29.05.2020

Michał Dunat, 248862

Oddano: 29.05.2020

Ocena:

Porównanie czasu wykonywania podstawowych operacji arytmetycznych w architekturach SISD i SIMD

Sprawozdanie z laboratorium „Organizacja i Architektura Komputerów”

Rok akad. 2019/2020, kierunek INF

Prowadzący: mgr inż. Tomasz Serafin

Spis treści

Przebieg pracy nad programem

Na początku zbudowałem odpowiednie zaplecze teoretyczne. Zebrałem i przyswoiłem informację o architekturach SISD i SIMD, instrukcji SSE, kompilatorze GCC, odświeżyłem wiedzę o jednostce zmiennoprzecinkowej procesora x87 FPU nabytą podczas laboratoriów 3 i doczytałem o wstawkach asemblerowych w języku C. Następnie przystąpiłem do napisania funkcji przeprowadzających wymagane operacje arytmetyczne (dodawanie, odejmowanie, mnożenie, dzielenie) przy wykorzystaniu wektorów (instrukcje SSE) i normalnych liczb (instrukcje FPU). Po sprawdzeniu poprawności wyników generowanych poprzez w/w funkcje uzupełniłem je o mechanizm mierzenia czasu wykonywania poszczególnych kroków algorytmów. Następnym przeze mnie podjętym działaniem było napisanie funkcji generujące pseudo-losowe liczby wypełniającej wektory w tablicy wektorów i utworzenie w „mainie" pętli odpowiedzialnych za wykonywanie pomiarów. Na samym końcu oprogramowałem zapis wyników pomiarów do pliku i dodałem komentarze do kodu.

Napotkane problemy

Mimo uprzedniej wiedzy nabytej o wstawkach asemblerowych problematyczne okazały się dwa zagadnienia:

- Wykorzystanie odpowiednich operandów wejściowych i wyjściowych .
- Ograniczenia w tychże wstawkach.

Dodatkowo znaczną chwilę zajęło poprawne przypisanie wyników pomiaru czasu wykonywania operacji do zmiennej i otrzymanie czasu w sekundach, a nie cyklach. Poza tym program nie przysporzył kłopotów.

Kluczowe fragmenty kodu

Struktura przechowująca 128-bitowy wektor przechowujący 4 liczby 32-bitowe zmiennoprzecinkowe wygląda następująco:

```
typedef struct Vector128{
    float firstNumber, secondNumber, thirdNumber, fourthNumber;
}Vector128;
```

Zdefiniowano ją w powyższy sposób, żeby przy każdej deklaracji tejże struktury nie używać słowa kluczowego „struct”.

Funkcja generująca liczby pseudo-losowe do wektorów przyjmuję jako argument tablicę wektorów, ponieważ w programie nie przewiduje się używania wektorów pojedynczo. Pętla for-loop wykonuje się 8192 razy, ponieważ tylko taki rozmiar będą przyjmować tablice.

```
void fillArrayOfVectorsWithRandomValues(Vector128* array)
{
    for(int i = 0; i < 8192; i++)
    {
        array[i].firstNumber = rand() % 999;
        array[i].secondNumber = rand() % 999;
        array[i].thirdNumber = rand() % 999;
        array[i].fourthNumber = rand() % 999;
    }
}
```

Do samego losowania liczby używana jest funkcja „rand()” z biblioteki <stdlib.h>. Liczby przyjmują wartości od 0 do 998 przez użycie „% 999”.

Zmienne odpowiedzialne za przechowywanie rozpoczęcia pomiaru, zakończenia pomiaru i całkowitego czasu pomiaru są zmiennymi globalnymi.

```
clock_t startMeasurement, endMeasurement;
double totalTime;
```

Do funkcji dodawania w SISD podane są wskaźniki na zmienne, które mają zostać dodane. Na początku deklarowana jest zmienna, która będzie przechowywać wynik. Następnie zostaje „rozpoczęty" pomiar czasu czyli przypisanie ilości cykli procesora do pierwszej zmiennej. Kolejna instrukcja to wstawka z assemblera, która ładuje pierwszy składnik sumy („%1", ponieważ jest to druga w kolejności zmienna podana w rozszerzonej części wstawki) na stos rejestrów jednostki zmiennoprzecinkowej i dodaje do niego drugi składnik („%2"). Dzięki ograniczeniu „=t" w sekcji output wstawki zapisana zostaje wartość ze szczytu stos w/w rejestrów do zmiennej odpowiadającej za przechowywanie wyniku bez zbędnego użycia instrukcji „fstp". W części input podane są wskaźniki na liczby, które chcemy dodać. Po tej instrukcji znowu zostaje pobrana wartość licznika cykli procesora. Nasz czas końcowy (który zostaje zwrócony przez funkcje) to różnica między drugim, a pierwszym pomiarem zcastowana do zmiennej typu double i podzielona przez stałą „CYCLE_PER_SEC" z biblioteki <time.h> dzięki czemu otrzymujemy czas w sekundach.

```
double additionSISD(float* a, float* b)
{
	float sum;
	startMeasurement = clock();
	asm(
		"fld %1 \n\t"
		"fadd %2 \n\t"
		:"=t"(sum)
		:"m"(*a), "m"(*b)
		);
	endMeasurement = clock();
	totalTime = ((double)(endMeasurement - startMeasurement)) / CLOCKS_PER_SEC;
	return totalTime;
}
```

Inne operacje arytmetyczne w SISD wyglądają analogicznie z tą różnicą, że zamiast instrukcji „fadd" we wstawce umieszczone są tam odpowiednio: „fsub", „fmul" oraz „fdiv".

Funkcje związane z SIMD różnią się od funkcji związanych z SISD dwoma szczegółami:

- Jako argumenty funkcje te przyjmują wektory 128-bitowe, czyli żeby przeprowadzić daną operację na całym wektor wystarczy tylko raz wywalać funkcję a nie 4 razy jak w przypadku SISD. Dodatkowo wynik przechowywany jest w wektorze, a nie zmiennej typu float.
- Wstawką asemblera.

Na początku wstawki pierwszy z wektorów („%1”, ponieważ jest to drugi w kolejności wektor podany w rozszerzonej części wstawki) zostaje umieszczony w rejestrze RAX, następnie przeniesiony do rejestru XMM0. Analogicznie postępuje się z drugim wektorem („%2”), ale on zostaje umiejscowiony w rejestrze XMM1. Następuje dodanie tychże wektorów i aktualizacja zmiennej przechowującej wynik („%0”, ponieważ jest to pierwszy w kolejności wektor podany w rozszerzonej części wstawki). Nazwy rejestrów są poprzedzone dwoma procentami, ponieważ komputer w takiej postaci odczyta to jako np. „%rax”. Ograniczenie „=m” pozwala umieścić wynik w dowolnym miejscu pamięci komputera.

```
double additionSIMD(Vector128* v1, Vector128* v2)
{
	Vector128 sum;
	startMeasurement = clock();
	asm(
		"movq %1, %%rax \n\t"
		"movups (%%rax), %%xmm0 \n\t"
		"movq %2, %%rax \n\t"
		"movups (%%rax), %%xmm1 \n\t"
		"addps %%xmm1, %%xmm0 \n\t"
		"movups %%xmm0, %0 \n\t"
		:"=m"(sum)
		:"m"(v1), "m"(v2)
		:"rax");
	endMeasurement = clock();
	totalTime = ((double)(endMeasurement - startMeasurement)) / CLOCKS_PER_SEC;
	return totalTime;
}
```

Inne operacje arytmetyczne w SIMD wyglądają analogicznie z tą różnicą, że zamiast instrukcji „addps” we wstawce umieszczone są tam odpowiednio: „subps”, „mulps” oraz „divps”.

Ilość powtórzeń pętli pomiarów jest kontrolowana przez zmienną reps. Na początku tablicy wektorów wypełniane są nowymi, pseudo-losowymi wartościami. Następnie wykonują się 4 pętle, każda wykonuje się 2048, 4096 lub 8192 razy (w zależności od wartości numberAmount), które dokonują pomiarów czasu poszczególnych operacji. Funkcje zwracają czas wykonania każdej operacji i wartości te są dodawane do odpowiednich liczników czasu (zadeklarowanych wcześniej). Przy zapisie do pliku zmienne te są dzielone przez reps, żeby otrzymać wartości średnie.

```
for(i = 0; i < reps; i++)
  {
    fillArrayOfVectorsWithRandomValues(vectorArray1);
    fillArrayOfVectorsWithRandomValues(vectorArray2);
    for(j = 0; j < numberAmount; j++)
    {
      avgAdditionTimeSISD += additionSISD(&vectorArray1[j].firstNumber, &vectorArray2[j].firstNumber);
      avgAdditionTimeSISD += additionSISD(&vectorArray1[j].secondNumber, &vectorArray2[j].secondNumber);
      avgAdditionTimeSISD += additionSISD(&vectorArray1[j].thirdNumber, &vectorArray2[j].thirdNumber);
      avgAdditionTimeSISD += additionSISD(&vectorArray1[j].fourthNumber, &vectorArray2[j].fourthNumber);

      avgAdditionTimeSIMD += additionSIMD(&vectorArray1[j], &vectorArray2[j]);
    }
    for(j = 0; j < numberAmount; j++)
    {
      avgSubtractionTimeSISD += subtractionSISD(&vectorArray1[j].firstNumber, &vectorArray2[j].firstNumber);
      avgSubtractionTimeSISD += subtractionSISD(&vectorArray1[j].secondNumber, &vectorArray2[j].secondNumber);
      avgSubtractionTimeSISD += subtractionSISD(&vectorArray1[j].thirdNumber, &vectorArray2[j].thirdNumber);
      avgSubtractionTimeSISD += subtractionSISD(&vectorArray1[j].fourthNumber, &vectorArray2[j].fourthNumber);

      avgSubtractionTimeSIMD += subtractionSIMD(&vectorArray1[j], &vectorArray2[j]);
    }
    for(j = 0; j < numberAmount; j++)
    {
      avgMultiplicationTimeSISD += multiplicationSISD(&vectorArray1[j].firstNumber, &vectorArray2[j].firstNumber);
      avgMultiplicationTimeSISD += multiplicationSISD(&vectorArray1[j].secondNumber, &vectorArray2[j].secondNumber);
      avgMultiplicationTimeSISD += multiplicationSISD(&vectorArray1[j].thirdNumber, &vectorArray2[j].thirdNumber);
      avgMultiplicationTimeSISD += multiplicationSISD(&vectorArray1[j].fourthNumber, &vectorArray2[j].fourthNumber);

      avgMultiplicationTimeSIMD += multiplicationSIMD(&vectorArray1[j], &vectorArray2[j]);
    }
    for(j = 0; j < numberAmount; j++)
    {
      avgDivisionTimeSISD += divisionSISD(&vectorArray1[j].firstNumber, &vectorArray2[j].firstNumber);
      avgDivisionTimeSISD += divisionSISD(&vectorArray1[j].secondNumber, &vectorArray2[j].secondNumber);
      avgDivisionTimeSISD += divisionSISD(&vectorArray1[j].thirdNumber, &vectorArray2[j].thirdNumber);
      avgDivisionTimeSISD += divisionSISD(&vectorArray1[j].fourthNumber, &vectorArray2[j].fourthNumber);

      avgDivisionTimeSIMD += divisionSIMD(&vectorArray1[j], &vectorArray2[j]);
    }
  }
```

Opis uruchomienia programu

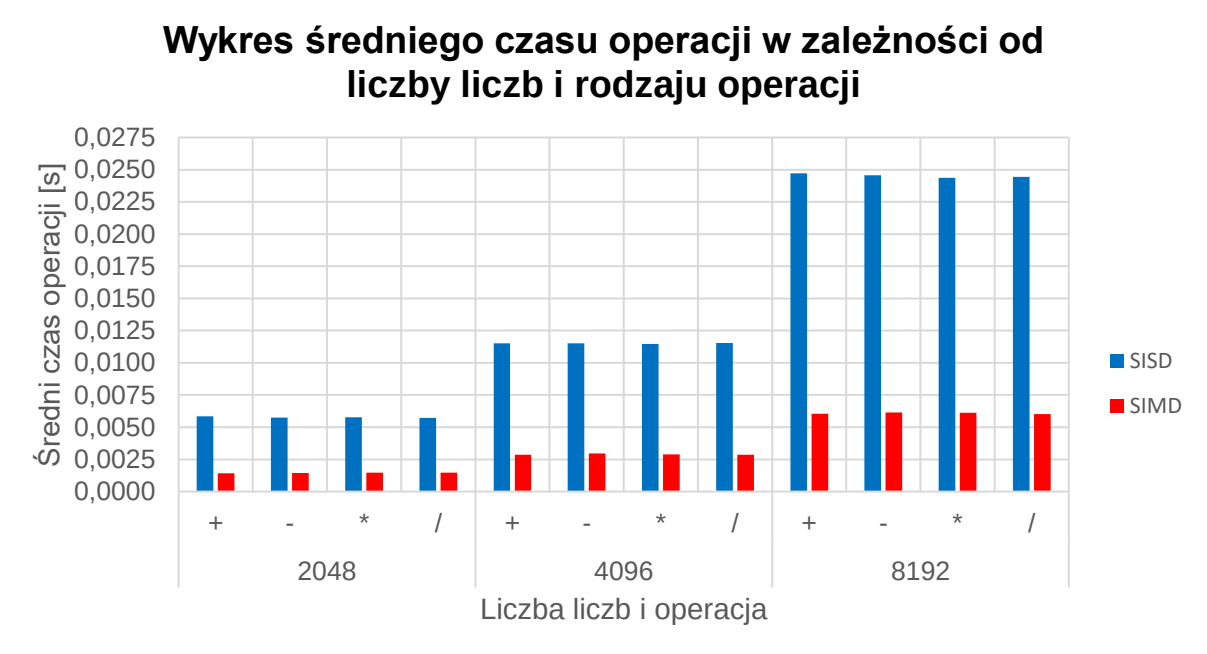
Program został zkompilowany za pomocą kompilatora GCC, pliku makefile i komendy „make”. Dodatkowo użyto opcji „-o” w celu ustalenia konkretnej nazwy pliku output. Zawartość makefile:

```
all:
	gcc 248862_Lab_4.c -o 248862_Lab_4
```

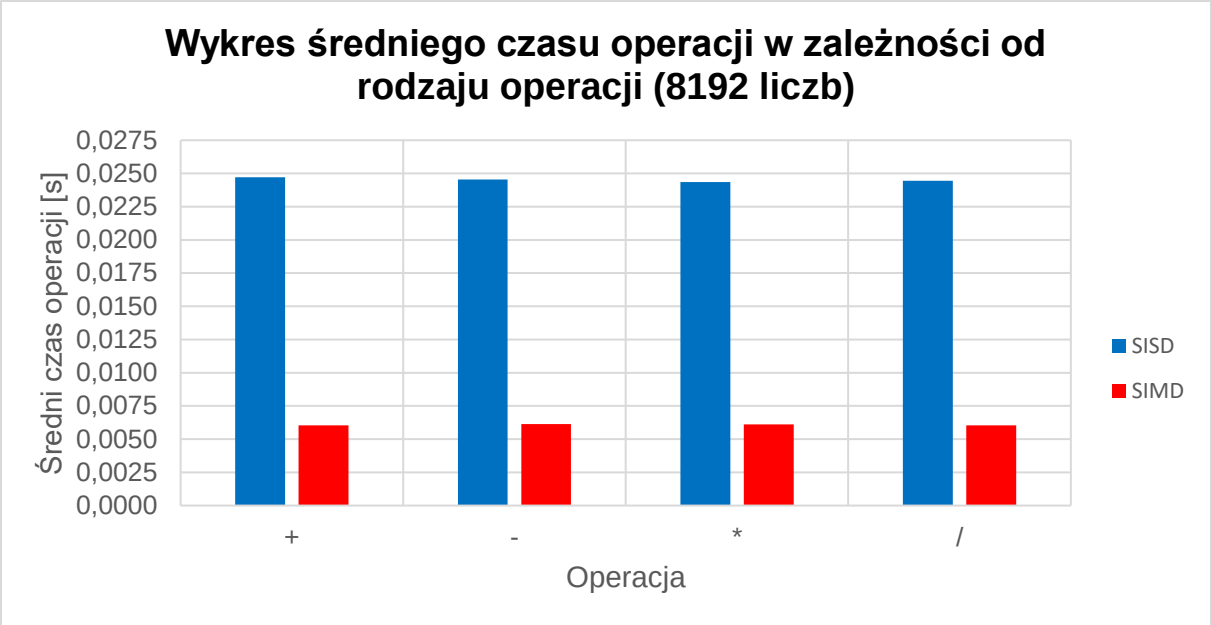
Wykresy i wnioski

Celem Części 1 laboratorium było zapoznanie się z ideą architektury SIMD i instrukcjami SSE, które spełniają założenia tejże architektury. Część 2 zaimplementowano w celu porównania zysku z używania instrukcji SSE.

Przeprowadzano pomiary dla 2048, 4096 i 8192 wektorów w tablicach. Pomiary składały się z 10 powtórzeń, a następnie obliczono średnią arytmetyczną.



Na powyższym wykresie łatwo zauważyć, że wykorzystanie architektury SIMD znacznie skraca czas wszystkich operacji arytmetycznych nie zależnie od ilości liczb. Zysk z użytkowania instrukcji SSE w porównaniu do standardowych instrukcji FPU wynosi średnio 300,2% ((SISD-SIMD)/SIMD) co jest równoznaczne 4-krotnemu zmniejszeniu czasu operacji.



Wykres ten dosadnie pokazuje że, czasy potrzebne na wykonanie poszczególnych operacji są sobie bardzo bliskie i ten stan rzeczy niezależny jest od wybranej architektury przeprowadzania obliczeń.

Poniżej zamieszczam tabelki z wynikami pomiarów czasu operacji i zyskiem z używania mechanizmów SIMD wyrażonym w procentach:

Operacja\Liczba liczb	2048	4096	8192
+	0.00584	0.0115	0.02471
-	0.00575	0.01151	0.02456
*	0.00577	0.01147	0.02436
/	0.00572	0.01153	0.02444

Tabela 1 Czasy wykonywania poszczególnych operacji w zależności od liczby liczb dla SISD

Operacja\Liczba liczb	2048	4096	8192
+	0.00142	0.00286	0.00604
-	0.00143	0.00296	0.00613
*	0.00146	0.00288	0.00612
/	0.00146	0.00287	0.00603

Tabela 2 Czasy wykonywania poszczególnych operacji w zależności od liczby liczb dla SIMD

Operacja\Liczba liczb	2048	4096	8192
+	311.49%	301.61%	309.29%
-	301.26%	288.42%	300.36%
*	295.20%	298.64%	297.97%
/	290.78%	301.95%	305.47%

Tabela 3 Zysk dla poszczególnych operacji w zależności od liczby liczb z wykorzystania mechanizmów SIMD