Cheng Jiang
Jiajun Wan
Yihang Liu

Kernel Report

ECE 18645
Prof. Tze Meng Low
Oct. 29th, 2021

Part 1
Algorithm Description

Our project focuses on the **Gaussian Mixture Model** within the Pair-wise Clustering topic:

A Gaussian mixture model**(GMM)** is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians. The Source code of the GMM model is taken from the scikit-learn data source, which implements different classes to estimate Gaussian Mixture models which can correspond to types of estimation strategies.

For our kernel design, we focus on the **Prediction Stage of the GMM model**.

```
start_time = time.time()
kernel_out = 2 * np.dot(X, means.T * precisions)
end_time = time.time() - start_time
log_prob = (
    np.sum(means ** 2, 1) * precisions
    - kernel_out
    + np.outer(np.einsum("ij,ij->i", X, X), precisions)
)
"""
2 * np.dot(X, means.T * precisions) depends on the means.T transpose of the mean matrix.
"""
```

(a) What are the independent operations?
The independent operations are our main kernels which are independent of each other.

(b) What are the dependent instructions that make up the independent operation?
The dependent instructions can be seen below:
means.T * precisions
The kernel output 2 * np.dot(X, means.T * precisions) depends on the multiplication of precision matrix and means.T transpose of the mean matrix.

(c) What functional unit(s) are used to compute those instructions?
FMA for the SIMD_FMA and ALU for other normal instructions.

Part 2
Performance Peak

(a) Describe the machine of your choice
We are running the code on the ECE Cluster Machine No. 16:
1. Hostname of the machine : ece016.ece.local.cmu.edu
2. CPU model and manufacturer : Intel(R) Xeon(R) and Intel
3. Base and maximum frequencies : Base level: 2.40 GHz, Max level: 2.90 GHz
4. Number of physical cores : 28
5. Number of hardware threads : 56

(b) What are the latency and throughput of the instructions identified in the previous question?
We have tested the latency and throughput of the instructions on the cluster Machine:
1. ADD Latency:
- Documented Latency for Xeon(R) machine: **3**, Throughput for Xeon(R) machine: **4**
2. FMA Latency:

- Documented Latency for Xeon(R) machine: **5**, Throughput for Xeon(R) machine: **12**

(c) Do you have the appropriate SIMD instructions on your machine?
Yes. We are currently only using SIMD_ADD and SIMD_FMA Instructions for label prediction.

(d) Are there specialized units that can be used your machine?
Not identified at this moment.

(e) Explain how you computed the theoretical peak of your kernel(s)?
In the np.dot, each cycle we do a FMA(one multiplication and one addition) and there are two pipes, since we use SIMD registers, each time we calculate 4 double-type data, so our theoretical peak is $2 \times 2 \times 4 = \mathbf{16}$. Considering the overhead about transposing, the theoretical peak is less than 16.

Part 3
Performance Baseline

(a)

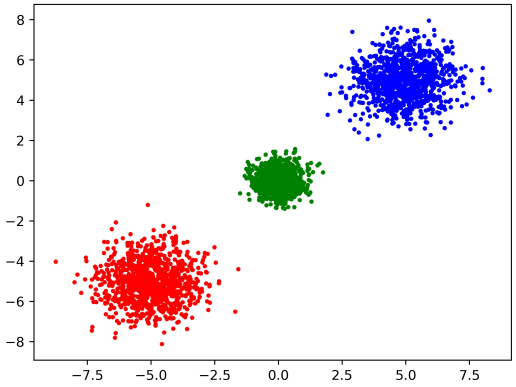

Figure 1. *Third and Fourth Dimensions Plot for the Gaussian Model Running Result*

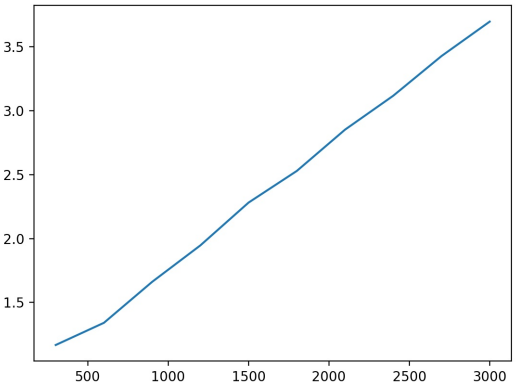

Figure 2. *Number of Data Point Input VS Running Time(sec/10000 loops)*

(b) How is the theoretical peak comparing to the existing baseline?
Suppose we have data input of shape (3000×5), 3000 data points with 5 dimensions, means shape of (3×5), precisions of shape (3×1), for $2 * np.dot(X, means.T * precisions)$, we need 15 FLOPS for the matrix transpose, $5 \times (3 + 2)$ for the matrix multiplication, $3000 \times (5 + 4)$ for the numpy dot product, and 3000 FLOPS for the multiplication with the scalar of 2. The total is 30,040 FLOPS. We run this for 100000

times and the total time for data points of 3000 is 3.75 seconds. The time for one run is 3.75 $\times 10^{-5}$. So the FLOPS/second is $\frac{30040}{3.75\times10^{-5}} = 0.8\times10^9$. Finally we divide the 0.8×10^9 by the base frequency of 2.4GHz. So the existing perfomance is $\frac{0.8\times10^9}{2.4\times10^9} =$ **0.33** FLOPS/Cycle. The theoretical peak of around **16** FLOPS/Cycle is around **48**× faster than the existing one.

Part 4
Design of Kernel Implementation

(a) Important parameters for determining the size of kernel.
Our kernel is basically transform of the equation below:

$$2 * np.dot(X, means.T * precisions)$$

There will be two kernels for the transformation:
1. Matrix transpose for the means.T operation;
- Parameter to determine the kernel size will be **number of rows in means**, which is the **dimension**(number of features) of the input data.
2. Matrix multiply for the dot product operation:
- Parameter to determine the kernel size will be **number of rows of X**, which is the **size** of the input data

(b) How will SIMD instructions be used?
Our SIMD instructions will be used to replace the matrix multiply and matrix transpose. For two separate kernels:
1. To replace the means.T operation in numpy, we will use SIMD_permute_pd and SIMD_shuffle_pd to design a kernel with size of 8 to speed up the transpose process of the means matrix. Kernel design as below:

```
__m256d a0 = _mm256_load_pd(means[0]);
__m256d a1 = _mm256_load_pd(means[1]);
...
_mm256_permute4x64_pd(a0, 0b11011000);
_mm256_permute4x64_pd(a1, 0b11011000);
...
_mm256_store_pd(b, a0);
_mm256_store_pd(b+4, a1);
...
```

The size of the transpose kernel is 8 * 8, which gives **64** operands in total in order to reach the theoretical peak on a 16 register ECE cluster machine.

2. To replace the dot product operation between two matrices, we use FMA and ADD:

```
for (int i = 0; i != k; ++i){
     b0 = _mm256_load_pd(&b[i*n]);
     ...
     a0 = _mm256_broadcast_sd(&a[i*m]);
     c0 = _mm256_fmadd_pd(a0, b0, c0);
     ...
}
_mm256_store_pd(&c[1*n], c1);
...
```

The size of the multiply kernel is 14 * 4, which gives **56** operands in total in order to reach the theoretical peak on a 16 register ECE cluster machine.