A Fast-Running Kernel for Gaussian Mixture Model

1st Cheng Leon Jiang
ECE Department
Carnegie Mellon University
Pittsburgh, PA
cljiang@andrew.cmu.edu

2nd Jiajun Wan
ECE Department
Carnegie Mellon University
Pittsburgh, PA
jiajunw2@andrew.cmu.edu

3rd Yihang Liu
ECE Department
Carnegie Mellon University
Pittsburgh, PA
yihangl@andrew.cmu.edu

***Abstract*—This document is the final project report for team Cheng Leon Jiang, Jiajun Wan, and Yihang Liu for the kernel design project for Class 18645, How to Write Fast Code I, at Carnegie Mellon University.**
The content of this report consists of Five parts: *Introduction, Updated Design, Parallelization, Performance Plots, and Future Directions*

I. Introduction

The Gaussian Mixture model, which we know as GMM, is an unsupervised Machine Learning Algorithm that represents the normally distributed sub-population over a whole population. Unlike the K-means algorithm, which outputs the label of data points, GMM computes the probability of each data point and assign them into the one with the highest probability. Normally, there existing two learning models in un-supervised learning: probability model and non-probability model. Probability model represents cases where we are trying to learn $P(Y|X)$ instead of one single label. Within the process, we can get the probability distribution of Y from previously unknown data label X, hence outputting the continuous probability of the label and doing the soft-assignment. Non-probability model, on the other hand, is by learning a model which outputs a decision function $Y = f(X)$, by inputting data point X, we will get one fixed result Y as the prediction, which is classified as hard assignment. The learning process of GMM including the following steps:

1. Initialize the data placement probability θ with some values(random or otherwise)
2. Model each region with a distinct distribution
3. Estimate with the soft parameter r_{nk} using Bayes' Rule, where we define $r_{nk} = p(z_n = k|x_n)$
4. Solve the MLE given the soft r_{nk} for parameter estimation purposes
5. Update θ using the r_{nk} using MLE
6. Loop back to step 1 and compute the process again, until converges.

There are many open-source libraries and packages online that implements the process of GMM, known as EM steps. Among these libraries, Scikit-learn, which is a famous and free software machine learning library for the Python programming language, has a specific API just for GMM calculation, known as *sklearn.mixture.gaussian_mixture*. Our goal of the project is to pick one specific API function within the *sklearn.mixture.gaussian_mixture* and improve its performance. Upon examination, we picked the API function *sklearn.mixture.gaussian_mixture.estimate_log_gaussian_prob* as our project baseline function. The function takes the parameters as following: input data points X, which is a matrix with dimensions n_samples by n_features; means, which is an matrix with shape n_components by n_features; precisions_chol, which array of Cholesky decomposition of the precision matrices of size n_components; covariance_type, which specifies the type of the covariance matrix of the Gaussian models and take input as the options: 'full', 'tied', 'diag', 'spherical'. Function returns log_prob, which is an matrix of the shape n_samples by n_components.

In the testing stage of this project, we will change the number of input data points X, n_samples, with number of Gaussian components and covariance type fixed at three and spherical. This is the only thing we will be altering among the parameters. This shows how our kernel will behave with the increment of sample data size. Ideally, as the number of input data points increases, the performance compared to the naive approach will increase.

II. Updated Design

A. How Our Design Is Different From Project Proposal

Comparing to the original project proposal back in September, chunks of changes have happened since then. Our initial idea of the project is to run a fast kernel for a much optimal research topic: Deep Clustering with Self-supervision using Pairwise Data Similarities(DCSS). This is a novel method presented by Mohammadreza Sadeghi and Narges Armanfard and has been proven to be one of the most efficient way to learn models in the topic self-supervision deep clustering. Our ideas were simple: if it had high-accuracy and fast convergence, then we can try to make it even faster. We were targeting all three stages of this method: train the model, cross-validation, and target prediction stage. Our initial idea is to design three kernels.

However, we soon realize the difficulty of implementing such idea: the implementation of DCSS uses a lot of deep learning accelerated PyTorch library functions, like Multi-Layer Perceptron, Convolution, and much more. This became the major obstacle of reaching our goal, because it is very difficult and time consuming to implement such deep learning specific accelerating functions in C. The most ideal way is to switch to another topic which can be relatively easier to be implemented in sequential C code. Hence, we picked GMM due to a few reasons:

1. There is an opensource library, sklearn, with detailed python source codes written in numpy. The GMM algorithm itself is composed to different sub-functions. It will be easier for us to pick specific function to implement in sequential C. For this case, *sklearn.mixture.gaussian_mixture.estimate_log_gaussian_prob*, which is the function that generates predictions in inference stage, became our choice of function
2. This function itself has multiple numpy function calls that allows us to design multiple kernels
3. Each kernel function takes inputs based on number of data points, allowing us to record the performance based on different size of input dataframe

This is essentially why we picked GMM as our final topic. In the next section, we will explain our design choice in each kernel and how our thoughts have evolved during re-designing or implementation stage, if any.

III. Kernel Design and Identify Independent Instructions

Kernel 1: Matrix Multiply

First kernel we are designing is the matrix multiplying kernel. For the specific case, it is the dot product of two matrices: *input matrix X*, and *input matrix log_prob2_means_T_precisions*. The first matrix takes the form of input data points x 4, which is n_samples times n_features. The second matrix takes the shape of n_features times n_components, which takes the form of 4 x 3, and the operation output the result log_prob2 matrix with the shape of input data poitns with three cluster means in each row, n_samples times n_components.

For independent instructions within this kernel, since it is essentially matrix multiply, we are doing some similar work that we had done in the class: write out the operation of multiplying each element and identify which ones can be done independently.

For example:

$$output = \begin{bmatrix} X_{00} & X_{01} & X_{02} & X_{03} \\ X_{10} & X_{11} & X_{12} & X_{13} \\ X_{20} & X_{21} & X_{22} & X_{23} \\ X_{30} & X_{31} & X_{32} & X_{33} \\ X_{40} & X_{41} & X_{42} & X_{43} \\ ... & ... & ... & ... \end{bmatrix} \begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \\ w_{30} & w_{31} & w_{32} \end{bmatrix}$$

Each operation $X_{ij} * w_{ij}$ can be computed independently since the result of each element in matrix $output$ is the multiply add of these operations. To efficiently complete the compute process, we will be using $\_mm256\_broadcast\_sd$ and $\_mm256\_fmadd\_pd$ for fast calculation purposes.

Before implementing this idea, there are two conditions need to be satisfied to avoid bubbles in the pipeline: m * n is less than number of total registers(16 with 4 doubles each), and large enough so it can get closer to the theoretical peak. In the benchmark testing stage, we found latency of Fused add on 015 ECE machine at Carnegie Mellon University to be 5 and has the throughput IPC 2, hence, the total elements needs to be larger than the product of these two. Then, we are trying to use as many registers as possible in the machine. LSCPU shows that the available registers in the ECE 015 machine is 16. Essentially, out condition has to satisfy:

$$Number\ Of\ Registers = 16 > m * n \geq L(FMA) * T(FMA)$$

What we designed in the end, is to use 9 registers as output registers for output matrix store, and use 3 as input registers for input matrix X, and 3 registers for broadcasting elements in matrix b. This boils down out kernel design as **12 x 3**. Another note-worthy idea we came up during the implementation stage for this kernel is the strategy of packing the data before feeding to the kernel. This idea will be be touched on and explained in detail in the following section of this paper.

Kernel 2: Matrix Row-wise Inner Product

In the second kernel, we are doing an inner product of vectors from each row of the input matrix X.

$$x = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & x_1^{(4)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & x_2^{(4)} \\ x_3^{(1)} & x_3^{(2)} & x_3^{(3)} & x_3^{(4)} \\ x_4^{(1)} & x_4^{(2)} & x_4^{(3)} & x_4^{(4)} \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

$$z = \begin{bmatrix} {x_1^{(1)}}^2 + {x_1^{(2)}}^2 + {x_1^{(3)}}^2 + {x_1^{(4)}}^2 \\ {x_2^{(1)}}^2 + {x_2^{(2)}}^2 + {x_2^{(3)}}^2 + {x_2^{(4)}}^2 \\ {x_3^{(1)}}^2 + {x_3^{(2)}}^2 + {x_3^{(3)}}^2 + {x_3^{(4)}}^2 \\ {x_4^{(1)}}^2 + {x_4^{(2)}}^2 + {x_4^{(3)}}^2 + {x_4^{(4)}}^2 \\ \cdots \end{bmatrix}$$

The most intuitive way is to directly do a SIMD Multiplication of the vector of one row which is exactly size of four. And then do a SIMD hadd, horizontal add, where the SIMD instruction horizontally adds adjacent pairs of doubles in a and b, and pack the results in dst. But this design is very inefficient. First of all, there are two types of SIMD instructions in the kernel that bring more complications. Second, the horizontal addition is dependent on the output of the first SIMD Multiplication instruction, creating a dependent chain, thus reducing the performance and ability to parallelize. In order to fully parallelize, we need to make the instructions in the kernel fully independent. One way to do this is to go along the column instead of the row. Because each row is independent of each other, the elements in one column are independent of each other. We can make the input X into column major order and read along the column.

Thus the design is as follows, we read input X in column major order, create a temporary variable with zeros, do a SIMD FMA of itself and the temporary variable created to calculate the squared value of element in this column, save to the temporary variable, move to the next column, then do a SIMD FMA of itself and the temporary variable to calculate the squared value of element in this column, save to the temporary variable, and finally if the current column is the last, we store the temporary value in the output matrix. By doing along the column, we can fully parallelize all the SIMD Loads and SIMD FMA, thus giving a high performance. Because we are parallelize only SIMD FMA, the theoretical peak is 2 * 4 * 2 = 16 FLOPs per cycle. In order to get as close as the theoretical peak, we need to parallelize as many SIMD FMAs as possible and use registers less then the maximum of 16.

We use 8 SIMD registers for storing the column of input X, 8 registers for holding the temporary values from SIMD FMA, thus parallizing 8 SIMD FMA instructions. The details of how and why parallelizing in this way is shown in the following parallelization section.

Kernel 3: Vector Outer Product

In this kernel, we need to optimize a vector outer product operation. To be more specific, the two input vector is einsum, which is the result of Einstein summation caculated in the kernel 2, and precision which stores elements contains parameters of the GMM. The einsum vector has n_samples elements and the precision vector contains n_features elements.

$$\mathbf{z} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \bigotimes \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} x_1w_1 & x_1w_2 & x_1w_3 \\ x_2w_1 & x_2w_2 & x_2w_3 \\ \vdots & \vdots & \vdots \\ x_nw_1 & x_nw_2 & x_nw_3 \end{bmatrix}$$

Two conditions are supposed to be ensured: registers should be less than 16, and try to fulfill the pipelines, and since the latency of the SIMD multiplication is about 3 cycles, we need at least 3 SIMD multiplications to fulfill the pipeline. Since it is a outer product operation, there are two input vector: precision and einsum, which stores the result of the Einstein summation. The whole idea is we multiply the elements in vector precision by -0.5 to reduce the operations in the behind, then get the first 4 elements of the result of Einstin summation and multiply them with vector precision one by one to get the first 4 x 3 sub matrix then move downward through the vector einsum to get the second 4 elements and so on, until we use up the vector einsum, finally we will get the whole matrix $log\_prob\_3^T$.

We use 3 SIMD registers to store the value of broadcast precision vector, the other 3 SIMD registers to store the value of the result from Einstein summation, and 9 SIMD registers for result value in each loop. In each loop, there are 9 independent SIMD multiplications. Firstly, we broadcast the 3 elements in vector precision, then in each kernel, we load a 4 x 3 matrix in 3 SIMD registers, then do 9 SIMD multiplications between these 3 pairs registers to get 9 4 x 3 matrices. Each multiplication is independent, and we have 9 multiplications in the kernel and we use 15 registers in total. In this case, we meet the 2 conditions mentioned above.

Kernel 4: Matrix/Vector Addition

In the final kernel, we are doing an addition among matrices and broadcasted vectors.

$$a = \begin{bmatrix} a_1^{(1)} & a_1^{(2)} & a_1^{(3)} \\ a_2^{(1)} & a_2^{(2)} & a_2^{(3)} \\ a_3^{(1)} & a_3^{(2)} & a_3^{(3)} \\ a_4^{(1)} & a_4^{(2)} & a_4^{(3)} \\ \cdots & \cdots & \cdots \end{bmatrix} \; b = \begin{bmatrix} b_1^{(1)} & b_1^{(2)} & b_1^{(3)} \\ b_2^{(1)} & b_2^{(2)} & b_2^{(3)} \\ b_3^{(1)} & b_3^{(2)} & b_3^{(3)} \\ b_4^{(1)} & b_4^{(2)} & b_4^{(3)} \\ \cdots & \cdots & \cdots \end{bmatrix} \; c = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}^T$$

$$z = \begin{bmatrix} a_1^{(1)} + b_1^{(1)} + c_1 & a_1^{(2)} + b_1^{(2)} + c_2 & a_1^{(3)} + b_1^{(3)} + c_3 \\ a_2^{(1)} + b_2^{(1)} + c_1 & a_2^{(2)} + b_2^{(2)} + c_2 & a_2^{(3)} + b_2^{(3)} + c_3 \\ a_3^{(1)} + b_3^{(1)} + c_1 & a_3^{(2)} + b_3^{(2)} + c_2 & a_3^{(3)} + b_3^{(3)} + c_3 \\ a_4^{(1)} + b_4^{(1)} + c_1 & a_4^{(2)} + b_4^{(2)} + c_2 & a_4^{(3)} + b_4^{(3)} + c_3 \\ \cdots & \cdots & \cdots \end{bmatrix}$$

Because each row only has three elements, if we do SIMD Addition in row major order, we would need to pad a column of zeros in the end. This is not ideal, so an alternative way to do this is to do SIMD Addition in column major order. By doing in this fashion, we just need to broadcast the elements in the vector c. In order to fully parallelize, we need to make the instructions in the kernel fully independent. Because each column is independent of each other, and each row is independent of each other, all the elements are independent of each other. Going along the column and simultaneously complete the calculation along the row can be fully parallelized. We can make the input matrix a and b into column major order and read along the column.

Thus the design is as follows, first we create broadcast variables with for input vector c, read input matrix a and b in column major order, do a SIMD Add of a and b and acumulate in a registers, do a SIMD Add of a and broadcasted c registers and accumulate in a registers, finally we store the values acumulated in registers a in the output matrix. By doing along the column, we can fully parallelize all the SIMD Loads and SIMD Add, thus giving a high performance. Because we are parallelize only SIMD Add, the theoretical peak is 1 * 4 * 1 = 4 FLOPs per cycle. In order to get as close as the theoretical peak, we need to parallelize as many SIMD Adds as possible and use registers less then the maximum of 16.

We use 6 SIMD registers for storing the column of input a, 6 registers for the column of input b, and 3 registers for broadcasting c, thus parallizing 6 SIMD Add instructions. The details of how and why parallelizing in this way is shown in the following parallelization section.

Other Improvements

Aside from 4 kernels we designed above, we also designed SIMD implementation for all other sections that can be parallelized.
Overall, three more sections can be parallelized using SIMD instruction:
1. $precisions = precisions\_chol **2$. The independent instructions within this are multiply and set, which can be implemented using 4 SIMD fmadd instructions;
2. $means **2$. The independent instructions within this are simply element-wise multiplication which can be done in parallel for each element;
3. $np.sum(means **2, 1) * precisions$. The independent instructions within this are addition and multiplication which we can parallel using simd add and simd mul instructions.

Although these operations will not impact the overall performance by a lot since their input sizes are relatively small, we can still see the difference after implementing SIMD instructions individually.

IV. Parallelization

Kernel 1: Matrix Multiply

Explanation of Parallel Computing Implementation:

Part 1:

```
__m256d c_temp_5 = _mm256_set_pd();
__m256d X_T_temp_i;
__m256d broad_temp_i;
...
```

First, we initialize the output temp using SIMD vectors. These temps also include input X temp and broadcast temp variables.

Part 2:

```
for (int i = 0; i < n_samples; i += 12)
```

The outer for loop serves the purpose of loop through all input data points inside the matrix X. Hence, we loop until n_samples with the increment of 12 each time.
This serves the purpose that each 12 elements of input X will be inside a dependent chain, ie. different packet of X divided by 12 can be computed in parallel.

Part 3: (function params are omitted)

```
X_T_temp_1 = _mm256_load_pd();
broad_temp1 = _mm256_broadcast_sd();
c_temp_1 = _mm256_fmadd_pd();
...
```

Inside the inner-loop is where the parallel computing takes place. First, we load in the data from input matrix X, 4 floating point doubles at a time; we then broadcast the single element inside the input matrix precisions(B), and implement the fmadd method to calculate and store the variable in the destination matrix c. Such dependent chain is written 9 times within the inner loop to ensure maxinum performance. Input element X and broadcasted temp are only initialized three times since they can be reused as input for the calculation.

Part 4: (function params are omitted)

```
_mm256_store_pd();
...
```

The final stage of the kernel involves store the vectors into the destination memory location. The same instruction is re-written 9 times within the kernel for parallel computing.

Aside from the specific implementation itself, the order where we are inputting the data also came up as the bottom-neck during the development of implementation. This is due to the fact that our output matrix has a number of columns less than 4 (n_components is fixed to be 3 in our case). Hence, one cannot do the normal store of having 4 elements store at once in each row of C. This problem is resolved by packing the data in a different order: instead of normally packing data in row major for input matrix X, we pack the data in column major. To compensate this approach, we also reading the data in input matrix precisions(B) by going down column direction one at a time.

As as result, the output matrix log_prob2 is also in column major. This issue is then resolved later during the last step by having all addition matrix in column-major order and doing one transpose over the final summation result matrix.

Kernel 2: Matrix Row-wise Inner Product

The goal is to parallelize as many SIMD FMA as possible but use registers under the limit of 16. Once the main goal is clear, we can first push the number of registers for input X to the limit. The maximum number of registers to hold a column of 4 input X values is 8, because if we use 8 registers for input X, we must have 8 registers left for the registers that hold the temporary values of FMA. By assigning registers in this way, we use all the 16 registers. With this configuration, 8 SIMD FMA instructions are parallelized. 32 elements are computed each time in the kernel. Now let's take a closer look at the actual implementation of the SIMD parallelization.

First we iterate along the columns, giving us a total of $\frac{nsamples}{32}$ iterations. In each iteration, we iterate along the rows with 4 iterations to perform the kernel calculation. When reading input X values in the leftmost column, we need to initialize the temporary register to zeros.

```
c_temp=_mm256_set_pd(0.0,0.0,0.0,0.0);
...
```

Iterate along the rows to perform the kernel calculation. First, we load 8 input X of total of 32 doubles along the current column. Second, we perform the SIMD FMA operation with X and temporary register c.

```
X=_mm256_load_pd((double *)X_input);
...
c_temp = _mm256_fmadd_pd(X, X, c_temp);
...
```

Finally, after we iterate over along the rows, the inner product for the 32 rows are calculated and stored in the 8 temporary registers. Now we store them into the output array.

```
_mm256_store_pd((double *)output, c_temp);
...
```

Kernel 3: Vector Outer Product

Explanation of Parallel Computing Implementation:
Part 1:

Firstly, we multiply the elements in the vector precision by -0.5 to reduce later operations:

```
precisions[0] *= -0.5;
```

Then we broadcast the elements in the vector precision and store them in SIMD registers:

```
broad_temp1 = _mm256_broadcast_sd(
    (double *)&precisions[0]
);
...
```

Part 2:

for (int i = 0; i $\leq$ $n\_samples$; i += 12)

Each iteration we calculate three 4 x 3 matrices and move downward 12 elements through the vector einsum.

In each for loop, we load 12 elements of vector einsum in 3 SIMD registers:

```
__m256d log_prob3_einsum_temp1 =
    _mm256_load_pd(
    (double *)&log_prob3_einsum[i]
);
...
```

Part 3:

Then we do 9 SIMD multiplications between the 3 pairs elements of vector einsum and precision:

```
__m256d log_prob3_temp1 =
    _mm256_mul_pd(
        broad_temp1, log_prob3_einsum_temp1
    );
...
```

Each multiplication in this multiplication chain is independent on each other, and considering 9 is larger than the latency of SIMD multiplication, this arrangement can reach the theoretical peak ideally.
Part 4:

The final stage of this kernel is to store the result matrix in to destination array. In each iteration, there are 9 repeat store operations:

```
_mm256_store_pd(
    (double *)&log_prob3_T[i], log_prob3_temp1
);
...
```

Kernel 4: Matrix/Vector Addition

The goal is to parallelize as many SIMD Add as possible but use registers under the limit of 16. Once the main goal is clear, we can first push the number of registers for input a and b to the limit. The maximum number of registers to hold a column of 4 input a and b values is 12, because if we use 12 registers for input a and b, each of 6, we have at least 4 registers left for the registers that hold the 3 broadcast value of c. By assigning registers in this way, we use 15 registers. With this configuration, 6 SIMD Add instructions are parallelized. 24 elements are computed each time in the kernel, 8 rows each time along the column. Now let's take a closer look at the actual implementation of the SIMD parallelization.

First we broadcast values in vector c

```
c_broad = _mm256_broadcast_sd(c);
...
```

Then we iterate along the columns, and in each iteration we parallel the operations along the rows, giving us a total of $\frac{nsamples}{8}$ iterations. In each iteration, we iterate along the rows with 3 iterations to perform the kernel calculation. First, we load values into 6 input a and 6 input b registers each of 8 rows of total of 24 doubles each along the column. Second, we perform the parallelized SIMD Add of a and b into a, and SIMD Add of a of c into a, with a total of 6 independent SIMD Adds covering the 8 rows for each column with a total of 24 elements

```
a = _mm256_load_pd(log_prob2);
...
b = _mm256_load_pd(log_prob3);
...
a = _mm256_add_pd(a, b);
...
a = _mm256_add_pd(a, c);
...
```

Now we store them into the output matrix.

```
_mm256_store_pd(output, a);
```

The designs above will theoretically provide us with a result that is as close to peak as possible.

V. Performance Plots

Our Final combined kernel running result shown as below: (unit: number of cycles each function takes)

	fast_kernel	origin_python
np.dot	599628	8709273
einsum	232604	9419105
np.outer	527759	16558341
last_add	829796	18534553

Figure 1. Result Comparison in Num of Cycles

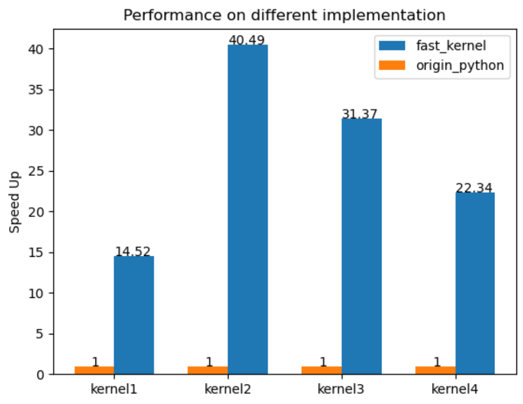

Figure 2. Performance on different implementation

The overall result average speed is by degree of **27.18** faster than the naive approach. Speed up Goal is Satisfied.

	Performance	Theoretical Peak	Percentage
kernel 1	12.01	16	75%
kernel 2	10.32	16	64.5%
kernel 3	1.6	8	20%
kernel 4	2.43	4	68.25%

Figure 3. Performance Comparison

From above we can see that besides the third kernel all other kernels have reached an acceptable range comparing to the theoretical peak.
The fact that the outer product kernel has such as low performance could be due to the shape of the input data: for our input matrix A, the value of k is 1, means the packing data speed will be slowed due to its shape. Given more time on the project, we could possibly looking into a solution to such issue.

VI. Future Directions

Overall, our project achieved our goal of getting the code speed up and getting close to theoretical peak. However, given that the semester is shortened, there are still many ways we can explore to make our code run even faster.

During our implementation, we have tried the approach of using pragma openMp by various ways, including parallel for, parallel region, parallel task and various scheduler methods for load balancing. Knowing that parallel comes with a cost, we expecting the performance will not be improved too much on some of the parallel methods, however, when we tested every method of pragma openMp, almost every openMP method had our code performance worsen by time of 10. For example, when we added pragma omp for for the for loop existing in either the second or the fourth kernel, the performance decreases with the increase number of threads.

This could either mean that there was nuance errors when we were implementing the pragma method, or, there still exists subtle errors within our SIMD implementation that hinders the implementation of parallel computing. We were aware that parallel computing comes with the cost, which means with either too less or too many number of threads, the performance of our code will start to drop. However, the fact that none of the thread numbers produces us with a better performance suggested other parts in the code algorithm we need to explore.

Nonetheless, due to the time constrain of this semester, we were not be able to get the final answer to this question, leaving us with opportunities in the future to further explore the possibility of improving the GMM algorithm.