Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №3 по курсу «Программирование графических процессоров»

Классификация и кластеризация изображений на GPU.

Студент: Л. Я. Вельтман
Преподаватель: К. Г. Крашенинников
А. Ю. Морозов
Группа: М8О-407Б
Дата:
Оценка:
Подпись:

Москва, 2020

Условие

Цель работы: Научиться использовать GPU для классификации и кластеризации изображений. Использование константной памяти.

Вариант 4: Метод спектрального угла.

Входные данные: На первой строке задается путь к исходному изображению, на второй – путь к конечному изображению. На следующей строке: число nc – количество классов. Далее идут nc строчек описывающих каждый класс. В начале j-ой строки задается число np_j – количество пикселей в выборке, за ним следуют np_j пар чисел – координаты пикселей выборки. $nc \leqslant 32, np_j \leqslant 2^{19}$, $w * h \leqslant 4 * 10^8$.

Программное и аппаратное обеспечение:
Device Number: 0
Device name: GeForce GT 545
TotalGlobalMem: 3150381056
Const Mem : 65536
Max shared mem for blocks 49152
Max regs per block 32768
Max thread per block 1024
multiProcessorCount : 3
maxThreadsDim 1024 1024 64
maxGridSize 65535 65535 65535
OS: macOS Catalina version 10.15.5
Text Editor: Sublime Text 3

1 Описание

Метод решения
Нужно посчитать значения средних по каждому каналу. Затем для каждого пикселя изображения определяется его класс по формуле:

$$jc = arg\ max_j \left[p^T * \frac{avg_j}{|avg_j|} \right]$$

Описание программы и метод решения

Сконвертируем изображение в бинарное представление при помощи скрипта. Считаем изображение из файла в RAM.

В этой работе нужно использовать константную память. Для моего варианта мне понадобится два массива для хранения значений оценок вектора средних и нормы вектора средних, для их размещения в константной памяти нужно заранее указать их размер, так как динамическое выделение памяти в отличие от глобальной памяти в константной не поддерживается, также нужно указать спецификатор __constant__. Для записи с хоста в константную память используется функция cudaMemcpyToSymbol.

Далее нужно выделить память для нашего изображения (массива) на девайсе, скопировать в нее массив изображения.

Затем вызывается ядро (SpectralAngleMethod) с заданным количеством блоков и нитей. Там вычисляется значение номера класса при помощи метода спектрального угла (угла между средним вектором эталонного класса и вектором классифицируемого пикселя). Вычисление происходит по следующей формуле:

$$D_i(f) = \cos\left[f \wedge \mu_i\right] = \frac{(f, \mu_i)}{\|f\|\|\mu_i\|} \sim \left(f, \frac{\mu}{\|\mu_i\|}\right)$$

Вычисляем норму вектора средних по всем каналам и норму вектора классифицируемого пикселя, эти полученные два значения перемножаются - теперь это наш знаменатель. Вычисляем векторное произведение между средним вектором эталонного класса и вектором классифицируемого пикселя и делим на полученный знаменатель. Далее находится наиболее вероятный класс из всех имеющихся с помощью argmax, то есть - это класс, при котором наблюдается наибольшее значение.

Вычисляем линейный индекс элемента строки исходной матрицы и записываем туда на место альфа канала значение найденного класса.

После вызова ядра данные копируются в массив и освобождается выделенная память.

2 Исходный код

```
1
2  #include <iostream>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <vector>
6  #include <cmath>
7
8
9  #define CSC(call) \
10 do { \
11     cudaError_t res = call; \
12     if (res != cudaSuccess) { \
13         fprintf(stderr, "ERROR in %s:%d. Message: %s\n", \
14                 __FILE__, __LINE__, cudaGetErrorString(res)); \
15         exit(0); \
16     } \
17 } while(0)
18
19
20 __constant__ float constAVG[32][3];
21 __constant__ float constNormaAVG[32];
22
23
24 __device__ void FormulaComputation(float* res, int numClasses, uchar4 curPixel)
25 {
26     float rgb[3];
27     float tmp[3];
28     float sum, denominator;
29
30     float normaPix = sqrt((float)(curPixel.x * curPixel.x + curPixel.y * curPixel.y +
           curPixel.z * curPixel.z));
31
32     for (int curClass = 0; curClass < numClasses; ++curClass)
33     {
34         rgb[0] = curPixel.x * constAVG[curClass][0];
35         rgb[1] = curPixel.y * constAVG[curClass][1];
36         rgb[2] = curPixel.z * constAVG[curClass][2];
37
38         denominator = normaPix * constNormaAVG[curClass];
39         rgb[0] /= denominator;
40         rgb[1] /= denominator;
41         rgb[2] /= denominator;
42
43         sum = 0.0;
44         for (int i = 0; i < 3; ++i) tmp[i] = 0.0;
45
46         for (int i = 0; i < 3; ++i)
```

```
47          {
48              tmp[i] += rgb[0];
49              tmp[i] += rgb[1];
50              tmp[i] += rgb[2];
51              sum += tmp[i];
52          }
53          res[curClass] = sum;
54      }
55  }
56
57
58  __device__ int ArgMax(float* arr, int numClasses)
59  {
60      float maxValue = arr[0];
61      int maxPoint = 0;
62
63      for (int i = 0; i < numClasses; ++i)
64      {
65          if (arr[i] > maxValue)
66          {
67              maxValue = arr[i];
68              maxPoint = i;
69          }
70      }
71      return maxPoint;
72  }
73
74
75  __global__ void SpectralAngleMethod(uchar4* pixels, int width, int height, int
        numClasses)
76  {
77      int idx = blockDim.x * blockIdx.x + threadIdx.x;
78      int idy = blockDim.y * blockIdx.y + threadIdx.y;
79
80      int xOffset = blockDim.x * gridDim.x;
81      int yOffset = blockDim.y * gridDim.y;
82
83      uchar4 curPixel;
84      float res[32];
85
86      for (int x = idx; x < width; x += xOffset)
87      {
88          for (int y = idy; y < height; y += yOffset)
89          {
90              curPixel = pixels[y * width + x];
91              FormulaComputation(res, numClasses, curPixel);
92              pixels[y * width + x].w = ArgMax(res, numClasses);
93          }
94      }
```

```
95  }
96
97
98  int main(int argc, const char* argv[])
99  {
100     std::string input, output;
101     int width, height, numClasses, numPixels;
102     uchar4* pixels;
103
104     std::cin >> input >> output >> numClasses;
105
106     int2 coordinate;
107     std::vector<std::vector<int2>> samples(numClasses);
108
109     for (int i = 0; i < numClasses; ++i)
110     {
111         std::cin >> numPixels;
112         for (int j = 0; j < numPixels; ++j)
113         {
114             std::cin >> coordinate.x >> coordinate.y;
115             samples[i].emplace_back(coordinate);
116         }
117     }
118
119     FILE* file;
120     if ((file = fopen(input.c_str(), "rb")) == NULL)
121     {
122         std::cerr << "ERROR: something wrong with opening the file!\n";
123         exit(0);
124     }
125     else
126     {
127         fread(&width, sizeof(int), 1, file);
128         fread(&height, sizeof(int), 1, file);
129         if (width * height > 400000000)
130         {
131             std::cerr << "ERROR: incorrect input.\n";
132             exit(0);
133         }
134         pixels = new uchar4[width * height];
135         fread(pixels, sizeof(uchar4), width * height, file);
136
137         fclose(file);
138     }
139
140     int numChannels = 3; // rgb
141     int maxElems = 32;
142
143     float avg[maxElems][numChannels];
```

```
144
145     for (int i = 0; i < numClasses; ++i)
146     {
147         avg[i][0] = 0.0;
148         avg[i][1] = 0.0;
149         avg[i][2] = 0.0;
150
151         numPixels = samples[i].size();
152         for (int j = 0; j < numPixels; ++j)
153         {
154             coordinate.x = samples[i][j].x;
155             coordinate.y = samples[i][j].y;
156             avg[i][0] += pixels[coordinate.y * width + coordinate.x].x;
157             avg[i][1] += pixels[coordinate.y * width + coordinate.x].y;
158             avg[i][2] += pixels[coordinate.y * width + coordinate.x].z;
159         }
160         avg[i][0] /= numPixels;
161         avg[i][1] /= numPixels;
162         avg[i][2] /= numPixels;
163     }
164
165
166     float normaAvg[32];
167     for (int i = 0; i < numClasses; ++i)
168     {
169         normaAvg[i] = std::sqrt(avg[i][0] * avg[i][0] + avg[i][1] * avg[i][1] + avg[i
                ][2] * avg[i][2]);
170     }
171
172     CSC(cudaMemcpyToSymbol(constAVG, avg, sizeof(float) * maxElems * numChannels));
173     CSC(cudaMemcpyToSymbol(constNormaAVG, normaAvg, sizeof(float) * maxElems));
174
175     uchar4* deviceRes;
176     CSC(cudaMalloc(&deviceRes, sizeof(uchar4) * width * height));
177     CSC(cudaMemcpy(deviceRes, pixels, sizeof(uchar4) * width * height,
            cudaMemcpyHostToDevice));
178
179     int xThreadCount = 16;
180     int yThreadCount = 16;
181
182     int xBlockCount = 16;
183     int yBlockCount = 16;
184
185     dim3 blockCount = dim3(xBlockCount, yBlockCount);
186     dim3 threadsCount = dim3(xThreadCount, yThreadCount);
187
188     SpectralAngleMethod<<<blockCount, threadsCount>>>(deviceRes, width, height,
            numClasses);
189     CSC(cudaGetLastError());
```

```
190
191     CSC(cudaMemcpy(pixels, deviceRes, sizeof(uchar4) * width * height,
            cudaMemcpyDeviceToHost));
192
193
194     if ((file = fopen(output.c_str(), "wb")) == NULL)
195     {
196         std::cerr << "ERROR: something wrong with opening the file.";
197         exit(0);
198     }
199     else
200     {
201         fwrite(&width, sizeof(int), 1, file);
202         fwrite(&height, sizeof(int), 1, file);
203         fwrite(pixels, sizeof(uchar4), width * height, file);
204
205         fclose(file);
206     }
207
208     CSC(cudaFree(deviceRes));
209
210     delete[] pixels;
211     return 0;
212 }
```

3 Результаты

```
Количество классов 3,в каждой выборке по 4 пикселя.
Исходный размер: 640x360

CPU
time = 41.578
```

blocks	threads	time
blocks = (4, 4)	threads = (4, 4)	3.738208
blocks = (4, 4)	threads = (16, 16)	0.771584
blocks = (16, 16)	threads = (16, 16)	0.689440
blocks = (16, 16)	threads = (32, 32)	0.811584
blocks = (32, 32)	threads = (16, 16)	0.735520
blocks = (32, 32)	threads = (32, 32)	0.971040

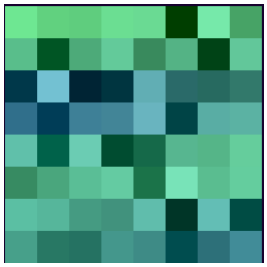
```
Количество классов 3,в каждой выборке по 4 пикселя.
Исходный размер: 5760x3840

CPU
time = 2425.51
```

blocks	threads	time
blocks = (4, 4)	threads = (4, 4)	367.511566
blocks = (4, 4)	threads = (16, 16)	71.320831
blocks = (16, 16)	threads = (16, 16)	61.579872
blocks = (16, 16)	threads = (32, 32)	60.696831
blocks = (32, 32)	threads = (16, 16)	61.897217
blocks = (32, 32)	threads = (32, 32)	62.587967

Для более видимой визуализации полученного результата работы программы вместо альфа канала я записывала полученный наиболее вероятный класс на место красного канала (RGB).

orig/alpha/red



4 Выводы

Для выполнения данной лабораторной работы нужно было реализовать алгоритм классификации изображений - метод спектрального угла. Метод спектрального угла использует только направление векторов, поэтому он не чувствителен к абсолютной яркости пикселей, так как меру их яркости определяет именно длина вектора. Все возможные яркости при этом обрабатываются одинаково, поскольку пиксели, обладающие более низкой яркостью, просто расположены ближе к началу координат диаграммы рассеяния. Цвет пикселей, соответствующий их классу в n-мерном пространстве признаков определяется направлением их радиус-векторов.

Находится спектральный угол, если он меньше заданного (обычно берут максимальное значение спектрального угла), то пиксель попадает в класс эталона, с которым идет сравнение.

Этот алгоритм является алгоритмом классификации с обучением, поэтому нужно вручную задавать классы и выборку, состоящую из пикселей, которые относятся к этому классу, что не является огромным плюсом.