Facial Recognition using Eigenface
ME 766 - High Performance Scientific Computing

Shubham Anand **160050060** | **Sudhanshu Raj** **160050041** | **Ankit** **160050044** | **Simranpreet Singh** **160050111**

1. INTRODUCTION

- The task of face recognition is to recognise a person from a given image. This is done by matching some set of feature derived from the given image with those existing in our database.
- There are some patterns that in all the input images. These patterns are being analyzed to identify one specific image from a set of input images.
- We compute characteristic features known as eigenfaces and calculate the eigen-coefficients for each image using these eigenfaces.
- Eigenface based facial recognition is used to identify a particular face from a large database. Our aim is to analyse the algorithm and parallise various segment of the program to enhance the speed-up (like matrix related operations) by using viennaCL library.

2. SAMPLE INPUT/OUTPUT FORMAT

- The database consists of images of 9 people and corresponding to each person there are 30 images. In total there are 270 images.
- These are gray scale images of size 168 x 192.
- The output of the code is the index of the image to which the input image is matched most closely to the above 270 images.
- The number of images which is 270 is not fixed. For analysing the timing of the code, we these number is varied from 180 to 540.

3. ALGORITHM

- For each gallery image, we compute the eigen-coefficients. We then store the eigen-coefficients and the identity of the person in a database.
- During the testing phase, we are given a probe image.We then compute the eigen coefficients for this image.
- We compare the above computed eigen-coefficients with eigen-coefficients stored in our database and find the closest match in terms of sum squared error.

4. LANGUAGE AND LIBRARY USED

- C++ for the base code
- CUDA for parallelization
- OpenMP for parallelization
- ViennaCL Library for computing eigenvalues/vectors and matrix multiplication

5. LIBRARY FUNCTIONS USED

- **prod** - For multiplying two matrices
- **copy** - For copying a data structure from host to device and vice-versa
- **qr_method_sym** - Calculating the eigenvalues and eigenvectors of a symmetric matrix
- **trans** - For transposing a matrix

6. TIMING ANALYSIS (for Input consisting of 270 images)

We used :

- **nvprof** for profiling
- **time** (command) for calculating the overall time for execution
- **nvvp** for visualising the performance and profiling

- Overall Time for Serial Execution : **82.512** s
- Overall Time for Parallel Execution (OpenMP) using 8 processors : **26.02** s
- Overall Time for Parallel Execution (CUDA) : **2.135** s
- Speedup for CUDA Execution = 85.512/2.135 = **40.052**
- Speedup for OpenMP Execution = 82.512/26.02 = **3.286**

Comparison of self-defined vs library Matrix Multiplication Function-

- Time taken using self-defined function = 1.72 s
- Time taken using library function = 0.128 s

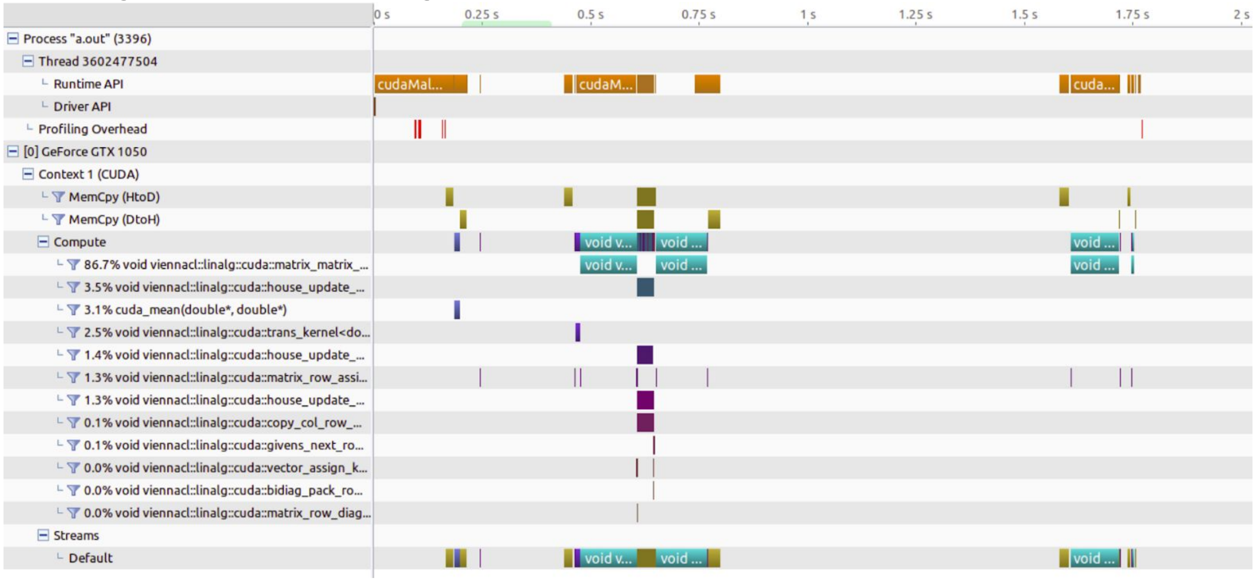
7. PROFILING

The profiling as done by nvprof is presented below --

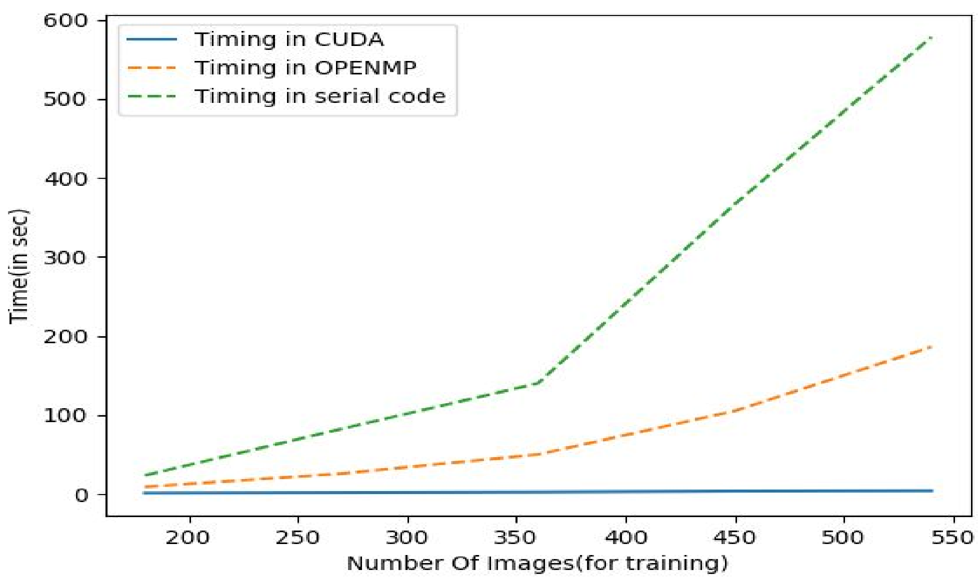
```
$ nvprof ./a.out

==31566== NVPROF is profiling process 31566, command: ./a.out
==31566== Profiling application: ./a.out
==31566== Profiling result:
          Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   65.14%  178.08ms         4  44.521ms  5.5187ms  65.568ms  void viennacl::linalg::cuda::matrix_matrix_row_row_row_prod_
                   15.74%  43.029ms       248  173.50us     608ns  13.373ms  [CUDA memcpy HtoD]
                    9.36%  25.592ms       186  137.59us     640ns  15.064ms  [CUDA memcpy DtoH]
                    3.07%  8.3939ms         1  8.3939ms  8.3939ms  8.3939ms  cuda_mean(double*, double*)
                    1.95%  5.3220ms       178  29.899us  2.3680us  70.913us  void viennacl::linalg::cuda::house_update_A_left_row_major_kernel
                    1.80%  4.9169ms         1  4.9169ms  4.9169ms  4.9169ms  void viennacl::linalg::cuda::trans_kernel
                    1.41%  3.8666ms        11  351.51us  2.2720us  892.46us  void viennacl::linalg::cuda::matrix_row_assign_kernel
                    0.71%  1.9295ms       178  10.839us  10.240us  13.056us  void viennacl::linalg::cuda::house_update_QL_row_major_kernel
                    0.69%  1.8741ms       178  10.528us  9.6960us  12.384us  void viennacl::linalg::cuda::house_update_A_right_row_major_kernel
                    0.08%  207.27us       178  1.1640us     960ns  1.6640us  void viennacl::linalg::cuda::copy_col_row_major_kernel
                    0.06%  173.22us        32  5.4130us  2.6560us  7.8720us  void viennacl::linalg::cuda::givens_next_row_major_kernel
                    0.00%  8.4160us         6  1.4020us  1.0560us  1.7600us  void viennacl::linalg::cuda::vector_assign_kernel
                    0.00%  1.7600us         1  1.7600us  1.7600us  1.7600us  void viennacl::linalg::cuda::bidiag_pack_row_major_kernel
                    0.00%  1.6320us         1  1.6320us  1.6320us  1.6320us  void viennacl::linalg::cuda::matrix_row_diagonal_assign_kernel
      API calls:   58.11%  213.23ms       434  491.31us  6.0650us  69.149ms  cudaMemcpy
                   38.04%  139.59ms        18  7.7552ms  6.5540us  135.61ms  cudaMalloc
                    1.78%  6.5198ms        16  407.49us  5.6200us  1.9130ms  cudaFree
                    1.67%  6.1199ms       769  7.9580us  6.3740us  35.356us  cudaLaunch
                    0.19%  702.67us      4793     146ns     117ns     771ns  cudaSetupArgument
                    0.12%  422.22us        94  4.4910us     543ns  167.33us  cuDeviceGetAttribute
                    0.05%  185.47us       769     241ns     157ns  1.0700us  cudaConfigureCall
                    0.02%  86.098us         1  86.098us  86.098us  86.098us  cuDeviceTotalMem
                    0.01%  52.917us         1  52.917us  52.917us  52.917us  cuDeviceGetName
                    0.00%  5.7180us         3  1.9060us     652ns  4.2410us  cuDeviceGetCount
                    0.00%  5.6800us        20     284ns     211ns     407ns  cudaGetLastError
                    0.00%  2.2640us         2  1.1320us     624ns  1.6400us  cuDeviceGet
```

Profiling Visualisation using **nvvp** --



Timing Comparison of **Serial**, **OpenMP** and **CUDA** execution --



8. CONCLUSION

We observe that the speedup achieved through CUDA execution **(40.052)** is far more better than the speedup achieved through openMP execution **(3.286).** This is due to the fact that there are more number of compute elements present in GPU than what can be achieved through the processors.

Using library functions (ViennaCL) also has a significant impact on the speedup. We have demonstrated this using timing analysis and comparing the two instances ie. with our own Matrix Multiplication function and one provided by the library.