|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| logowydzialu | Instytut Informatyki Politechniki Śląskiej  Zespół Mikroinformatyki i Teorii Automatów Cyfrowych | | | logoii | |
| **Rok akademicki:** | **Rodzaj studiów\*: SSI/NSI/NSM** | **Przedmiot (Języki Asemblerowe/SMiW):** | | **Grupa** | **Sekcja** |
| **2018/2019** | **SSI** | **Języki Asemblerowe** | | **5** | **9** |
| **Imię:** | **Bartłomiej** | **Prowadzący**:  OA/JP/KT/GD/BSz/GB | | **AO** | |
| **Nazwisko:** | **Krasoń** |
| ***Raport końcowy*** | | | | | |
| **Temat projektu:**  Algorytm solaryzacji obrazu | | | | | |
| **Data oddania:**  **dd/mm/rrrr** | | | **04/12/2018** | | |

# Temat projektu:

Tematem projektu jest stworzenie aplikacji konsolowej w języku C++, która przetwarza obrazy bitmap (24-bitowych) algorytmem solaryzacji obrazu.

# Założenia

## Założenia części głównej

* Wykorzystanie C/C++ jako języku wysokiego poziomu (aplikacja konsolowa)
* Zmierzenie i porównanie czasów wykonania dla dwóch sposobów
* Wykorzystanie wielowątkowości (od 0 do 64 wątków) – z ustawianiem dowolnej wartości poprzez wykrywanie liczby rdzeni procesora
* Program ma być napisany dla procesorów 64-bitowych
* Aplikacja umożliwia pomiar rzeczywistego czasu wykonania samego algorytmu

## Założenia funkcje bibliotek

* Obie biblioteki wykonują ten sam algorytm (rezultat ich działania powinien być taki sam)
* Biblioteka w języku C++ musi zostać zbudowana dla wersji ”*Release*”
* Biblioteka w asemblerze musi wykorzystywać instrukcje wektorowe SIMD

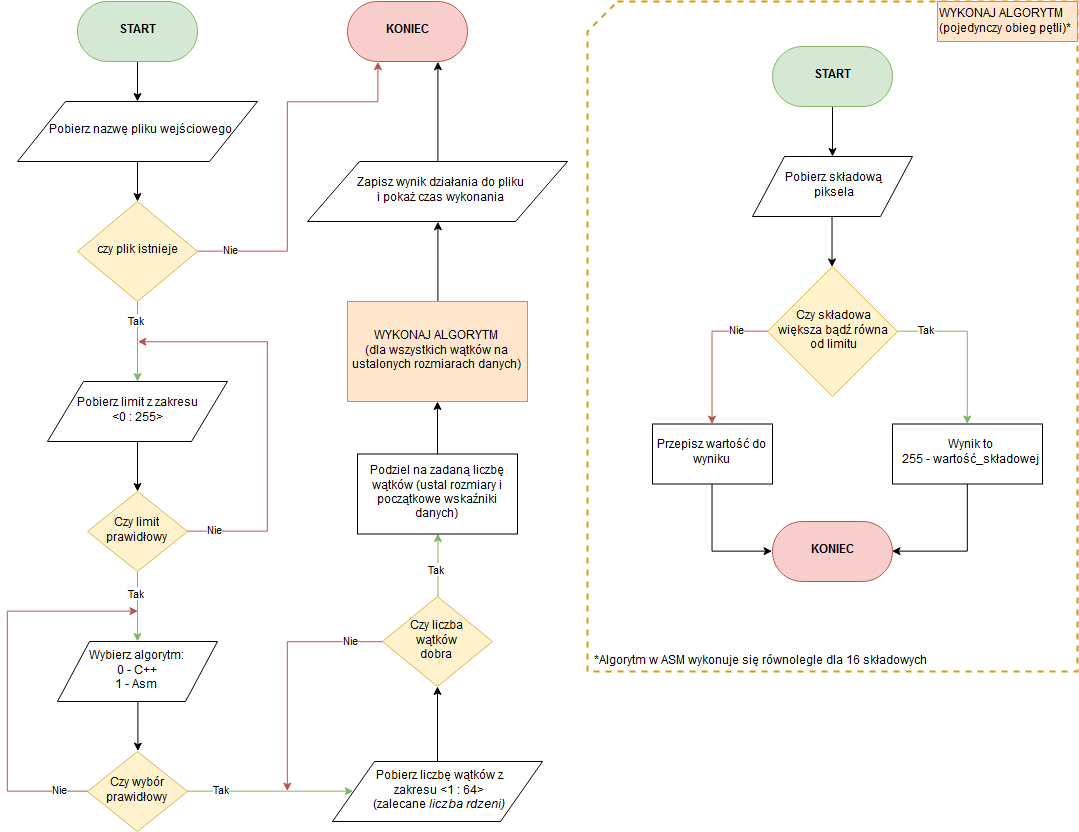
# Analiza zadania

Solaryzacja jest to zjawisko całkowitego lub częściowego odwrócenia obrazu negatywowego w pozytywowy. Stopień ”*odwrócenia*” obrazu zależy od zadanego progu jasności powyżej którego następuje odwrócenie wartości składowych RGB piksela. Wartość progu *ip* może oscylować w granicach od 0 do 255.

Algorytm można zapisać jako równanie:

Tak więc aby programistycznie rozwiązać ten problem, zdecydowałem się na wczytanie bitmapy do programu w formacie BGR. Następnie przekazuje wskaźnik na tablice jednowymiarową przechowującą wartości składowych pikseli do funkcji realizujących algorytm solaryzacji obrazu. Algorytm w programie został zaimplementowany na wzór powyżej zaprezentowanego równania.

# Schemat blokowy



# Opis programu w języku wysokiego poziomu

Aplikacja główna odpowiada przede wszystkim za interakcję z użytkownikiem. Pobiera niezbędne dane do wykonania algorytmu, takie jak zadany próg jasności, czy liczba wątków na ilu algorytm ma się wykonać. Więcej szczegółów na temat interakcji z użytkownikiem w rozdziale **ODNOŚNIK**. Po uruchomieniu - program przeprowadza ”*wywiad*” z użytkownikiem, który wprowadza potrzebne dane. Każde wprowadzenie danych jest sprawdzane (zakres progu/ilości wątków, istnienie pliku wejściowego itp.). Po prawidłowym podaniu wszystkich niezbędnych danych program wykonuje algorytm w wybranym języku, przy wykorzystaniu zadanej liczby wątków, na bitmapie, do której ścieżkę podał użytkownik. Czas wykonywania algorytmu (wraz z podziałem na wątki) jest mierzony przez aplikację. Po wykonaniu algorytmu następuje zapis wynikowych danych do pliku o nazwie zadanej schematem:

*(nazwa\_pliku\_wejściowego)*\_result\_P*(wartość\_progu)*\_*(*ASM*/*CPP*)*.bmp

Po wykonaniu zapisu, na konsoli wypisywany jest czas wykonania algorytmu i na tym etapie praca programu kończy się.

Program główny składa się z:

* Klasy **Bitmap** – która odpowiada za obsługę bitmap w programie, posiada następujące metody:
  + **Bitmap(const string &filePath)** – konstruktor, odpowiada za wczytanie danych z pliku o ścieżce *fileName*. Dane są szczytywane w formacie BGR. Pobierane również informacje z *headera* bitmapy o rozmiarze obrazu.
  + **bool isOpen() –** sprawdza czy konstruktor prawidłowo pobrał dane z pliku zadanego ścieżką *fileName*
  + **BYTE\* getPixelDataPointerBGR()** - zwraca wskaźnik na zaalokowane dane wejściowe bitmapy, wczytane przez konstruktor
  + **BYTE\* getResultDataPointer()** - zwraca wskaźnik na zaalokowaną pamięć, której zawartość będzie zapisywana metodą *saveBMPTo()*
  + **long getImageSize()** – zwraca rozmiar wczytanego obrazu
  + **void saveBMPTo(const string &filePatch)** – tworzy nową bitmapę, do której zapisuje zawartość zaalokowanej pamięci spod wskaźnika zwracanego metodą *getResultDataPointer()*. Dane zapisywane są do pliku wskazanego ścieżką *filePath*.
  + **~Bitmap()** - destruktor, zwalnia zaalokowaną pamięć
* Klasy **Functions** – zawiera funkcje, odpowiadające za odpowiedni podział wykonania algorytmu na wątki, posiada metody:
  + **static void solarizeForNBytesUsingXCoresASM(BYTE \*\_begin, BYTE \*\_result, BYTE limit, const long n, const int numberOfCores)** – odpowiada za wielowątkowe wykonanie algorytmu asemblerowego. W swoim ciele wywołuje funkcję asemblerową zaimportowaną z biblioteki *Asm.dll*.
  + **static void solarizeForNBytesUsingXCoresCPP(BYTE \*\_begin, BYTE \*\_result, BYTE limit, const long n, const int numberOfCores)** – odpowiada za wielowątkowe wykonanie algorytmu zaimplementowanego w C++. W swoim ciele wywołuje funkcję asemblerową zaimportowaną z biblioteki *Cpp.dll*.
* Funkcji **int main()** – która odpowiada za konsolową interakcję w użytkownikiem
* Funkcji **int getCoresCount()** – która zwraca liczbę wykrytych rdzeni, procesora używanego przez komputer

Każda klasa programu jest podzielona na pliki nagłówkowe i źródłowe zgodne z nazwą klasy. Funkcje *main()* i *getCoresCount()* zawierają się w pliku *SolaryzacjaObrazu.cpp*

# Opis funkcji bibliotek

## Biblioteka CPP.dll

Biblioteka ta składa się z dwóch funkcji, z czego tylko jedna jest exportowana:

* Funkcja **void solarize(BYTE \*\_data, BYTE \*\_result, const BYTE limit)** – odpowiada za zmianę bądź nie pojedynczej składowej piksela. Działa zgodnie z algorytmem opisanym w rozdziale **.** Parametry:
  + **BYTE \*\_begin** – wskaźnik na dane wejściowe
  + **BYTE \*\_result** – wskaźnik na dane wynikowe
  + **const BYTE limit** – próg jasności zadany przez użytkownika
* Funkcja **void solarizeCPP(BYTE \*\_begin, BYTE \*\_result, const BYTE limit, const long n)** – EXPORTOWANA – odpowiada za wykonanie funkcji *solarize()* na tablicy danych zadanej długością *n*. Funkcja ta importowana jest do programu głównego identyczną deklaracją. Parametry:
  + **BYTE \*\_begin** – wskaźnik na dane wejściowe
  + **BYTE \*\_result** – wskaźnik na dane wynikowe
  + **const BYTE limit** – próg jasności zadany przez użytkownika
  + **const long n** – rozmiar danych przekazanych jako wskaźnik na tablicę jednowymiarową

## Biblioteka ASM.dll

Biblioteka ta zawiera funkcje asemblerową:

* Funkcja **solarizeASM**, która importowana do programu głównego jest jako:

**int \_stdcall solarizeASM(BYTE\* data\_ptr, long size, BYTE limit, BYTE\* result\_ptr)** – wykonuje algorytm opisany w rozdziale . Algorytm jest wykonywany równolegle na 16 bajtach danych. Funkcja najpierw sprawdza, czy zadany próg jasności jest równy 0. Jeśli tak, następuje optymalizacja – funkcja wykonuje skok do fragmentu w którym, wszystkie bity danych wejściowych są bezwarunkowo zamieniane na negacje i zapisywane są na wskaźnik danych wynikowych (dla progu 0, niepotrzebne jest porównywanie, gdyż zawsze składowa będzie większa bądź równa 0). Gdy próg jest różny od 0, algorytm wykonuje się normalnie. W algorytmie wykorzystywane są instrukcje wektorowe SIMD takie jak:

* + **pcmpgtb xmm1, xmm2** – która porównuje osobno każdy bajt z paczki 16 bajtów rejestru xmm1 z odpowiadającymi im bajtami rejestru xmm2, jeśli bajt w rejestrze xmm1 jest większy od odpowiadającego mu bajtu rejestru xmm2, do rejestru xmm1 na tym bajcie wpisywana jest wartość FF, w przeciwnym razie do tego bajtu wpisywana jest wartość 00. Z rozkazem tym związane są 2 problemy:

Problem 1: rozkaz ten porównuje bajty, jako bajty ze znakiem (z zakresu -127 do 128), przez co wymusza na nas przed podaniem mu danych, ich konwersji na bajty bez znaku (zakres 0 do 255). Konwersji takie możemy dokonać wykonując operację XOR miedzy bajtem do konwersji a stałą 80h, co też jest uwzględnione w powyższej funkcji

Problem 2: rozkaz ten wykonuje prównanie ”*większy*” a nie ”*większy bądź równy*”, przez co zmuszeni jesteśmy przed porównaniem wartości danych z progiem, zmniejszyć próg o 1. Możliwe to jest tylko wtedy, gdy dla progu = 0 zapewnimy specjalną obsługę, co też jest uwzględnione w powyższej funkcji.

* + **movups** – umożliwia operację przekazywania danych między pamięcią a rejestrami xmm (w obie strony)
  + **vpand**, **vpxor**, **vpor** – operacje logiczne wykonywane na całych rejestrach xmm

Parametry/rejestry funkcji:

* + **BYTE\* data\_ptr (rejestr RCX -> RSI)** – wskaźnik na dane wejściowe. Przy wywołaniu przekazywany na rejestr RCX. W trakcie wykonywania algorytmu, aktualny wskaźnik danych wejściowych przechowujemy w rejestrze RSI.
  + **long size (rejestr RDX)** – rozmiar tablicy jednowymiarowej danych do przetworzenia z uwzględnionym paddinigiem, służy do określenia momentu przerwania pętli
  + **BYTE limit (rejestr R8 -> XMM7)** – próg jasności zadany przez użytkownika. Przy wywołaniu przekazywany na rejestr R8. Następnie jest powielany na wszystkie 16 bajtów rejestru XMM7, z którym później porównywane są dane wejściowe.
  + **BYTE\* result\_ptr (rejestr R9 -> RDI)** – wskaźnik na dane wynikowe. Przy wywołaniu przekazywany na rejestr R9. W trakcie wykonywania algorytmu, aktualny wskaźnik danych wynikowych przechowujemy w rejestrze RDI.
  + **Rejestr XMM5** – stała 80h powielona na wszystkich 16 bajtach, używana do konwersji danych w celu prawidłowego porównania ich z wartością progu rozkazem *pcmpgtb*.
  + **Rejestr XMM6** – maska jedynek, na potrzeby wykonywania operacji logicznych w algorytmie.

# Opis danych wejściowych/testowych programu

Program wykonuje swoje działanie na obrazach zapisanych w formacie 24-bitowej bitmapy. Działanie programy było przetestowane właśnie na plikach tego formatu lecz o różnych wymiarach. Testowane rozmiary to m. in.: 153x153, 400x340, 800x480, 1920x1080, 7680x4320.

# Opis uruchamiania/testowania programu

Aplikacja jest uruchamiana jako plik wykonywalny *SolaryzacjaObrazu.exe*. Tak jak każdy tego typu plik może zostać uruchomiona z konsoli, bądź przez dwukrotne kliknięcie ikony. Program nie posiada argumentów uruchomieniowych. Po uruchomieniu programu, przeprowadzany jest ”wywiad” z użytkownikiem, który proszony jest o podanie:

* ścieżki do pliku pliku wejściowego
* próg jasności w zakresie <0 : 255>
* wybór implementacji algorytmu:
  + 0 – C++
  + 1 – Asm
* liczbę wątków w zakresie <1 : 64>

Następnie algorytm wykonuje się, następuje utworzenie pliku wynikowego, program wyświetla czas wykonania algorytmu, po czy kończy swoją pracę.

Program został przetestowany różnymi rozmiarami obrazów wejściowych (szczegóły w poprzedni rozdziale). Wykonanie algorytmu zostało przetestowane na liczbie wątków od 1 do 64 równocześnie. Program został przetestowany dla wszystkich możliwych progów jasności. Wszystkie testy odbyły się zarówno dla implementacji w C++ jak i w asemblerze. Wyniki działania algorytmu napisanego w asemblerze były zawsze takie same jak w C++ oraz zgodne z oczekiwanym rezultatem.

# Wyniki pomiarów czasu

## Obraz 1920x1080