lab_11 - Instrukcja do ćwiczenia

Teoria:

http://galaxy.agh.edu.pl/~amrozek/AK/lab12.pdf

Pomiar czasu wykonania:

Pomiar czasu wykonania pewnego fragmentu kodu (np. funkcji) dokonywany jest poprzez odpowiednie użycie dwóch funkcji zawartych w pliku **eval_time.c**: **init_time** i **read_time**. Funkcja **init_time** powinna być wywołana bezpośrednio przed kodem, którego czas wykonania chcemy mierzyć, zaś funkcja **read_time** bezpośrednio po nim. Przykładowe użycie obu funkcji wygląda następująco:

```
#include <stdio.h>
#include "eval_time.h"
void main(void)
{
      double times[3];

      init_time();
      some_function();
      read_time( times );
      printf("T0=%lf T1=%lf T2=%lf\n",times[0],times[1], times[2] );
}
```

Czas **T0** jest czasem spędzonym na poziomie systemu operacyjnego (realizacja funkcji systemowych), **T1** to czas spędzony na poziomie użytkownika, zaś **T2** jest łącznym czasem jaki upłynął od rozpoczęcia pomiaru do jego zakończenia (tzw. czas zegarowy).

$$T0 + T1 \leq T2$$

Pojedynczy pomiar obarczony jest znacznym błędem ze względu na działanie w systemie wielozadaniowym, więc aby uzyskać miarodajne wyniki należy:

- powtórzyć pomiar kilka-kilkanaście razy,
- mierzyć wyłącznie kod nie zawierający wywołań funkcji systemowych,
- wykorzystywać jedynie czas **T1**.

Optymalizacja kodu:

Wykorzystywany kompilator (**gcc**) pozwala na znaczącą optymalizację generowanego kodu, ale uzyskane rezultaty (czasy działania) zależą w znacznym stopniu od sprzętu (generacji procesora, częstotliwości zegara, wielkości pamięci RAM, wielkości pamięci cache, itp.). Poziom optymalizacji ustalany jest na etapie kompilacji kodu źródłowego przy pomocy opcji **Ox**, gdzie **x** jest liczbą z zakresu **[0..3]** – im większa wartość, tym wyższy poziom optymalizacji

(**ale nie musi to przekładać się to na szybsze działanie kodu!**). Poziom **0** oznacza brak optymalizacji.

Praktyka (lab_11.c):

Działania:

1. Przygotowujemy kod do mierzenia czasów wykonania zawarty w **eval_time.c**.
2. C (Compile) – polecenie: **gcc –O3 –c eval_time.c**
3. Przechodzimy do programu **lab_11.c** – jest to zmodyfikowana wersja programu **mat_mat.c**, a wprowadzone zmiany dotyczą zwiększenia rozmiarów tablic (macierzy) wykorzystywanych w mnożeniu, wykorzystania algorytmu blokowego mnożenia macierzy oraz możliwości dokonania wszystkich pomiarów w trakcie pojedynczego uruchomienia programu.
4. CL (Compile, Link) – polecenie: **gcc –O0 –o lab_11 lab_11.c eval_time.o**
5. R (Run) – polecenie: **./lab_11**
6. Przykladowe efekty uzyskane po uruchomieniu wyglądają następująco:

```
N=128   naive   =   0.010 s  GFLOPS = 0.411
N=128   unroll4 =   0.004 s  GFLOPS = 0.932
N=128   b_4     =   0.007 s  GFLOPS = 0.604
N=128   b_8     =   0.006 s  GFLOPS = 0.731
N=128   b_16    =   0.005 s  GFLOPS = 0.765
N=128   b_32    =   0.005 s  GFLOPS = 0.814
N=128   b_64    =   0.005 s  GFLOPS = 0.808
N=256   naive   =   0.047 s  GFLOPS = 0.715
N=256   unroll4 =   0.043 s  GFLOPS = 0.784
N=256   b_4     =   0.056 s  GFLOPS = 0.595
N=256   b_8     =   0.047 s  GFLOPS = 0.714
N=256   b_16    =   0.045 s  GFLOPS = 0.740
N=256   b_32    =   0.045 s  GFLOPS = 0.745
N=256   b_64    =   0.044 s  GFLOPS = 0.761
N=256   b_128   =   0.053 s  GFLOPS = 0.637
N=512   naive   =   0.413 s  GFLOPS = 0.650
N=512   unroll4 =   0.347 s  GFLOPS = 0.774
N=512   b_4     =   0.479 s  GFLOPS = 0.561
N=512   b_8     =   0.427 s  GFLOPS = 0.628
N=512   b_16    =   0.402 s  GFLOPS = 0.668
N=512   b_32    =   0.398 s  GFLOPS = 0.675
N=512   b_64    =   0.465 s  GFLOPS = 0.577
N=512   b_128   =   0.498 s  GFLOPS = 0.539
N=512   b_256   =   0.475 s  GFLOPS = 0.566
N=1024  naive   =   7.480 s  GFLOPS = 0.287
N=1024  unroll4 =   3.587 s  GFLOPS = 0.599
N=1024  b_4     =   4.174 s  GFLOPS = 0.514
N=1024  b_8     =   3.490 s  GFLOPS = 0.615
N=1024  b_16    =   3.261 s  GFLOPS = 0.659
N=1024  b_32    =   3.257 s  GFLOPS = 0.659
N=1024  b_64    =   3.657 s  GFLOPS = 0.587
N=1024  b_128   =   3.736 s  GFLOPS = 0.575
```

```
N=1024  b_256   =   3.765 s  GFLOPS = 0.570
N=1024  b_512   =   6.423 s  GFLOPS = 0.334
```

7. Powtarzamy kompilację zwiększając poziom optymalizacji poprzez **O1** do **O3** i porównujemy uzyskane wyniki. Generalnie czasy wykonania powinny się zmniejszać ze wzrostem poziomu optymalizacji, ale w niektórych przypadkach (testowany kod+konfiguracja sprzętowa) wynik mogą się nie zmieniać lub mogą się pogorszyć.
8. Testujemy jeszcze jeden poziom optymalizacji: **-Ofast** – nie jest on zalecany, gdyż w pewnych warunkach mogą się pojawić błędy w obliczeniach zmiennoprzecinkowych.
9. Jeżeli testowany program działa z sensowną prędkością (algorytm **naive** dla **N=1024** trwa poniżej **20**-**30** sekund), to zwiększamy górną granicę pętli **for** w funkcji **main** z **1024** na **2048** (taka zmiana spowoduje wydłużenie czasu mniej więcej **8x**). W przypadku uruchamiania programu na słabszym sprzęcie (albo maszynie wirtualnej) lepiej poprzestać na wartości **1024**.
10. Wracamy do poziomu optymalizacji **–O3**.
11. CL (Compile, Link) – polecenie: **gcc –O3 –o lab_11 lab_11.c eval_time.o**
12. Uruchamiamy program kilkukrotnie (im więcej tym lepiej w kontekście wiarygodności i powtarzalności) – aby ułatwić sobie dalsze prace można wyniki zapisywać do plików tekstowych:

```
./lab_8b   >  result_1.txt
./lab_8b   >  result_2.txt
      …
./lab_8b   >  result_n.txt
```

13. Wykorzystujemy zgromadzone dane (po odrzuceniu tych, które znacznie się różnią od pozostałych) do szczegółowej analizy i prezentacji w formie tabeli/wykresu) – pełne wyniki (nowe algorytmy) pojawią się na kolejnych zajęciach i dopiero wtedy będzie można dokonać pełnego porównania metod!.