  kernel(cpxcuda*, cpxcuda*, int, int, bool)
                   18.30%  29.761ms         2  14.881ms  13.586ms  16.175ms  [CUDA memcpy DtoH]
                   16.24%  26.408ms         4  6.6020ms  689.63us  12.516ms  [CUDA memcpy HtoD]
```

Figure 15: GPU profiling

The graph above shows profiling produced by nvprof. Notice that the proportion of kernel computation is rather high. Considering input/output data copying can't be avoided, our algorithm's performance is rather high.

4.3 Environment

All above experiments are conducted on GHC machine.

5 LIST OF WORK

We worked together on all coding and analysis through pair coding. Each taking lead on different parts.

References

[1] `https://blog.csdn.net/enjoy_pascal/article/details/81478582/`.

[2] youtube channels: Reducible; 3blue1brown.

[3] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pages 1381–1384. IEEE, 1998.

[4] Wikipedia. Fast Fourier transform — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Fast\%20Fourier\%20transform&oldid=1073961290`, 2022. [Online; accessed 22-March-2022].