Filip Sołdon, 249454

Wrocław, 26.05.2020 r.

Termin: Poniedziałek, 13:00-16:00 TN

Sprawozdanie z laboratorium nr 4

„Architektura Komputerów 2”

Rok akademicki: 2019/2020, kierunek: INF

Prowadzący:

Mgr inż. Tomasz Serafin

1. Cel ćwiczenia:

Celem laboratorium nr 4 było stworzenie programu wykonującego podstawowe operacje arytmetyczne (dodawanie, odejmowanie, mnożenie, dzielenie) na wektorach 128-bitowych (metoda SIMD) oraz analogiczny bez ich wykorzystania (metoda SISD). Należało zmierzyć czas wykonywania operacji dla 2048, 4096 i 8192 liczb posługując się językiem C połączonym z asemblerem, a następnie przeanalizować je – w postaci wykresów i wniosków końcowych.

2. Przebieg pracy na programem:

Pracę nad programem zacząłem od zapoznania się z instrukcją do laboratorium nr 4 oraz niezbędnymi instrukcjami realizującymi operacje na wektorach 128-bitowych. Następnym krokiem było przemyślenie, w jaki sposób zaimplementować wektory liczb w programie. W tym celu zastosowałem odpowiednie struktury do przechowywania wektorów liczb. Kolejnym krokiem było przemyślenie, w jakiś sposób wypełniana będzie struktura przechowująca wektory. W tym celu stworzyłem funkcję generującą liczby pseudolosowe wypełniającą strukturę wektorów liczbami typu float. Następnie przeszedłem do implementacji podstawowych działań arytmetycznych z i bez wykorzystania SSE w formie wstawek asemblerowych. Potem przeszedłem do projektowania programu oraz później do jego implementacji. Program zostały napisany w wersji 32-bitowej. Program był testowany w środowisku Linux Ubuntu 64-bit.

3. Napotkane problemy:

Pierwszym napotkanym problemem był wybór, na jakich liczbach wykonywane będą operacje. W pierwszej kolejności wybrałem liczby typu całkowitego, jednak okazało się, że zaimplementowanie mnożenia i dzielenia jest trudne do wykonania ze względu na brak odpowiednich instrukcji. Dlatego ostatecznie program działa na liczbach zmiennoprzecinkowych.

Kolejnym często pojawiającym się problemem były wstawki języka asemblerowego w kodzie C, które po skompilowaniu bardzo często zwracały błąd – następowało naruszenie ochrony pamięci. Problem został rozwiązany.

4. Kluczowe fragmenty kodu:

a) struktura do przechowywania wektorów 128-bitowych składających się z 4 liczb typu float (po 32 bity każda)

```
12 struct wektorFloat
13 {
14         float num1;
15         float num2;
16         float num3;
17         float num4;
18 };
```

b) funkcja generująca liczby pseudolosowe do wypełnienia wektorów – generator wektorów:

```
228 struct wektorFloat generatorWektorow()
229 {
230         struct wektorFloat wektor;
231         wektor.num1 = (rand() % 10000000) / 100.0;
232         wektor.num2 = (rand() % 10000000) / 100.0;
233         wektor.num3 = (rand() % 10000000) / 100.0;
234         wektor.num4 = (rand() % 10000000) / 100.0;
235
236         return wektor;
237 }
```

c) funkcja wykonująca operację dodawania metodą SIMD wraz z obliczaniem czasu wykonania danej operacji:

```
22 double dodawanieSIMD(struct wektorFloat a, struct wektorFloat b)
23 {
24         time_t t_poczatek;
25         time_t t_koniec;
26
27         t_poczatek = clock();
28         asm(
29                 "movups (%0), %%XMM0\n"
30                 "movups (%1), %%XMM1\n"
31                 "addps %%XMM0, %%XMM1\n"
32                 :
33                 : "r" (&a), "r" (&b)
34            );
35         t_koniec = clock();
36         return difftime(t_koniec, t_poczatek) / (double)CLOCKS_PER_SEC;
37 }
```

Argumentami funkcji są wektory a i b, a zwracaną wartością typu double jest czas wykonania pojedynczej operacji. Przed wykonaniem operacji dodawania, rozpoczynamy mierzenie czasu przez wywołanie funkcji clock() z biblioteki time.h, która zwraca przybliżoną wartość czasu procesora zużytego przez program. W kodzie asemblerowym ładujemy wektor a do rejestru xmm0, a wektor b do rejestru xmm1. Następnie wykonujemy operację dodawania za pomocą rozkazu „addps”. Przyrostek „ps” oznacza, że instrukcja operuje na wszystkich elementach równocześnie. Następnie kończymy zliczanie czasu przez ponowne wywołanie funkcji clock() i przypisanie czasu zmiennej t_koniec. Wartość zwracana to różnica czasu zakończenia i rozpoczęcia zliczania czasu liczona za pomocą funkcji difftime(), której wartość dzielimy przez stałą, aby uzyskać czas w sekundach.

Pozostałe operacje bazujące na metodzie SIMD są analogiczne – różnią się rozkazem (subps, mulps, divps).

d) funkcja wykonująca operację dodawania metodą SISD wraz z obliczaniem czasu wykonania danej operacji:

```
 93 double dodawanieSISD(struct wektorFloat a, struct wektorFloat b)
 94 {
 95
 96         time_t t_poczatek;
 97         time_t t_koniec;
 98
 99         t_poczatek = clock();
100         asm(
101                 "fld (%0)\n"
102                 "fld (%1)\n"
103                 "faddp %%st(0), %%st(1)\n"
104                 "fstp %%st(3)\n"
105
106                 "fld 4(%0)\n"
107                 "fld 4(%1)\n"
108                 "faddp %%st(0), %%st(1)\n"
109                 "fstp %%st(3)\n"
110
111                 "fld 8(%0)\n"
112                 "fld 8(%1)\n"
113                 "faddp %%st(0), %%st(1)\n"
114                 "fstp %%st(3)\n"
115
116                 "fld 12(%0)\n"
117                 "fld 12(%1)\n"
118                 "faddp %%st(0), %%st(1)\n"
119                 "fstp %%st(3)\n"
120                 :
121                 : "r" (&a), "r" (&b)
122            );
123         t_koniec = clock();
124         return difftime(t_koniec, t_poczatek) / (double)CLOCKS_PER_SEC;
125 }
```

Argumentami funkcji są wektory a i b, a zwracaną wartością typu double jest czas wykonania pojedynczej operacji. Wektory są używane tylko jako argumenty przekazywane do funkcji. Zliczanie czasu odbywa się dokładnie tak samo jak w przypadku operacji metodą SIMD. W kodzie asemblerowym wstawiamy kolejno liczby typu float z wektorów a i b oraz wykonujemy kolejno operacje dodawania za pomocą rozkazu „faddp”. Operacje są wykonywane dla czterech bloków instrukcji.

Pozostałe operacje bazujące na metodzie SISD są analogiczne – różnią się rozkazem (fsub, fmul, fdiv).

e) funkcja obliczająca średni czas wykonywania operacji dla zadanej liczby pomiarów:

```
241 double testDodawanie(int wielkosc, int typ)
242 {
243         double suma = 0;
244         int i;
245         int j;
246
247         for(i = 0; i < liczbaPomiarow; i++)
248         {
249                 for(j = 0; j < wielkosc; j++)
250                         if(typ == 1)
251                                 suma += dodawanieSIMD(generatorWektorow(), generatorWektorow());
252                         else
253                                 suma += dodawanieSISD(generatorWektorow(), generatorWektorow());
254         }
255         suma /= (double) liczbaPomiarow;
256
257         return suma;
258 }
```

f) funkcja generująca wyniki i zapisująca je w zgodnie z szablonem podanym przez prowadzącego:

```
319 void wynik(int wielkosc)
320 {
321         FILE *file;
322                         char name[10] = {"Wyniki.txt"};
323
324                         if(file = fopen(name, "a"))
325                         {
326                                 fprintf(file, "Typ obliczen: SISD\n");
327                                 fprintf(file, "Liczba liczb: %d \n", wielkosc);
328                                 fprintf(file, "Liczba pomiarow: %d \n", liczbaPomiarow);
329                                 fprintf(file, "Sredni czas [s]:\n");
330                                 fprintf(file, "+ %f \n", testDodawanie(wielkosc, 0));
331                                 fprintf(file, "- %f \n", testOdejmowanie(wielkosc, 0));
332                                 fprintf(file, "* %f \n", testMnozenie(wielkosc, 0));
333                                 fprintf(file, "/ %f \n\n", testDzielenie(wielkosc, 0));
334
335                                 fprintf(file, "Typ obliczen: SIMD\n");
336                                 fprintf(file, "Liczba liczb: %d \n", wielkosc);
337                                 fprintf(file, "Liczba pomiarow: %d \n", liczbaPomiarow);
338                                 fprintf(file, "Sredni czas [s]:\n");
339                                 fprintf(file, "+ %f \n", testDodawanie(wielkosc, 1));
340                                 fprintf(file, "- %f \n", testOdejmowanie(wielkosc, 1));
341                                 fprintf(file, "* %f \n", testMnozenie(wielkosc, 1));
342                                 fprintf(file, "/ %f \n\n", testDzielenie(wielkosc, 1));
343                         }
344                         else
345                         {
346                                 printf("Blad.\n");
347                         }
348
349
350 }
```

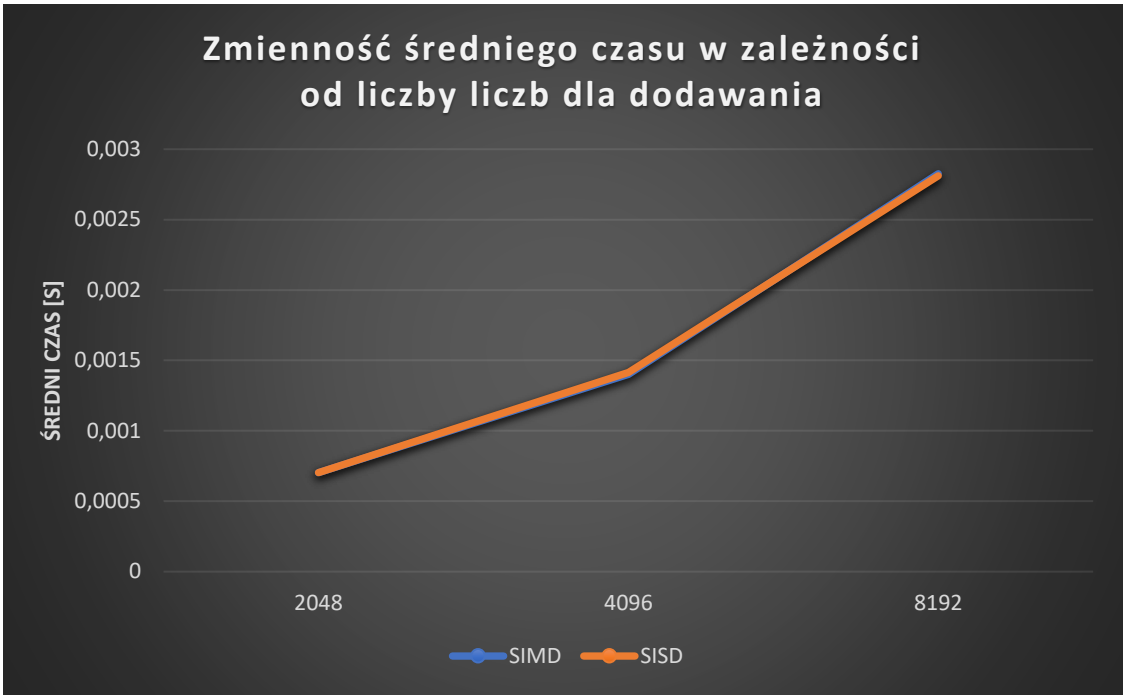
5. Opis uruchomienia programu:

Zawartość pliku „makefile”:

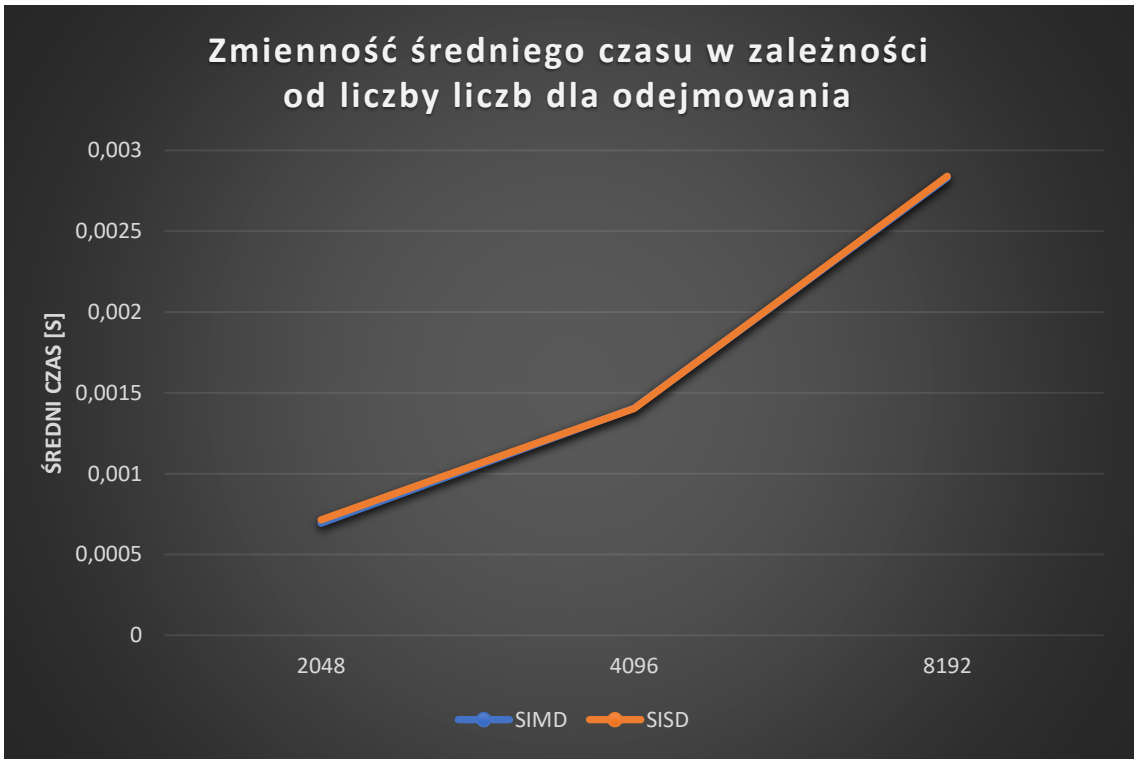
```
all: Lab4

Lab4: Lab4.c
        gcc -m32 Lab4.c -o Lab4
```

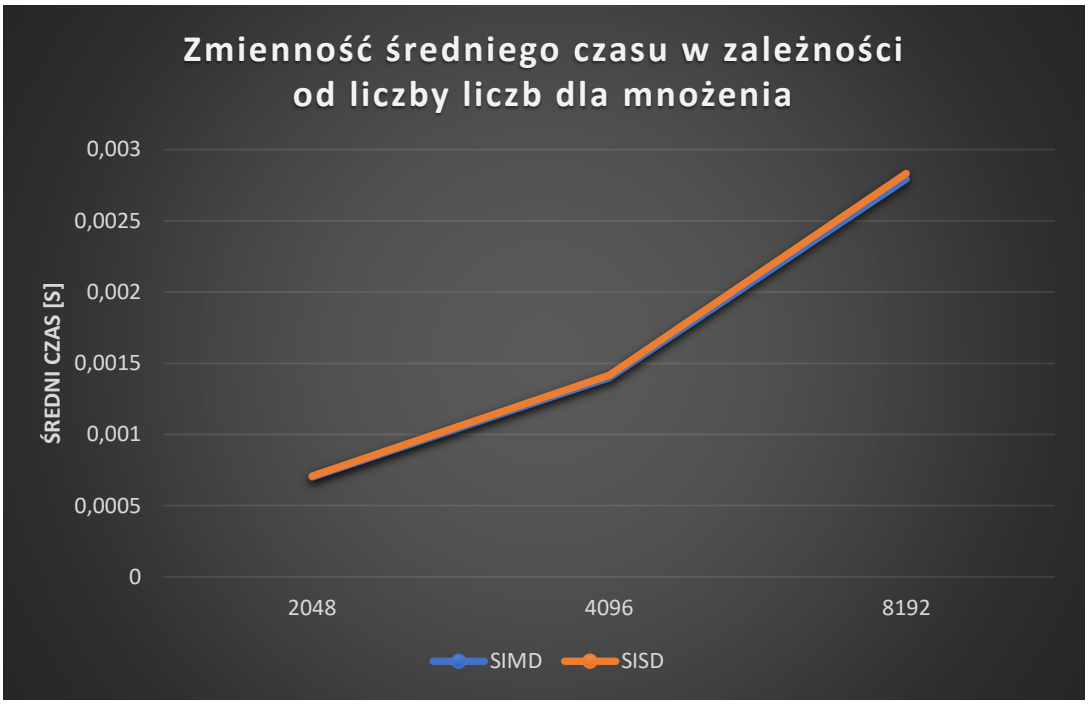
6. Wykresy:



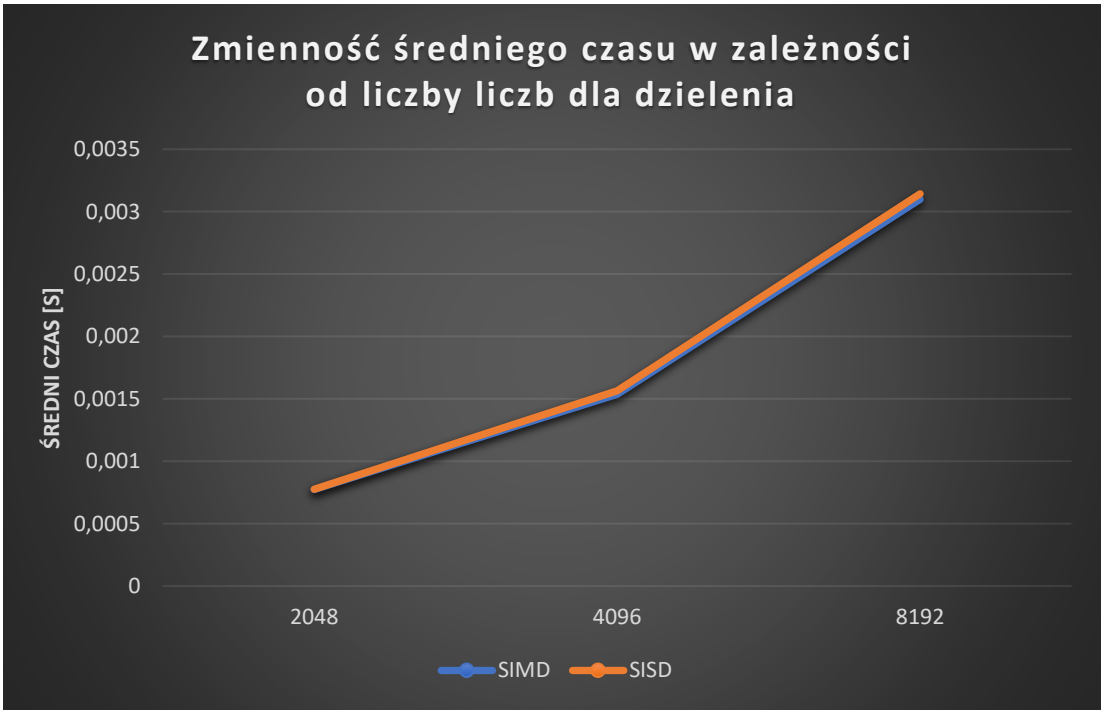
Wykres 1. przedstawiający zmienność średniego czasu w zależności od liczby liczb dla operacji dodawania metodą SIMD/SISD



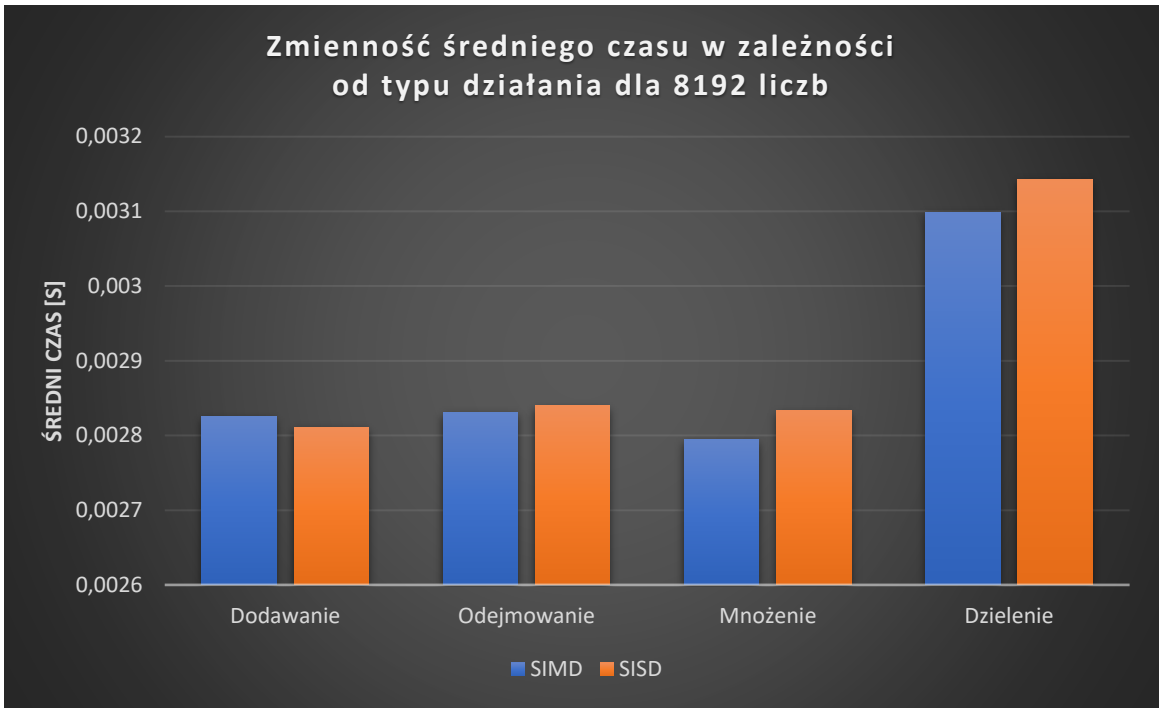
Wykres 2. przedstawiający zmienność średniego czasu w zależności od liczby liczb dla operacji odejmowania metodą SIMD/SISD



Wykres 3. przedstawiający zmienność średniego czasu w zależności od liczby liczb dla operacji mnożenia metodą SIMD/SISD



Wykres 4. przedstawiający zmienność średniego czasu w zależności od liczby liczb dla operacji dzielenia metodą SIMD/SISD



Wykres 5. przedstawiający zmienność średniego czasu w zależności od typu działania dla 8192 liczb

	2048	4096	8192
Dodawanie	0,00%	1,06%	-0,53%
Odejmowanie	2,94%	-0,21%	0,32%
Mnożenie	0,99%	1,13%	1,38%
Dzielenie	0,26%	1,79%	1,40%

Tabela 1. przedstawiająca zysk/stratę z zastosowania mechanizmów SIMD w stosunku do SISD wyrażony w procentach

7. Wnioski:

Pierwsze wnioski, jakie nasuwają się po spojrzeniu na wykresy to fakt, iż różnica średniego czasu pomiędzy operacjami wykonanymi metodą SIMD i SISD jest znikoma, a wręcz niezauważalna. Wyniki powinny być jednak z goła inne, ponieważ logiczne wydaje się, iż równoległe przetwarzanie wielu instrukcji powinno być szybsze od wykonywania ich sekwencyjnie. Wynik jest z pewnością obarczony błędem wynikającym z przeprowadzenia testów na maszynie wirtualnej, co tłumaczy fakt, iż w warunkach domowych nie byłem w stanie przeprowadzić testów pozbawionych wpływów z np. aplikacji działających w tle.

Kolejnym istotnym element jest to, że można doszukać się w wynikach pewnego błędu systematycznego, ponieważ po każdym uruchomieniem testów wyniki były inne, co z pewnością wynika z wspomnianego wcześniej faktu, iż działanie programu mogą przerywać aplikacje i usługi działające w tle i obciążające procesor, na co nie mamy realnego wpływu.

Analizując wykres wykonany w celu porównania czasu działania operacji metodą SIMD i SISD dla 8192 liczb, można zauważyć także, iż operacje wykonywane metodą SIMD są nieznacznie szybsze w przypadku odejmowania, mnożenia i dzielenia, natomiast operacja dodawania jest nieznacznie szybsza w przypadku metody SISD.