Міністерство освіти і науки України

Прикарпатський національний університет імені Василя Стефаника

кафедра комп'ютерних наук та інформаційних систем

Архітектура комп'ютерів

Лабораторна робота №2

Дослідження паралельності архітектури на рівні ядра процесора

(звіт)

Виконав студент гр. ІПЗ-21

Коваль Сергій

Перевірив

д.т.н, проф.. Стрілецький Ю.Й.

_____

Івано-Франківськ 2024

Відповідно до мети було досліджено ефективність використання паралелізму на рівні ядра процесора для обробки масиву даних, в результаті чого, було виконано завдання зазначені нижче.

1 Згортка ядра із зображенням

Для дослідження ефективності однопоточного процесу було написано програму, що виконує згортку ядра із зображенням.

Згортка - це по-піксельної обробки зображення в процесі чого утворюється нове зображення що є репрезентацією фільтру значень матриці. У моєму випадку матриця містить лише ва ненульові елементи, а саме A0,0 =-1 та A1,1.=1 Що означає що результатом згортки буде дуже сильне збільшення контрастності зображення. Усі пікселі зображення, верхній лівий сусід, яких має значення таке ж як і в них перетворяться у нуль.

Фрагмент коду 1.1 Ядро зображення (матриця

```
int kernel[3][3] = {
  {-1, 0, 0},
  {0, 1,0},
  {0, 0, 0}
};
```

1.1 Зчитування даних з файлу

Першим етапом є зчитування даних з файлу за допомогою бібліотеки fstream. Перш за все, опрацьовуються розмір зображення, що необхідно для коректної обробки вмісту файлу. Після чого, завантажуються пікселі зображення у двовимірний масив. Оскільки формат файлу є pgm, то розділення на кольорові канали немає.

1.2 Проходження по пікселях зображення

Програма виконує ітерацію по окремих пікселях зображення записаних у двовимірний масив де кожна комірка зовнішнього масиву містить масив що складається із пікселів відповідного рядка зображення. Ітерація проходить від (1;1) і до (HEIGHT-2; WIDTH-2). Не зачіпаючи пікселі розташовані скраю.

Фрагмент коду 1.2 реалізаця обходу масиву зображення

```
for (int i = 1; i < height - 1; i++) {
        for (int j = 1; j < width - 1; j++) {
```

```
                //обчислення згортки
                //запис у вихідний масив
        }
}
```

1.3 Обчислення згортки

У внутрішньому циклі розміщено ще два цикли, всередині яких відбувається обчислення матриці згортки. Це обчислення полягає в накладанні відповідних значень комірок ядра (матриці) на відповідні сусідні пікселі зображення, після чого результат записується в масив, що і буде кінцевим результатом, який потім буде записано у файл.

Фрагмент коду 1.3 реалізація обчислення згортки

```
sum = 0;
for (int ki = -1; ki <= 1; ki++) {
        for (int kj = -1; kj <= 1; kj++) {
                sum += img[i + ki][j + kj] * kernel[ki + 1][kj + 1];
        }
}
result[i][j] = sum / 9;
```

1.4 Перенесення Країв зображення у масив нового зображення

Після маніпуляцій з обчислення згорток необхідно перемістити перші і останні пікселі з вихідного масиву у масив нового зображення. Це завдання виконують два цикли що виконуються перед записом результату у файл.

Фрагмент коду 1.4 перенесення країв оригінального зображення

```
 for (int i = 0; i < HEIGHT; i++) {
     result[i][0] = img[i][0];
     result[i][w - 1] = img[i][WIDTH - 1];
   }
   for (int j = 0; j < WIDTH; j++) {
     result[0][j] = img[0][j];
     result[HEIGHT - 1][j] = img[HEIGHT - 1][j];
   }
```

Примітка: код усієї програми знаходиться у додатку 1

2 Згортка із використанням паралельності

Для досягнення паралельності функцію обчислення згортки було переписано використовуючи SIMD інструкції.

2.1 Створення вектору для ядра та відповідних пікселів зображення

Першою зміною є створення вектору для зберігання значень ядра у вигляді 128-бітного вектору. Використовуються значення ядра фільтра (матриця 3x3) і додаються нулі, щоб заповнити весь 256-бітовий вектор, те ж саме було зроблено зі значеннями пікселів у вкладеному циклі для ітерації по комірках масиву зображення.

Фрагмент коду 1.3 Створення вектору для ядра та, у вкладеному циклі, відповідних пікселів зображення

```
__m256i expanded_kernel = _mm256_set_epi8(
   kernel[2][2], kernel[2][1], kernel[2][0],
   kernel[1][2], kernel[1][1], kernel[1][0],
   kernel[0][2], kernel[0][1], kernel[0][0],
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
);

for (int i = 1; i < height - 1; i++) {
  for (int j = 1; j < width - 1; j++) {
     __m256i expanded_pixels = _mm256_set_epi8(
          img[i + 1][j + 1], img[i + 1][j], img[i + 1][j - 1],
          img[i][j + 1], img[i][j], img[i][j - 1],
```

```
            img[i - 1][j + 1], img[i - 1][j], img[i - 1][j - 1],
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        );
        //Логіка обчислення
        // Запис у вихідний масив
        }
        }
```

2.2 Обчислення згортки за допомогою SIMD інструкцій

Після створення векторів, вони конвертуються у 128-бітові вектори. Потім ці вектори множаться, і результат розділяється на дві частини. Кожна частина обробляється за допомогою горизонтального додавання. Результати обробки двох частин об'єднуються у єдину суму, використовуючи додаткову операцію додавання. З отриманої суми витягуються перші два значення, які відповідають кінцевим результатам для цієї комірки зображення. Ці значення додаються для отримання фінальної величини, яка записується у відповідну комірку нового зображення.

Фрагмент коду 2.2 Обчислення згортки за допомогою SIMD шнструкцій

```
__m128i pixels = _mm_set_epi8(
    img[i - 1][j - 1], img[i - 1][j], img[i - 1][j + 1],
    img[i][j - 1],  img[i][j], img[i][j + 1],
    img[i + 1][j - 1], img[i + 1][j], img[i + 1][j + 1],
    0, 0, 0, 0, 0, 0, 0
);
__m256i expanded_pixels = _mm256_cvtepu8_epi16(pixels);
__m256i expanded_kernel = _mm256_cvtepu8_epi16(kernel_reg);

__m256i product = _mm256_mullo_epi16(expanded_pixels, expanded_kernel);
```

```
__m128i result_low = _mm256_extractf128_si256(product, 0);

__m128i result_high = _mm256_extractf128_si256(product, 1);


result_low = _mm_hadd_epi16(result_low, result_low);

result_low = _mm_hadd_epi16(result_low, result_low);

result_high = _mm_hadd_epi16(result_high, result_high);

result_high = _mm_hadd_epi16(result_high, result_high);

__m128i final_sum = _mm_add_epi16(result_low, result_high);

int result_value = _mm_extract_epi16(final_sum, 0) +
_mm_extract_epi16(final_sum, 1);

result[i][j] = ( result_value);
```

Примітка: код усієї програми знаходиться у додатку 2

3 Зображення

Згортки було протестовано на першому зображенні формату pgm (3.1 рис.).

Результатом виконання згортки без використання паралельності є зображення посередині (3.2 рис), і відповідно останнє зображення (3.3 рис) було отримано у результаті виконання програми з використанням паралельності.

рис. 3.1

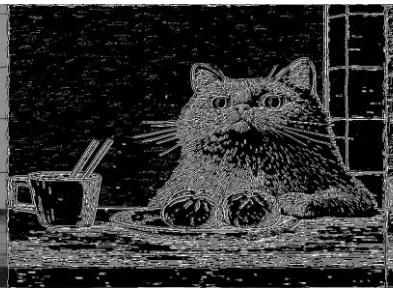

рис. 3.2

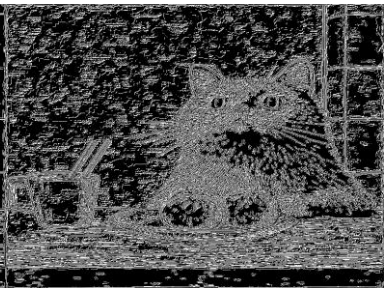

рис. 3.3

4 Час

Для вимірювання часу було використано бібліотеку “time.h”.

Першу програму було виконано за 16 мілісекунд (рис. 4.1) у той же час як програма із використанням паралельності змогла закінчити своє виконанння на 2 мілісекунди швидше (рис. 4.2).

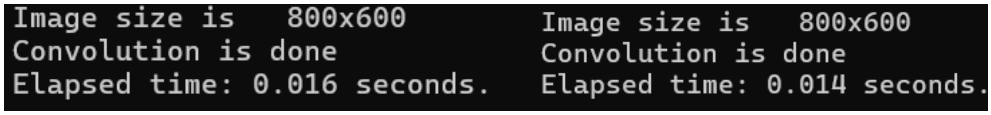

рис 4.1

рис 4.2

Висновок

У цій лабораторній роботі я вивчив алгоритм згортки зображення з ядром 3x3 для виділення особливостей. Також я освоїв використання асемблерних інструкцій AVX2 для оптимізації процесу згортки. Використовуючи SIMD-команди (Single Instruction, Multiple Data), я зміг виконувати паралельні операції множення та додавання, що пришвидшило виконання алгоритму. Це дозволило мені порівняти час виконання традиційної програми та програми з оптимізацією і побачити, як використання паралельних обчислень може зменшити час обробки зображень великих розмірів.

Додаток 1

```
#include <iostream>
#include <fstream>
#include <string>
#include <time.h>
using namespace std;

const int WIDTH = 800;
const int HEIGHT = 600;

void read_pgm_image(const string& filename, unsigned char img[HEIGHT][WIDTH], int& width, int& height, int& max_intensity) {
  ifstream file(filename, ios::binary);

  if (file.is_open()) {
    string magic, creator_info;
    file >> magic;
    getline(file, creator_info);
    file >> width >> height >> max_intensity;

    if (width != WIDTH || height != HEIGHT) {
      cout << "Error: Image size does not match 800x600.\n";
      file.close();
      return;
    }

    file.get();

    for (int i = 0; i < HEIGHT; i++) {
      for (int j = 0; j < WIDTH; j++) {
        img[i][j] = file.get();
```

```
        }
      }
      file.close();
    }
    else {
      cout << "Error opening the file!\n";
    }
}

void write_pgm_image(const string& filename, unsigned char img[HEIGHT][WIDTH], int width, int height, int max_intensity) {
    ofstream file(filename, ios::binary);

    if (file.is_open()) {
      file << "P5\n" << width << ' ' << height << '\n' << max_intensity << '\n';

      for (int i = 0; i < HEIGHT; i++) {
        for (int j = 0; j < WIDTH; j++) {
          file.put(img[i][j]);
        }
      }
      file.close();
    }
    else {
      cout << "Error creating the file!" << endl;
    }
}

void convolve(unsigned char img[HEIGHT][WIDTH], unsigned char result[HEIGHT][WIDTH], int width, int height) {
    int kernel[3][3] = {
      {1, 0, 0},
```

```
        {0, -1, 0},
        {0, 0, 0}
    };

    int sum;

    for (int i = 1; i < height - 1; i++) {
        for (int j = 1; j < width - 1; j++) {
            sum = 0;

            for (int ki = -1; ki <= 1; ki++) {
                for (int kj = -1; kj <= 1; kj++) {
                    sum += img[i + ki][j + kj] * kernel[ki + 1][kj + 1];
                }
            }
            result[i][j] = sum/4;
        }
    }
    for (int i = 0; i < HEIGHT; i++) {
        result[i][0] = img[i][0];
        result[i][WIDTH - 1] = img[i][WIDTH - 1];
    }
    for (int j = 0; j < WIDTH; j++) {
        result[0][j] = img[0][j];
        result[HEIGHT - 1][j] = img[HEIGHT - 1][j];
    }
}

int main() {
    clock_t start_ticks, end_ticks;

    string input_image = "C:/Users/Сергій/Desktop/test.pgm";
```

```
    string output_image = "C:/Users/Сергій/Desktop/result.pgm";

    int width, height, max_intensity;

    unsigned char img[HEIGHT][WIDTH];
    unsigned char result[HEIGHT][WIDTH];

    read_pgm_image(input_image, img, width, height, max_intensity);

    start_ticks = clock();
    convolve(img, result, width, height);
    end_ticks = clock();

    write_pgm_image(output_image, result, width, height, max_intensity);

    cout << "Image size is\t" << WIDTH << "x" << HEIGHT << endl;
    cout << "Convolution is done\n";
    cout << "Elapsed time: " << double(end_ticks - start_ticks) / CLOCKS_PER_SEC << " seconds.\n";

    return 0;
}
```

Додаток 2

```
#include <iostream>
#include <fstream>
#include <string>
#include <stdio.h>
#include <time.h>
```

```
#include <immintrin.h>  // Для AVX2
const int WIDTH = 800;
const int HEIGHT = 600;
int kernel[3][3] = {
  {-1, 0, 0},
  {0, 1,0},
  {0, 0, 0}
};

void read_pgm_image(const std::string& filename, unsigned char
img[HEIGHT][WIDTH], int& width, int& height, int& max_intensity) {
  std::ifstream file(filename, std::ios::binary);

  if (file.is_open()) {
    std::string magic, creator_info;
    file >> magic;
    std::getline(file, creator_info);
    file >> width >> height >> max_intensity;

    if (width != WIDTH || height != HEIGHT) {
      std::cout << "Error: Image size does not match 800x600.\n";
      file.close();
      return;
    }

    file.get();

    for (int i = 0; i < HEIGHT; i++) {
```

```
        for (int j = 0; j < WIDTH; j++) {
            img[i][j] = file.get();
        }
    }
    file.close();
  }
  else {
    std::cout << "Error opening the file!\n";
  }
}

void write_pgm_image(const std::string& filename, unsigned char
img[HEIGHT][WIDTH], int width, int height, int max_intensity) {
  std::ofstream file(filename, std::ios::binary);

  if (file.is_open()) {
    file << "P5\n" << width << ' ' << height << '\n' << max_intensity << '\n';

    for (int i = 0; i < HEIGHT; i++) {
      for (int j = 0; j < WIDTH; j++) {
        file.put(img[i][j]);
      }
    }
    file.close();
  }
  else {
    std::cout << "Error creating the file!" << std::endl;
  }
```

```
}

int scale(int value, int min_old, int max_old, int min_new, int max_new) {
   if (max_old == min_old) {  // Ensure no division by zero
      return min_new;  // Or handle this case as needed
   }
   return (value - min_old) * (max_new - min_new) / (max_old - min_old) + min_new;
}

void convolve(unsigned char img[HEIGHT][WIDTH], unsigned char
result[HEIGHT][WIDTH] ) {

   __m128i kernel_reg = _mm_set_epi8(
      kernel[0][0], kernel[0][1], kernel[0][2],
      kernel[1][0], kernel[1][1], kernel[1][2],
      kernel[2][0], kernel[2][1], kernel[2][2],
      0, 0, 0, 0, 0, 0, 0
   );

   for (int i = 1; i < HEIGHT - 1; i++) {
      for (int j = 1; j < WIDTH- 1; j++) {
         __m128i pixels = _mm_set_epi8(
            img[i - 1][j - 1], img[i - 1][j], img[i - 1][j + 1],
            img[i][j - 1],  img[i][j], img[i][j + 1],
            img[i + 1][j - 1], img[i + 1][j], img[i + 1][j + 1],
            0, 0, 0, 0, 0, 0, 0
         );
         __m256i expanded_pixels = _mm256_cvtepu8_epi16(pixels);
```

```
        __m256i expanded_kernel = _mm256_cvtepu8_epi16(kernel_reg);

        __m256i product = _mm256_mullo_epi16(expanded_pixels, expanded_kernel);
        __m128i result_low = _mm256_extractf128_si256(product, 0);
        __m128i result_high = _mm256_extractf128_si256(product, 1);

        result_low = _mm_hadd_epi16(result_low, result_low);
        result_low = _mm_hadd_epi16(result_low, result_low);
        result_high = _mm_hadd_epi16(result_high, result_high);
        result_high = _mm_hadd_epi16(result_high, result_high);

        __m128i final_sum = _mm_add_epi16(result_low, result_high);
        int result_value = _mm_extract_epi16(final_sum, 0) +
_mm_extract_epi16(final_sum, 1);
        result[i][j] = ( result_value);

      }
    }
    for (int i = 0; i < HEIGHT; i++) {
      result[i][0] = img[i][0];
      result[i][WIDTH - 1] = img[i][WIDTH - 1];
    }

    for (int j = 0; j < WIDTH; j++) {
      result[0][j] = img[0][j];
      result[HEIGHT - 1][j] = img[HEIGHT - 1][j];
    }
}
```

```
int main() {
    clock_t start_ticks, end_ticks;

    std::string input_image = "C:/Users/Сергій/Desktop/test.pgm";
    std::string output_image = "C:/Users/Сергій/Desktop/result2.pgm";
    std::cout << "Convolution is done\n";

    int width, height, max_intensity;

    unsigned char img[HEIGHT][WIDTH];
    unsigned char result[HEIGHT][WIDTH];

    read_pgm_image(input_image, img, width, height, max_intensity);

    start_ticks = clock();
    convolve(img, result);
    end_ticks = clock();

    write_pgm_image(output_image, result, width, height, max_intensity);

    std::cout << "Image size is\t" << WIDTH << "x" << HEIGHT << std::endl;
    std::cout << "Convolution is done\n";
    std::cout << "Elapsed time: " << double(end_ticks - start_ticks) / CLOCKS_PER_SEC
<< " seconds.\n";

    return 0;
}
```