

Piotr Koronczok, 272955  
WT-15

Wrocław, May 31, 2024

prowadzący: dr hab. inż. Tadeusz Tomczak

Laboratorium Wprowadzenia do Wysokowydajnych Komputerów

## Badanie parametrów wydajnościowych dla kodu obliczającego całkę oznaczoną z użyciem jednostki wektorowej z wykorzystaniem równoległości

### Contents

<b>1</b>	<b>Cele oraz zakres ćwiczenia</b>	<b>2</b>
<b>2</b>	<b>Przebieg ćwiczenia</b>	<b>2</b>
2.1	Konstrukcja plików źródłowych . . . . .	3
2.2	Użycie flag kompilatora gcc . . . . .	5
2.3	Sprawdzenie poprawności . . . . .	6
2.4	Specyfikacja maszyny . . . . .	7
2.5	Określenie intensywności arytmetycznej . . . . .	8
2.6	Określenie wydajności - <i>GFLOPS</i> . . . . .	10
2.7	Określenie IPC . . . . .	12
2.8	Porównanie z danymi katalogowymi . . . . .	12
2.9	Porównanie z wersją x87 . . . . .	13
<b>3</b>	<b>Podsumowanie</b>	<b>13</b>

## 1 Cele oraz zakres ćwiczenia

Cele ćwiczenia:

- implementacja całki oznaczonej z wykorzystaniem jednostki wektorowej oraz równoległości dla funkcji:

$$f(x) = \frac{x^4 - 1}{1 - 3x}$$

- wykonanie pomiarów *rdtsc* oraz *perf*
- optymalizacja i ponowne pomiary
- porównanie z danymi katalogowymi *IPC* oraz *GFLOPS*
- porównanie z wersją używającą x87 do wykonywania obliczeń

## 2 Przebieg ćwiczenia

Elementami składowymi ćwiczenia były pliki:

- *calka.c* - program główny; zawiera implementację funkcji obliczającej całkę, funkcje pomocnicze (np. rzeczywistą funkcję dla której liczona jest całka) oraz zawiera kod wywołujący funkcje do odczytu licznika *tsc* (Time-Stamp Counter)
- *func.c* - zawiera dwie funkcje (serializującą oraz nie) odczytujące licznik *tsc*

Pliki kompilowane i linkowane były przy użyciu kompilatora *GNU* - *gcc*.

## 2.1 Konstrukcja plików źródłowych

Do wytworzenia kodu wykorzystującego jednostkę wektorową została przyjęta koncepcja manipulacji opcjami kompilatora *gcc* w celu uzyskania porządkanych efektów przy niezmienionym, w porównaniu do poprzednich laboratoriów (liczenie całki przy użyciu x87), programie głównym w języku *C*.

Listing 1: Program główny napisany w języku *C*.

```
#include <stdio.h>

unsigned long func(); // deklaracje funkcji
unsigned long func_s(); // do odczytu licznika tsc

void set_cw(); // ustawienie rejestru kontrolnego

// funkcja do obliczania wartosci zadanej funkcji
double funkcja(double x)
{
    return (((x * x * x * x) - x) / (1 - x * 3));
}

// funkcja do obliczania calki oznaczona
double calka(double a, double b)
{
    double x, y, wynik = 0.0;
    int n = 1000000000;

    double dx = (b - a) / n;
    x = a;

    for (int i = 0; i < n; i++)
    {
        y = (((x * x * x * x) - x) / (1 - x * 3));
        wynik += (dx) * y;
        x += dx;
    }

    return wynik;
}

int main()
{
    // set_cw(); // ustawienie sowa kontrolnego dla x87 w celu porownania z in
```

```

double a = 1.0, b = 2.0; // granice przedziału
double wynik = 0.0;      // wynik całki

unsigned long start = func_s();

wynik = calka(a, b);

unsigned long end = func_s();

printf("Wynik: %0.17g ", wynik);
printf("rdtsc: %lu\n", end - start);

return 0;
}

```

Zawiera m.in. funkcję *main*, deklaracje funkcji odczytujących licznik *tsc* oraz właściwą implementację obliczeń. Wylczenie całki odbywa się zaleconą metodą prostokątów.

$$\int_a^b f(x)dx \cong \sum_{x_i} f(x_i)dx$$

Kod programu nie jest został zmieniony w porównaniu do wersji z zajęć laboratoryjnych, na których używany był koprocesor x87 do obliczeń zmiennoprzecinkowych. W celu zapewnienia odpowiedniego czasu wykonania oraz minimalizacji istotności narzutów dostosowane zostały jedynie ilości iteracji pętli wywołujących funkcję *calka* oraz tej odpowiedzialnej za obliczanie sum poszczególnych prostokątów.

Listing 2: Funkcje do odczytu *tsc* z poziomu assemblera

```

.global func, func_s # zapewnienie linknerowi widoczności funkcji
.text                # sekcja kodu
func:
    push %ebp        # zapamiętanie na stosie rejestrow
    mov %esp, %ebp   # przez funkcję wywoływana
    rdtsc             # odczytanie tsc to %edx:%eax
    leave            # przywrócenie rejestrow
    ret              # powrót do miejsca wywołania

func_s:
    push %ebx        # zapisanie %ebx na stosie
    xor %eax, %eax    # zerowanie %eax
    cpuid             # zapewnienie, że wcześniejsze instrukcje są wykonane
    rdtsc             # odczytanie tsc do %edx:%eax
    pop %ebx          # przywrócenie rejestru
    ret

```

## 2.2 Użycie flag kompilatora gcc

Kluczowym celem ćwiczenia było wykorzystanie przez program jednostki wektorowej SIMD ( w tym przypadku AVX-512) oraz przetwarzania wartości spakowanych (*packed*). Na zapewnienie tej oraz innych własności programu pozwoliło użycie następujących opcji gcc:

- -m32 - generacja kodu dla architektury 32-bitowej
- -o - zapewnienie pliku wyjściowego
- -S - wytworzenie pliku *.s*; jedynie kompilacja
- -no-pie - wyłączenie tworzenia plików w formacie *PIE*, użycie jej jest reakcją na ostrzeżenie linkera: *"warning: creating DT\_TEXTREL in a PIE"*
- -O1 - ustawienie poziomu optymalizacji kompilatora
- -march=native - poinformowanie kompilatora, aby generował kod zoptymalizowany pod architekturę używanego procesora, w tym przypadku jest to *x86\_64*
- -ffast-math - ustawienie różnych ustawień obliczeń zmiennoprzecinkowych; pomimo braku przestrzegania dokładnej implementacji zasad IEEE 754 flaga została użyta z uwagi na poprawność działania programu dla różnych granic całkowania
- -ftree-vectorize - pozwala na wektoryzację kodu

Listing 3: Wytworzenie kodu *calka.s* dla jednostki wektorowej

```
gcc -no-pie -m32 -O1 -march=native -ffast-math  
-ftree-vectorize -S calka.c func.s
```

Listing 4: Tworzenie pliku wykonywalnego

```
gcc -no-pie -m32 -o calka calka.s func.s
```

Listing 5: Tworzenie pliku wykonywalnego dla koprocatora x87 do porównania

```
gcc -no-pie -m32 -mno-sse -o calka calka.c func.s
```

W ramach optymalizacji zostały podjęte próby:

- rozwijania pętli przy użyciu `#pragma GCC unroll`
- rozwijania pętli przy użyciu flagi `-funroll-loops`
- zwiększenia poziomu optymalizacji kompilatora - flagi `-O2` oraz `-O3`

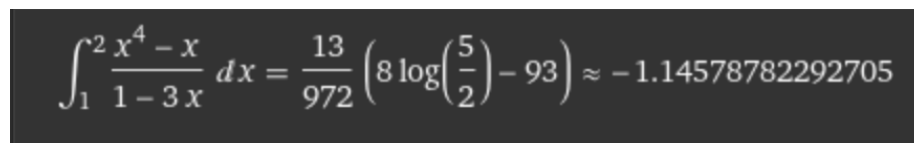
Nie poprawiło to czasu wykonania algorytmu, a nawet w przypadku użycia opcji wyższego poziomu optymalizacji niż pierwotnie przyjęty `-O1` wydłużyło go o około 1.5%. Z uwagi na ten fakt analizowany dalej kod nie był ulepszany.

## 2.3 Sprawdzenie poprawności

Sprawdzenie poprawności wykonywanego kodu odbyło się poprzez wywołanie funkcji obliczającej całkę i porównanie z wynikiem programu wolframalfa


Całka oznaczona na przedziale  $x \in < 1, 2 >$ :

$$\int_1^2 \frac{x^4 - x}{1 - 3x} dx$$



The image shows a screenshot of the WolframAlpha interface. It displays the integral  $\int_1^2 \frac{x^4 - x}{1 - 3x} dx$  and its result:  $\frac{13}{972} \left( 8 \log\left(\frac{5}{2}\right) - 93 \right) \approx -1.14578782292705$ .

Figure 1: Wynik programu wolframaplha



The image shows a screenshot of a program's output. It displays the text "Wynik: -1.1457878206563348 rdtsc: 1883029575".

Figure 2: Wynik programu całka przy użyciu jednostki wektorowej

## 2.4 Specyfikacja maszyny

Pomiary zostały wykonane na procesorze firmy *AMD* - AMD Ryzen™ 5 7640U. Pełna specyfikacja dostępna pod adresem strony producenta. System operacyjny to Ubuntu 22.04.4 LTS. Wykonanie *lscpu* pozwoliło na wyświetlenie najważniejszych informacji o maszynie.

```
(base) piotr@piotr-framework:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:         48 bits physical, 48 bits virtual
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list:   0-11
Vendor ID:             AuthenticAMD
Model name:            AMD Ryzen 5 7640U w/ Radeon 760M Graphics
CPU family:            25
Model:                 116
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             1
Stepping:              1
Frequency boost:       enabled
CPU max MHz:           6466.7959
CPU min MHz:           1600.0000
BogoMIPS:              6987.03
```

Figure 3: Parametry maszyny

## 2.5 Określenie intensywności arytmetycznej

$$I = \frac{N_A}{S_{MEM}}$$

$N_A$  - liczba wykonanych operacji arytmetycznych  
 $S_{MEM}$  - rozmiar przesłanych danych

Intensywność arytmetyczna dominowana jest przez pętlę *for*, w której to sumowane są pola poszczególnych prostokątów pozwalających na obliczenie przybliżonego pola pod krzywą. Operacje wykonywane w jej wnętrzu widoczne są w kodzie assemblerowym 4.

```
int n = 1000000000;  
  
for (int i = 0; i < n; i++)  
{  
    y = funkcja(x);  
    wynik += (dx) * y;  
    x += dx;  
}
```

$$N_A = n \cdot 80$$

$$S_{MEM} = 0B$$

Gdzie  $n$  to ilość iteracji pętli, dla badanego kodu  $n = 1000000000$ .

$$I = \frac{N_A}{S_{MEM}} = \frac{n \cdot 80FLOP}{n \cdot 0B} = \infty \frac{FLOP}{B}$$

Przy danej formie pętli głównej4 intensywność arytmetyczna dąży do  $\infty$ , a operacje wykonywane są jedynie na rejestrach.



```

.L4:
    vmovapd %zmm2, %zmm1
    vaddpd  %zmm5, %zmm2, %zmm2
    vmulpd  %zmm1, %zmm1, %zmm0
    vmulpd  %zmm1, %zmm0, %zmm0
    vaddpd  %zmm7, %zmm0, %zmm0
    vmulpd  %zmm1, %zmm0, %zmm0
    vmulpd  (%esp), %zmm1, %zmm1
    vsubpd  %zmm1, %zmm6, %zmm1
    vdivpd  %zmm1, %zmm0, %zmm0
    vmulpd  %zmm4, %zmm0, %zmm0
    vaddpd  %zmm0, %zmm3, %zmm3
    incl    %eax
    cmpl    $125000000, %eax
    jne     .L4

```

Figure 4: Główna pętla sumująca poszczególne prostokąty w pliku calka.s - zapewnienie o użyciu działań na reprezentacjach spakowanych

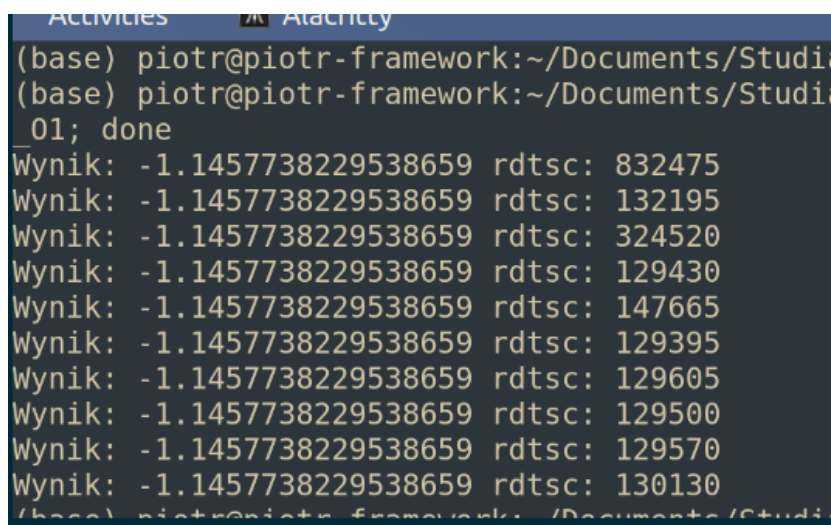
## 2.6 Określenie wydajności - *GFLOPS*

Podczas zajęć laboratoryjnych pomiary zostały wykonane na kodzie, w którym:

- funkcja całka z kodu w  $C$  wywoływana była 100000 razy
- pole pod krzywą było podzielone na 10000 podprzedziałów

$$n_{total} = 10000 \cdot 100000 = 1000000000$$

Uzyskane zostały następujące wyniki odczytów rdtsc:



```
Activities Alacritty
(base) piotr@piotr-framework:~/Documents/Studia
(base) piotr@piotr-framework:~/Documents/Studia
_01; done
Wynik: -1.1457738229538659 rdtsc: 832475
Wynik: -1.1457738229538659 rdtsc: 132195
Wynik: -1.1457738229538659 rdtsc: 324520
Wynik: -1.1457738229538659 rdtsc: 129430
Wynik: -1.1457738229538659 rdtsc: 147665
Wynik: -1.1457738229538659 rdtsc: 129395
Wynik: -1.1457738229538659 rdtsc: 129605
Wynik: -1.1457738229538659 rdtsc: 129500
Wynik: -1.1457738229538659 rdtsc: 129570
Wynik: -1.1457738229538659 rdtsc: 130130
(base) piotr@piotr-framework:~/Documents/Studia
```

Figure 5: Wyniki pomiarów uzyskane podczas zajęć

Po odrzuceniu błędów grubych wyniki są na poziomie 130000 cykli, podczas gdy ten sam kod skompilowany do wersji używającej *x87* generował wyniki na poziomie 1300000000. Wskazywałoby to na około 10000 krotne przyspieszenie. Po głębszej analizie kodu wytworzonego przez wersję z zajęć laboratoryjnych udało się zlokalizować źródło tego zachowania.

Błąd pojawił się podczas optymalizacji kompilatora *gcc*, który usunął "niepotrzebne" wywołania funkcji *calka*, a więc w rzeczywistości była wykonana ona jedynie raz.

W celu usunięcia tego problemu oraz zastosowania odpowiedniego czasu wykonania badanego kodu funkcja obliczająca całkę oznaczoną wywoływana jest raz dla  $n$  podprzedziałów, daje to  $n_{total} = 1000000000 \cdot iteracji$ .

```

call    calka
addl    $16, %esp
fstpl   -32(%ebp)
movl    $1000000, %eax
.L8:
decl    %eax
jne     .L8

```

Figure 6: Problematyczny fragment kodu calka.s z zajęć laboratoryjnych

Listing 6: Wywołanie procesu z użyciem taskset  
`taskset -c 1 ./calka`

odczyty rdtsc	czas [s]
935803925	0.21268
935310810	0.21257
936724110	0.21289
942123770	0.21412
929726385	0.21130
935890830	0.21270
941201835	0.21391
936535425	0.21285
941080245	0.21388
934200540	0.21232
936499865	0.21284
936185040	0.21277
936965295	0.21295
936512220	0.21284
940228940	0.21369
934756410	0.21244
937669880	0.21311
936543615	0.21285
936672450	0.21288
936795160	0.21291

Table 1: Pomiary czasów wykonanie przy dwukrotnym uruchomieniu programu dziesięciokrotnie dla częstotliwość taktowania 4.4 GHz

Po weryfikacji wyników i odrzuceniu błędów grubych:

$$\overline{czas} = 0.212773s$$

$$\frac{FLOP}{s} = \frac{125000000 \cdot 80}{0.212773} = 46998444351 FLOPS = 46.998 GFLOPS$$

## 2.7 Okreśnię IPC

Użycie programu *perf* pozwoliło na zbadanie liczby instrukcji na cykl.

Listing 7: Użycie programu *perf*

```
sudo perf stat taskset -c 1 ./calka
```

```

wynik: -1.1457878206563348 rdtsc: 937712930
Performance counter stats for 'taskset -c 1 ./calka':

    269.59 msec task-clock                #   0.997 CPUs utilized
         4 context-switches              #    14.837 /sec
         1 cpu-migrations                 #     3.709 /sec
        122 page-faults                   #   452.543 /sec
1,175,529,803 cycles                      #    4.360 GHz                (82.45%)
   76,706 stalled-cycles-frontend        #    0.01% frontend cycles idle (83.68%)
   233,265 stalled-cycles-backend        #    0.02% backend cycles idle (83.69%)
1,749,518,444 instructions                #    1.49 insn per cycle
                                                # 0.00 stalled cycles per insn (83.68%)
   125,475,670 branches                   #   465.436 M/sec              (83.68%)
     5,502 branch-misses                  #    0.00% of all branches     (82.82%)

0.270313366 seconds time elapsed

0.266327000 seconds user
0.003975000 seconds sys

```

Figure 7: Wynik *perf stat* dla badanego algorytmu programu. Pomiar wykonany pięciokrotnie.

$$IPC = \frac{1749769395}{1176198903} = 1.49$$

## 2.8 Porównanie z danymi katalogowymi

Cykli na iteracje pętli:

$$\frac{cykl}{iter} = \frac{1177223997}{125000000} = 9.417$$

Instruction	Op1	Op2	Op3	APM Vol	Cpuid flag	Ops	Unit	Latency	Throughput
VMOVAPD	zmm1	zmm2		4	AVX512	1		0	6
VADDPD	zmm1	zmm2	zmm3/mem512/mem64bcst	4	AVX512	1	FP2/3	3	1
VMULPD	zmm1	zmm2	zmm3/mem512/mem64bcst	4	AVX512	1	FP0/1	3	1
VSUBPD	zmm1	zmm2	zmm3/mem512/mem64bcst	4	AVX512	1	FP2/3	3	1
VDIVPD	zmm1	zmm2	zmm3/mem512/mem64bcst	4	AVX512	1	FP1	13.5	0.1-0.2

Figure 8: Dane katalogowe z *Software Optimization Guide for the AMD Zen4 Microarchitecture*, rev. 1.0, 57647.

## 2.9 Porównanie z wersją x87

Użycie programu *pref* pozwoliło na określenie parametrów kodu w wersji *x87*.

```

Wynik: -1.1457879583967987 rdtsc: 1393923676
Performance counter stats for 'taskset -c 1 ./calka':

      5,317.77 msec task-clock                #    1.000 CPUs utilized
         25      context-switches           #    4.701 /sec
          1      cpu-migrations              #    0.188 /sec
        120      page-faults                #   22.566 /sec
23,387,439,259 cycles                       #    4.398 GHz                    (83.30%)
      448,954   stalled-cycles-frontend     #    0.00% frontend cycles idle   (83.30%)
      1,226,215 stalled-cycles-backend     #    0.01% backend cycles idle    (83.31%)
24,004,225,027 instructions                 #    1.03 insn per cycle
                                     # 0.00 stalled cycles per insn   (83.38%)
      1,002,024,528 branches                 # 188.430 M/sec                   (83.38%)
        46,954   branch-misses              #    0.00% of all branches       (83.33%)

      5.319044161 seconds time elapsed

      5.318726000 seconds user
      0.000000000 seconds sys

```

Figure 9: Wynik pref stat dla x87

$$czas_{x87} = 5.32s$$

$$przyspieszenie = 19.8$$

	czas [s]	L. inst.	L. cykli	IPC	przyspieszenie
x87	5.31	24004225027	23387439259	1.03	1
AVX	0.268	1751943618	1178438825	1.49	19.8

Table 2: Porównanie wersji x87 oraz wektorowej AVX

## 3 Podsumowanie

Wykonane ćwiczenie pozwoliło na wykorzystanie jednostki wektorowej do obliczenia całki oznaczonej poprzez odpowiednie manipulacje flagami kompilatora *gcc*. W porównaniu do wersji kodu z użyciem koprocessora *x87* widoczne jest znaczne polepszenie czasu wykonania.