Application of AVX instruction set in Fourier space similarity calculation

Ian Zhang, *Renmin University of China, 2021201774*

***Abstract*—In the realm of cryo-EM (cryo-electron microscopy), Fourier space image similarity calculation for 3D reconstruction is computationally demanding. This paper addresses this challenge through data structure refinement, manual load balancing, and AVX(Advanced Vector Extension) vectorization, leading to an average speedup of 5.01 times over traditional methods in various image set tests.**

***Index Terms*—Cryo-EM, Fourier Space Image Similarity, AVX, Load Balancing**

I. Introduction

CRYO-EM (cryo-electron microscopy) is increasingly becoming a mainstream technology for studying the architecture of cells, viruses and protein assemblies at molecular resolution. [1] However, the process of 3D reconstruction from 2D projections in cryo-EM is computationally intensive, especially in Fourier space image similarity calculations. These calculations often encounter performance bottlenecks due to inefficient data structures, underutilization of parallelization, and ineffective compiler optimizations in C++.

This paper proposes a solution to enhance computational efficiency in cryo-EM image processing to address these problems. Our approach includes data structure optimization, manual load balancing, and the use of Advanced Vector Extensions (AVX), a SIMD (single instruction, multiple data streams) instruction set.

By distributing computing tasks evenly across processors and leveraging AVX's parallel processing capabilities, we significantly improve the processing efficiency of Fourier space image similarity calculations, addressing the inherent computational bottlenecks in cryo-EM.

II. Background

A. Cryo-electron microscopy (cryo-EM)

Cryo-EM is a vital technique for studying cellular structures, viruses, and protein complexes at the molecular level. It employs a transmission electron microscope to observe specimens at cryogenic temperatures. By combining microscopic images from different angles, it enhances signal-to-noise ratios and reconstructs three-dimensional structures from two-dimensional projections, unveiling macromolecular intricacies. [1]

B. Fourier space image similarity algorithm

A critical step in cryo-EM data processing is the calculation of Fourier space image similarity. This involves computing the similarity between two-dimensional real images of samples and their corresponding three-dimensional projection images. The similarity is quantified using Equation (1):

$$diff = \sum_{i=1}^{N} a \cdot \|\text{image}_i - \text{proj}_i\| \tag{1}$$

This calculation is executed through a serial algorithm, as outlined in Algorithm 1.

Algorithm 1: Fourier Space Image Similarity

Result: Calculate the image similarity in Fourier Space
Input: Arrays $dat, pri, ctf, sigRcp$;
Integer num;
Double $disturb0$;
Output: Double $result$

$result \leftarrow 0.0$;
for $i \leftarrow 0$ **to** $num - 1$ **do**
  $result \leftarrow result$ + (norm$(dat[i] - ctf[i] \times pri[i]) \times sigRcp[i]$);
end
return $result \leftarrow result \times disturb0$

C. AVX instruction set

Advanced Vector Extensions (AVX) is a SIMD (Single Instruction, Multiple Data streams) instruction set supported by modern Intel and AMD CPUs. SIMD programming enables parallel processing across multiple cores within a single CPU, allowing simultaneous execution of basic arithmetic and data transfer operations like addition, multiplication, and square root. In this study, we utilize AVX vectorization to enhance the efficiency of Fourier space image similarity calculations in cryo-EM data analysis, leveraging its parallel processing capabilities to significantly speed up these computationally intensive tasks. [2]

III. Methodology

A. Data Structure Optimization

In C++, optimizing code for efficiency often involves managing low-level operations directly to reduce overhead from function calls and object management. [3] Based on this, we optimized the data structure used in the algorithm. Specifically, we separated complex numbers into their real and imaginary parts, resulting in four real-number arrays: dat_0, dat_1, pri_0 *and* pri_1. Additionally, we adopted single-precision floating-point numbers (float) instead of double precision, trading off some accuracy for significant efficiency gains.

Algorithm 2: Data Structure Opzimization

result $\leftarrow 0.0$
for $i \leftarrow 0$ **to** $num - 1$ **do**
  $tmp_1 \leftarrow dat_0[i]$ - $ctf[i] \times pri_0[i]$
  $tmp_2 \leftarrow dat_1[i]$ - $ctf[i] \times pri_1[i]$
  $result \leftarrow result$+ $((tmp_1 \times tmp_1 + tmp_2 \times tmp_2) \times sigRcp[i])$
end
return $result \leftarrow result \times disturb0$

B. Load Balancing Optimization

To address load imbalance issues in parallelization with OpenMP, we implemented manual load balancing. [4] This involved distributing tasks evenly among threads based on the number of available CPU threads. Algorithm 4 illustrates our approach, utilizing arrays $begin$ and end to define the starting and ending points of the data processed by each thread.

Algorithm 3: Thread Work Index

Input: Integer m;
Output: Arrays $begin, end$
$threadNum \leftarrow$ ompGetNumProcs();
$begin, end \leftarrow Array[threadNum]$;
$numPerThread \leftarrow m/threadNum$;
for $i \leftarrow 0$ **to** $threadNum - 1$ **do**
  $begin[i] \leftarrow i \times numPerThread$;
  $end[i] \leftarrow begin[i] + numPerThread$;
end
return $begin, end$

Algorithm 3 is the pioneer algorithm of Algorithm 4. It realizes the calculation of the values of the $begin$ and end arrays, thus completing the allocation of tasks.

Algorithm 4: Load Balancing Opzimization

1 $result \leftarrow 0.0$;
2 $threadNum \leftarrow$ ompGetNumProcs();
3 $localSum \leftarrow 0$;
4 # parallel loop with omp
5 **for** $j \leftarrow 0$ **to** $threadNum - 1$ **do**
6 $i_{begin} \leftarrow begin[j]$;
7 $i_{end} \leftarrow end[j]$;
8 **for** $i \leftarrow i_{begin}$ **to** i_{end} **do**
9 $tmp_1 \leftarrow dat_0[i]$ - $ctf[i] \times pri_0[i]$;
10 $tmp_2 \leftarrow dat_1[i]$ - $ctf[i] \times pri_1[i]$;
11 $localSum \leftarrow localSum + ((tmp_1 \times tmp_1 + tmp_2 \times tmp_2) \times sigRcp[i])$;
12 **end**
13 $result \leftarrow result + localSum$
14 **end**
15 **return** $result \leftarrow result \times disturb0$

C. AVX Vectorization

Advanced Vector Extensions (AVX) provided a valuable solution to optimize sparse matrix-vector multiplication. We harnessed AVX's parallel processing capabilities to significantly accelerate computations. This involved calling AVX2 functions to replace the calculations in 9-11 lines of Algorithm 4.

Algorithm 5: AVX2 Vectorization of Algorithm 4

1 Vectorize the Arrays $dat_0 ... sigRcp$;
2 $tmp_1 \leftarrow$ _mm256_fnmadd_ps($ctfVec, pri0Vec, dat0Vec$);
3 $tmp_2 \leftarrow$ _mm256_fnmadd_ps($ctfVec, pri1Vec, dat1Vec$);
4 $tmpSq_2 \leftarrow$ _mm256_mul_ps(tmp_2, tmp_2);
5 $tmpSum \leftarrow$ _mm256_fmadd_ps($tmp_1, tmp_1, tmpSq_2$);
6 $localSum \leftarrow$ _mm256_fmadd_ps($tmpSum, sigRcpVec, localSum$);
7 **return** $localSum$

IV. Experiment

A. Device Information

The device information of experiment is shown in Table I.

B. Experiment Process and Results

We conducted experiments using image sets of varying sizes. The results are presented in Figure 1. Specifically, '$original$' denotes the baseline algorithm with static policy parallelism using OpenMP only, '$opt1$' represents data structure optimization, '$opt2$' includes load balancing optimization, and '$opt3$' integrates AVX vectorization.

TABLE I
Device Information

Configuration	Content
Model name	13th Gen Intel(R) Core(TM) i9-13900HX
Physical Core	16
Logical Core	32
Architecture	x86_64
OS	Ubuntu 20.04.6 LTS
Memory	16 GiB
Compiler	gcc 9.4.0
Toolkit version	OpenMP 4.5, AVX2

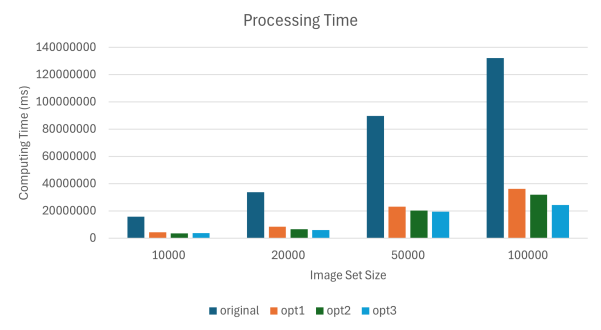

Fig. 1. Processing Time of Algorithms on different Image Sets

As shown in Table II, we compared the speedup achieved by '$opt3$' relative to the '$original$' program on various dataset sizes. This speedup remained consistent with average value 5.01x even as the dataset size increased, demonstrating the stability of our optimization approach.

TABLE II
Speedup Performance

Image Set Size	Speedup
10000	4.25
20000	5.74
50000	4.59
100000	5.44

V. Conclusion

Our study introduces an effective solution to enhance the computational performance of Fourier space image similarity calculations in cryo-EM. By optimizing data structures, applying manual load balancing, and utilizing AVX vectorization, we achieved a significant speedup in processing time. The results from various image set tests validate our approach, highlighting its potential applicability in similar scientific computing challenges. Future work will aim at further optimization and exploring broader applications of these techniques.

References

[1] J. L. S. Milne, M. J. Borgnia, A. Bartesaghi, E. E. H. Tran, L. A. Earl, D. M. Schauder, J. Lengyel, J. Pierson, A. Patwardhan, and S. Subramaniam, "Cryo-electron microscopy – a primer for the non-microscopist," *FEBS Journal*, vol. 280, pp. 28-45, 2013.
[2] H. Jeong, W. Lee, S. Kim, and S.-H. Myung, "Performance of SSE and AVX Instruction Set," arXiv:1211.0820 [hep-lat], Nov. 2012.
[3] Optimizing C++/Code optimization/Faster operations, Wikibooks, [Online]. Available: https://en.wikibooks.org/wiki/Optimizing_C\%2B\%2B/Code_optimization/Faster_operations. [Accessed: Jan. 15, 2024].
[4] M. F. Ali and R. Z. Khan, "THE STUDY ON LOAD BALANCING STRATEGIES IN DISTRIBUTED COMPUTING SYSTEM," *IJCSES*, vol. 3, no. 2, April 2012.