Gliwice, 16.01.2021

Politechnika Śląska

Języki Asemblerowe

Studia stacjonarne 1-go stopnia
Informatyka, semestr V

Projekt pt.
Rozmycie Gaussowskie - C++ vs Asm

Autor:
Emanuel Jureczko
ej306101@student.polsl.pl

Prowadząca:
Magdalena Pawlyta

Wstęp

Celem projektu jest porównanie dwóch identycznych bibliotek, z których jedna jest napisana w języku C++, a druga w Asemblerze. Ich funkcjonalnością jest wykonanie rozmycia Gaussowskiego na przekazanym obrazie.
Jest to filtr rozmywający obraz, który wykorzystuje funkcję Gaussa do obliczania transformacji zastosowanej do każdego piksela.

Istnieją przynajmniej dwa podejścia implementacyjne, z wykorzystaniem jądra dwuwymiarowego o rozmiarze NxN oraz z wykorzystaniem jądra jednowymiarowego o rozmiarze Nx1. Więcej informacji na temat algorytmu w prezentacji. W projekcie zaimplementowano wersję z jednowymiarowym jądrem z racji na potencjalnie szybsze działanie.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

Jednowymiarowy wzór Gaussa

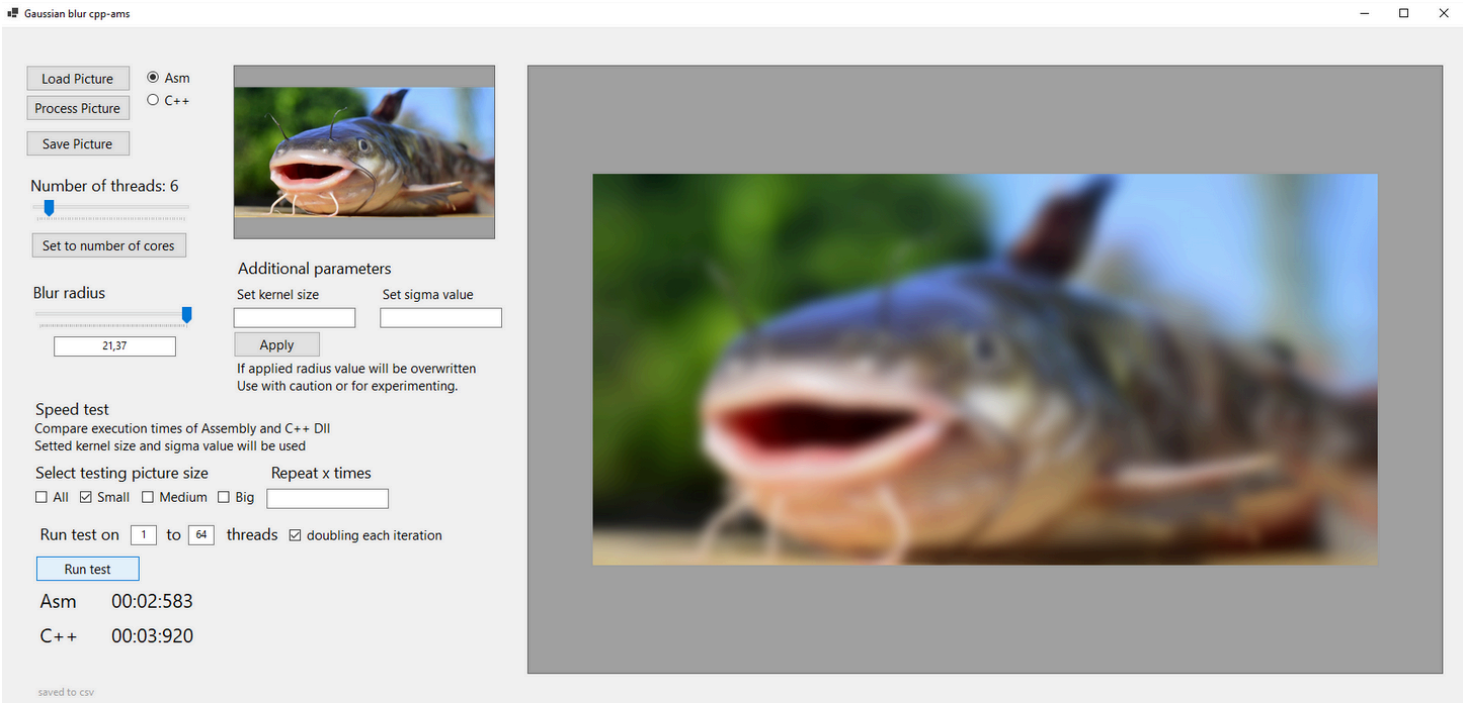
Program

Interfejs

Został napisanym w języku C#. Umożliwia użytkownikowi:

- Wybór graficznego pliku
- Dostosowanie ilości wątków
- Dostosowanie parametrów rozmycia
- Wybór używanej biblioteki
- Podgląd wybranego pliku
- Wykonanie rozmycia
- Podgląd rozmytego obrazu
- Zapis rozmytego obrazu
- Wykonanie pomiar czasu wykonania rozmycia przy użyciu danej biblioteki
- Zapis danych pomiarowych do pliku zewnętrznego

Poza tym interfejs wyświetla informacje o danej akcji wykonywanej przez program w lewym dolnym rogu.



Interfejs aplikacji
użyte zdjęcie: https://rybyswiata.pl/ryby-slodkowodne/sum-smazony-czy-pieczony/

Parametry wejściowe

Funkcje C++ i Asm wywoływane w kodzie C# oczekują następujących parametrów:

Funkcje tworzące jądro (kernel)

- wartość sigma (float)
- rozmiar jądra (Byte)
- wskaźnik na tablicę liczb zmiennoprzecinkowych o potrzebnym rozmiarze, do której zostanie zapisany wynik (float*)

Biblioteki udostępniają osobne funkcje do wykonania rozmycia dla jednego jak i wielu wątków. Co prawda funkcja wielowątkowa może obsłużyć również przypadek jednego wątku jednak postanowiono pozostawić wcześniej utworzoną funkcję pracującą tylko z jednym wątkiem. Jej użycie może okazać się minimalnie szybsze. Z racji na problematykę związaną z wielowątkowością najoptymalniejszym rozwiązaniem było podzielenie wykonywania rozmycia na dwie osobne funkcje.

Jednowątkowa funkcja wykonująca rozmycia

- dane pikseli bitmapy (unsigned char*)
- szerokość bitmapy (int)
- wysokość bitmapy (int)
- stride bitmapy (int)
- rozmiar jądra (int)
- wskaźnik na tablicę w której ma zostać zapisany finalny (rozmyty) obraz (unsigned char*)
- wskaźnik na jądro (float*)

Funkcja wykonująca pierwszy etap rozmycia (poziomo)

- dane pikseli bitmapy (unsigned char*)
- szerokość bitmapy (int)
- wysokość początkowa fragmentu bitmapy (int)
- wysokość końcowa fragmentu bitmapy (int)
- stride bitmapy (int)
- rozmiar jądra (int)
- wskaźnik na tablicę pustych danych pikseli. Potrzebna do zapisania wyniku pierwszego etapu (unsigned char*)
- wskaźnik na jądro (float*)

Funkcja wykonująca drugi etap rozmycia (pionowo)

- wynikowe dane pikseli z pierwszego etapu rozmycia (unsigned char*)
- szerokość bitmapy (int)
- wysokość bitmapy (int)
- wysokość początkowa fragmentu bitmapy (int)
- wysokość końcowa fragmentu bitmapy (int)
- stride bitmapy (int)
- rozmiar jądra (int)
- wskaźnik na tablicę w której ma zostać zapisany finalny (rozmyty) obraz (unsigned char*)
- wskaźnik na jądro (float*)

Fragmentu kodu

Poniższy fragment kodu znajduje się w obu etapach rozmycia wielowątkowego jak i w funkcji jednowątkowej. Wykonujemy w nim mnożenie trzech wartości piksela (RGB) przez aktualną wartość zawartą w kernelu, a następnie dodajemy uzyskaną wartość do rejestru sumy i zaplętamy operację.

```
717     continue:
718         ; get bytes
719         ; xmm3 = kerI kerI kerI kerI
720         vbroadcastss xmm3, dword ptr [rdi + r13 * 4]      ; Load one float value (kernel[i]) and put it in all 4 xmm0 slots
721
722         ; xmm1 = xxxx rImg gImg bImg
723         movzx ebx, byte ptr [rsi + rax]     ; Load and expand byte1 into the 32-bit eax register
724         cvtsi2ss xmm1, ebx                  ; Convert eax (32bit int) to float and put it in xmm1
725
726         movzx ebx, byte ptr [rsi + rax + 1]
727         cvtsi2ss xmm2, ebx
728         insertps xmm1, xmm2, 16             ; Insert float from xmm2 into xmm1 in second slot (offset 0x10)
729
730
731         movzx ebx, byte ptr [rsi + rax + 2]
732         cvtsi2ss xmm2, ebx
733         insertps xmm1, xmm2, 32
734
735         mulps xmm1, xmm3                    ; xmm1 = xmm1 * xmm3 (multiplies all elements in the packets)
736         addps xmm0, xmm1                    ; xmm0 = xmm0 + xmm1
737
738
739         inc r13                         ; i++
740         jmp i_loop                      ; back to I loop
```

fragment kodu funkcji rozmywającej

linia 720 - pobranie aktualnej wartość float z tablicy kernel oraz rozprzestrzenienie jej do każdej sekcji rejestru xmm3
linie 723, 726, 731 - pobranie aktualnej wartość int z tablicy danych o pikselach oraz zapis w rejestrze ebx
linie 724, 727, 732 - konwersja liczby w ebx z typu int na float oraz zapis w pierwszej sekcji rejestru xmm
linie 728, 729 - zapisać liczby będącej w xmm2 do odpowiednio 2-giej i 3-ciej sekcji rejestru xmm1
linia 735 - wymnożenie trzech wartości pixela w xmm1 przez wartość z jądra będącą rozprzestrzenioną w xmm3 jednym rozkazem. Operacja wygląda następująco:

xmm1 = xxx	rImg	gImg	bImg
xmm3 = kerI	kerI	kerI	kerI
xmm1 = xxx	kerI*rImg	kerI*gImg	kerI*bImg

gdzie r/g/bImg to wartości pixela, a kerI aktualną wartością kernela. xxx reprezentuje ignorowaną zawartość w rejestrze

linia 736 - dodanie odpowiednich sekcji rejestrów xmm0 i xmm1 do siebie
linia 739 - inkrementacja licznika pętli

linia 740 - skok do początku pętli

Tym sposobem wykonujemy tylko raz mnożenie i dodawanie kosztem dwóch przeniesień.

Pomiary czasowe

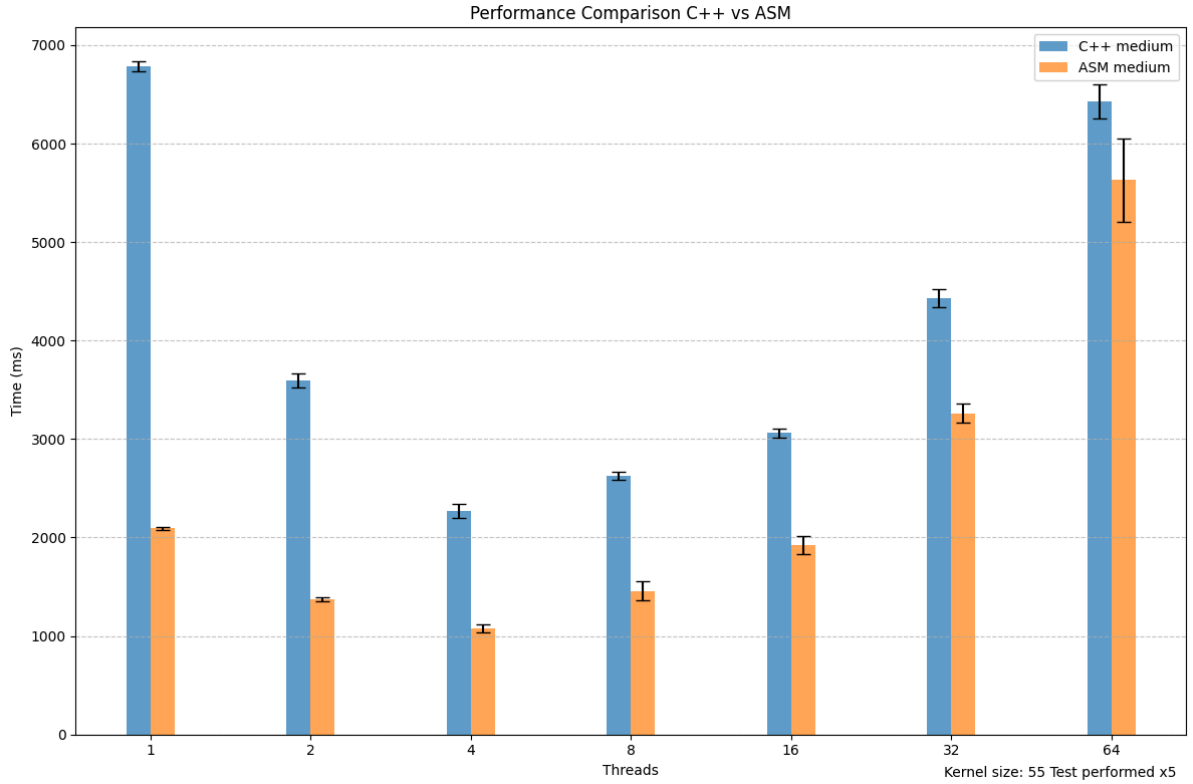
Testy zostały przygotowany według wytycznych:
średnia + odchylenie standardowe z 5 wywołań (z pominięciem pierwszego): 3 rodzaje danych wejściowych, dwie biblioteki, 7 konfiguracji wątków (1,2,4,8,16,32 i 64) różne rodzaje optymalizacji.

Sprzęt na którym pomiary zostały wykonane to AMD FX(tm)-6300 Six-Core o 3 rdzeniach fizycznych, co daje 6 rdzeni logicznych.

Wpierw zastosowano optymalizację czasową - Maximize Speed (/O2)
Rozmiar danych:

- małe 900x450
- średnie 2600x1462
- duże 4624x3472

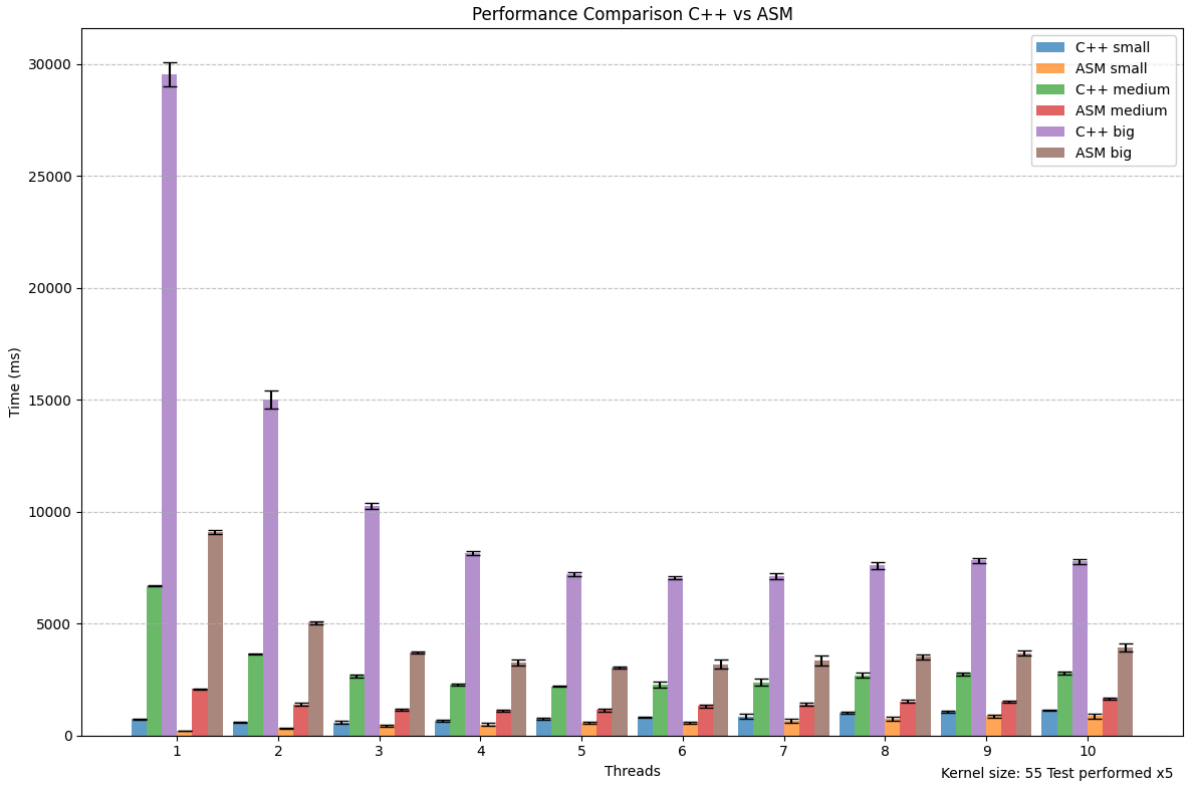
Dane średniej wielkości, 1 - 64 wątki



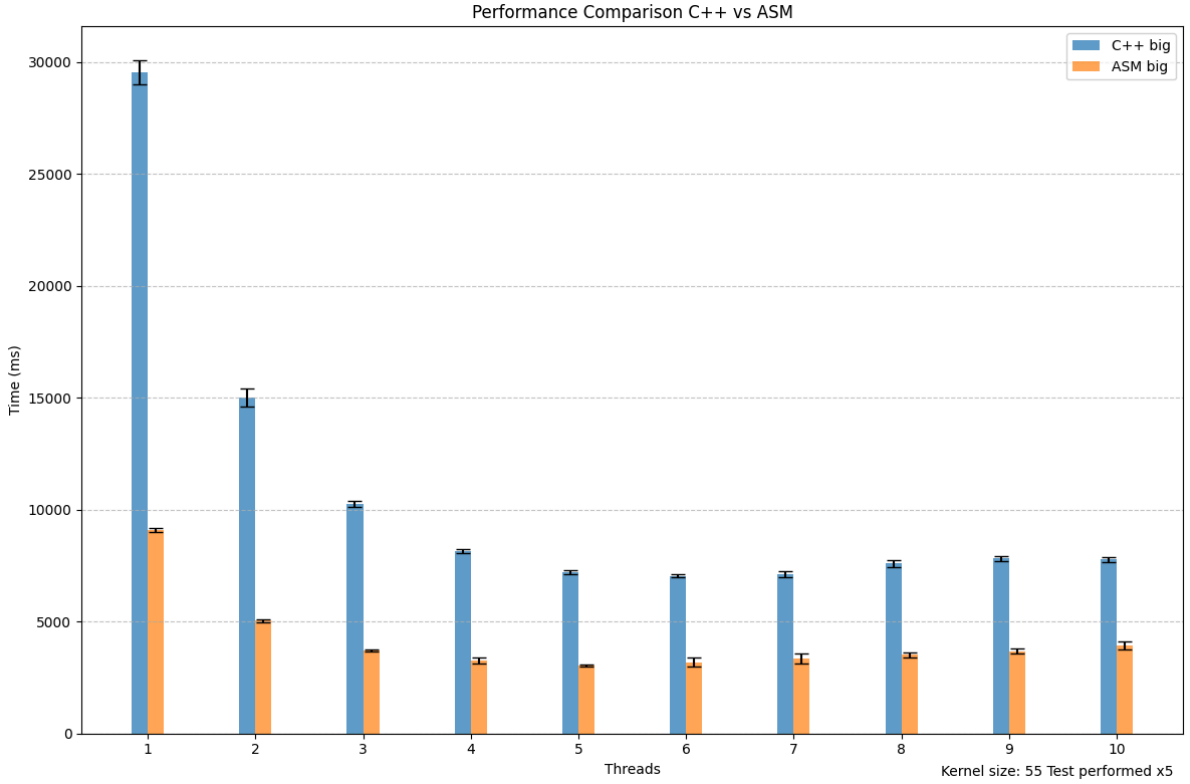
Zauważamy, że w kolejnych testach nie musimy testować większej liczby wątków* ponieważ nie uzyskamy więcej informacji niż obecnie. Zatem zakres wątków został zmieniony do 1-10.

*liczba wątków - wartości ustawiona w programie, a zatem liczba definiująca na ile pod obrazów ma zostać podzielony obraz. Podzielne obrazy są przetwarzane w miarę możliwości równolegle.

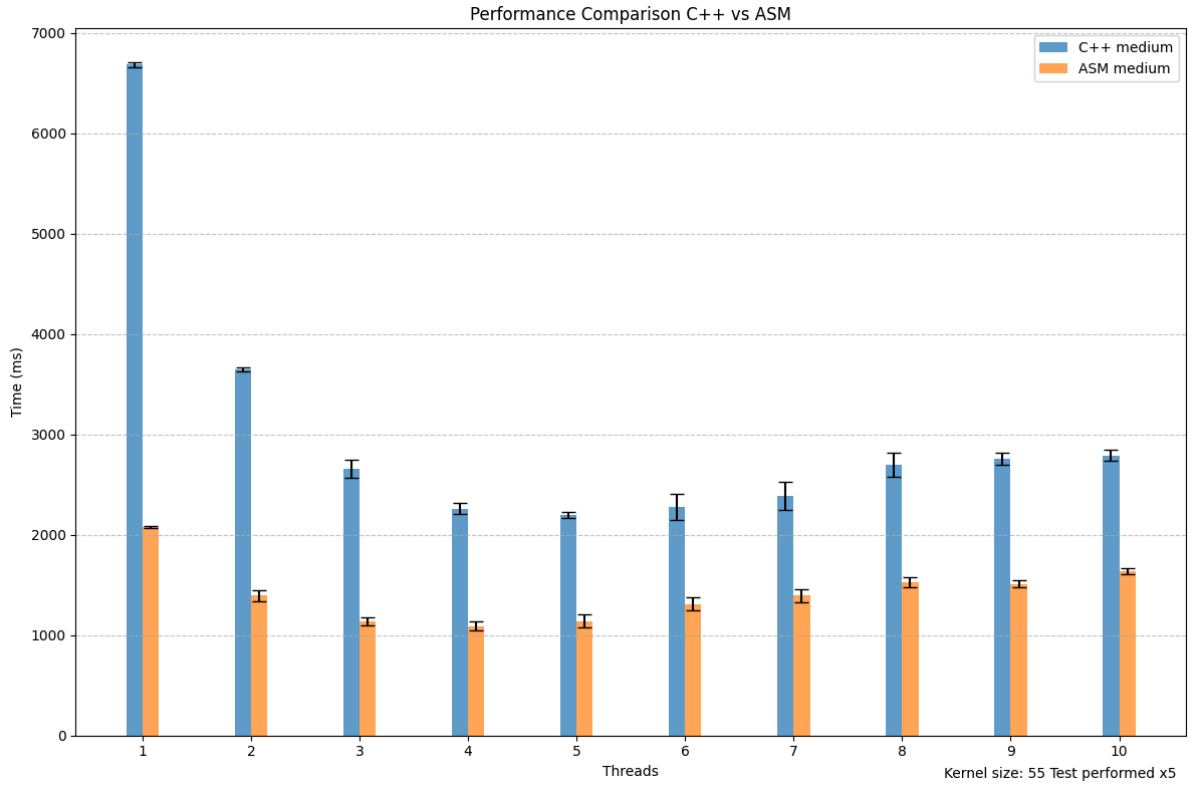
Dane różnej wielkości, 1 - 10 wątki



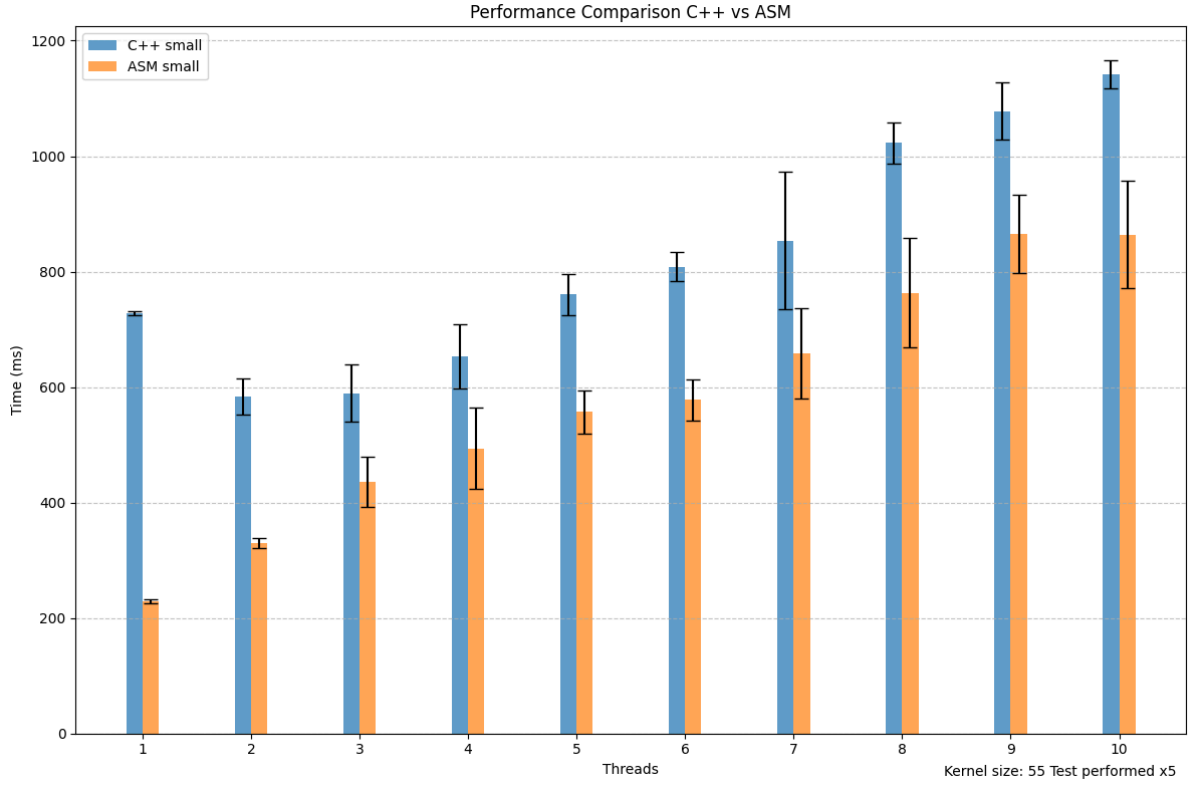
Dane dużej wielkości, 1 - 10 wątki



Dane średniej wielkości, 1 - 10 wątki



Dane małej wielkości, 1 - 10 wątki



Różne optymalizacje

Pomiary na różnych optymalizacjach postanowiono wykonać tylko na liczbie wątków równej rzeczywistej liczbie wątków logicznej oraz tylko na danych o średniej wielkości.
/O2 - Maksymalna optymalizacja (preferuj szybkość)
/O1 - Maksymalna optymalizacja (preferuj rozmiar)
/Od - brak optymalizacji
Niestety wyniki dla trybów:

- /O2 z żadną preferencją
- /O2s z preferencją czasową
- /O2t z preferencją pamięciową
- /O1 z żadną preferencją
- /O1s z preferencją czasową
- /O1t z preferencją pamięciową

były na tyle zbliżone, że bez wiedzy który jest którym nie bylibyśmy w stanie ich rozróżnić.

Zatem przeprowadzono kolejne testy na dużych danych wejściowych. Różnice wydajności były tylko przy braku optymalizacji. Zaś optymalizacja z preferencją rozmiarów pozwoliła zaoszczędzić aż 1KB. Wyniki przedstawiono poniżej

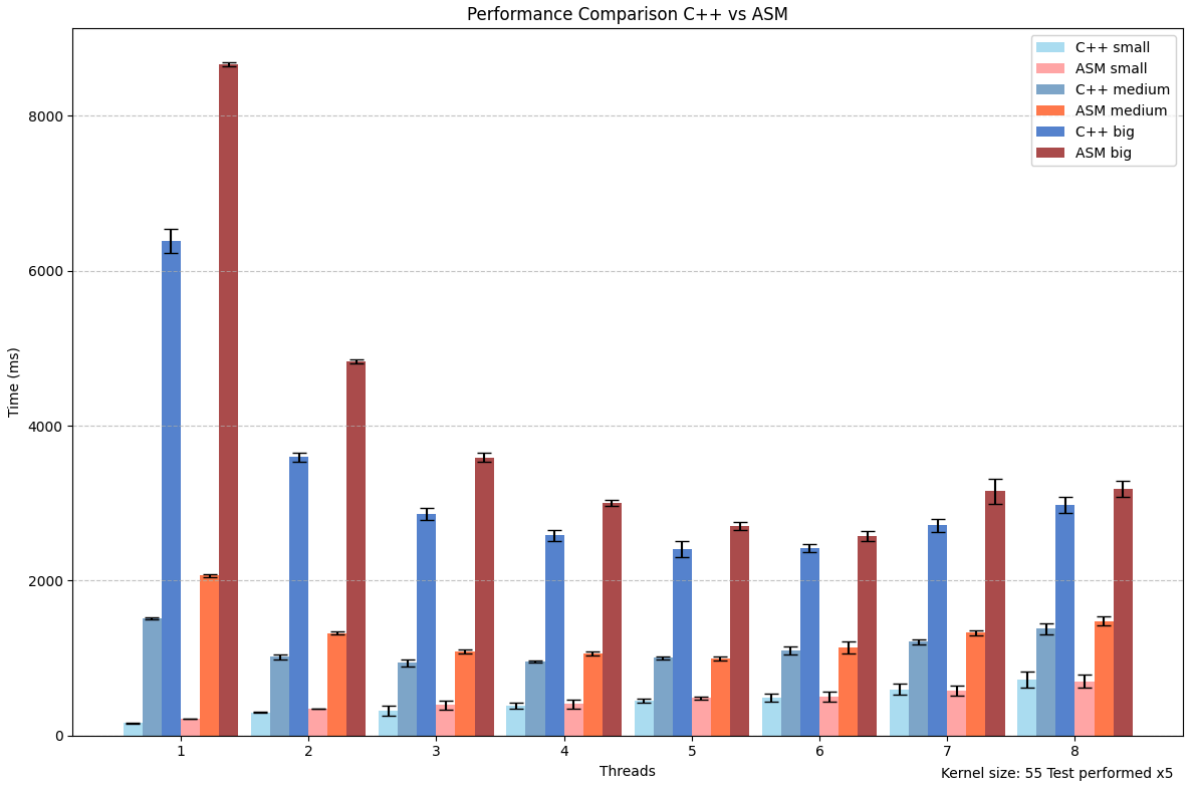
Pomiar i jednostka	/Od	/O1	/O2
Średni czas w ms	6184,5	5794,75	5807
Rozmiar w KB	14	13	14

Nie były to zadowalające wyniki więc postanowiłem trochę poczytać o wcześniej wspomnianej opcji o bardzo przekonującej nazwie “/O2 - Maksymalna optymalizacja (preferuj szybkość)“ wraz z dodatkowo włączoną opcją “preferuje szybki kod /Ot”, która okazała się nie być taka skuteczna jak wskazuje jej wybrzmienie. Dowiedziałem się, że w innych kompilatorach (GCC/Clang) flaga -O3 jest bardziej agresywna w optymalizacjach.
Sprawdziłem więc co więcej robią owe kompilatory, a następnie dowiedziałem się jak ręcznie włączyć dodatkowe opcje w VS dla kompilatora MSVC.
Z racji że mój procesor nie obsługuje AVX2 włączyłem przez zakładkę C/C++ -> Wiersz polecenia opcję “/arch:AVX”, która odpowiada za 256-bitowe operacje na liczbach zmiennoprzecinkowych.
Stety lub niestety okazało się, że włączenie tej opcji sprawiło, że biblioteka napisana w C++ okazała się na równie szybka, a nawet często lekko szybsza niż w Asm. Nowe zestawienie optymalizacji wygląda następująco:

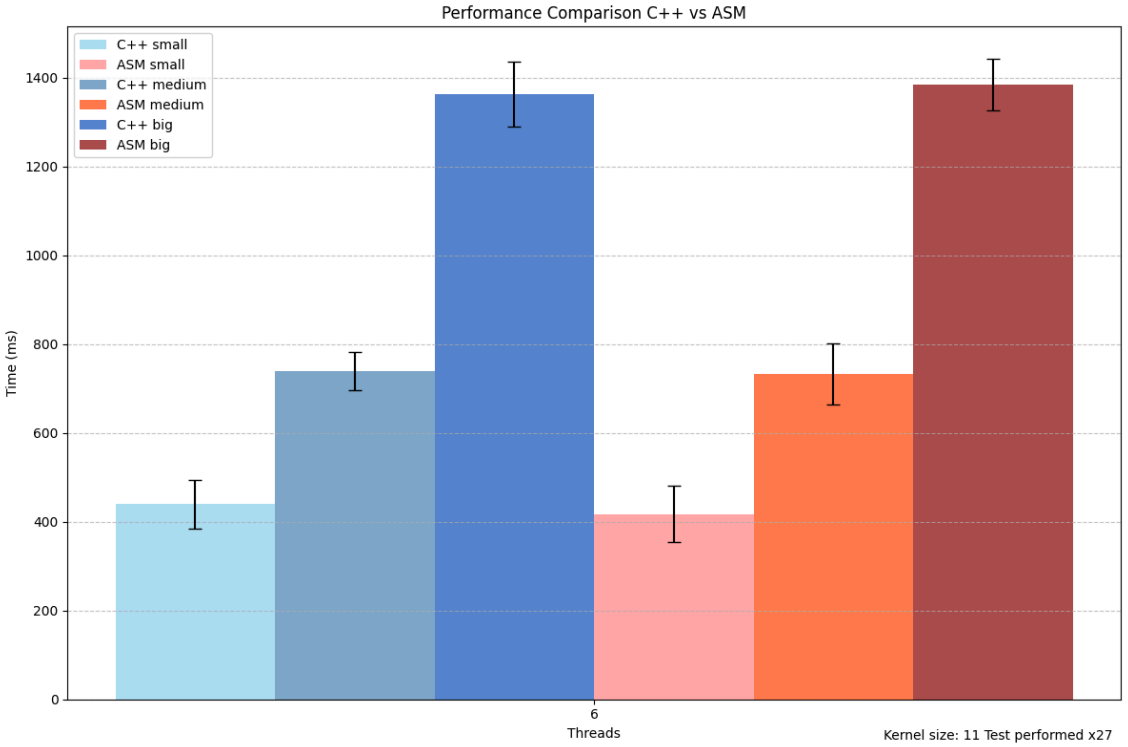
Pomiar i jednostka	/0d	/01	/02	/02 /arch:AVX
Średni czas w ms	6184,5	5794,75	5807	2537

Rozmiar w KB	14	13	14	15

Dane różnej wielkości dane, AVX w C++, 1 - 8 wątki



Dane różnej wielkości dane, AVX w C++, 6 wątków



Size-name	Time-cpp-ms	Time-asm-ms	Threads	Iterations
small	407	373	6	1
small	407	484	6	2
small	497	492	6	3
small	505	396	6	4
small	362	320	6	5
small	454	315	6	6
small	461	340	6	7
small	361	429	6	8
small	486	356	6	9
small	463	412	6	10
small	426	358	6	11
small	452	382	6	12
small	521	436	6	13
small	530	420	6	14
small	391	384	6	15
small	512	481	6	16
small	407	495	6	17
small	345	520	6	18
small	382	392	6	19
small	428	471	6	20
small	487	451	6	21
small	474	314	6	22
small	378	455	6	23
small	363	330	6	24
small	470	470	6	25
small	410	447	6	26
small	446	508	6	27
medium	597	704	6	1
medium	739	740	6	2
medium	697	691	6	3
medium	695	779	6	4
medium	734	733	6	5
medium	763	681	6	6
medium	723	860	6	7
medium	712	691	6	8
medium	751	638	6	9
medium	775	747	6	10
medium	802	695	6	11

medium	709	726	6	12
medium	772	806	6	13
medium	732	721	6	14
medium	723	788	6	15
medium	735	783	6	16
medium	713	618	6	17
medium	688	614	6	18
medium	740	771	6	19
medium	702	696	6	20
medium	833	767	6	21
medium	780	597	6	22
medium	839	826	6	23
medium	685	811	6	24
medium	775	757	6	25
medium	712	694	6	26
medium	696	809	6	27
big	1323	1413	6	1
big	1368	1311	6	2
big	1435	1361	6	3
big	1265	1509	6	4
big	1338	1360	6	5
big	1483	1428	6	6
big	1243	1492	6	7
big	1362	1373	6	8
big	1450	1381	6	9
big	1282	1373	6	10
big	1270	1431	6	11
big	1422	1396	6	12
big	1371	1423	6	13
big	1360	1367	6	14
big	1293	1300	6	15
big	1368	1397	6	16
big	1381	1420	6	17
big	1534	1364	6	18
big	1408	1484	6	19
big	1302	1343	6	20
big	1360	1309	6	21
big	1254	1376	6	22
big	1409	1305	6	23
big	1398	1366	6	24
big	1285	1398	6	25
big	1386	1433	6	26

big	1410	1282	6	27

Wnioski

Zależności pomiędzy ilością wątków, a rozmiarem danych nie powinno nikogo zaskoczyć. Krótko podsumowując:

- podział > liczba wątków logicznych = tracimy na wydajności poprzez niepotrzebne poświęcony czas na operację związane z utworzeniem wątku
- podział małych danych na jeszcze mniejsze = możliwą utratą wydajności przy większych ilościach wątków.
- teoretyczne optimum = liczbie wątków logicznych

Podczas pisania sprawozdania miesiąc po ukończonej pracy nad projektem wpadły mi w oko miejsca na potencjalne poprawki na których zyskałyby obie biblioteki, ale wydaje się, że assembler bardziej. Zostanie to niestety zakwalifikowane jako rozrywkowe prace w wolnym czasie, a nie dalsza część tego projektu.

Pisanie w asemblerze na pewno pomogło w głębszym zrozumieniu co dzieje się na niższym poziomie abstrakcji, jak i pomogło lepiej zrozumieć działanie procesora. Jednak jak pokazały ostatnie wyniki testów nie jest to najefektywniejsze. Czas poświęcony na tworzenie kodu asemblerowego był może i nawet o rząd większy niż czas na kod w C++.

Niespodziewanie powiększyłem również swoją wiedzę o kompilatorach jak i o złudnych nazwach opcji typu “Maksymalna optymalizacja (preferuj szybkość)”, która, jak się okazuje, wcale nie skutkował najszybszym kodem.