Parallelizing Gradient Boosted Regression for Facial Landmark Recognition

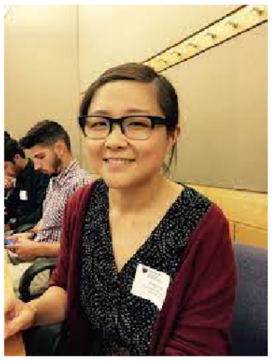
Zhecheng Yao, Yixian Gan, Rebecca Qiu, Lucy Li

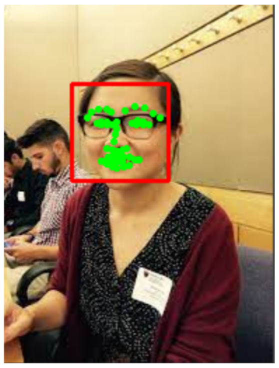
May 6, 2023

Abstract

In this paper we present the optimization of **FaceX-Train**, the training algorithm of the facial recognition program **FaceX**, using MPI distributed parallelism and other optimization methods including vectorization and Eigen. First, we analyze the architecture of the training model, a two-level gradient boosted regression, and determine the components of the outer and inner regressions to be parallelized. We then proceed to implement our parallel design on the chosen components, and we test the implementations by conducting performance benchmarks and scaling analysis. As a result, We achieved a 2.2x speedup for the training process of the model with our optimization implementation.

1 Background and Significance





On Github repository there is a facial recognition package called **FaceX** (`https://github.com/delphifirst/FaceX`), which performs face contour recognition to predict the position of facial landmarks and face boundary as demonstrated in the images on the right. The training part of the package, **FaceX-Train**, takes in a large set of images with annotated landmarks and trains the model by using a two-level boosted regressor to make progressive inference and minimizing the alignment error at each step [1]. We consider it significant to optimize this algorithm because facial recognition programs are widely used in real-world applications such as security, personalization, and accessibility. It plays a crucial role in preventing fraud, identifying suspects in criminal investigations, users authentication, and much more. The accuracy of facial recognition models depends heavily on the volume of the training images. Therefore, enhancing the training efficiency will allow the models to scale up to large datasets and learn facial features more sufficiently.

2 Scientific Goals and Objectives

Our goal with this project is to optimize the sequential model of **FaceX-Train** to achieve higher training efficiency and scalability through parallelization. The serialized algorithm takes approximately 82.16 seconds to train one regressor on 1000 images of 250×250 dimension with 5 landmarks on one Intel Xeon E5-2683 v4 CPU. However, if we choose a dataset with more images, larger image dimension or more annotated landmarks, the training time would take significantly longer. Therefore, we seek to shorten the training time by employing an arsenal of parallelization and vectorization tools including OpenMP, MPI, SIMD, and ISPC. To achieve the desired distributed-memory structure across multiple nodes for our software design, we need to access computing hours on a HPC architecture to train the optimized model and validate the results.

3 Algorithms and Code Parallelization

Part 1. Methods and Algorithms

FaceX-Train consists of two levels of gradient boosting regression: outer regression and inner regression.

1. Outer Regression uses T stages of Explicit Shape Regression (ESR) algorithm and sequentially learns to reduce the alignment error between the input shape S^t and the mean shape of all training images $\hat{S}$, where $t = 1, 2, \ldots, T$. Each stage t iterates over N inner regressions. We use MPI to optimize the read in function of training image data `GetTrainingData()` with auxiliary functions `serializeDataPoint()` and `deserializeDataPoint()`.

2. Inner Regression uses Learn Stage Regressor (*LSR*) algorithm and iterates over K Fern loops, sequentially learning to greedily fit the regression target. Each Fern loop updates and selects useful features distributed around salient landmarks, where features that maximize correlation with the target Y is selected. The methods of the *LSR* algorithm we chose to study and the parallelization methods we used are:

   - `FernRegressor()`: MPI
   - `Covariance()`: SIMD & ISPC
   - `Procrustes()`: SIMD, with auxiliary function `_mm256_get1_pd()`.
   - `OMP()`: Eigen, with auxiliary function `OMP_sub_routine()`.

The serial **FaceX-Train** algorithm takes advantage of the **OpenCV** library to manipulate matrix operations efficiently. We build our optimized model on top of the existing **OpenCV** implementations. We shall also replace them with more efficient methods when necessary. In the following sub-sections, we describe our parallelization strategies in greater details. First, we shall discuss the performance of MPI in optimizing each parallelizable component and their strong scaling performance. Second, we perform performance benchmark and roofline analysis on optimizing a number of individual kernels, particularly the bottlenecks in our sequential implementation.

I. Parallelization with MPI

1. Read Image Data

In the original implementation, the function `GetTrainingData()` reads training data (images) sequentially, which creates a bottleneck in the program. To overcome this limitation, we can leverage MPI to parallelize the reading of the training data.

Algorithm 1 Read Training Image Data using MPI

```
 1: procedure GetTrainingData_MPI(TrainingParameters tp)
 2:     Read labels.txt file where each line contains one image's file path and its associated facial coordinate information
 3:     Calculate each process rank's corresponding lines
 4:     for each image in each rank's corresponding lines do
 5:         Load image, labeled face rectangle's x,y coordinates, and each facial landmark x,y coordinates
 6:         Store in defined DataPoint struct and store in local result
 7:     end for
 8:     Serialize local result as a character array representing data bytes for send buffer
 9:     Gather the sizes of each process's send buffer
10:     Calculate displacements based on gathered send buffer sizes for MPI Gatherv
11:     Gather serialized local result data at process rank 0 into receive buffer
12:     Deserialize receive buffer data and merge into the complete result containing full image data
13:     Broadcast the size of that complete result to all ranks
14:     Each rank prepare properly sized receive buffer
15:     Serialize the complete result on rank 0 to create the complete result buffer
16:     Broadcast the complete result buffer to all ranks
17:     Deserialize complete result buffer and update each rank's local result to have full image data
18: end procedure
```

In `GetTrainingData_MPI()`, each process reads a subset of images. The data is divided into roughly equal parts based on the number of processes.

First, the loaded training data is serialized, then, the sizes of the serialized data are gathered at the root process (process_rank 0): this aids us to use MPI_Gatherv to gather the serialized data at the root process, which then deserializes and merges all ranks' data into complete image data. Root process then broadcasts that full image data to all ranks, and finally each nonroot process deserializes the complete image data and update their local DataPoints.

2. Performing Outer Layer Regression

Very little communication is needed for the outer layer regression, as all ranks share the pixel value data and each individual rank can operate on a portion of this available set of data. A barrier is applied before the completion of the data transformation procedure so that the inner Fern regressor takes in the fully processed dataset. In conjunction with MPI, OpenMP is applied to multiple for loops within the regression for thread level parallelism.

Algorithm 2 Outer Layer Regression

```
1: Compute the amount of pixel value data corresponding to each process
2: for Every chunk of pixel value data do
3:     Perform procrustes transformation on this chunk of training data
4:     for Each training parameter 1...P (thread level parallelism with OpenMP parallel for) do
5:         Calculate the pixel position by adding offset to the initial shape of the training data
6:         Set all pixel values outside of the facial rectangle to 0
7:     end for
8: end for
9: Use MPI_AllGatherv to gather the local pixels in each rank
10: Apply MPI_Barrier and wait until all operations above finish
11: Compute pixel - pixel covariance using pixel value data
12: for Each training parameter 1...K do
13:     Fit inner Fern Regressor
14:     for Each regression target (thread level parallelism with OpenMP parallel for) do
15:         Applies a trained regression model and compute the shape difference
16:     end for
17: end for
```

3. Fitting Fern Regressor

FaceX-Train uses a two-stage regressor, where the bottom level uses Fern as base regressor. Fitting these Ferns is the primary task in the entire process of training FaceX model, and this part takes the majority of the training time from our initial profiling. As each Fern is trained to fit the residual of previous Ferns, the training of all Ferns must be done sequentially, but it is possible to accelerate the training of each individual Fern by parallel computing.

Two MPI operations are involved during the training. Each rank first selects $\frac{F}{n}$ features based on covariance feature selection and calculate the threshold for each feature, where F is the total number of feature for the regressor and n is the number of processes. Then a `MPI_Allgatherv` operation is performed to gather all features from ranks and broadcast results to all ranks. Using all F features, each process regresses $\frac{1}{n}$ of all samples to the mean shape. Regression result of all ranks are communicated via `MPI_Allreduce` operation.

Algorithm 3 Fern Regression

```
1: procedure Fern Regression(pixels value of all training sample, target $Y$)
2:     Compute the number of features each process needs to select, ideally $\frac{F}{n}$
3:     for $i = 1 ... \frac{F}{n}$ do
4:         Project regression target $Y$ onto a random vector
5:         Calculate the Covariance of the $Y$ projection
6:         for Each pixel of training sample, $j = 1 \cdots$ n_pixels do
7:             Calculate the covariance between the pixel values of all training samples and $Y$ projection
8:         end for
9:         for $j = 1 \cdots$ n_pixels do
10:            for $k = 1 \cdots$ n_pixels do
11:                Calculate the pixel - pixel correlation
12:            end for
13:        end for
14:        Select the pixel pair that has the maximal correlation as the feature $i$.
15:        Compute the threshold to use for this feature (pixel pair), based on all training samples.
16:    end for
17:    Performs MPI_Allgatherv operation to gather features and corresponding thresholds from all processes.
18:    for Each sample $i$ in a fraction of all samples, $i = \frac{\text{rank}}{n} * N \cdots \frac{\text{rank}+1}{n} * N$ do
19:        Adjust regression outputs based on shape difference between sample $i$ and targets
20:    end for
21:    Sum up the total adjustment using all samples via MPI_Allreduce
22:    Normalized adjustment and store it as regression output
23: end procedure
```

II. Parallelization with Other Methods

1. ISPC for Covariance Calculation

In the original implementation, the function `Covariance()` calculates the covariance of two arrays, and this kernel is identified to be the biggest bottleneck of our model through our initial profiling, which takes up more than 80% of the total training time. Therefore it is crucial to optimize this function and we make use of ISPC to parallelize the calculation. In this section, we describe the ISPC-based optimization of this function called `CovarianceISPC()`.

Algorithm 4 Covariance Calculation using ISPC

```
1: procedure CovarianceISPC(uniform pointers x,y to 2 const uniform double, const uniform int $size$)
2:     Initialize variables: $a$, $b$, $c$ with value 0
3:     foreach $i$ from 0 to $size$ (exclusive)
4:             Update $a \leftarrow a + x[i]$
5:             Update $b \leftarrow b + y[i]$
6:             Update $c \leftarrow c + x[i] * y[i]$
7:     end foreach
8:     Calculate $ans \leftarrow \frac{reduce\_add(c)}{size} - \left(\frac{reduce\_add(a)}{size}\right) * \left(\frac{reduce\_add(b)}{size}\right)$
9:     return $ans$
10: end procedure
```

In the `CovarianceISPC()` function, we use the `foreach` construct provided by ISPC to perform parallel execution of the loop body. The loop iterates through the input arrays x and y, updating the variables a, b, and c in parallel. After the loop, we use the *reduce_add*() function to perform a parallel reduction of the values of a, b, and c. Finally, we compute the covariance value *ans* by dividing the reduced sums and returning the result.

2. Eigen for Orthogonal Matching Pursuit

Model compression is used to store the output of ferns at the end of the outer regression layer. Fern-based regression leads to high storage cost. Hence, Orthogonal Matching Pursuit (OMP) is used. OMP is a model compression technique used to find a sparse representation of the model parameters using a learned set of basis functions. There are two inputs to the OMP function. First, it takes in a regression target, which is denoted as $\hat{y} \in \mathbb{R}^{N\times 1}$. Second, OMP chooses the best set of basis from a set of candidate basis vectors to try to best represent $\hat{y}$, this set of M candidates is denoted as matrix $B \in \mathbb{R}^{N\times M}$. The overall objective is creating $\hat{y}$ by blending different basis vectors B_j's, where j denotes the columns of B.

Q represents an upper bound on the number of non-zero coefficients in the sparse representation. This is maximal number of B_j components we can pick to represent the matrix. Higher values of Q increase the computational complexity of the OMP algorithm.

Algorithm 5 Orthogonal Matching Pursuit

```
1: procedure Orthogonal Matching Pursuit($\hat{y}, B, Q$)
2:     $residual \leftarrow \hat{y}$
3:     $std :: vector < std :: vector < double >> \mathbf{B_{sparse}}$;          ▷ Initialize an empty matrix $\mathbf{B_{sparse}}$
4:     for $i = 1$ to $Q$ do
5:         $B_j \leftarrow \arg\max_{B_{j'} \in B} |\langle residual, B_{j'} \rangle|$     ▷ Iterate over $M$ columns of $B$, find the column maximizing product w. residual
6:         Update the support $\mathbf{B_{sparse}}$, by appending new column $B_j$ to $\mathbf{B_{sparse}}$
7:         $x_i \leftarrow \arg\min_{x_i} \|\hat{y} - \mathbf{B_{sparse}} x_i\|$          ▷ Find the optimal $x_i$ where projection $\mathbf{B_{sparse}} x_i$ minimizes norm
8:         $residual \leftarrow residual - \mathbf{B_{sparse}} x_i$
9:     end for
10:    return $\hat{S}$ where $\hat{s}_i = x_i$ (or 0)          ▷ Return a matrix of the $x_i$'s
11: end procedure
```

In the above code, matrix $\mathbf{B_{sparse}} \in \mathbb{R}^{N\times i}$, where i is the current iteration count, vector $\hat{y} \in \mathbb{R}^{N\times 1}$, vector $x_i \in \mathbb{R}^{i\times 1}$, vector $residual \in \mathbb{R}^{N\times 1}$

First, we provide a high level overview of the algorithm and discuss the number of FLOPS required.

1. **Basis vector selection** OMP is a greedy algorithm that builds the support of $\hat{y}$ incrementally. The outer iteration from 1 to Q incrementally appends the new column B_j to the matrix $\mathbf{B_{sparse}}$. Inside this outer iteration, the inner iteration loops through the M columns of B to finding the basis B_j (the column of B) that has the greatest inner product with the residual vector. As both B_j and *residual* are vectors of length N, the dot product performs $2 * N - 1$ FLOPS. We

perform iteration over a total of Q outer loops and M inner loops, these operations take $(2*N-1)*M*Q$ FLOPS in total. An additional $2*M*Q$ FLOPS are incurred in taking the absolute value and traversing across M dimensions to find the optimal basis. Hence the total number of FLOPS is $\mathbf{(2*N-1)*M*Q+2*M*Q}$. $\hat{y}$, B and Q take $\mathbf{N+NM+1}$ read operations. Creation of B_j and storing max value and index takes $\mathbf{QMN+2QM}$ write operations.

2. **Solving Linear System** For every iteration i, we find the optimal value $x_i = \left(\mathbf{B_{sparse}}^\top \mathbf{B_{sparse}}\right)^{-1} \mathbf{B_{sparse}}^\top \hat{y}$. As matrix inversion is an expensive operation, the OpenCV native baseline implementation uses SVD to solve this linear system, taking inputs $\mathbf{B_{sparse}}^\top \mathbf{B_{sparse}}$ and $\mathbf{B_{sparse}}^\top \hat{y}$. First, we compute the number of FLOPS needed to compute these two input matrices. Let the current iteration number be $i \in (1, Q)$, as the dimension of $\mathbf{B_{sparse}}^\top \in \mathbb{R}^{i\times N}$. In this iteration, the multiplication of $\mathbf{B_{sparse}}^\top \mathbf{B_{sparse}}$ takes $(2N-1)i^2$ FLOPS. Since we loop through all values of i from 1 to Q, we use the summation identity $\sum_{r=1}^{n} r^2 = (n/6)(n+1)(2*n+1)$, the total number of FLOPS is $\mathbf{(2N-1)(Q/6)(Q+1)(2*Q+1)}$. Creation of this matrix takes $\mathbf{N^2}$ write operations. Next, we compute $\mathbf{B_{sparse}}^\top \hat{y}$, noting that $\hat{y} \in \mathbb{R}^{N\times 1}$. For a single iteration this operation takes $(2*N-1)*i$ flops. Since we loop through all values of i from 1 to Q, we use the summation identity $\sum_{r=1}^{n} r = (n)(n+1)/2$, hence these operation take $\mathbf{(Q)(Q+1)/2*(2*N-1)}$ FLOPS. Creation of this vector takes $\mathbf{(Q)(Q+1)/2}$ writes. After computing the two input matrices, we carry out SVD. The number of FLOPS required in this step is difficult to analyze analytically. Common literature shows that this value is empirically approximated as $13*i^3$ (see link: http://mitran-lab.amath.unc.edu/courses/MATH547/lessons/Lesson25.pdf). Since we permute all values of i from 1 to Q, we again use the summation identity $\sum_{r=1}^{n} r^3 = 0.25*(n^4+2*n^3+n^2)$ the total number of FLOPS is $\mathbf{13*0.25*(Q^4+2*Q^3+Q^2)}$. Writing the result of SVD takes $\mathbf{(Q)(Q+1)/2}$ writes.

3. **Recomputing the Residual** The final step, we update the value of the $residual \leftarrow residual - \mathbf{B_{sparse}} x_i$. First, for every iteration, the multiplication operation takes $N(2i-1)$ FLOPS. Hence, across all iterations, this totals $((Q)(Q+1)/2-1)N$ FLOPS. The subtraction operation across all iterations takes another $N*Q$ flops. Hence, the total number of FLOPS is $\mathbf{((Q)(Q+1)/2-Q)N+N*Q}$. Writing the residual vector with new values take $\mathbf{QN}$ operations.

In our optimized kernel, we use the Eigen library instead of the native OpenCV implementation. We hypothecate Eigen to yield better performance. On a high level, the Eigen library has several advantages. First, it utilizes compiler optimizations such as loop unrolling, vectorization, and instruction-level parallelism. Second, Eigen is designed to maximize cache utilization, which is crucial for performance, optimizing memory access patterns to take advantage of the cache hierarchy and minimize data movement. In utilizing this library, we expect to observe evidence of better cache utilization and a reduction in total run time. Eigen's expression templates allows it to intermediate temporaries, improving data locality.

In particular, we look into code snippets of direct comparison between the two implementations and why one may be more efficient than the other.

Snippet 1 - Eigen reduces wall time

In line 5 of the algorithm pseudo-codes, the inner product is performed as:

```
\\ Native OpenCV implementation
double current_value = abs(static_cast<cv::Mat>(residual.t() * base.col(j)).at<double>(0));
\\ Eigen Implementation
Eigen::MatrixXd column = baseEigen.col(j);
double current_value = (residual.transpose() * column).cwiseAbs().sum();
```

First, the Eigen implementation, 'cwiseAbs()' and 'sum()' functions are lazy evaluations whoes computation can be postponed until they are absolutely necessary.

Second, across the Q iterations, we incrementally append the selected column B_j together to construct matrix B_{sparse}. We are extracting the column B_j QM times in total, and performing the appending to B_{sparse} operation Q times. As a result of these frequent column accesses, Eigen is more efficient as it uses column-major order by default for storing matrices, meaning that B_j is stored in a continuous block of memory. Our read / write calculations above made the simplifying assumption of infinite cache sizes. In reality, this asssumption is violated and OpenCV's row major order leading to poor spatial locality, higher cache misses and increased memory latency.

3. Intel Intrinsics for Procrustes

In the original implementation, the function `Procrustes()` computes the optimal linear transformation between two sets of points through rotation, translation, and possibly scaling. This function is called with multiple functions including `MeanShape()` and `Regress()`. In this section, we describe the Intel Intrinsics based optimization of this function called `ProcrustesV()`.

The original kernel takes in two `Point2d` objects, a class defined in the **OpenCV** library that represents two double values in Cartesian coordinate, as input and returns the transformation vector. It performs a series of additions, subtractions and multiplications on various combinations of the x and y coordinates of the input points. It then constructs a 4×4 matrix and a 4×1 matrix using the computed values, and computes the transformation vector via matrix inversion and multiplication. For the optimized `ProcrustesV()` function, we parallelized the computation in this kernel by using AVX2 and loop unrolling. We decide on this approach because this function mainly involves a loop that performs a series of algebraic operations, which can be easily vectorized using AVX2 instructions. `_mm256_add_pd`, `_mm256_mul_pd`, and `_mm256_sub_pd` are the three main instructions of which we take advantage.

4. Intel Intrinsics for Computing Threshold Inside Fern Regression

In addition to splitting work among multiple processes using MPI operation, we also exploit DLP for specific kernels inside Fern regression to optimize each process.

After selecting the i^{th} feature, Fern also needs to calculate the threshold for this feature. This kernel essentially performs element-wise subtraction on two double arrays, and find the maximum difference. The kernel loops through two arrays with size `n_training_sample * augment_factor`, and perform 1 subtraction and 2 comparison on each element. The total number of operations is then $3 * N *$ `augment_factor`, giving operational intensity 3/16 flop/Byte. We utilize vectorization to exploit this data structure for optimization. To minimize overhead, we use Intel Intrinsics on this kernel. `_mm256_sub_pd`,`_mm256_min_pd`, and `_mm256_max_pd` are the three major instructions used here.

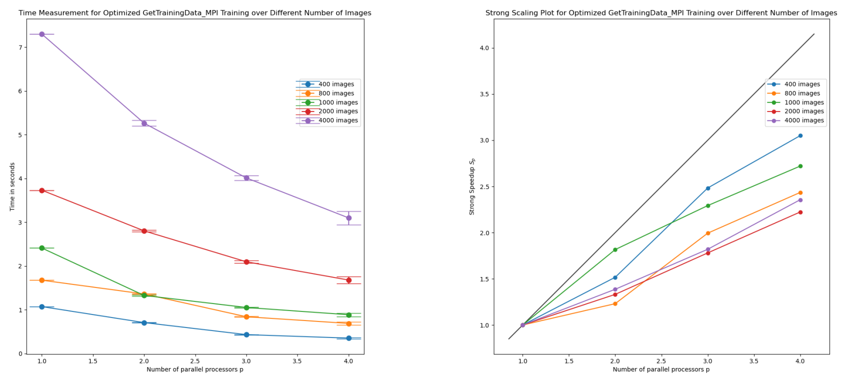
Part 2. Validation, Verification

To ensure the numerical consistency of individually optimized kernels, we write multiple tests to see whether the results produced by the optimized functions are identical to those of the original. For the kernels that mostly performs matrix operations, we generate random input data to feed to the kernel. For example, for the `Procrustes()` function, we compared the transformation vectors returned by the two functions on the same set of randomly generated `Point2d` data points. To check for the consistency of our MPI implementations, we feed both the optimized and the sequential models the same testing image and compare the predicted landmarks and face boundaries they produced. Our parallel implementation indeed produce the same face landmark labelled images as the baseline sequential model.

4 Performance Benchmarks and Scaling Analysis

I. Strong Scaling analysis for MPI kernels

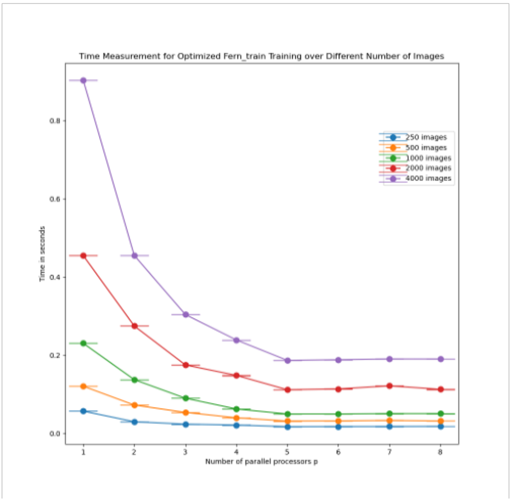
To determine the performance of our MPI kernels on the Intel Xeon E5-2683 16 core CPU, we have run our individual MPI model using varying sizes of datasets of images and calculated their respective T_p. Dividing the serial time T_s against T_p we obtain Speedup S_p for different counts of total process ranks. We have tested running with different number of nodes/processes and plotted time measurement and strong scaling graphs for the MPI optimized implementation kernels in Section 3. From Figure 1 and Figure 2 we observe that for larger image sizes, MPI parallelization will more significantly reduce the processing time.



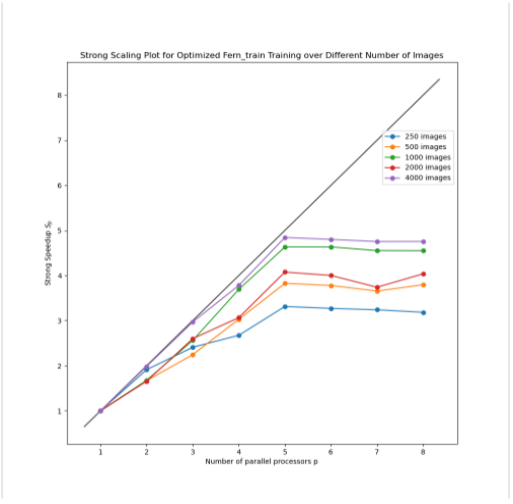
(a) Time measurement for GetTrainingData_MPI function

(b) Strong scaling plot for GetTrainingData_MPI function

Figure 1: Time measurements and strong scaling plots for GetTrainingData_MPI function

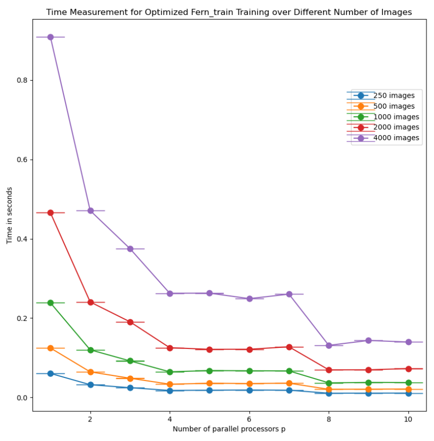


(a) Time measurement for fern regression

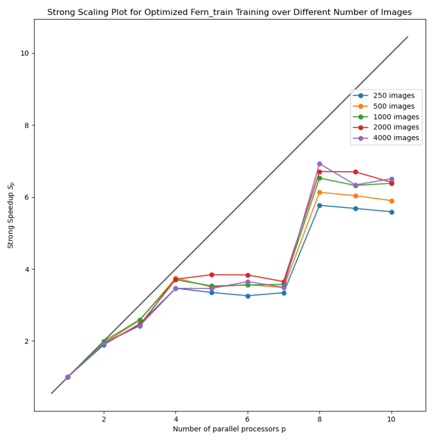


(b) Strong scaling plot for fern regression

Figure 2: Time measurements and strong scaling plots for fern regression with 5 features



(a) Time measurement for fern regression



(b) Strong scaling plot for fern regression

Figure 3: Time measurements and strong scaling plots for fern regression with 8 features

One important observation we want to point out is that for the fern regressor, the performance starts to plateau once a certain number of parallel processes has been reached. We observe that the plateau happens when the number of parallel processes is the same as the number of features we selected for the fern regression. We experimented with 5 features and 8 features and found that the performance of the fern regressor starts to level off at 5 and 8 parallel processes, respectively, as shown in Figure 2 and Figure 3. The reason for this will be explained in section 5.

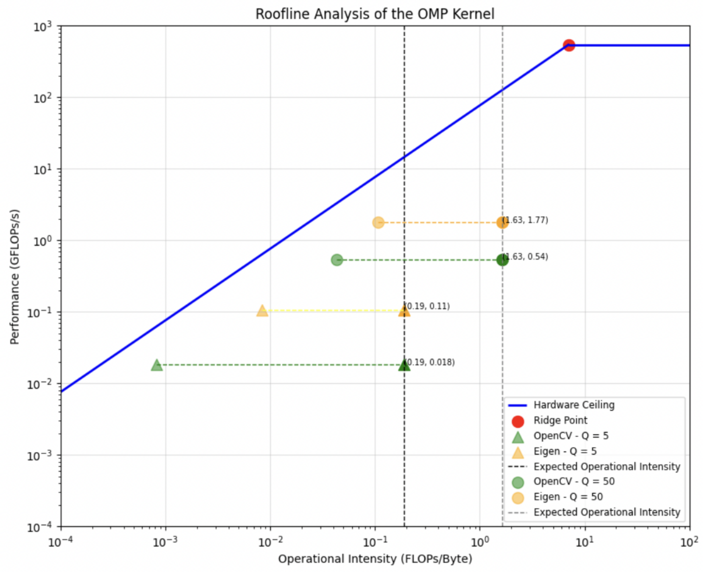
II. Roofline and Scaling Analysis for Other Kernels

We then performed roofline analysis for each of the individual kernels using other parallelization methods. We also measured the runtimes of the original and optimized kernel with different input dimensions to see how well our optimized kernels scale to varying data sizes.

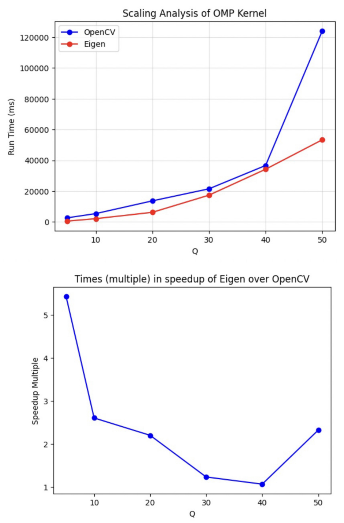
1. Below shows results of the roofline and scaling analysis of the Orthogonal Matching Pursuit (`OMP()`) Kernel.

   - As we increase the value Q as well as scaling the value N simultaneously in the same order. The authors of **FaceX-Train** establishes that $N = 2Q$ and sets $M = 512$. Whilst we are able to compute the FLOPs and total load store operations by suming up the calculations made in the previous section, substituting Q, M and N. These are plotted as darker colored circles (for $Q = 50$) and triangles, (for $Q = 5$). Additional to the roofline analysis, we record the total cycles, total instructions, using PAPI. While the number of read / write calculations we performed before were under the assumption that the cache size was infinite, we also record the total load stores estimates given by PAPI. We deviate from the theoretical value (faint circles and triangles)

- The Eigen implementation yields superior performance. This is likely due to the exploitation of efficient memory access patterns and column major storage. The hypothesis posited in the previous optimization section rings true. We observe significantly lower overall number of L1 - data cache misses associated with the Eigen implementation, for instance, total L1 cache misses for $Q = 5$ with OpenCV is 26327, whereas total L1 cache misses for $Q = 5$ with the Eigen library is 7554. Additionally the Eigen library better exploits vectorization and SIMD as well as lazy evaluations, leading to overall wall time reduction.
- As we increase the Q (and correspondingly, N), the difference between the performance between the two implementations shrinks. When $Q = 5$, Eigen had 6 times higher performance than OpenCV. When $Q = 50$, this decreases to about 3 times. This potentially suggest, as the problem size grows larger, the time taken to solve the linear system dominates while Eigen's advantage in column major order fades, reducing the performance gap.



(a) Roofline Analysis

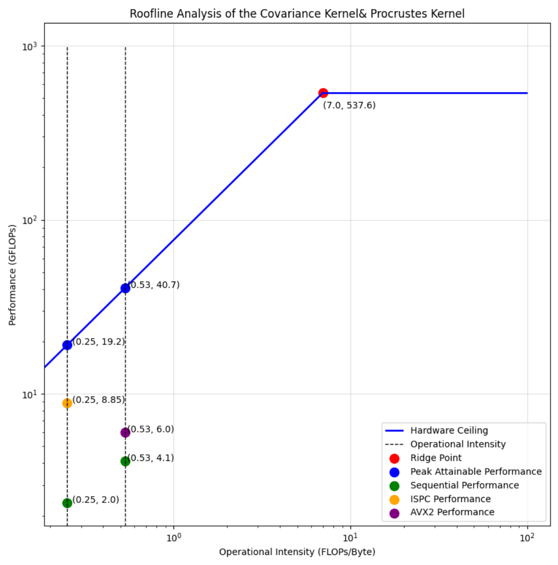


(b) Scaling Analysis

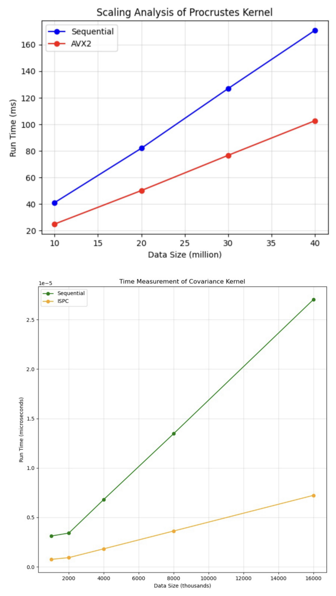
Figure 4: Performance Analysis of the OMP Kernel

	Total Cycles		**Total Instructions**		**Instructions Per Cycle**		**Total Load/Store**	
Q	Eigen	OpenCV	Eigen	OpenCV	Eigen	OpenCV	Eigen	OpenCV
5	1255996	8461569	1970265	14356939	1.56869	1.696723	810050	8250574
10	6693214	17357234	9608980	29768743	1.43563	1.715063	4381939	16933256
20	20235997	44440877	20848926	62083941	1.030289	1.397001	9322681	34627355
30	56335694	70204670	35720274	98404818	0.6340611	1.401685	15242770	53520937
40	111047054	111472485	53808581	140869635	0.4845566	1.263717	22251444	74266317

2. We have also computed performances and operational intensities for sequential and vectorization optimized Covariance and Procrustes kernels. We have plotted them in Figure 5.



(a) Roofline Analysis



(b) Scaling Analysis

Figure 5: Performance Analysis of the Procrustes and Covariance Kernels

III. Integrating All Optimized Kernels

We integrated all the optimized kernels to obtain the parallelized **FaceX-Train** model. We measured its wall time with the same training parameters and images, and compared it to the baseline. As a result, for the training size of 4000 images and 5 features, we see that optimized training time is only 45.6% of the baseline, and thus achieves a 2.2x speedup in training.

In Table 1, we also provided some key figures of the jobs that we ran during the development of this project.

	Sequential	Parallelized
Typical wall clock time (seconds)	200.51	91.52
Typical job size (nodes)	1	5
Memory per node (GB)	64	64
Maximum number of input files in a job	4000	4000

Table 1: Workflow parameters of the sequential and parallelized cases used during project development.

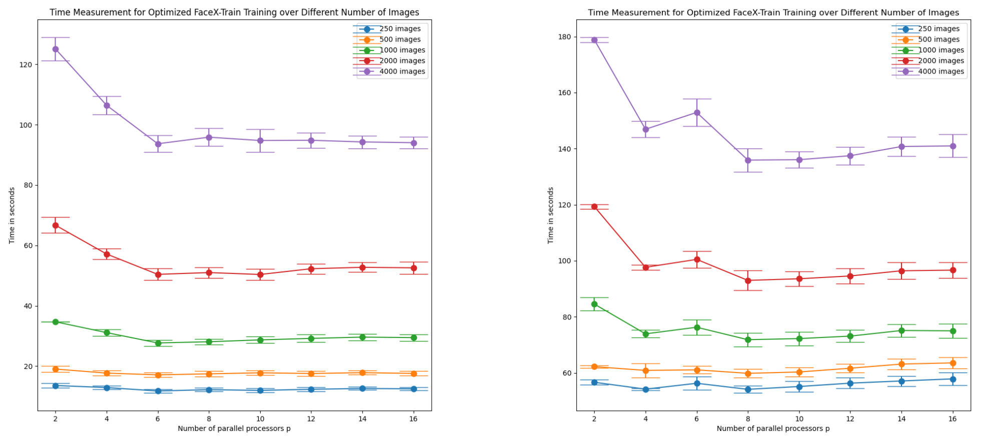
Process Count	**1**	**2**	**3**	**4**	**5**	**6**	**8**	**10**	**12**	**14**	**16**	**baseline**	**speedup**
250 Images	14.70	13.56	13.02	12.89	11.71	11.74	12.19	12.10	12.22	12.31	12.21	51.54	4.41
500 Images	21.31	19.01	18.01	17.67	16.32	17.09	17.43	17.74	17.77	17.60	17.54	59.85	3.67
1000 Images	44.51	34.68	31.32	31.10	27.40	27.66	28.06	28.68	29.18	29.41	29.44	81.93	2.99
2000 Images	84.26	66.70	58.75	57.14	49.52	51.93	50.40	51.93	50.99	52.23	52.57	120.86	2.44
4000 images	158.72	125.12	113.11	106.43	91.52	93.68	95.86	94.76	94.15	94.68	94.04	200.51	2.19

Table 2: Time (s) Taken to process various input file quantities

Given that most of the components within each regressor are parallelizable, we would like to keep exploring the possibilities of further optimizing the **FaceX-Train**. Some of the future works may include using the MPI to parallelize the `MeanShape()`, `ArgumentData()`, and `CreateTestInitShapes()` functions, which computes the mean shape of all training data, augments the images, and creates initial test shapes, respectively. Additionally, we have only touched upon the training part of the **FaceX** program so far. We would also like to extend our work to optimizing the real-time recognition components of the algorithm.

5 Resource Justification

As we increase the rank/node count from 1 to 16 with the number of features set to 5, the optimal rank occurs at 5 across all sizes of datasets. The speedup acheived with MPI is higher for smaller image datasets. As shown in Figure 1 the image read time for 16 ranks is 1.56 seconds while for 1 rank is 9.32 seconds, a 6 times reduction in read kernel runtime. MPI functions may be also useful for non-time related purposes, for instance reading large images when each node only has a limited memory.



(a) Time measurement for Optimized FaceX-Train (F = 5)   (b) Time measurement for Optimized FaceX-Train (F = 8)

Figure 6: Optimal Node Finding for Different Feature Count

Based on Figure 6, we see that for a training size of 4000 images with 5 features, the optimal job size of the representative benchmark is 5 nodes. With the corresponding wall time of 91.52 seconds, we compute the node hours to be 0.128 hours.

The optimal number of processors plateaus at 5 as expected. The reason is that fitting Fern regressors takes up the majority of total training time, so that the time required to train each individual Fern significantly affected the overall performance. During the training process of each individual Fern, the feature selection process is the most computing-intensive step, so we split up the work to multiple processes and share selected features among all processes. To verify our hypothesis that the number of nodes we should provision depends on fern features count, we additionally examined overall run time with 8 features as shown in Figure 6. We observe that 8 nodes is optimal universally across all input dataset sizes. Hence, we may indeed conclude that the optimal number of nodes equates the number of fern regressors. Using N processes for N features is the optimal solution as either fewer or more nodes would cause workload imbalance, and using more processes also induces larger communication overhead. Note that this number of features used by Fern is a tunable hyper-parameter. Using more features enhances the model accuracy but may take longer to train. Exploiting the scheme of parallel computing, our project is able to get more work done in approximately the same amount of time, given more computing resource (nodes) available. Per computing resource granted, we would be able to select more features for each Fern, and thus achieves higher model accuracy.

Here we provide a table of multiple test cases and their corresponding required resource to train the model with 5 features.

	Test case A	Test case B	Test case C	Test case D	Test case E
training size	250	500	1000	2000	4000
total node hours	0.016	0.023	0.038	0.069	0.128

Table 3: Justification of the resource request

References

[1] Cao X, Wei Y, Wen F, et al. *Face alignment by explicit shape regression[J]*. International Journal of Computer Vision, 2014, 107(2): 177-190.