Dokumentacja

Opis problemu — cel projektu

Analiza problemu i teoretyczna propozycja jego rozwiązania, opis użytych algorytmów

- Opis typów algorytmów
  - Przekształcenia punktowe
  - Przekształcenia kontekstowe
  - Przekształcenia morfologiczne
  - Dostępne algorytmy:

Implementacja

- Struktura programu
- Wybrane fragmenty kodu

Możliwości rozbudowy

Instrukcja obsługi

- Informacja o argumentach przekazywanych do programu
- Kompilacja i uruchomienie z poziomu Visual Studio
- Uruchomienie pliku wykonywalnego

Opis problemu — cel projektu

Celem projektu jest stworzenie zestawu funkcji przetwarzających obrazy wykorzystując architekturę CUDA do wykonywania równoległego algorytmów.

Użytkownik wskazuje ścieżkę do pliku obrazu na którym algorytm ma zostać zastosowany oraz odpowiednie opcje przetworzenia obrazu. Wynik działania programu zapisywany jest do pliku.

Analiza problemu i teoretyczna propozycja jego rozwiązania, opis użytych algorytmów

Opis typów algorytmów

Przekształcenia punktowe

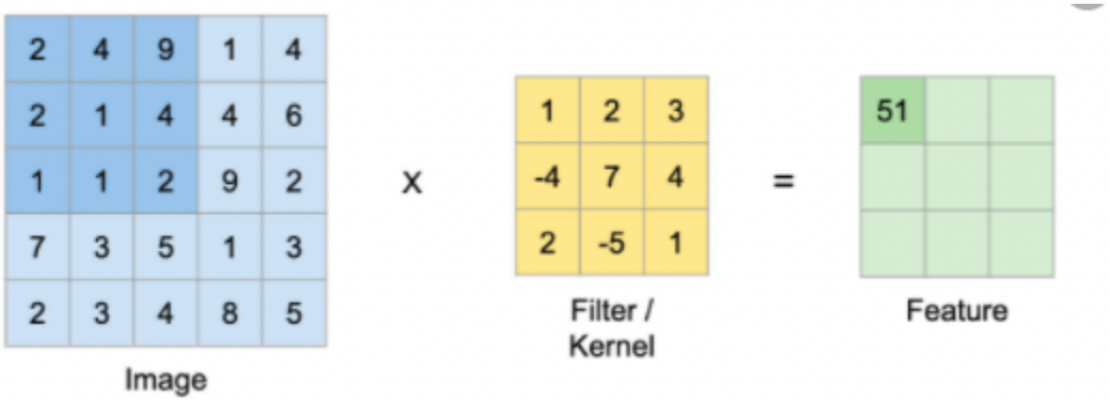
Cechą charakterystyczną punktowych przekształceń obrazu jest to, że poszczególne elementy obrazu
(punkty) modyfikowane są niezależnie od stanu elementów sąsiednich. Innymi słowy, jeden punkt
obrazu wynikowego otrzymywany jest w wyniku wykonania określonych operacji na pojedynczym
punkcie obrazu wejściowego

Przekształcenia kontekstowe

Zasada wyznaczania wartość piksela na obrazie wyjściowym w przekształceniach kontekstowych polega na wykorzystaniu wartości pikseli w pewnym sąsiedztwie (w otoczeniu K) tego piksela.

Przydatnym pojęciem w operacjach kontekstowych jest **konwolucja**, czyli inaczej **splot.**

W skrócie chodzi o to, że mnożymy wartości pikseli znajdujących się w pewnym sąsiedztwie punktu przez wartości wag, a całość sumujemy:



Wagi przez które mnożymy wartości pikseli nazywamy jądrem (kernelem).

Przekształcenia morfologiczne

Element strukturalny obrazu - wycinek obrazu, inaczej pewien podzbiór elementów, z wyróżnionym punktem - najczęściej jest to punkt centralny.

Operacje morfologiczne nie modyfikują całego obrazu, lecz tą jego część, której otoczenie odpowiada wzorcowi elementu strukturalnego.

Na czym polega przekształcenie morfologiczne?
- element strukturalny (np. koło jednostkowe) jest przemieszczane po całym obrazie,
- dla każdego punktu obrazu dokonujemy porównania fragmentu obrazu z wybranym elementem
strukturalnym,
- w każdym punkcie obrazu sprawdzamy, czy konfiguracja pikseli obrazu jest identyczna ze wzorcem elementu strukturalnego,
- gdy wzorzec zostanie dopasowany to na obrazie wykonywane są operacje.

Dostępne algorytmy:

1. Filtry Dolnoprzepustowe (różne kernele) w tym Filtr Gaussa - przekształcenia kontekstowe
2. Filtry Górnoprzepustowe (różne kernele) w tym wykrywanie krawędzi filtrem Sobela - przekształcenia kontekstowe
3. Binaryzacja - przekształcenia punktowe
4. Dylatacja - przekształcenia morfologiczne
5. Erozja - przekształcenia morfologiczne
6. Inwersja kolorów - przekształcenia punktowe

Implementacja

Struktura programu

Projekt został stworzony jako rozwiązanie Visual Studio.

Plik *main.cpp* zawiera funkcję główną programu, wywołującą poszczególne operacje na obrazie.

Wykorzystując bibliotekę openCV dla c++ obraz wskazany przez użytkownika zapisywany jest w specjalnej strukturze typu *Mat*, zawierającej wartości pikseli, wymiary oraz liczbę kanałów obrazu.

W zależności od argumentów przekazanych przez użytkownika uruchamiane jest odpowiednie przetwarzanie wraz z wybranymi opcjami. Więcej o tym w instrukcji obsługi.

Folder *images* powinien zawierać obraz który użytkownik chce przekształcać. Tam też pojawi się wynik działania programu jako plik jpg *"result.jpg"*.

Plik *functions.h* zawiera definicje funkcji, których zadaniem jest przygotowanie danych oraz alokacja pamięci, a następnie wywołanie funkcji przetwarzającej obraz na GPU.

Plik *filtersKernels.h* zawiera definicje kerneli oraz dodatkowych zmiennych wykorzystywanych w filtrowaniu obrazu. W zależności od podanej przez użytkownika opcji, odpowiedni zestaw zmiennych jest przekazywany do funkcji przetwarzającej.

Poszczególne pliki z rozszerzeniem *cu* zawierają konkretne funkcje wykonujące przetwarzanie obrazu.

binaryzation.cu - binaryzacja obrazu do wartości 1 lub 255 w zależności od wybranego progu warunkowego. Obraz przed przetworzeniem konwertowany jest do skali szarości z jednym kanałem kolorów.

imageInversion.cu - zawiera operację odwracania kolorów w obrazie. Dana wartość jest odejmowana od maksymalnej możliwej wartości.

filtering.cu - zawiera operację filtrowania obrazu na podstawie wybranego kernela w formie tablicy liczb całkowitych. Obraz wyjściowy może być zarówno w skali szarości jak i w formacie RGB.

erosion.cu - zawiera operację zarówno erozji jak i dylatacji. Wynikiem zazwyczaj jest pomniejszony lub powiększony obiekt widoczny na obrazie. Dla lepszego efektu warto przeprowadzić operację na odpowiednim, zbinaryzowanym obrazie . Przykładowy plik znajduje się w folderze images. Obraz przed przetworzeniem konwertowany jest do skali szarości.

W folderze doc znajduje się dokumentacja kodu stworzona w narzędziu doxygen.

Wybrane fragmenty kodu

```cuda
__global__ void imageInversionCuda(unsigned char* input, unsigned char* output, int imageSize) {

    int index = threadIdx.x + blockIdx.x * blockDim.x;

    if (index < imageSize) {
        output[index] = 255 - input[index];
    }
}
```

Funkcja odpowiedzialna za odwrócenie kolorów w obrazie. Na podstawie indeksu unikalnego dla danego wątku, dana wartość jest odejmowana od maksymalnej wartości obrazu

```cuda
__global__ void binaryzationCuda(unsigned char* input, unsigned char* output, int imageSize, int threshold) {

    int index = threadIdx.x + blockIdx.x * blockDim.x;

    if (index < imageSize) {

        if (input[index] < threshold) {
            output[index] = 1;
        }
        else {
            output[index] = 255;
        }
    }
}
```

Funkcja odpowiedzialna za binaryzację obrazu do wartości 1 lub 255. Na podstawie indeksu unikalnego dla danego wątku, dana wartość jest przewartościowywana w zależności od przekazanego progu.

```cuda
__global__ void erosionCuda(unsigned char* input, unsigned char* output, int width, int height, int structSize) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    int yIndex = blockIdx.y * blockDim.y + threadIdx.y;
    if (yIndex < height && xIndex < width) {

        int top = max(yIndex - structSize, 0);
        int bottom = min(height, yIndex + structSize);
        int left = max(xIndex - structSize, 0);
        int right = min(width, xIndex + structSize);
        int value = 255;
        for (int i = top; i <= bottom; i++) {
            for (int j = left; j <= right; j++) {
                value = min((float)value, (float)input[i * width + j]);
            }
        }
        output[yIndex * width + xIndex] = value;
    }
}
```

Funkcja odpowiedzialna za wykonanie erozji. Parametr structSize symuluje nam element strukturalny. Im jest większa wartość, tym większy obszar rozważanego otoczenia dla danego piksela.

Każdy wątek rozważa jedynie część obrazu. Z wybranego otoczenia dla danego piksela, wybierana jest maksymalna wartość. W praktyce przy odpowiednim obrazie, jeśli któraś z wartości otoczenia wynosi 0 (kolor czarny), to wartość rozważanego piksela będzie wynosić 0.

Dla dylatacji, kod wygląda podobnie, zamiast minimalnej wartości wybierana jest maksymalna.

W filtrowaniu filtrami górno lub dolnoprzepustowymi najpierw wskazujemy odpowiedni kernel. Przykładowy zapis kernela wraz z dodatkowymi wartościami znajduje się na obrazku.

```
// blur
int radius1 = 1;
int weight1 = 9;

/*
@brief represents kernel for blur. Values should be divied by 9, so weight should equal 9
*/
int kernel1[] =
{
  1, 1, 1,
  1, 1 ,1,
  1, 1, 1
};
```

Zmienna radius określa odległość krawędzi kernela od punktu centralnego (dla 3x3 będzie to 1, dla 5x5 wynosi 2 itd.). Weight to wartość przez którą wyliczone wartości będą dzielone.

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Kernel dla rozmycia obrazu. Docelowo każda wartość wynosi 1/9

```
__global__ void filteringCudaGray(unsigned char* input,
    unsigned char* output,
    int width,
    int height,
    int radius, int weight, int rowLength, int* filterKernel) {

    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    int yIndex = blockIdx.y * blockDim.y + threadIdx.y;

    if ((xIndex < width) && (yIndex < height)) {

        int accChannel1 = 0;


        int outputId = yIndex * rowLength + xIndex;

        for (int i = -radius; i <= radius; i++) {
            for (int j = -radius; j <= radius; j++) {

                // assign value from filter kernel
                int temp = filterKernel[i + radius + (j + radius)];
```

Sama funkcja filtrująca polega na przejściu przez każdy element kernela

```
                //get pixel from input (3 for 3 channels)
                int currInputId = ((xIndex + i)) + (yIndex + j) * rowLength;

                //get values for each channel
                const unsigned char channel1 = input[currInputId];

                //sum piixel multipled by filter value
                accChannel1 += int(channel1) * temp;

            }
        }
        accChannel1 = accChannel1 / weight;

        // convert value to unsigned char expected by Mat type
        output[outputId] = (unsigned char)(accChannel1);
```

A następnie przemnożeniu wartości i zsumowanie ich przez co powstaje nowa wartość dla rozważanego piksela.

Powyższe przykłady prezentują kod dla obrazu w skali szarości. Dla skali RGB odpowiednie indeksy muszą być przemnożone przez liczbę kanałów.

Możliwości rozbudowy

- Implementacja dodatkowych operacji na podstawie już istniejących. Przykładowo operacji otwarcia i zamknięcia

- Implementacja dodatkowych nowych operacji na przykład filtrów pasmoprzepustowych, operacje logiczne na obrazach, złożenia obrazów
- Dostosowanie rozwiązania dla uniwersalnego uruchamiania spoza Visual Studio
- Przyspieszenie operacji wykorzystując pamięć współdzieloną

Instrukcja obsługi

Informacja o argumentach przekazywanych do programu

Pierwszy argument - nazwa pliku obrazu znajdującego się w folderze images

Drugi argument - nazwa funkcji przetwarzającej wybranej spośród nazw: *filtering, erosion, dilatation, inversion, binaryzation*

Trzeci argument - cyfra w zależności od wybranej funkcji ma różne znaczenie.
Dla filtering → rodzaj kernela wraz z dodatkowymi zmiennymi. Cyfra odpowiada cyfrze w nazwie kernela
Dla binaryztion → próg warunkowy dla wartości
Dla erosion i dilatation → wielkość elementu strukturalnego, im większa wartość, tym bardziej widoczny efekt

Czwarty argument - istotny tylko dla filtering - “true” oznacza przekształcenie obrazu do skali szarości przed przetworzeniem, “false” zostawia obraz oryginalny.

KAŻDE URUCHOMIENIE POWINNO ZAWIERAĆ 4 ARGUMENTY
Wymaga tego struktura programu. Argumenty nieistotne dla danej funkcji mogą przyjąć dowolną wartość.

Kompilacja i uruchomienie z poziomu Visual Studio

Plik projektu znajduje się w tym samym katalogu co kod źródłowy.

Na urządzeniu powinna być zainstalowana biblioteka openCV dla c++. Można to zrobić przez menager pakietów vcpkg i określenie dostępu do biblioteki dla Visual Studio. Link do instrukcji: https://vcpkg.io/en/getting-started.html

Program powinien być uruchomiony z odpowiednimi argumentami. Aby to zrobić należy otworzyć

Properties → Configuration Properties → Debugging → Command Arguments

Wynik w folderze images znajdującego się tam gdzie plik projektu

Uruchomienie pliku wykonywalnego

Przykładowe uruchomienie programu z poziomu terminala w systemie operacyjnym Windows:
.\projekt.exe lena.png filtering 1 true

Wynik w folderze images znajdującego się tam gdzie plik wykonywalny