Politechnika
Wrocławska

W4N

Porównanie efektywności operacji arytmetycznych wektorowych SIMD i zwykłych SISD

Organizacja i Architektura Komputerów

Autor:
Filip Kwiek 280947

Termin zajęć:
Środa np. 9:15

Prowadzący:
Mgr. inż. Tomasz Serafin

Spis treści

1 Wprowadzenie **3**
1.1 Rys historyczny . . . 3
1.2 Operacje SISD i SIMD . . . 3

2 Użyty sprzęt **3**

3 Przebieg pracy nad programem **3**

4 Uruchamianie programu **3**

5 Napotkane problemy **4**

6 Przebieg eksperymentu **4**

7 Operacje SIMD **4**
7.1 Dodawanie: . . . 4
7.2 Odejmowanie: . . . 5
7.3 Mnożenie: . . . 5
7.4 Dzielenie: . . . 5
7.5 Wykresy . . . 6

8 Operacje SISD **6**

9 Wnioski **6**

1 Wprowadzenie

Celem eksperymentu było porównanie efektywności czasowej operacji arytmetycznych wektorowych i niewektorowych.

1.1 Rys historyczny

1.2 Operacje SISD i SIMD

2 Użyty sprzęt

- Procesor: AMD Ryzen 5 3550h
- System Operacyjny: Arch linux
- Kernel: 6.4.14-zen1-2-zen
- Ram: 32Gb DDR4 2666MHZ

3 Przebieg pracy nad programem

Na labolatoriach wprowadzających stworzyłem funkcje odpowiadające za operacje oparte na wektorach, napisane za pomocą wstawek assemblerowych w moich programach pisanych w C. Następnie w domu przez resztę czasu stworzyłem funkcje odpowiadające za operacje SISD, obsługę obu typów operacji, a następnie tworzenie pliku z danymi, które powstały w wyniku badania. Ponadto do obróbki danych powstał skrypt w pythonie, który odpowiada za generację wykresów na podstawie danych z pliku.

4 Uruchamianie programu

Na potrzeby programu powstał makefile, który odpowiada za kompilację wszystkich plików i został stworzony na podstawie zasobów dostępnych w internecie:

```
# Compiler
CXX = g++
# Compiler flags
CXXFLAGS = -Wall -g
# Include paths (add -I flags for header directories if needed)
INCLUDES = -I.
# Target executable
TARGET = main
# For deleting the target
TARGET_DEL = main.out

# Find all source files recursively
SRCS := $(shell find . -name "*.cpp")
# Generate object file paths (preserving directory structure)
OBJS := $(SRCS:.cpp=.o)
# Find all header files recursively (for dependency generation)
HEADERS := $(shell find . -name "*.h")

# Default rule to build and run the executable
all: $(TARGET) run

# Rule to link object files into the target executable
$(TARGET): $(OBJS)
    $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJS)

# Rule to compile .cpp files into .o files with automatic header dependencies
%.o: %.cpp
```

```
	$(CXX) $(CXXFLAGS) $(INCLUDES) -MMD -MP -c $< -o $@

# Include dependency files
-include $(OBJS:.o=.d)

# Rule to run the executable
run: $(TARGET)
	./$(TARGET)

# Clean rule to remove generated files
clean:
	del $(TARGET_DEL) $(OBJS) $(OBJS:.o=.d)
```

W folderze źródłowym, tym samym gdzie znajduje się makefile należy uruchomić polecenie: *make*. Po wpisaniu polecenia, program się uruchomi i wpisze dane do pliku z wynikami pomiarów czasu operacji.

5 Napotkane problemy

Jedyne problemy na jakie się natknąłem podczas pisania programu, to mój własny brak wiedzy, tj. nieznajomość, w jaki sposób należy się odnosić to tablic zapisanych w pamięci w wstawkach assemblerowych, ale po przeczytaniu dokumentacji, przestało to stanowić problem.

6 Przebieg eksperymentu

W obu wypadkach działałem na liczbach zmiennoprzecinkowych, które był generowane pseudolosowo za pomocą następujących funkcji z biblioteki *random*:

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<float> distribution(0, 1000);
```

Następnie mierzyłem czas za pomocą funkcji *std* :: *chrono* :: *high_resolution_clock* :: *now*() z biblioteki *chrono*.

7 Operacje SIMD

7.1 Dodawanie:

```
    double gen_add_vectors(int n)
{
    // data generation
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_real_distribution<float> distribution(0, 1000);
    auto startTime = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < n / 4; i++)
    {
        float vec1[4];
        float vec2[4];

        for (int j = 0; j < 4; j++)
        {
            vec1[j] = static_cast<float>(distribution(gen));
            vec2[j] = static_cast<float>(distribution(gen));
        }
```

```
        __asm__ volatile(
            "movaps (%0), %%xmm0\n"
            "movaps (%1), %%xmm1\n"
            "addps %%xmm1, %%xmm0\n"
            :
            : "r"(vec1), "r" (vec2)
            : "xmm0", "xmm1"
        );
    }

    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration<double, std::milli>(endTime - startTime).count();

    return duration;
}
```

W powyższym kodzie najpier następuje generacja losowych liczb zmiennoprzecinkowych, które następnie wpisywane są do dwóch tablic czterolelementowych. Następnie dodaję te dwa wektory do siebie za pomocą wstawki assemblerowej.

Pozostałe operacje są robione w taki sam sposób różniąc się jedynie wstawkami assemblerowymi.

7.2 Odejmowanie:

```
__asm__ volatile(
    "movaps (%0), %%xmm0\n"
    "movaps (%1), %%xmm1\n"
    "subps %%xmm1, %%xmm0\n"
    :
    : "r"(vec1), "r" (vec2)
    : "xmm0", "xmm1"
);
```

7.3 Mnożenie:

```
__asm__ volatile(
    "movaps (%0), %%xmm0\n"
    "movaps (%1), %%xmm1\n"
    "mulps %%xmm1, %%xmm0\n"
    :
    : "r"(vec1), "r" (vec2)
    : "xmm0", "xmm1"
);
```

7.4 Dzielenie:

```
__asm__ volatile(
    "movaps (%0), %%xmm0\n"
    "movaps (%1), %%xmm1\n"
    "divps %%xmm1, %%xmm0\n"
    :
    : "r"(vec1), "r" (vec2)
    : "xmm0", "xmm1"
);
```

7.5 Wykresy

W wyniku eksperymentu powstały następujące wykresy:

8 Operacje SISD

9 Wnioski