
	Instytut Informatyki Politechniki Śląskiej Zespół Mikroinformatyki i Teorii Automatów Cyfrowych			
Rok akademicki:	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot (Języki Asemblerowe/SMiW):	Grupa	Seksja
2018/2019	SSI	Języki Asemblerowe	5	1
Imię:	Adam	Prowadzący: AO/JP/KT/GD/BSz/GB	AO	
Nazwisko:	Kincel			
<h2><i>Raport końcowy</i></h2>				
Temat projektu: <p style="text-align: center;">Odejmowanie macierzy o liczbach typu zmiennoprzecinkowego.</p>				
Data oddania: dd/mm/rrrr		10/12/2018		

1. Założenia projektu

Program składa się z dwóch części: programu głównego (C++) oraz dwóch bibliotek "DLL": jedna w języku wysokopoziomym (C++) natomiast druga w języku niskopoziomym (ASM). W części głównej następuje odczyt plików wejściowych, sprawdzanie liczby rdzeni procesora (chyba, że użytkownik podał liczbę wątków w linii poleceń), podział danych na wątki, uruchamianie procedur w języku C++ oraz asemblerowym, liczenie czasu owych procedur, zapis danych do plików wyjściowych oraz ich porównywanie.

W bibliotece C++ następuje odejmowanie dwóch macierzy wejściowych.

W bibliotece ASM również następuje odejmowanie dwóch macierzy wejściowych jednakże z wykorzystaniem instrukcji wektorowych.

2. Analiza zadania

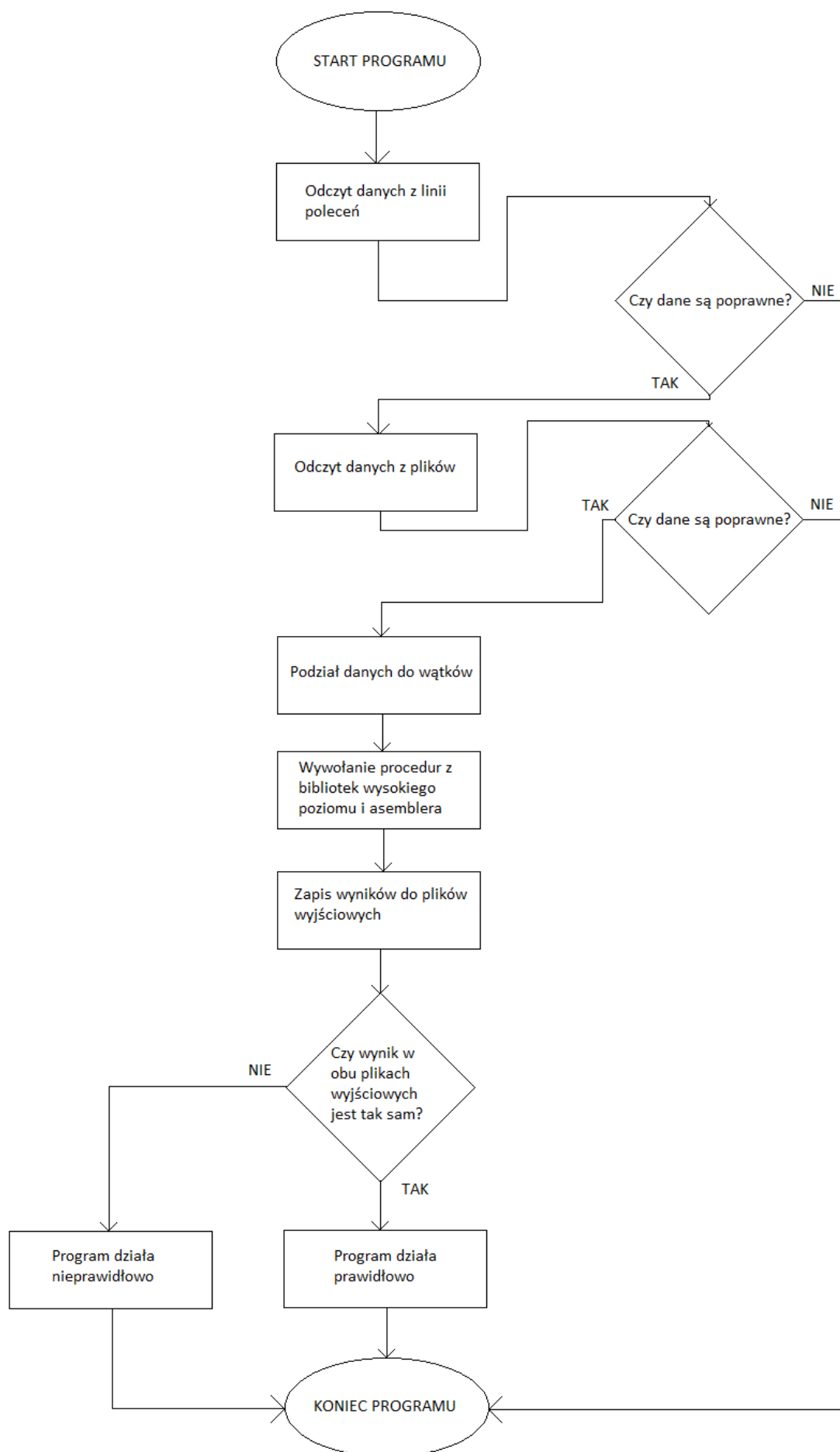
Odejmowanie dwóch macierzy nie jest zadaniem skomplikowanym. Pierwszą rzeczą jaką należy zrobić to sprawdzić czy dwie macierze mają ten sam rozmiar, czyli:

$$\begin{aligned} \text{liczba wierszy macierzy1} &= \text{liczba wierszy macierzy2} \\ \text{liczba kolumn macierzy1} &= \text{liczba kolumn macierzy2} \end{aligned}$$

Następnie należy odjąć od każdego elementu w pierwszej macierzy element w drugiej macierzy o takim samym położeniu. Macierz wynikowa jest takiego samego rozmiaru jak macierz1 oraz macierz2.

Problem ten rozwiązałem za pomocą jednowymiarowej tablicy dynamicznej. Rozważałem tablicę dwuwymiarową jednak zdecydowałem, że dzięki mojemu finalnemu rozwiązaniu uniknę przejścia pomiędzy wierszami co jest trochę problematyczne w języku asemblerowym.

3. Schemat blokowy programu



4. Opis programu w języku wysokiego poziomu i zmiennych globalnych

Na początku w programie głównym występuje pobranie procedur z bibliotek załadowanych dynamicznie. Po tej czynności odczytuję z linii poleceń nazwę plików wejściowych (po przełączniku „-i”) oraz liczbę wątków (po przełączniku „-t”) z warunkiem, że podana liczba jest z zakresu $<1,64>$. Jeśli liczba wątków jest nieprawidłowa to funkcją z biblioteki *thread* odczytuję liczbę rdzeni procesora, na którym uruchamiany jest program. Po odczycie danych wejściowych następuje zapis danych z plików do jednowymiarowych tablic dynamicznych oraz zapis wymiarów podanych macierzy w celu sprawdzenia poprawności rozmiaru. Jeśli pliki mają niepoprawny format bądź rozmiar program kończy swoją pracę informując użytkownika o błędzie. Po poprawnym odczycie danych następuje podział danych do wątków. Algorytm podziału danych do wątków jest następujący:

- Ilość liczb w macierzy jest dzielona przez 4, reszta przypisywana do ostatniego wątku
- Wynik całkowity z punktu a) jest dzielony przez liczbę wątków, reszta jest przepisana od pierwszego do przedostatniego wątku w zależności jaka ta liczba ma wartość
- Wynik całkowity z punktu b) jest równomiernie dzielony na liczbę wątków

Przykład:

Macierz ma rozmiar 7×11 , uruchamiany jest na 17 wątkach, otrzymujemy:

Numery wątków	Ilość liczb przydzielonych do wątku
1,2	8
3,4,5,6,7,8,9,10,11,12,13,14,15,16	4
17	5

Taki przydział danych do wątku jest bardzo wygodny, ponieważ w języku assemblerowym ładuję po cztery liczby do rejestrów SSE. Dzięki temu przydziałowi liczb mamy pewność, że dany wątek nie nadpisze danych z innego wątku wykonującego się równolegle. Zwracając uwagę na fakt, że ostatni wątek może mieć liczbę niepodzielną przez 4 zaalokowałem sobie pamięć na macierz wynikową o 3 (maksymalna reszta przy dzieleniu przez 4) pozycję. Dzięki tej operacji mam pewność, że nie odwołam się do nie zaalokowanej pamięci.

Po zapisie danych do wątku następuje najpierw wywołanie procedury w bibliotece wysokiego poziomu, następnie w języku assemblerowym. W trakcie wywoływania poszczególnych procedur uruchamiany jest stoper, który liczy czas potrzebny na odjęcie dwóch macierzy. Po tej operacji następuje zapis macierzy wynikowych do dwóch plików „OutputCpp.txt” oraz „OutputAsm.txt”. Na koniec programu wywoływania jest funkcja porównująca zawartość tych dwóch plików oraz wypisywany jest komunikat czy pliki są równe czy nie.

W programie głównym nie występują zmienne globalne.

5. Opis funkcji bibliotek DLL/C++ oraz DLL/ASM

W obu bibliotekach występuje jedna funkcja *MyProc* z następującymi parametrami:

- startingPos*- numer pozycji, od której rozpoczyna się odejmowanie
- numberOfPos*- liczba elementów do odjęcia
- t1*- pierwsza macierz wejściowa
- t2*- druga macierz wejściowa
- destMatrix*- macierz wynikowa

- biblioteka DLL/C++

W procedurze *MyProc2* występuje jedna pętla *for* przechodząca *numberOfPos* razy, a zaczynająca się w *startingPos*. W pętli *for* występuje odejmowanie *t1* od *t2*. Wynik tej operacji jest zapisywany w tablicy *destMatrixCpp*.

- biblioteka DLL/ASM

Argumenty procedury umiejscowione są w następujących rejestrach:

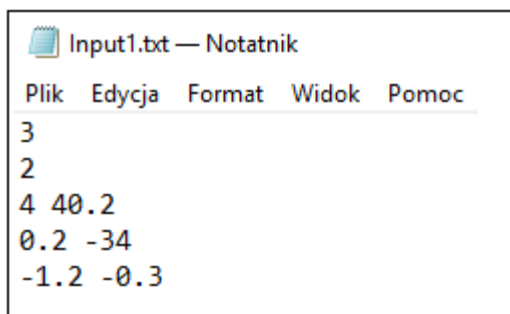
- startingPos*- rxc
- numberOfPos*- rdx
- t1*- r8
- t2*- r9
- destMatrixASM*- r10

Pierwszą operacją jaką zrobiłem to odczytanie ze stosu adresu na początek tablicy wynikowej i zapisanie jej do rejestru r10. Następnie występuje instrukcja mnożenia pozycji startowej przez 4 w celu przesunięcia się o określoną ilość bajtów. Wykorzystany rejestr tej operacji to r11. Potem przesuwam adres we wszystkich tablicach o ilość bitów zapisanych w r11. Następnie zapamiętuję adres macierzy wejściowych w *rax* oraz *rbx*. W tym momencie następuje pętla wykonująca się $rdx/4$ razy. W jej ciele wkładam po 4 liczby do rejestrów xmm0 oraz xmm1 i odejmuję otrzymując wynik w xmm0. Wynik zapisywany jest do pamięci tablicy wynikowej, który jest przechowywany w r10. Przed zakończeniem przebiegu pętli przesuwam adresy wszystkich tablic o 16 bajtów (4x4 bajty) oraz sprawdzam czy pętla jeszcze powinna się wykonywać: czy rdx jest równe zero lub czy rdx jest mniejsze od zera, ponieważ może wystąpić sytuacja gdy ostatni wątek dostanie liczbę niepodzielną niż 4.

6. Opis struktury danych wejściowych programu

Do prawidłowego działania programu niezbędne są dwa pliki wejściowe o nazwach „*Input1.txt*” oraz „*Input2.txt*”. W pierwszej linii występuje liczba wierszy, a w drugiej liczba kolumn. Od linii trzeciej jest macierz, w której liczby są oddzielone spacjami. Każdy następny wiersz jest w nowej linii.

Przykładowy plik wejściowy:



7. Opis uruchamiania programu i jego testowanie

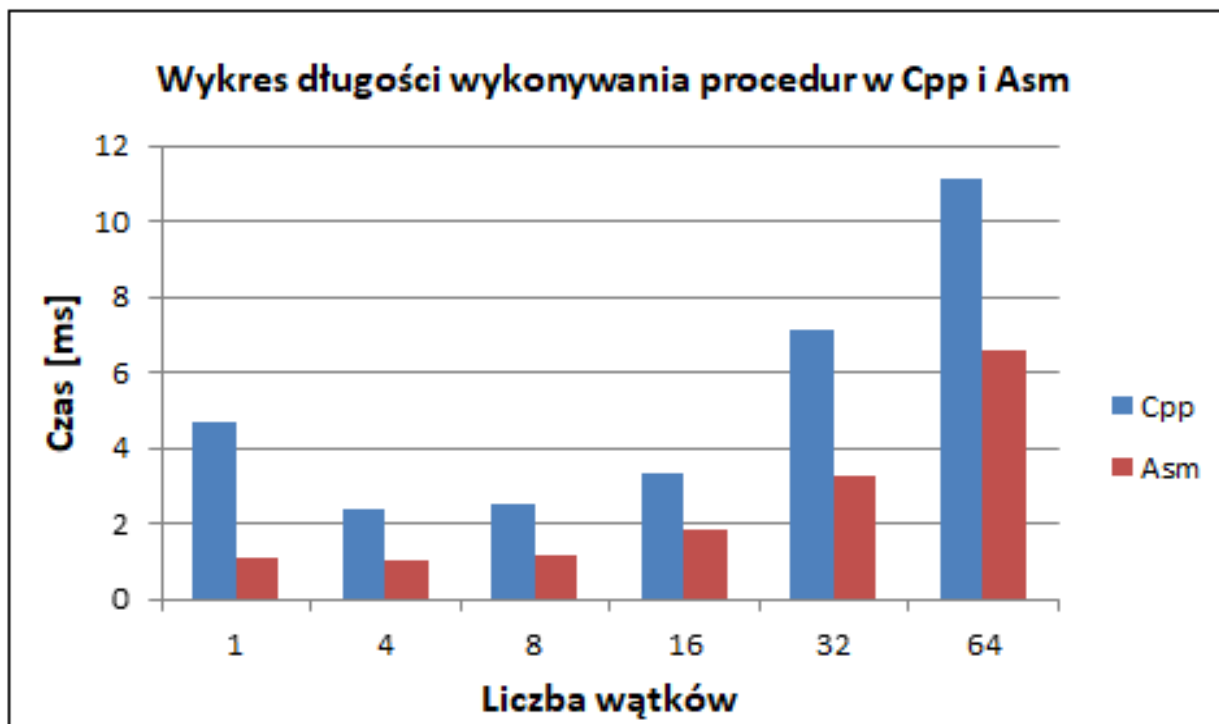
Od stworzenia wersji finalnej program został uruchomiony kilkadziesiąt razy za każdym razem z innymi danymi. Dane testowe zostały wylosowane losowymi liczbami rzeczywistymi z zakresu $<-10000,10000>$. Rozmiar danych za każdym razem ulegał zmianie. Pod koniec działania programu zostaje wywołana funkcja porównująca dwa pliki wyjściowe. W trakcie całego testowania za każdym razem wynik porównania brzmiał „Pliki są równe” co dowodzi o poprawności działania programu.

Program został zabezpieczony przed następującymi błędami:

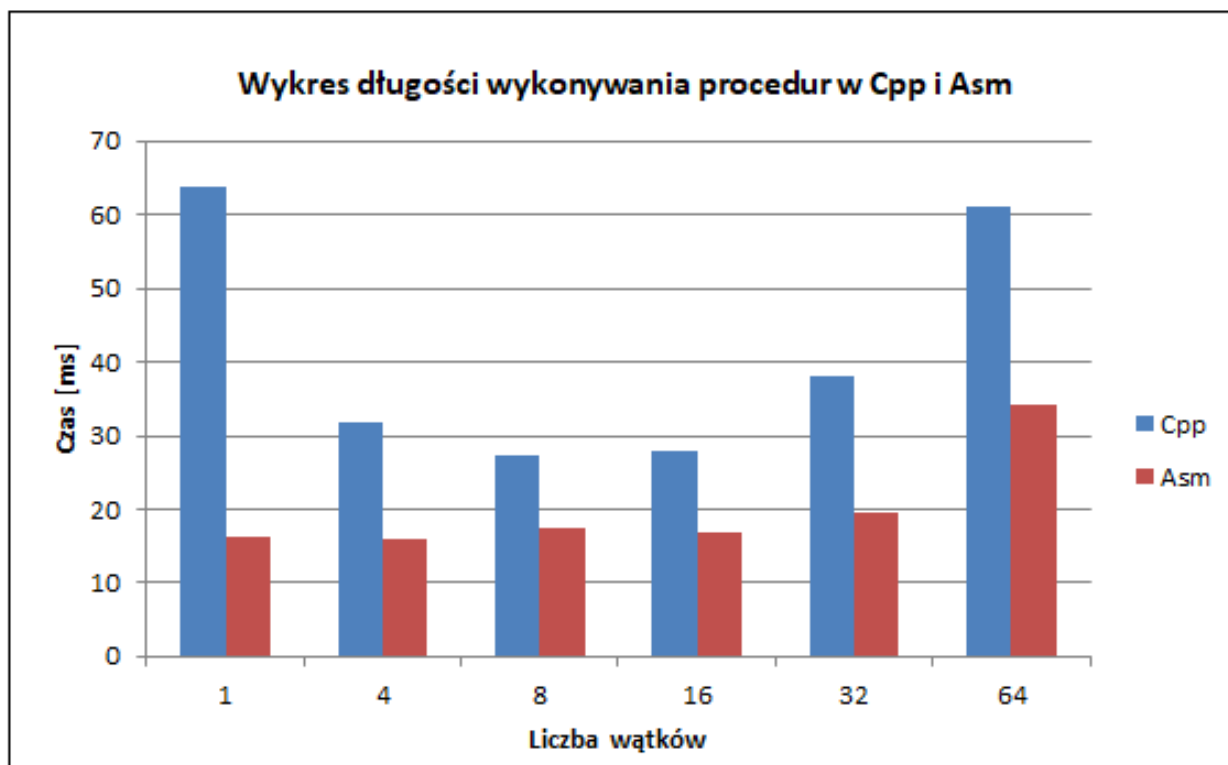
- Brak plików po przełączniku „-i”
- Niepoprawna liczba po przełączniku „-t”
- Niepoprawna nazwa (lub jej brak) pliku wejściowego
- Niepoprawne dane w plikach wejściowych
- Różne rozmiary macierzy wejściowych
- Niepoprawne stworzenie plików wyjściowych

8. Wyniki pomiarów czasu wykonania programu dla różnych danych testowych i obu wersji bibliotek przedstawione na wykresach porównawczych

Poniżej znajdują się 2 wykresy porównujące szybkość wykonywania procedur obu wersji bibliotek C++ oraz ASM w zależności od liczby wątków.

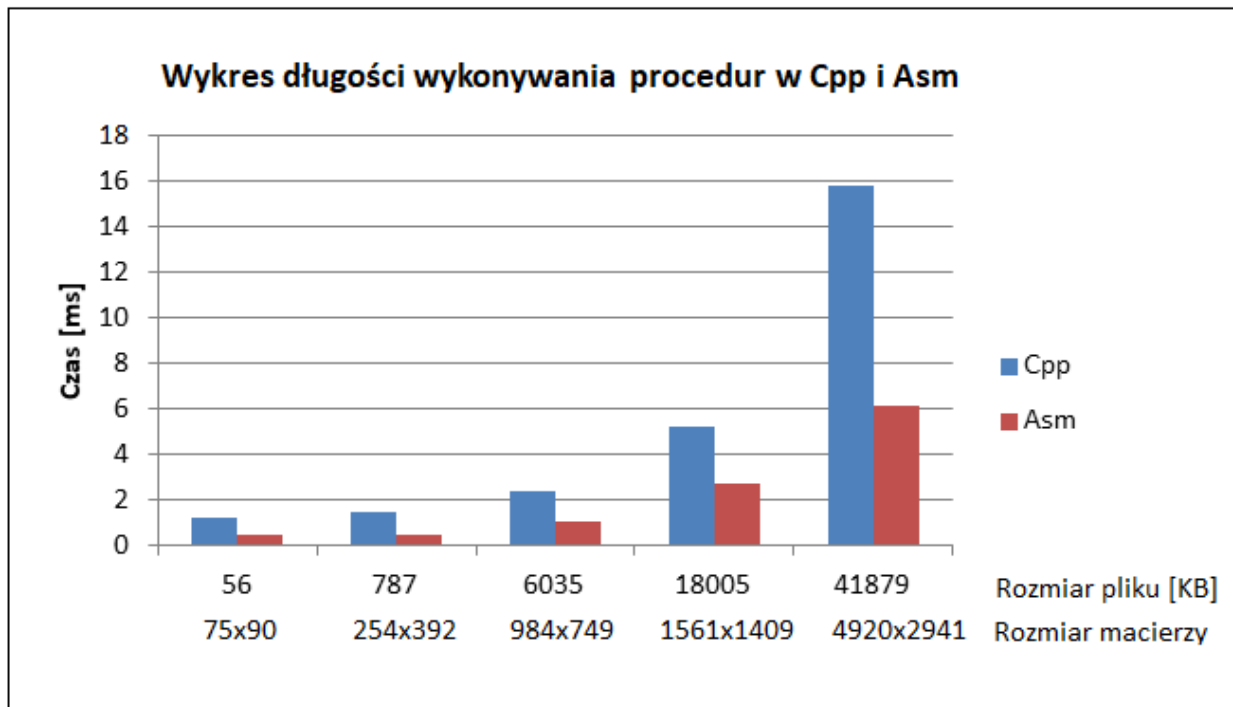


Rozmiar pliku: 6,035MB
Rozmiar macierzy: 984x749



Rozmiar pliku: 118,433 MB
Rozmiar macierzy: 4920x2941

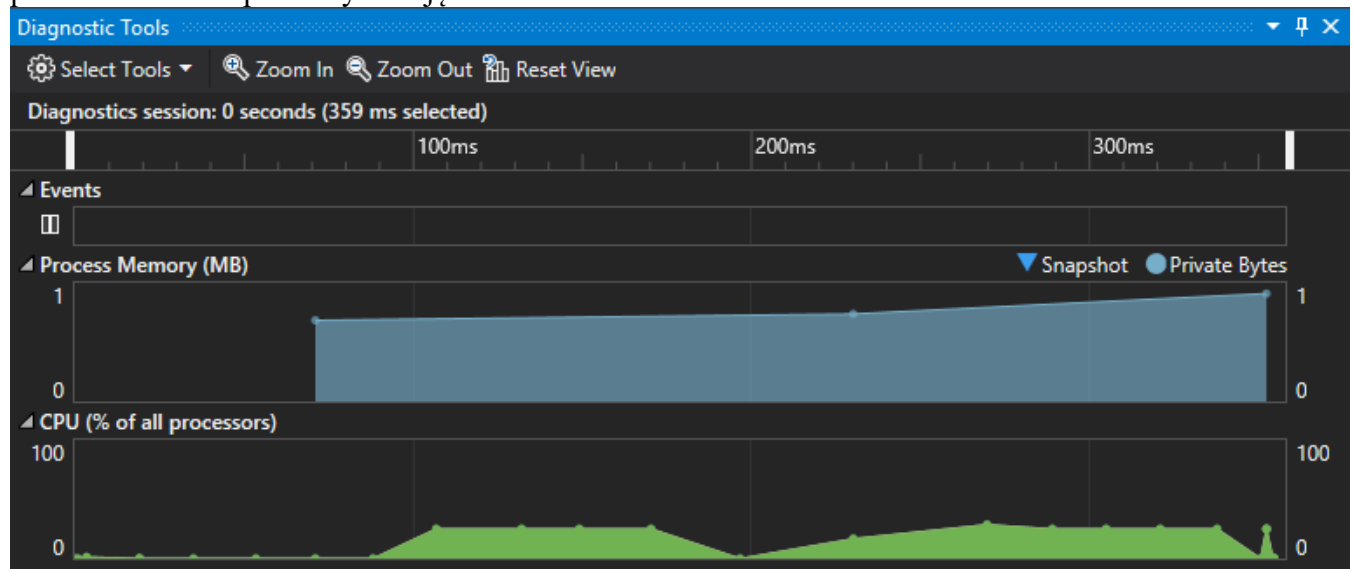
Na poniższym wykresie możemy zaobserwować porównanie dwóch bibliotek w zależności od rozmiaru plików wejściowych.



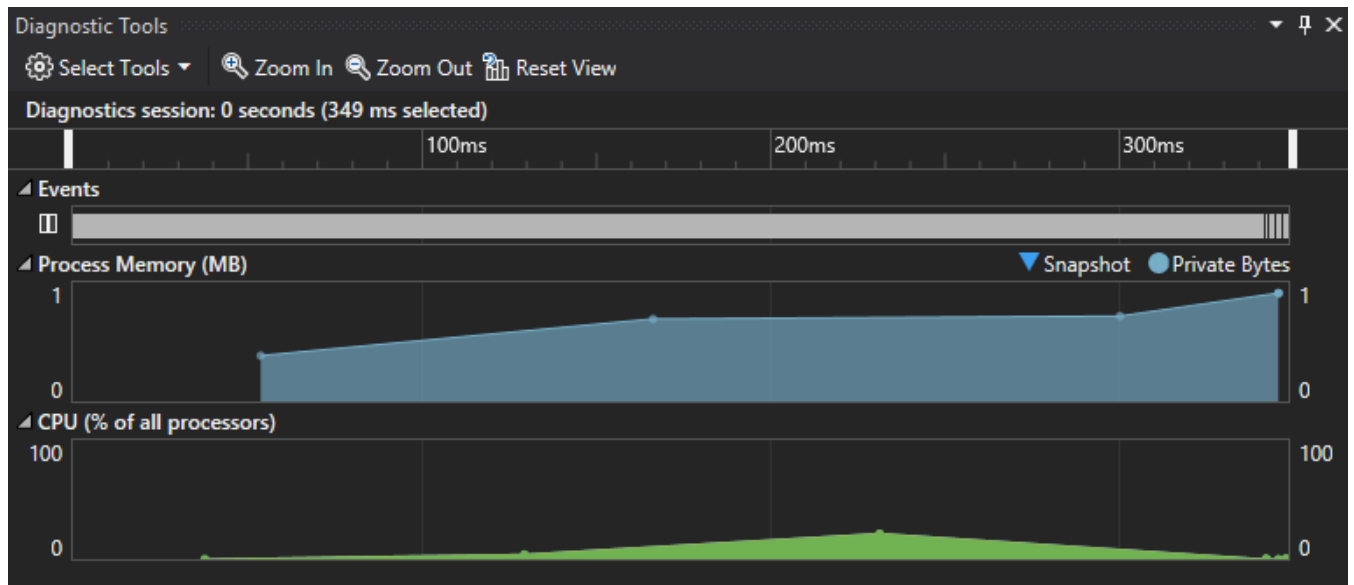
Liczba wątków: 4

9. Analiza działania programu z wykorzystaniem modułu profilera VS2015

Program został przetestowany narzędziem wbudowanym w środowisku Visual Studio 2015 o nazwie *Diagnostic Tools*. Analizie zostało podjęte procedury bibliotek DLL w C++ oraz ASM. Wyniki przedstawiono na poniższych zdjęciach.



C++



ASM

Widoczne jest większe zużycie procesora podczas działania w bibliotece napisanej w języku wysokopoziomym.

10. Instrukcja obsługi programu

Program jest konsolowy, uruchamiany z linii poleceń, w której należy podać kolejno:

- 1) Nazwę programu: MAIN.exe
- 2) Po przełączniku „-i” nazwy dwóch plików wejściowych
- 3) Po przełączniku „-t” liczbę wątków z warunkiem, że owa liczba musi się zawierać w przedziale od 1 do 64. Natomiast jeśli nie podano liczby wątków sczytywana jest ilość rdzeni procesora, na którym uruchamiany jest program.

Po poprawnym uruchomieniu programu pokazują się użytkownikowi następujące informacje:

- a) Liczba wątków
- b) Czasy wykonania procedur w C++ oraz ASM
- c) Potwierdzenie zapisu danych do plików wyjściowych
- d) Wynik porównywania plików

Przykładowe uruchomienie programu:

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.17134.407]
(c) 2018 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\Adam\Desktop\Studia\JA\PROJEKT\x64\Debug>MAIN.exe -i Input1.txt Input2.txt -t 5
Liczba watkow: 5
Czas dla cpp: 15072 microseconds
Czas dla asm: 7517 microseconds
Wynik dla Cpp znajduje sie w pliku OutputCpp.txt
Wynik dla Asm znajduje sie w pliku OutputAsm.txt
Pliki sa rowne

C:\Users\Adam\Desktop\Studia\JA\PROJEKT\x64\Debug>
```


11. Wnioski

Moim projektem z Języków Asemblerowych było napisanie programu mającego za zadanie odejmowanie dwóch macierzy o typie zmiennoprzecinkowym w języku: wysokiego poziomu (wybrałem C++) oraz asemblerowym. Zadanie to nie sprawiło mi żadnego problemu w języku wysokiego poziomu. Natomiast trochę problemów przysporzyło mi rozwiązanie tego samego problemu w języku niskopoziomowym.

Ważną kwestią, którą chciałem opisać był podział danych na wątki. Na początku postanowiłem podzielić dane równomiernie do każdego wątku. Jednakże potem zabrałem się do pisania w asemblerze wykorzystując instrukcję SSE. Zauważyłem, że są rejestry, które umożliwiają umiejscowienie aż 4 liczb zmiennoprzecinkowych pojedynczej precyzji. Stwierdziłem, że lepiej będzie, aby każdy wątek dostał do przetworzenia liczbę podzieloną przez 4. Tylko ostatni wątek może mieć liczbę niepodzielną przez 4, ale dzięki zaalokowaniu pamięci o 3 większej niż pierwotny rozmiar pozwoliło mi uniknąć odwołania się do pamięci nie zaalokowanej. Następnym problemem jakim napotkałem to reprezentacja liczb zmiennoprzecinkowych w rejestrach. Po zaznajomieniu się z rejestrami XMM problem zniknął, ponieważ korzystając z tych rejestrów możemy operować bezpośrednio na liczbach zmiennoprzecinkowych.

Po napisaniu dwóch bibliotek postanowiłem przetestować swój program. Do testów napisałem sobie dodatkową funkcję porównującą dwa pliki wyjściowe. Dla małych macierzy ta funkcja była bezproduktywna, jednak dla ogromnych macierzy dzięki funkcji nie musiałem sprawdzać tysiąca liczb z jednego i drugiego pliku. Niewątpliwie ułatwiło mi to zadanie i miałem pewność, że napisany przeze mnie program działa poprawnie. Po upewnieniu się o poprawności działania dwóch bibliotek postanowiłem porównać czasy działania procedur w języku wysokopoziomowym oraz niskopoziomowym. Po przeprowadzeniu tego eksperymentu zauważyłem, że procedura w języku wysokiego poziomu wykonywała się wolniej niż w asemblerze. Uważam, że dzieje się tak, ponieważ instrukcje wektorowe działają szybko, a dodatkowo pozwalają nam na przetworzenie aż 4 liczb jednocześnie. Program najszybciej działał na 4 wątkach (najwolniej na 64). Moim zdaniem spowodowane jest to tym, że na testowanej architekturze liczba rdzeni wynosi właśnie 4 i dzięki temu program na tylu wątkach działa najszybciej.

Podsumowując, projekt z przedmiotu Języki Asemblerowe pozwolił mi w znaczny sposób zrozumieć programowanie w języku niskopoziomowym. Dodatkowo poznałem instrukcje wektorowe, które bardzo przyspieszają czas działania programu. Program w C++ zajął mi całe dwie linie kodu, natomiast w asemblerze około 25 linii i nieporównywalnie więcej czasu. Pomimo tego to właśnie program w języku niskopoziomowym wykonuje się szybciej i to jest niewątpliwie zaleta oraz przewaga nad językami wysokopoziomowymi.

12. Literatura

<http://students.mimuw.edu.pl/~zbyszek/asm/en/instrukcje-sse.html>

https://pl.wikipedia.org/wiki/Streaming_SIMD_Extensions