
	Instytut Informatyki Politechniki Śląskiej Zespół Mikroinformatyki i Teorii Automatów Cyfrowych			
Rok akademicki:	Rodzaj studiów:	Przedmiot:	Grupa	Sekcja
2022/2023	SSI	JA	2	1
Imię:	Daniel	Prowadzący:	AO	
Nazwisko:	Cogiel			
<h2>Raport końcowy</h2>				
Temat projektu: <h1>Horyzontalne rozmycie obrazu</h1>				
Data oddania: dd/mm/rrrr		03/02/2023		

1. Cel projektu.

Celem implementowanego algorytmu jest rozmycie horyzontalne obrazu z maską 5x1 (.bmp z głębią 24-bitową). Algorytm został zaimplementowany w bibliotekach dynamicznych – wersja napisana w języku assemblerowym x64 i wersja C++ dla porównania osiągnięć czasowych. Interfejs graficzny został zaimplementowany w języku C++ przy pomocy biblioteki Windows Forms.

2. Parametry wejściowe.

Uruchomienie aplikacji nie wymaga podawania żadnych parametrów, natomiast samo uruchomienie algorytmu wymaga wybrania obrazu do rozmycia, co jest możliwe przy pomocy stworzonego interfejsu graficznego. Program został zaprojektowany pod kątem przetwarzania obrazów .bmp z głębią 24-bitową.

3. Kod algorytmu assemblerowego.

```
.data
myVar db 0, 4, 8, 12, 12 dup (-1)

.CODE

;#####
; BlurProc(unsigned char* imageData, unsigned char* blurredImageData,
; unsigned long bytesPerLine, unsigned long linesToProcess)
; RCX - imageData
; RDX - blurredImageData
; R8 - bytesPerLine
; R9 - linesToProcess
;
; R10 - horizontal counter
; R11 - vertical counter
; R12 - start new line address counter (because line may not be dividable by 3)
; R14 - imageData current pointer
; R15 - blurredImageData current pointer
; XMM0 - currently calculated pixel
; XMM1 - register to store values of pixels used to calculated average value of current pixel
; XMM6 - vector of dword 3s used to multiply xmm0 values
; XMM7 - vector of dword 4s used to right-shift xmm0 values
;#####

BlurProc proc
    add rcx, 9                ;set proper starting point for original image pointer
    mov r14, rcx              ;move original image pointer to r14
    mov r15, rdx              ;move blurred image pointer to r15
    mov r10, r8               ;set horizontal counter to bytesPerLine value
```

```

mov r11, r9
mov r12, 0
mov eax, 3
movd xmm6, eax
vpbroadcastd xmm6, xmm6
mov eax, 4
movd xmm7, eax
vpbroadcastd xmm7, xmm7
VerticalLoop:
cmp r11, 0
je Finish
HorizontalLoop:
cmp r10, 0
jle ExitRowLoop
pmovzxbd xmm0, dword ptr [r14]
pmovzxbd xmm1, dword ptr [r14+3]
padd xmm0, xmm1
pmovzxbd xmm1, dword ptr [r14+6]
padd xmm0, xmm1
pmovzxbd xmm1, dword ptr [r14+3]
padd xmm0, xmm1
pmovzxbd xmm1, dword ptr [r14+6]
padd xmm0, xmm1
pmulld xmm0, xmm6
padd xmm0, xmm6
vpsrlvd xmm0, xmm0, xmm7
psrhubd xmm0, xmm0, word ptr [r14+3]
pextrd eax, xmm0, 0
mov [r15], dword ptr eax
sub r10, 3
add r14, 3
add r15, 3
jmp HorizontalLoop
ExitRowLoop:
dec r11
mov r10, r8
add r12, r8
mov rax, rcx
add rax, r12
mov r14, rax
mov rax, rdx
add rax, r12
mov r15, rax
jmp VerticalLoop
Finish:
ret
BlurProc endp
END

```

```

;set vertical counter to linesToProcess value
;set new line address counter to 0
;move 4 dwords of value 3 to xmm6
;
;
;move 4 dwords of value 4 to xmm7
;
;
;if vertical counter == 0
;finish algorithm
;if horizontal counter == 0
;move on to next line
;load 4 bytes of original image data to xmm0's dwords
;load 4 bytes of original image 3 bytes to the left to xmm1's dwords
;add xmm1's and xmm0's dwords
;load 4 bytes of original image 6 bytes to the left to xmm1's dwords
;add xmm1's and xmm0's dwords
;load 4 bytes of original image 3 bytes to the right to xmm1's dwords
;add xmm1's and xmm0's dwords
;load 4 bytes of original image 6 bytes to the right to xmm1's dwords
;add xmm1's and xmm0's dwords
;perform divide by 5 on xmm0's dwords (multiply dwords by 2^4 / 5 = 3)
;add 2^4 / 5 to xmm0's dwords
;shift xmm0's dwords right by 4 bits
;convert 4 xmm0's dwords to 4 bytes in last dword
;move xmm0's last dword value to eax
;move eax's value to where blurred image pointer is pointing
;lower horizontal counter by 3
;move original image pointer 3 bytes right
;move blurred image pointer 3 bytes right
;jump to next iteration
;decrement vertical counter
;reset horizontal counter
;increment new line address counter by bytesPerLine value
;move initial original image pointer value to rax
;add new line address counter to initial original image pointer value
;set current original image pointer to rax's 4-byte aligned value
;move initial blurred image pointer value to rax
;add new line address counter to initial original image pointer value
;set current blurred image pointer value to rax's 4-byte aligned value
;move on to next iteration
;Return

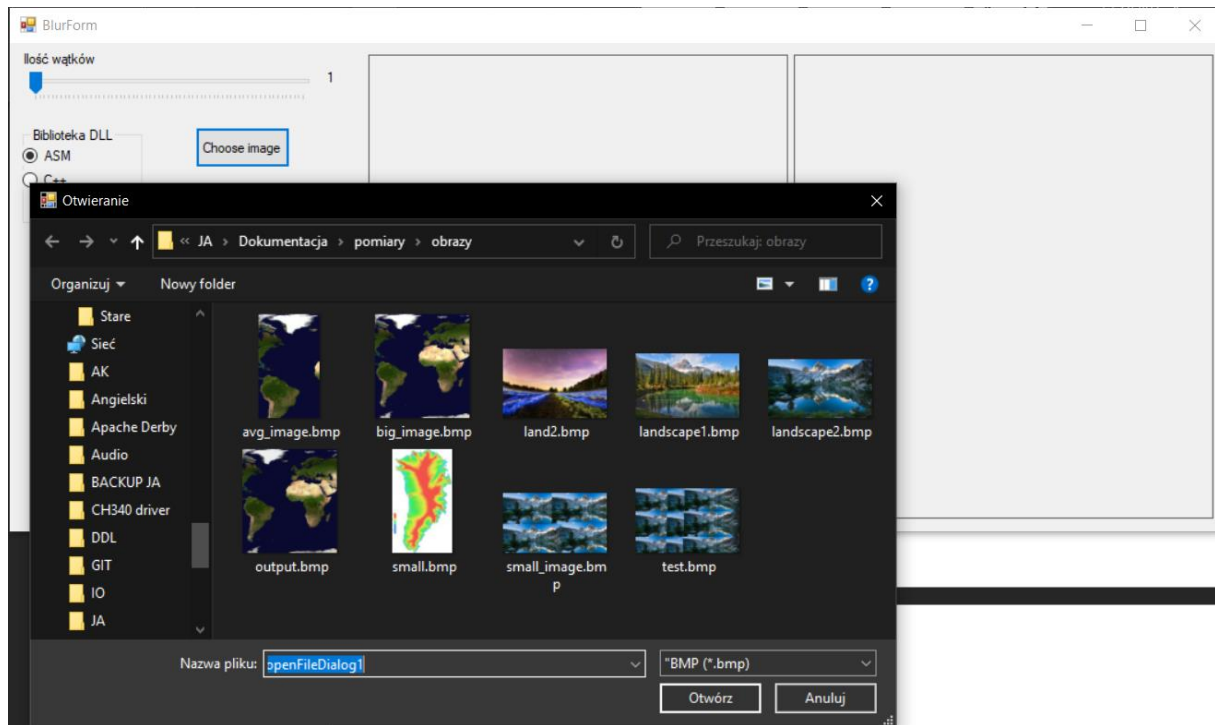
```

4. Interfejs użytkownika.

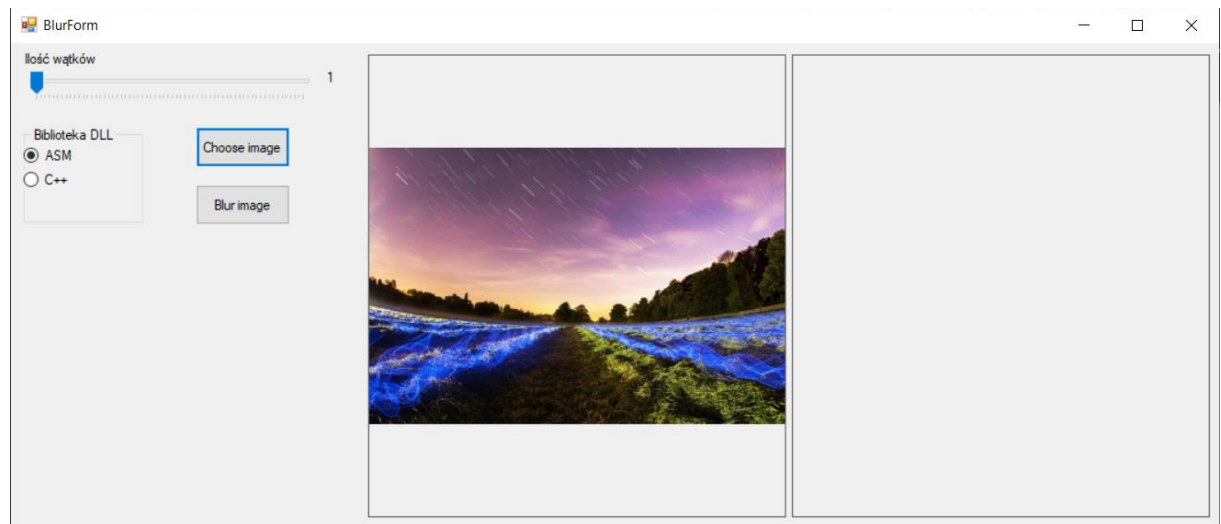
- *od razu po uruchomieniu aplikacji*



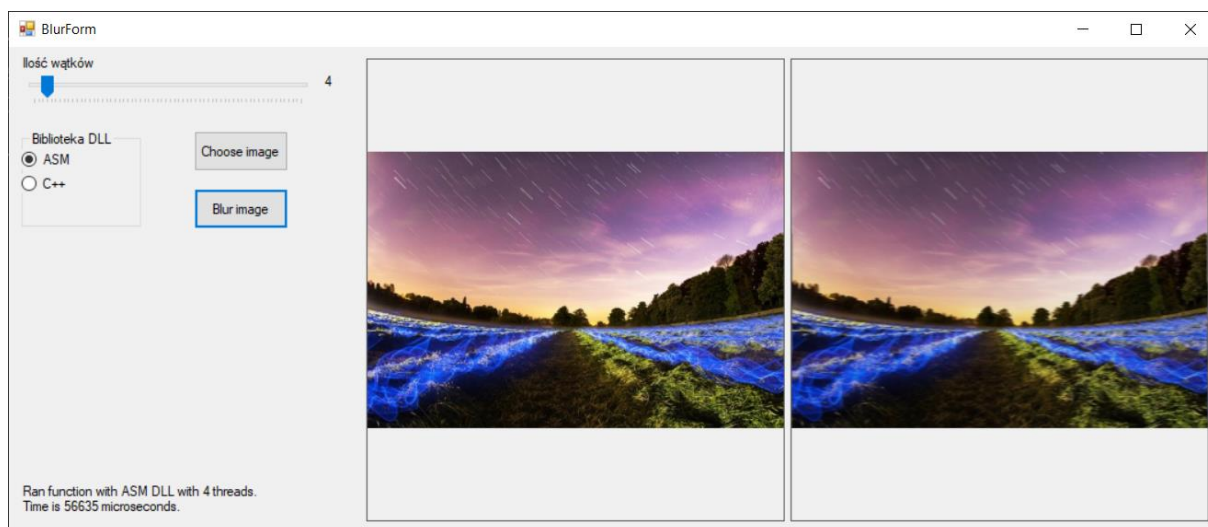
- *wybór obrazu do przetworzenia*



- *po wyborze obrazu*

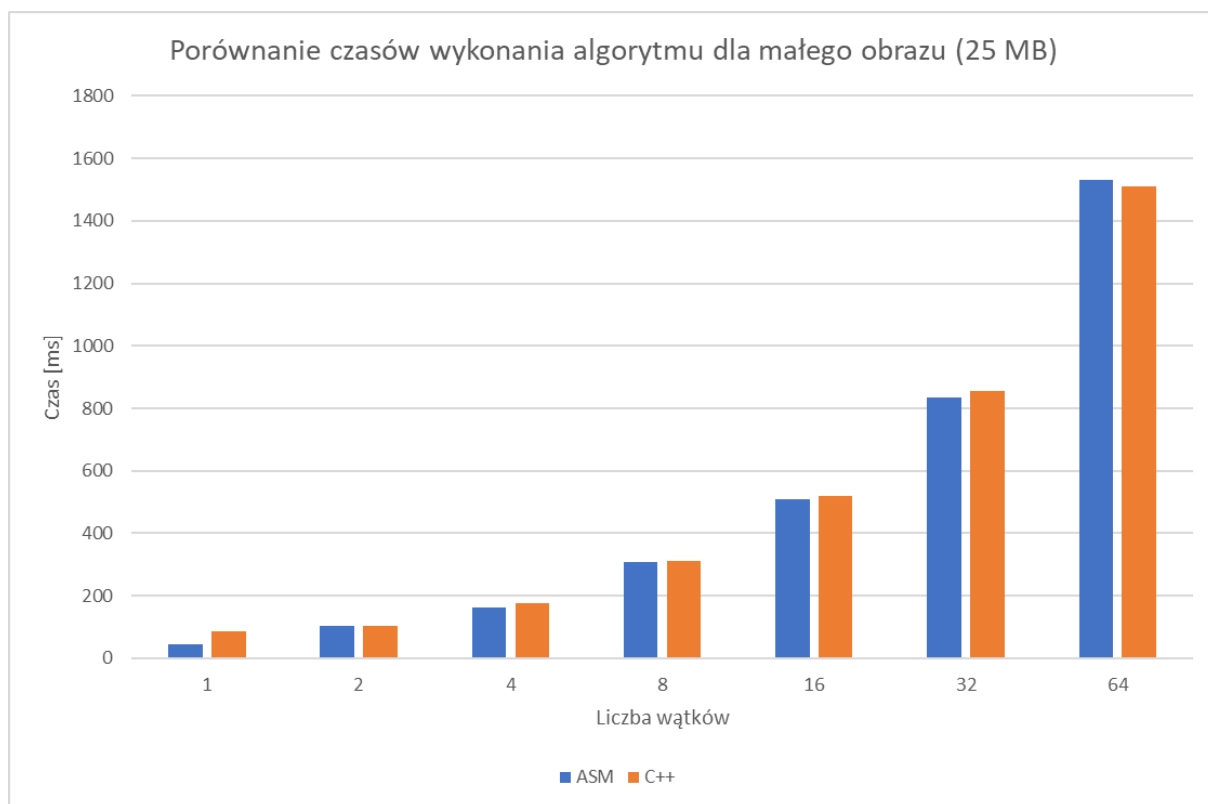


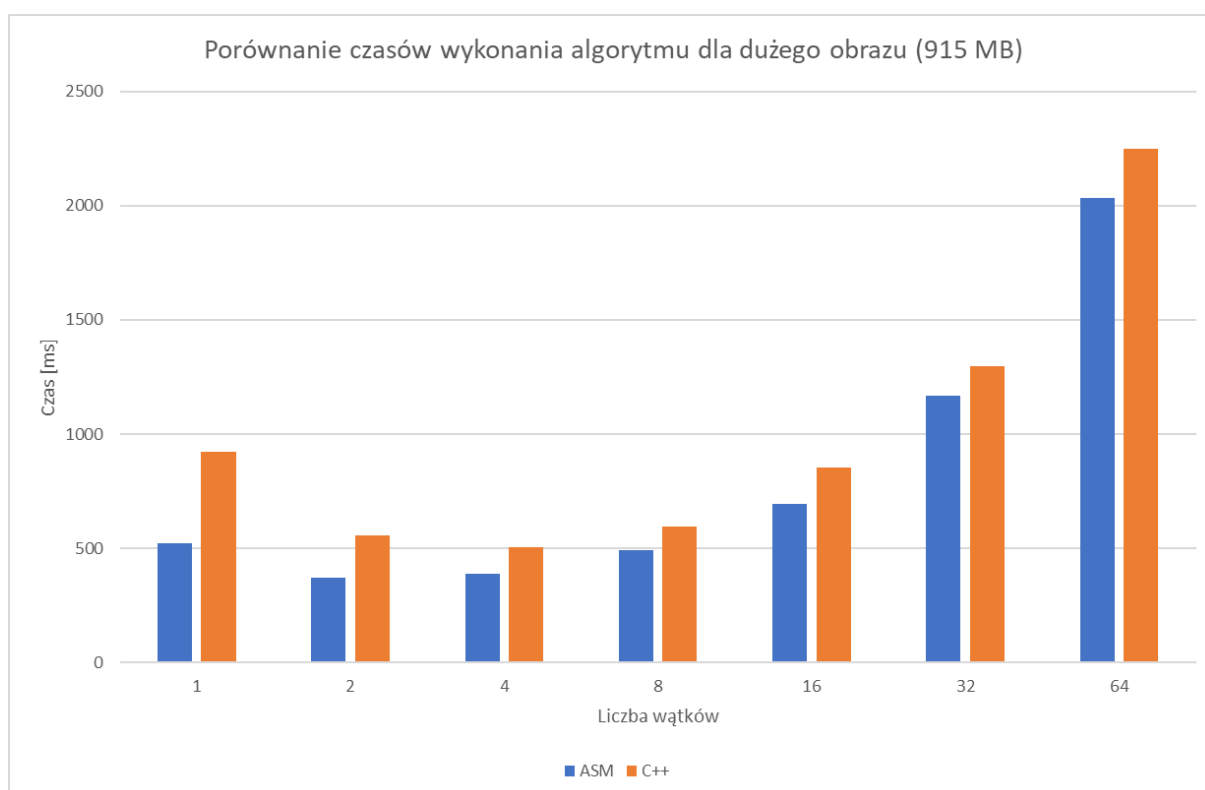
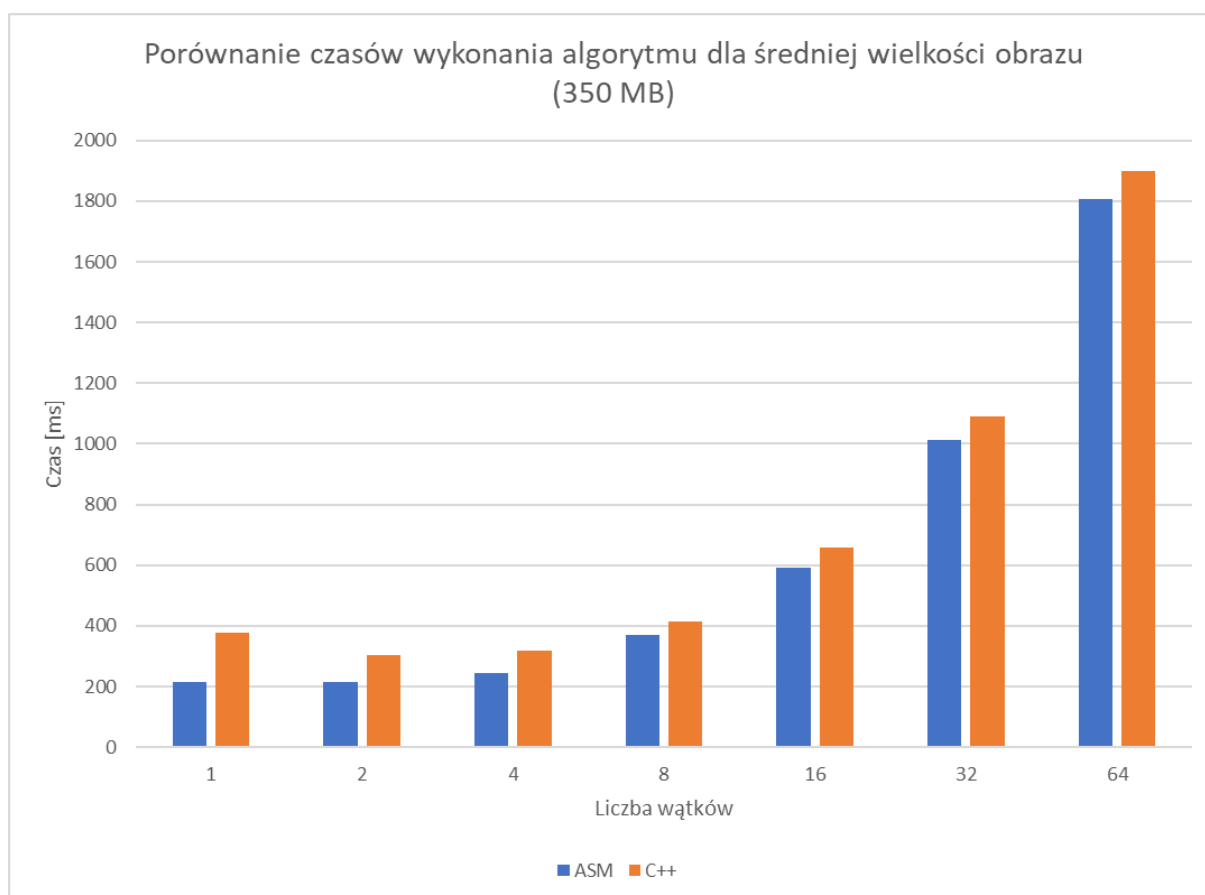
- *po przetworzeniu*



5. Szybkość działania.

Biblioteka C++ uruchamiana z optymalizacją szybkościową /O2.





6. Testowanie i uruchamianie.

Program został przetestowany dla plików .bmp (z głębia 24-bitową) o szerokim zakresie wielkości (do 915 MB). Zaimplementowany algorytm działa poprawnie, wszystkie obrazy są rozmywane horyzontalnie. Program uruchamia się zarówno w trybie Release x64, jak i Debug x64. Podczas testów szybkościowych biblioteka DLL C++ została ustawiona z optymalizacją czasową preferującą szybkość (/O2) .

7. Wnioski.

Z przedstawionych wykresów czasowych wynika, że zgodnie z założeniem pisanie algorytmów w języku assemblerowym faktycznie skraca czas jego wykonania. W przypadku zaimplementowanego algorytmu przyspieszenie szczególnie dobrze widoczne jest dla obrazów o dużej wielkości, a najlepiej przewagę assemblera nad C++ widać w przypadku uruchamiania algorytmu w jednym wątku. Ze względu na dużą szybkość algorytmu korzyści z pracy wielowątkowej widać dopiero w przypadku obrazów o rozmiarze około 300 MB. Algorytm testowany był na procesorze 4-rdzeniowym i dla obrazu 915 MB faktycznie widać, że w przypadku C++ największa szybkość wykonania jest osiągnięta dla 4 wątków. Dla tego samego obrazu assembler największą szybkość osiąga dla 2 wątków. Aby program assemblerowy najszybciej działał na 4 wątkach wymagany byłby jeszcze większy obraz. Należy zwrócić uwagę na to, że przy implementacji algorytmu assemblerowego użyto instrukcji wektorowych (SSE), co również pozwoliło na przyspieszenie szybkości wykonania. W trakcie implementacji wielokrotnie napotykane były błędy naruszenia dostępu do pamięci – jest to rzecz, na którą szczególnie należy uważać podczas implementacji algorytmów w języku assemblerowym.