



| | | | | |
|--|--|--------------------------------------|---|----------|
|  | Instytut Informatyki Politechniki Śląskiej Zespół Mikroinformatyki i Teorii Automatów Cyfrowych | |  | |
| Rok akademicki: | Rodzaj studiów*: SSI/NSI/NSM | Przedmiot (Języki Asemblerowe/SMiW): | Grupa | Sekcja |
| 2018/2019 | SSI | Języki Asemblerowe | 3 | 6 |
| Imię: | Miroslaw | Prowadzący: OA/JP/KT/GD/BSz/GB | AO | |
| Nazwisko: | Ściebura | | | |
| <h2 style="text-align: center;">Raport końcowy</h2> | | | | |
| Temat projektu: <div style="text-align: center; margin-top: 100px;"> <h1>Fraktal Julii</h1> </div> | | | | |
| Data oddania: dd/mm/rrrr | | 05/12/2018 | | |

1. Wstęp – Temat projektu i założenia

Tematem projektu było utworzenie aplikacji konsolowej, tworzącej bitmapę z fraktalem Julii. Aplikacja miała składać się z programu głównego w języku wysokiego poziomu (C, C++ lub C#) i biblioteki dll w dwóch wersjach: napisanej w języku C/C++ oraz w języku asemblera.

Program miał pobierać odpowiednie informacje z linii poleceń (która wersja biblioteki dll będzie wykorzystywana, wartość realis i imaginalis stałej zespolonej c oraz opcjonalnie liczbę wątków – bez podania zostanie wykorzystana maksymalna sprzętowa).

W bibliotekach zostały zaimplementowane procedury obliczające wartości kolejnych pikseli bitmapy z fraktalem w oparciu o przekazane wskaźniki początków tablic z wartością realis i imaginalis oraz tablicy z bajtami bitmapy, wartość indeksu wiersza od którego zaczynamy liczyć i offsetu wierszy, czyli ilości wierszy jaką będziemy obliczać w danym wątku.

Program został napisany w wersji dla procesorów 64-bitowych. W aplikacji zostały użyte instrukcje wektorowe SIMD oraz mechanizm wielowątkowości. Używanym środowiskiem programistycznym było Microsoft Visual Studio 2015. Zaimplementowany został pomiar czasu wykonania całego algorytmu wyznaczania wartości pikseli bitmapy.

2. Analiza zadania

Zbiór Julii jest fraktalem będącym podzbiorem płaszczyzny zespolonej. Zbiór ten tworzą punkty płaszczyzny zespolonej ($p \in \mathbb{C}$), dla których ciąg liczb zespolonych opisany równaniem rekurencyjnym:

$$\begin{aligned} z_0 &= p \\ z_{n+1} &= z_n^2 + c \end{aligned}$$

który nie dąży do nieskończoności:

$$\lim_{n \rightarrow \infty} z_n \neq \infty$$

Liczba c jest liczbą zespoloną będącą parametrem zbioru ($c \in \mathbb{C}$). Można wykazać, iż poprzednie warunki są równoważne:

$$\forall_{n \in \mathbb{N}} |z_n| < 2$$

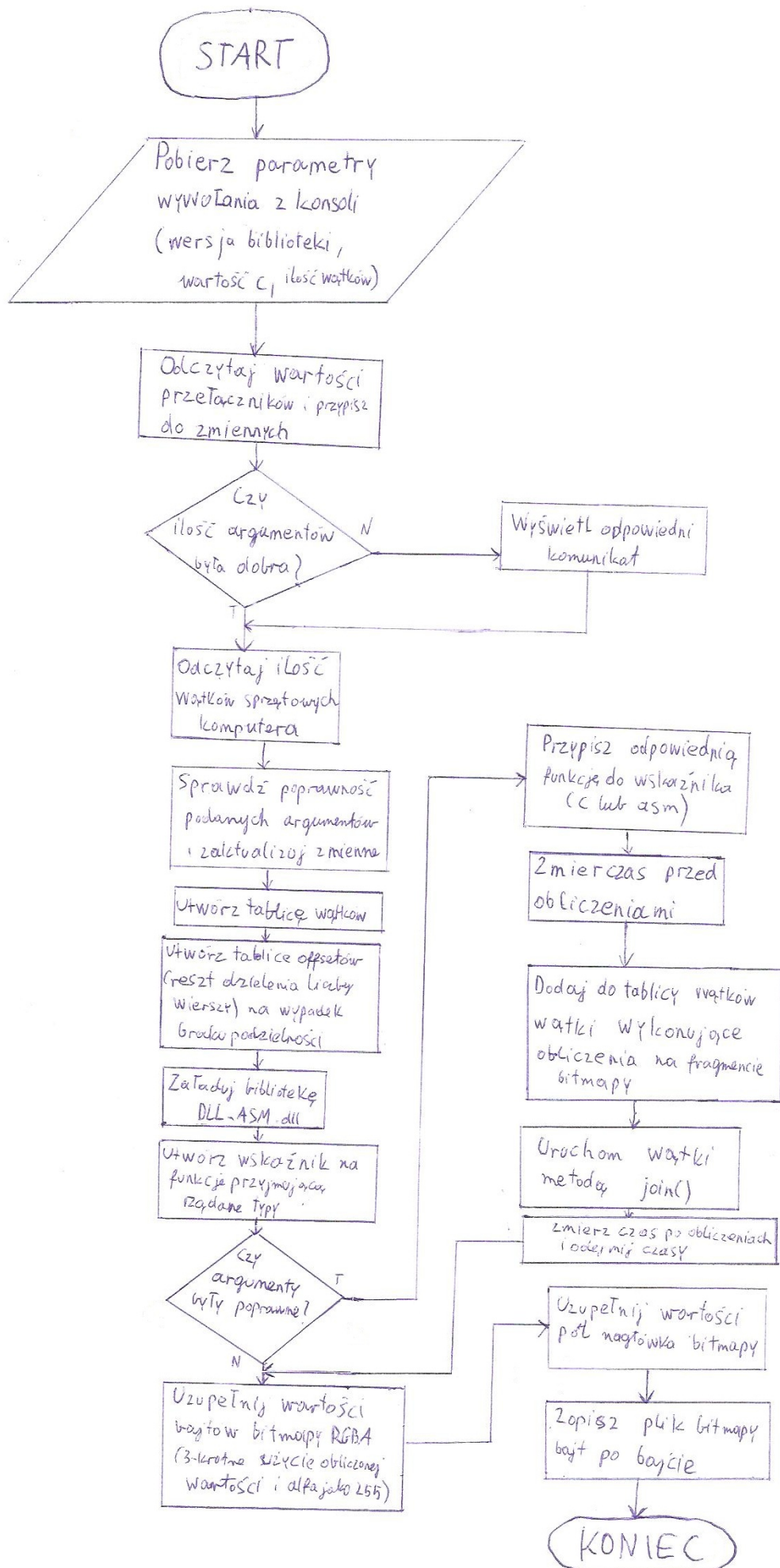
Zadanie obliczania wartości bajtów opierało się o zaimplementowanie obliczeń w oparciu o następujący pseudokod:

```
begin
  z := punkt;
  iteracja := 0;
  repeat
    z := z^2 + c;
    iteracja := iteracja + 1;
  until (|z| < 2) and (iteracja < MAX_ITERACJA)
  bajt := iteracja;
end
```

gdzie punkt to wartość realis i imaginalis analizowanego punktu płaszczyzny zespolonej, c to podana przez nas stała zespolona a bajt to wartość piksela w bitmapie (powielona na RGB – mapa w skali szarości).

Wybrane przeze mnie rozwiązanie zostało wybrane jako najbardziej przejrzyste podejście do problemu. Jest ono dość uproszczone, co pozwala na łatwe zrozumienie sensu algorytmu. Poza tym, jest to chyba najbardziej poprawne podejście.

3. Schemat blokowy



4. Specyfikacja zewnętrzna - opis struktury danych wejściowych programu

Program pobiera przełączniki i dane z linii poleceń. Pierwszym z potrzebnych przełączników jest przełącznik rodzaju używanej biblioteki. Mamy możliwość wpisania „-c” dla biblioteki utworzonej w języku C++ i „-a” dla biblioteki utworzonej w języku assemblera. Oprócz tego musimy podać przełącznik „-l” i po spacji wartości *realis* i *imaginalis* stałej *c*, rozdzielone przecinkiem (separatorem w liczbie rzeczywistej jest kropka). Opcjonalnie można podać po przełączniku „-t” i spacji liczbę rdzeni będącą liczbą naturalną, jednak w przypadku liczby większej od maksymalnej sprzętowej ilości wątków to ona zostanie wykorzystana. Program nie wczytuje żadnych dodatkowych plików – wytwarza całkowicie nową bitmapę na podstawie danych przełączników.

5. Specyfikacja wewnętrzna - opis programu w języku wysokiego poziomu

Na początku tworzymy dwa obiekty *vector*, przechowujące dane typu *unsigned char* (jeden bajt bez znaku). Jedną z nich, piksele, o wielkości szerokość razy wysokość bitmapy służy do obliczenia wartości pikseli bitmapy (bitmapa w skali szarości – wartości kopiowane na wartości RGB, alfa ustawiana na maksimum). Druga to tablica *Wyjsciova*, która jest większa od pierwszej o 4 razy (ilość bajtów na piksel – mapa typu RGBA), do której będziemy wpisywać wartości RGBA pikseli bitmapy (pomoże nam to później przy zapisie do pliku bajt po bajcie. W pliku nagłówkowym *MAIN.h* zostały zdefiniowane struktury *BMPHeader* i *DIBHeader* służące do określenia wartości w nagłówkach pliku bitmapy (zgodnie ze standardem dla bitmapy typu RGBA). Obiekty *vector* zostały wykorzystane, gdyż są one tworzone jak zwykła tablica (fragment pamięci, możliwość zwyczajnego przesuwania wskaźnika), natomiast nie ma możliwości przepełnienia stosu (bardzo duża ilość elementów).

Program odczytuje przełączniki i parametry podane w linii poleceń i sprawdza ich poprawność. Jedną ze zmiennych przypisywanych w zależności od przełącznika jest string z trybem (wartość „-a” dla biblioteki assemblerowej i „-c” dla biblioteki C++).

Mamy też zmienną *liczbaWatkow*, która powinna przyjąć wartość od 0 do maksymalnej liczby wątków sprzętowych (początkową wartością jest -1 w celu kontroli). W przypadku podania liczby wątków większej od maksymalnej sprzętowej przypisywana jest maksymalna sprzętowa liczba wątków.

Następnie wykorzystuje strumień *istringstream* i klasę *complex* do odpowiedniego parsowania *realis* i *imaginalis* podanej stałej *c*. Przy błędzie strumienia wyświetlany jest odpowiedni komunikat. Wartości te są umieszczane w dwuelementowej tablicy *float* celem przekazania wskaźnika jej początku do odpowiedniej funkcji.

Kolejnym krokiem jest utworzenie obiektu *vector* przechowującego z pustym wektorem do przechowania wątków.

Później dokonujemy odpowiedniego podziału linii w przypadku braku podzielności liczby wierszy na odpowiednią liczbę wątków. W tym celu tworzymy tablicę liczb całkowitych *offsety* i dokonujemy przypisania wiersza do odpowiedniej liczby początkowych pól tej tablicy (reszta jest równa 0).

W kolejnym kroku ładujemy bibliotekę „*DLL_ASM.dll*”. Możemy teraz przejść do wykonania odpowiednich obliczeń bajtów fraktala. Jeżeli parametry programu są poprawne (liczba wątków większa od -1, ilość argumentów programu większa od 3 i brak błędu strumienia dla liczby zespolonej), to w zależności od trybu wybieramy bibliotekę (C++ lub Assembler). Następnie odnotowujemy czas (*czas1*). Potem dodajemy do wektora kolejne wątki z żadaną przez nas wersją funkcji i jej parametrami wywołania (wskaźnik na tablicę z *realis* i *imaginalis* stałej *c*, wskaźnik na miejsce w tablicy piksele od którego zaczniemy obliczenia w danym wątku, indeks początkowego elementu, ilość obliczanych wierszy). Po dodaniu wszystkich wątków na każdym z nich wywołujemy metodę *join()*, która pozwala na rozpoczęcie działania wątku (w przypadku braku wolnego wątku czeka z dołączeniem do zwolnienia się któregoś z zajętych). Tym samym rozpoczęte są obliczenia tworzące fraktal. Po zakończeniu odnotowywany jest czas (*czas2*) i wyliczany czas trwania obliczeń w sekundach. Po wszystkim wyświetlany jest komunikat o rozpoczęciu zapisu do pliku.

Następnie przepisujemy odpowiednie elementy wektora piksele na elementy wektora tablicaWyjsciova (uzyskanie danych RGBA – kopiujemy wartości na R, G i B a A ustawiamy na 255).

W ostatnim kroku uzupełniamy wartości nagłówka BMP i nagłówka DIB odpowiednimi wartościami w celu utworzenia poprawnej informacji o pliku (utworzone struktury BMPHeader i DIBHeader) następnie przeprowadzamy zapis binarny wartości tych nagłówków do pliku. Następnie zapisujemy wartości pikseli bitmapy bajt po bajcie, „do góry nogami” (tak przechowywane są piksele w bitmapie według standardu). Po zakończeniu zapisu plik jest zamykany, wyświetlany komunikat o zakończeniu działania i potrzebie wciśnięcia klawisza Enter do zakończenia działania.

6. Specyfikacja wewnętrzna - opis funkcji bibliotek dll w języku C/C++ i Asemblerze

a) DLL_C.dll

Biblioteka ta zawiera jedną funkcję: fraktalJulii.

```
void fraktalJulii(float* tablica, unsigned char* tablicaPixeli, int index, int offsetPikseli)
```

Funkcja ta przyjmuje wskaźnik na początek tablicy z wartościami realis i imaginalis stałej zespolonej c, wskaźnik na miejsce w tablicy (odpowiednik kolejnego piksela w bitmapie) od którego będziemy przeprowadzać obliczenia, indeks miejsca w tablicy od którego zaczynamy obliczenia oraz offsetPixeli – wartość ta to ilość pikseli które będziemy obliczać w danym wątku.

Funkcja ta nie zwraca, wpisuje obliczone wartości do tablicaPixeli

Funkcja ta działa dla obszaru płaszczyzny zespolonej o wartościach realis od -1,6 do 1,6 i imaginalis od -1,2 do 1,2. Indeks i offsetPixeli są odpowiednio przeliczane na wiersze. Następnie w podwójnej pętli (kolejne piksele w wierszach) obliczana jest wartość piksela zgodnie z pseudokodem z analizy zadania. Po tym jak iteracje przestaną być przeprowadzane (przekroczenie maksymalnej ilości lub kwadrat modułu jest większy lub równy 4), ilość iteracji jest wpisywana pod odpowiedni indeks tablicy.

b) DLL_ASM.dll

Biblioteka ta zawiera jedną procedurę: fraktalJulii.

FraktalJulii proc

Parametry do funkcji przekazywane są zgodnie z konwencją fastcall czyli kolejno przez rejestry RCX, RDX, R8, R9. Otrzymuje te same dane które otrzymuje funkcja język C++ - wskaźnik na początek tablicy z wartościami realis i imaginalis stałej zespolonej c, wskaźnik na miejsce w tablicy (odpowiednik kolejnego piksela w bitmapie) od którego będziemy przeprowadzać obliczenia, indeks miejsca w tablicy od którego zaczynamy obliczenia oraz offsetPixeli (wartość ta to ilość pikseli które będziemy obliczać w danym wątku).

Początkowo, w liniach 80-89 obliczana jest wartość współczynników dzięki którym otrzymamy dane kolejnych punktów na płaszczyźnie. Następnie (linie 90-95) ładujemy dane stałej c do rejestrów XMM w celu ich późniejszego dodawania w pętli (realis w XMM4, imaginalis w XMM5). Potem (linie 96-110) następuje obliczenie indeksu wiersza początkowego w wątku, sumy jako warunku końca obliczeń i inicjalizacja liczników (EDX – licznik wiersza, ECX – licznik piksela w wierszu, R8D – suma wierszy i offsetu wierszy, R12D – wiersz od którego zaczynamy). Później wstawiamy na do odpowiednich rejestrów SIMD odpowiednie informacje potrzebne do wykonania pętli. Odpowiada za to makroinstrukcja wstawpoczek. Wstawia ona na odpowiadający pozycji w XMM bit w R9 jedynekę, wpisuje do XMM6 i XMM7 liczniki pętli z ECX i EDX a także inkrementuje odpowiednio liczniki ECX i EDX wraz z kontrolą przejścia poza ostatni piksel.

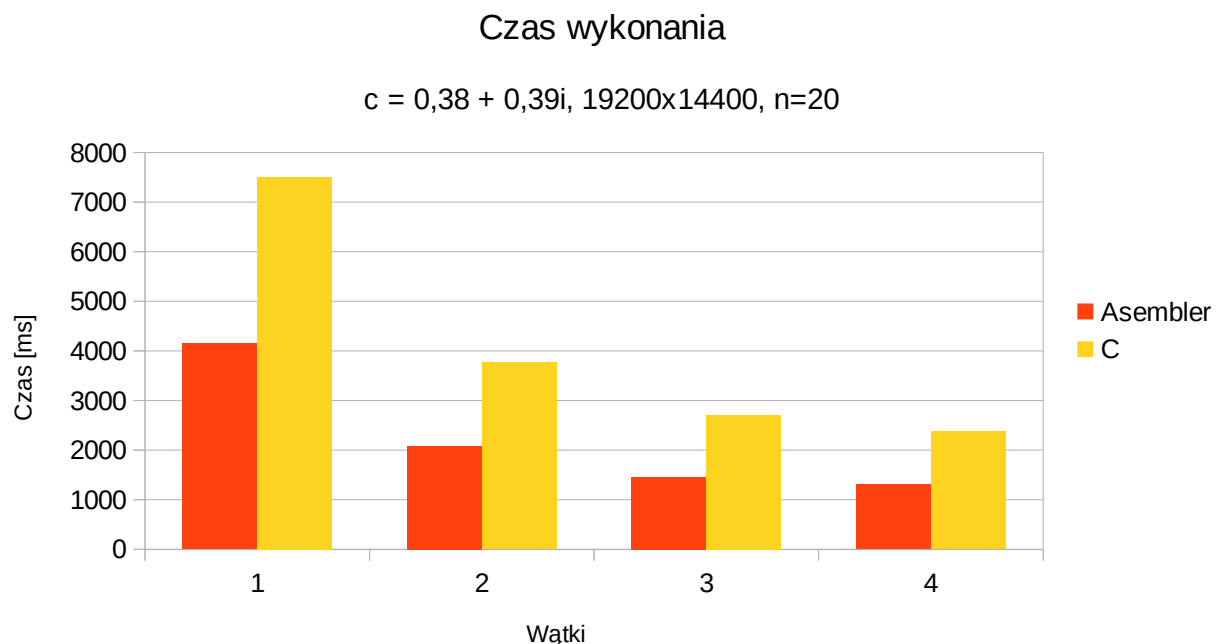
Następnie, po etykiecie pętla mamy konwersję na float wartości w XMM6 i XMM7. Następnie następują odpowiednie mnożenia poczwórnych wartości w wektorach w celu uzyskania wartości *realis* i *imaginalis* punktów płaszczyzny (linie 117-130). Następnie, we fragmencie opisanym etykietą *petlado* mamy implementację pseudokodu z analizy). Inkrementacja i porównanie licznika następuje w liniach 144-150 i wykorzystuje rozkazy PCMPGTD (porównaj czy liczby z rejestrów są większe na danych pozycjach, wpisz do rejestru docelowego maskę jedynek lub maskę zer w zależności czy warunek był spełniony czy nie) i MOVMSKPS (przenieś najstarszy bit z każdej pozycji rejestru XMM na odpowiadające najmłodsze pozycje rejestru docelowego). Następnie sprawdzamy na których pozycjach warunek został spełniony (wynik porównania jako wartość czterobitowa w R10, przemaskowanie tej wartości z maską w R11 w celu otrzymania jedynieżądanego bitu i porównania go z zerem. W zależności wartości dokonywana jest wymiana na danej pozycji określona makroinstrukcją *wymiana*, która na określonej pozycji wymienia dane w odpowiednich rejestrach (XMM0 – indeks elementu, XMM1 licznik iteracji, XMM6 *realis*, XMM7 – *imaginalis*). Makroinstrukcja kontroluje rejestr R9 (jeżeli zero na odpowiadającym bicie to instrukcja się nie wykona) i modyfikuje go gdy piksele do obliczania się skończą (następuje wyzerowanie odpowiedniego bitu R9). Później, w liniach 171-180 następuje obliczenie modułu i porównanie go z 4. Kontrola ewentualnej wymiany następuje jak w przypadku przekroczenia maksymalnej liczby iteracji. Na koniec, we fragmencie opisanym etykietą *czykoniec* sprawdzamy czy w rejestrze R9B jest choć jedna 1 (dalej musimy operować na wartości). Jeżeli są tam same zera, obliczenia zostają zakończone.

7. Opis uruchamiania programu i jego testowania

Program był uruchamiany z różnymi wartościami stałej *c*, co pozwoliło na zauważenie zależności przyspieszenia wykonania od danych. Przy danych które pozwalały na utworzenie obrazu mocno „zbinaryzowanego” (duży obszar pola białego – piksele dla których było dużo iteracji i mało wymian) można było uzyskać znacznie większe przyspieszenia przy użyciu kodu asemblerowego. Dla obrazów o większym udziale skali szarości i większym zróżnicowaniu powierzchni przyspieszenia te były mniejsze. Program był również uruchamiany z różną ilością wątków a także bez przełącznika „-t” w celu analizy wpływu większej ilości wątków na czas obliczeń. Oprócz tego testowany brak i zły format podanych wartości *realis* i *imaginalis* stałej *c*.

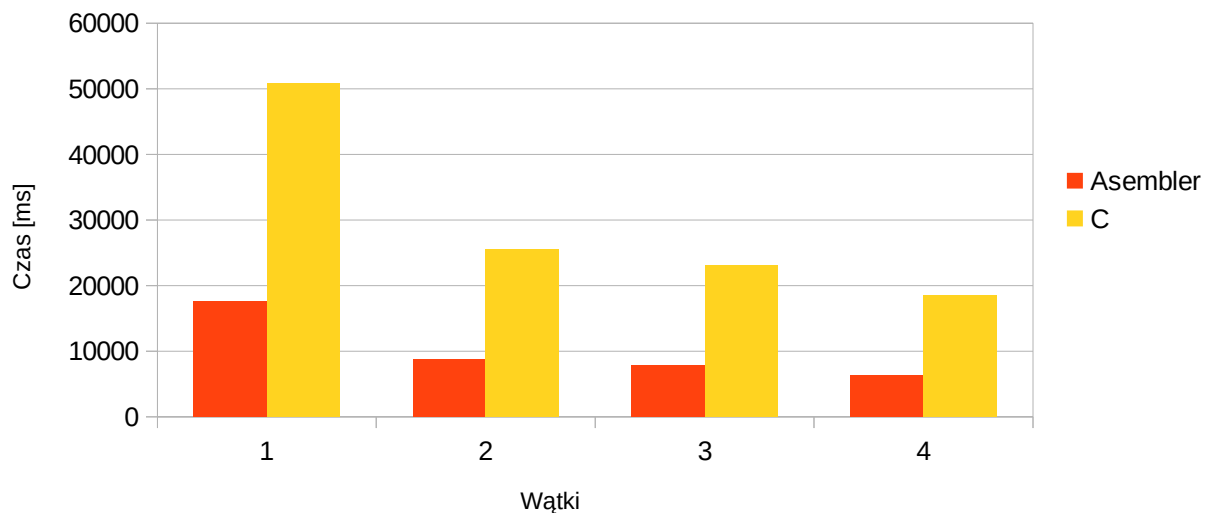
Program był uruchamiany wielokrotnie – dla tych samych i różnych ilości wątków w celu uzyskania ustabilizowanego wyniku czasowego (średnia czasu z *n* prób).

8. Wyniki pomiarów czasu wykonania

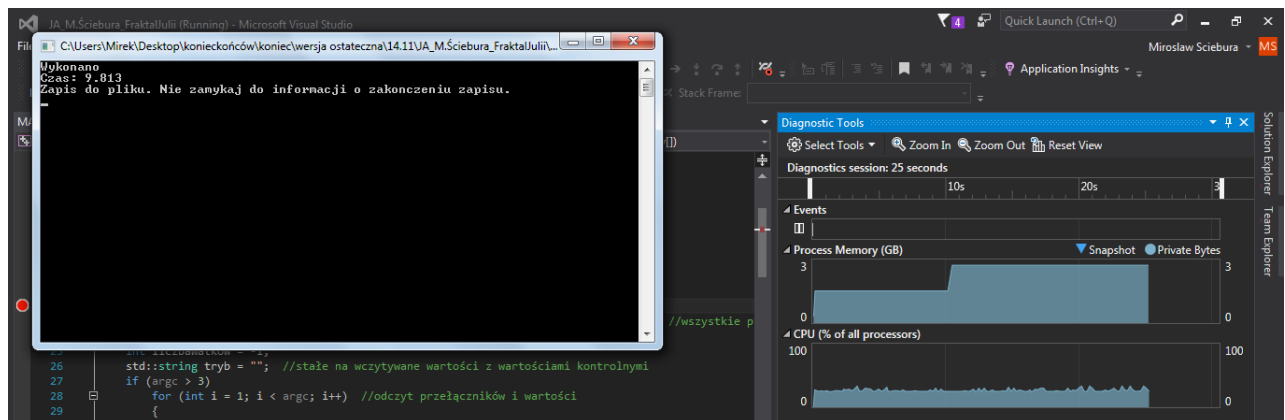


Czas wykonania

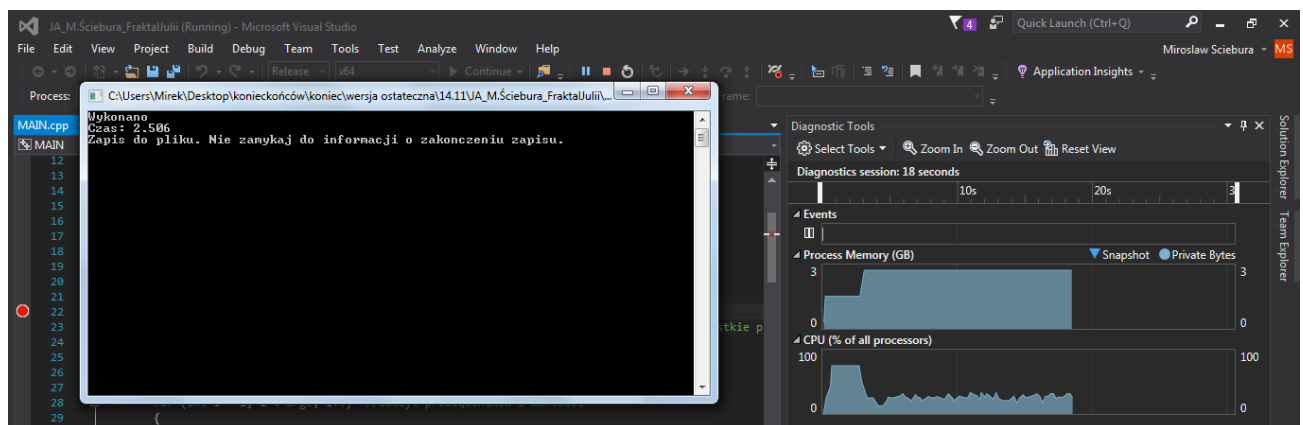
$c = -0,123 + 0,745i$, 19200×14400 , $n=5$



9. Analiza działania przy użyciu modułu profilera Microsoft Visual Studio 2015



Można zauważyć, że w momencie rozpoczęcia zapisu do pliku pamięć znacznie wzrasta (z 1GB do 3GB dla mapy 19200×14400). Użycie procesora przez cały czas kształtowało się na poziomie 25%, co miało związek z użyciem jednego wątku do obliczeń, a następnie wykonywaniem zapisu do pliku przez jeden wątek MAIN.



W przypadku użycia 3 wątków zużycie procesora w trakcie obliczeń jest odpowiednio większe (ok. 75%), a następnie kształtuje się na poziomie 25% w związku z użyciem jednej wątku (MAIN) do zapisu do pliku. Użycie pamięci nie zmienia się.

10. Instrukcja obsługi programu

- Otwórz wiersz polecenia systemu Windows (cmd.exe)
- Przejdź do folderu zawierającego program MAIN.exe i biblioteki DLL_C.dll i DLL_ASM.asm
- Uruchom program MAIN.exe z odpowiednimi przełącznikami:
 - -c dla użycia biblioteki języka C++, -a dla biblioteki języka Asemblera
 - przełącznik -l w celu podania stałej c, liczba podawana jest po spacji jako dwie wartości zmiennoprzecinkowe, rozdzielone przecinkiem
 - -t i liczbę wątków po spacji (liczba naturalna)przykładowe przełączniki: -a -l 0.38,0.39 -t 3
- Poczekaj na obliczenie pikseli w bitmapie a następnie na zapis do pliku
- Zamknij konsolę i otwórz powstały plik

11. Wnioski

Podczas testów można było zauważyć, iż przyspieszenie może mocno zależeć od danych (dla mocno „binarnego” fraktala mamy przyspieszenia rzędu 3 razy, dla bardziej zróżnicowanego około

2). Oprócz tego można odnotować, że zastosowanie 2 rdzeni pozwala w obu przypadkach na dwukrotne

przyspieszenie obliczeń, dla 3 przyspieszenie zależy od danych (w pierwszym przypadku przy podzieleniu w każdym wątku będzie podobna liczba danych podobnego typu, w drugim „środkowy” będzie trwał bardzo długo – dużo iteracji). Jako, że testowy komputer posiadał 4 wątki sprzętowe, ustalenie liczby wątków na 4 nie dawało dużego przyspieszenia w stosunku do 3 wątków – jeden wątek jest zarezerwowany dla procesu programu głównego MAIN. Warto też zauważyć, iż każdy kolejny wątek powyżej liczby sprzętowych wątków komputera nie da zbyt dużego przyspieszenia, gdyż i tak będziemy musieli czekać na zakończenie działania przez inny wątek.

Realizacja projektu pozwoliła na dobre przybliżenie kwestii optymalizacji kodu i samego używania języka Asemblera oraz dała dobrą bazę do dalszego używania instrukcji SIMD.

12. Literatura i źródła

- E. Wróbel “Programowanie w języku asemblera MASM. Laboratorium”, Wydawnictwo Politechniki Śląskiej, Gliwice 2005
- Materiały wykładowe przedmiotu “Języki Asemblerowe”, sem IV
- pl.wikipedia.org/wiki/Zbi%C3%B3r_Julii
- en.wikipedia.org/wiki/Julia_set
- www.algorytm.org/fraktale/zbior-julii.html
- www.felixcloutier.com/x86/