Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Архитектура процессоров и технология CUDA

ОТЧЕТ

по лабораторной работе №5

на тему

ОПТИМИЗАЦИЯ АЛГОРИТМА ДЛЯ РАБОТЫ НА GPU

Студенты: Е.А. Петрович

М.А. Ходосевич

Преподаватель: Т.С. Жук

МИНСК 2024

**1 ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ**

**1.1 Цель работы**

Изучить способы оптимизации алгоритма для работы на GPU применительно к задаче обработки изображений.

**1.2 Исходные данные к работе**

Реализовать заданный по варианту фильтр на CPU и GPU. Сравнить производительность двух типов архитектур.

Обязательные требования:

1. Провести поиск проблемных участков кода и устранение данных недостатков. Выполнить фильтрацию входного изображения.

2. Оптимизация выбранного алгоритма с учетом особенностей доступа к глобальной памяти.

3. Маска фильтра задается по варианту.

4. На вход приходит цветное и в оттенках серого изображение, соответственно представленный алгоритм должен уметь адаптироваться к типу изображения (можно использовать `OpenCV` самой последней версии). Допускается реализация только для изображений в оттенках серого.

5. Размер входного изображения: не определен. Не допускается работать с квадратным изображением и с изображением кратным размеру grid и block. Изменять размер исходного изображения (например: добавлять и убирать фиктивные границы) запрещено.

6. Тип данных: `unsigned char`.

7. Использование текстурной памяти запрещено. Использование дополнительных массивов запрещено.

Опциональные условия:

1. Реализация как для изображений в оттенках серого, так и для цветных. При этом программа должна сама определять тип изображения и обрабатывать его соответствующим образом.

2. Использовать функционал `loadPPM` в CUDA Samples. Использовать NVIDIA© Visual Profiler.

**2 ВЫПОЛНЕНИЕ РАБОТЫ**

В лабораторной работе необходимо реализовать обработку изображения с помощью минимального фильтра 5х5. Минимальные и максимальные фильтры служат для морфологического сужения или расширения соответственно. То есть, для изображения выбирает пиксель с минимальной или максимальной интенсивностью из окрестности. В результате применения максимального фильтра происходит увеличения ярких объектов, бликов, при применении минимального фильтра увеличиваются темные участки.

Исходя из предоставленной выше информации, были сделаны предположения о том, что при применении минимального фильтра 5х5 выходное изображение будет более темное и «размыленное». За «размыленность» (или расплывчатость) выходного изображения отвечает размер маски. Чем больше маска, тем больше расплывется изображение.

В результате было написано работоспособное приложение, которое применяет фильтр на входное изображение, получает и сохраняет выходное. Обработка изображений происходит с использованием OpenCV. Программа работает как с цветными изображениями, так и с черно-белыми. Количество каналов (а соответственно и тип изображения) определяется автоматически с помощью функции channels() (OpenCV). На рисунке 2.1 будет приведен результат работы с черно-белым изображением. В таблице 2.1 приведены временные результаты работы с цветными и черно-белыми изображениями. Листинг кода приведен в приложении А.



*а*



*б*

Рисунок 2.1 – Результаты работы: *а* – входное изображение; *б* – выходное изображение

Таблица 2.1 – Результаты работы

|  |  |  |  |
| --- | --- | --- | --- |
| Изображение | CPU, секунд | GPU, секунд | GPU c memcpy, секунд |
| Ч/б | 0,239 | 0,011 | 0,014 |
| Цветное | 0,603 | 0,024 | 0,028 |

**3 ВЫВОДЫ**

В ходе лабораторной работы были изучены способы оптимизации алгоритма для работы на GPU применительно к задаче обработки изображений. В соответствии с выданным вариантом (минимальный фильтр) был реализован алгоритм, который применяет данный фильтр на изображение. Все обязательные требования соблюдены. По окончании фильтрации выводятся результаты. Так как все требования выполнены, а результаты работы представлены, все отображается корректно, можно сделать вывод, что лабораторная работа проведена успешно.

**ПРИЛОЖЕНИЕ А**

*(обязательное)*

Исходный текст программы

Содержимое файла kernel.cu:

1. #include <cuda\_runtime.h>
2. #include "device\_launch\_parameters.h"
3. #include <chrono>
4. #include <iostream>
5. #include <vector>
6. #include <opencv2/opencv.hpp>
7. #include <opencv2/imgproc.hpp>
8. #define MASK\_WIDTH 5
9. #define MASK\_RADIUS (MASK\_WIDTH / 2)
10. #define THREADS\_PER\_BLOCK 32
11. bool loadImage(const std::string& filename, int& width, int& height, std::vector<unsigned char>& data, bool& isGrayscale) {
12. cv::Mat img = cv::imread(filename, cv::IMREAD\_UNCHANGED);
13. if (img.empty()) return false;
14. isGrayscale = (img.channels() == 1);
15. width = img.cols;
16. height = img.rows;
17. if (!isGrayscale) {
    * 1. cv::cvtColor(img, img, cv::COLOR\_BGR2RGB);
18. }
19. data.assign(img.data, img.data + img.total() \* img.elemSize());
20. return true;
21. }
22. bool saveImage(const std::string& filename, int width, int height, const std::vector<unsigned char>& data, bool isGrayscale) {
23. int channels = isGrayscale ? 1 : 3;
24. cv::Mat img(height, width, isGrayscale ? CV\_8UC1 : CV\_8UC3, const\_cast<unsigned char\*>(data.data()));
25. if (!isGrayscale) {
    * 1. cv::cvtColor(img, img, cv::COLOR\_RGB2BGR);
26. }
27. return cv::imwrite(filename, img);
28. }
29. void applyMinFilterCPU(const std::vector<unsigned char>& input, std::vector<unsigned char>& output, int width, int height) {
30. for (int y = 0; y < height; ++y) {
    * 1. for (int x = 0; x < width; ++x) {
      2. if (x >= MASK\_RADIUS && x < width - MASK\_RADIUS && y >= MASK\_RADIUS && y < height - MASK\_RADIUS) {
         1. for (int c = 0; c < 3; ++c) {
            1. int min\_val = 255;
            2. for (int dy = -MASK\_RADIUS; dy <= MASK\_RADIUS; ++dy) {

for (int dx = -MASK\_RADIUS; dx <= MASK\_RADIUS; ++dx) {

int pixel = input[((y + dy) \* width + (x + dx)) \* 3 + c];

min\_val = std::min(min\_val, pixel);

}

* + - * 1. }
        2. output[(y \* width + x) \* 3 + c] = static\_cast<unsigned char>(min\_val);
      1. }
    1. }
    2. }

1. }
2. }
3. void applyMinFilterCPU\_Grayscale(const std::vector<unsigned char>& input, std::vector<unsigned char>& output, int width, int height) {
4. for (int y = 0; y < height; ++y) {
   * 1. for (int x = 0; x < width; ++x) {
     2. if (x >= MASK\_RADIUS && x < width - MASK\_RADIUS && y >= MASK\_RADIUS && y < height - MASK\_RADIUS) {
        1. int min\_val = 255;
        2. for (int dy = -MASK\_RADIUS; dy <= MASK\_RADIUS; ++dy) {
           1. for (int dx = -MASK\_RADIUS; dx <= MASK\_RADIUS; ++dx) {

int pixel = input[(y + dy) \* width + (x + dx)];

min\_val = std::min(min\_val, pixel);

* + - * 1. }
      1. }
      2. output[y \* width + x] = static\_cast<unsigned char>(min\_val);
    1. }
    2. }

1. }
2. }
3. \_\_global\_\_ void applyMinFilterCUDA(unsigned char\* input, unsigned char\* output, int width, int height) {
4. int x = blockIdx.x \* blockDim.x + threadIdx.x;
5. int y = blockIdx.y \* blockDim.y + threadIdx.y;
6. if (x >= MASK\_RADIUS && x < width - MASK\_RADIUS && y >= MASK\_RADIUS && y < height - MASK\_RADIUS) {
   * 1. for (int c = 0; c < 3; ++c) {
     2. int min\_val = 255;
     3. for (int dy = -MASK\_RADIUS; dy <= MASK\_RADIUS; ++dy) {
        1. for (int dx = -MASK\_RADIUS; dx <= MASK\_RADIUS; ++dx) {
           1. int pixel = input[((y + dy) \* width + (x + dx)) \* 3 + c];
           2. min\_val = min(min\_val, pixel);
        2. }
     4. }
     5. output[(y \* width + x) \* 3 + c] = static\_cast<unsigned char>(min\_val);
     6. }
7. }
8. }
9. \_\_global\_\_ void applyMinFilterCUDA\_Grayscale(unsigned char\* input, unsigned char\* output, int width, int height) {
10. int x = blockIdx.x \* blockDim.x + threadIdx.x;
11. int y = blockIdx.y \* blockDim.y + threadIdx.y;
12. if (x >= MASK\_RADIUS && x < width - MASK\_RADIUS && y >= MASK\_RADIUS && y < height - MASK\_RADIUS) {
    * 1. int min\_val = 255;
      2. for (int dy = -MASK\_RADIUS; dy <= MASK\_RADIUS; ++dy) {
      3. for (int dx = -MASK\_RADIUS; dx <= MASK\_RADIUS; ++dx) {
         1. int pixel = input[(y + dy) \* width + (x + dx)];
         2. min\_val = min(min\_val, pixel);
      4. }
      5. }
      6. output[y \* width + x] = static\_cast<unsigned char>(min\_val);
13. }
14. }
15. int main() {
16. int width, height;
17. bool isGrayscale;
18. std::string path = "3.jpg";
19. std::vector<unsigned char> image;
20. if (!loadImage(path, width, height, image, isGrayscale)) {
    * 1. std::cerr << "Failed to load image!" << std::endl;
      2. return -1;
21. }
22. std::vector<unsigned char> outputCPU(image.size());
23. std::vector<unsigned char> outputGPU(image.size());
24. auto startCPU = std::chrono::high\_resolution\_clock::now();
25. if (isGrayscale) {
    * 1. applyMinFilterCPU\_Grayscale(image, outputCPU, width, height);
26. }
27. else {
    * 1. applyMinFilterCPU(image, outputCPU, width, height);
28. }
29. auto endCPU = std::chrono::high\_resolution\_clock::now();
30. std::chrono::duration<double> durationCPU = endCPU - startCPU;
31. std::cout << "CPU time: " << durationCPU.count() << " seconds\n";
32. unsigned char\* d\_input, \* d\_output;
33. cudaMalloc(&d\_input, image.size());
34. cudaMalloc(&d\_output, image.size());
35. auto startGPUcpy = std::chrono::high\_resolution\_clock::now();
36. cudaMemcpy(d\_input, image.data(), image.size(), cudaMemcpyHostToDevice);
37. dim3 blockSize(THREADS\_PER\_BLOCK, THREADS\_PER\_BLOCK);
38. dim3 gridSize((width + THREADS\_PER\_BLOCK - 1) / THREADS\_PER\_BLOCK, (height + THREADS\_PER\_BLOCK - 1) / THREADS\_PER\_BLOCK);
39. auto startGPU = std::chrono::high\_resolution\_clock::now();
40. if (isGrayscale) {
    * 1. applyMinFilterCUDA\_Grayscale << <gridSize, blockSize >> > (d\_input, d\_output, width, height);
41. }
42. else {
    * 1. applyMinFilterCUDA << <gridSize, blockSize >> > (d\_input, d\_output, width, height);
43. }
44. cudaDeviceSynchronize();
45. auto endGPU = std::chrono::high\_resolution\_clock::now();
46. std::chrono::duration<double> durationGPU = endGPU - startGPU;
47. std::cout << "GPU time: " << durationGPU.count() << " seconds\n";
48. cudaMemcpy(outputGPU.data(), d\_output, image.size(), cudaMemcpyDeviceToHost);
49. auto endGPUcpy = std::chrono::high\_resolution\_clock::now();
50. std::chrono::duration<double> durationGPUcpy = endGPUcpy - startGPUcpy;
51. std::cout << "GPU memcpy time: " << durationGPUcpy.count() << " seconds\n";
52. bool match = true;
53. for (size\_t i = 0; i < image.size(); ++i) {
    * 1. if (outputCPU[i] != outputGPU[i]) {
      2. match = false;
      3. std::cout << "Mismatch at pixel " << i << ": CPU = " << static\_cast<int>(outputCPU[i]) << ", GPU = " << static\_cast<int>(outputGPU[i]) << "\n";
      4. break;
      5. }
54. }
55. if (match) {
    * 1. std::cout << "CPU and GPU results match.\n";
56. }
57. else {
    * 1. std::cout << "CPU and GPU results do not match.\n";
58. }
59. saveImage("outputCPU.jpg", width, height, outputCPU, isGrayscale);
60. saveImage("outputGPU.jpg", width, height, outputGPU, isGrayscale);
61. cudaFree(d\_input);
62. cudaFree(d\_output);
63. return 0;
64. }