
		<b>Instytut Informatyki Politechniki Śląskiej</b>  <b>Zespół Mikroinformatyki</b> <b>i Teorii Automatów Cyfrowych</b>			
Rok akademicki	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot: ( Języki Asemblerowe/SMIW)	Grupa	Sekcja	
<b>2024/2025</b>	<b>SSI</b>	<b>Języki Asemblerowe</b>	<b>-</b>	<b>12</b>	
Prowadzący przedmiot:	dr inż. Magdalena Pawlyta			Termin: ( dzień tygodnia godzina)	
Imię:	Julia			poniedziałek	
Nazwisko:	Żółty			9:45-11:15	
Email:	jz306799@student.polsl.pl				
<b>Raport końcowy</b>					
Temat projektu:					
<p>Filtr sepii z efektem szumu.</p>					
Data oddania: dd/mm/rrrr			29.01.2025r		

# 1. Opis projektu

Celem projektu było stworzenie aplikacji umożliwiającej użytkownikowi nałożenie efektu Sepii na wybrany obraz z dodatkowym efektem szumu, który symuluje tzw. "postarzanie" obrazu. Algorytm przekształca obraz na skalę szarości, a następnie dodaje do każdego kanału piksela wartość przemnożoną przez parametr **P** wprowadzany przez użytkownika. Dodatkowo, generowany jest losowy szum w zakresie ustalonym przez parametr **X**, również definiowany przez użytkownika, co pozwala uzyskać efekt nieregularności w odcieniach.

Algorytm został zaimplementowany w dwóch niezależnych bibliotekach DLL:

- **ASM (Assembler)** – wykorzystuje zaawansowane techniki optymalizacji, w tym instrukcje SIMD (Single Instruction, Multiple Data), aby równolegle przetwarzać dane.
- **C++** – implementacja w języku wysokiego poziomu, korzystająca z tradycyjnych mechanizmów programowania proceduralnego.

Obie biblioteki mają takie samo zastosowanie, a zaimplementowane w nich algorytmy generują identyczne wyniki. Po zakończeniu przetwarzania obraz jest wyświetlany w aplikacji wraz z informacją o czasie jego przetwarzania, co pozwala na porównanie efektywności obu metod.

Dodanie funkcji generowania szumu umożliwia użytkownikowi kontrolowanie intensywności efektu "postarzania" obrazu, co pozwala uzyskać bardziej naturalny wygląd. Rezultaty działania programu mogą być przedstawione w postaci wizualnej (przetworzony obraz) oraz w formie analizy czasów przetwarzania przy różnych liczbach wątków.

## 2. Założenia projektu

- 2.1. Projekt dotyczy architektury procesorów Intel X86/64 .
- 2.2. Projekt wykonany jest w środowisku Visual Studio 2022.
- 2.3. Założenie projektu w środowisku VS polega na utworzeniu powiązanych projektów:
  - Aplikacja Interfejsu Użytkownika napisana w języku wysokiego poziomu (np. C++, C#). Aplikacja zapewnia okienkowy interfejs umożliwiający użytkownikowi parametryzację działania programu oraz kontrolę procesu przetwarzania.
  - Biblioteki DLL wywoływanej dynamicznie z poziomu aplikacji głównej i zawierającej implementację algorytmu w języku C++.

- Biblioteki DDL napisanej w assemblerze X64 udostępniającej funkcje biblioteczne, które również są wywoływane dynamicznie przez aplikację główną.
- 2.4. Aplikacja Interfejsu Użytkownika niezależnie od implementowanego algorytmu zawiera elementy wspólne:
- Opcję wyboru biblioteki, która zostanie wywołana do wykonania algorytmu (C++ lub ASM)
  - Pasek postępu wykonania funkcji bibliotecznej z wyświetlonym czasem przetwarzania.
- 2.5. Biblioteka w C++ ma być uruchomiona w trybie Release.
- 2.6. Biblioteka w ASM ma wykorzystywać instrukcje wektorowe (SSE/AVX).
- 2.7. Projekt ma wykorzystywać wielowątkowość, w celu analizy szybkości wykonania w zależności od wybranej ilości wątków procesu przetwarzania. Użytkownik ma decydować o ilości wątków w zakresie od 1 do 64.
- 2.8. Aplikacja zabezpieczona przez próbą użycia niepoprawnych danych wejściowych.
- 2.9. Wykonywany Program powinien umożliwiać zmianę wywoływania biblioteki DLL dynamicznie w trakcie działania Aplikacji bez konieczności jej przeładowania.
- 2.10. Ponowne otwarcie pliku obrazu umożliwia ponowne wykonanie algorytmu bez konieczności przeładowywania Aplikacji.

### 3. Opis kodu

#### 3.1. Działanie algorytmu

Algorytm przekształca obraz operując jednocześnie na czterech pikselach, przekształcając każdy piksel obrazu w odcienie szarości, uwzględniając wagi przypisane do kanałów RGB. Każdy kanał piksela jest przemnożony przez odpowiednią wagę – 0.299 dla kanału czerwonego, 0.587 dla zielonego oraz 0.114 dla niebieskiego – a ich suma daje wynikową wartość szarości. Wartość ta następnie powielana jest na wszystkie kanały RGB, co tworzy jednolity odcień szarości dla danego piksela.

Algorytm uwzględnia efekt szumu, który wprowadzany jest poprzez generowanie losowej wartości opartej na parametrze X. Szum ten jest dodawany

do wartości szarości, co pozwala uzyskać efekt postarzenia obrazu i nadaje mu bardziej nieregularny wygląd.

Następnie algorytm nakłada efekt sepii, dodając do wartości szarości odpowiednio przemnożone wartości na podstawie parametru P. Dla kanału czerwonego jest to  $2 * P$ , dla zielonego P, a dla niebieskiego szarość pozostaje niezmienną.

Wszystkie obliczone wartości są ograniczane do standardowego zakresu jasności pikseli od 0 do 255, aby zapewnić poprawność danych. Ostatecznie przetworzone wartości są zapisywane w buforze pikseli, tworząc finalny obraz z efektem sepii oraz postarzenia.

3.2. Funkcje realizujące algorytm ( w ASM i C++) przyjmują następujące osiem argumentów:

- **Wskaźnik na bufor pikseli (pixelBuffer)** – adres w pamięci, gdzie zapisane są dane obrazu, które mają zostać przetworzone.
- **Szerokość obrazu (width)** - liczba pikseli w jednym wierszu obrazu.
- **Liczba bajtów na piksel (bytesPerPixel)** - informuje o wielkości pojedynczego piksela w bajtach. Wartość ta wynosi 3 dla obrazów w formacie RGB.
- **Liczba bajtów na wiersz (stride)** - liczba bajtów przypadająca na jeden wiersz obrazu w pamięci.
- **Parametr P** - umożliwia kontrolę intensywności efektu sepii.
- **Wiersz początkowy (startRow)**
- **Wiersz końcowy (endRow)** - zakres wierszy obrazu, które mają zostać przetworzone przez algorytm, umożliwiając równoległe przetwarzanie wątków.
- **Parametr X** - odpowiada za dodanie szumu do obrazu, kontrolując jego intensywność w celu uzyskania efektu postarzenia obrazu.

3.3. Kod biblioteki ASM wraz z komentarzami.

```
;
;
;
; Autor: Julia Żółty
; Data obrony projektu: 27.01.2025
;
; Plik biblioteki DLL. Zawiera funkcję do konwersji obrazu kolorowego
na sepię oraz szum napisaną w assemblerze.
; Argumenty:
; rcx = pixelBuffer - wskaźnik na bufor przechowujący piksele obrazu
; rdx = width - szerokość obrazu
; r8 = bytesPerPixel - ilość bajtów w pikselu (3)
```

```
; r9 = stride - ilość bajtów na wiersz
; P = [rsp + 28h] - parametr P (sepii)
; startRow = [rsp + 30h] - początek wiersza
; endRow = [rsp + 38h] - końcowy wiersz
; X = [rsp +40h] - paramert X (szum)
;
```

---

.data

;Wagi podane w formacie IEEE-754 rozumianym przez asm zeby unikac dodatkowego pakowania/rozpakowywania floatow, dd rezerwuje 4 bajty w pamieci i zapisuje tam wartosc 32-bitowa.

weight\_blue dd 1038710997 ; 0.114f - Waga B

weight\_green dd 1058424226 ; 0.587f - Waga G

weight\_red dd 1050220167 ; 0.299f - Waga R

;Mnozники parametru P (BGR), do 16 bajtow wyrownane

p\_multipliers db 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 0, 0, 0

;Operuje jednocześnie na 4 pikselach poniewaz RGB jest wyrownane 24 lub 32 bajtow wiec -> nie mozna zalozyc ze bedzie podzielne przez 16 ale bedzie przez 12

;Wszystkie maski sa wyrownane do 16 bajtow - rejestry xmm to rejestry 16-bajtowe

;MASKI DO EKSPORTU KANALOW B G R (bufor wziety z bitmapy zatem jest w odwroconej kolejnosci, zamiast RGB)

;Eksport kanalu B (bez utraty czegokolwiek 80h)

blue\_channel db 0, 80h, 80h, 80h, 3, 80h, 80h, 80h, 6, 80h, 80h, 80h, 9, 80h, 80h, 80h

;Maska po nalozeniu: B1 0 0 0 B2 0 0 0 B3 0 0 0 B4 0 0 0

;Kopiuwane bity: bit0 0 0 0 bit3 0 0 0 bit6 0 0 0 bit9 0 0 0

;Eksport kanalu G

green\_channel db 1, 80h, 80h, 80h, 4, 80h, 80h, 80h, 7, 80h, 80h, 80h, 10, 80h, 80h, 80h

;Maska po nalozeniu: G1 0 0 0 G2 0 0 0 G3 0 0 0 G4 0 0 0

;Kopiuwane bity: bit1 0 0 0 bit4 0 0 0 bit7 0 0 0 bit10 0 0 0

;Eksport kanalu R

red\_channel db 2, 80h, 80h, 80h, 5, 80h, 80h, 80h, 8, 80h, 80h, 80h, 11, 80h, 80h, 80h

;Maska po nalozeniu: R1 0 0 0 R2 0 0 0 R3 0 0 0 R4 0 0 0

;Kopiuwane bity: bit2 0 0 0 bit5 0 0 0 bit8 0 0 0 bit11 0 0 0

```

;MASKI DO ZAPISU KANALOW B G R

;Niezaleznie od maski zapisujemy tylko bity 0, 4, 8 i 12, daje nam to
mozliwosc powielenia od razu szarosci, ostatnie 4 bajty sa nieuzywane,
wiec sa zerowane
;Zerowane w celu aby wyrownac do 12 bajtow (szarosc 4 pikseli aby potem
nie dodawalo zadnych dodatkowych warosci)

;Zapis kanalu B
blue_save db 0, 80h, 80h, 4, 80h, 80h, 8, 80h, 80h, 12, 80h, 80h, 80h,
80h, 80h, 80h
;Maska po nalozeniu: gray1 0 0 gray2 0 0 gray3 0 0 gray4 0 0 0 0 0
;Kopiuwane bity: bit0 0 0 bit4 0 0 bit8 0 0 bit12 0 0 0 0 0

green_save db 80h, 0, 80h, 80h, 4, 80h, 80h, 8, 80h, 80h, 12, 80h, 80h,
80h, 80h, 80h
;Maska po nalozeniu: 0 gray1 0 0 gray2 0 0 gray3 0 0 gray4 0 0 0 0 0
;Kopiuwane bity: 0 bit0 0 0 bit4 0 0 bit8 0 0 bit12 0 0 0 0 0

red_save db 80h, 80h, 0, 80h, 80h, 4, 80h, 80h, 8, 80h, 80h, 12, 80h,
80h, 80h, 80h
;Maska po nalozeniu: 0 0 gray1 0 0 gray2 0 0 gray3 0 0 gray4 0 0 0 0
;Kopiuwane bity: 0 0 bit0 0 0 bit4 0 0 bit8 0 0 bit12 0 0 0 0

rng_values dd 0, 0, 0, 0 ; Tymczasowa tablica przechowujaca dane RNG, 4
piksele

.code
ApplySepiaFilterAsm PROC

; Przygotowanie danych. Inicjalizacja zmiennych przekazanych jako
argumenty funkcji
    mov r10d, dword ptr [rsp + 30h] ; startRow -> r10d ;r10 64bitowy,
r10d 32bitowy
    mov r11d, dword ptr [rsp + 38h] ; endRow -> r11d

    mov r14d, dword ptr [rsp + 40h] ; X -> r14d

    mov rbx, rcx ; rcx -> rbx ; kopiuje rcx do rbx, przechowywanie
wskaźnika bufora pikseli w rbx

; ładowanie stałych do rejestrów SIMD
    movdqu xmm5, xmmword ptr [p_multipliers] ; kopiuje mnożniki do
xmm5, p_multipliers -> xmm5
    movd xmm6, dword ptr [rsp + 28h] ; P -> xmm6
    pshufd xmm6, xmm6, 0 ; powielenie P na całym rejestrze xmm6

```

```

    pmulld xmm6, xmm5 ; przemnozenie przez p_multipliers

; ładowanie wag do rejestrów SIMD
; Instrukcja z zestawu AVX vbroadcastss wrzuca do rejestru i powiela
vbroadcastss xmm7, weight_blue ; weight_blue -> xmm7 (+
rozprowadzenie)
vbroadcastss xmm8, weight_green ; weight_green -> xmm8 (+
rozprowadzenie)
vbroadcastss xmm9, weight_red ; weight_red -> xmm9 (+
rozprowadzenie)

; ładowanie masek i masek do zapisu do rejestrów SIMD
movdqu xmm10, xmmword ptr [blue_channel] ; blue_channel -> xmm10
movdqu xmm11, xmmword ptr [green_channel] ; green_channel -> xmm11
movdqu xmm12, xmmword ptr [red_channel] ; red_channel -> xmm12

;Zabezpieczenie przez nakładaniem dodatkowych niepotrzebnych rzeczy
movdqu xmm13, xmmword ptr [blue_save] ; blue_save -> xmm13
movdqu xmm14, xmmword ptr [green_save] ; green_save -> xmm14
movdqu xmm15, xmmword ptr [red_save] ; red_save -> xmm15

RowLoop: ;Petla do wierszy
    cmp r10d, r11d ; Jesli obecny wiersz odpowiada koncowemu, porownuje
endRow i startRow
    jge EndProc ; skocz do zakonczenia procesu, jump if greater or
equal

    mov rcx, rbx ; rbx -> rcx przywracam wskaznik buforu pikseli rcx z
rbx
    mov eax, r10d ; obecny wiersz -> eax
    imul eax, r9d ; eax * bytesPerPixel (3)
    add rcx, rax ; ecx -> eax (przejscie do wiersza ktory ma byc
nastepny)

    xor r13, r13 ; zerowanie licznika pikseli

PixelLoop: ;Petla do pikseli
    cmp r13d, edx ; jesli licznik pikseli odpowiada szerokosci obrazu
    jge NextRow ; skocz do przejścia do następnego wiersza

    jmp NoiseGenerator ; Skok bezwarunkowy do generatora szumu

NG_Return:
;ładujemy 4 piksele (8+4 bajty) w 2 krokach, xmm max 16 bajtow
    movlps xmm0, qword ptr [rcx] ; ładowanie pierwszych 64 bitow (8
bajtow) spod wskaznika rcx do gornej czesci xmm0 (2 2/3 piksela)
    movd xmm1, dword ptr [rcx+8] ; ładowanie kolejnych 32 bitow (4
bajtow) spod wskaznika rcx do dolnej czesci xmm1 (1 1/3 piksela)

```

```

;movd , semiwektorowy
movlhps xmm0, xmm1 ; laczenie obu rejestrów - kopiowanie 32 bitów z
dolnej czesci xmm1 na najblizsze wolne miejsca w gornej czesci xmm0,
zeby bylo bez nadpisania

movdqa xmm2, xmm0 ; kopiowanie zawartosci xmm0 do xmm2
pshufb xmm2, xmm10 ; wyciagniecie kanalu B
mulps xmm2, xmm7 ; mnozenie przez wage B

movdqa xmm3, xmm0 ; kopiowanie zawartosci xmm0 do xmm3
pshufb xmm3, xmm11 ; wyciagniecie kanalu G
mulps xmm3, xmm8 ; mnozenie przez wage G

movdqa xmm4, xmm0 ; kopiowanie zawartosci xmm0 do xmm4
pshufb xmm4, xmm12 ; wyciagniecie kanalu R
mulps xmm4, xmm9 ; mnozenie przez wage R

paddb xmm2, xmm3 ; B + G [dodawanie 8-bitowe czyli doublewordy],
asm skonwertowal do wartosci 32 bit
paddb xmm2, xmm4 ; (B + G) + R [dod. 8-bitowe]
;xmm2 = gray1 0 0 0 gray2 0 0 0 gray3 0 0 0 gray4 0 0 0,
otrzymalismy szarosc

paddb xmm2, xmm5 ; Dodanie szumu

;ograniczenie skali wartosci piksela 0-255
pxor xmm0, xmm0 ; Zerowanie xmm0
pcmpeqd xmm1, xmm1 ; Wszystkie bity xmm1 na 1
psrld xmm1, 24 ; Przesuniecie bitów o 24 w lewo - tworzy liczbe 255

pmaxsd xmm2, xmm0 ; Clamp 0 , ograniczenie wartosci minimalne
pminsd xmm2, xmm1 ; Clamp 255 , ograniczenie wartosci maksymalnej

;rozpropagowanie szarosci na kanaly
movdqa xmm0, xmm2 ; kopiowanie sumy kanalów do xmm0
pshufb xmm0, xmm13 ; wyodrebnienie kanalu B do zapisu, z maska do
zapisu

movdqa xmm1, xmm2 ; kopiowanie sumy kanalów do xmm1
pshufb xmm1, xmm14 ; wyodrebnienie kanalu G do zapisu

movdqa xmm3, xmm2 ; kopiowanie sumy kanalów R do xmm3
pshufb xmm3, xmm15 ; wyodrebnienie kanalu R do zapisu

paddb xmm3, xmm1 ; laczenie xmm1 z xmm3 (kanaly R i G) [dod. 8-
bitowe], dodajemy bajty a nie inty
paddb xmm3, xmm0 ; laczenie xmm0 z xmm3 (kanaly R, G i B) [dod. 8-
bitowe]

```



```

    ; xmm3 = gray1 gray1 gray1 gray2 gray2 gray2 gray3 gray3 gray3
gray4 gray4 gray4 0 0 0 0

    paddusb xmm3, xmm6 ; dodawanie wektorowe parametru P (0, P, 2P)
    ;"u" ogranicza ze w zakresie 0-255

    ;Zapisanie
    movq qword ptr [rcx], xmm3 ; zapisanie (xmm3)pierwszych 64
bitow=8bajtow=2i2/3piksela, qworda, pod wskaznik rcx czyli nadpisujemy
dane
    movhps xmm3, xmm3 ; przeniesienie najwcześniej dolnych zajetych
bitow xmm3 na poczatek rejestru
    movd dword ptr [rcx + 8], xmm3 ; zapisanie pozostałych 32
bitow=4bajtow=1i1/3piksela

    mov r12, 4 ; r12 ustawiam na 4
    imul r12, r8 ; r12 * bytesPerPixel(czyli 3)
    add rcx, r12 ; r12 dodaje do rcx (wskaznika)
    add r13d, 4 ; r13 += 4, inkrementacja licznika pikseli o 4
    jmp PixelLoop ; Skok na poczatek petli

NextRow: ;petlna skoku do następnego wiersza
    inc r10d ; r10++ (startRow)
    jmp RowLoop ; Skok do petli wierszy

EndProc:
    ret

NoiseGenerator:
    ; ZABEZPIECZENIE REJESTROW NA STOSIE, żeby się nie nadpisały
    push rcx
    push rax
    push rdx
    push rsi
    push r15
    push r13

    xor r13, r13 ; Zerowanie licznika r13

    ;lea - przypisanie do rcx wskaznika ktory wskazuje na adres rng_values
- tymczasowa tablica
    lea rcx, [rng_values] ; Wskaznik na rng_values

NG_Loop:
    mov r15d, r14d ; X -> r15d
    shl r15d, 1 ;shift left r15d*2 = 2X , mnozenie *2 przez
przesuniecie bitowe w lewo
    add r15d, 1 ; r15d + 1 = 2X + 1 ( inc r15d)

```

```

;Pobieranie czasu, liczba 64 bitowa rozprowadzona miedzy 2 rejestry -
dolne 32bity eax i edx
;Sygnatura czasowa procesora rejestruje liczbę cykli zegara od
ostatniego zresetowania.
    rdtsc ; Odczytanie danych z licznika taktow CPU (do rejestrow eax i
edx)

;XORSHIFT dla wiekszej losowosci danych, rand() z cpp ma to
wbudowane
;bez tego wartości były bardzo podobne, takie same
    mov esi, eax      ; przeniesienie wartości z rejestru eax do
rejestru esi. Wartość początkowa do przekształcenia.
    xor esi, edx      ; xor łączy bity z esi i edx, generując nową,
losową wartość.
    mov edx, esi      ;przeniesienie wyniku z rejestru esi do edx
    shl edx, 13       ;przesunięcie bitów w rejestrze edx o 13 pozycji w
lewo. (mnożenie 2^13)
    xor esi, edx      ;operacja XOR między rejestrami esi i edx. Dodaje
losowość do obecnej wartości w esi.
    mov edx, esi      ;esi->edx
    shr edx, 17       ;przesuniecie bitowe w rejestrze edx o 17 w prawo
(dzielenie przez 2^17)
    xor esi, edx      ;operacja XOR między rejestrami esi i edx. Dodaje
losowość do obecnej wartości w esi.
    mov edx, esi      ;esi->edx
    shl edx, 5        ;przesunięcie bitów w rejestrze edx o 5 pozycji w
lewo (mnożenie przez 2^5)
    xor esi, edx      ;ostateczna operacja XOR między esi i edx. Wynik
jest bardziej losowy, bazujący na początkowej wartości.
    ; Koncowo esi zawiera nasz seed na bazie zegara CPU

    test esi, esi ; Sprawdzenie bitu znaku seeda, operacja AND na esi
    jns NG_Randomize ; Jesli seed dodatni, pomin negacje, jns
testowanie bitu znaku
    neg esi ; Jesli ujemny zaneguj

```

#### NG\_Randomize:

```

    xor eax, eax ; Zerowanie eax
    xor edx, edx ; Zerowanie edx
    mov eax, esi ; esi (wartosc koncowa operacji xorshift)-> eax
    cdq ; Rozszerzenie 32-bit eax na 64-bit na eax + edx, po to aby
użyć instrukcji idiv
    idiv r15d ; Dzielenie z reszta przez r15d - wynik w eax, reszta w
edx(ta reszta nas interesuje), div zeruje zerszte z dzielenia
    mov r15d, edx ; edx -> r15d
    sub r15d, r14d ; r15d - X odejmujemy X wg algorytmu

```

```

    mov [rcx + r13*4], r15d ; Zapisanie wygenerowanej liczby do
tymczasowej tablicy rng_values, *4 bo 4 bajtowe slowa(dw, inty)

    inc r13 ; r13++
    cmp r13, 4 ; Dopoki r13 < 4, dopóki nie mamy 4 wartości
    jl NG_Loop ; Kontynuuj petle, jl jeżeli nie

    xorps xmm5, xmm5 ; Zerowanie xmm5 (jakby zostały tam jakies
smieci), xor dla wektorowych
    movdqu xmm5, xmmword ptr [rng_values] ; Przeniesienie zawartosci
tymczasowej tablicy do xmm5
    ;teraz w xmm5 4 wartosci dla 4 pikseli

    ; PRZYWROCENIE DANYCH ZE STOSU, w odwrotnej kolejnosci
    pop r13
    pop r15
    pop rsi
    pop rdx
    pop rax
    pop rcx

    jmp NG_Return ; Skok bezwarunkowy z powrotem do PixelLoop

ApplySepiaFilterAsm ENDP
END

```

### 3.4. Istotne fragmenty kodu ASM.

#### 3.4.1. Dwuetapowe wczytanie 96 bitów danych.

```

movlps xmm0, qword ptr [rcx] ; ladowanie pierwszych 64 bitow
movd xmm1, dword ptr [rcx+8] ; ladowanie kolejnych 32 bitow
movhlps xmm0, xmm1 ; laczenie obu rejestrow - kopiowanie 32 bitow z
dolnej czesci xmm1 na najblizsze wolne miejsca w gornej czesci xmm0

```

#### 3.4.2. Wyciągnięcie pojedynczych kanałów na podstawie maski dla kanałów BGR (przykład dla kanału B).

```

movdqa xmm2, xmm0 ; kopiowanie zawartosci xmm0 do xmm2
pshufb xmm2, xmm10 ; wyciagniecie kanalu B
mulps xmm2, xmm7 ; mnozenie przez wage B

```

#### 3.4.3. Dodanie przemnożonych kanałów RGB w celu uzyskania wartości szarości.

```

paddb xmm2, xmm3 ; B + G
paddb xmm2, xmm4 ; (B + G) + R [dod. 32-bitowe]

```

#### 3.4.4. Dodanie szumu do wartości szarości.

```

paddb xmm2, xmm5 ; Dodanie szumu

```

#### 3.4.5. Ograniczenie wartości pikseli do zakresu [0,255].

```
pxor xmm0, xmm0 ; Zerowanie xmm0
pcmpeqd xmm1, xmm1 ; Wszystkie bity xmm1 na 1
psrld xmm1, 24 ; Przesunięcie bitów o 24 w lewo - tworzy liczbę 255

pmaxsd xmm2, xmm0 ; Clamp 0 , ograniczenie wartości minimalne
pminsd xmm2, xmm1 ; Clamp 255 , ograniczenie wartości maksymalnej
```

#### 3.4.6. Wyodrębnienie pojedynczych kanałów do zapisu (przykład dla kanału B).

```
movdqa xmm0, xmm2 ; kopiowanie wartości szarości xmm2 do xmm0
pshufb xmm0, xmm13 ; wyodrębnienie kanału B do zapisu, z maską do zapisu
```

#### 3.4.7. Złączenie rejestrów w celu uzyskania danych dla wszystkich kanałów RGB.

```
paddb xmm3, xmm1 ; łączenie xmm1 z xmm3 (kanały R i G)
paddb xmm3, xmm0 ; łączenie xmm0 z xmm3 (kanały R, G i B)
```

#### 3.4.8. Dodanie parametru P do szarości z ograniczeniem do zakresu [0,255].

```
paddusb xmm3, xmm6 ; dodanie parametru P (0, P, 2P) do wartości szarości
```

#### 3.4.9. Dwuetapowe zapisanie 96 bitów do pamięci.

```
movq qword ptr [rcx], xmm3 ; zapisanie (xmm3) pierwszych 64 bitów
movhlps xmm3, xmm3 ; przeniesienie najwcześniej dolnych zajętych bitów xmm3 na początek rejestru
movd dword ptr [rcx + 8], xmm3 ; zapisanie pozostałych 32 bitów
```

#### 3.4.10. Obliczenie zakresu szumu ( $2 \cdot X + 1$ ).

```
mov r15d, r14d ;  $X \rightarrow r15d$ 
shl r15d, 1 ; shift left  $r15d \cdot 2 = 2X$  , mnożenie  $\cdot 2$  przez przesunięcie bitowe w lewo
add r15d, 1 ;  $r15d + 1 = 2X + 1$  ( inc r15d)
```

#### 3.4.11. Generowanie losowej liczby za pomocą zegara procesora RDTSC.

```
rdtsc ; Pobranie liczby cykli zegara procesora (do rejestrów eax i edx)
mov esi, eax ; Przeniesienie dolnej części liczby (EAX) do ESI
```

#### 3.4.12. Generowanie szumu przy użyciu XORSHIFT.

```
mov edx, esi ; Kopiowanie wartości ESI do EDX
shl edx, 13 ; przesunięcie bitów w rejestrze edx o 13 pozycji w lewo.
xor esi, edx ; operacja XOR między rejestrami esi i edx.
mov edx, esi ; esi->edx
shr edx, 17 ; przesunięcie bitowe w rejestrze edx o 17 w prawo.
xor esi, edx ; operacja XOR między rejestrami esi i edx.
mov edx, esi ; esi->edx
shl edx, 5 ; przesunięcie bitów w rejestrze edx o 5 pozycji w lewo
xor esi, edx ; ostateczny XOR dla maksymalnej losowości
```

### 3.4.13. Konwersja liczby losowej na zakres szumu (-X do X).

```
mov eax, esi ; esi (wartosc koncowa operacji xorshift)-> eax
cdq ; Rozszerzenie 32-bit eax na 64-bit na eax + edx
idiv r15d ; Dzielenie z reszta przez r15d - wynik w eax, reszta w
edx(ta reszta nas interesuje), div zeruje zreszte z dzielenia
mov r15d, edx ; edx -> r15d
sub r15d, r14d ; Przesunięcie zakresu do [-X, X] przez odjęcie X
```

### 3.4.14. Zapisanie wyników szumu do tymczasowej tablicy rng\_values

```
mov [rcx + r13*4], r15d ; Zapisanie wygenerowanej liczby do tymczasowej
tablicy rng_values
```

## 3.5. Kod biblioteki C++

```
#include "pch.h"
#include <iostream>
#include <cstdlib> // rand()
#include <windows.h>

//Funkcja pomocnicza do ograniczania wartości do zakresu [low, high]
//Jeśli wartość jest mniejsza niż 'low', zwraca 'low', jeśli większa niż
'high', zwraca 'high'.
//W przeciwnym razie zwraca oryginalną wartość.
inline int clamp(int value, int low, int high) {
    return (value < low) ? low : (value > high ? high : value);
}

//Eksportowana funkcja ApplySepiaFilter, która przyjmuje zakres wierszy
obrazu do przetworzenia
extern "C" __declspec(dllexport) void ApplySepiaFilter(unsigned char*
pixelBuffer, int width, int bytesPerPixel, int P, int X, int startRow, int
endRow, int stride)
{
    // Iteracja po wierszach obrazu od startRow do endRow
    for (int y = startRow; y < endRow; y++)
    {
        //Początek bieżącego wiersza w buforze pamieci
        int rowStart = y * stride;

        // Iteracja po pikselach w bieżącym wierszu
        for (int x = 0; x < width; x++)
        {
            // Obliczenie indeksu bieżącego piksela w buforze
            int pixelIndex = rowStart + x * bytesPerPixel;

            //Pobranie kanałów RGB z bufora
            unsigned char B = pixelBuffer[pixelIndex]; //Blue
            unsigned char G = pixelBuffer[pixelIndex + 1]; //Green
            unsigned char R = pixelBuffer[pixelIndex + 2]; //Red

            //Konwersja na odcień szarości
            int gray = static_cast<int>(0.299 * R + 0.587 * G + 0.114 *
B);

            // Wyliczenie nowych wartości kanałów dla efektu sepia
            int newB = gray;
            int newG = P + gray;
            int newR = 2 * P + gray;
```

```

        // Ograniczenie wartości do zakresu [0, 255]
        newB = clamp(newB, 0, 255);
        newG = clamp(newG, 0, 255);
        newR = clamp(newR, 0, 255);

        // Dodanie efektu postarzenia z parametrem X
        int noise = (rand() % (2 * X + 1)) - X; // losowa wartosc z
przedziału [-X, X]
        newB = clamp(newB + noise, 0, 255);
        newG = clamp(newG + noise, 0, 255);
        newR = clamp(newR + noise, 0, 255);

        //Zapisanie przerobionych wartości RGB do bufora
        pixelBuffer[pixelIndex] = static_cast<unsigned char>(newB);
//kanał Blue
        pixelBuffer[pixelIndex + 1] = static_cast<unsigned
char>(newG); //kanał Green
        pixelBuffer[pixelIndex + 2] = static_cast<unsigned
char>(newR); //kanał Red
    }
}
}

```

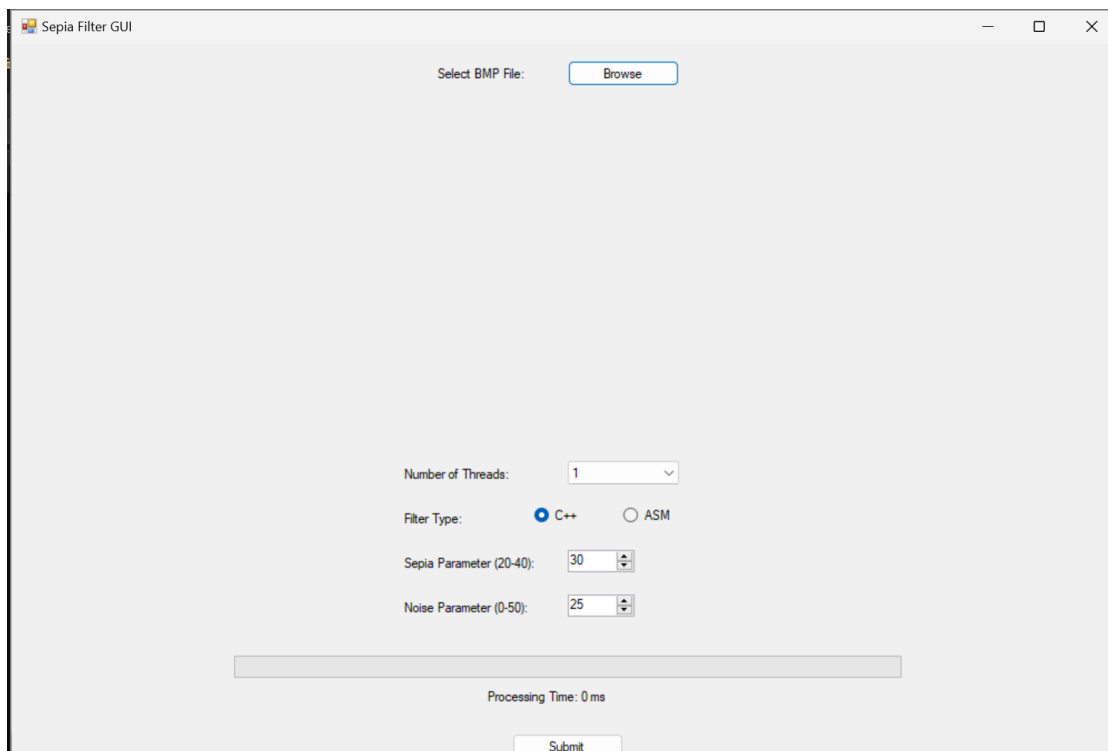
## 4. Interfejs użytkownika

### 4.1. Opis interfejsu użytkownika

Interfejs zawiera następujące pola:

- Browse – pole wyboru pliku BMP, umożliwia wczytanie obrazu do przetworzenia.
- Number of Threads – rozwijana lista liczby wątków do wyboru.
- Filter Type – opcje wyboru rodzaju filtra C++ lub ASM.
- Sepia Parametr – pole liczbowe do wybrania wartości kontrolującej intensywność efektu sepia.
- Noise Parametr – pole liczbowe do wybrania wartości kontrolującej intensywność efektu szumu (efekt postarzenia).
- Pasek postępu.
- Processing Time – informuje o czasie wykonania algorytmu.
- Submit – przycisk do uruchomienia procesu przetwarzania obrazu.

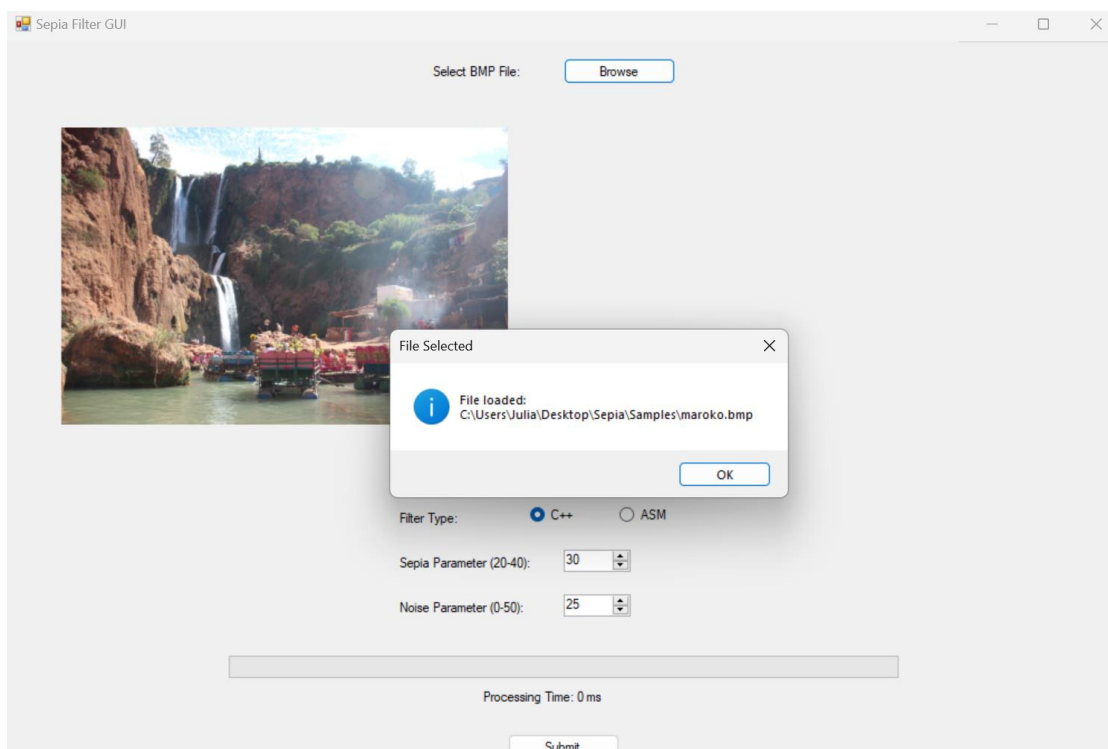
### 4.2. Interfejs został zaimplementowany w Windows Forms



*Zrzut ekranu 1 Wygląd Interfejsu Użytkownika po włączeniu program*

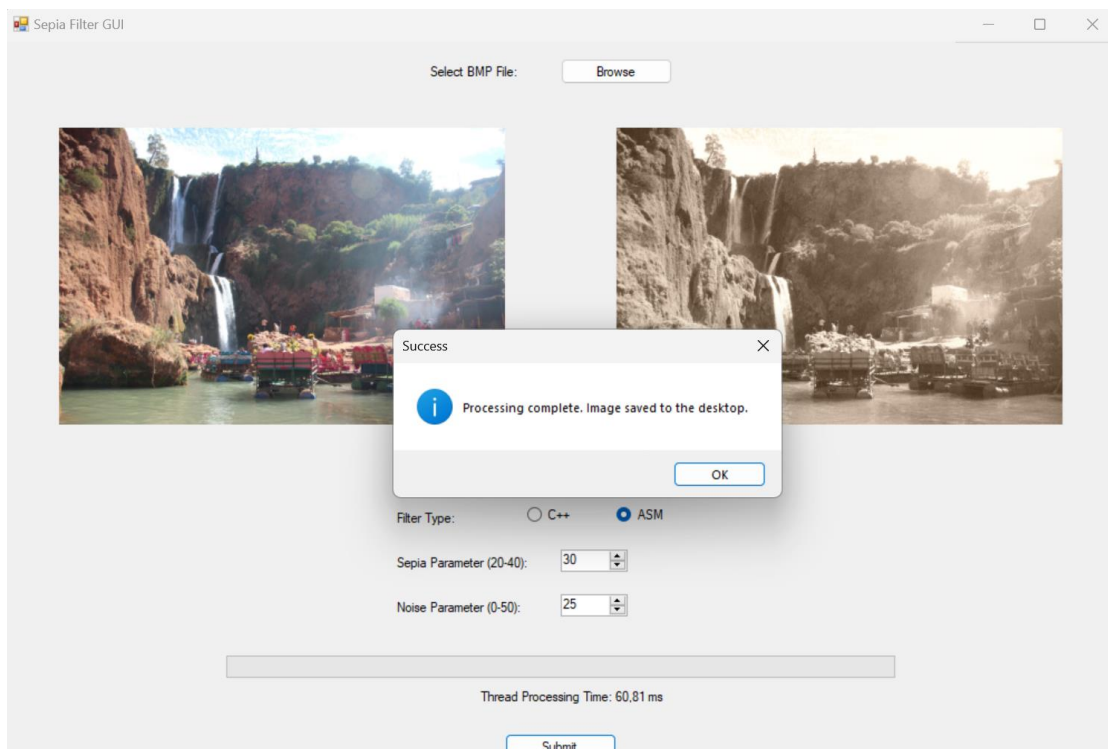
#### 4.3. Efekt działania aplikacji

Po wgraniu obrazu pojawia się komunikat o jego poprawnym załadowaniu.



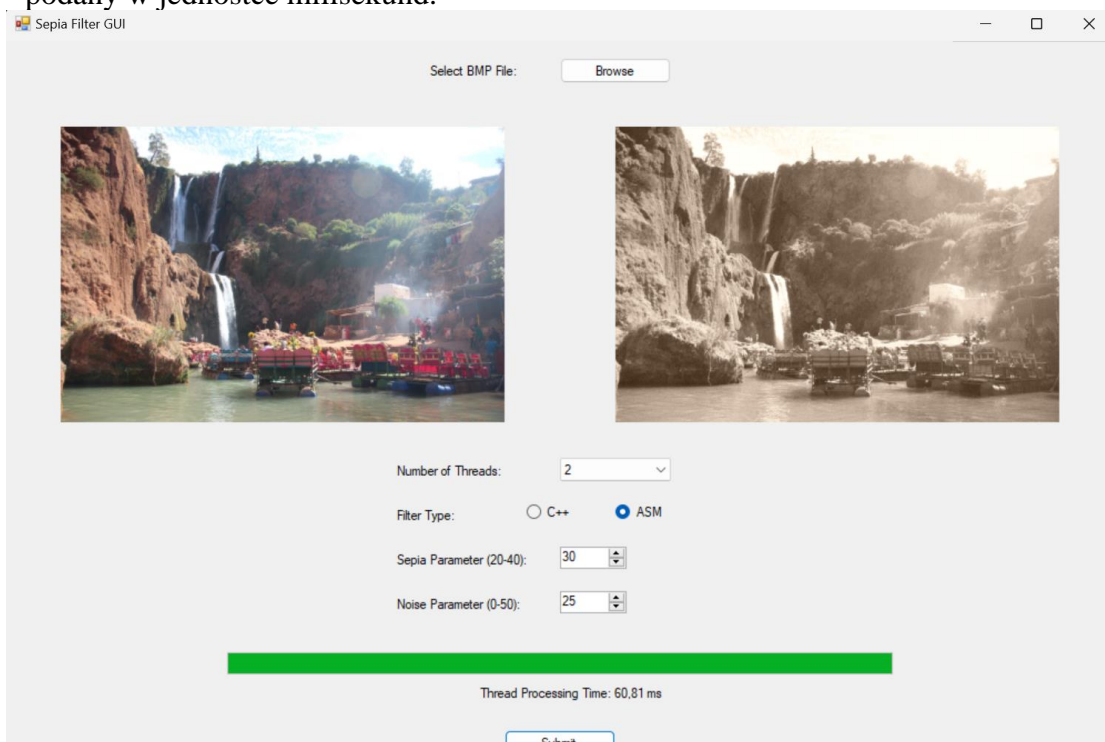
*Zrzut ekranu 2 Poprawne załadowanie pliku*

W następnym kroku użytkownik wybiera odpowiednie parametry, w zależności od tego jaki efekt chce uzyskać. Następnie klika przycisk “Submit”. Pojawia się komunikat o poprawnym zakończeniu procesu, oraz że przetworzony obraz został zapisany na pulpicie urządzenia z którego korzysta.



*Zrzut ekranu 3 Proces zakończył się sukcesem*

Po kliknięciu “OK” pasek ładowania program zmienia kolor na zielony co świadczy o pozytywnym zakończeniu przetwarzania obrazu. Poniżej widać czas przetwarzania podany w jednostce milisekund.



*Zrzut ekranu 4 Końcowy efekt działania programu*



## 5. Wyniki czasowe

5.1. Przeanalizowano wyniki czasu wykonania algorytmu dla trzech plików obrazów o rozmiarach 2400x1600, 1200x800, 480x360 pikseli z wykorzystaniem biblioteki ASM oraz biblioteki CPP bez optymalizacji, z optymalizacją czasową oraz optymalizacją pamięciową. Wykonano 5 pomiarów czasowych dla każdej istotnej liczby wątków [1, 2, 4, 8, 16, 32, 64] z których wyciągnięto średnią ważoną.

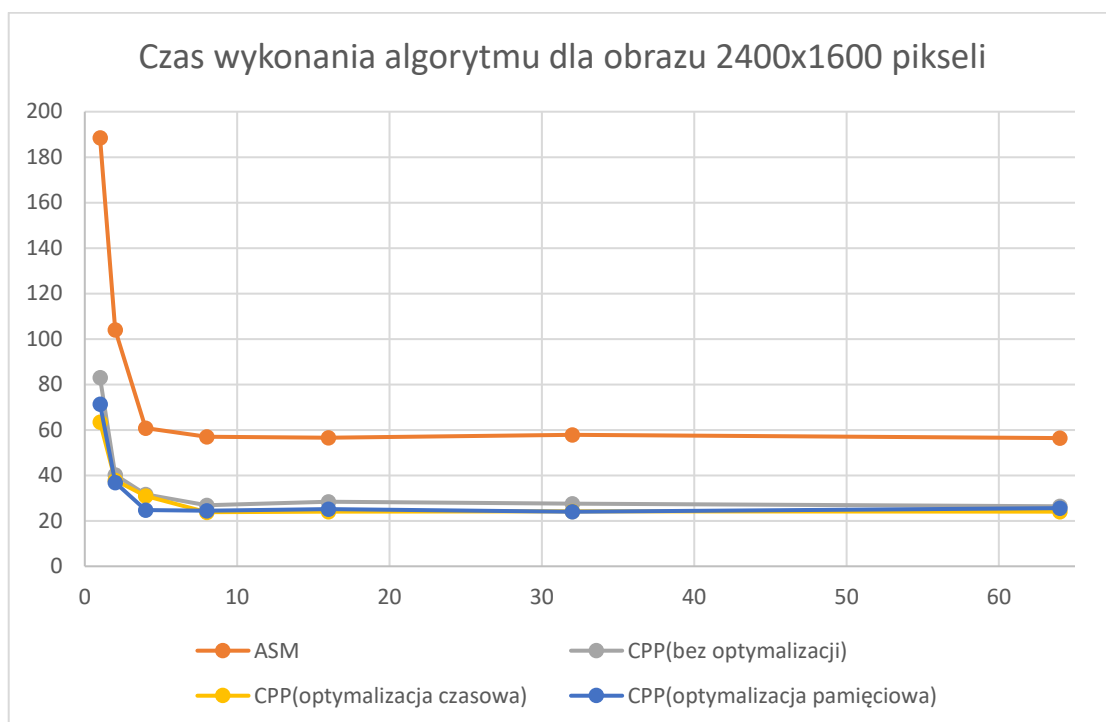
5.2. Tabele z uzyskanymi wynikami.

2400x1600 pikseli				
Wątki	ASM [ms]	CPP(bez optymalizacji) [ms]	CPP(optymalizacja czasowa) [ms]	CPP(optymalizacja pamięciowa) [ms]
1	188,6	83	63,4	71,4
2	104	40,2	37,8	36,8
4	60,8	31,6	31	24,8
8	56	26,8	23,8	24,4
16	56,6	28,4	24	25,2
32	57,8	27,6	24,2	24
64	56,4	26,4	24	25,6

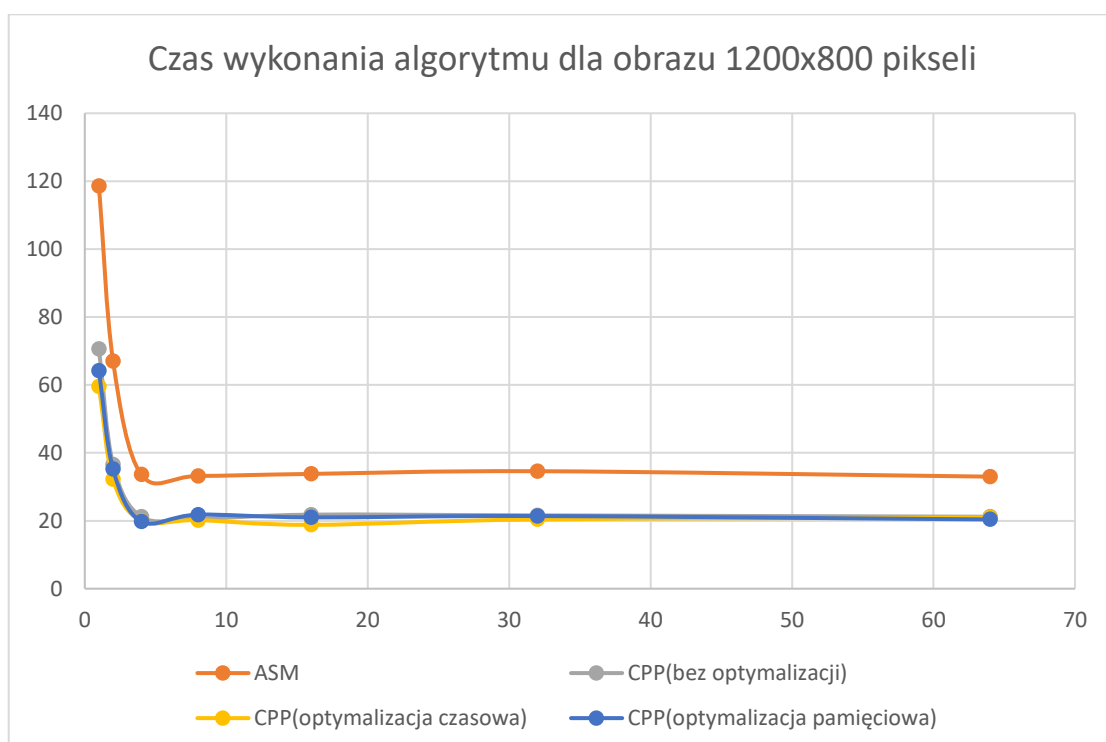
1200x800 pikseli				
Wątki	ASM [ms]	CPP(bez optymalizacji) [ms]	CPP(optymalizacja czasowa) [ms]	CPP(optymalizacja pamięciowa) [ms]
1	118,6	70,6	59,6	64,2
2	67	36,6	32	25,2
4	33,6	21,2	20	19,8
8	33,2	20,6	20,2	21,8
16	33,8	21,8	18,8	21
32	34,6	21,6	20,4	21,4
64	33	21,2	21	20,4

480x360 pikseli				
Wątki	ASM [ms]	CPP(bez optymalizacji) [ms]	CPP(optymalizacja czasowa) [ms]	CPP(optymalizacja pamięciowa) [ms]
1	48,4	42,4	37	40,8
2	28	23	20,4	21,3
4	16	12,8	10,6	11,8
8	17,4	13,6	11,2	12,4
16	18,2	12,8	11	11,6
32	16,4	12,6	11	12
64	16,2	13	11,4	12,2

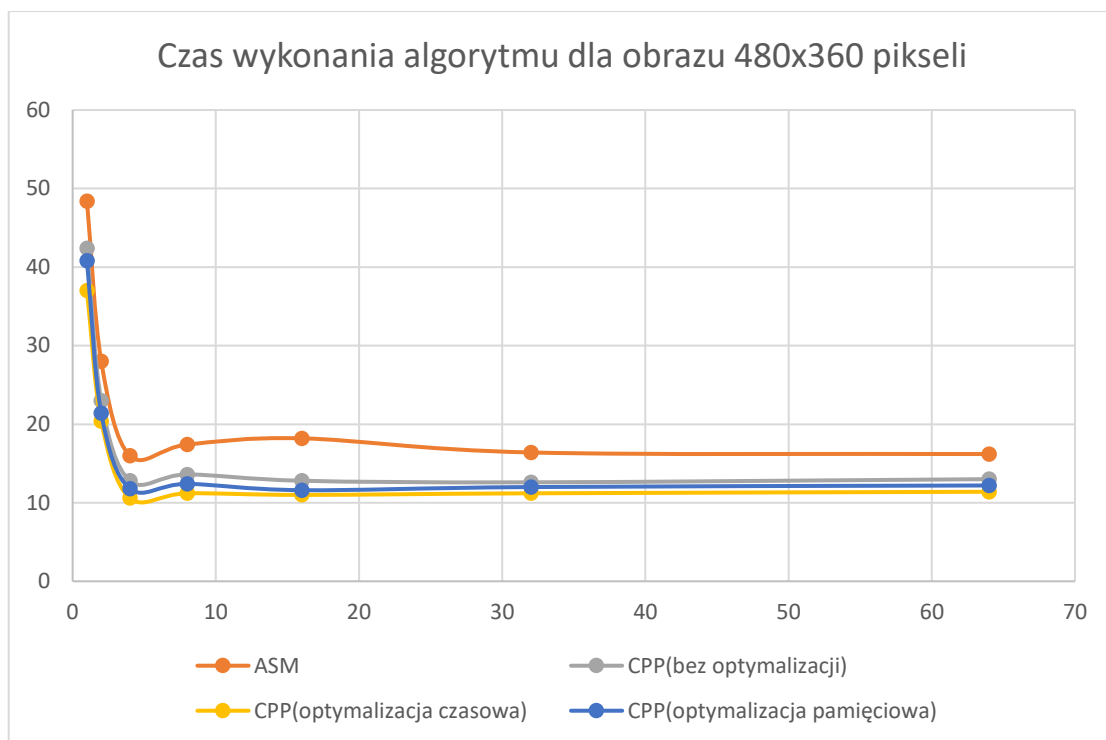
### 5.3. Wykresy czasowe.



Wykres 1 Wykres dla bitmapy o wielkości 2400x1600 pikseli



Wykres 2 Wykres dla bitmapy o wielkości 1200x800 pikseli



Wykres 3 Wykres dla bitmapy o wielkości 480x360 pikseli

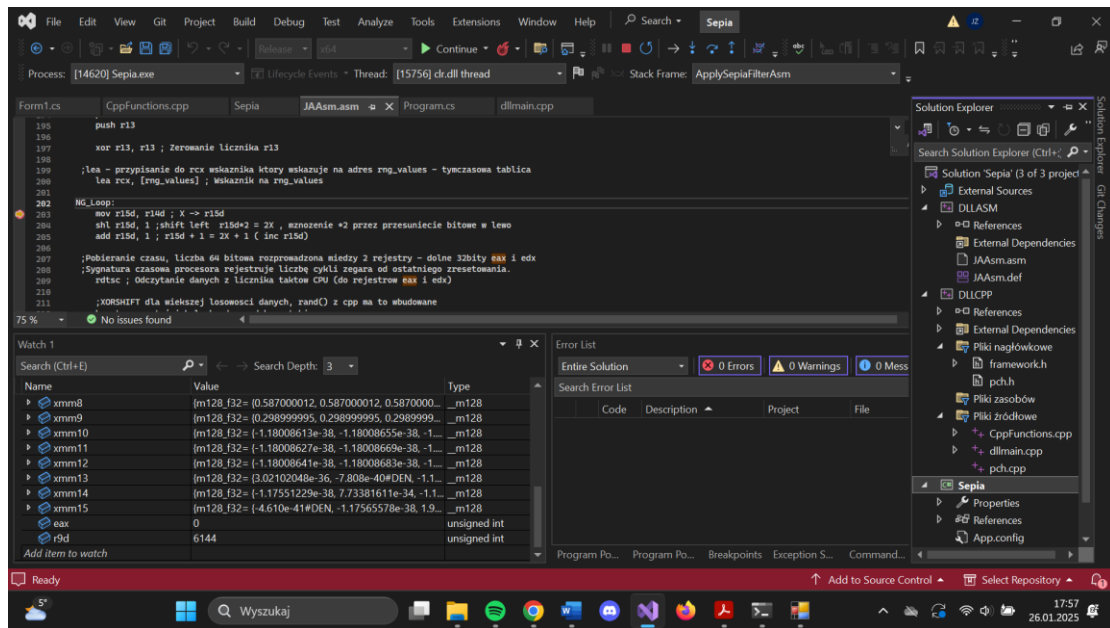
Z powyższych wykresów można wywnioskować, że wykonanie algorytmu w ASM zawsze zajmuje więcej czasu niż w CPP, chociaż od 4 wątków wartość ta jest już zbliżona ze względu na wykonanie pomiarów na komputerze 4 wątkowym. Przed przystąpieniem do pisania projektu zakładałam, że wyniki będą odwrotne. Oto kluczowe punkty dlaczego działanie ASM okazało się wolniejsze niż CPP:

- Kod w CPP wykorzystuje funkcję `rand()` do generowania szumu, która działa szybko w kontekście generowania pojedynczych liczb. Nie jest ona natomiast tak precyzyjna jak algorytm XORSHIF w ASM.
- Algorytm ASM jest bardziej złożony pod kątem generowania szumu, ale złożoność instrukcji tj. `rdtsc`, `xor`, `shl`, `shr`, `idiv` wprowadza dodatkowy narzut czasowy, który nie wpływa korzystnie na optymalizację czasową.
- Kod ASM operuje na blokach 16 bajtowych przy pomocy rejestrów SIMD, co wymaga dopasowania struktury danych. W przeciwieństwie do tego kod w CPP działa na poziomie pojedynczych pikseli. Nie wymaga to dodatkowego wyrównania ani skomplikowanych masek.

Wykresy pokazują zgodną zależność, że im większy obraz tym czasy wykonania są dłuższe. Sama optymalizacja CPP nie przyniosła jednak spektakularnych oszczędności czasowych.

## 6. Testowanie

Testowanie programu zostało przeprowadzone w trybie Release dla architektury X64. W trakcie pisania biblioteki ASM możliwe było debugowanie kodu oraz podgląd wykorzystywanych rejestrów.



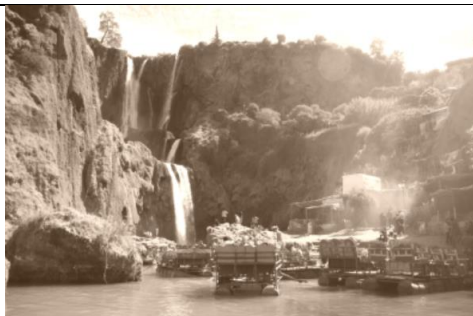
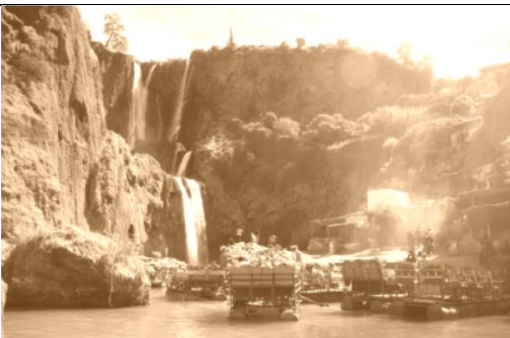






Zrzut ekranu 5 Testowanie kodu asemblerowego

Dodatkowo porównano wyniki końcowe działania aplikacji, dla różnych skrajnych wartości P i X oraz dla dwóch bibliotek CPP i ASM. Testowano plik wejściowy o wymiarach 2048 x 1365 pikseli.



*Obraz 1 Plik wejściowy do testów.*

	P = 20	P = 40
X=0	CPP	CPP
		
	ASM	ASM
		
X=50	CPP	CPP
		
	ASM	ASM
		

## 7. Wnioski

Realizacja projektu wykazała interesujące różnice w wydajności między implementacją algorytmu w języku asemblera a językiem wysokiego poziomu (C++). Pomimo wstępnego założenia, że program asemblera będzie szybszy dzięki precyzyjnemu dostępowi do sprzętu i użyciu instrukcji wektorowych SIMD, w tym przypadku czasy wykonania w C++ okazały się krótsze. Projekt umożliwił praktyczne zrozumienie wyzwań związanych z implementacją algorytmów niskopoziomowych, takich jak konieczność wyrównania pamięci czy zarządzanie rejestrami SIMD.

Przeprowadzane testy na 4-rdzeniowym procesorze pokazały, że optymalne wykorzystanie wszystkich wątków ma kluczowe znaczenie dla osiągnięcia krótszych czasów przetwarzania. Projekt pozwolił także na zdobycie praktycznych umiejętności w implementacji dynamicznie dołączanych bibliotek (DLL) w różnych językach oraz ich integracji z aplikacją wielowątkową. Dzięki temu doświadczeniu możliwe było lepsze zrozumienie ograniczeń i zalet zarówno asemblera, jak i C++.