Andrzej Gierlak, 236411 Wrocław 27.04.2021

Grupa: E07-03e Wtorek, 12:15-15:15 TN

Prowadzący: Mgr inż. Tomasz Serafin

OiAK
Laboratorium 3
ZADANIE W JĘZYKU C++ z wstawkami ASM – kompilacja GCC

1 Przebieg pracy nad programem

Zacząłem od analizy wytycznych i poszukania informacji na temat łączenia języka cpp z asemblerem (w tym również sam język c). Udało mi się przetestować dodawanie liczb całkowitych we wstawce asemblerowej. Następnie przygotowałem potrzebne elementy programu z części cpp; zaimplementowałem odpowiednie biblioteki, przetestowałem działanie generowania liczb oraz zapisu do pliku tekstowego. Chwilkę pochyliłem się nad pytaniem: "jak dodawać wektory w asemblerze", jednak wyszukanie "SIMD" w dokumentacji intela pozwoliło mi na zapoznanie się z instrukcjami:

- MOVUPS
- ADDPS
- SUBPS
- MULPS
- DIVPS

Początkowo ładowałem dane z i do tablicy za pomocą mniejszej, pomocniczej tablicy, jednak po uzyskaniu pierwszych wyników, zobaczyłem, że można bezpośrednio podać tablicę z indeksem, żeby asembler pobierał i wysyłał wyniki do i z tablic dwuwymiarowych. Spodziewałem się wyniku: SIMD znacznie szybsze od SISD.

2 SISD a SIMD

W Taksonomii Flynna wyróżnia się podział na:

- SISD
- SIMD
- MISD
- MIMD

Z bezpośredniego rozwinięcia skrótów mamy: Single Instruction Single Data oraz Single Instruction Multiple Data. Są to typy architektur, jakie może mieć komputer. SISD oznacza "zwyczajne" wykonywanie poleceń po kolei, dla jednej danej naraz, podczas, gdy SIMd pozwala na wykonaniu instrukcji dla wielu danych naraz, a w kontekście obecnego zadania SIMD pozwala na wykonanie działań na wektorach- nie na każdym poszczególnym elemencie z osobna, tylko na wszystkich elementach(ograniczone pamięciowo, dla float wektor może być o rozmiarze 4) wektora, korzystając z równoległości. SIMD jest charakterystyczne dla komputerów wektorowych.

3 Napotkane problemy

Największą trudnością było zrozumieć, jak połączyć kod z asemblera z cpp oraz, jak używać działań typowo z architektury SIMD w asemblerze. Dodatkowo miałem wątpliwości co do pierwszych uzyskanych wyników, więc musiałem poprawić kod: zmieniłem sposób odczytywania i zapisywania

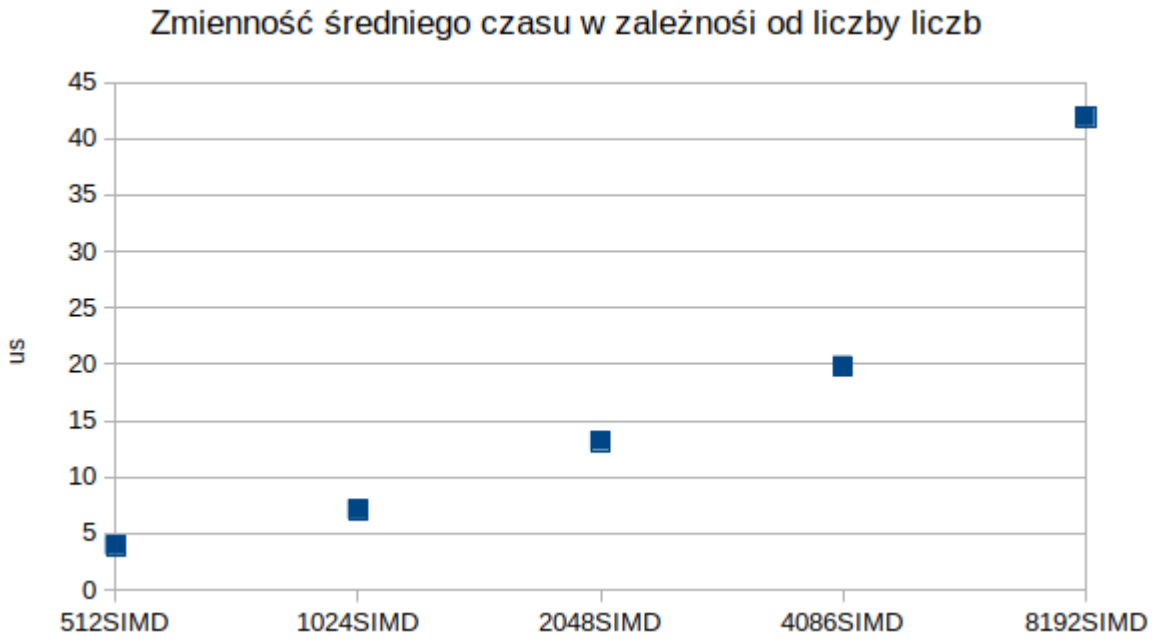
danych oraz sposób iterowania funkcji (nie sprawdza wielokrotnie tego samego warunku if obecnie). Pozostałą wątpliwość odnośnie czasu wykonywania dodawania. Może być to wniosek wynikający z odmienności dodawania od innych działań (mało prawdopodobne), a bardziej prawdopodobne, że pierwsza inicjalizacja danych najbardziej obciąża procesor i stąd widoczna różnica. Problemem od strony technicznej było interpretowanie treści zadania: wykonaj dla 2048 liczb - nie byłem pewien czy chodzi o odrębne liczby, liczby wynikowe, czy zwyczajnie ilość sumowanych 128 bitowych wektorów, więc dodatkowo przeprowadziłem analizę dla dwóch dodatkowych wielkości.

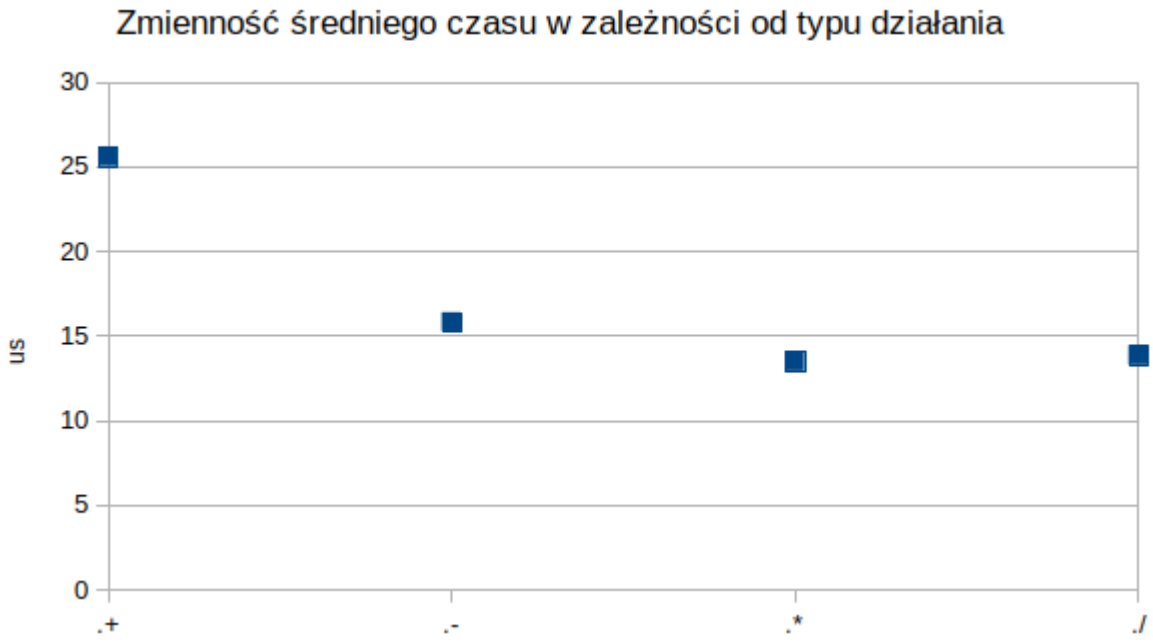
4 Opis uruchomienia programu

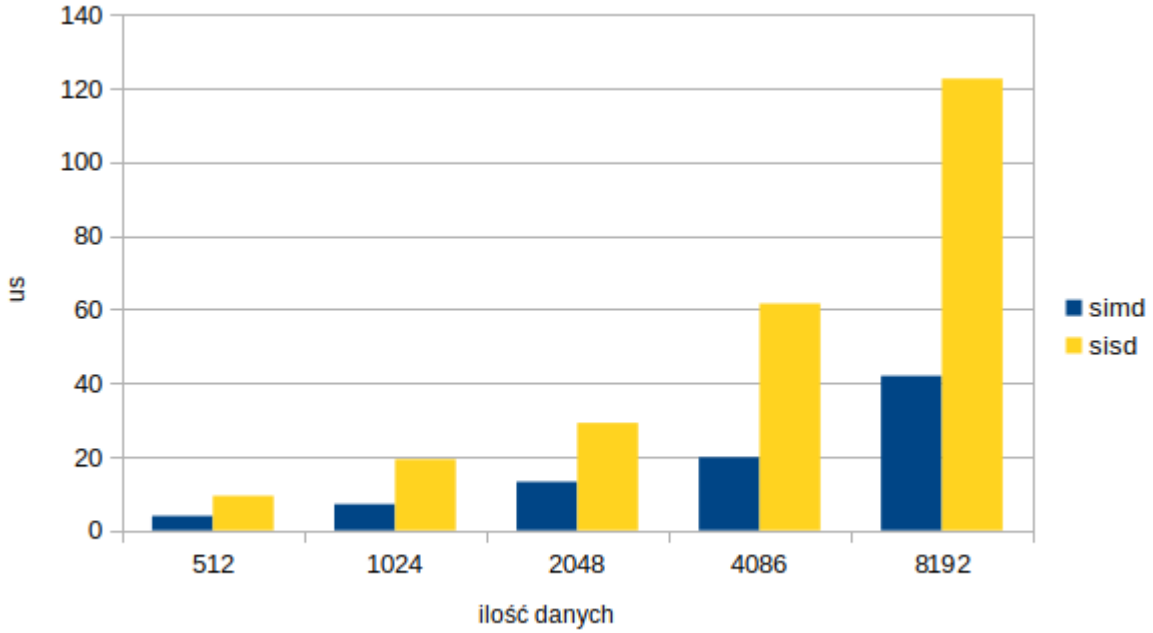
Do kodu załączony jest plik makefile, więc jedynie należy wykonać polecenie make, które utworzy nam plik wykonywalny wektory.out, który uruchamiamy standardowo ./wektory.out . W programie umyślnie nie implementowałem menu z względu na liczne testy i zmiany.

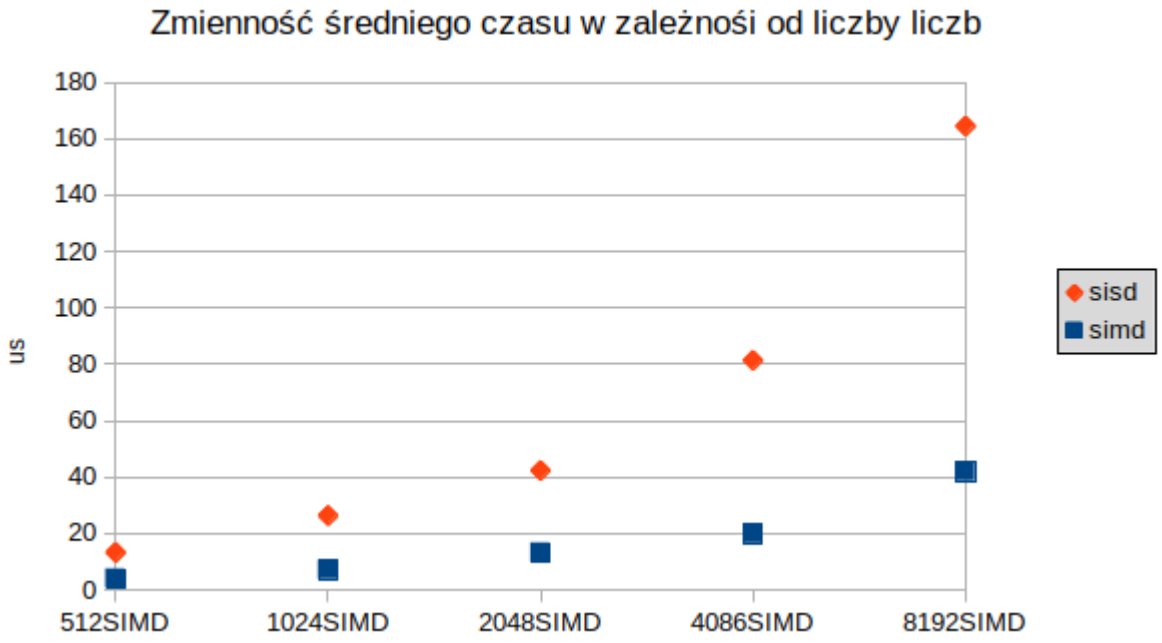
Działaniem programu sterujemy za pomocą 2 zmiennych: storageSize, który określa ilość wektorów wynikowych oraz localSIMD, to wartość typu bool, która określa, czy wykorzystujemy równoległe dodawanie wektorów, czy iterujemy po jednym elemencie z każdego wektora, aby otrzymać wynik.

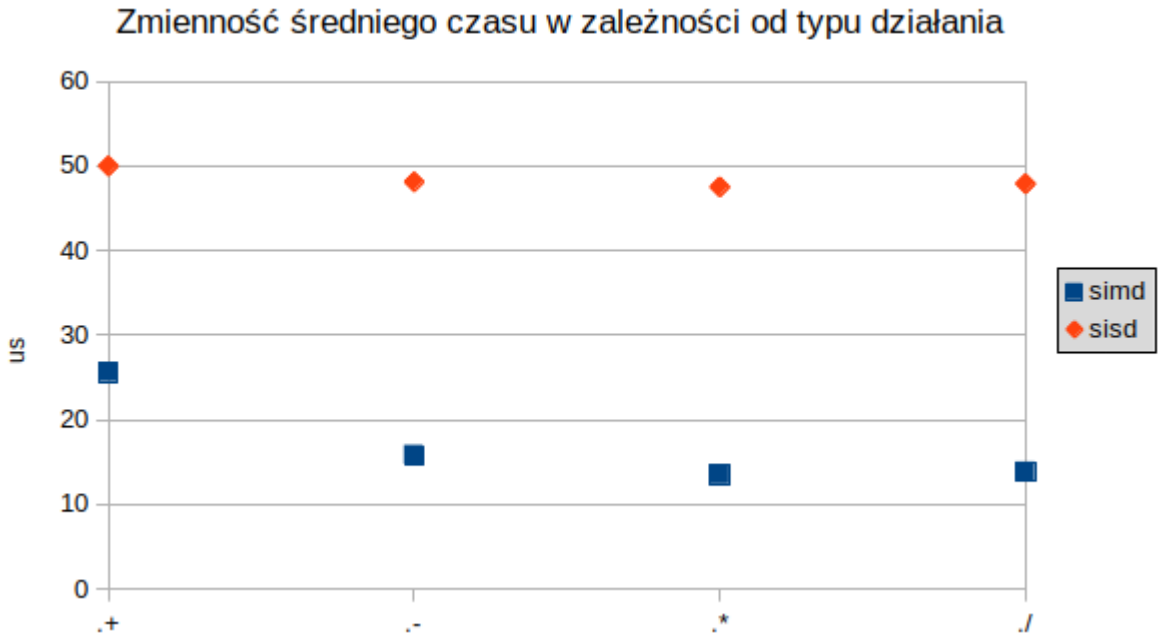
5 Wykresy

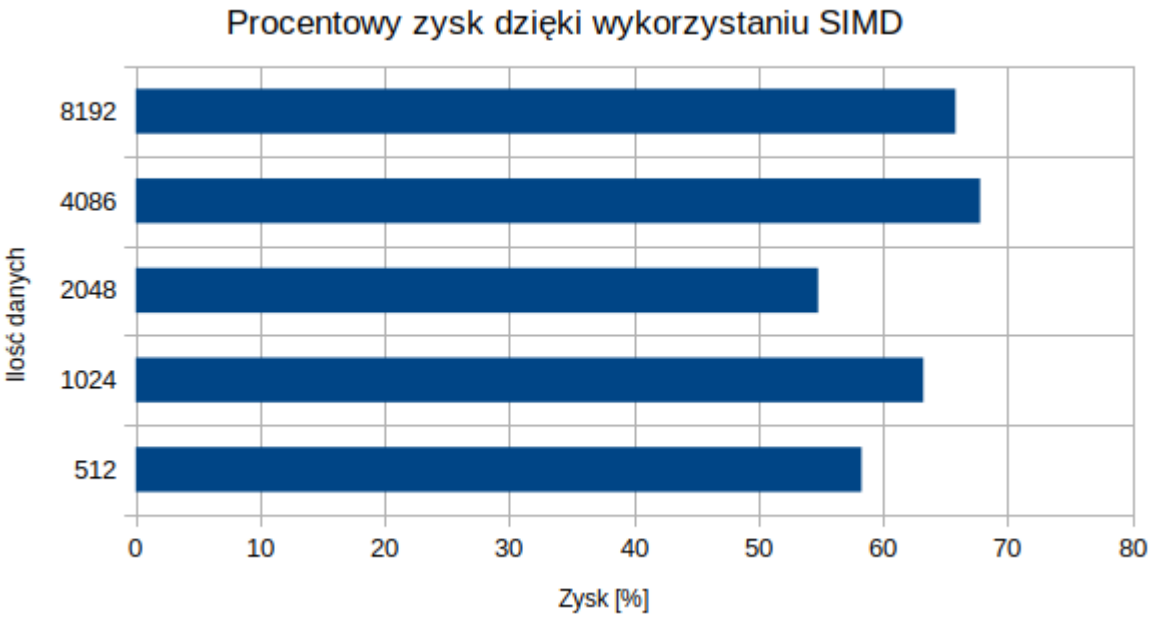



Zmienność średniego czasu w zależności od typu działania
us
30
25
20
15
10
5
0
.+
.-
.*
./


us
140
120
100
80
60
40
20
0
512
1024
2048
4086
8192
ilość danych
simd
sisd


Zmienność średniego czasu w zależnośi od liczby liczb
us
180
160
140
120
100
80
60
40
20
0
512SIMD
1024SIMD
2048SIMD
4086SIMD
8192SIMD
sisd
simd


Zmienność średniego czasu w zależności od typu działania
us
60
50
40
30
20
10
0
.+
.-
.*
./
simd
sisd



6 Kluczowe fragmenty kodu

W programie mamy do czynienia z wykonywaniem działań na wektorach. Możemy to robić na dwa sposoby: używając działań na wektorach podstawowych dla architektury SIMD, lub w "standardowy"(SISD) sposób czyli, każdy element brać do działań po kolejnym.

```
void subSIMD(int index){
    __asm__ __volatile__(
        "MOVUPS (%0), %%xmm0\n"
        "MOVUPS (%1), %%xmm1\n"
        "SUBPS %%xmm1, %%xmm0\n"
        "MOVUPS %%xmm0, (%2)"
        :
        : "r"(storageOfDataA[index]), "r"(storageOfDataB[index]),
        "r" (storageOfResults[index])
    );
}
```

Na przedstawionym fragmencie widzimy funkcję używającą wstawki asmeblerowej, aby wykonywać odejmowanie wektorowe. Analogicznie tak wygląda funkcja w SISD:

```
void subSISD(int index){
    for(int j=0;j<4;j++){
        storageOfResults[index][j]=
        storageOfDataA[index][j]-storageOfDataB[index][j];
    }
}
```

Dane są przechowywane w postaci i inicjowane, jak podano poniżej

```
float storageOfDataA[storageSize][4];
float storageOfDataB[storageSize][4];
float storageOfResults[storageSize][4];
.
.
.
void initializeData(){
    srand(time(NULL));
    for(int i=0;i<storageSize;i++){
        for(int j=0;j<4;j++){
            storageOfDataA[i][j]= (rand() %1000)+1; //chwilowe uproszczenie
            zeby nie bylo dzielenia przez 0
            storageOfDataB[i][j]= (rand() %1000)+1;
        }
    }
}
```

7 Wnioski

Niezależnie od badanego działania, możemy stwierdzić, że sposób przetwarzania danych równolegle jest zdecydowanie i niezaprzeczalnie szybszy od działań architektury SISD. Pomijając błąd wynikający z wczytywania i zapisywania danych, przypuszczałem, że działania na wektorach powinny być szybsze, dzięki używaniu pełnej mocy procesora, czyli wykorzystaniu rejestrów i instrukcji stworzonych z myślą o działaniach na wektorach. Przypuszczam, że przy dokładniejszym pomiarze, jeśli poprawiłbym miejsca pomiaru czasu wykonywania instrukcji, oraz upewnił się, że samo odczytywanie i zapisywanie danych nie jest brane pod uwagę spodziewałbym się zobaczyć zysków postaci około czterokrotnie szybszych obliczeń na korzyść SIMD. Moje badania dowiodły, że SIMD jest szybsze co najmniej dwukrotnie z dużym wskazaniem na większą różnicę.

Drugim, bardziej osobistym wnioskiem było: asembler jest użyteczny i potężny. Dzięki nakładce z języka cpp mogłem swobodnie oprogramować resztę programu, a dzięki wstawkom z asemblera i zapewnianej przez niego kontroli procesora, mogłem znacznie, zauważalnie przyśpieszyć obliczenia.