|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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

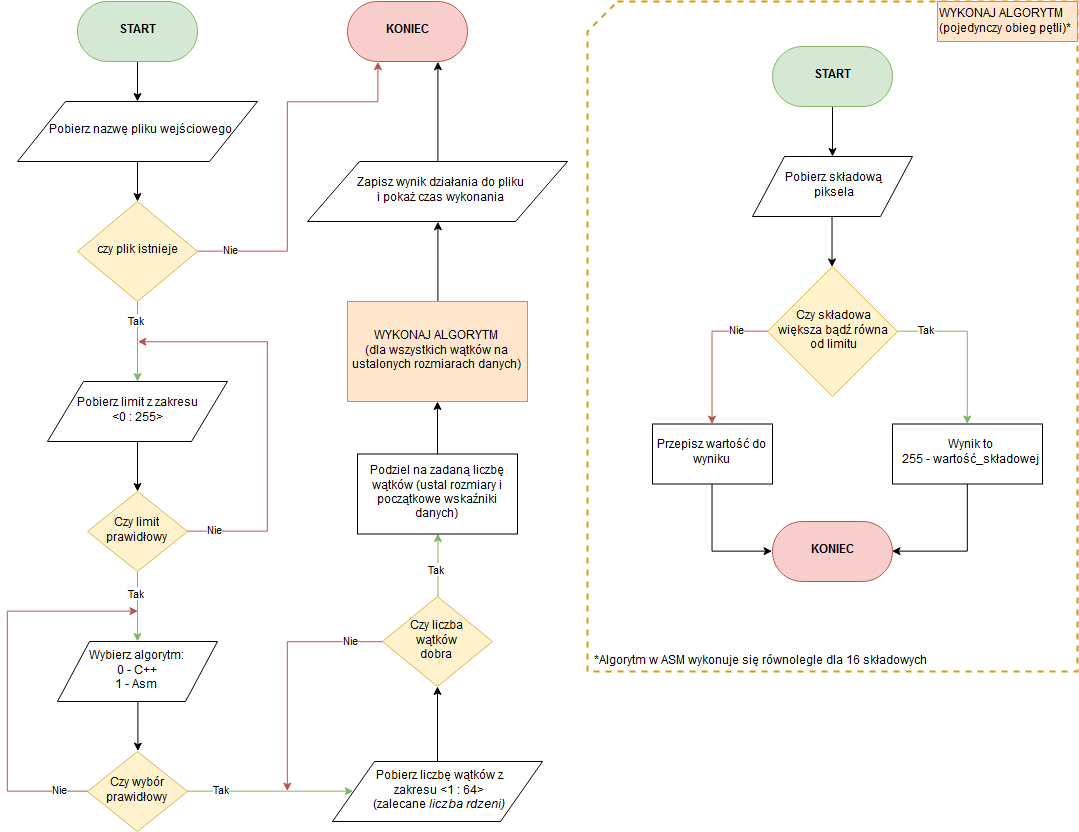
# Analiza zadania

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.

1. Opis programu w języku wysokiego poziomu:

Część główna zajmuje się wykonaniem wszystkiego poza głównym algorytmem edycji obrazu. Znajduje się w niej klasa BMP zawierają wszystkie wymagane metody oraz pola, potrzebne do prawidłowego działania programu oraz metoda main. W klasie BMP znajdują się:

* readBMP - metoda zajmująca się odczytem z pliku BMP, posiada jeden parametr, który jest nazwą pliku, jaki zostać ma odczytany. Do tablicy info wpisany zostaje cały nagłówek pliku, natomiast do tablicy data wpisane zostają wszystkie piksele (w modelu RGB, w kolejności składowych niebieski, zielony, czerwony). Metoda dodatkowo rozwiązuje problem paddingu oraz wpisuje do tablicy size odpowiednie parametry obrazu. Metoda nic nie zwraca.
* saveBMP – metoda zajmująca się zapisem do pliku, posiada jeden parametr, który jest nazwą pliku, do którego ma zostać zapisany obraz. Nagłówek nowego pliku jest pobrany ze zmiennej, do której trafił podczas odczytu pliku, a następnie wpisane do pliku są dane pikseli.
* grayScaleAsm oraz grayScaleCpp - dwie metody zajmujące się podziałem na wątki oraz wywołaniem metody z biblioteki, przekazując jej wszystkie parametry, umożlwiające poprawne działanie dla konkretnego wątku. Otrzymują dwa parametry: ilość wątków, jaka ma zostać stworzona oraz kolor, jaki ma zostać edytowany.

W main znajduje się wyświetlenie ilości rdzeni logicznych oraz wywołanie metod z klasy BMP.

6. Opis funkcji bibliotek:

Biblioteki w C++ oraz asemblerze wykonują główny algorytm programu. W obu bibliotekach znajduje się jedna metoda o nazwie ChangeImage. Przyjmuje ona trzy parametry: tablicę size, dane wejściowe odczytane z pliku oraz kolor, jaki ma zostać edytowany. Metoda zawiera trzy pętle: pętlę poruszającą się po kolumnach, zagnieżdżoną pętlę poruszającą się po wierszach oraz zagnieżdżoną pętlę, która dla danego piksela sprawdza, czy piksel jest danego koloru, jaki został przekazany w parametrze wejściowym (czerwony, zielony lub niebieski). W bibliotece asemblerowej używane rejestry to:

* r15 – przechowujący tablicę size
* r14 – dane wejściowe
* r12 – kolor, jaki ma zostać edytowany
* r11 oraz xmm6 – licznik pętli kolumn
* r10 oraz xmm8 – licznik pętli wiersza
* r19 – rejestr używany do zapisywania edytowanych składowych piksela w określone miejsce w tablicy
* r8 – licznik pętli dla danego piksela
* rcx oraz xmm15 – rejestr, w którym przechowana jest szerokość obrazu
* rdx – rejestr, w którym przechowywana jest wysokość obrazu
* xmm10 – rejestr używany przejściowo podczas obliczania, gdzie ma być wpisana edytowana składowa piksela
* xmm5 oraz xmm3 – rejestry przejściowo używane podczas wykonywania skali szarości na pikselu
* xmm2, xmm1 oraz xmm0 – rejestry używane do rozpakowania danych do rejestru xmm2

Metoda sprawdza, czy wartości składowych piksela są odpowiednie, aby móc uznać go za piksel o kolorze takim, jaki ma być poddany edycji. Jeśli tak, to wszystkie składowe piksela są sumowe, a następnie suma ta jest dzielone przez 3, a ostateczny wynik wpisany w miejsce składowych piksela. Dzięki temu otrzymana zostaje skala szarości w miejsce zadanego koloru. Jeśli piksel nie zostaje rozpoznany jako piksel o kolorze do edycji, następuje skok o 3 miejsca w tablicy, do następnego. Po przejściu do końca tablicy, następuje zakończenie działania metody i powrót do programu głównego.

1. Dane wejściowe i testowe:

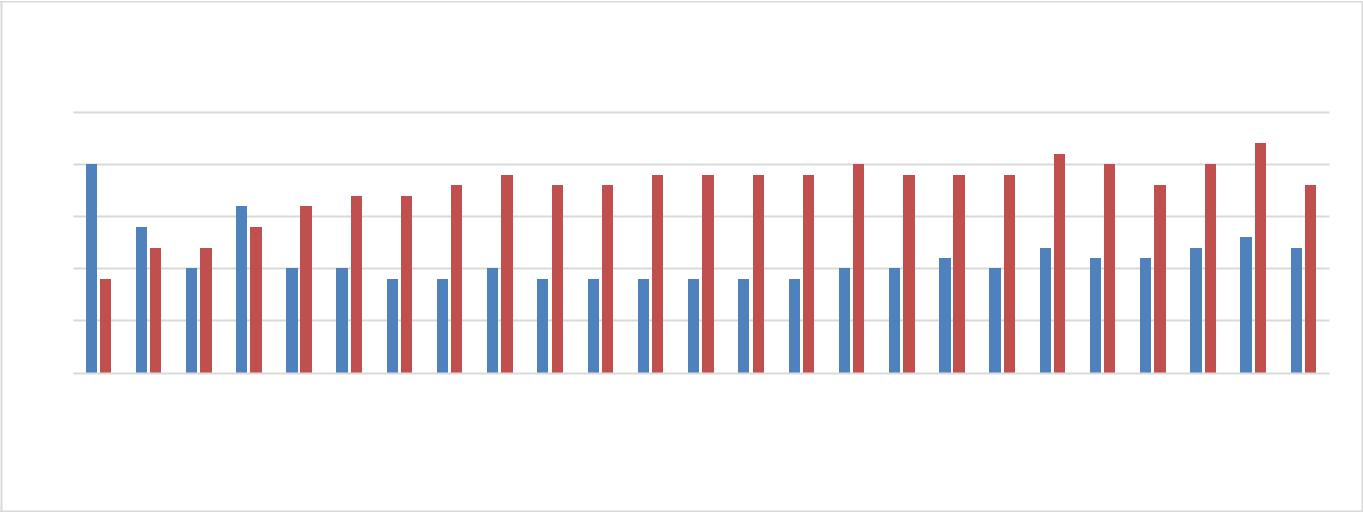
Dane wejściowe w postaci obrazu muszą być w formacie 24-bitowej bitmapy, również dane wyjściowe zostają stworzone w takim samym formacie. Do testowania używana jest również dowolny obraz, o ile jego format to 24-bitowa bitmapa.

8. Opis uruchamiania programu i testowania:

Program uruchamiany jest w konsoli. Następnie algorytm zostaje uruchomiony na podanej ilości wątków – każdy wątek obsługuje określoną ilość wierszy obrazu. Następnie wyświetlony jest w milisekundach czas wykonania algorytmu przy użyciu bibliotek w C++ oraz w asemblerze. Następnie po kliknięciu dowolnego przycisku program zostaje zamknięty.

Aby wykonać testowanie, należy w metodzie main zakomentować fragment kodu nazwany „Uruchomienie” oraz odkomentować fragment kodu nazwany „Testowanie”. Wtedy po uruchomieniu programu, algorytm jest wykonywany przy użyciu od 1 do 64 wątków dla metod z obu bibliotek. Następnie aby zamknąć program, wystarczy wcisnąć dowolny przycisk.

1. Wyniki pomiarów czasu wykonania programu:
   * Wykresy czasowe dla obrazu o małej rozdzielczości:



Kolor czerwony

25

20

15

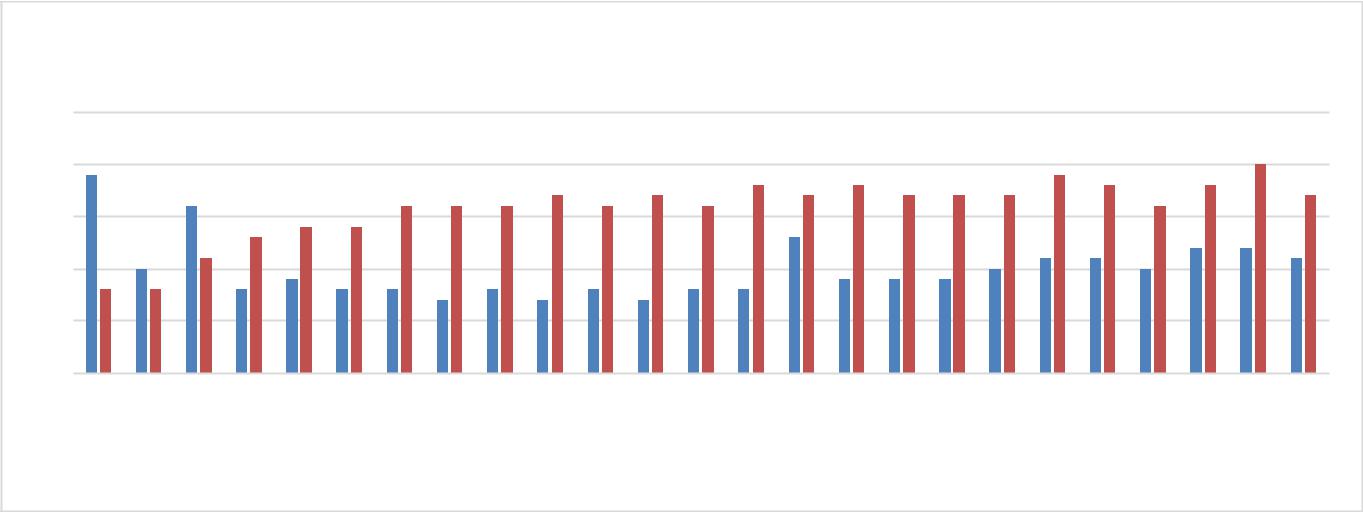
10

5

0

1 2 3 4 5 6 7 8 9 10 11 12 16 20 24 28 32 36 40 44 48 52 56 60 64

 redasm  redcpp



Kolor zielony

25

20

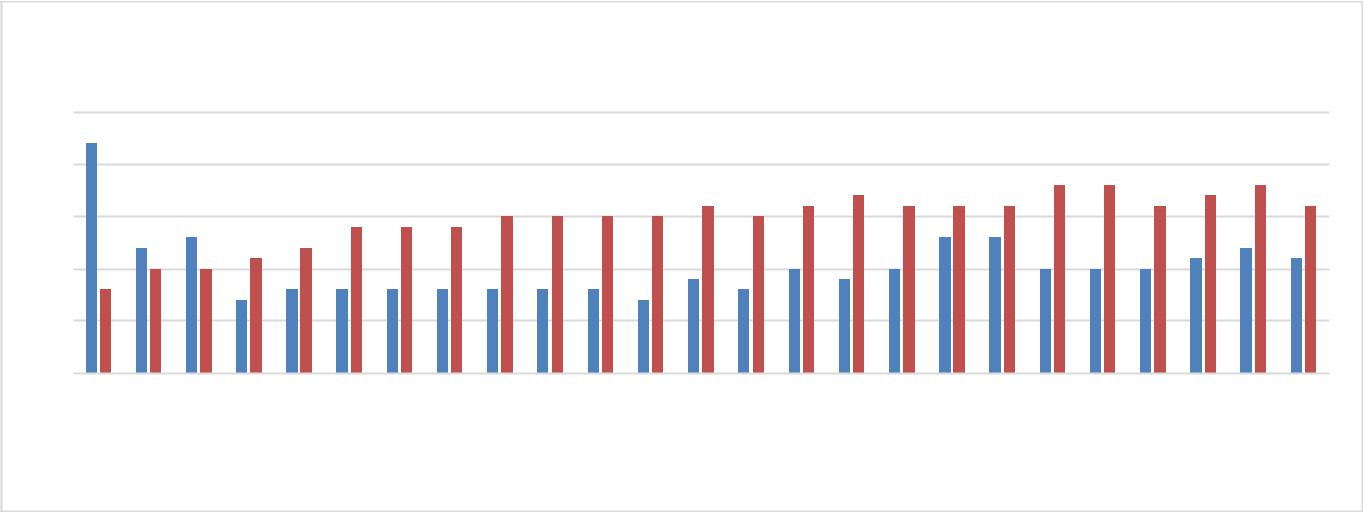
15

10

5

0

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | 11 | 12 | 16 | | 20 | 24 | 28 | 32 | 36 | 40 | 44 | 48 | 52 | 56 | 60 | 64 |  |
|  |  |  |  |  |  |  |  |  |  |  |  | greenasm | |  |  | greencpp | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Kolor niebieski

25

20

15

10

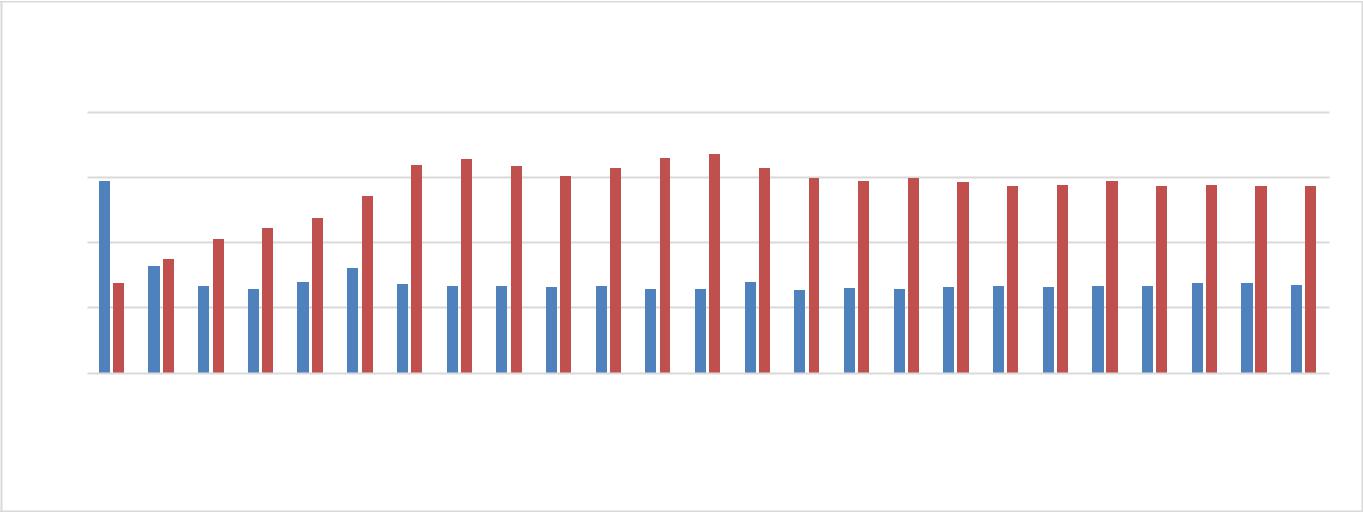
5

0

1 2 3 4 5 6 7 8 9 10 11 12 16 20 24 28 32 36 40 44 48 52 56 60 64

 blueasm  bluecpp

* Wykresy czasowe dla obrazu o dużej rozdzielczości:



Kolor czerwony

400

300

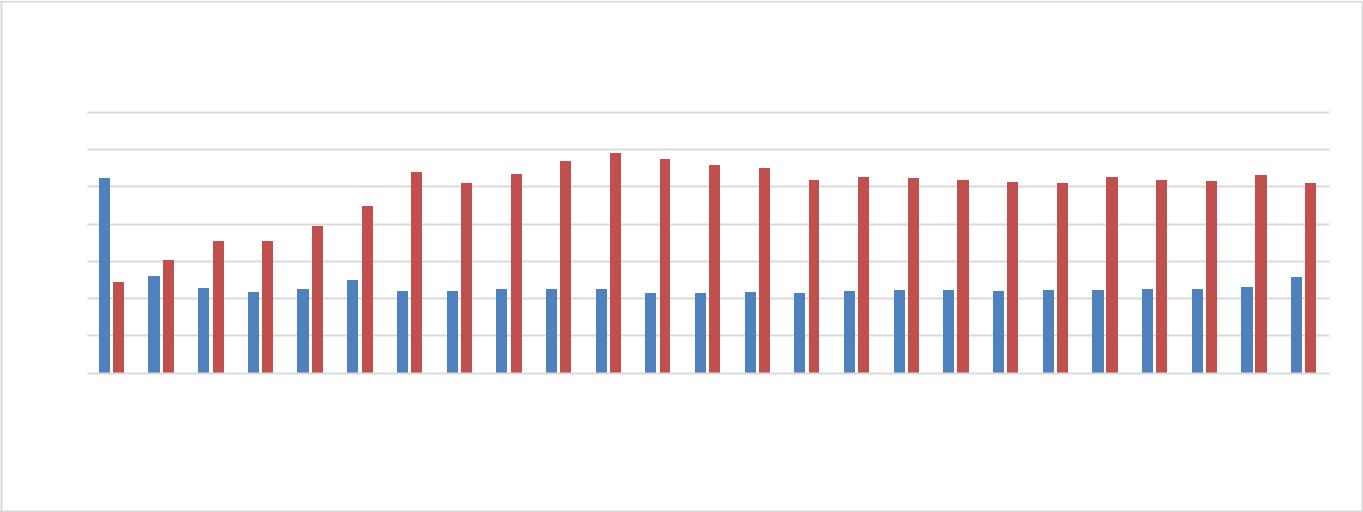
200

100

0

1 2 3 4 5 6 7 8 9 10 11 12 16 20 24 28 32 36 40 44 48 52 56 60 64

 RedAsm  RedCpp



Kolor zielony

350

300

250

200

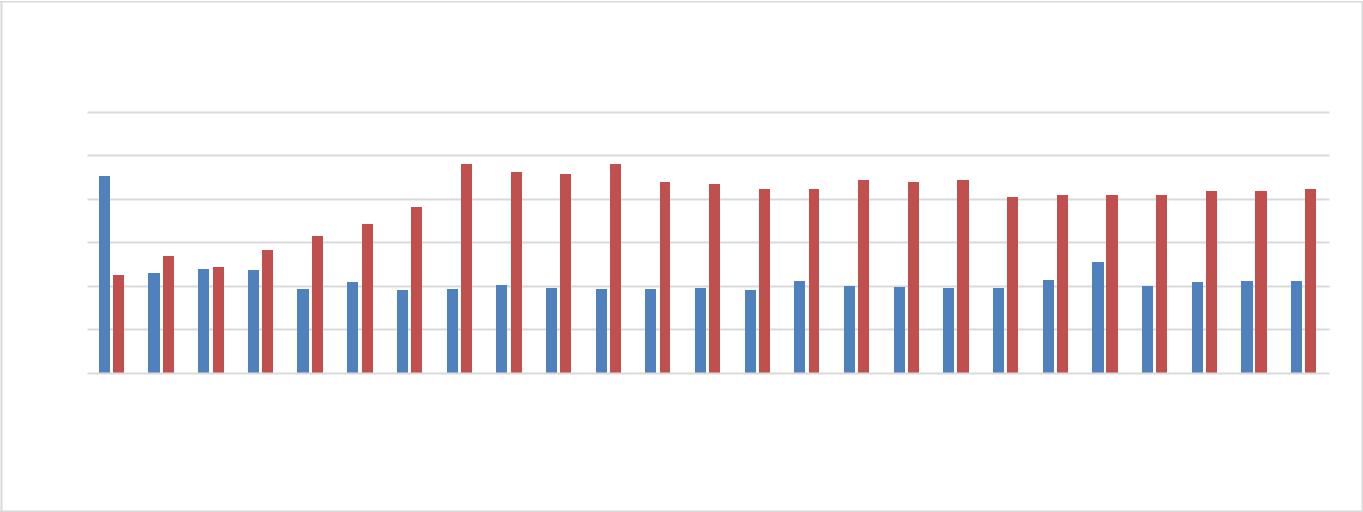
150

100

50

0

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | 11 | 12 | 16 | | | 20 | 24 | 28 | 32 | 36 | 40 | 44 | 48 | 52 | 56 | 60 | 64 |  |
|  |  |  |  |  |  |  |  |  |  | GreenAsm | | |  |  |  | GreenCpp | |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |



Kolor niebieski

300

250

200

150

100

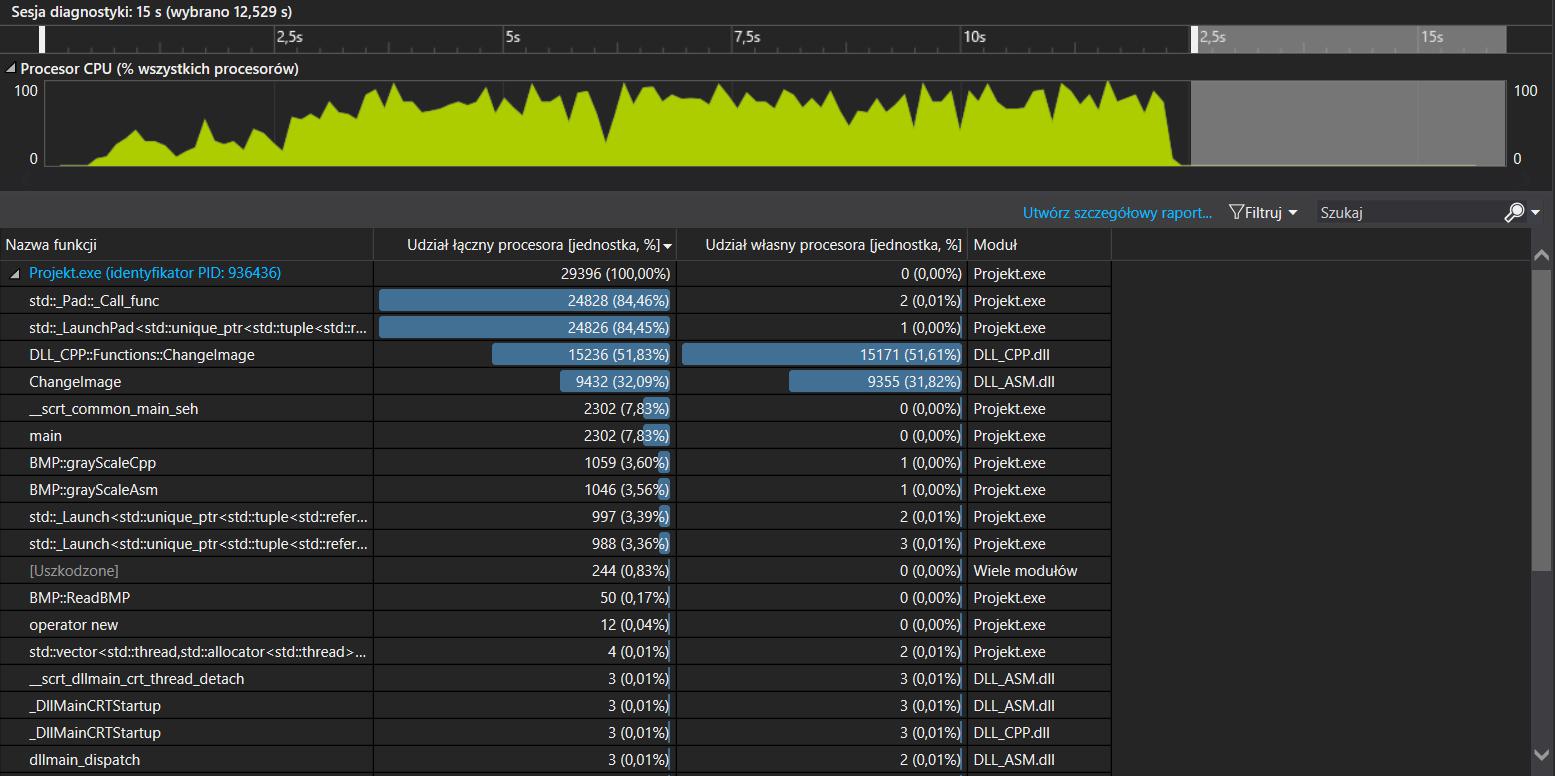
50

0

1 2 3 4 5 6 7 8 9 10 11 12 16 20 24 28 32 36 40 44 48 52 56 60 64

 BlueAsm  BlueCpp

1. Analiza działania z wykorzystanie profilera:



1. Instrukcja obsługi programu:

Po włączeniu programu w trybie „Uruchomienie”, wyświetlana zostaje ilość rdzeni logicznych. Następnie należy podać ilość wątków, przy użyciu jakiej algorytm ma się wykonać. Należy podać wartość z przedziału od 0 do 64, w innym przypadku program zostanie uruchomiony przy użyciu liczby wątków zgodnej z liczbą rdzeni logicznych procesora.

Po włączeniu programu w trybie „Testowanie”, brak jakiejkolwiek interakcji

W kodzie w metodzie main w dwóch miejscach można dokonywać korekt, aby zmodyfikować działanie programu.

Aby zmienić kolor, jaki ma zostać edytowany, należy zmienić drugi parametr wywoływanych w main metod grayScaleAsm oraz grayScaleCpp. „1” pozwala na usunięcie koloru czerwonego, „2” zielonego, a „3” czerwonego.

Aby zmienić obraz, jaki zostanie zmodyfikowany, należy w każdym wywołaniu metody ReadBMP zmienić pierwszy parametr na nazwę pliku (włącznie z rozszerzeniem, wszystko w cudzysłowie) jaki ma zostać edytowany. Aby zmienić nazwę pliku, do jakiego wyjściowy obraz ma zostać zapisany, należy zmodyfikować pierwszy parametr w każdym wywołaniu metody SaveBMP.

1. Wnioski:

Napisanie tego projektu pozwoliło wyciągnąć kilka wniosków:

* + Czas wykonywania metody w asemblerze nie zawsze jest krótszy niż jego odpowiednika w

C++.

* + Wzrost użytej ilości wątków nie jest równoważny z krótszym czasem wykonania algorytmu.
  + Wielkość danych testowych jest kluczowa – w zależności od wielkości danych czas wykonywania asemblera wraz ze wzrostem ilości wątków może szybciej lub wolniej rosnąć.

1. Literatura:

Tworzenie skali szarości: [www.algorytmy.org](http://www.algorytmy.org/)