МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №*3*
по курсу «Программирование графических процессоров»

Классификация и кластеризация изображений на GPU.

Выполнил: *Чистяков К.С.*
Группа: *8О-406Б-21*
Преподаватель: А.Ю. Морозов

Москва, 2025

Условие

1. **Цель работы.** Научиться использовать GPU для классификации и кластеризации изображений. Использование константной памяти и одномерной сетки потоков.
2. **Вариант задания.** Вариант 1. Метод максимального правдоподобия.

Для некоторого пикселя p, номер класса jc определяется следующим образом:

$$jc = arg\,max_j\left[-(p - avg_j)^T * cov_j^{-1} * (p - avg_j) - log(|det(cov_j)|)\right]$$

Пример:

Входной файл	hex: in.data	hex: out.data
in.data out.data 2 4 1 2 1 0 2 2 2 1 4 0 0 0 1 1 1 2 0	03000000 03000000 A2DF4C00 F7C9FE00 9ED84500 B4E85300 99D14D00 92DD5600 A9E04C00 F7D1FA00 D4D0E900	03000000 03000000 A2DF4C01 F7C9FE00 9ED84501 B4E85301 99D14D01 92DD5600 A9E04C01 F7D1FA00 D4D0E900

Входной файл	hex: out.data
in.data out.data 5 4 5 0 0 2 6 1 1 1 6 2 0 7 1 1 0 1 2 6 0 4 0 4 3 0 3 1 0 1 0 0 4 0 3 6 2 5 2 7 2 9 6 4 5 1 7 0 2 1 2 3 4 1 1 5 3 3 2 6	08000000 08000000 D2E27502 CFF65201 D3ED5701 D6E76902 C8F35B01 8E168200 CFF45001 AE977604 D3DC7102 7D1E7B00 AB9A8004 D9E58602 AB967E04 AE9D8004 87058200 D0F95B01 74148000 D0F55901 86136C00 85077400 D6E27702 D3609F03 D1609F03 CC5EA103 CC739D03 7C127F00 AA988804 AFA07D04 D0E37702 7D117A00 D6EB5901 D6E37C02 C9F85701 D655A103 D7EA7402 93127D00 D35BA403 D4DD7902 B0A18404 D6DE7502 D765A900 AD928404 D0D87C02 D7E97F02 CD509E00 CAF85201 CFF75601 CEF45E01 D0E86902 D1D17F02 AD928104 AFA18304 D4DB5C02 88077D00 C6F75701 7D127D00 A99A8E04 C8609E03 D15DA503 AB957E04 AE9A8004 79218100 D065A103 A99E9A04

Программное и аппаратное обеспечение

Графический процессор (Google Colab)

Compute capability	7.5
Name	Tesla T4
Total Global Memory	15828320256

Shared memory per block	49152
Registers per block	65536
Warp size	32
Max threads per block	(1024, 1024, 64)
Max block	(2147483647, 65535, 65535)
Total constant memory	6553
Multiprocessors count	40

Процессор AMD Ryzen 5 4500U with Radeon Graphics (2.38 GHz)

Technology	7 nm
Cores	6
Threads	6
Core Speed	1390 MHz
Cache (L1 Data)	6 x 32 KBytes (8-way)
Cache (L1 Inst.)	6 x 32 KBytes (8-way)
Cache (Level 2)	6 x 512 KBytes (8-way)
Cache (Level 3)	2 x 4 MBytes (16-way)

Оперативная память

Number of modules	2
Module Manuf.	Samsung
Module Size	8 GBytes (total - 16 GBytes)
Generation	DDR4

Жесткий диск - 512 ГБ SSD
OS - Windows 10 Домашняя + Linux
IDE - VS Code + Google Colab
Compiler - nvcc, gcc

Метод решения

Реализованный алгоритм основан на методе максимального правдоподобия, который используется для классификации пикселей изображения на основе его цвета, то есть каждому пикселю изображения присваивается какой-то класс на основе его правдоподобной вероятности.

Для этого мы для каждого класса пикселей вычисляем среднее значение цветов по обучающим точкам, вычисляем ковариационную матрицу цветов для каждого класса, которая описывает разброс данных, а также вычисляем обратную ковариационную матрицу и логарифм детерминанта ковариационной матрицы, чтобы использовать в функции правдоподобия.

По данной нам в условии формуле мы вычисляем для каждого пикселя функцию правдоподобия по каждому классу и выбираем класс с максимальным значением для этой функции.

Описание программы

Вся программа описана в одном файле.

- CSC - для отлова ошибок CUDA;
- void computeAVG(uchar4* pixels, int width, int height, Point* points, int np, double avg[DIM]) - функция для вычисления среднего значения цветов класса
- void computeCovMatrix(uchar4 * pixels, int width, int height, Point * points, int np, double avg[DIM], double variance[DIM][DIM]) - функция для вычисления ковариационной матрицы
- double computeDeterminant(double matrix[3][3]) - функция для вычисления детерминанта матрицы 3 на 3
- double computeLogDeterminant(double variance[DIM][DIM]) - функция для вычисления логарифма детерминанта
- void computeInvMatrix(double variance[DIM][DIM], double inverse[DIM][DIM]) - функция для вычисления обратной матрицы
- __device__ double calculatePropability(uchar4 p, int classIdx) - функция для вычисления правдоподобия
- __device__ int determinePixelClass(uchar4 p, int nc) - функция для определения класса пикселя
- __global__ void kernel(uchar4 * pixels, int width, int height, int nc) - функция ядра(параллельно классифицирует каждый пиксель изображения, использует determinePixelClass для присвоения класса, записывает класс в альфа - канал)
- int main() - главная функция, в которой происходит считывание входных данных, загрузка изображения в память, запуск всех функций для вычисления параметров классов, копирование параметров в постоянную память, выделение памяти на GPU, копирование данных, запуск ядра, запись изображения в файл, а также освобождение памяти.

Ядро:

```
__global__ void kernel(uchar4 *pixels, int width, int height, int nc) {
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   int offsetx = blockDim.x * gridDim.x;
   int size = width * height;
   while (idx < size) {
      pixels[idx].w = determinePixelClass(pixels[idx], nc);
      idx += offsetx;
   }
}
```

Результаты

Замеры времени работы ядер с различными конфигурациями и размерами изображений (512x512, 1024x1024, 2048x2048, 4096x4096). А также приведён замер работы аналогичной программы без использования технологии CUDA - на CPU.

1. **Конфигурация <<< 1, 32 >>>**

Размер исходного изображения	Время (в мс)
512x512	195.779068
1024x1024	520.713074
2048x2048	1568.671631
4096x4096	5647.872559

2. **Конфигурация <<< 128, 64 >>>**

Размер исходного изображения	Время (в мс)
512x512	4.318144
1024x1024	17.202528
2048x2048	68.755646
4096x4096	273.251221

3. **Конфигурация <<< 512, 512 >>>**

Размер исходного изображения	Время (в мс)

512x512	3.504064
1024x1024	13.950720
2048x2048	55.864799
4096x4096	218.769119

4. **Конфигурация <<< 1024, 1024 >>>**

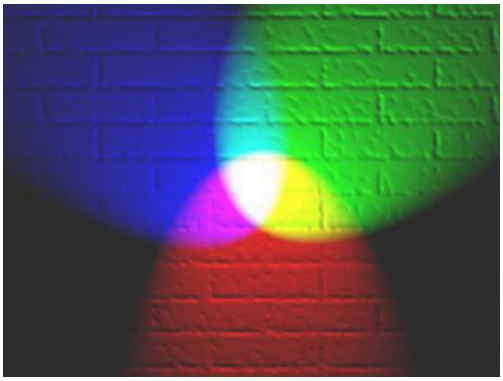
Размер исходного изображения	Время (в мс)
512x512	3.814976
1024x1024	13.956576
2048x2048	55.737503
4096x4096	223.615875

5. **CPU**

Размер исходного изображения	Время (в мс)
512x512	326,173
1024x1024	1322,426
2048x2048	5273,061
4096x4096	23150,196

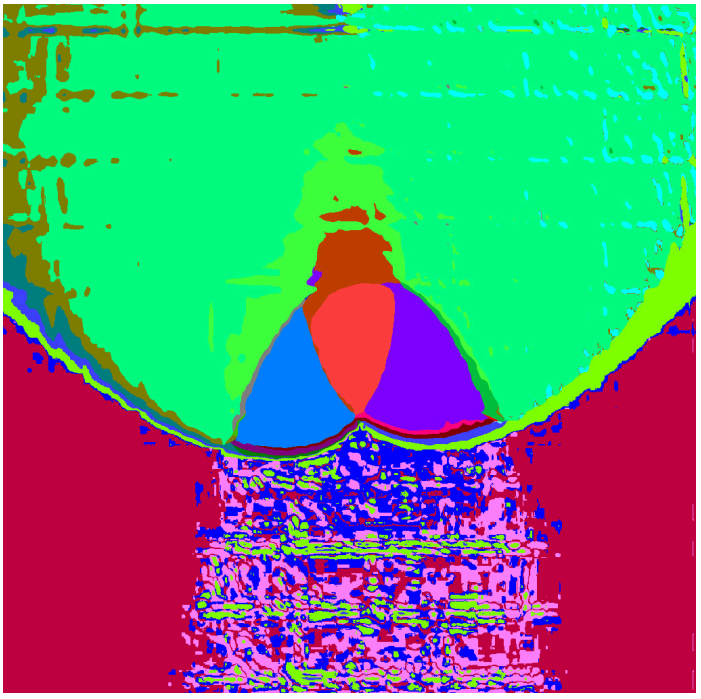
Картинки:

Исходное изображение:



Выходное изображение:

Если чуть изменить kernel и добавить структуру с цветами, то получится поярче, хоть и из-за может быть не очень корректной выборки (рандомной) пикселей некоторые цвета сливаются:
Выходное изображение:



Изменения:
Добавлена структура с различными цветами:

```
uchar4 classColors[MAX_CLASSES] = {
    {255, 0, 0, 255},
```

```
    {0, 255, 0, 255},
    {0, 0, 255, 255},
    {255, 255, 0, 255},
    {0, 255, 255, 255},
    {255, 0, 255, 255},
    {128, 0, 0, 255},
    {0, 128, 0, 255},
    {0, 0, 128, 255},
    {128, 128, 0, 255},
    {0, 128, 128, 255},
    {128, 0, 128, 255},
    {255, 128, 0, 255},
    {128, 255, 0, 255},
    {0, 255, 128, 255},
    {0, 128, 255, 255},
    {128, 0, 255, 255},
    {255, 0, 128, 255},
    {192, 192, 192, 255},
    {128, 128, 128, 255},
    {64, 64, 64, 255},
    {255, 64, 64, 255},
    {64, 255, 64, 255},
    {64, 64, 255, 255},
    {255, 255, 128, 255},
    {128, 255, 255, 255},
    {255, 128, 255, 255},
    {192, 64, 0, 255},
    {64, 192, 0, 255},
    {0, 192, 64, 255},
    {0, 64, 192, 255},
    {192, 0, 64, 255}
};
```

Храним её в константной памяти:

```
__constant__ uchar4 d_classColors[MAX_CLASSES];
```

Изменения в kernel:

```
__global__ void kernel(uchar4* pixels, int width, int height, int nc) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  int offsetx = blockDim.x * gridDim.x;
  int size = width * height;
  while (idx < size) {
    int classIdx = determinePixelClass(pixels[idx], nc);
    pixels[idx] = d_classColors[classIdx];
```

```
    idx += offsetx;
  }
}
```

Ну и в main:
cudaMemcpyToSymbol(d_classColors, classColors, sizeof(classColors));

Выводы

В ходе выполнения лабораторной работы я научился использовать GPU для классификации и кластеризации изображений, а также использовал константную память и одномерную сетку потоков.

Этот алгоритм может применяться в области статического анализа и машинного обучения (как раз таки классификация изображений или оценки параметров распределений), в области экономики (оценка параметров экономических моделей), физике (оценка параметров физических моделей) и т.д.

Сравнивая время работы ядер с различными конфигурациями и размерами исходных изображений, а также аналогичной программы без использования технологии CUDA - на CPU, можно заметить, что программа на CPU работает заметно медленнее даже самой маленькой конфигурации <<< 1, 32 >>>, в сравнении с большими конфигурациями этот разрыв становится ещё больше. Всё потому, что на GPU мы обрабатываем изображение одновременно множеством потоков, за счёт чего и происходит ускорение. В общем, из всех тестов можно сделать вывод, что указанный алгоритм основанный на методе максимального правдоподобия всегда работает значительно быстрее на GPU по сравнению с CPU.