Check Reader

Mengxiong Liu
Zhengchao Liu
Qiwen Chen

Our Goals

Given a image of a check, do the following:

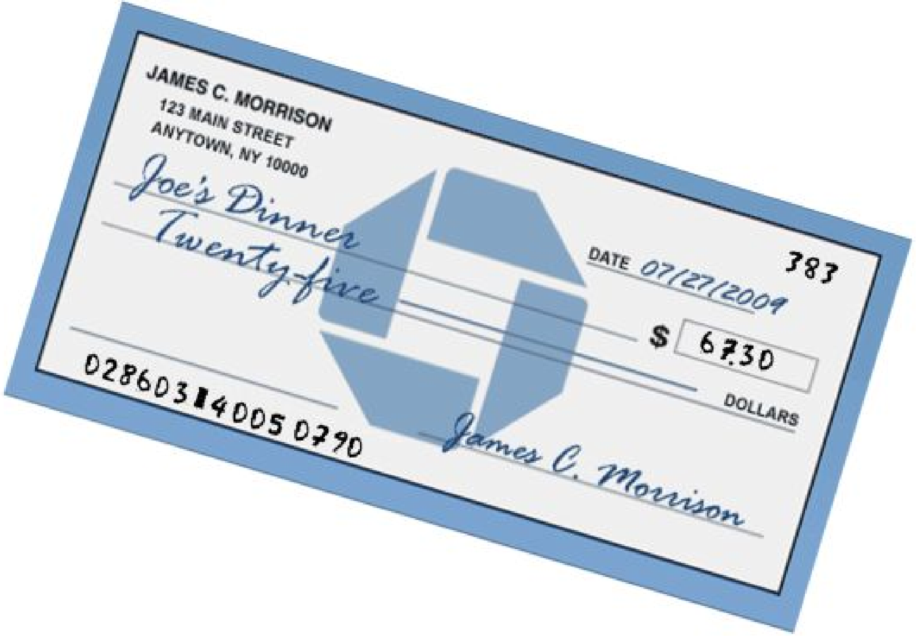
- ❖ Grab the check from the background
- ❖ Verify if the check is valid
- ❖ Output the routing number, account number, check number and the amount of money

Examples

Check only


```
Valid Check from Chase Bank
Routing Number: 028603140
Account Number: 050790
Amount: 67.30
Check Number: 383
```

Rotation




```
Valid Check from Chase Bank
Routing Number: 028603140
Account Number: 050790
Amount: 67.30
Check Number: 383
```

Rotation and Displacement


```
Valid Check from Chase Bank
Routing Number: 028603140
Account Number: 050790
Amount: 67.30
Check Number: 383
```

Rotation and Displacement and On desk

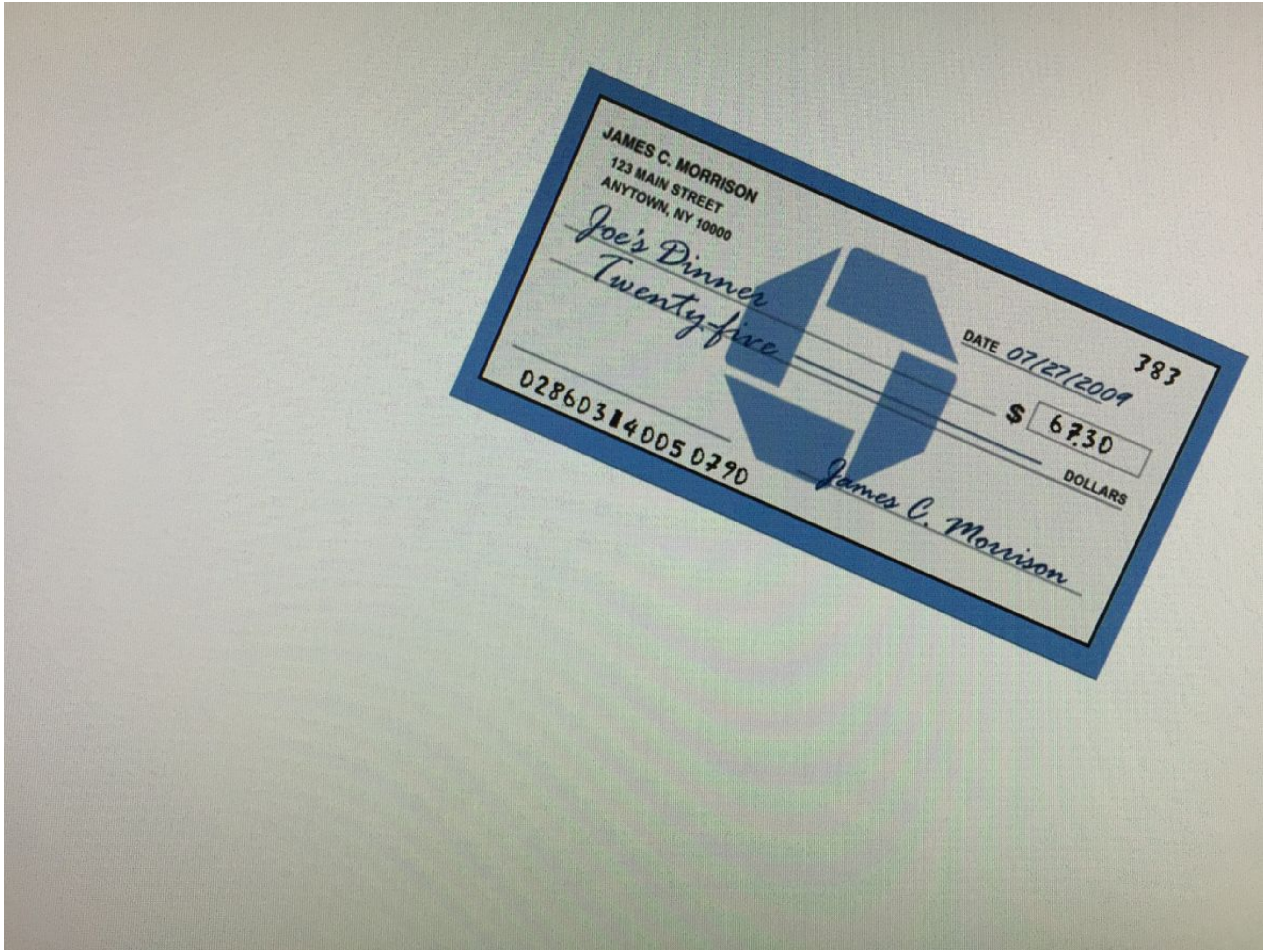

```
Valid Check from Chase Bank
Routing Number: 028603140
Account Number: 050790
Amount: 67.30
Check Number: 383
```

Actual Photo

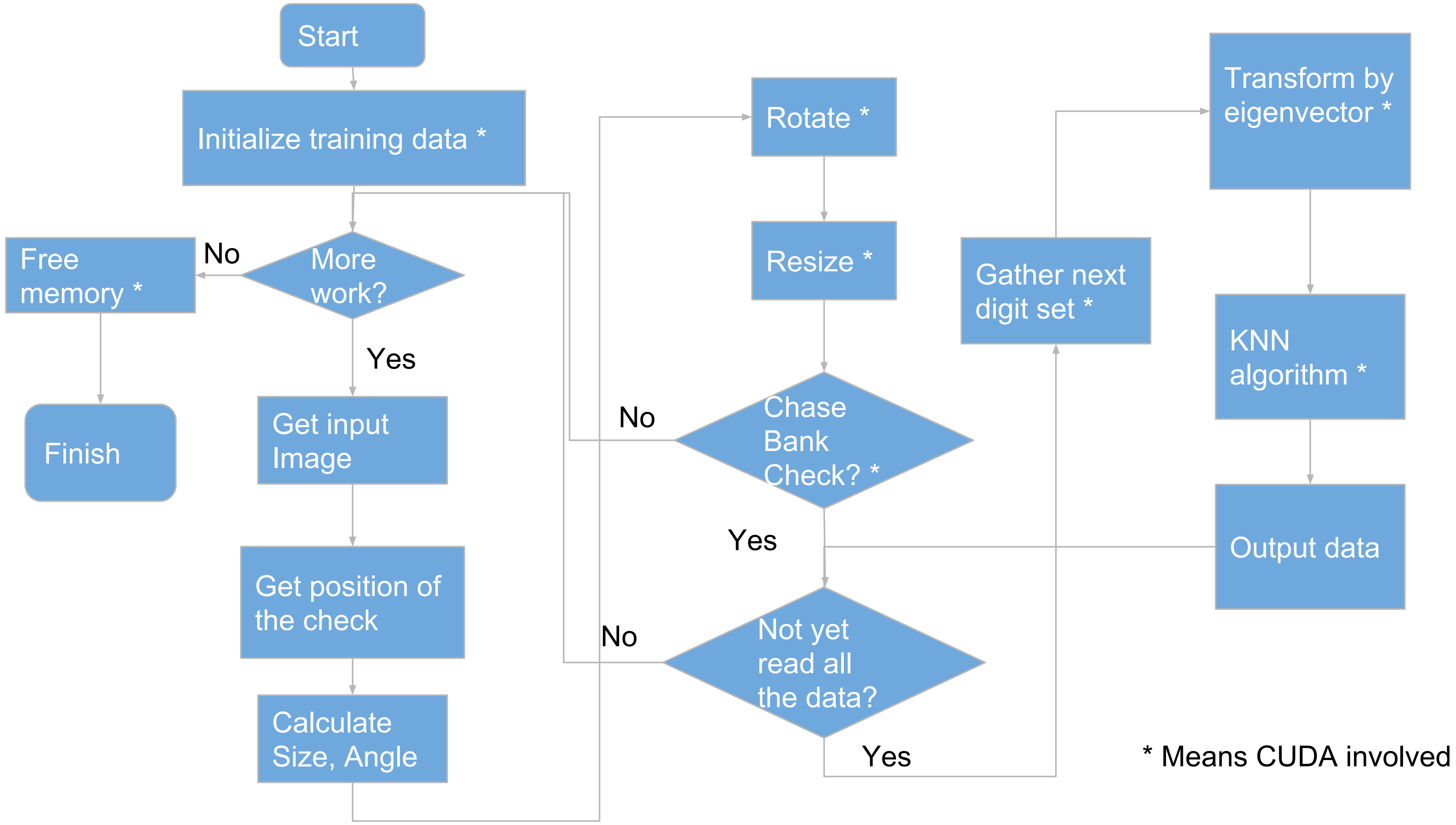
```
Valid Check from Chase Bank
Routing Number: 028603140
Account Number: 050790
Amount: 67.30
Check Number: 383
```

Failure Case

```
Valid Check from Chase Bank
Routing Number: 028603140
Account Number: 05079/
Amount: 0.00
Check Number: 383
```



Workflow


Start
Initialize training data *
More work?
No
Free memory *
Finish
Yes
Get input Image
Get position of the check
Calculate Size, Angle
Rotate *
Resize *
Chase Bank Check? *
No
Yes
Not yet read all the data?
No
Yes
Gather next digit set *
Transform by eigenvector *
KNN algorithm *
Output data
* Means CUDA involved

How Did We Do It

- ❖ Grab the check from the back ground
  - ➢ Locate the check
  - ➢ Rotate and resize the check if necessary
- ❖ Verify if the check is valid
  - ➢ Check if the color of the check if correct
- ❖ Output the numbers
  - ➢ Grab the numbers
  - ➢ Use a machine learning algorithm to recognize individual numbers

What Are We Parallelizing

- Image processing
  - Rotation
  - Resize
  - Verification
  - Cast to format for recognition
  - Locate the digit with extremely high accuracy
  - Separate the digits from the background
  - Noise reduction
- Digit recognition
  - PCA transform, multiply the input digit vector to a precomputed eigenvector matrix
  - K-Nearest Neighbors algorithm, a.k.a, KNN algorithm
    - Reliable for digits recognition
    - “Notorious” for computational intensiveness

CUDA Approaches

- Matrix Multiplication
  - PCA transform
- Reduction
  - KNN algorithm
  - Verification
- Convolution
  - Noise reduction
- Image Mapping
  - Rotate and rotate check

Optimizations

1 tEst dIgit rEad fRom gLobal mEmory

The digit being recognized in KNN algorithm will be visited thousands of times, so we put that to constant memory instead

LESS MEMORY CONFLICT in VERIFICATION

Our previous version of verification was not efficient and robust enough, since we are using histogram to keep track of colors, which induces amount of memory conflict. We now apply reduction to verify blue pixels (essential characteristic of a Chase Bank’s check). We apply padding so the dimension of our array is a perfect exponential of 2. It’s more accurate and faster now.

x15 lEss gLobal mEmory vIsit

KNN kernel function decode a set of digits at a time. Each block is responsible of comparing test data to one piece of training data. Therefore, training data is loaded one time only.

x16 lEss mEmory uSe and FLOP iN kNn

By multiplying test data (as a vector) by a 1024*64 eigenvector matrix, we reduce the dimension of the input vector from 1024 to 64. So, we reduce constant (test data) and shared (training data) memory use and FLOP by a factor of 16 in KNN

Less control divergence in Tiled Matrix Multiplication Kernel

We pick 64 eigenvectors consciously to avoid unnecessary control divergence in matrix multiplication kernel

Attemps

Intensify

Sometimes the image is of poor quality, under which circumstance BFS, used to locate check, will fail. So we tried to use techniques in the last MP to smooth our original image. We tried implementing the histogram. Though it works on few cases, it makes other cases worse off, especially in cases with light background. We then coped with the problem by modifying our existing code and parameters.

Attemps

Harris Edge Detection

1. Compute x and y derivatives of image

$$I_x = G_\sigma^x * I \quad I_y = G_\sigma^y * I$$

2. Compute products of derivatives at every pixel

$$I_{x2} = I_x . I_x \quad I_{y2} = I_y . I_y \quad I_{xy} = I_x . I_y$$

3. Compute the sums of the products of derivatives at each pixel

$$S_{x2} = G_{\sigma'} * I_{x2} \quad S_{y2} = G_{\sigma'} * I_{y2} \quad S_{xy} = G_{\sigma'} * I_{xy}$$

4. Define at each pixel (x, y) the matrix

$$H(x,y) = \begin{bmatrix} S_{x2}(x,y) & S_{xy}(x,y) \\ S_{xy}(x,y) & S_{y2}(x,y) \end{bmatrix}$$

5. Compute the response of the detector at each pixel

$$R = Det(H) - k(Trace(H))^2$$

6. Threshold on value of R. Compute nonmax suppression.

Attemps

Harris Edge Detection

We tried implementing the Harris Edge Detection algorithms, but we realize we still need BFS. Then we tried reduction, but we did not manage to convert that part into CUDA, because the size of the shared memory is to be determined at compile time.

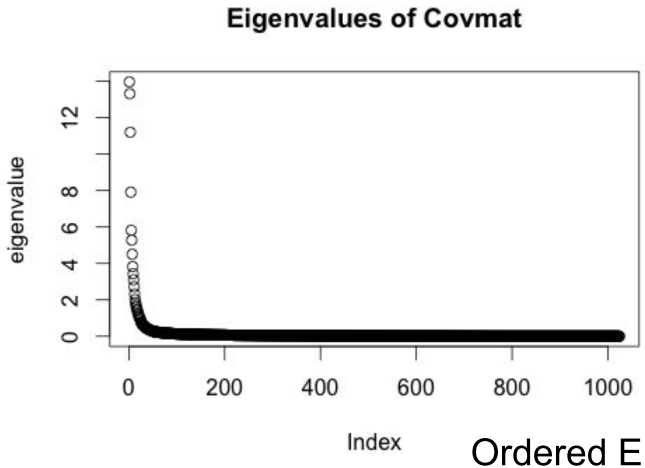
Even so, for our program, we realize that the speed of BFS method is comparable to the Harris Edge Detector algorithms because even after the Harris Edge Detection, reduction and several line search is still required, which would be pretty time consuming.

Attemps

Principal component analysis:

We preprocess the training data in R, wrote matrix multiply code, and revised our code. Now the program will only need datas with dimensions of 64 to perform KNN algorithm effectively.

PCA code in Github.

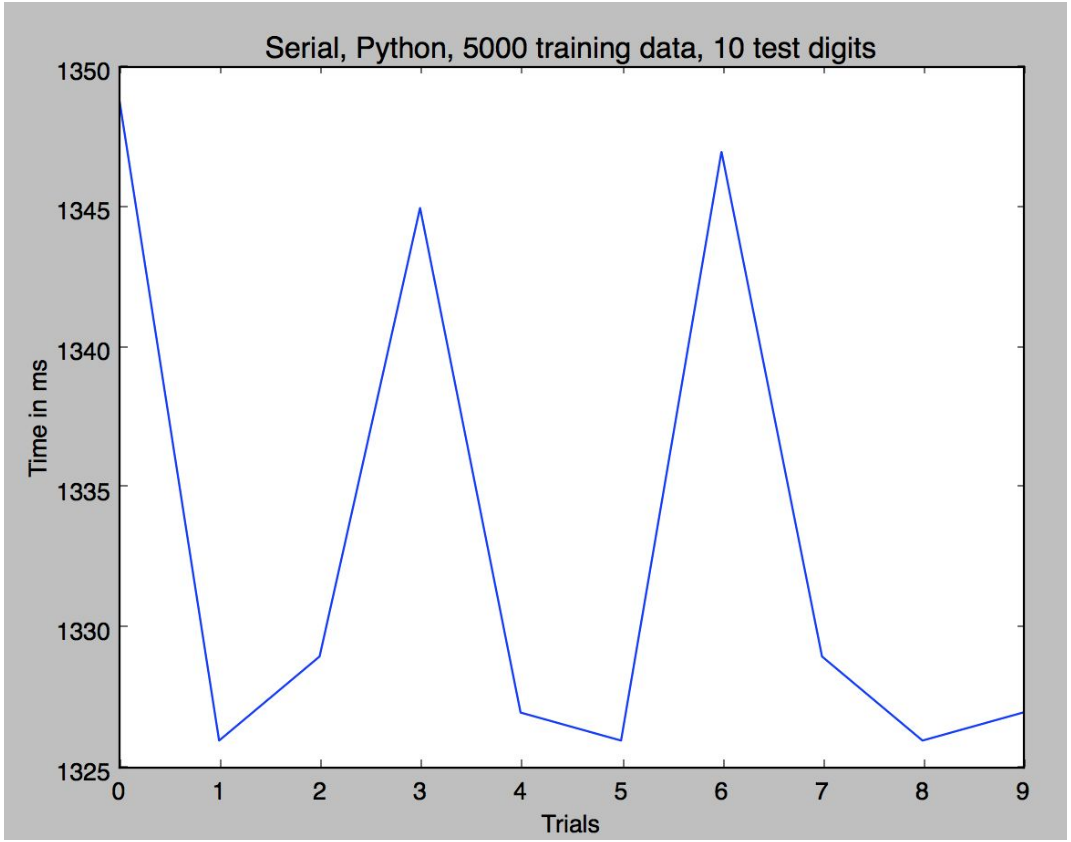


Ordered Eigenvalues

Serial Code We Wrote

About 1335 ms for 10 test digits with 5000 training data in serial Python, with Numpy, code

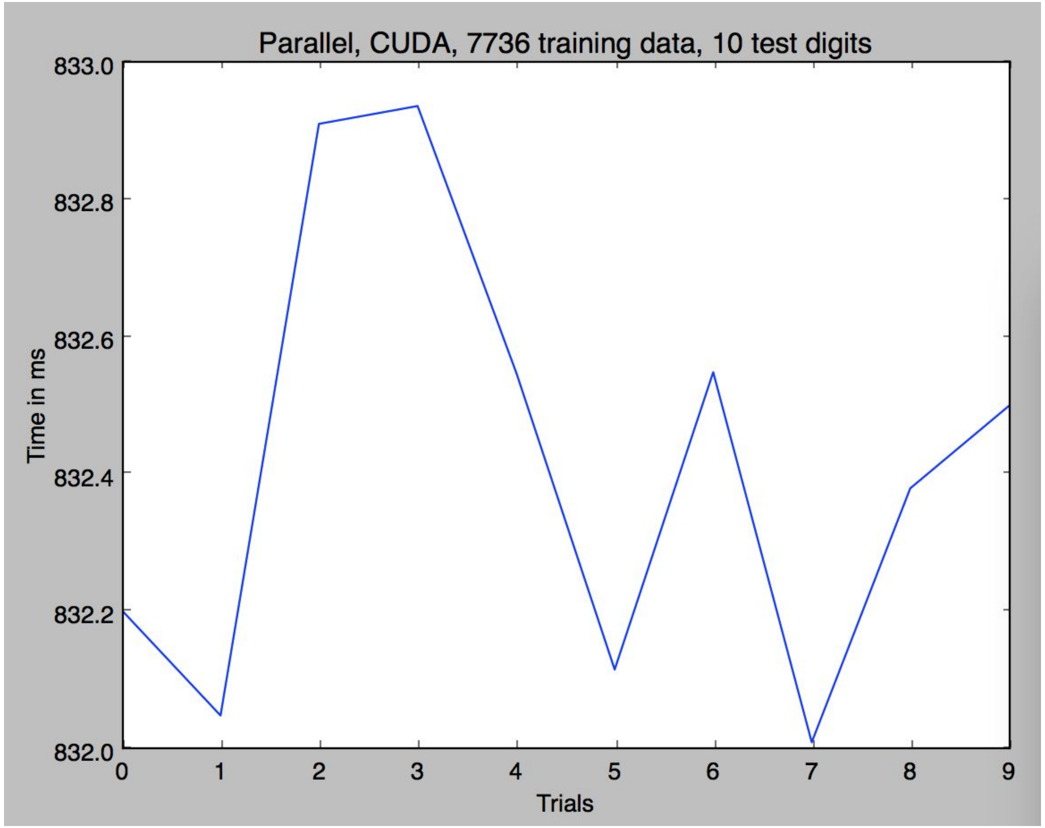
Code source:
Zhengchao Liu
Machine:
EWS Siebel



Our First Version

About 832 ms for 15 test digits with 7736 training data in parallel CUDA code

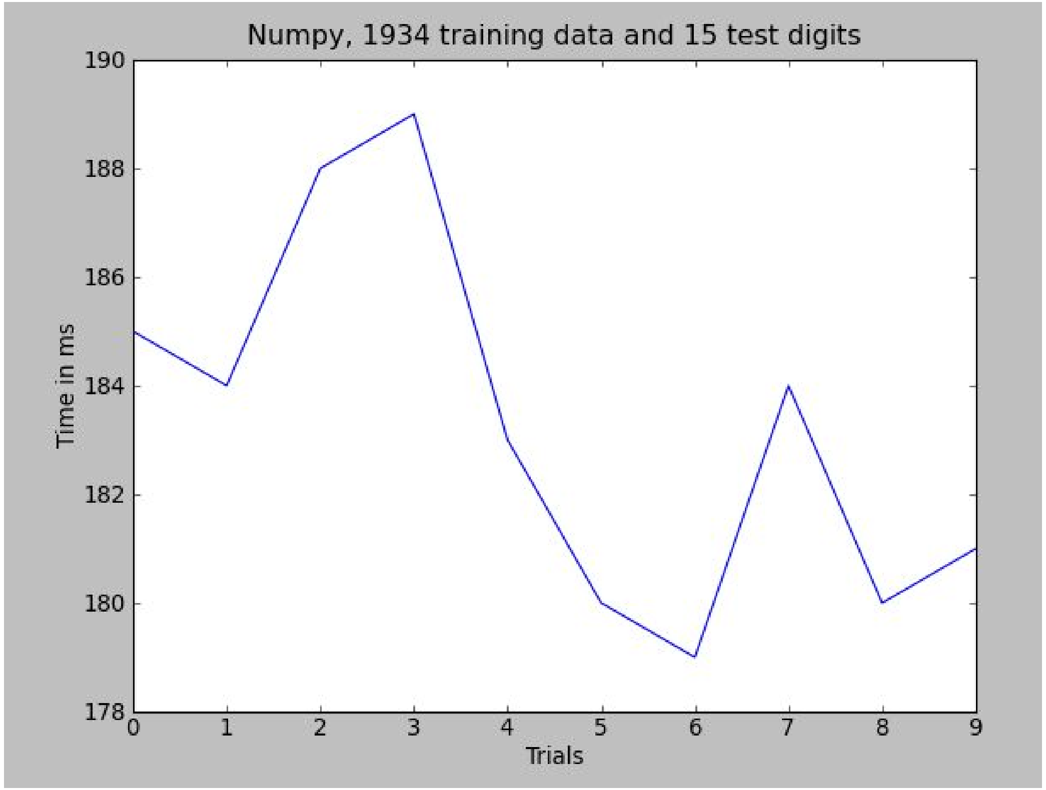
Code source:
Mengxiong Liu
Zhengchao Liu
Qiwen Chen
Machine:
EWS Siebel



Origin Serial Code

About 180 ms for 15 test digits with 1934 training data in Numpy

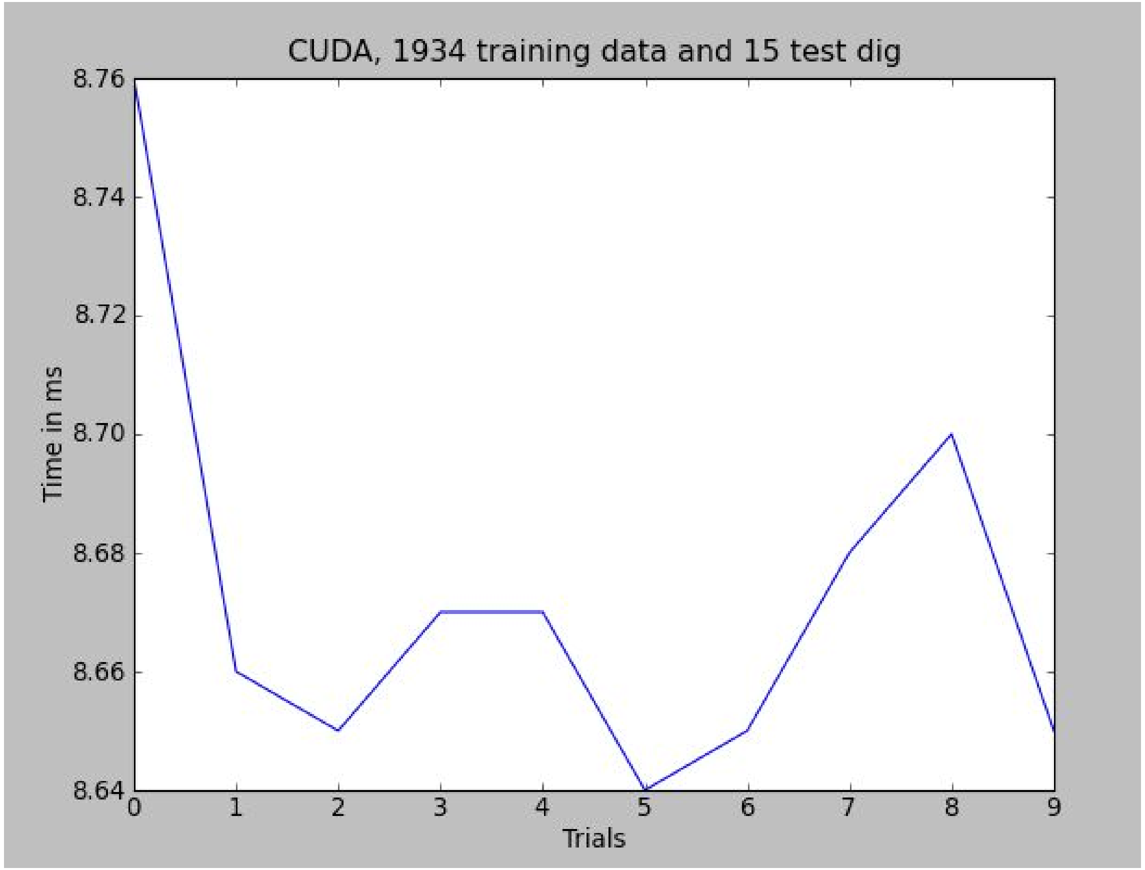
Code source:
Machine Learning in Action, Chapter 2
Machine:
EWS DCL



Our Code

About 8.7 ms for 15 test digits with 1934 training data in CUDA

Code source:
`Mengxiong Liu`
Zhengchao Liu
Qiwen Chen
Machine:
EWS DCL



Future Enhancements

- Pipeline
  - we were trying to piping the CPU code for gathering the digits and the GPU code for KNN algorithms so that the CPU can do gathering while GPU KNN the previous set of the digits. However, we then realize the KNN also includes a portion of CPU code for sorting, which makes piping tricky to implement. It’s still a good idea for future enhancements though.

Future Enhancements

- Corner detection
  - Keep Optimizing the Harris Edge Detection Algorithms to replace the BFS section.

Lesson Learned

- Apply CUDA to real life application
- Deal with CUDA in larger scale than MPs
- Optimize our working code
- Do our first final project in college, ~1000 line program from scratch
- Version control techniques like branches

Contributions

Mengxiong Liu:

Locating check, Rotation, Resizing, reduce dimension with PCA

Zhengchao Liu:

Gathering digits, KNN, Slides, Poster

Qiwen Chen:

Verification, Test datas, Poster

Thank you!