https://github.com/krzsztfwtk

https://github.com/krzsztfwtk/polsl-ja-proj

Raport z projektu z Języków Asemblerowych

„Obliczanie wartości wielomianu za pomocą algorytmu Hornera”

krzsztfwtk

polsl-ja-proj

Gliwice 2025

Spis treści

https://github.com/krzsztfwtk/polsl-ja-proj

1 Wstęp

Wielomiany są podstawowymi funkcjami w matematyce i informatyce, wykorzystywanymi w różnych dziedzinach, takich jak analiza numeryczna, przetwarzanie sygnałów czy grafika komputerowa. Obliczanie wartości wielomianu dla danego argumentu jest kluczowe w wielu algorytmach i aplikacjach. Celem zaimplementowanego algorytmu było głównie narysowanie wykresu zadanego wielomianu.

1.1 Definicja wielomianu

Obliczanie wartości wielomianu polega na wyznaczeniu wyniku funkcji wielomianowej $W(x)$ dla danej wartości x. Wielomian stopnia n można zapisać poniższym wzorem.

$$W(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

gdzie a_i to współczynniki wielomianu.

1.2 Metoda Hornera

Metoda Hornera przekształca wielomian do postaci zagnieżdżonej, aby zmniejszyć liczbę operacji mnożenia. Poniższy wzór przedstawia postać zagnieżdżoną wielomianu.

$$W(x) = a_0 + x\left(a_1 + x\left(a_2 + x(\ldots + x(a_{n-1} + xa_n))\right)\right)$$

Poniżej znajduje się **Lst. 1 Pseudokod podstawowej wersji algorytmu Hornera** bez instrukcji wektorowych oraz bez wielowątkowości. Pseudokod bardziej zaawansowanych wersji algorytmu Hornera (wykorzystujących AVX lub wielowątkowość) został przedstawiony w prezentacji projektu.

```
procedure Horner(a, x, n)
begin
    wynik := a[n];
    for i := n - 1 downto 0 do
    begin
        wynik := wynik * x + a[i];
    end;
    return wynik;
end;
```

Lst. 1 Pseudokod podstawowej wersji algorytmu Hornera

Złożoność czasowa algorytmu Hornera w zależności od ilości współczynników jest liniowa przy założeniu, że operacją dominującą jest mnożenie.

$$T = \sum_{i=0}^{n} 1 = n \in O(n)$$

2 Struktura i funkcjonalność rozwiązania

Program został zaprojektowany z myślą uwzględniając różnorodne scenariusze zastosowań, w celu przeprowadzenia eksperymentów. Głównym celem było stworzenie narzędzia, które pozwala na efektywne przetwarzanie danych wejściowych i generowanie wyników, jednocześnie oferując użytkownikowi możliwość konfiguracji kluczowych elementów działania. Dzięki zastosowaniu wielowątkowości czy instrukcji AVX, program może być testowany przy różnych ustawieniach.

2.1 Parametry wejściowe

Program umożliwia dostosowanie wielu parametrów wejściowych, które wpływają zarówno na sposób działania algorytmu oraz na sposób przetwarzania wyników.

- **Punkty** – zbiór punktów wejściowych, w których zostanie obliczona wartość wielomianu. Mogą być wygenerowane przez program za pomocą interfejsu graficznego lub wprowadzone ręcznie. Aby nie zajmować dużo miejsca w pamięci masowej plik z punktami jest podawany w formacie: *<minimum> <maksimum> <krok>*. Właściwe wartości punktów są generowane w trakcie działania programu i przechowywane w pamięci RAM. Na przykład jeżeli plik opisujący punkty będzie miał zawartość: *-10.0 10.0 0.01*, to wygenerowany zbiór punktów będzie zawierał wszystkie punkty większe lub równe -10, mniejsze lub równe 10 oraz będące wielokrotnością 0.01.
- **Współczynniki** – wektor współczynników wielomianu, który zostanie obliczony.
- **Opcja zapisu wyników** – umożliwia włączenie lub wyłączenie zapisu wyników. Przy dużych danych, czas potrzebny na zapisanie wyników może przekraczać czas obliczeń, dlatego zapis jest opcjonalny.
- **Plik wynikowy** – ścieżka do pliku wyjściowego, w którym zostaną zapisane obliczone wartości wielomianu.
- **Opcja rysowania wykresu** – możliwość wygenerowania i wyświetlenia wykresu wizualizującego dane lub wyniki. Przy dużych danych, czas potrzebny na narysowanie wykresu może przekraczać czas obliczeń, dlatego rysowanie wykresu jest opcjonalne.
- **Wykorzystanie AVX** – parametr decydujący o włączeniu lub wyłączeniu implementacji wykorzystującej instrukcje AVX, co może znacząco przyspieszyć obliczenia.
- **Wykorzystanie wielowątkowości** – możliwość włączenia trybu wielowątkowego, co pozwala na równoległe przetwarzanie danych w celu poprawy wydajności.
- **Implementacja** – wybór między różnymi wersjami implementacji w różnych językach programowania. Dostępna jest implementacja języku C++ oraz w asemblerze. Program przy włączeniu wyświetla okno, w którym należy wskazać odpowiednie biblioteki DLL, zakłada on, że użytkownik poproszony o wskazanie biblioteki napisanej w języku C++ wskaże tę właśnie bibliotekę. Jeżeli użytkownik przekornie wskaże inną bibliotekę, która również implementuje odpowiednie funkcje do algorytmu Hornera, to program będzie zachowywał się tak, jakby ta biblioteka była biblioteką napisaną w C++, ponieważ nie ma możliwości sprawdzenia w jakiej technologii została napisana biblioteka ładowana dynamicznie. Analogiczna sytuacja może wystąpić z biblioteką ASM.
- **Liczba wątków** – Parametr określający liczbę wątków procesora, które wykorzysta program, jeżeli został włączony tryb wielowątkowy.

https://github.com/krzsztfwtk/polsl-ja-proj

2.2 Kod źródłowy ASM DLL

Poniżej przedstawione są wybrane fragmenty kodu zostały opatrzone komentarzami, które wyjaśniają funkcjonalność poszczególnych instrukcji oraz ich wpływ na ogólną wydajność programu.

2.2.1 Podstawowa procedura do obliczenia wartość wielomianu

Procedura **HornerPolynomial** implementuje algorytm Hornera do obliczania wartości, wykorzystując rejestry zmiennoprzecinkowe i iterację wsteczną po współczynnikach. Wynik obliczeń jest zwracany w rejestrze XMM0, zgodnie z konwencją wywoływania funkcji w systemie Windows x64.

```
HornerPolynomial proc
    ; Argumenty:
    ; RCX = wskaźnik do x (float)
    ; RDX = wskaźnik do tablicy współczynników a[] (float[])
    ; R8  = liczba współczynników n (int)

    ; Zarezerwuj miejsce na stosie
    sub rsp, 32 ; Zarezerwuj miejsce na zmienne lokalne

    ; Załaduj x do xmm1
    movss xmm1, dword ptr [rcx] ; xmm1 = x
    dec r8  ; r8 = n - 1 (indeks ostatniego elementu)

    ; wynik zapisany w xmm0, ponieważ w konwencji wywoływania
    ; Windows x64, wartość zwracana funkcji musi być umieszczona
    ; w rejestrze RAX dla wartości całkowitych
    ; lub w XMM0 dla wartości zmiennoprzecinkowych.
    movss xmm0, dword ptr [rdx + r8*4] ; w = a[r8]

    ; Iteracja od przedostatniego a[] do początku
    dec r8                              ; r8 = n - 2
StartLoop:
    cmp r8, 0           ; Czy r8 >= 0?
    jl EndLoop          ; Jeśli nie, wyjdź z pętli

    ; Pobierz a[r8] do xmm2
    movss xmm2, dword ptr [rdx + r8*4]  ; xmm2 = a[r8]
    mulss xmm0, xmm1                    ; w = w * x
    addss xmm0, xmm2                    ; w = w + a[r8]
    dec r8                              ; r8--

    jmp StartLoop                       ; Powrót do pętli

EndLoop:
    add rsp, 32
    ret
HornerPolynomial endp
```

Lst. 2 Procedura do obliczenia wartości wielomianu wykorzystująca wartości skalarne

https://github.com/krzsztfwtk/polsl-ja-proj

2.2.2 Procedura do obliczenia wektora wartości wielomianu

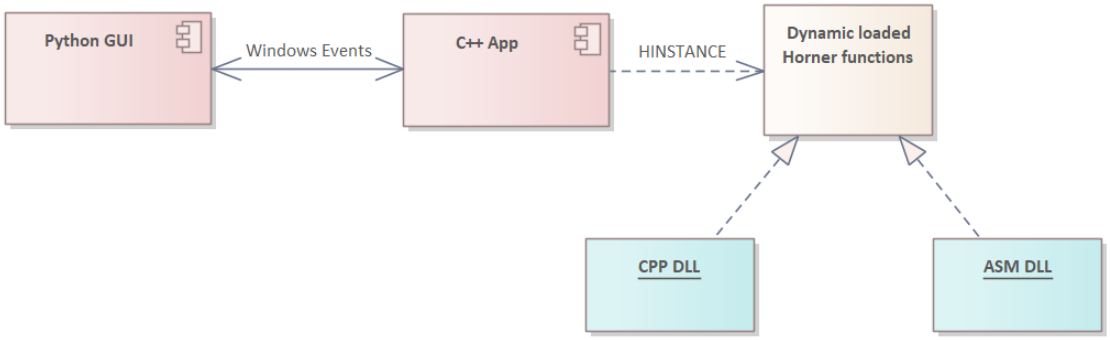
Procedura **HornerPolynomialAvx** wykorzystuje instrukcje AVX do równoległego obliczania wartości wielomianu dla ośmiu argumentów jednocześnie. Wynik jest zwracany w rejestrze YMM0 jako wektor zawierający osiem wartości zmiennoprzecinkowych 32-bitowych (float).

```
HornerPolynomialAvx proc
    ; Argumenty:
    ; RCX = wskaźnik do tablicy x (float[8])
    ; RDX = wskaźnik do tablicy współczynników a[] (float[])
    ; R8  = liczba współczynników n (int)

    ; Zarezerwuj miejsce na stosie
    sub rsp, 32

    ; Załaduj tablicę x do ymm0 (8 elementów float)
    vmovups ymm1, ymmword ptr [rcx] ; ymm1 = x[0..7]
    dec r8                          ; r8 = n - 1 (indeks ostatniego)

    ; Zainicjalizuj wynik jako ostatni współczynnik (8 floatów)
    vbroadcastss ymm2, dword ptr [rdx + r8*4] ; ymm2 = a[r8]
    vmovups ymm0, ymm2        ; ymm1 = a[r8]

    ; Iteracja od przedostatniego a[] do początku
    dec r8                              ; r8 = n - 2

StartLoopAvx:
    cmp r8, 0                           ; Czy r8 >= 0?
    jl EndLoopAvx                       ; Jeśli nie, wyjdź z pętli

    ; Pobierz współczynnik a[r8] do ymm2
    vbroadcastss ymm2, dword ptr [rdx + r8*4] ; ymm2 = a[r8]

    ; Wykonaj obliczenia Hornera: w = w * x + a[r8]
    vmulps ymm0, ymm0, ymm1         ; w = w * x
    vaddps ymm0, ymm0, ymm2         ; w = w + a[r8])

    ; Zmniejsz indeks
    dec r8                          ; r8--
    jmp StartLoopAvx                ; Powrót do pętli

EndLoopAvx:
    ; Przywróć stos
    add rsp, 32
    ret
HornerPolynomialAvx endp
```

Lst. 3 Procedura do obliczenia wartości wielomianu wykorzystująca wektory

2.3 Struktura programu

Aplikacja składa się z interfejsu graficznego napisane w języku programowania Python oraz modułu obliczeniowego napisanego w C++. Interfejs umożliwia użytkownikowi wprowadzanie parametrów, po czym zapisuje ustawienia do pliku `config.ini`. Następnie sygnalizuje modułowi C++ gotowość do przetwarzania za pomocą zdarzenia systemowego `Global\ComputeEvent`. Moduł C++ nasłuchuje na to zdarzenie, przeprowadza obliczenia na podstawie odczytanych danych z plików podanych w konfiguracji, a wyniki zapisuje w pliku wyjściowym również podanym w konfiguracji. Kluczowe funkcje do obliczeń są zaimplementowane w bibliotekach DLL. Są dwie osobne biblioteki, jedna została napisana w asemblerze, a druga w C++. Obie biblioteki implementują te same funkcje. Pozwala to na wiarygodne porównanie ich wydajności. Taka architektura eliminuje problemy wielowątkowości związane z ograniczeniami interpretera Pythona. Poniższy diagram pseudo-UML prezentuje zależności różnych komponentów oprogramowania wytworzonego na potrzeby projektu.

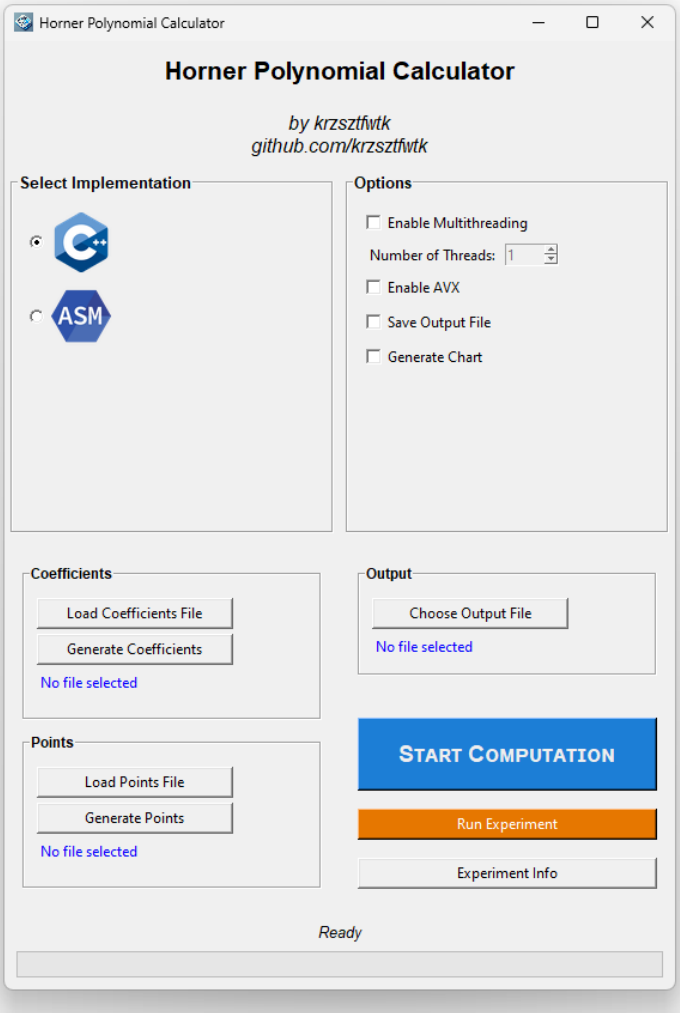


Rys. 1 Diagram pseudo-UML prezentuje relacje między komponentami oprogramowania.

2.4 Interfejs użytkownika

Interfejs graficzny aplikacji umożliwia użytkownikowi łatwe zarządzanie procesem obliczania wartości wielomianu. Główne okno aplikacji składa się z kilku sekcji funkcjonalnych, które odpowiadają za wprowadzanie danych, wybór parametrów oraz monitorowanie obliczeń.

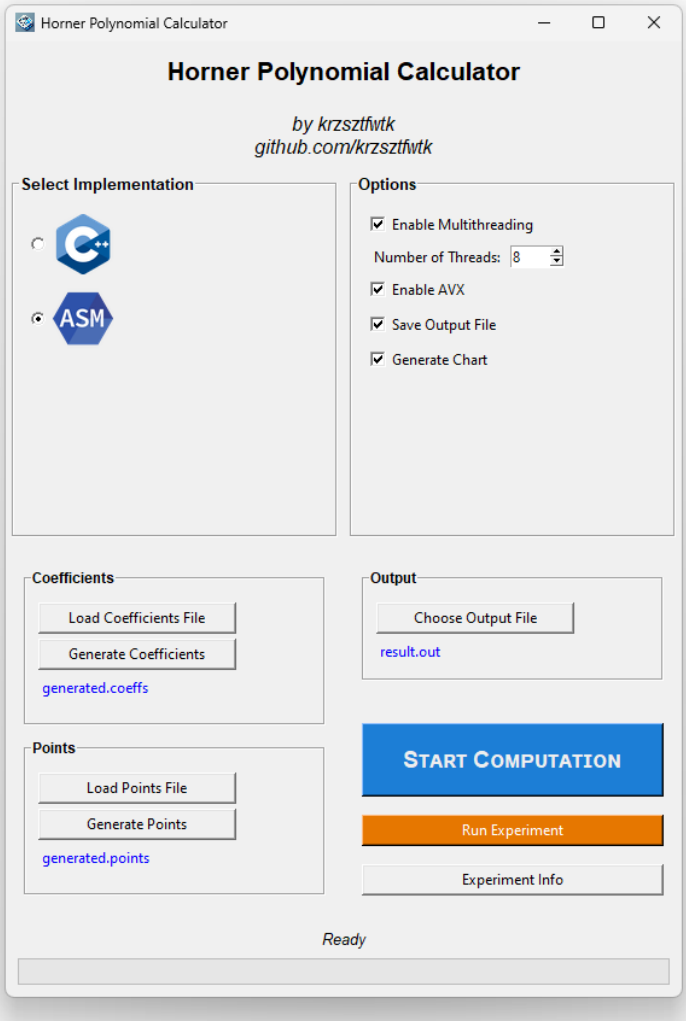
https://github.com/krzsztfwtk/polsl-ja-proj



Rys. 2 Zrzut ekranu wyboru ustawień interfejsu

W górnej części interfejsu znajdują się przyciski umożliwiające załadowanie plików z danymi lub automatyczne wygenerowanie współczynników i punktów. Wybrane pliki są wyświetlane obok przycisków, co pozwala na szybką orientację w bieżących ustawieniach. **Rys. 2 Zrzut ekranu wyboru ustawień interfejsu** prezentuje wygląd tej sekcji przed wprowadzeniem danych.

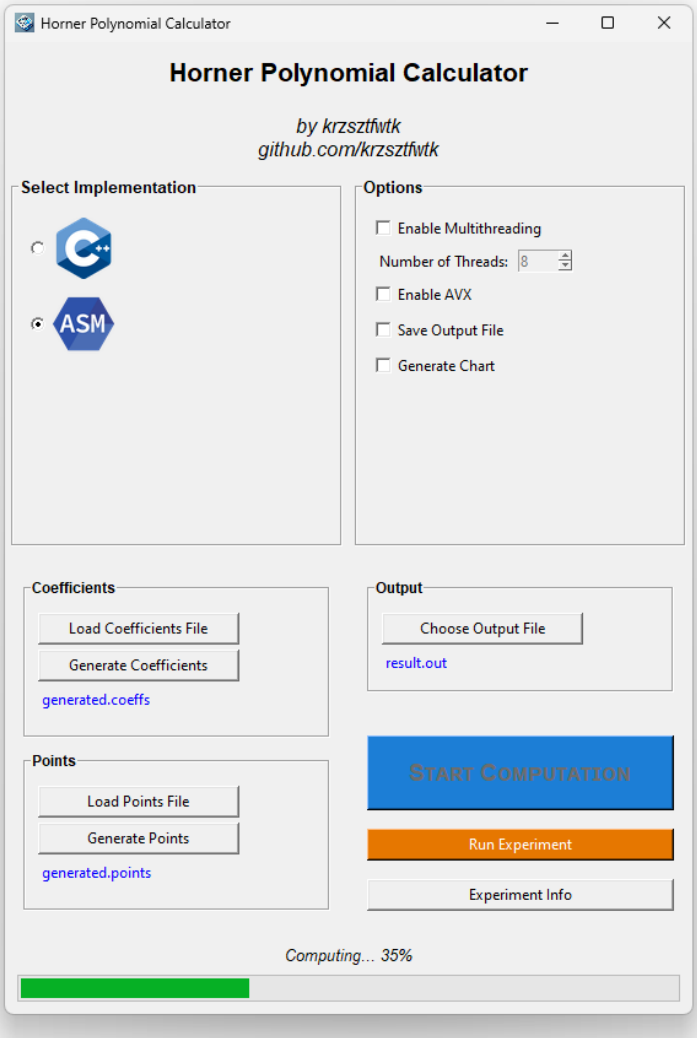
https://github.com/krzsztfwtk/polsl-ja-proj



Rys. 3 Zrzut ekranu interfejsu przed rozpoczęciem obliczeń

W centralnej części interfejsu użytkownik ma możliwość wyboru implementacji obliczeń (C++ lub asembler), konfiguracji wielowątkowości, instrukcji AVX oraz generowania wykresu wyników. Dzięki temu aplikacja pozwala eksperymentować z różnymi ustawieniami wpływającymi na wydajność i prezentację danych. **Rys. 3 Zrzut ekranu interfejsu przed rozpoczęciem obliczeń** ilustruje przykładową konfigurację przed uruchomieniem.

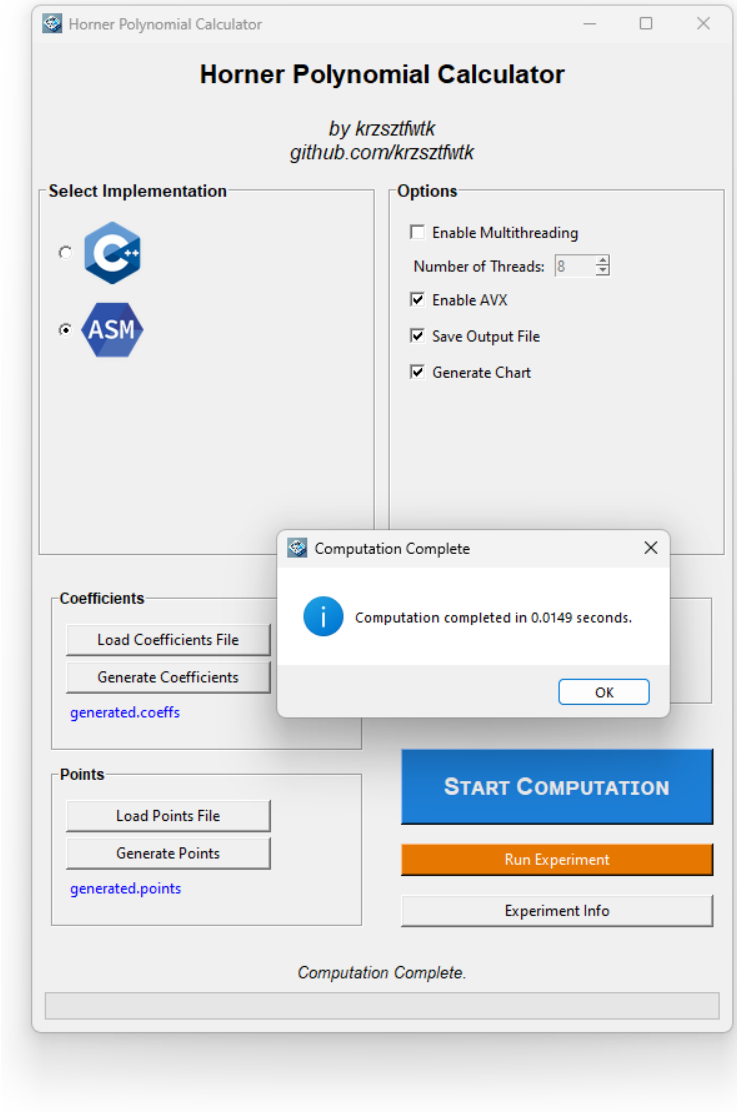
https://github.com/krzsztfwtk/polsl-ja-proj



Rys. 4 Zrzut ekranu interfejsu z paskiem postępu podczas obliczeń

W dolnej części interfejsu znajduje się przycisk *Start Computation*, który rozpoczyna proces obliczeń zgodnie z wybranymi parametrami. Podczas wykonywania obliczeń użytkownik widzi pasek postępu oraz komunikaty informujące o statusie operacji, co widać na **Rys. 4 Zrzut ekranu interfejsu z paskiem postępu podczas obliczeń**.

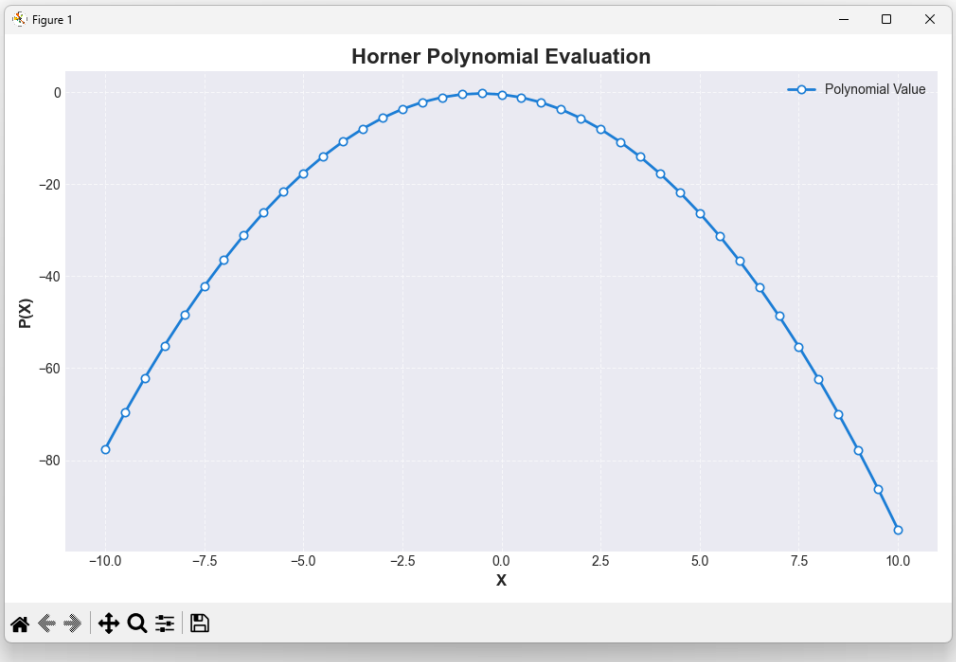
https://github.com/krzsztfwtk/polsl-ja-proj



Rys. 5 Zrzut ekranu interfejsu po wykonaniu obliczeń

Po zakończeniu wyświetlany jest czas wykonania oraz, jeśli wybrano taką opcję, wykres wyników. Na **Rys. 5 Zrzut ekranu interfejsu po wykonaniu obliczeń** widoczny jest komunikat po zakończeniu obliczeń.

https://github.com/krzsztfwtk/polsl-ja-proj



Rys. 6 Zrzut ekranu wykresu z wynikami

Jeżeli zostało zaznaczone pole wyboru *Generate Chart* zostanie wyświetlony wykres zadanego wielomianu w zadanej dziedzinie. Wykres widoczny jest na **Rys. 6 Zrzut ekranu wykresu z wynikami**.

Aplikacja umożliwia ponowne wykonywanie obliczeń z różnymi parametrami bez potrzeby ponownego uruchamiania, co pozwala na efektywne eksperymentowanie z ustawieniami.

https://github.com/krzsztfwtk/polsl-ja-proj

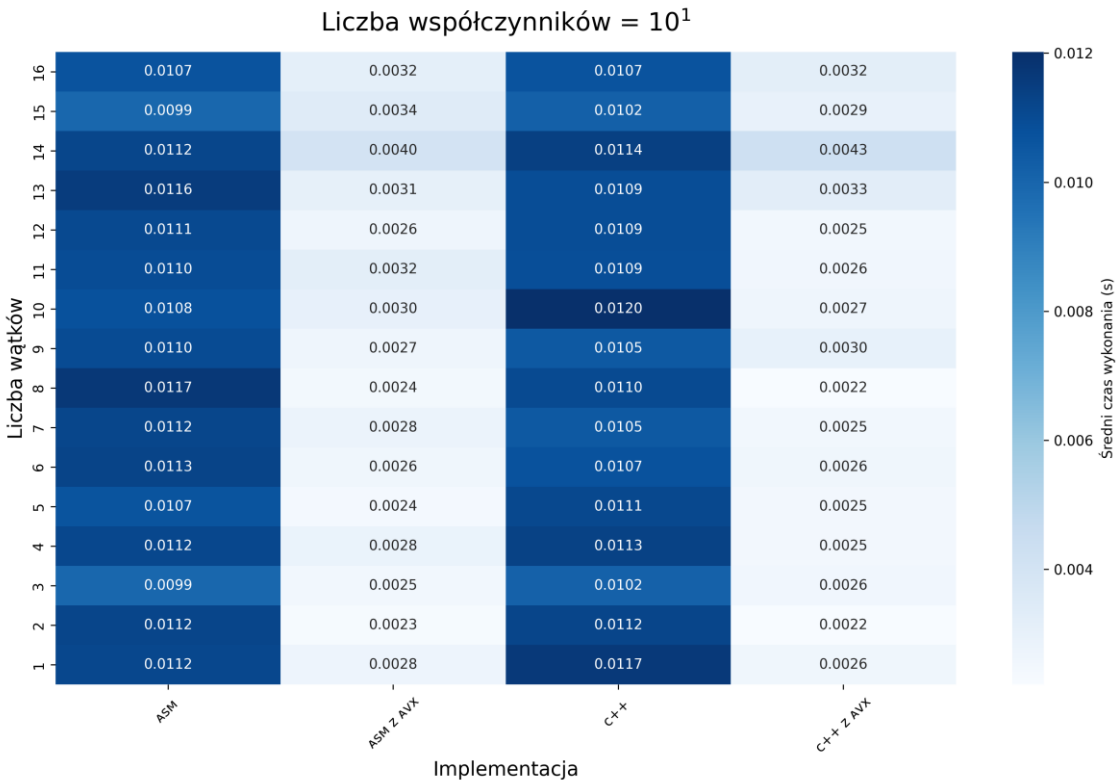
3 Pomiary wydajności

Przeprowadzony eksperyment porównał wydajność implementacji w języku C++ oraz w asemblerze. Ze względu na dużą liczbę wyników częściowych, w raporcie zamieszczono jedynie najistotniejsze porównania i statystyki. Szczegółowe dane pomiarowe mogą zostać odtworzone za pomocą przygotowanego eksperymentu, który można przeprowadzić z poziomu GUI.

Eksperyment przeprowadzono na systemie operacyjnym *Microsoft Windows 11 Pro* w wersji *10.0.22631 Build 22631*. Wykorzystano procesor *11th Gen Intel(R) Core(TM) i7-11700K* o taktowaniu 3.60 GHz i 16 rdzeniach. System był wyposażony w 15.4 GiB pamięci RAM, z czego 14.6 GiB było dostępne podczas eksperymentu. Przy tej konfiguracji oczekiwane przyspieszenie dzięki wykorzystaniu instrukcji AVX wynosi maksymalnie 8 razy. Natomiast przyspieszenie możliwe do osiągnięcia poprzez wielowątkowość wynosi maksymalnie 16 razy. Program oraz biblioteka napisane w C++ zostały skompilowane na poziomie optymalizacji **/O2**, co oznacza maksymalną optymalizację pod względem czasu.

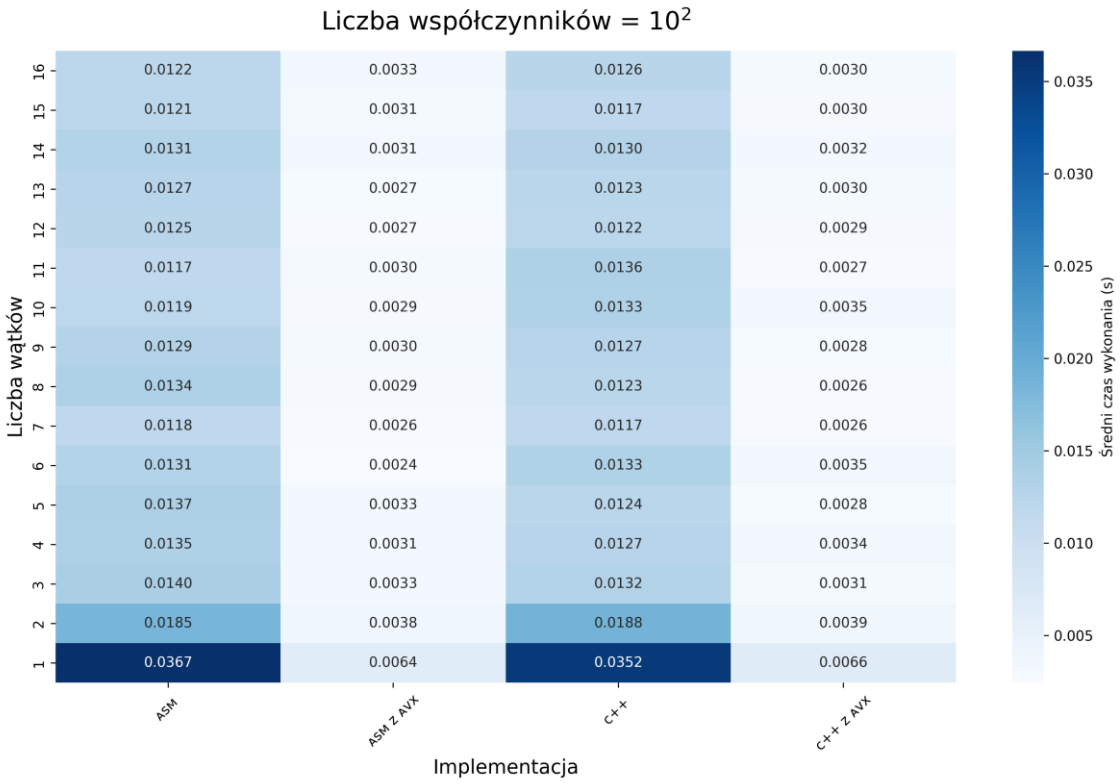
3.1 Średni czas obliczeń na mapach cieplnych

Poniższe mapy cieplne przedstawiają średni czas wykonania obliczeń dla różnych implementacji algorytmu, liczby wątków oraz konfiguracji z użyciem i bez użycia AVX. Dla każdej konfiguracji zostało wykonanych 5 niezależnych pomiarów. Dane te pozwalają na łatwą wizualizację wydajności oraz identyfikację optymalnych ustawień dla różnych rozmiarów współczynników wielomianu.

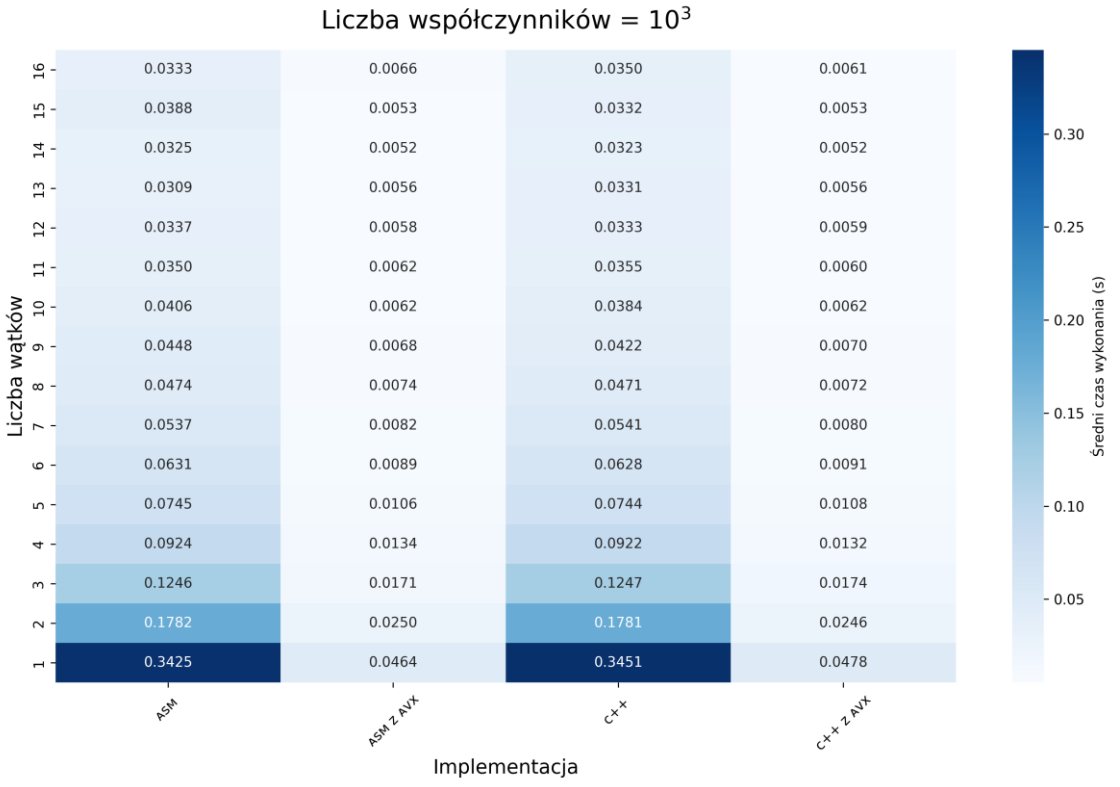


Rys. 7 Średni czas obliczenia wartości wielomianu o 10 współczynnikach w 200 000 punktach

https://github.com/krzsztfwtk/polsl-ja-proj

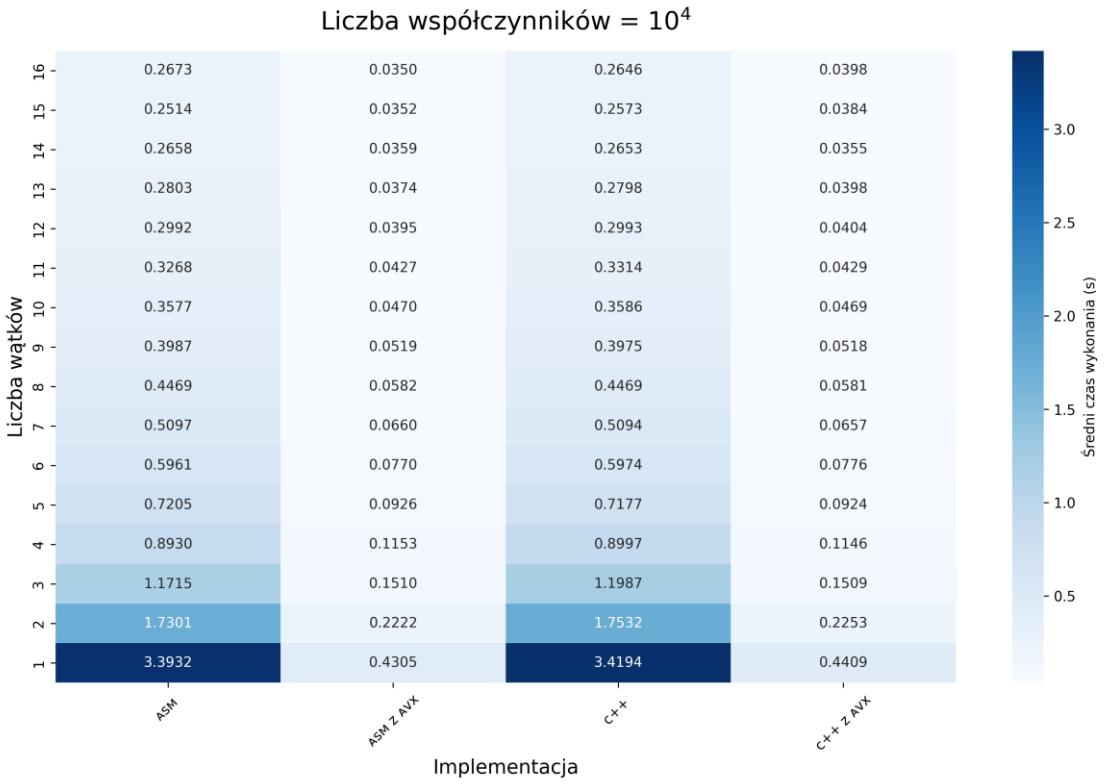


Rys. 8 Średni czas obliczenia wartości wielomianu o 100 współczynnikach w 200 000 punktach

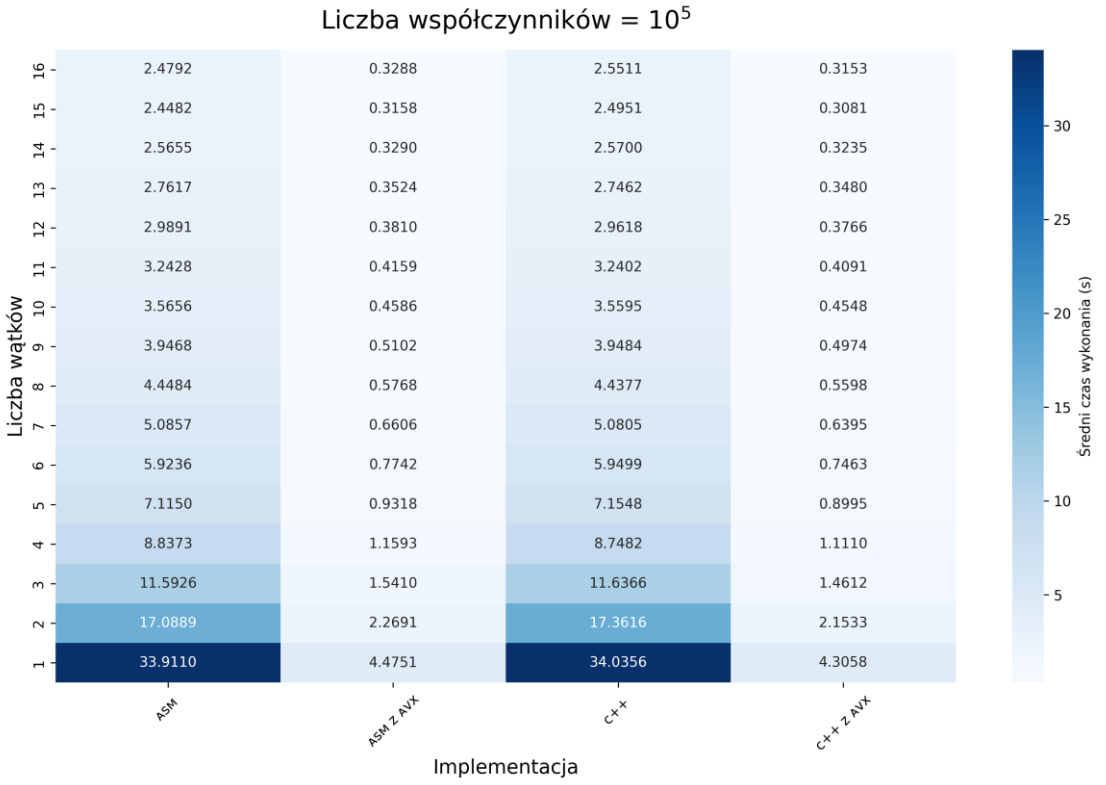


Rys. 9 Średni czas obliczenia wartości wielomianu o 1 000 współczynnikach w 200 000 punktach

https://github.com/krzsztfwtk/polsl-ja-proj



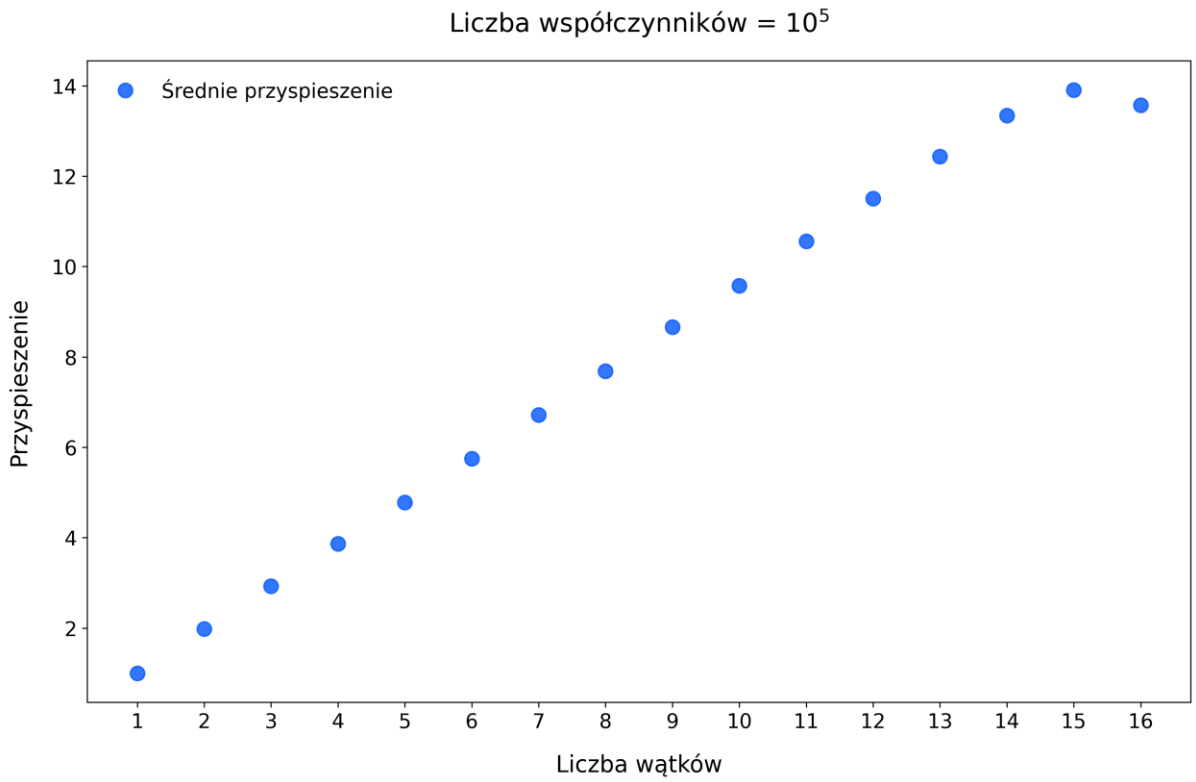
Rys. 10 Średni czas obliczenia wartości wielomianu o 10 000 współczynnikach w 200 000 punktach



Rys. 11 Średni czas obliczenia wartości wielomianu o 100 000 współczynnikach w 200 000 punktach

Na powyższych wykresach można zauważyć, że:

- Zależność czasu obliczeń od ilości współczynników wielomianu jest liniowa, co jest zgodne z analizą algorytmu przedstawioną sekcji **Metoda Hornera**.
- Przy jednakowych konfiguracjach wydajność implementacji w ASM jest w przybliżeniu równa wydajności implementacji w C++.
- Zastosowanie AVX przyspiesza algorytm około 8 razy. Średnie przyspieszenie obliczone ze wszystkich pomiarów wynosi **6.06** razy. Im więcej współczynników ma wielomian, tym większe jest przyspieszenie dzięki AVX, dla liczby współczynników równej 10^5 wynosi ono **7.82** razy. Przy większej ilości współczynników proporcjonalnie mniej czasu program poświęca na operacje, których nie da się przyspieszyć AVX, a więcej na właściwe obliczenia.
- Podobna zależność do AVX występuje przy wielowątkowości – im więcej współczynników, tym większe jest przyspieszenie spowodowane wielowątkowością. Zależność średniego przyspieszenia od liczby wątków jest bliska liniowej od 1 do 15 wątków. Przy 16 wątkach zaobserwowano spadek wydajności w stosunku do 15 wątków. Prawdopodobnie wynika to narzutu wywołanego obsługą systemu *Microsoft Windows 11*. Program nie mógł wykorzystać w pełni 16 wątków, ponieważ część zasobów procesora była wykorzystana na działanie usług systemowych oraz innych programów działających w tle. Ustawienie większej liczby wątków niż liczba dostępnych rdzeni procesora zmniejsza wydajność, ponieważ zwiększa to narzut związany z zarządzaniem wątkami, a obliczenia nie są w rzeczywistości realizowane równolegle. Widać to dobrze na **Rys. 12 Wykres zależności przyspieszenia od liczby wątków**.



Rys. 12 Wykres zależności przyspieszenia od liczby wątków

Poniższa tabela prezentuje przyspieszenia uzyskane przy obliczaniu wielomianu o liczbie współczynników równej 10^5 dla różnych implementacji oraz konfiguracji względem bazowego przypadku, którym jest wykonanie programu z implementacją C++ bez AVX na jednym wątku (założono, że przy konfiguracji bazowej przyspieszenie wynosi 1).

https://github.com/krzsztfwtk/polsl-ja-proj

Tab. 1 Porównanie przyspieszeń dla wybranych konfiguracji implementacji i liczby wątków

implementacja	liczba wątków	wykorzystanie AVX	średni czas, s	Przyspieszenie
C++	1	nie	34.04	1.000
ASM	1	nie	33.91	1.004
C++	1	tak	4.306	7.905
ASM	1	tak	4.475	7.606
C++	15	tak	0.308	110.5
ASM	15	tak	0.316	107.8

Dzięki zastosowaniu instrukcji AVX oraz wielowątkowości przy 15 wątkach udało się przyspieszyć działanie programu około 110 razy w porównaniu do bazowej konfiguracji. Maksymalne teoretyczne przyspieszenie wynosi $15 \cdot 8 = 120$, co oznacza, że osiągnięto około 92% potencjału. Autor programu ocenia ten wynik jako bardzo dobry. Warto jednak zauważyć, że programista, który nie posiada wiedzy o AVX oraz wielowątkowości, nie byłby w stanie osiągnąć takich wyników, ponieważ kompilator automatycznie nie optymalizuje kodu w tym zakresie. To podkreśla znaczenie umiejętności niskopoziomowej optymalizacji w zwiększaniu wydajności aplikacji.

4 Instrukcja użycia programu

Program został zaprojektowany z myślą o łatwym procesie kompilacji i obsługi, dzięki czemu użytkownicy mogą szybko skonfigurować środowisko oraz rozpocząć korzystanie z aplikacji lub wykonać testy.

4.1 Uruchomienie

Program był rozwijany oraz uruchamiany w środowisku systemu operacyjnego *Microsoft Windows*. Wymaga on poprawnej konfiguracji oraz skompilowania komponentów. Proces uruchamiania składa się z kilku kroków, opisanych poniżej.

A. Kompilacja komponentów – Należy załadować solucję w programie Visual Studio, a następnie skompilować ją w trybie *Release* za pomocą skrótu klawiszowego **[Ctrl + Shift + B]**. Po poprawnej kompilacji pliki `App.exe`, `CppDll.dll` oraz `AsmDll.dll` zostaną wygenerowane w katalogu `.\x64\Release\`.

B. Uruchomienie programu – Aplikację można uruchomić przez skrót klawiszowy **[Ctrl + Shift + F5]** w solucji Visual Studio, wtedy zostanie uruchomiony serwer aplikacji oraz interfejs graficzny. Alternatywnie można uruchomić serwer aplikacji i interfejs graficzny oddzielnie zgodnie z puntami **C** i **D**.

C. Uruchomienie serwera aplikacji – Należy uruchomić plik `App.exe` znajdujący się w katalogu `.\x64\Release\`. Warto utworzyć link symboliczny do tego pliku. Program poprosi o wskazanie plików `CppDll.dll` oraz `AsmDll.dll` za pomocą okien dialogowych (*MessageBox*) i eksploratora plików (*OPENFILENAMEW*). Trzeba wskazać odpowiednie pliki DLL (lub nieodpowiednie[1]). Jeśli pliki zostaną poprawnie załadowane, aplikacja uruchomi się w trybie serwera i wyświetli w konsoli komunikat: „Waiting for computation requests...”. Jeżeli załadowanie się nie powiedzie, zostanie wyświetlone okno dialogowe z błędem.

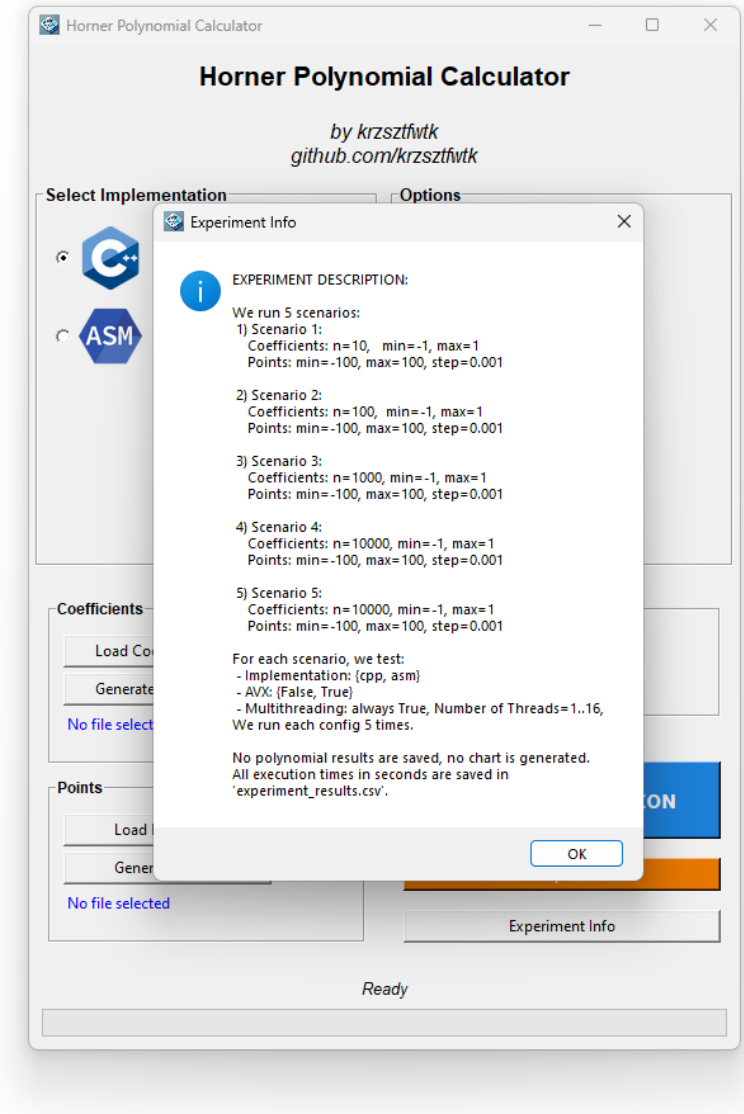
D. Uruchomienie interfejsu graficznego – Należy udać się do katalogu `.\Python\` i uruchomić interfejs za pomocą polecenia *`python main.py`*. Interfejs graficzny zostanie wyświetlony i będzie gotowy do użycia. Szczegółowe informacje dotyczące obsługi interfejsu znajdują się w sekcji **Interfejs użytkownika**.

4.2 Przeprowadzenie eksperymentu

Program został zaprojektowany w taki sposób, aby umożliwić każdemu użytkownikowi przeprowadzenie eksperymentu testującego wydajność implementacji bez konieczności posiadania zaawansowanej wiedzy technicznej. Intuicyjny interfejs graficzny pozwala na łatwe uruchamianie eksperymentu. Program umożliwia również wyświetlenie informacji na temat metodologii wbudowanego eksperymentu.

[1] Jeżeli po poproszeniu przez aplikację o CPP DLL zostanie wskazany plik `AsmDll.dll` oraz po poproszeniu o ASM DLL zostanie wskazany plik `CppDll.dll`, to program będzie działał i zwracał poprawne wyniki. Jednak po takiej konfiguracji przy ustawieniu implementacji ASM w rzeczywistości zostaną użyte funkcje z implementacji C++ i odwrotnie.

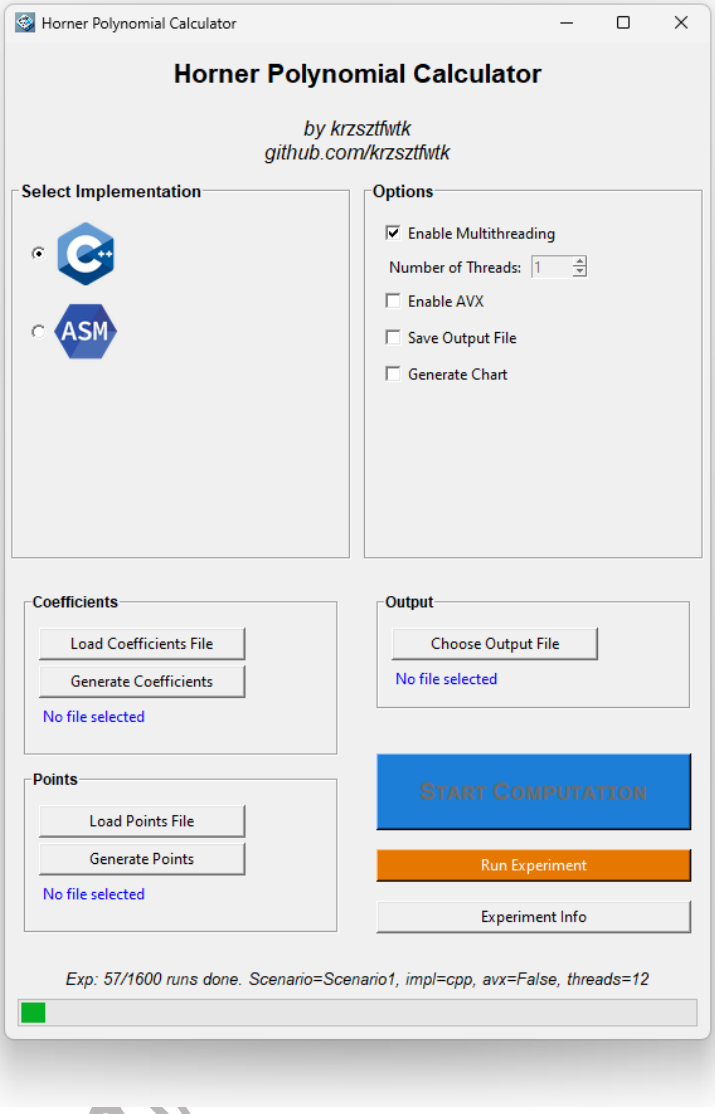
https://github.com/krzsztfwtk/polsl-ja-proj



Rys. 13 Zrzut ekranu okna z informacjami o eksperymencie

Okno które przedstawia **Rys. 13 Zrzut ekranu okna z informacjami o eksperymencie** można otworzyć klikając w przycisk *Experiment Info*.

https://github.com/krzsztfwtk/polsl-ja-proj



Rys. 14 Zrzut ekranu interfejsu użytkownika podczas przeprowadzania eksperymentu

Użytkownik może na bieżąco śledzić postęp eksperymentu, co widać na **Rys. 14 Zrzut ekranu interfejsu użytkownika podczas przeprowadzania eksperymentu**. Wyniki napisane są w pliku `experiment_results.csv`. Poniżej znajduje się przykładowa zawartość tego pliku.

```
scenario,implementation,avx,threads,run_number,time_seconds
Scenario1,cpp,False,1,1,0.0796567
Scenario1,cpp,False,1,2,0.0785439
Scenario1,cpp,False,1,3,0.0804012
Scenario1,cpp,False,1,4,0.075971
Scenario1,cpp,False,1,5,0.078664
Scenario1,cpp,False,2,1,0.0708728
...
```

Lst. 4 Początek pliku zawierającego wyniki eksperymentu

5 Wnioski

Aplikacja, dzięki swojej strukturze i intuicyjnemu interfejsowi, umożliwiła łatwe przeprowadzenie eksperymentów i analizę wyników w różnych konfiguracjach. Projekt pozwolił na analizę wydajności algorytmu Hornera zaimplementowanego w bibliotekach ładowanych dynamicznie napisanych w C++ oraz ASM. Eksperyment wykazał, że wielowątkowość i AVX znacząco przyspieszają obliczenia, szczególnie dla dużych wielomianów.

Przyspieszenie dzięki wykorzystaniu AVX zgodnie z oczekiwaniami było bliskie 8 razy, ponieważ w rejestrach AVX2 można wykonywać obliczenia na 8 wartościach typu float (4 bajty). Natomiast maksymalnie przyspieszenie wynikające z wielowątkowości wystąpiło dla 15 wątków i wyniosło około 14 razy. Różnica między ilością wątków a przyspieszeniem wynika z narzutu związanego z zarządzaniem wątkami. Wprowadzenie dodatkowego 16-go wątku zmniejszyło wydajność, ponieważ część zasobów procesora była zużywana na inne programy, co oznaczało, że 16-ty wątek nie mógł być w pełni wykorzystany przez program i wprowadził dodatkowy narzut.

Porównanie implementacji w C++ (optymalizacja **/O2**) i asemblerze pokazało, że ich wydajność jest zbliżona. Pokazuje to, że kod biblioteki ASM nie był napisany *bardzo źle*, ponieważ gdyby był napisany *bardzo źle* to implementacja ASM działałaby istotnie wolniej niż implementacja C++ przy maksymalnej optymalizacji.

Maksymalnie przyspieszenie wyniosło około 110 razy dzięki AVX i wielowątkowości. Wymagało niskopoziomowej optymalizacji, której kompilator sam nie zapewnia. Programista bez wiedzy w tym zakresie nie byłby w stanie uzyskać takich wyników, co podkreśla znaczenie umiejętności optymalizacji niskopoziomowej.

https://github.com/krzsztfwtk/polsl-ja-proj

6 Bibliografia

„Horner's Method.” Wikipedia. Dostęp online: 2024-12-20.
https://en.wikipedia.org/wiki/Horner%27s_method

7 Spis ilustracji

8 Spis tabel