TEMAT PROJEKTU:

FILTR RETRO DO ZDJĘĆ

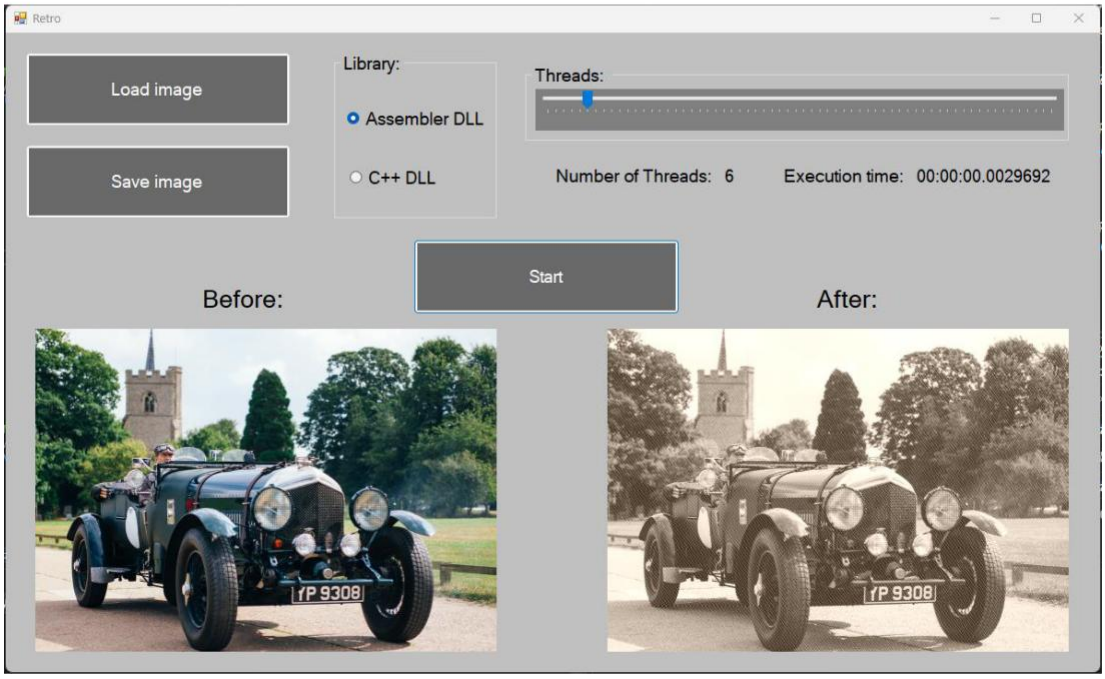
Dawid Kosiński

Cel pracy: Celem projektu było porównanie implementacji algorytmu filtru retro obrazu w języku wysokiego poziomu, w tym przypadku C++, oraz języku asemblera. Głównym celem było zbadanie różnic w wydajności obu implementacji oraz analiza prędkości wykonania algorytmu w zależności od różnych konfiguracji wątków.

1. Opis interfejsu użytkownika

Interfejs użytkownika został zaprojektowany w formie aplikacji okienkowej. Użytkownik ma możliwość wczytania obrazu w formacie BMP, wyboru biblioteki (asembler lub C++) oraz ustawienia liczby wątków. Po zastosowaniu efektu retro użytkownik ma możliwość zapisania przetworzonego obrazu w wybranym przez siebie miejscu na dysku. Przypadki użycia obejmują:

- Wczytanie obrazu z pliku BMP.
- Wybór biblioteki (asembler lub C++).
- Ustawienie liczby wątków.
- Przetworzenie obrazu za pomocą zastosowanego filtru.
- Zapisanie przetworzonego obrazu.



Założenia dotyczące parametrów wejściowych:

- Pliki wejściowe muszą być w formacie BMP.

2. Opis biblioteki Asemblerowej

Algorytm rozpoczyna swoje działanie od inicjalizacji rejestrów r11 i r10 na podstawie parametrów funkcji. Rejestry te określają początek (r11) i koniec (r10) obszaru, na którym zostanie zastosowany efekt retro. Poza tym, wczytywane są współczynniki efektu retro z pamięci do odpowiednich rejestrów xmm1, xmm2 i xmm3.

Następnie algorytm oblicza długość przetwarzanego zakresu, który będzie podlegał efektowi retro. Długość ta jest dzielona przez 4, ponieważ każdy piksel jest reprezentowany przez 4 bajty (RGBA).

Inkrementacja i sprawdzenie warunków dla piksela.

W poniższym fragmencie kodu, algorytm sprawdza resztę z dzielenia licznika pikseli (r12) przez 11. Jeśli reszta z dzielenia wynosi 0 lub 1, to na piksel zostanie nałożony biały kolor, w przeciwnym przypadku, piksel będzie poddany efektowi retro.

```
inc r12
mov rax, r12
mov rbx, 11
cdq
idiv rbx
cmp rdx, 0
je applyWhite
cmp rdx, 1
je applyWhite
```

W poniższym fragmencie kodu, piksel jest modyfikowany za pomocą operacji arytmetycznych, takich jak mnożenie i dodawanie, z wykorzystaniem odpowiednich współczynników efektu retro zawartych w rejestrach xmm1, xmm2 i xmm3.

```
movdqu xmm0, oword ptr[rcx]
mulps xmm0, xmm2
movshdup xmm4, xmm0
addps xmm0, xmm4
movhlps xmm4, xmm0
addps xmm0, xmm4
punpckldq xmm0, xmm0
punpcklqdq xmm0, xmm0
addps xmm0, xmm1
minps xmm0, xmm3
```

Co 11 i 12 piksel zostaje zastąpiony białym kolorem. Piksel biały ma wartości maksymalne w skali kolorów, co oznacza, że wszystkie składowe kolorów (czerwona, zielona i niebieska) są ustawione na maksymalną wartość.

```
applyWhite:
    movdqu xmm0, xmm3
```

Po przetworzeniu piksela, jego wartość zostaje zapisana z powrotem do pamięci, a wskaźnik przesuwany jest na kolejny piksel. Licznik pikseli jest odpowiednio aktualizowany.

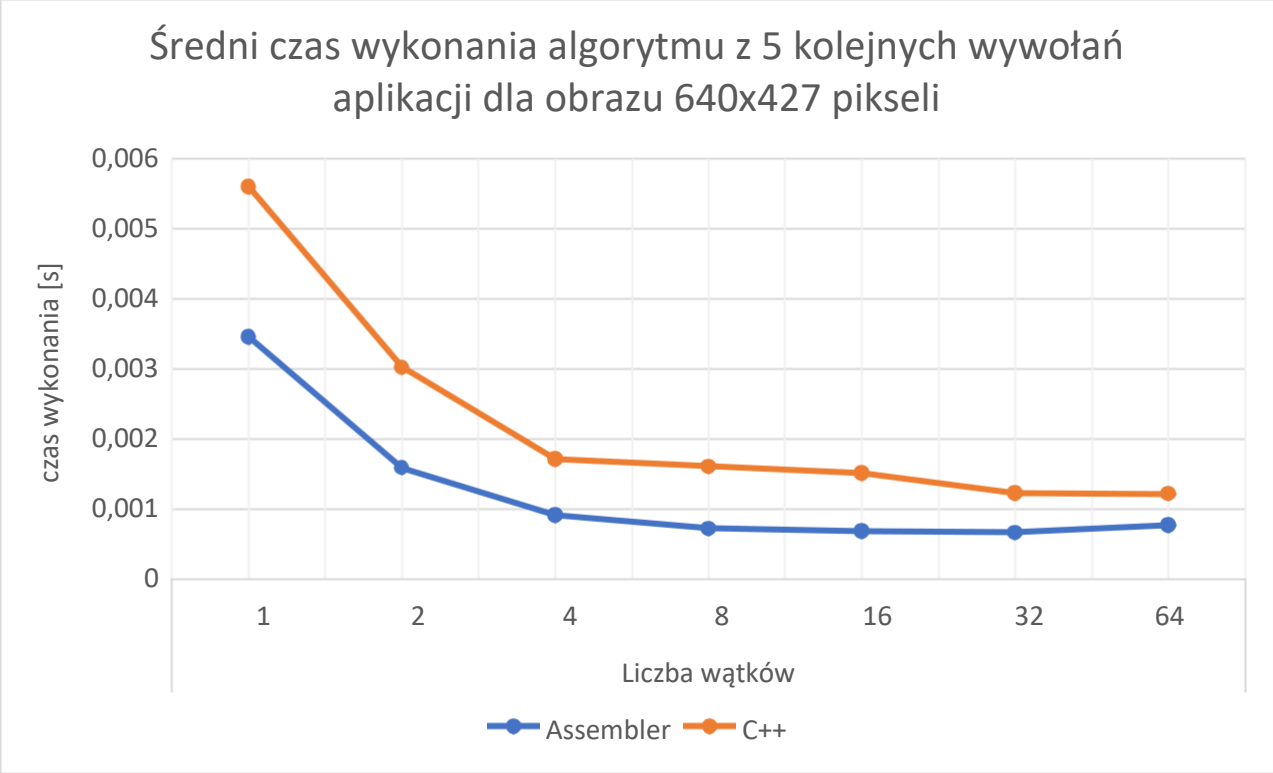
Gdy cały zakres pikseli został przetworzony, algorytm kończy swoje działanie i zwraca sterowanie do miejsca, z którego został wywołany.

3. Raport szybkości działania

Prędkość wykonania algorytmu została zmierzona dla trzech różnych wielkości obrazu oraz dla różnych konfiguracji wątków (1, 2, 4, 8, 16, 32, 64). Została również przeprowadzona analiza porównawcza prędkości wykonania dla implementacji w języku wysokiego poziomu (C++) oraz języku asemblera.

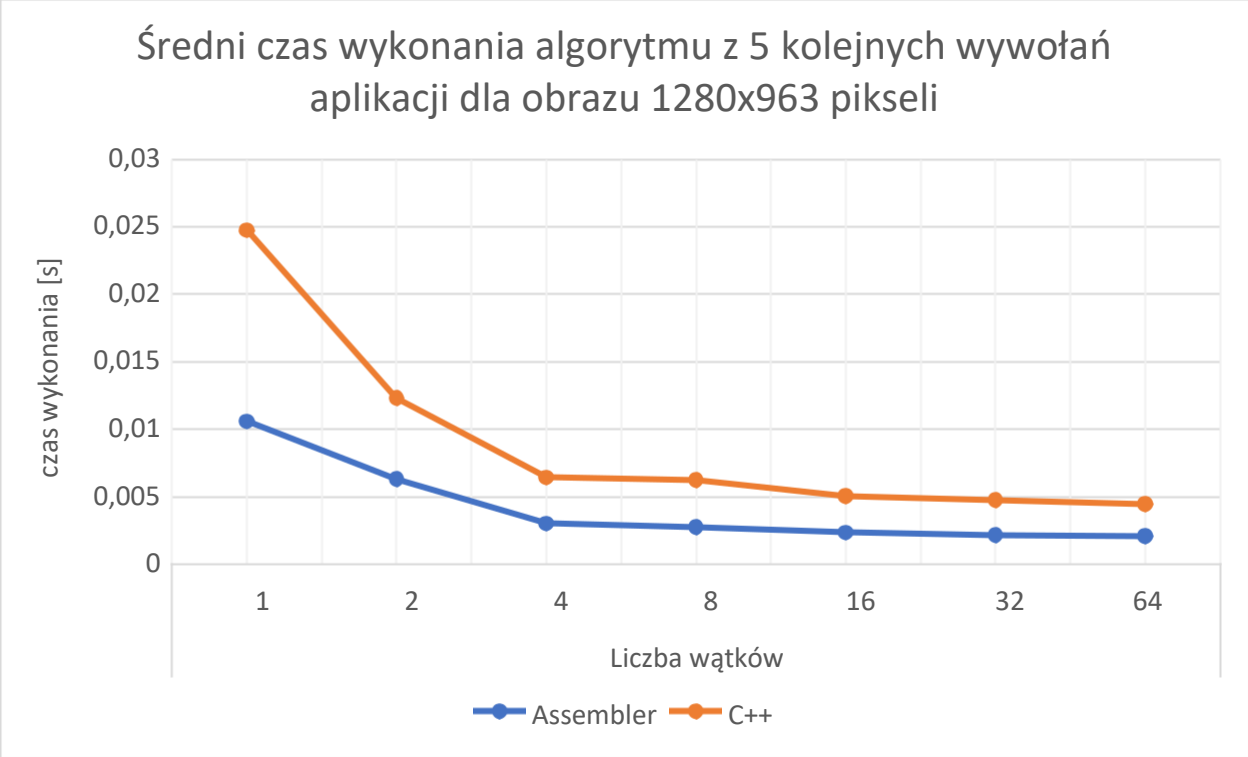
a)

Średni czas wykonania algorytmu z 5 kolejnych wywołań aplikacji dla obrazu 640x427 pikseli							
	Liczba wątków						
	1	2	4	8	16	32	64
Assembler średni czas [s]	0,0034635	0,00159476	0,000924	0,00073122	0,0006936	0,00067138	0,00077808
C++ średni czas [s]	0,0056058	0,0030311	0,001722	0,0016131	0,001523	0,0012328	0,0012248



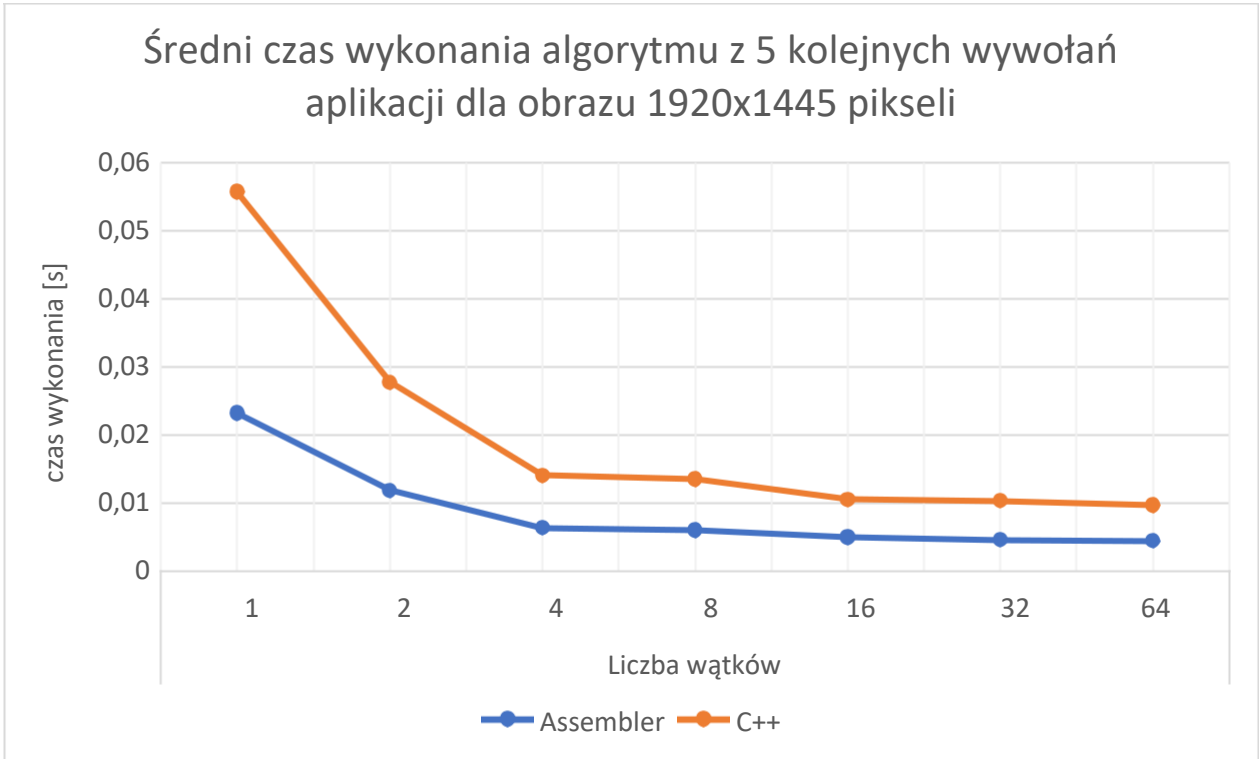
b)

Średni czas wykonania algorytmu z 5 kolejnych wywołań aplikacji dla obrazu 1280x963 pikseli							
	Liczba wątków						
	1	2	4	8	16	32	64
Assembler średni czas [s]	0,0106207	0,0063369	0,0030364	0,0027809	0,0023757	0,0021881	0,0021089
C++ średni czas [s]	0,0247434	0,0123187	0,0064673	0,0062364	0,0050611	0,0047645	0,004457



c)

Średni czas wykonania algorytmu z 5 kolejnych wywołań aplikacji dla obrazu 1920x1445 pikseli							
	Liczba wątków						
	1	2	4	8	16	32	64
Assembler średni czas [s]	0,023283	0,0119207	0,0064018	0,0060739	0,005065	0,0046849	0,0045091
C++ średni czas [s]	0,055749	0,027762	0,0141048	0,0135825	0,0105981	0,0104069	0,0097419



4. Opis uruchamiania

Aplikację można uruchomić poprzez kompilację kodu źródłowego. Po uruchomieniu aplikacji, użytkownik może wybrać obraz do przetworzenia, wybrać bibliotekę C++ lub Assembler oraz ustawić liczbę wątków. Następnie należy kliknąć przycisk "Start" w celu przetworzenia obrazu. Aplikacja wyświetli przetworzony obraz, który można zapisać na dysku.

5. Podsumowanie

Wyniki eksperymentów wykazały, że implementacja algorytmu w języku asemblera była znacznie szybsza niż implementacja w języku C++. Wraz ze wzrostem liczby wątków, czas wykonania algorytmu zmniejszał się, osiągając najlepsze wyniki dla większej liczby wątków. Wyniki te sugerują, że implementacja w języku asemblera jest bardziej efektywna pod względem wydajności, zwłaszcza dla dużych obrazów i dużych ilości wątków. Dodatkowo, realizacja w języku asemblera była bardziej skomplikowana w implementacji i wymagała większej ilości czasu na rozwinięcie.

Biblioteka Asembler

Zalety:

- Wyższa wydajność, szczególnie przy użyciu instrukcji wektorowych.
- Większa kontrola nad sprzętem i zarządzaniem pamięcią.
- Możliwość dokładnej optymalizacji krytycznych sekcji kodu.

Wady:

- Trudniejszy w utrzymaniu i debugowaniu.
- Wyższe ryzyko błędów i trudności w zarządzaniu pamięcią.

Biblioteka C++

Zalety:

- Łatwość debugowania i utrzymania kodu.
- Szeroka dostępność bibliotek i narzędzi pomocniczych.
- Lepsza czytelność i łatwiejsze zarządzanie pamięcią.

Wady:

- Niższa wydajność w porównaniu do optymalizowanego kodu asemblera.