
	Instytut Informatyki Politechniki Śląskiej Zespół Mikroinformatyki i Teorii Automatów Cyfrowych			
Rok akademicki:	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot (Języki Asemblerowe/SMiW):	Grupa	Sekcja
2019/2020	SSI	Języki Asemblerowe	3	1
Imię:	Mateusz	Prowadzący: OA/JP/KT/GD/BSz/GB	KH	
Nazwisko:	Czarnecki			
<h2><i>Raport końcowy</i></h2>				
Temat projektu: <h1 style="text-align: center;">Rozmycie Gaussowskie Obrazu</h1>				
Data oddania: dd/mm/rrrr				

Temat i założenia projektu

Tematem projektu jest rozmycie obrazu z wykorzystaniem filtra Gaussa. Projekt składa się z trzech części: programu głównego napisanego w języku C# z wykorzystaniem WPF oraz dwóch bibliotek dll napisanych w językach Cpp oraz ASM. Program jest aplikacją okienkową która umożliwia użytkownikowi wybranie obrazu, rozmycie go a następnie zapis obrazu wynikowego.

Użytkownik otrzymał możliwość wyboru interesującej go biblioteki (asm lub cpp). Może on także wybrać liczbę wątków w której będzie odbywało się przetwarzanie obrazu, chociaż program automatycznie ustawia tę wartość na liczbę fizycznych rdzeni procesora. Liczba rdzeni musi mieścić się w zakresie od 1 do 64.

Analiza problemu

W przetwarzaniu obrazu mamy do czynienia z pikselami, każdy z nich składa się z czterech części – kanału Alfa, oraz wartości składowych czerwonej, zielonej oraz niebieskiej. Wszystkie z czterech składowych mogą przyjmować wartość z zakresu od 0 do 255. Takie składowe można więc zapisać wykorzystując 8 bitów, czyli jeden bajt. Wartość kanału alfa nie jest interesująca z punktu widzenia omawianego problemu, nie będzie więc poruszana w dalszej części raportu. Odpowiednie wartości pozostałych składowych piksela (Red, Green i Blue) pozwalają nam stworzyć kolor który przyjmie piksel.

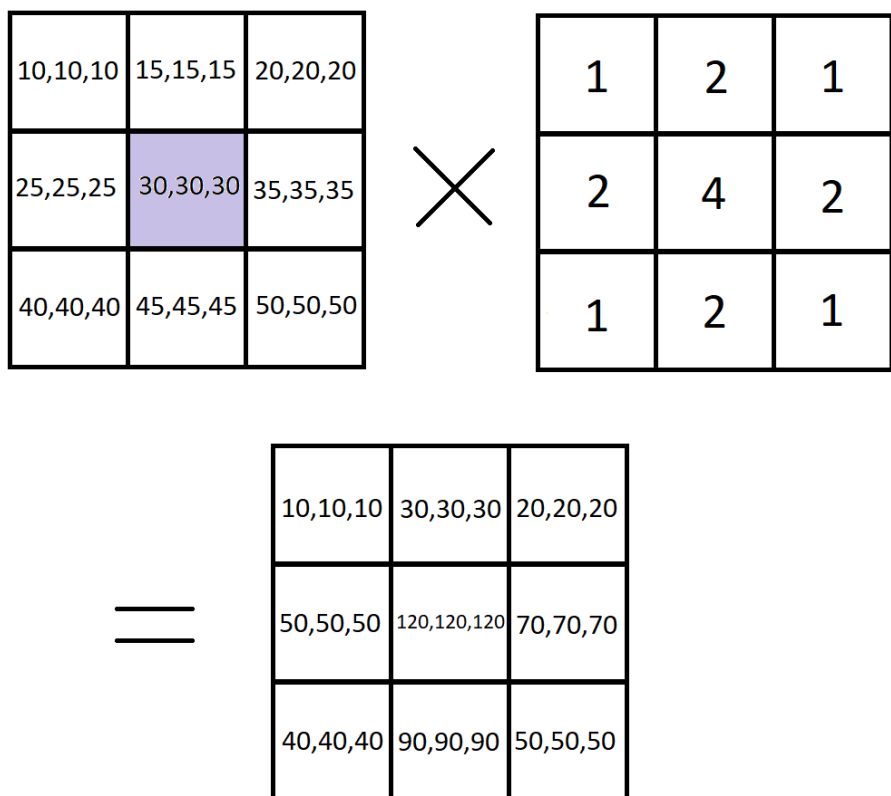
Rozmycie obrazu wymaga zmiany wartości RGB w pikselach obrazu w odpowiedni sposób. Podczas omawiania filtrów bardzo ważne jest pojęcie tak zwanej maski, czyli tablicy zawierającej odpowiednie wartości które będą nadawały wagi odpowiednim częściom przetwarzanego obrazu. Maską wykorzystywaną w filtrze Gaussa wygląda następująco:

1	2	1
2	4	2
1	2	1

Kolejnym ważnym pojęciem jest norma filtra, jest to wartość która służy do przywrócenia oryginalnej jasności obrazu po przemnożeniu go przez maskę. Wartości pikseli są najpierw mnożone przez wartości maski, więc następnie muszą zostać podzielone przez wartość normy aby obraz zachował oryginalną jasność, czyli nie został rozjaśniony ani przyciemniony. Wartość filtra maski jest sumą wszystkich jej elementów, więc w tym przypadku jest to $1 + 2 + 1 + 2 + 4 + 2 + 1 + 2 + 1 = 16$.

Należy również zwrócić uwagę na to, że gdy wykorzystujemy filtry, obliczamy nowe wartości RGB dla wybranego piksela nie tylko na podstawie jego wartości, ale bierzemy pod uwagę także jego sąsiedztwo. W przypadku wybranego filtra, w celu obliczenia wartości jednego piksela, istotna jest dla nas informacja o wartościach RGB tego piksela oraz wartościach ośmiu otaczających go pikseli.

Aby ułatwić zrozumienie algorytmu zastosowanego rozmycia posłużymy się przykładem, obliczymy nową wartość dla przykładowego piksela w przykładowej sytuacji. Obliczany piksel został zaznaczony na poniższym obrazku kolorem fioletowym.



Przykład przedstawia pierwszą część algorytmu - mnożenie odpowiedniej części obrazu (czyli obliczanego piksela wraz jego sąsiedztwem) przez maskę filtra, otrzymujemy nową tablicę z wartościami.

10,10,10	30,30,30	20,20,20
50,50,50	120,120,120	70,70,70
40,40,40	90,90,90	50,50,50

$$\begin{aligned}\Sigma R &= 10+30+20+50+120+70+40+90+50 = 480 \\ \Sigma G &= 10+30+20+50+120+70+40+90+50 = 480 \\ \Sigma B &= 10+30+20+50+120+70+40+90+50 = 480\end{aligned}$$

Następnie obliczamy sumę wartości RGB składających się na przetwarzany wycinek obrazu.

$$\begin{aligned}R &= 480/16 = 30 \\ G &= 480/16 = 30 \\ B &= 480/16 = 30\end{aligned}$$

	30,30,30	

Ostatnim krokiem jest podzielenie otrzymanej sumy przez normę, otrzymujemy zestaw nowych wartości RGB które zastępują stare wartości dla obliczanego piksela.

Operację taką należy zastosować dla wszystkich pikseli z których składa się obraz aby uzyskać rozmycie. Można powiedzieć, że uśredniamy wartości każdego piksela jego otoczeniem. Bardzo łatwo można zauważyć że nie wszystkie piksele w obrazie posiadają pełne otoczenie. Piksele znajdujące się na brzegu obrazka nie posiadają otoczenia co stanowi problem w wykorzystaniu algorytmu. Zdecydowałem się nie poddawać pikseli granicznych działaniu algorytmu, jest to najłatwiejsze rozwiązanie tego problemu, pozwala zaoszczędzić odrobinę pamięci, przyspiesza czas działania programu a przekłamanie które niesie za sobą ignorowanie tych pikseli jest znikome i nie da się go zauważyć dla obrazków o sensownej rozdzielczości.

Zdefiniowanie fragmentu kodu realizowanego z wykorzystaniem bibliotek.

Biblioteki będą wykonywały wszystkie operacje przedstawione w powyższym algorytmie. Argumentami przekazywanymi do bibliotek będą:

- Wskaźnik na tablicę pamięci, w której znajduje się 27 kolejno ułożonych wartości RGB. Są to wartości które przyjmuje aktualnie przetwarzany piksel oraz jego sąsiedztwo (9 pikseli, w każdym z nich 3 wartości)
- Wskaźnik na tablicę w pamięci, której zawartością jest aktualna maska filtra (istnieje możliwość rozbudowy programu, do bibliotek można przekazać dowolną maskę)
- Rozmiar tablicy wejściowej (w wypadku realizowanego rozmycia Gaussa jest to zawsze wartość 27). Ten argument jest przekazywany tylko po to, aby istniała możliwość rozbudowy programu w taki sposób, aby używał większych masek np 5x5 lub 7x7.
- Norma filtra (w wypadku realizowanego rozmycia Gaussa jest to zawsze wartość 16, przekazywanie tej wartości umożliwia podmiannę filtra w przypadku chęci rozbudowy programu)

Aby umożliwić działanie algorytmu, dane muszą zostać odpowiednio przygotowane. Zajmuje się tym kod napisany w języku C#. Przygotowanie wartości odbywa się już na etapie ładowania wybranego zdjęcia. Najpierw zostaje stworzona tablica wskaźników o długości równej ilości wszystkich przetwarzanych pikseli w obrazie (czyli wszystkich oprócz brzegowych). Każdemu pikselowi odpowiada jeden element tablicy wskaźników. Wskaźniki w zadeklarowanej tablicy wskaźników wskazują na 27-elementowe tablice wartości RGB należących do przetwarzanego piksela oraz jego sąsiedztwa. W ten sposób otrzymujemy tablicę, która dla każdego piksela przechowuje informacje o nim samym i jego sąsiedztwie. To właśnie te 27-elementowe tablice są przekazywane do bibliotek dll.

Tak więc biblioteki po otrzymaniu powyższych parametrów zajmują się mnożeniem wartości przez filter, obliczają sumę obliczonych wartości a następnie dzielą ją przez przekazaną normę. Otrzymujemy w ten sposób wartości R, G oraz B obliczanego piksela które są zapisywane na pozycjach 0, 1 oraz 2 w tablicy wartości, która była argumentem funkcji.

Przedstawienie sposobu realizacji w bibliotekach:

Kod ASM

```
;ASM Library that contains procedure to filter properly formated image with given 3x3 filter
;autor Mateusz Czarnecki

.code

;===== ARGUMENTS =====
;RCX - pointer to array with filter [IntPtr 32bit]
;RDX - pointer to array with part of the image [IntPtr 32bit]
;R8 - length of array with part of the image [int 32bit]
;XMM3 - norm [float 32bit]
;=====

;===== Register Description =====
;RCX - pointer to array with filter
;RDX - pointer to array with part of the image
;R8 - length of array with part of the image

;R14 - index of color array
;R15 - index of filter array

;XMM0 - sum RGB values of current part of the image |x|R|G|B|
;XMM1 - current pixel |x|R|G|B|
;XMM2 - current filter |x|filter|filter|filter|
;XMM3 - norm |x|norm|norm|norm|
;=====

filterProc PROC

    PUSH R14
    PUSH R15

    XOR R14, R14          ; set index of color array to 0
    XOR R15, R15          ; set index of filter array to 0
    XORPS XMM0, XMM0       ; set all bits of XMM0 register to 0

    UNPCKLPS XMM3, XMM3    ; unpack norm in the register |x|norm|x|norm|
    UNPCKLPS XMM3, XMM3    ; unpack norm in the register |norm|norm|norm|norm|

filterLoop:

    MOVUPS XMM1, [RDX + R14] ; move current pixel to XMM1 register |x|R|G|B|

    MOVUPS XMM2, [RCX + R15] ; move filter to XMM1 register
    UNPCKLPS XMM2, XMM2      ; unpack current filter value in the register |x|filter|x|filter|
    UNPCKLPS XMM2, XMM2      ; unpack current filter value in the register
    ;|filter|filter|filter|filter|

    MULPS XMM1, XMM2         ; multiply current pixel by filter
    ADDPS XMM0, XMM1         ; add multiplied values to sum of all pixels
    ADD R14, 12              ; move index of color array to the next pixel
    ADD R15, 4               ; move index of filter array to next filter element
    SUB R8, 3                ; decrement filter loop counter
    CMP R8, 0                ; check end of the loop
    JNZ filterLoop           ; if it is not the end, loop again

    DIVPS XMM0, XMM3         ; divide sum of all pixels by norm

    MOVAPS [RDX], XMM0       ; store result in memory

    POP R15
    POP R14

    RET                      ; return from procedure

filterProc endp

end
```

Kod C++

```
#include "pch.h" // use stdafx.h in Visual Studio 2017 and earlier
#include <utility>
#include <limits.h>
#include "DLL_C.h"
#include <xmmintrin.h>

void filterImage(float* filterPointer, float* arrayPointer, int length, float filterNorm)
{
    __m128 XMM0; // sum of RGB |x|R|G|B|
    __m128 XMM1; // currently processed pixel |x|R|G|B|
    __m128 XMM2; // current filter |x|filter|filter|filter|
    __m128 XMM3; // norm |x|norm|norm|norm|

    XMM3 = _mm_load_ss(&filterNorm); // loading norm to the register |0|0|0|norm|
    XMM3 = _mm_unpacklo_ps(XMM3, XMM3); // unpacking norm |0|norm|0|norm|
    XMM3 = _mm_unpacklo_ps(XMM3, XMM3); // second norm unpacking |norm|norm|norm|norm|

    XMM0 = _mm_setzero_ps(); // set all bits of XMM0 register to 0
    int offset = 0; // setting offset
    while (length != 0)
    {
        XMM1 = _mm_loadu_ps(arrayPointer + (__int64)offset * 3); // moving currently
//processed pixel (R,G,B) to XMM1 |x|R|G|B|
        XMM2 = _mm_loadu_ps(filterPointer + offset); // loading filter to the register
        XMM2 = _mm_unpacklo_ps(XMM2, XMM2); // unpacking filter
        XMM2 = _mm_unpacklo_ps(XMM2, XMM2);

        XMM1 = _mm_mul_ps(XMM1, XMM2); // multiplying current RGB values by filter value
        XMM0 = _mm_add_ps(XMM0, XMM1); // adding current multiplied RGB values to sum of
//all colors

        offset++; // moving memory pointer
        length -= 3; // decrementing loop counter
    }

    XMM0 = _mm_div_ps(XMM0, XMM3); // dividing sum of colors by norm
    _mm_store_ps(arrayPointer, XMM0); // moving calculated colors back to memory
}
```

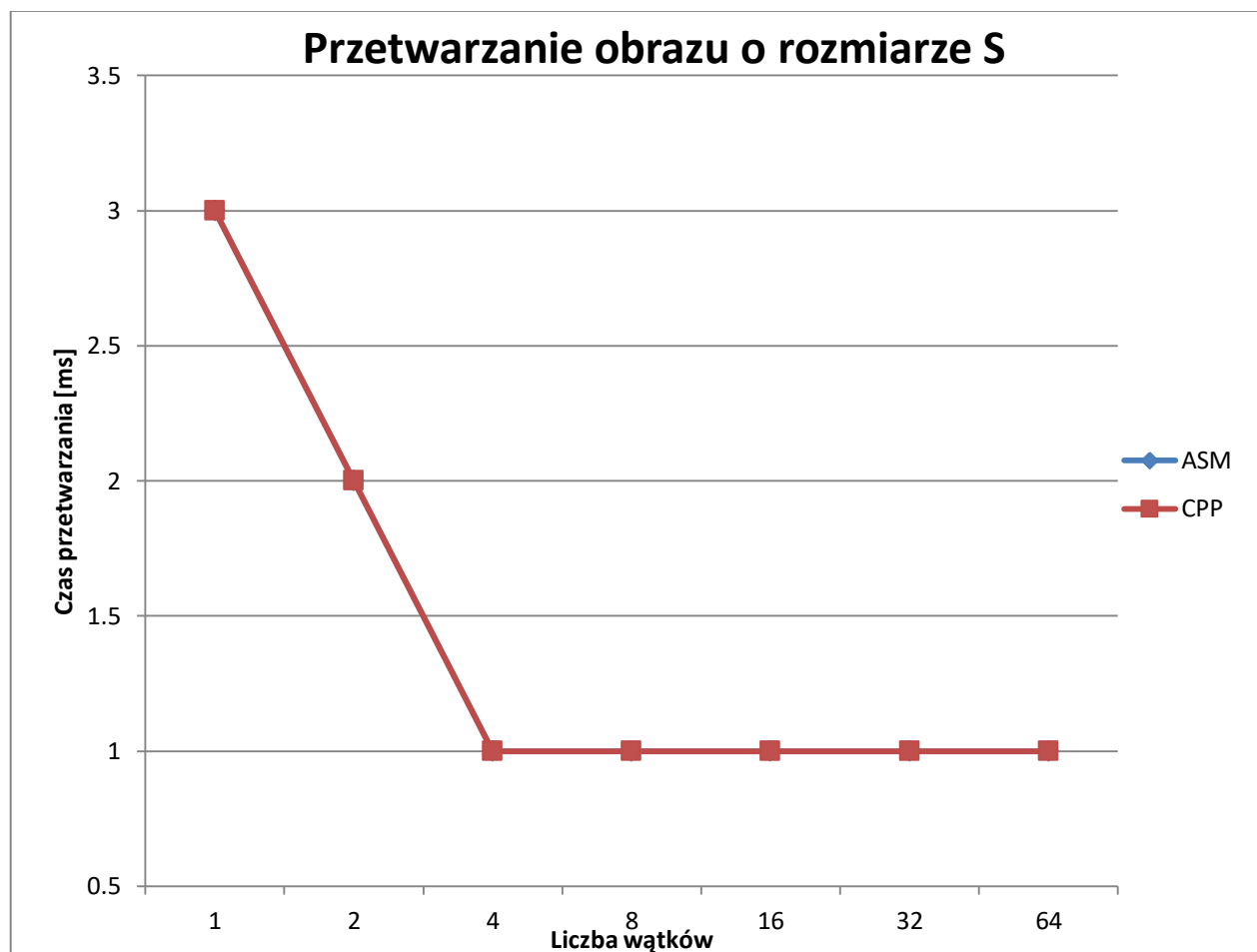
Jak widać obydwie biblioteki wykorzystują instrukcje wektorowe. Zamiast przetwarzać każdy kolor w pikselu osobno, wszystkie trzy wartości (R, G, B) są ładowane do rejestru i w tym samym czasie wykonywane jest na nich mnożenie, dodawanie i dzielenie.

Porównanie czasów działania bibliotek:

Porównania są przeprowadzone dla programu skompilowanego w trybie Release
Rozdzielczość wykorzystanych obrazów:

- S – 232x217
- M – 800x600
- L – 1920x1080

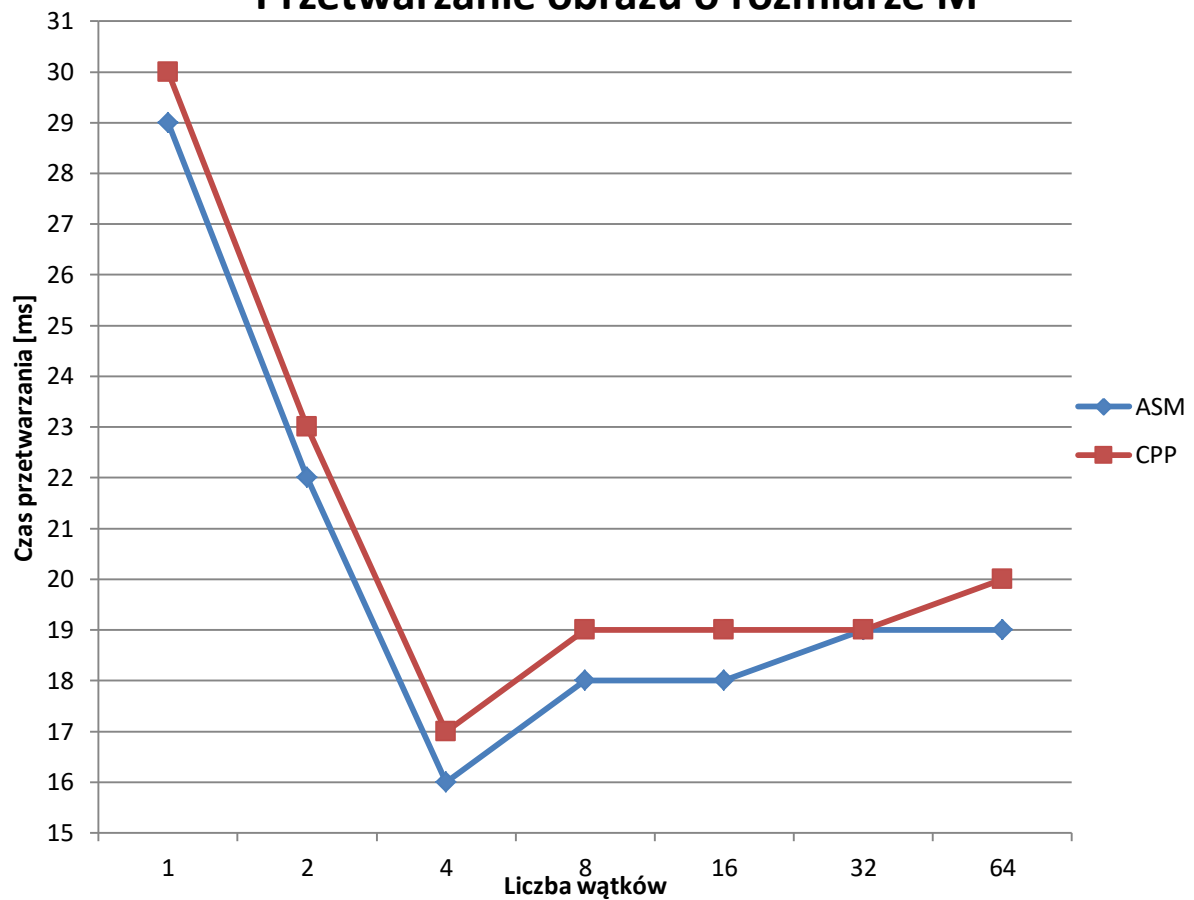
Rozmiar S		
Liczba wątków	ASM [ms]	CPP [ms]
1	3	3
2	2	2
4	1	1
8	1	1
16	1	1
32	1	1
64	1	1



Rozmiar M

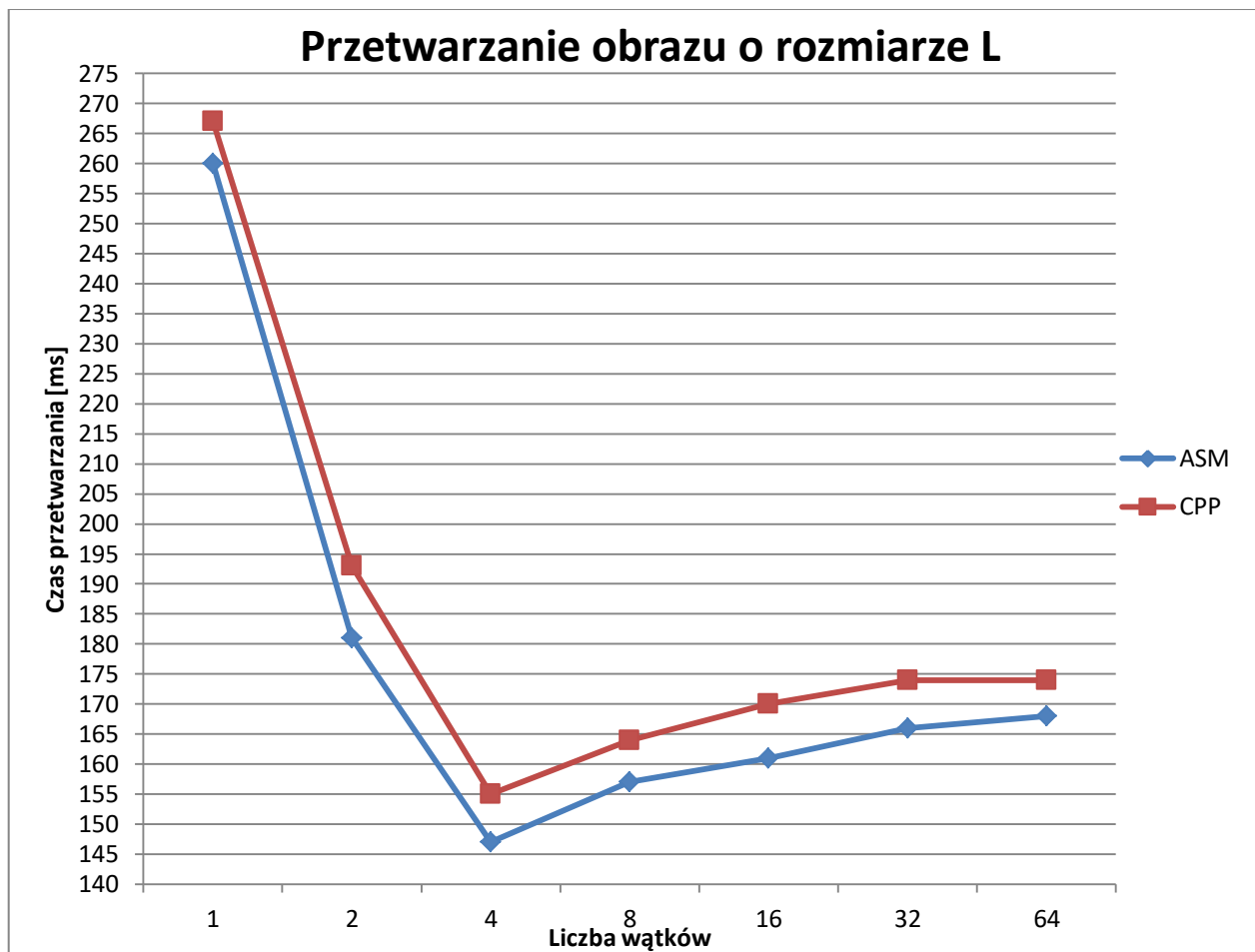
Liczba wątków	ASM [ms]	CPP [ms]
1	29	30
2	22	23
4	16	17
8	18	19
16	18	19
32	19	19
64	19	20

Przetwarzanie obrazu o rozmiarze M



Rozmiar L

Liczba wątków	ASM [ms]	CPP [ms]
1	260	267
2	181	193
4	147	155
8	157	164
16	161	170
32	166	174
64	168	174

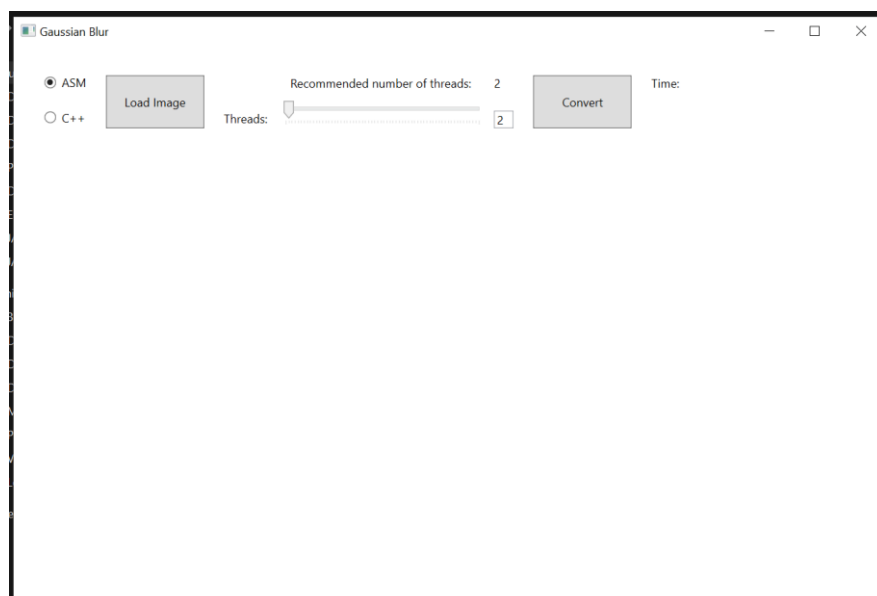


Analizując powyższe wykresy można dojść do wniosku, że program napisany za pomocą asemblera jest troszkę szybszy niż taki sam program napisany za pomocą języka c++. Różnica pomiędzy szybkościami rośnie wraz ze wzrostem rozmiaru obrazu. W przypadku najmniejszego obrazu nie dało się w ogóle zaobserwować różnicy natomiast dla obrazu L różnica ta wynosi do 10 ms.

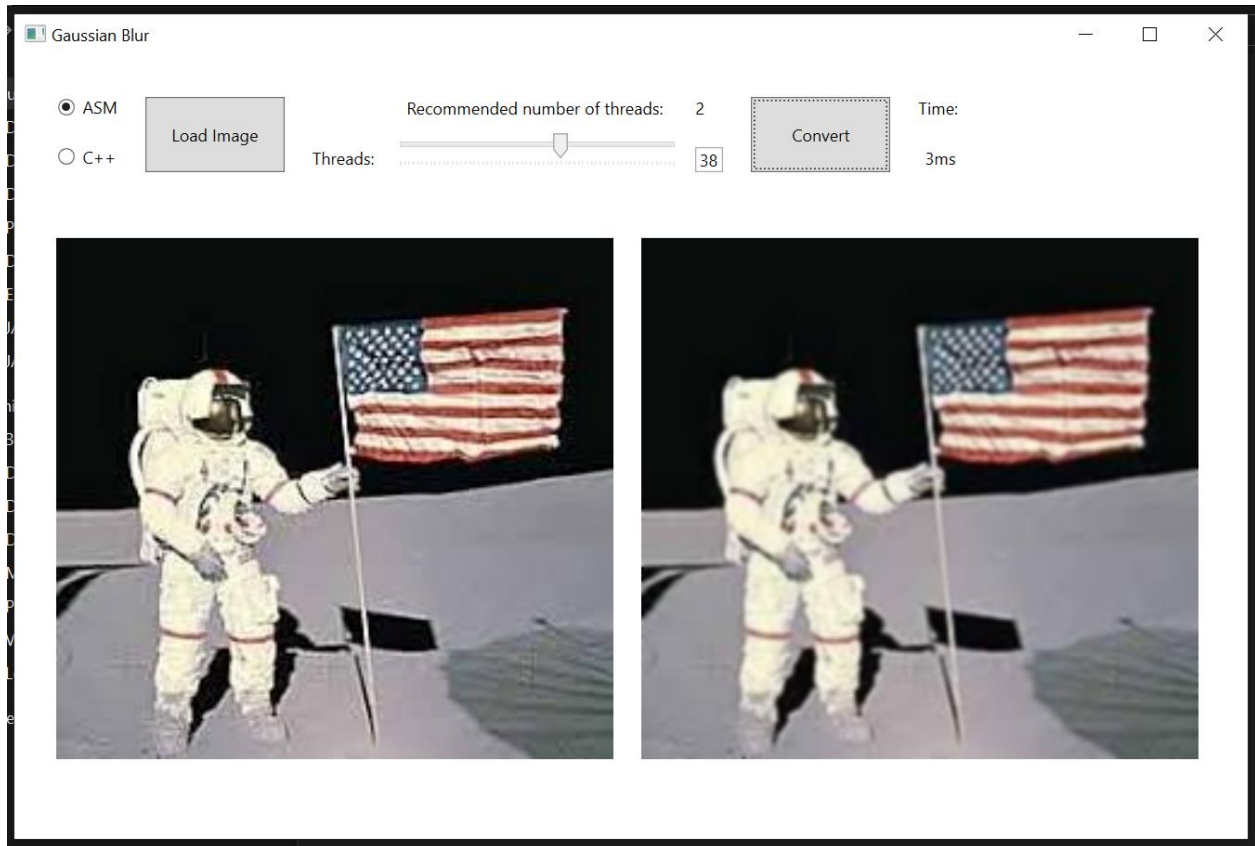
We wszystkich trzech przypadkach najlepszy czas udało się uzyskać dla 4 wątków a najgorszy dla 1 wątku. Ustawienie ilości wątków większej niż 4 spowalnia działanie programu, prawdopodobnie czas wykonania zmierza asymptotycznie do stałej wartości.

Przedstawienie działania programu:

Po uruchomieniu aplikacji, prezentuje się ona w następujący sposób:



Mamy możliwość wyboru biblioteki asm lub C++, przycisk Load Image pozwala nam załadować obraz który chcemy przetworzyć, suwak pozwala wybrać ilość wątków w których będzie zachodzić przetwarzanie obrazu. Program wykrywa ilość rdzeni procesora i ustawią ich liczbę jako wartość domyślną. Przycisk Convert pozwala nam dokonać konwersji obrazu a czas przetwarzania zostaje wyświetlony okno etykiety time.



Okno programu po przeprowadzonej filtracji na obrazku w rozmiarze S. Nowo powstały obraz jest automatycznie zapisywany w miejscu oryginalnego obrazu ze zmienioną nazwą. Nowa nazwa zawiera informację o liczbie wątków które zostały użyte do przetwarzania oraz nazwę wybranej biblioteki.

Przed:



Po:



Podsumowanie:

Wykonanie projektu znacząco zwiększyło moją wiedzę z zakresu tworzenia aplikacji z wykorzystaniem asemblera. Dowiedziałem się jak stworzyć bibliotekę dll w asm która może pomóc obsłużyć fragmenty programu które potrzebują optymalizacji i tym samym przyspieszyć działanie całej aplikacji. Asembler na początku sprawił mi bardzo duże trudności, napisanie pierwszych kilku działających linijek okazało się być najtrudniejszą częścią projektu. Mimo to po wielu próbach gdy udało się po raz pierwszy uruchomić bibliotekę praca nad projektem zaczęła przebiegać znacznie sprawniej. Po opanowaniu podstaw języka pisanie coraz trudniejszych fragmentów kodu nie stanowiło już większego problemu. Stworzenie biblioteki w języku C++ z wykorzystaniem instrukcji wektorowych także nie stanowiło żadnego problemu dzięki bibliotece `xmmintrin`. Biblioteka ta posiada zestaw funkcji odpowiadających poleceniom asemblerowym dzięki czemu kod wygląda bardzo podobnie do kodu asemblerowego.

Zastosowanie asemblera przyspieszyło przetwarzanie obrazu tylko w niewielkim stopniu, myślę że sprawni programista byłby w stanie dokonać lepszej optymalizacji. Myślę, że taki stan rzeczy jest też spowodowany tym, że biblioteka C++ także używa instrukcji wektorowych co musi mieć wpływ na jej wydajność. Pomimo, że nie udało mi się osiągnąć rewelacyjnych rezultatów bardzo cieszę się, że stworzyłem ten projekt, ponieważ bardzo dużo mnie nauczył. Gdyby w przyszłości dane mi było stworzyć podobny projekt, jestem pewien że rezultat byłby znacznie lepszy.

Zastosowany w tworzeniu projektu WPF wraz z językiem C# okazał się być rewelacyjnym narzędziem. Podpięcie zewnętrznych bibliotek stworzonych w innych językach było bardzo proste a samo tworzenie interfejsu użytkownika było bardzo przyjemne i intuicyjne. Znacznym ułatwieniem okazała się być pętla „`Parallel.For`” dzięki której problem wielowątkowości, którego rozwiązanie w innych językach mogłoby być bardzo czasochłonne i trudne, został rozwiązany za pomocą jednej linii kodu.