Análise dos Componentes Principais com CuPy

Aluno: João Gabriel Cavalcante França - 202201695

Motivação -

CUDA Based Speed Optimization of the PCA Algorithm

- Este trabalho de implementação é baseado no paper CUDA Based Speed Optimization of the PCA Algorithm, de 2016, escrito por três autores, Salih Görgünoğlu, Kadriye Öz, Abdullah Çavuşoğlu, da Universidade de Karabük, na Turquia.

- O paper aborda a paralelização do PCA, através da API de CUDA em C, aplicado ao contexto dos Eigenfaces. Os autores demonstram testes e os resultados obtidos, a partir da execução em três GPU's com diferentes configurações.

Problema

In the min-max normalization stage, each thread travels through the pixels that form a face image and finds the minimum and maximum values of them. The same thread makes a second traversing operation on them but this time to normalize the values within the range of 0-1. As figure 3. illustrates for 400 face images, speedup rates of up to 5-6 times are obtained.

- O paper, no entanto, omite partes fundamentais do código que facilitariam a melhor compreensão do leitor a respeito da implementação. Em certos trechos, os autores descrevem através de linguagem natural o caminho percorrido pelas threads.

- O meu trabalho de implementação visa cobrir as principais etapas da Análise dos Componentes Principais aplicada aos Eigenfaces, realizando testes e explicitando os códigos utilizados, que posteriormente serão disponibilizados em um repositório no Github.

Recapitulando o PCA - Qual o problema e o que o PCA resolve?

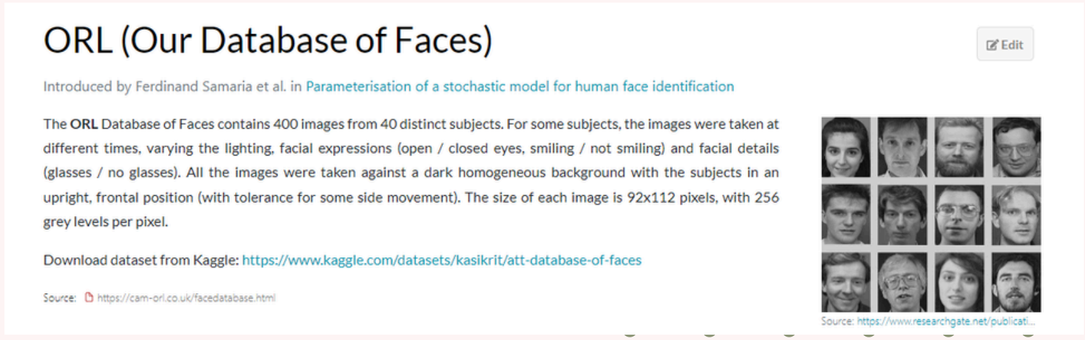
- É um algoritmo para redução de dimensionalidade
- Projetamos nossos dados de alta dimensão em um espaço de menor dimensão, nos eixos que preservem a maior variância possível.
- No reconhecimento facial, temos imagens de alta resolução de muitos indivíduos que, no método dos Eigenfaces, devem ser processadas a fim de gerar uma face média, que será combinada linearmente para gerar as outras faces do dataset.

Características

- A implementação do PCA será em Python, para maior facilidade na manipulação dos dados/imagens, com a utilização de alguns bibliotecas utilitárias. A paralelização do PCA será por via CuPy, uma API de CUDA em Python.

- As imagens de faces utilizadas nos testes serão do dataset ORL, o mesmo utilizado no paper CUDA Based Speed Optimization of the PCA Algorithm;

- 400 imagens em tons de cinza, 92x112, de 40 indivíduos diferentes.



Um pouco sobre o CuPy

- Fácil instalação e possui uma API de alto nível, bem convidativa para iniciantes na programação paralela.
- Apesar de possuir uma API de alto nível para determinadas funções. Podemos também implementar kernels de forma semelhante a API em C/C++, através da definição e inicialização de um RawKernel.
- Dessa forma conseguimos escrever nosso código em C/C++ em uma string e passamos para o modelo RawKernel.

cupy.RawKernel

```
class cupy.RawKernel(unicode code, unicode name, tuple options=(), unicode
backend=u'nvrtc', bool translate_cucomplex=False, *, bool
enable_cooperative_groups=False, bool jitify=False)                [source]
```

User-defined custom kernel.

This class can be used to define a custom kernel using raw CUDA source.

The kernel is compiled at an invocation of the `__call__()` method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is reused by other processes.

Parameters:
- **code** (*str*) – CUDA source code.
- **name** (*str*) – Name of the kernel function.
- **options** (*tuple of str*) – Compiler options passed to the backend (NVRTC or NVCC). For details, see https://docs.nvidia.com/cuda/nvrtc/index.html#group__options or https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#command-option-description
- **backend** (*str*) – Either *nvrtc* or *nvcc*. Defaults to *nvrtc*
- **translate_cucomplex** (*bool*) – Whether the CUDA source includes the header *cuComplex.h* or not. If set to `True`, any code that uses the functions from *cuComplex.h* will be translated to its Thrust counterpart. Defaults to `False`.
- **enable_cooperative_groups** (*bool*) – Whether to enable cooperative groups in the CUDA source. If set to `True`, compile options are configured properly and the kernel is launched with `cuLaunchCooperativeKernel` so that cooperative groups can be used from the CUDA source. This feature is only supported in CUDA 9 or later.
- **jitify** (*bool*) – Whether or not to use Jitify to assist NVRTC to compile C++ kernels. Defaults to `False`.

Exemplo

[20]:

```
kernel_code = """
extern "C" __global__
void vector_add(const float* x, const float* y, float* z, int n) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n) {
        z[tid] = x[tid] + y[tid];
    }
}
"""

# compilação do kernel CUDA
vector_add = cp.RawKernel(kernel_code, 'vector_add')

# tamanho dos vetores
n = 10
x = cp.random.random(n).astype(cp.float32)
y = cp.random.random(n).astype(cp.float32)
z = cp.zeros_like(x)

# configuração do número de blocos e threads
threads_per_block = 32
blocks_per_grid = (n + threads_per_block - 1) // threads_per_block

# execução do kernel
vector_add((blocks_per_grid,), (threads_per_block,), (x, y, z, n))
```

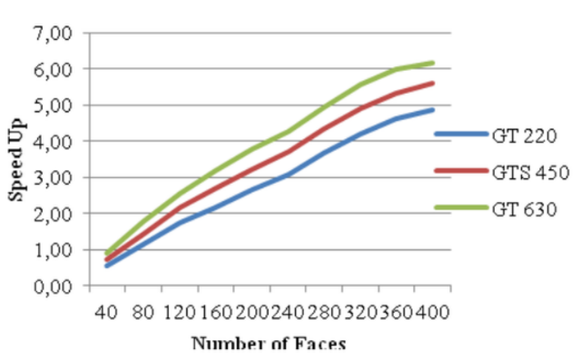
Etapas implementadas

- O paper original dividiu a aplicação do PCA no Eigenfaces em 7 partes, que em minha apresentação passada foram subdivididas em dois grandes grupos, as de menor performance com código paralelizado (Normalização, Cálculo da Face Média, Cálculo da Diferença Φ e Transposição da matriz de diferenças) e as de maior performance com código paralelizado (Cálculo da Matriz Reduzida, Cálculo da Eigenface, Cálculo do Vetor de Características)

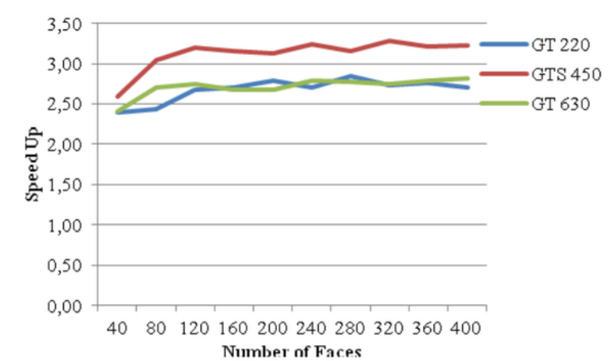
- No meu trabalho de implementação cobrirei as etapas de menor performance, a fim de focar na comparação entre uma placa da modernidade com as placas utilizadas nos testes do paper, que tiveram ganhos menos expressivos.

Relembrando as etapas de menor speedup

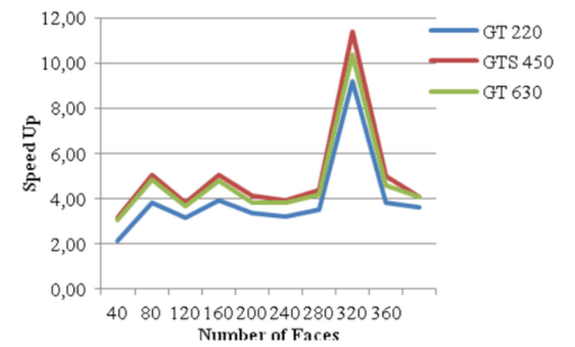
1. Normalização



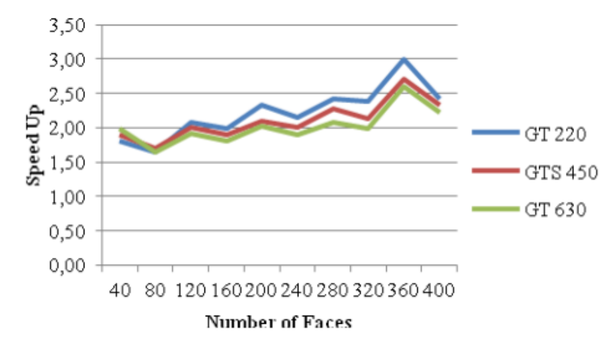
2. Cálculo da Face Média



3. Cálculo da Diferença Φ



4. Transposição da matriz de diferenças



Considerações sobre a implementação

- Sabemos que Python é uma linguagem de alto nível. A fim de reduzir as diferenças nesse sentido, já que a implementação do paper original utiliza a API de CUDA em C, tentei implementar as etapas da forma mais literal possível, tanto as sequenciais, quanto as paralelas, utilizando a notação de rawKernel apresentada anteriormente no código em CuPy, ao invés de chamar métodos abstratos de alto nível.

```python
def dif_s(array1, array2):

    if array1.shape != array2.shape:
         raise ValueError("dif tamanho.")

    return array1 - array2
```

```python
def dif_s(array1, array2):

    if array1.shape != array2.shape:
         raise ValueError("dif tamanho.")

    # inicializa o vetor de dif
    diff = np.zeros_like(array1)

    for idx in np.ndindex(array1.shape):
        diff[idx] = array1[idx] - array2[idx]

    return diff
```

Implementação

1. Normalização

1.1 Sequencial com NumPy

1.1 Normalização Sequencial

[20]:

```python
def normalizeS(image_array):

    image_array = image_array.astype(np.float32)

    # inicializa xmin e xmax
    xmin = float('inf')
    xmax = float('-inf')

    for row in image_array:
        for pixel in row:
            if pixel < xmin:
                xmin = pixel
            if pixel > xmax:
                xmax = pixel

    normalized_array = (image_array - xmin) / (xmax - xmin)

    # Converte o array normalizado de volta para a faixa de 0-255
    normalized_array = (normalized_array * 255).astype(np.uint8)

    return normalized_array
```

- # 1. Normalização

1.2 Paralela com CuPy

1.2 Normalização Paralela

[117]:

```
# rawKernel
kernel_code = '''
extern "C" __global__
void normalize_kernel(float* image, float xmin, float xmax, float* result, int size) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < size) {
        result[idx] = (image[idx] - xmin) / (xmax - xmin) * 255.0;
    }
}
'''

normalize_kernel = cp.RawKernel(kernel_code, 'normalize_kernel')

def normalizeP(image_array):

    image_array = cp.asarray(image_array, dtype=cp.float32)

    xmin, xmax = cp.min(image_array), cp.max(image_array)

    normalized_array = cp.empty_like(image_array)

    size = image_array.size

     # std configs
    block_size = 256
    grid_size = (size + block_size - 1) // block_size
    block_dim = (block_size,)
    grid_dim = (grid_size,)

    normalize_kernel(grid_dim, block_dim, (image_array, xmin, xmax, normalized_array, size))

    return normalized_array.astype(cp.uint8)
```

2. Face Média

2.1 Sequencial com NumPy

2.1 Cálculo da Média Sequencial

[137]:
```
def media_s(arr):
    soma = 0
    contagem = 0
    for elemento in arr:
        soma += elemento
        contagem += 1
    media = soma / contagem
    return media
```

- Lembrando que o cálculo da face média é a etapa em que transformamos cada imagem NxN do nosso dataset (de M imagens) em uma matriz-coluna de tamanho N^2, calculamos a média destas M matrizes-coluna e colocamos em uma matriz Ψ, que será utilizada no cálculo da diferença entre as faces ($\Phi i = \Gamma i - \Psi$)

2. Face Média

2.2 Paralela com CuPy

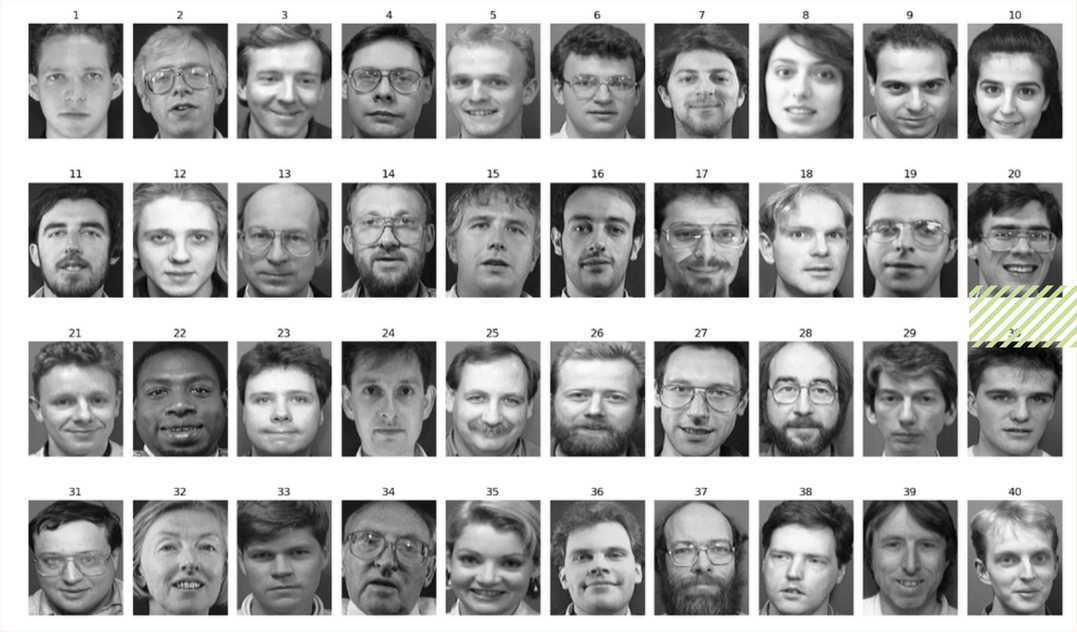
2.2 Cálculo da Média Paralelo

•[171]:

```
kernel_code = r'''
extern "C" __global__
void sum_reduce(const float* x, float* y, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {
    atomicAdd(y, x[i]);
  }
}
'''

# Compilação do kernel
module = cp.RawKernel(kernel_code, name='sum_reduce')

def media_p(arr):
    arr_gpu = cp.asarray(arr, dtype=cp.float32)

    # inicializando para armazenar
    result_gpu = cp.zeros(1, dtype=cp.float32)

    # std configs
    block_size = 256
    grid_size = (arr_gpu.size + block_size - 1) // block_size
    block_dim = (block_size,)
    grid_dim = (grid_size,)

    module(grid_dim, block_dim, (arr_gpu, result_gpu, arr_gpu.size))

    result = cp.asnumpy(result_gpu)

    media = result[0] / arr_gpu.size

    return media
```

• 2. Face Média

Resultado Gráfico



... + 360 faces

```
•[72]: # calcular a face média utilizando media_s
mean_face_vector = np.apply_along_axis(media_s, 0, Data)
mean_face_image = mean_face_vector.reshape(112, 92)  # Re

# chama a função auxiliar
display_image(mean_face_image, title='Face Média')
```

- # 3. Cálculo da Diferença Φ

3.1 Sequencial com NumPy

3.1 Cálculo da Diferença Sequencial

[244]:

```python
def dif_s(array1, array2):

    if array1.shape != array2.shape:
         raise ValueError("dif tamanho.")

    diff = np.zeros_like(array1)

    for idx in np.ndindex(array1.shape):
        diff[idx] = array1[idx] - array2[idx]

    return diff
```

3. Cálculo da Diferença Φ

3.2 Paralelo com CuPy

3.2 Cálculo da Diferença Paralelo

[249]:

```
kernel_code = r'''
extern "C" __global__
void vector_diff(const float* x, const float* y, float* z, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        z[i] = x[i] - y[i];
    }
}
'''

module = cp.RawKernel(code=kernel_code, name='vector_diff')

def dif_p(vetor1, vetor2):
    vetor1_gpu = cp.asarray(vetor1, dtype=cp.float32)
    vetor2_gpu = cp.asarray(vetor2, dtype=cp.float32)

    if vetor1_gpu.size != vetor2_gpu.size:
        raise ValueError("dif tamanho.")

    resultado_gpu = cp.zeros_like(vetor1_gpu, dtype=cp.float32)

    # std configs
    block_size = 256
    grid_size = (vetor1_gpu.size + block_size - 1) // block_size
    block_dim = (block_size,)
    grid_dim = (grid_size,)

    module(grid_dim, block_dim, (vetor1_gpu, vetor2_gpu, resultado_gpu, vetor1_gpu.size))

    # transferindo o resultado de volta para a CPU
    resultado = cp.asnumpy(resultado_gpu)

    return resultado
```

4. Transposição da Matriz de Diferenças

4.1 Sequencial com NumPy

4.1 Transposição Sequencial

•[324]:

```python
def transpose_s(matrix):

    num_rows = len(matrix)
    num_cols = len(matrix[0])

    # create transpose matrix
    transposed_matrix = [[0] * num_rows for _ in range(num_cols)]

    for i in range(num_rows):
        for j in range(num_cols):
            transposed_matrix[j][i] = matrix[i][j]

    return transposed_matrix
```

4. Transposição da Matriz de Diferenças

4.2 Paralelo com CuPy

4.2 Transposição Paralela

•[328]:

```
kernel_code = r'''
extern "C" __global__
void transpose(const float* in, float* out, int rows, int cols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < rows && y < cols) {
        out[y * rows + x] = in[x * cols + y];
    }
}
'''

module = cp.RawKernel(kernel_code, name='transpose')

def transpose_p(matrix):
    matrix_gpu = cp.asarray(matrix, dtype=cp.float32)

    rows, cols = matrix_gpu.shape

    transposed_matrix_gpu = cp.zeros((cols, rows), dtype=cp.float32)

    # std configs
    block_size = 256
    block_dim = (block_size, 1)
    grid_size = (rows + block_size - 1) // block_size
    grid_dim = (grid_size, cols)

    module(grid_dim, block_dim, (matrix_gpu, transposed_matrix_gpu, rows, cols))

    transposed_matrix = cp.asnumpy(transposed_matrix_gpu)

    return transposed_matrix
```

INF INSTITUTO DE INFORMÁTICA UFG UNIVERSIDADE FEDERAL DE GOIÁS

Resultados

Metodologia

- Os testes são comparativos em relação a versão de código sequencial e paralelizado. Além disso, será disponibilizado o gráfico da respectiva etapa para comparação com o paper original.
- O speedup é calculado com base em 100 execuções sequenciais e 100 execuções paralelizadas em 10 grupos com quantidades crescentes de imagens para avaliação de performance.

```
sizes = [40, 80, 120, 160, 200, 240, 280, 320, 360, 400]
```

Especificações

AMD Ryzen 7 5700X 8-Core Processor 3.40 GHz
32GB RAM
GeForce RTX 4060 - 8GB
Windows 10 Pro

Versões

- CUDA Toolkit 12.5
- Python 3.12.3
- Cupy-CUDA12x 13.2.0
- NumPy 1.26.4

+Disponível no requirements.txt do repositório!

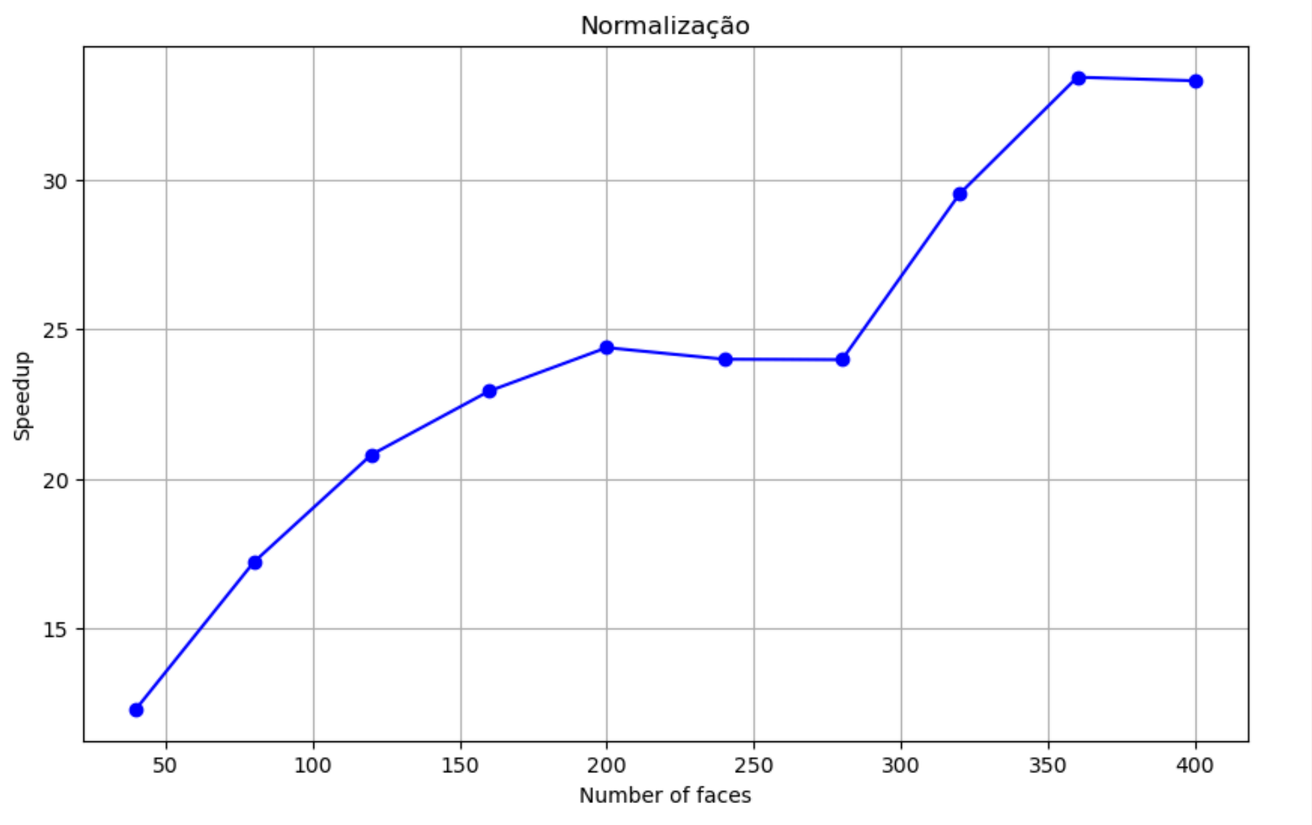
INF INSTITUTO DE INFORMÁTICA
UFG UNIVERSIDADE FEDERAL DE GOIÁS

Estrutura para os testes

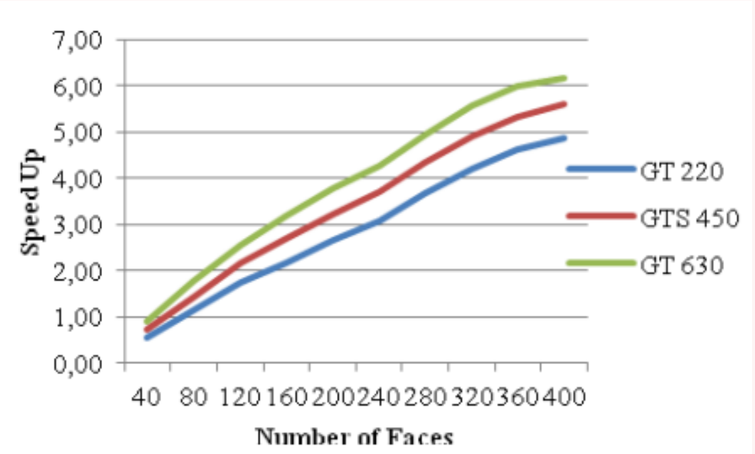
```
•[40]: for i in range(num_parts):
           data_part = parts[i]

           timeS = timeit.timeit(lambda: normalizeS(data_part), number=100)
           timeP = timeit.timeit(lambda: normalizeP(data_part), number=100)

           ratio = timeS / timeP

           ratios.append(ratio)
           x_values.append((i + 1) * part_size)

       plot_speedup(x_values, ratios, "Normalização")
```

```
def plot_speedup(x_values, ratios, title):
    plt.figure(figsize=(10, 6))
    plt.plot(x_values, ratios, marker='o', linestyle='-', color='b')
    plt.xlabel('Number of faces')
    plt.ylabel('Speedup')
    plt.title(title)
    plt.grid(True)
    plt.show()
```

Normalização



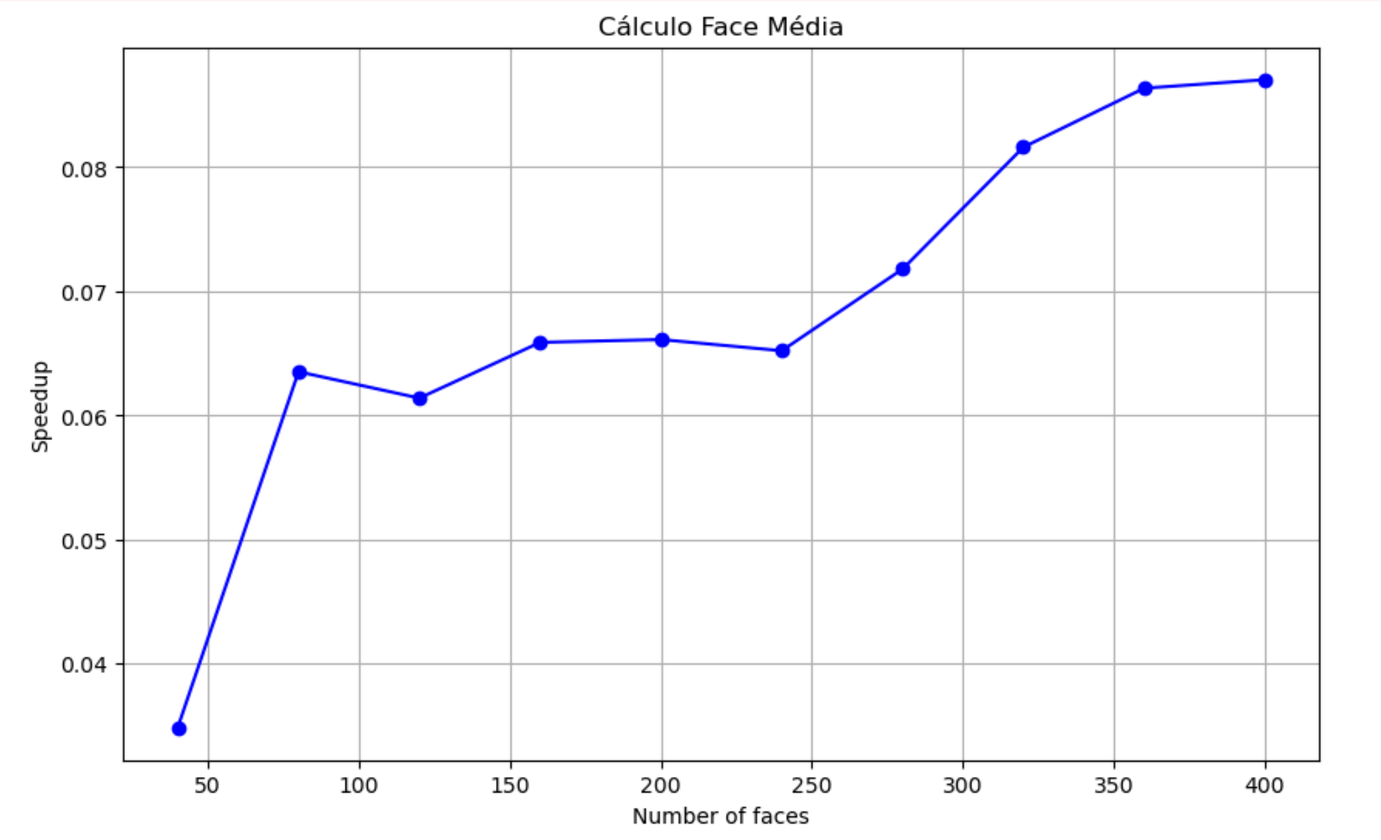
Paper original



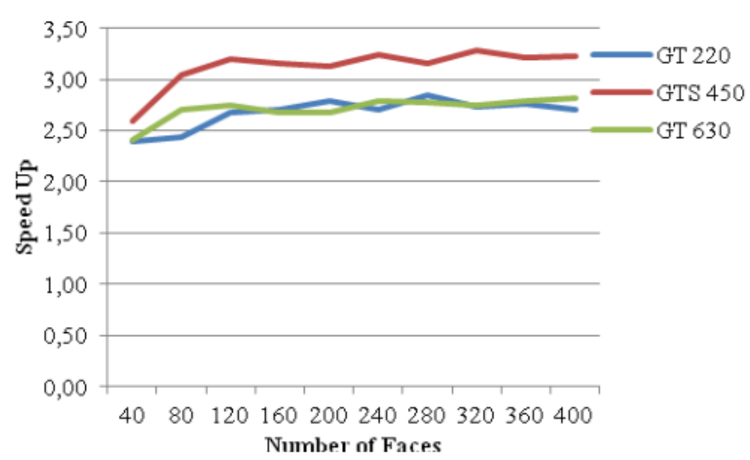
- Speedups de ~1 até ~6

- Speedups de ~5 até ~34
- Resultado em comum: quanto mais faces, maior o desempenho

Cálculo da Face Média



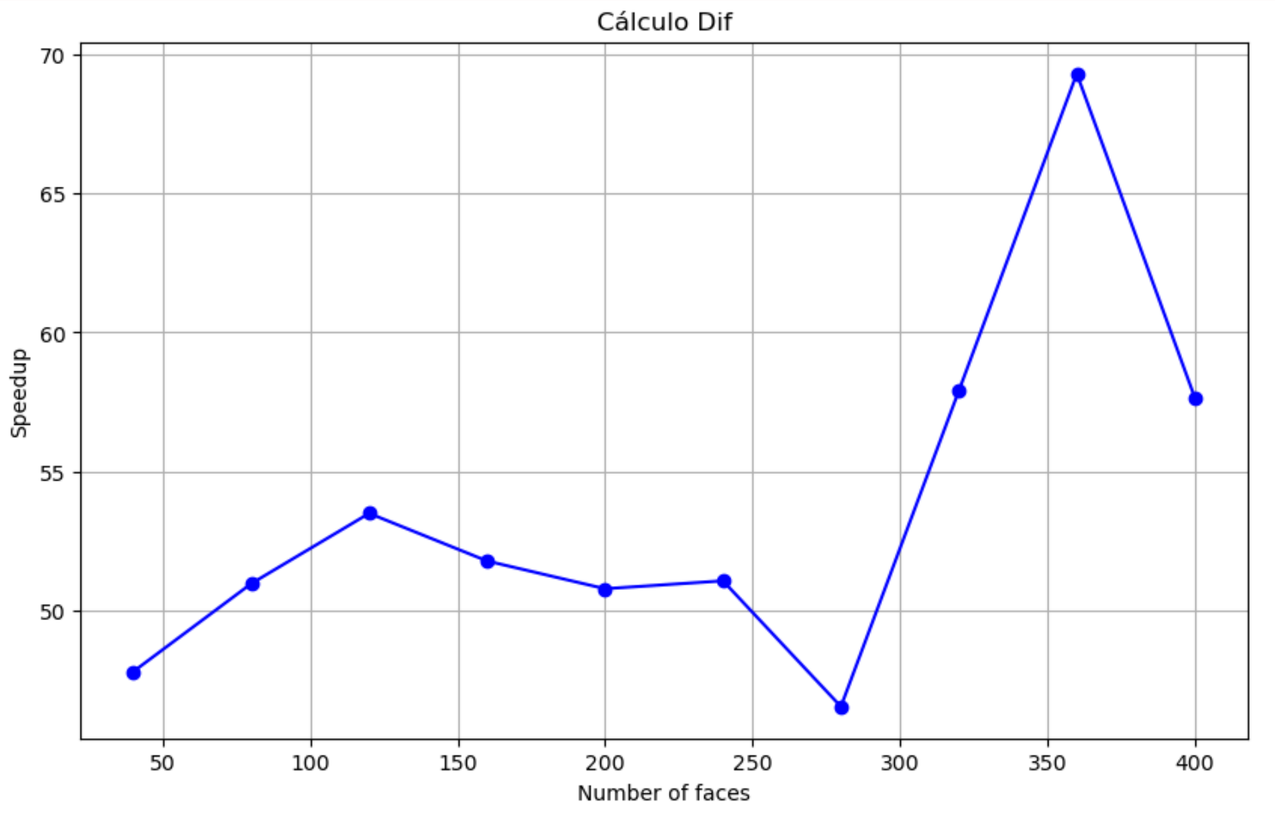
Paper original



- Speedups de ~2,5 até ~3,25

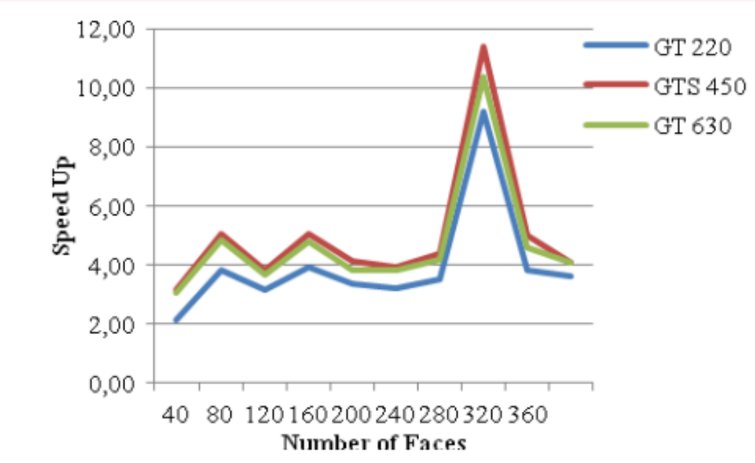
- Solução paralela pior: speedups ~0,01 ~ 0,08
- Resultado em comum: o aumento do número de faces não provocou ganhos significativos no speedup

Cálculo do Vetor de Diferenças (Ψ)



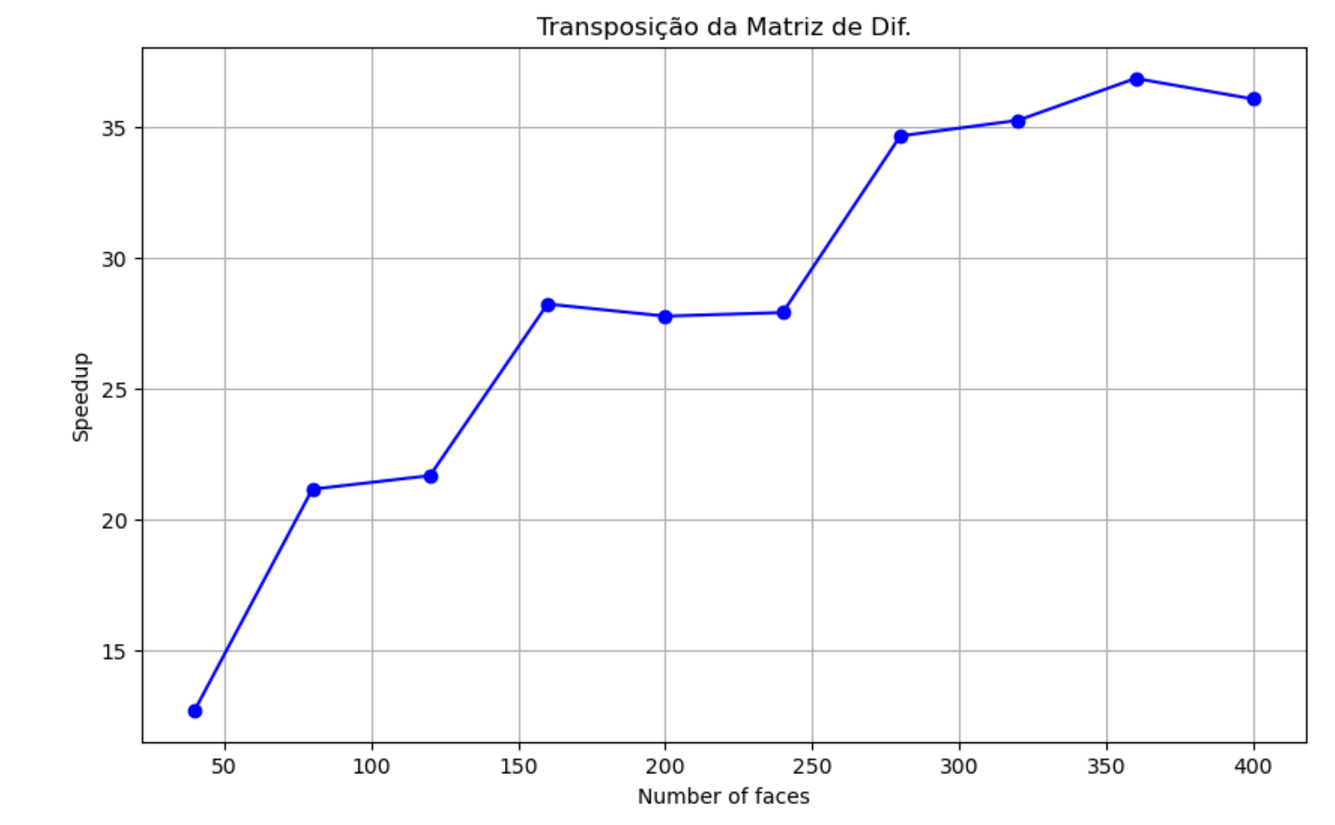
- Speedups de ~28 até ~68
- Etapa de maior speedup em concordância com o paper
- Resultado em comum: pico por volta de ~300 ~360

Paper original



- Speedups de ~2 a ~12

Transposição da Matriz de Diferenças (A^T)



- Speedups de ~5 até ~38

Paper original

- Speedups de ~1,75 até ~3

- Resultado em comum: quanto mais faces, maior o desempenho

Conclusões

- Percebe-se que o código paralelo possui um ganho significativo de performance em determinadas etapas do PCA, com destaque para a etapa da construção do Vetor de Diferenças, entre uma instância e a face média.
- O que demonstra que o ganho performático da paralelização do PCA é válido ainda nas etapas de suposto menor ganho.
- Apesar do resultado ter divergido no Cálculo da Face Média, as outras etapas mostraram ganhos excelentes, superando, como o esperado, os resultados com placas de épocas anteriores.

Referências

CUDA Based Speed Optimization of the PCA Algorithm - https://www.temjournal.com/content/52/TemJournalMay2016_152_159.pdf

CUPY Installation on Windows + Basics - https://www.youtube.com/playlist?list=PLNOdyLYEhS3fOq95SgjgBqjGmljfKXS5g

CuPy Docs - https://docs.cupy.dev/en/stable/reference/index.html

AT&T Database of Faces: The ORL Database of Faces - https://www.kaggle.com/datasets/kasikrit/att-database-of-faces/data

Obrigado!