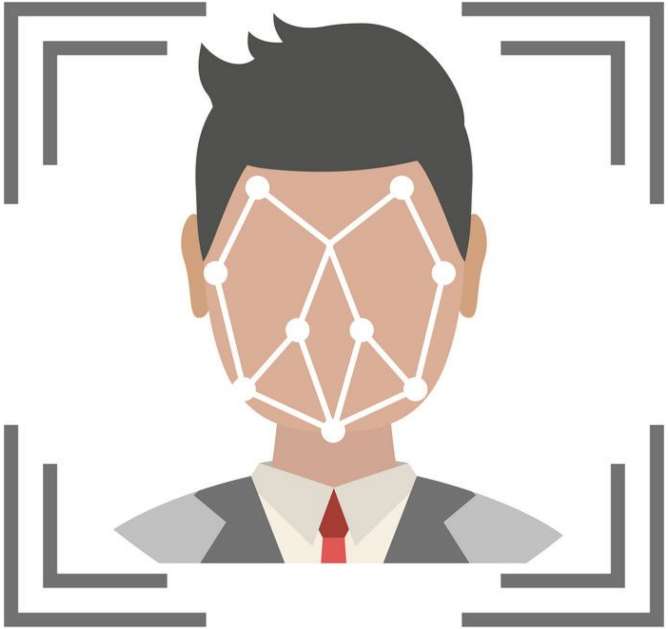
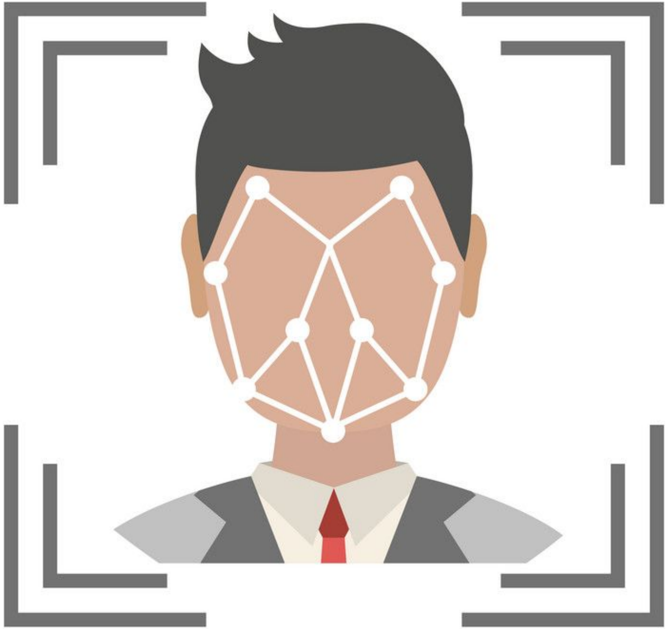
Parallelizing Gradient Boosted Regression for Facial Landmark Recognition

Zhecheng Yao, Yixian Gan, Rebecca Qiu, Lucy Li
May 2023



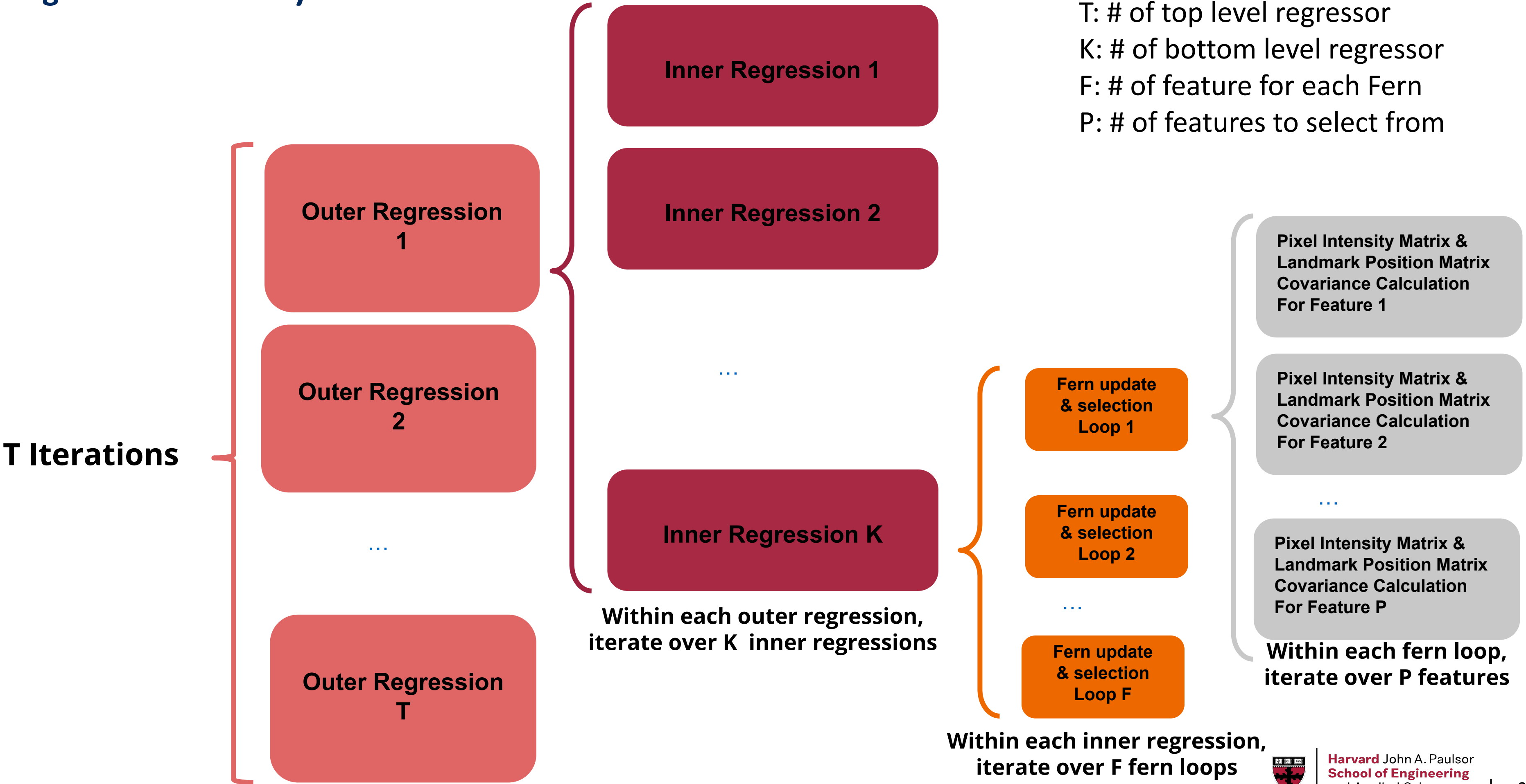
Harvard John A. Paulson
School of Engineering
and Applied Sciences

Contents



Regression Hierarchy

T: # of top level regressor
K: # of bottom level regressor
F: # of feature for each Fern
P: # of features to select from



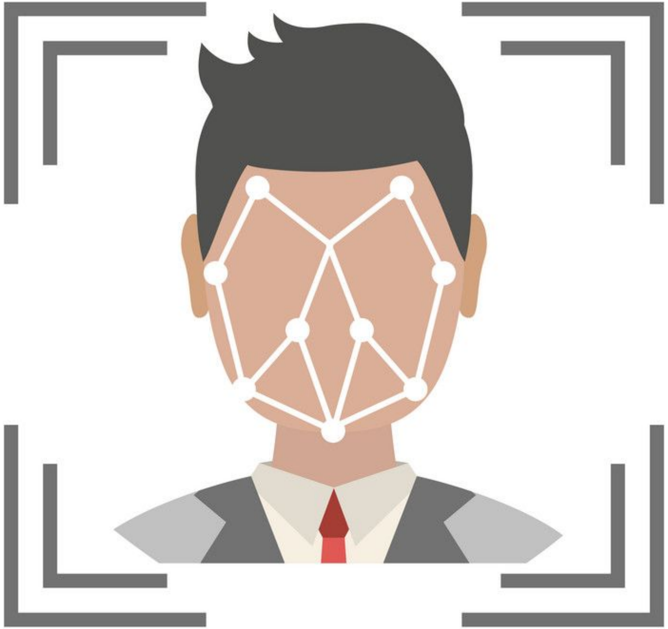
Sequential Baseline

Baseline → **Training Time ~ 200s**

Data Size: 4000 images
Image Dimension: 250x250
CPU: Intel Xeon E5-2683 v4
Number of Regressors: 1

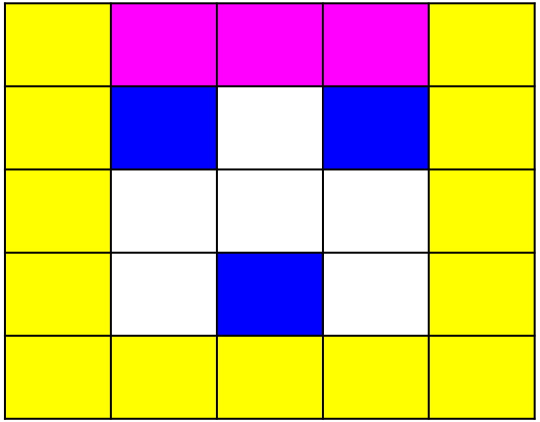
Objective → **Increase training efficiency by implementing parallelism.**

Contents

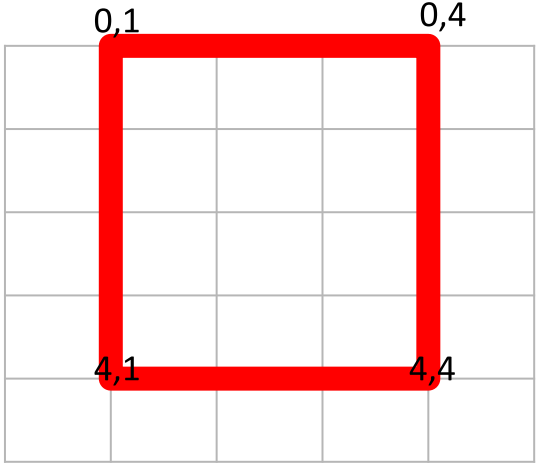


Get Training Data Sequential

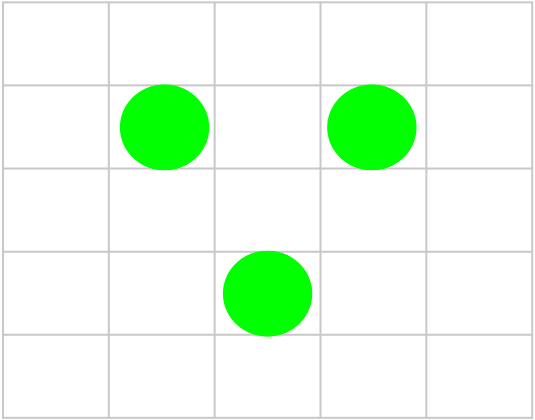
Work Logic



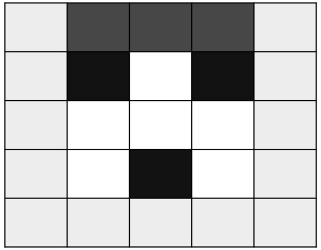
colored image of any size



four coordinates of face rectangle



facial landmark coordinates
(same landmark count for all input images)



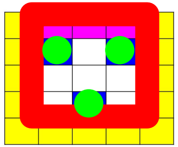
grayscale matrix

vector { (0,1), (0,4), (4,1), (4,4) }

vector { (1.5,1.5), (2.5,3.5), (3.5,1.5) }

stored under a “DataPoint” struct
Then all DataPoints are stored in a vector of DataPoints

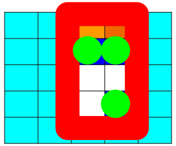
Get Training Data MPI



Rank 0 Image read

DataPoint 0

serial code has m regressor
each regressor need all S images' data
for MPI code with rank count n
we still want m regressors
each train with S images
-> need communication to share image data



Rank 1 Image read

DataPoint 1

Rank 2 ...

...

grayscale matrix 0

face rectangle vector 0

landmark vector 0

...

communication
to distribute partial read data

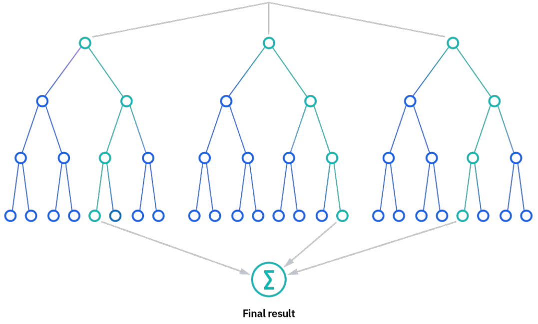
grayscale matrix 1

face rectangle vector 1

landmark vector 1

...

regressor



Design choice:
regressor parallelism
on feature-level
not data-level

Design Breakdown

Step 1 Each Rank Load Assigned Images

- Read info text file
- Text lines w/ data paths assigned to each rank
- Each rank do parallel read in of images, face rectangle & facial landmarks

Step 2 Gather All Images to Rank 0

- Serialize local DataPoints into array for send buffer
- Gather sizes of each rank's send buffer to rank 0 and calculate displacements
- Gatherv collect all image data
- Deserialize rank 0 receive buffer into complete DataPoints

Step 3 Rank 0 Scatter Images to all ranks

- rank 0 serialize complete DataPoints into array for send buffer
- Broadcast send buffer size to all ranks so each rank can prepare properly sized receive buffer
- Broadcast complete DataPoints
- Each rank deserialize complete DataPoints and update DataPoints

DataPoint 1

grayscale matrix

$$\begin{bmatrix} 150 & 100 & 100 & 100 & 150 \\ 150 & 50 & 255 & 50 & 150 \\ 150 & 255 & 255 & 255 & 150 \\ 150 & 255 & 50 & 255 & 150 \\ 150 & 150 & 150 & 150 & 150 \end{bmatrix}$$

face rectangle vector

{ (0,1), (0,4), (4,1), (4,4) }

landmark vector

{ (1.5,1.5), (2.5,3.5), (3.5,1.5) }

DataPoint 2, 3…

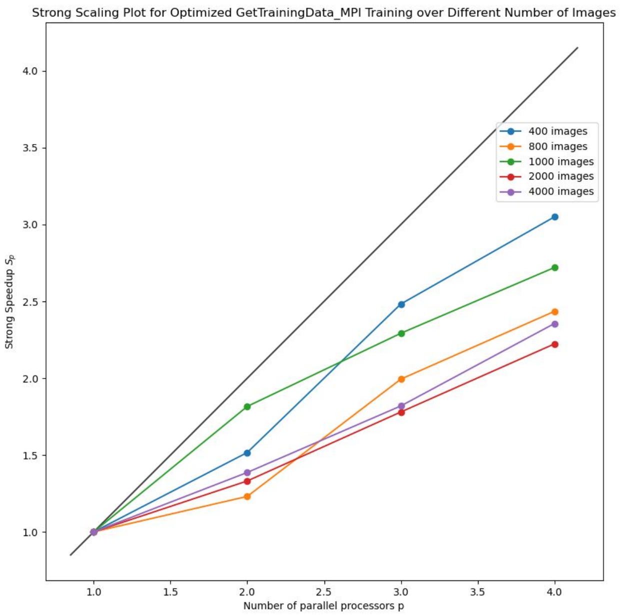
Serialize

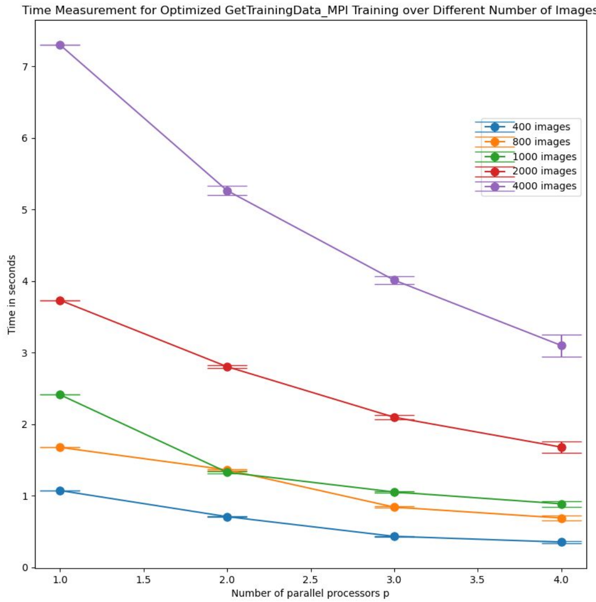
Deserialize

Send/Recv Buffer

{ 5, 5, // matrix size
150 , 100, 100…. // first row in matrix
150, 50 …. // second and other rows
4, 2 // face rectangle vector size, each has x&y
0, 1, 0, 4 …. // face rectangle vector values
3, 2, 1.5, 1.5 …. // similar logic for landmark vector …}

Parallel Result





- Highlights
  - More ranks provide better Speedup (~2-3x for 1-4 ranks)
  - More Images take longer time to process
  - Serialization/deserialization takes toll on performance

Regression Hierarchy

T Iterations

Outer Regression 1

Outer Regression 2

...

Outer Regression T

Inner Regression 1

Inner Regression 2

...

Inner Regression K

Within each outer regression, iterate over K inner regressions

Fern update & selection Loop 1

Fern update & selection Loop 2

...

Fern update & selection Loop F

Within each inner regression, iterate over F fern loops

Pixel Intensity Matrix & Landmark Position Matrix Covariance Calculation For Feature 1

Pixel Intensity Matrix & Landmark Position Matrix Covariance Calculation For Feature 2

...

Pixel Intensity Matrix & Landmark Position Matrix Covariance Calculation For Feature P

Within each fern loop, iterate over P features

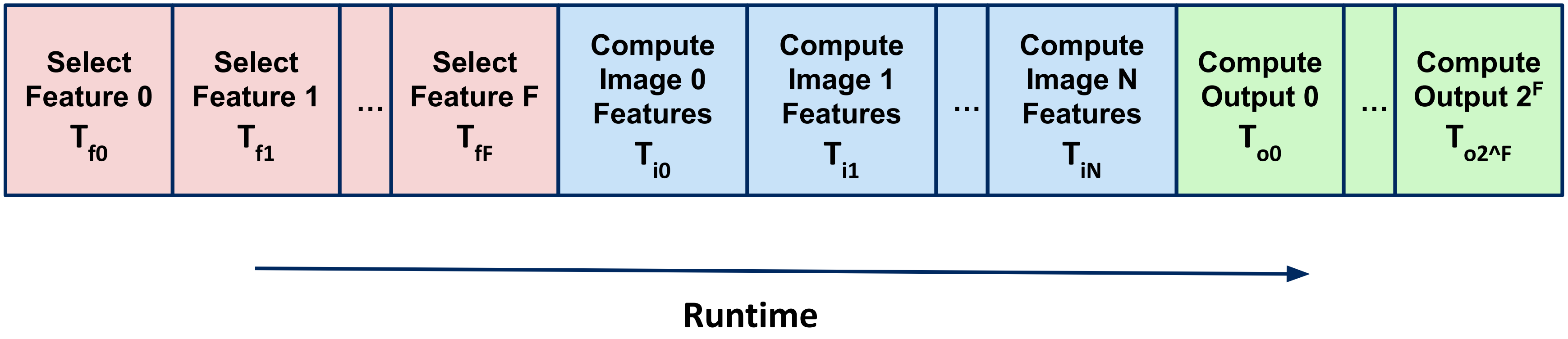
Fern Regress Sequential

T_{f0}= Time to select feature 0

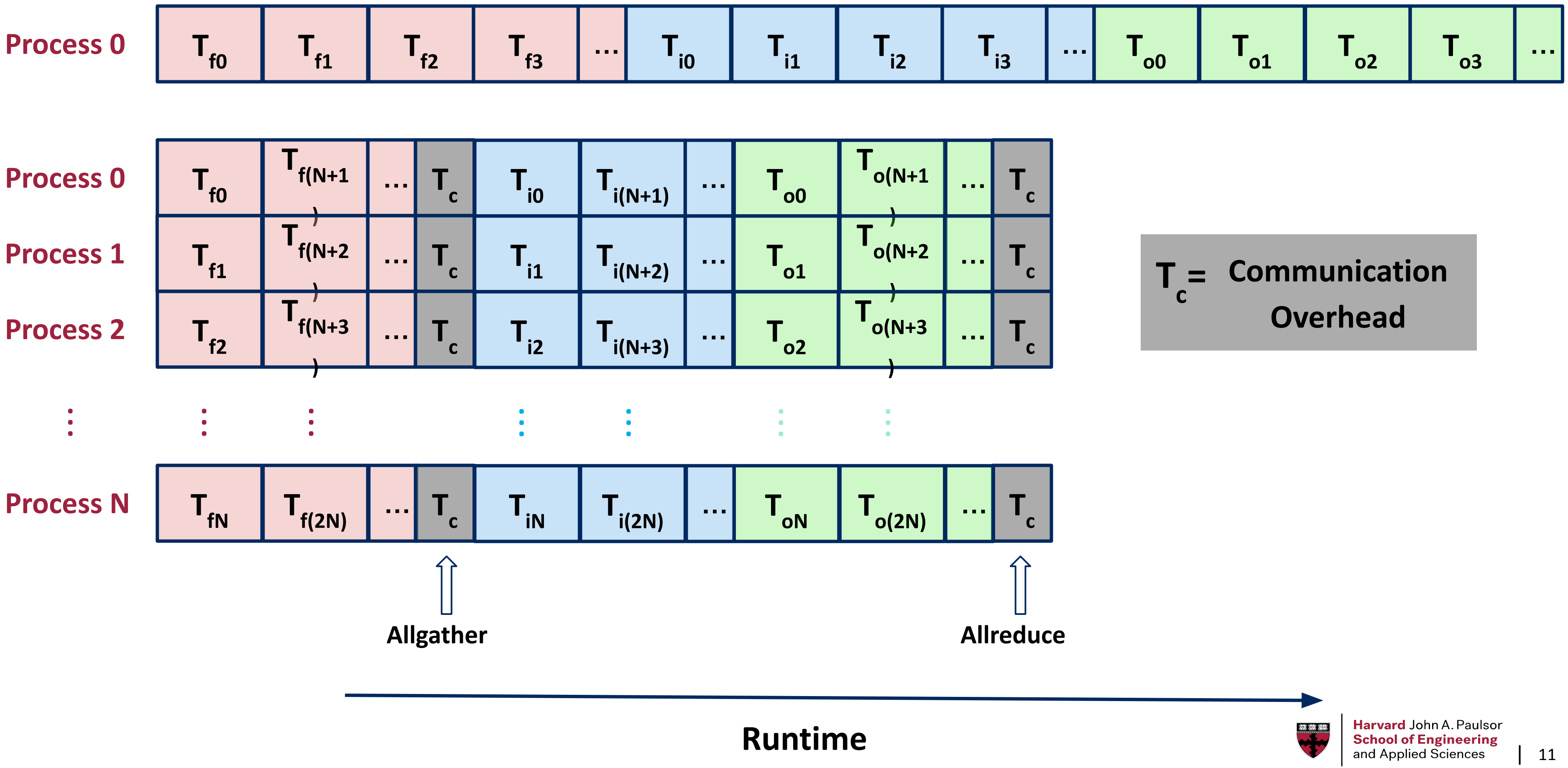
T_{i0}= Time to compute feature of image 0

T_{o0}= Time to compute outputs 0

Process 0

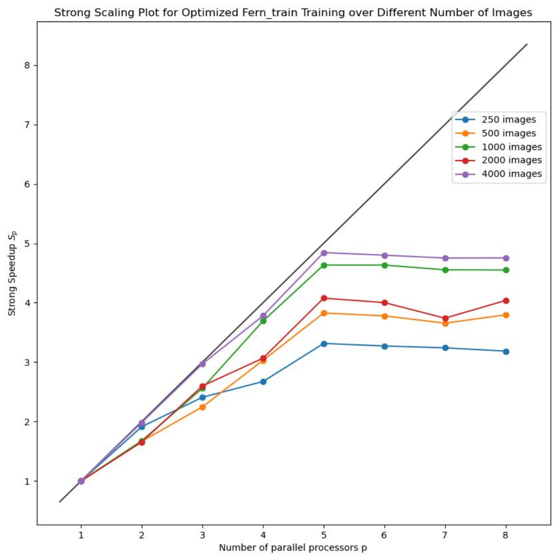


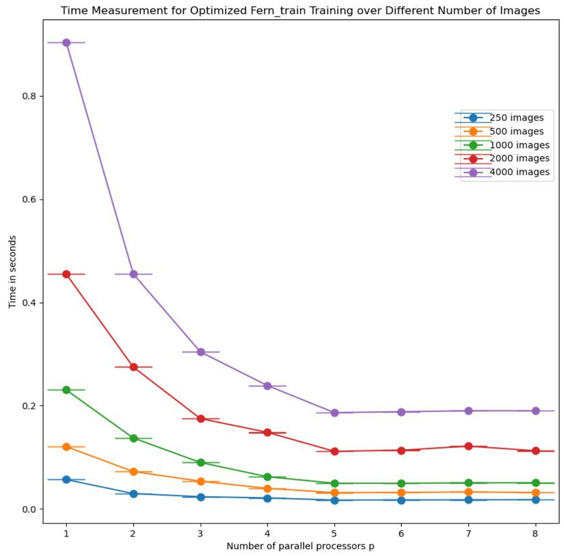
Fern Regress MPI



Fern Regress MPI

Parallel Result



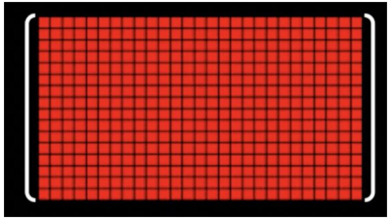


- Highlights
  - More ranks give better performance up to a threshold
  - Larger improvements for larger problem size (# of training images)
  - Efficiency depends on training hyper-parameter

Orthogonal Matching Pursuit - OpenCV vs Eigen

Target: Reconstructing of noisy result vector from the Fern regressions $\hat{y} \in \mathbb{R}^{N\times 1}.$

1. **Samples a matrix** B

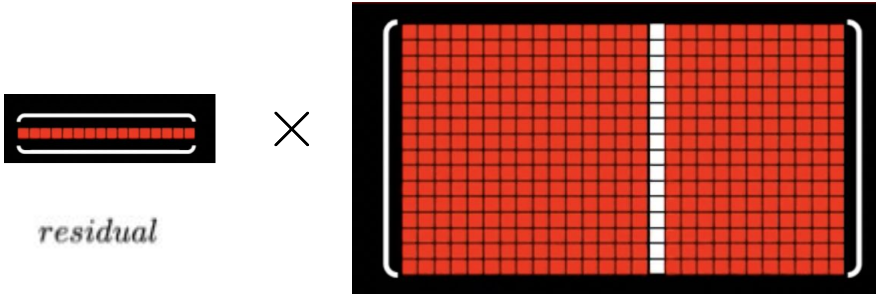


$B \in \mathbb{R}^{N\times M}$

2. $residual \leftarrow \hat{y}$

3. **OMP chooses the best** Q **candidate basis vector in matrix** B **maximizing dot product (Greedy)**

$$B_j \leftarrow \arg\max_{B_{j'} \in B} |\langle residual, B_{j'} \rangle|$$



$residual$

B_j

- **FLOPS:**
  (N multiplications + N - 1 additions) M (columns) Q (iterations)

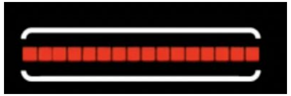
Best Basis Vector Selection

Orthogonal Matching Pursuit - OpenCV vs Eigen

3

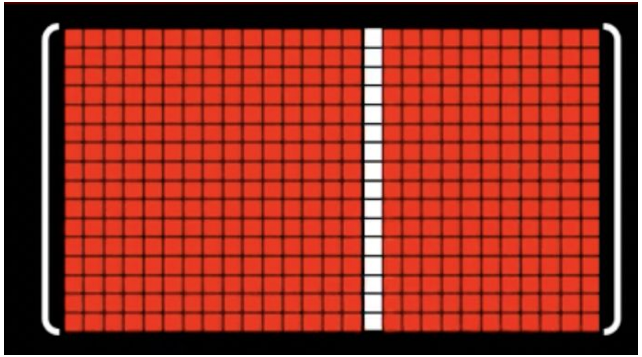
$$B_j \leftarrow \arg\max_{B_{j'} \in B} |\langle residual, B_{j'} \rangle|$$

```
\\ 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();
```



$residual$

$\times$



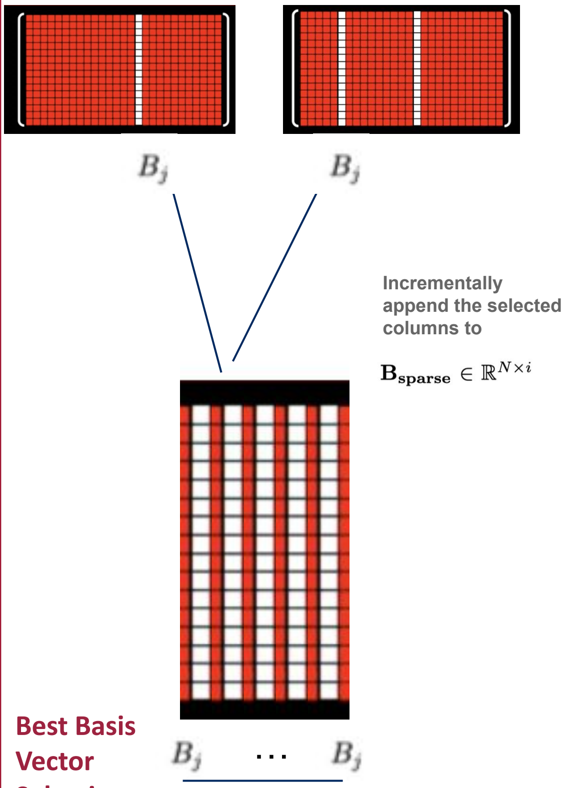
B_j

- **FLOPS:**
  (N multiplications + N - 1 additions) M (columns) Q (iterations)

Best Basis Vector Selection

Orthogonal Matching Pursuit - OpenCV vs Eigen

3



Best Basis Vector Selection

Q (total iterations)

4

Dimensions

$$\mathbf{B}_{\mathbf{sparse}}{}^{\top} \in \mathbb{R}^{i \times N}$$

$$\mathbf{B}_{\mathbf{sparse}} \in \mathbb{R}^{N \times i}$$

$$\hat{y} \in \mathbb{R}^{N \times 1}$$

Optimization Problem

$$x_i \leftarrow \arg\min_{x_i} \|\hat{y} - \mathbf{B}_{\mathbf{sparse}} x_i\|$$

$$x_i = \left(\mathbf{B}_{\mathbf{sparse}}{}^{\top} \mathbf{B}_{\mathbf{sparse}}\right)^{-1} \mathbf{B}_{\mathbf{sparse}}{}^{\top} \hat{y}.$$

	FLOPS in the i-th iteration	FLOPS in the Q total iterations
$\mathbf{B}_{\mathbf{sparse}}{}^{\top} \mathbf{B}_{\mathbf{sparse}}$	$(2N-1)i^2$	$(\mathbf{2N}-\mathbf{1})(\mathbf{Q}/\mathbf{6})(\mathbf{Q}+\mathbf{1})(\mathbf{2}*\mathbf{Q}+\mathbf{1})$
$\mathbf{B}_{\mathbf{sparse}}{}^{\top} \hat{y}$	$(2*N-1)*i$	$(\mathbf{Q})(\mathbf{Q}+\mathbf{1})/\mathbf{2}*(\mathbf{2}*\mathbf{N}-\mathbf{1})$
SVD	$13*i^3$	$\mathbf{13}*\mathbf{0.25}*(\mathbf{Q}^4+\mathbf{2}*\mathbf{Q}^3+\mathbf{Q}^2)$

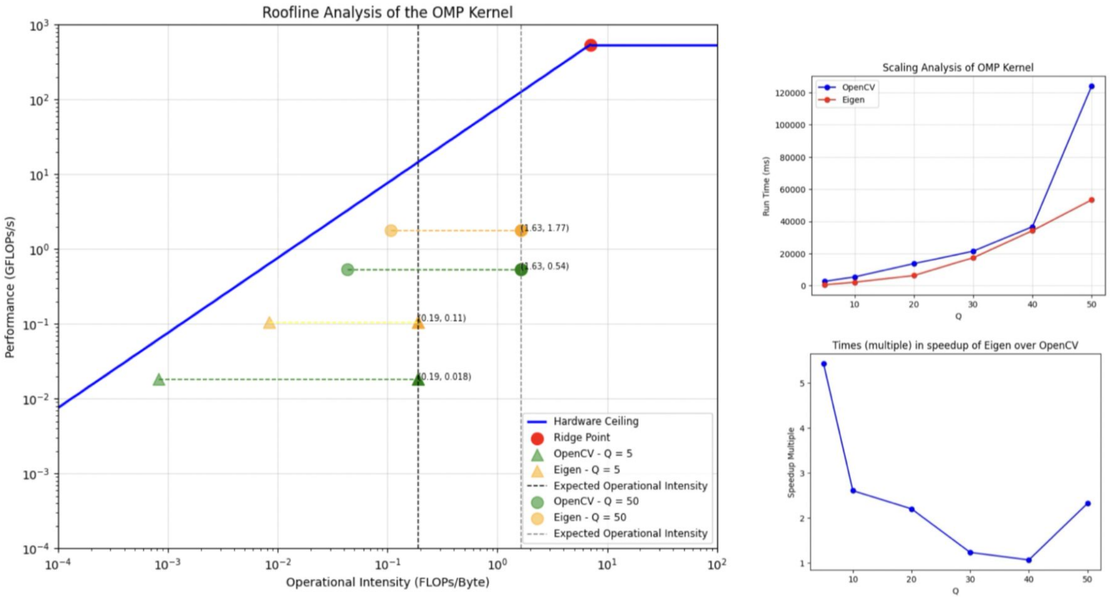
Solving the Linear Systems

5

FLOPS in the Q total iterations

$$residual \leftarrow residual - \mathbf{B}_{\mathbf{sparse}} x_i \qquad ((\mathbf{Q})(\mathbf{Q}+\mathbf{1})/\mathbf{2}-\mathbf{1})\mathbf{N}+\mathbf{N}*\mathbf{Q}$$

Orthogonal Matching Pursuit - OpenCV vs Eigen



(a) Roofline Analysis (b) Scaling Analysis

Figure 4: Performance Analysis of the OMP Kernel

- Total L1 cache misses for Q = 5 with OpenCV is **26,327**
- Tota L1 cache misses for Q= 5 with the Eigen library is **7554**

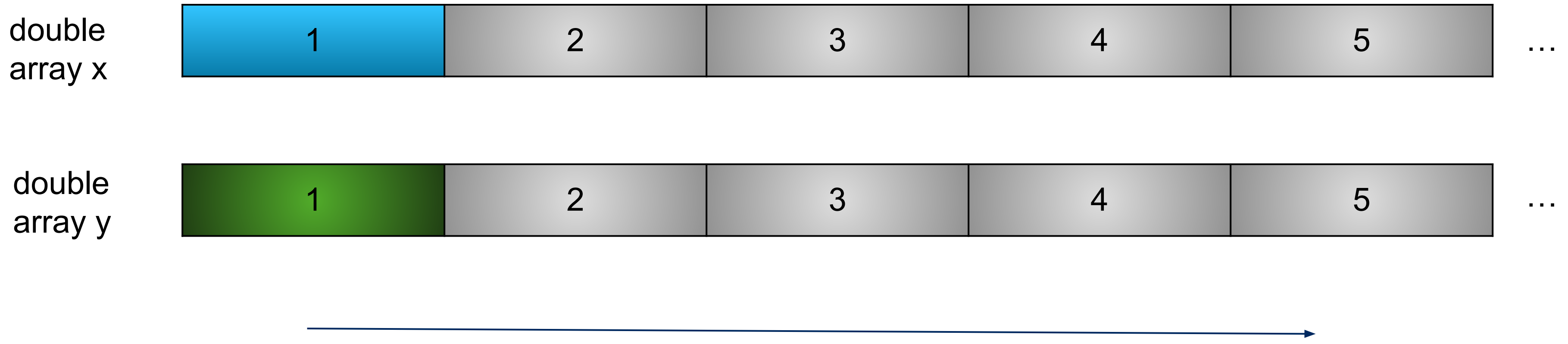
Note: **OpenCV** in green, **Eigen** in yellow

Faint triangles and circles are PAPI measured

theoretical OI is darker circles and triangles

Covariance Sequential Version

2 arrays both of size n



Time/ n Iterations

Within i-th step

sum_x += ith

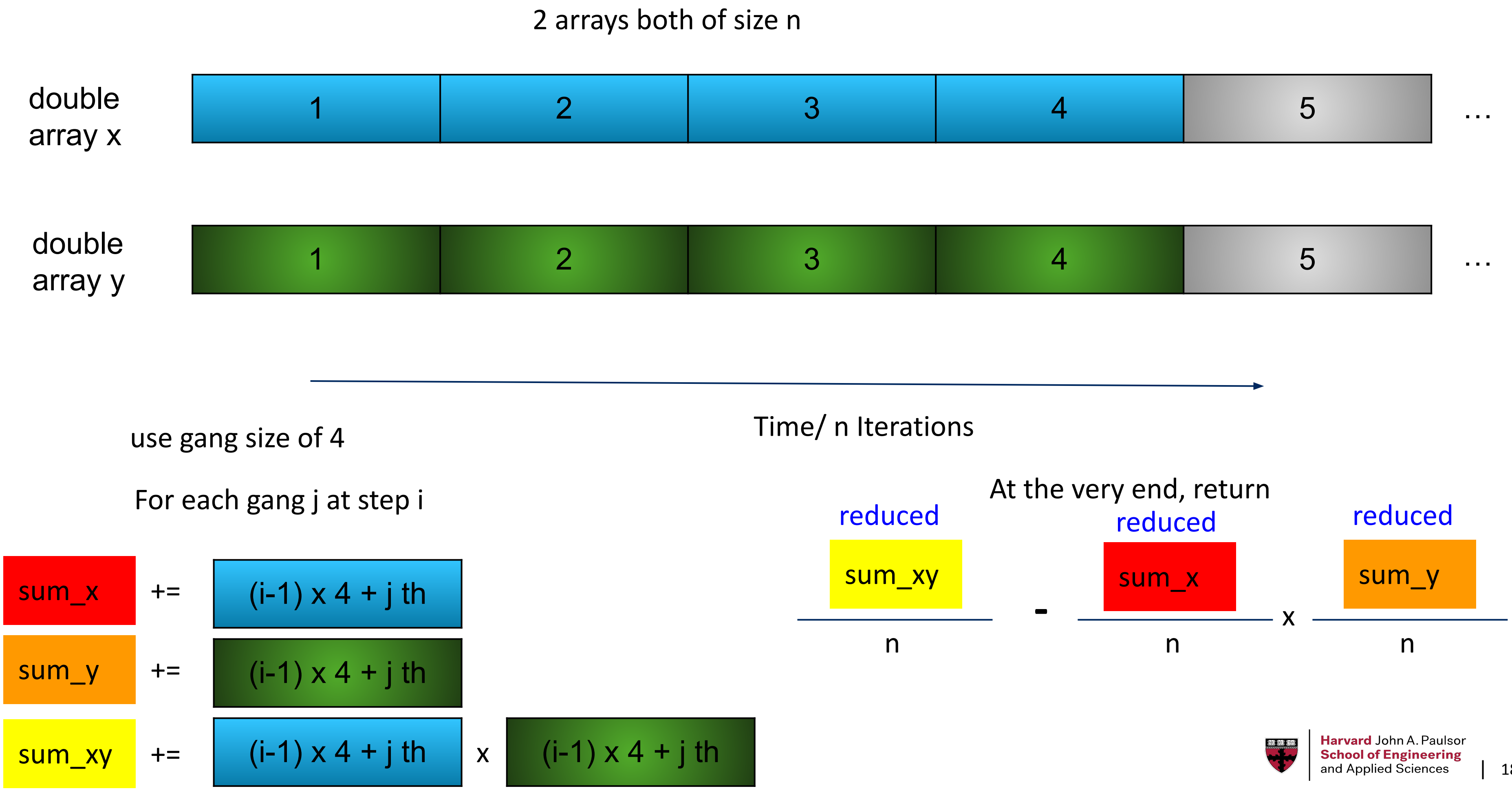
sum_y += ith

sum_xy += ith x ith

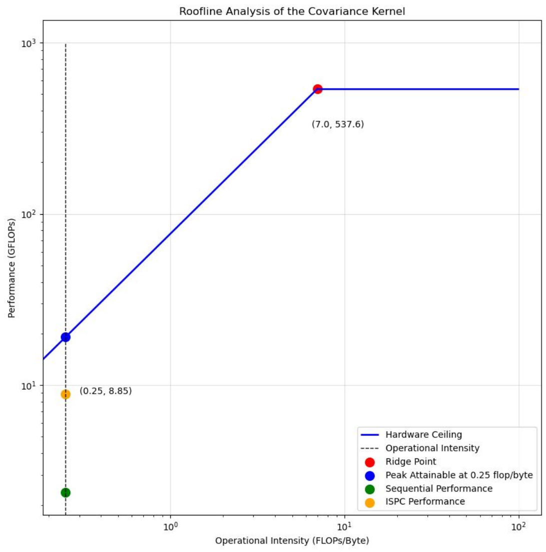
At the very end, return

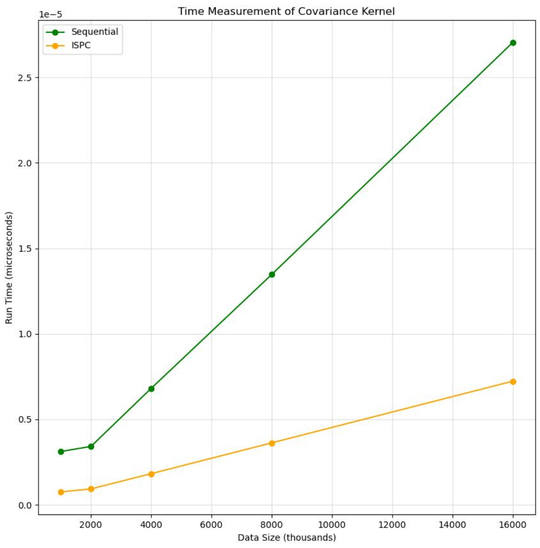
$$\frac{\text{sum\_xy}}{n} - \frac{\text{sum\_x}}{n} \times \frac{\text{sum\_y}}{n}$$

Covariance ISPC Version



Covariance - ISPC Performance Analysis





- Highlights
  - ISPC at same level of magnitude as nominal peak performance
  - Could push further with loop unrolling to saturate all registers
  - ~4x time reduction for a medium sized data input (O2 flag)

Procrustes: find the optimal linear transformation between two sets of DataPoints

2 arrays of size N

Point2d Object a	a[0] = (a[0].x, a[0].y)	a[1] = (a[1].x, a[1].y)		…	a[N] = (a[N].x, a[N].y)
Point2d Object b	b[0] = (b[0].x, b[0].y)	b[1] = (b[1].x, b[1].y		…	b[N] = (b[N].x, b[N].y)

Time →

Within the for loop:

X1 += a[i].x

X2 += b[i].x

Y1 += a[i].y

Y2 += b[i].y

Z += $(b[i].x)^2 + (b[i].y)^2$

C1 += a[i].x · b[i].x + a[i].y · b[i].y

C2 += a[i].y · b[i].x - a[i].x · b[i].y

At the very end:

$$\textbf{Matrix A} = \begin{bmatrix} X2 & -Y2 & N & 0 \\ Y2 & X2 & 0 & N \\ Z & 0 & X2 & Y2 \\ 0 & Z & -Y2 & X2 \end{bmatrix}$$

$$\textbf{Matrix b} = \begin{bmatrix} X1 & Y1 & C1 & C2 \end{bmatrix}$$

Return:

Transformation Vector = A.inv() · b

Procrustes - AVX2

2 arrays of size N

Point2d Object a	a[0] = (a[0].x, a[0].y)	a[1] = (a[1].x, a[1].y)		…	a[N] = (a[N].x, a[N].y)
Point2d Object b	b[0] = (b[0].x, b[0].y)	b[1] = (b[1].x, b[1].y		…	b[N] = (b[N].x, b[N].y)

Time →

Within the for loop : unroll by 2

sumX1 += a[i].x

sumX2 += b[i].x

sumY1 += a[i].y

sumY2 += b[i].y

sumZ += $(b[i].x)^2 + (b[i].y)^2$

sumC1 += a[i].x · b[i].x **x4** + a[i].y · b[i].y **x4**

sumC2 += a[i].y · b[i].x **x4** - a[i].x · b[i].y **x4**

At the very end:

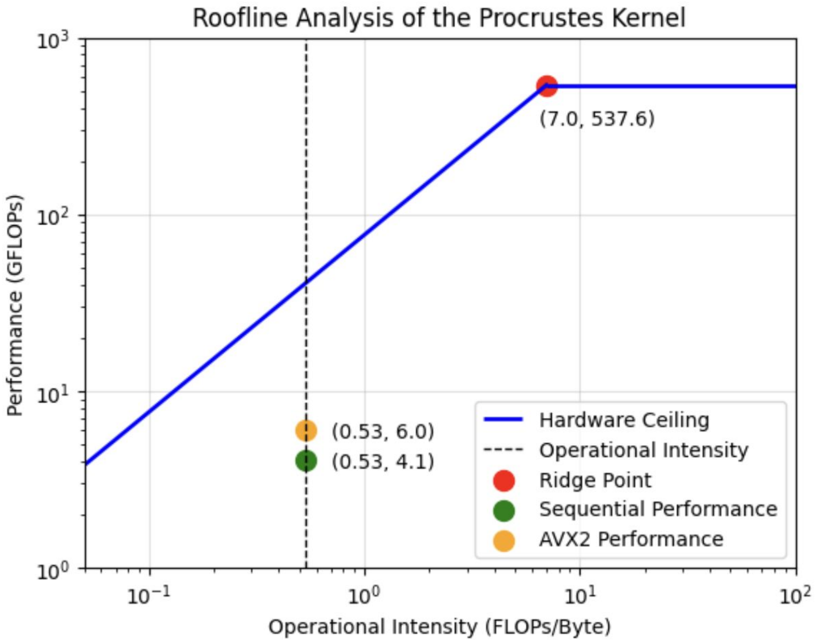
$$\text{Matrix A} = \begin{bmatrix} X2 & -Y2 & N & 0 \\ Y2 & X2 & 0 & N \\ Z & 0 & X2 & Y2 \\ 0 & Z & -Y2 & X2 \end{bmatrix}$$

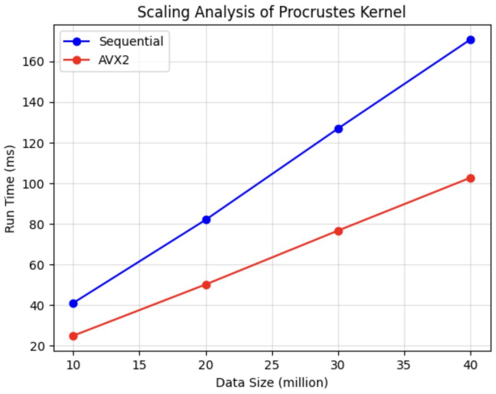
$$\text{Matrix b} = \begin{bmatrix} X1 & Y1 & C1 & C2 \end{bmatrix}$$

Return:

Transformation Vector = A.inv() · b

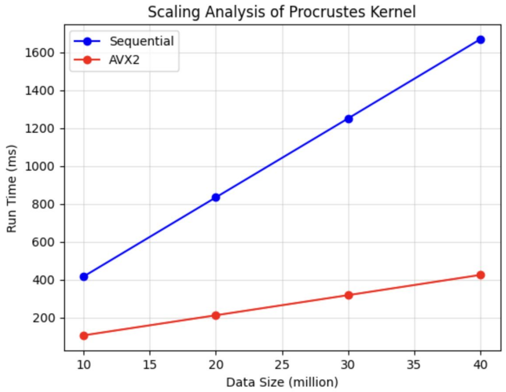
Procrustes - AVX2





with -O3 flag

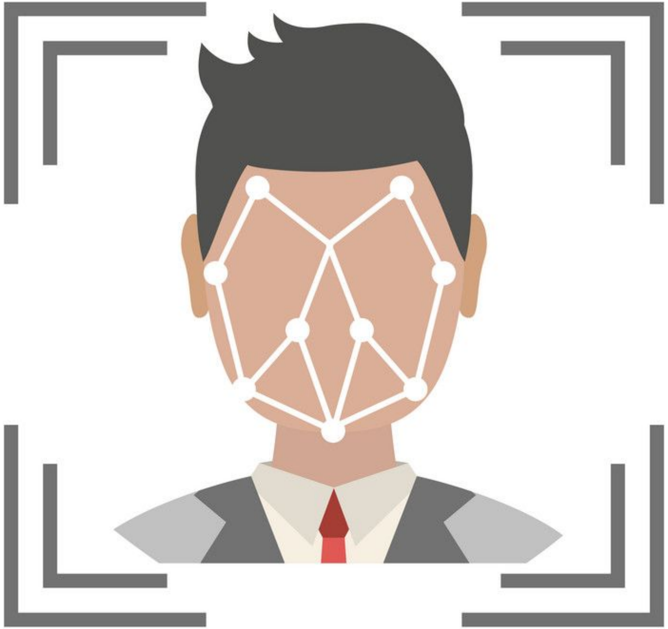
2x Speedup



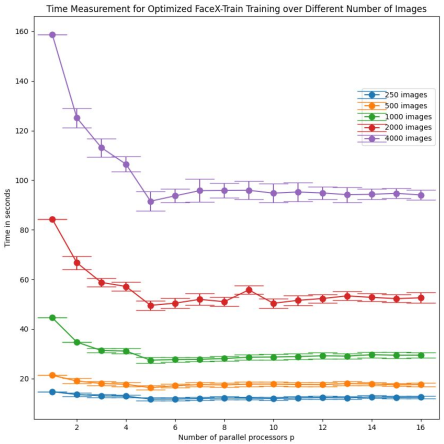
without -O3 flag

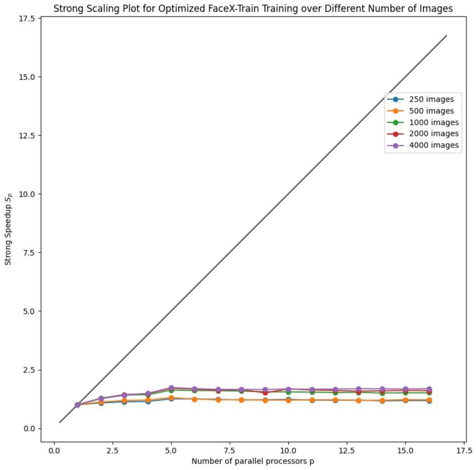
4x Speedup

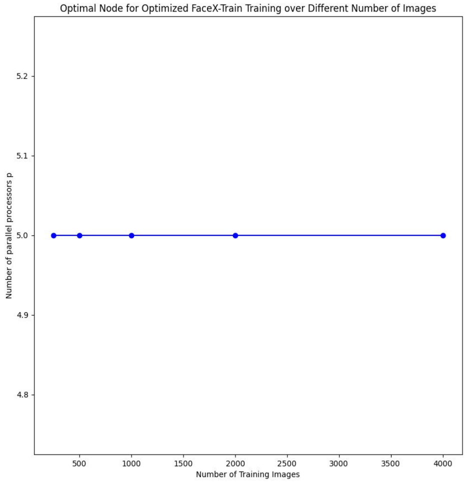
Contents



Performance Benchmarks and Strong Scaling







Optimal Number of Parallel Processes: 5

Sequential vs. Parallelized

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

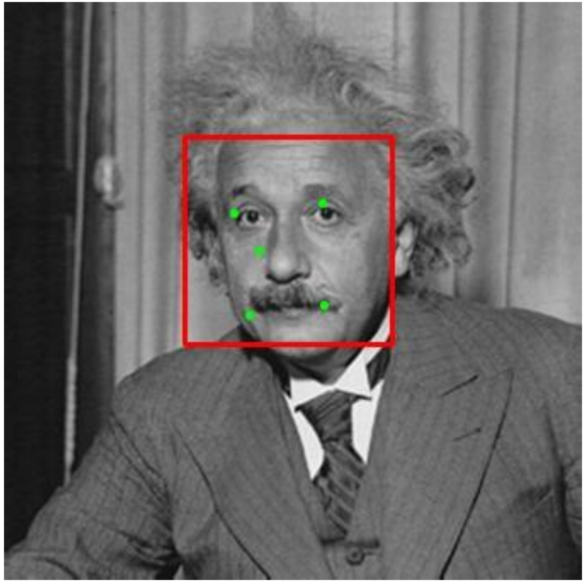
Table 2: Time (s) Taken to process various input file quantities

5 processes running in parallel

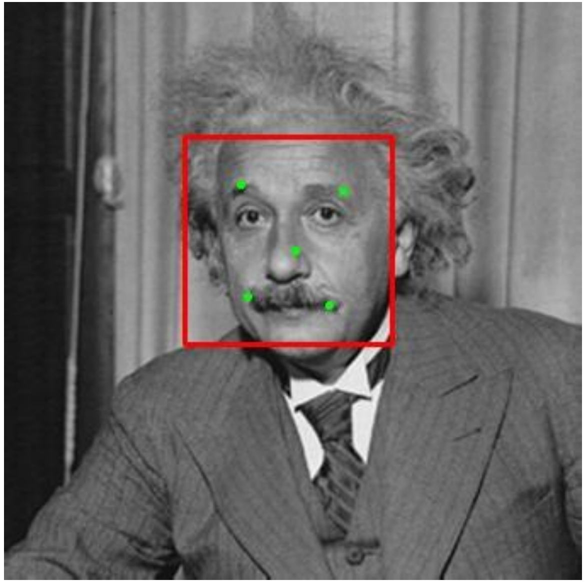
4.4x Speedup for training 250 images

2.2x Speedup for training 4000 images

Sequential vs. Parallelized



Sequential



Parallelized

Future Works

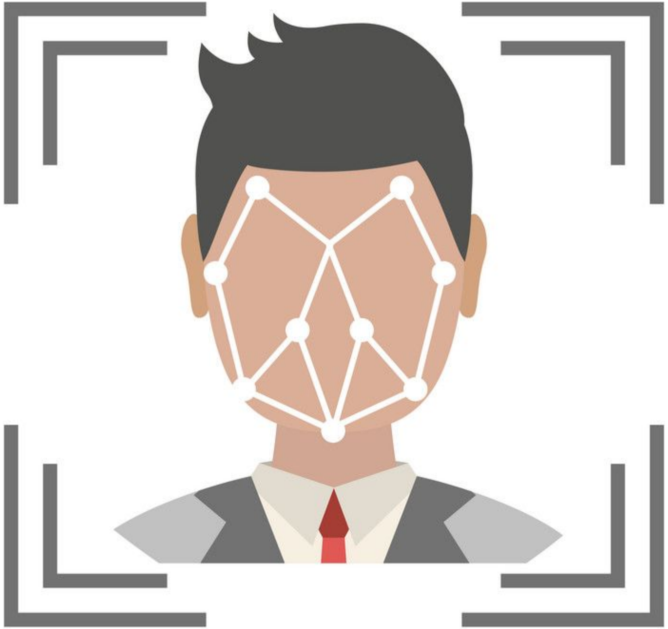
- Real time recognition
  - Optimize FaceX
- Hyperparameter tuning automation
- Integrated Preprocessing Code
  - Handle initialization calculation faster
  - Provide starter code in repo, tested to deliver identical DataPoints when compared to serial version

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.

Parallelizing Gradient Boosted Regression for Facial Landmark Recognition

Zhecheng Yao, Yixian Gan, Rebecca Qiu, Lucy Li
May 2023



Harvard John A. Paulson
School of Engineering
and Applied Sciences