Laboratorium 12. Pomiar wydajności algorytmów mnożenia macierzy, pamięć cache

Teoria - na podstawie książki: J.W. Demmel Applied Numerical Linear Algebra, MIT, 1996

Czas wykonania programu sekwencyjnego obliczeniowego (bez obsługi urządzeń I/O, przerwań itp.) można oszacować na podstawie liczby operacji zmiennoprzecinkowych (*f*), liczby transferów danych (*m*) z wolnej pamięci operacyjnej do szybkiej pamięci podręcznej (ew. rejestrów):

$$t = t_{arithm} \cdot f + t_{mem} \cdot m \,.$$

W przeprowadzonej - uproszczonej - analizie zakładamy:

- stały czas wykonania wszystkich operacji arytmetycznych (t_{arithm} - w przypadku dodawania i mnożenia w większości procesorów jest stały lub „porównywalny"),

- minimalną liczbę niezbędnych operacji arytmetycznych (tzn. nie da się tu już nic poprawić – nie może ich już być mniej),

- stały, niezależny od algorytmu / sposobu napisania programu (pobieranie danych sekwencyjne lub ze skokami) czas przesyłania danych z pamięci głównej do *cache* (t_{mem}) - o tym o później…

- pomijalny czas obsługi pętli (modyfikacja liczników, porównania, skoki warunkowe itp.).

Przy takich założeniach można wykazać, że czas wykonania programu zależy od stosunku liczby operacji zmiennoprzecinkowych do liczby transferów danych:

$$t = t_{arithm} \cdot f + t_{mem} \cdot m = t_{arithm} \cdot f \cdot \left(1 + \frac{m \cdot t_{mem}}{f \cdot t_{arithm}}\right) = t_{arithm} \cdot f \cdot \left(1 + \frac{t_{mem}}{q \cdot t_{arithm}}\right),$$

gdzie: $q = f / m$ jest **wskaźnikiem wydajności**, mówiącym ile operacji arytmetycznych przypada na jeden dostęp do pamięci (ile użytecznej pracy można wykonać w stosunku do przenoszenia danych z i do pamięci).

Czysto teoretycznie – im większe q, tym program może się szybciej wykonać. W najlepszym przypadku, gdyby wszystkie dane znajdowały się w rejestrach ($m = 0$ - pamięć niepotrzebna), czas wykonania wyniósłby $t_{arithm} * f$.

Przykład: dodanie do siebie dwóch dużych (niemieszczących się w pamięci *cache*, lub komputer nie ma *cache*) macierzy kwadratowych.

Każda z macierzy o wymiarach n x n zawiera n^2 elementów i trzeba wykonać tyle samo operacji dodawania.
Trzeba odczytać odpowiadające sobie pary elementów w obu macierzach ($2n^2$ odczytów pamięci), dodać (n^2 operacji arytmetycznych) i ponownie zapisać, co daje łącznie $3n^2$ dostępów do pamięci.

Wskaźnik wydajności wynosi: $q = n^2 / 3n^2 = 1/3$, czyli na jedno dodawanie przypadają trzy dostępy do pamięci operacyjnej.

Jednocześnie należy pamiętać że: $t_{mem} >> t_{arithm}$ (czas cyklu pracy jądra procesora jest wielokrotnie krótszy od pełnego czasu dostępu do pamięci dynamicznej RAM). Przyśpieszenie taktowania samego procesora nie przyniesie zatem znaczącego wzrostu wydajności obliczeń, gdyż ta jest zdominowana przez częstotliwość magistrali procesor-pamięć i czas dostępu do pamięci.

Typowy – naiwny algorytm mnożenia macierzy (*Double precision General Matrix Multiply - DGEMM*)

wykonujący działanie:

$$\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B} \ ,$$

wygląda tak:

```
void dgemm_naive (int n, double* A, double* B, double* C)
{
int i,j,k;
double cij;

for (i=0; i<n; ++i)
    for (j=0; j<n; ++j)
    {
      cij = C[i+j*n];                  // cij = C[i][j]
      for(k=0; k<n; ++k )
          cij += A[i+k*n] * B[k+j*n];  // cij += A[i][k]*B[k][j]
      C[i+j*n] = cij;                  // C[i][j] = cij
    }
}
```

gdzie n jest rozmiarem zadania czyli liczbą rzędów/kolumn - zakładamy, że macierze są kwadratowe. Zakładamy również, że przestrzeń adresowa komputera jest liniowa, a tablice w programie – jednowymiarowe. Kolejne elementy macierzy dwuwymiarowych przechowywane są wierszami (*row major order*) – jak na poniższym schemacie:

$$\begin{matrix} & i \rightarrow \\ \begin{matrix} j \\ \downarrow \end{matrix} & \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \\ & \mathbf{C} \end{matrix} += \begin{matrix} & i \rightarrow \\ \begin{matrix} k \\ \downarrow \end{matrix} & \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \\ & \mathbf{A} \end{matrix} \bullet \begin{matrix} & k \rightarrow \\ \begin{matrix} j \\ \downarrow \end{matrix} & \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \\ & \mathbf{B} \end{matrix}$$

Pobieżna analiza algorytmu:

W pętli wewnętrznej (k) obliczany jest iloczyn skalarny (*dot product*) i-tej kolumny macierzy **A** i j-tego wiersza macierzy **B**. Dwie wewnętrzne pętle (j-k) wykonują mnożenie wektora (i-tej kolumny **A**) przez macierz **B**.

Oszacowanie liczby dostępów do (wolnej) pamięci operacyjnej:

W uproszczeniu zakładamy, że typowy komputer posiada niewielką szybką, statyczną pamięć *cache* i dużą, wolną, dynamiczną pamięć główną – i to liczba dostępów do tej ostatniej będzie miała wpływ na wydajność algorytmu.

Ponadto, w tym miejscu interesuje nas tylko czy pewna porcja danych „zmieści się” w *cache*. Na razie nie wnikamy w sposób sterowania tej pamięci oraz pobierania i ułożenia w niej danych (w kolejnych komórkach lub „porozrzucane” w dużych odstępach) – chociaż wiadomo, że *cache* jest ładowany całymi liniami wypełnionymi sąsiadującymi elementami…

Dostęp do elementów macierzy **C** nie zależy od indeksu k. Odczyt elementu *Cij* można wykonać przed wejściem w pętlę wewnętrzną, a jego zapis – po ukończeniu tej pętli. W czasie wykonywania pętli k, akumulowana wartość *Cij* powinna być przechowywana w rejestrze. W sumie wykonuje się $2n^2$ dostępów do macierzy **C** (dwie operacje: zapis i odczyt, dwie pętle (i-j) wykonywane n razy).

Elementy macierzy **B** są odczytywane – niestety – n^3 razy (para pętli j-k wykonywana jest n razy w pętli zewnętrznej i).

Jeżeli zadanie (n) jest na tyle małe (albo pamięć *cache* na tyle duża), żeby pomieścić wszystkie (niesąsiadujące ze sobą!), raz odczytane elementy i-tej kolumny macierzy **A** – dostęp do niej odbędzie się n^2 razy. Jeśli rozmiar zadania jest duży i nie jesteśmy w stanie utrzymać w *cache* całej i-tej kolumny **A** – trzeba liczyć się z n^3 dostępów.

Wskaźniki wydajności q prezentują się następująco:

$$q = \frac{f}{m} = \frac{2n^3}{n^3 + 3n^2} \approx 2$$ - jeśli możemy przechowywać całą kolumnę macierzy **A** w *cache*,

$$q = \frac{f}{m} = \frac{2n^3}{2n^3 + 2n^2} \approx 1$$ - jeżeli zadanie jest za duże aby utrzymać całą kolumnę **A** w *cache*.

Liczba operacji arytmetycznych $f = 2n^3$: dodawanie (akumulacja) i mnożenie wykonywane są w każdej iteracji (złożoność również $O(n^3)$). Ponownie na tym etapie nie interesują nas szczegóły techniczne: mnożenie i akumulacja są podstawowymi operacjami w obliczeniach macierzowych. Większość obecnie produkowanych CPU, (GP)GPU i DSP posiada dedykowane jednostki wykonawcze i instrukcje FMA (*Fused Multiply Addition*) pozwalające wykonać wykonać w/w operacje w jednym etapie, ze zminimalizowaną liczbą zaokrągleń i konwersji typów danych.

Jedną z modyfikacji klasycznego algorytmu mnożenia macierzy jest **algorytm blokowy**
(funkcja *dgemm_blocked* w pliku do ćwiczeń)

Ideą jest podział dużych macierzy **A**, **B** i **C** o wymiarach n na b_c kwadratowych bloków, których rozmiar (n_b - liczba rzędów/kolumn) jest podwielokrotnością (dla prostoty) rozmiaru całego zadania. Jednocześnie rozmiar bloków musi być tak dobrany, aby trzy takie fragmenty macierzy: $\mathbf{A}_b$, $\mathbf{B}_b$ i $\mathbf{C}_b$ **zmieściły się w pamięci *cache*.**

$$\underset{\mathbf{C}}{\begin{bmatrix} \overset{\mathbf{C}_b}{\boxed{\begin{matrix}0 & 1\\ 4 & 5\end{matrix}}} & \boxed{\begin{matrix}2 & 3\\ 6 & 7\end{matrix}} \\ \boxed{\begin{matrix}8 & 9\\ 12 & 13\end{matrix}} & \boxed{\begin{matrix}10 & 11\\ 14 & 15\end{matrix}} \end{bmatrix}} += \underset{\mathbf{A}}{\begin{bmatrix} \overset{\mathbf{A}_b}{\boxed{\begin{matrix}0 & 1\\ 4 & 5\end{matrix}}} & \boxed{\begin{matrix}2 & 3\\ 6 & 7\end{matrix}} \\ \boxed{\begin{matrix}8 & 9\\ 12 & 13\end{matrix}} & \boxed{\begin{matrix}10 & 11\\ 14 & 15\end{matrix}} \end{bmatrix}} \bullet \underset{\mathbf{B}}{\begin{bmatrix} \overset{\mathbf{B}_b}{\boxed{\begin{matrix}0 & 1\\ 4 & 5\end{matrix}}} & \boxed{\begin{matrix}2 & 3\\ 6 & 7\end{matrix}} \\ \boxed{\begin{matrix}8 & 9\\ 12 & 13\end{matrix}} & \boxed{\begin{matrix}10 & 11\\ 14 & 15\end{matrix}} \end{bmatrix}}$$

Obliczenia - blok po bloku - wykonywane są takim samym algorytmem jak omówiony poprzednio (funkcja *block*). Całkowita liczba operacji zmiennoprzecinkowych nie uległa zmianie ($2n^3$). Pełna procedura zawarta została w pliku źródłowym i (chyba) nie wymaga większego komentarza. Ponadto została również dobrze opisana w rozdziale 5.4 książki:

D. A. Patterson, J. L. Kennedy, Computer Organisation and Design, The Hardware/Software Interface, Fifth edition, Morgan Kaufmann, 2014.

Oszacowanie liczby dostępów do pamięci, przy założeniu, że w pamięci *cache* mieszczą się trzy bloki $\mathbf{A}_b$, $\mathbf{B}_b$ i $\mathbf{C}_b$, a skokami w adresowaniu kolejnych elementów się nie przejmujemy:

podobnie jak w algorytmie naiwnym, dostęp do bloku $\mathbf{C}_b$ nie zależy od k – mamy w jego przypadku n^2 odczytów i n^2 zapisów (*cache* ↔ pamięć główna). Dla $\mathbf{A}_b$ i $\mathbf{B}_b$ liczby dostępów wynoszą $b_c{*}n^2$. Teoretycznie, wydajność powinna więc wprost zależeć od wymiaru bloku n_b:

$$q = \frac{f}{m} = \frac{2n^3}{2n^2 + 2b_c n^2} = \frac{n}{1 + b_c} \approx \frac{n}{b_c} = n_b$$

Maksymalny wymiar bloku jest jednak ograniczony rozmiarem pamięci *cache*, ponadto muszą się w niej jednocześnie zmieścić trzy takie bloki. Wymagany rozmiar pamięci podręcznej to:
$mem = 3 \cdot n_b^2 \cdot sizeof(data_type)$,
a optymalnym rozmiarem bloku wydaje się* być:

$$n_b \le \sqrt{\frac{mem}{3 \cdot sizeof(data_type)}},$$

np. trzy bloki złożone z 32 x 32 elementów typu *double* zajmą 24 kB pamięci *cache* (typowy współczesny CPU x86-64 ma 32 - 64 kB pamięci danych *cache* L1).

Tak zdefiniowana – teoretyczna - wydajność algorytmu blokowego wynosi:

$$q = \sqrt{\frac{mem}{3 \cdot sizeof(data_type)}} \; .$$

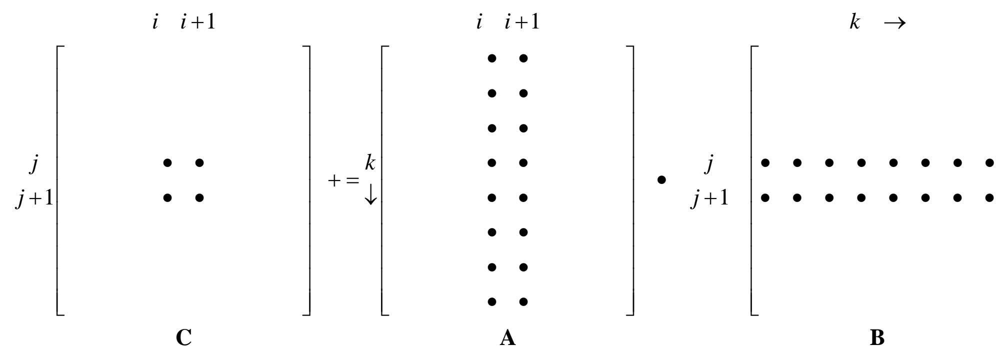
*W praktyce – **proszę to sprawdzić!** Poza *cache* L1, procesor posiada dużo pojemniejsze pamięci podręczne L2 i L3, z drugiej strony oprócz przetwarzanych fragmentów macierzy, w *cache* przeładowywać się będą również inne dane (w tym systemu operacyjnego i innych programów).

Rozwijanie pętli – loop unrolling (funkcja *dgemm_unroll4* w pliku do ćwiczeń)

Ta prosta technika optymalizacji szybkości wykonania programu w przypadku algorytmu *GDEMM* jest kolejnym wariantem „blokowania".

Stosując czterokrotne rozwinięcie, w wewnętrznej pętli (k) nie jest obliczany jeden element (iloczyn skalarny), ale cztery. Mnożone są dwupasmowe macierze (dwie kolumny pierwszej, dwa rzędy drugiej) dające w wyniku blok 2 x 2.

Wszystkie cztery elementy bloku wynikowego w czasie wykonywania pętli (k) są akumulowane nie w pamięci RAM, a bezpośrednio w (szybkich) rejestrach procesora (*%XMM* – wchodzących w skład rozszerzenia wektorowego SSE). Następnie, po zakończeniu pętli (k) zapisywane są w odpowiednich miejscach macierzy **C**.



Korzyścią czasową jest **dwukrotne użycie** (w pętli k, patrz kod źródłowy) raz załadowanych elementów macierzy **A** i macierzy **B**. Teoretyczna wydajność algorytmu powinna być (można udowodnić i sprawdzić doświadczalnie…) dwukrotnie większa od wersji naiwnej.

Dodatkowo; pętle i i j są wykonywane ze skokiem 2 (odpowiadającym rozmiarowi obliczanego bloku), tym samym całkowita liczba iteracji spadła czterokrotnie – proporcjonalnie do stopnia rozwinięcia pętli. Dzięki temu procesor ma do wykonania czterokrotnie mniej instrukcji związanych z obsługą pętli (porównań i skoków warunkowych).

Jaki jest z tego zysk czasowy we współczesnych procesorach superskalarnych to już inna sprawa: instrukcje „obsługujące" pętle mogą być wykonywane równolegle z arytmetyką zmiennoprzecinkową, w innych jednostkach wykonawczych. Dodatkowo, CPU stosuje obecnie makro-fuzję typowych instrukcji sterujących pętlami *for* (*cmp-jne*, *dec-jnz*), posiada układ *branch prediction* oraz wewnętrzny bufor instrukcji z detekcją krótkich pętli (a tutaj takie mamy). Wszystkie instrukcje objęte pętlą mogą więc się zmieścić w takim buforze – procesor wtedy nie musi nawet pobierać instrukcji z pamięci *cache*.

Wszystkie opisane tu algorytmy wykonują $2n^3$ operacji zmiennoprzecinkowych i mają jednakową złożoność $O(n^3)$.

Zadanie 1. Pomiar wydajności algorytmów mnożenia macierzy.

1. Sprawdzić parametry procesora na stanowisku gdzie będą przeprowadzane testy. Nie tylko jego typ (dokładnie-pełny numer!) i częstotliwość taktowania, ale również rozmiary i organizację pamięci *cache* L1-L3.

Linux:

```
lscpu
cat /proc/cpuinfo
```

Windows: np. program CPU-Z

2. Zmierzyć czas (CPU użytkownika) oraz osiąganą liczbę operacji zmiennoprzecinkowych podwójnej precyzji na sekundę (FLOPS – *FLoating Operations Per Second*, w praktyce Mega/Giga FLOPS) dla algorytmów:

- naiwnego (*dgemm_naive*)

- blokowego (*dgemm_blocked*)

- z rozwinięciem pętli (*dgemm_unroll4*)

dla rozmiarów macierzy o rozmiarze (n) 128, 256, 512, 1024 (ew. 2048).

W przypadku algorytmu blokowego dla każdego rozmiaru macierzy sprawdzić podział na bloki o rozmiarach:
n_b = 4, 8, 16, 32, 64, 128 (dla uproszczenia zakładamy n mod $n_b = 0$).

Proszę pamiętać, że praca przebiega w systemie wielozadaniowym. Czas i dla każdego przypadku należy zmierzyć kilkukrotnie i wyznaczyć średnią. Wyniki znacząco odbiegające od średniej – odrzucić (np. wystąpiło wywłaszczenie procesu, obsługa przerwań itp.).

Program *mat_mat.c* zawiera trzy w/w funkcje i przykład użycia-pomiaru. Można go **dowolnie przebudować** w celu ułatwienia sobie zadania i automatyzacji pomiarów.

Program, jak i bibliotekę do pomiaru czasu kompilować z włączoną optymalizacją –O3

Wyniki przedstawić na wykresach typu: GFLOPS *vs* rozmiar zadania (n) - w sposób umożliwiający łatwe porównanie wydajności różnych algorytmów dla różnej wielkości zadań. W podobny sposób przedstawić wydajność algorytmu blokowego dla różnej wielkości zadań i fragmentacji.

Spróbować oszacować optymalny rozmiar bloku dla użytego procesora (rozmiaru *cache*).

Zadanie 2. Usunięcie „skoków” w przestrzeni adresowej podczas odczytu kolejnych elementów macierzy.

Problem do tej pory pomijany (tzn. w tej instrukcji - na wykładzie został omówiony):

przedstawione algorytmy w wewnętrznych pętlach (k) mnożą elementy kolumn(y) macierzy **A** z odpowiadającymi elementami wiersza/wierszy macierzy **B**. W zależności od sposobu mapowania dwuwymiarowej macierzy w jednowymiarowej, liniowej przestrzeni adresowej, dostęp do kolejnych elementów jednej macierzy będzie odbywał się w sposób sekwencyjny: element po elemencie, a do drugiej ze „skokiem” równym rozmiarowi macierzy.

Przypomnienie: podczas ładowania z pamięci jednego elementu pobierany jest do pamięci podręcznej cały blok danych (łącznie z „sąsiedztwem”) o długości linii *cache* – 64 bajty, co przy wyrównaniu danych do granicy długości linii, umożliwia jednoczesne pobranie ośmiu kolejnych liczb typu *double* lub szesnastu typu *float*.

Ponieważ przyjęto (tj. w języku C/C++ i w programie) ułożenie elementów wierszami (*row major order*), kolejne elementy macierzy **B** będą pobierane z sąsiadujących ze sobą lokalizacji w pamięci (w zależności od typu danych: co 4 lub 8 bajtów). Po jednorazowym załadowaniu jednej linii *cache* użyte do obliczeń zostaną

wszystkie przechowywane w niej liczby (tj. wykonując jeden „wolny” dostęp do pamięci RAM pobieramy do *cache*, w zależności od typu, osiem lub szesnaście kolejnych elementów macierzy, które to już mogą być dostarczone do CPU dużo szybciej).

Inaczej sprawa wygląda z dostępem do macierzy **A,** gdzie kolejne elementy są pobierane spod adresów różniących się o: $k * n *$ (4 lub 8) bajtów. Odstęp ten, w praktycznych zastosowaniach, może wielokrotnie przekraczać długość linii *cache*. W takim przypadku mimo jednoczesnego pobrania do szybkiej pamięci podręcznej kilku sąsiednich liczb, wykorzystana do obliczeń zostanie tylko jedna. Kolejne elementy macierzy **A** będą za każdym razem pobierane z wolnej pamięci operacyjnej, a pamięć *cache* będzie niepotrzebnie zbyt często przeładowywana nową, w większości niewykorzystaną w odpowiednim czasie zawartością.

Zadanie właściwe:

Usunąć „skoki” w przestrzeni adresowej, występujące przy odczycie kolejnych elementów macierzy **A** w omówionych poprzednio wersjach algorytmów mnożenia macierzy. Np. przetransponować macierz*: zmienić sposób przechowywania elementów z *row major order* na *column major order*, odpowiednio modyfikując jej adresowanie w miejscu odczytu.

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \qquad \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

`row major order` `column major order`

Po modyfikacji i **sprawdzeniu poprawności** wyników powtórzyć testy wydajności jak w zadaniu 2. Porównać wyniki pomiarów z osiągami analogicznych wersji algorytmu sprzed modyfikacji.

* alternatywnie, w tym konkretnym przypadku: zadaniu czysto testowym, można od razu w odpowiedni sposób wypełnić macierz wartościami początkowymi.

Zadanie 3. Szesnastokrotne rozwinięcie pętli.

Procesor ma szesnaście rejestrów wektorowych – *%XMM0 … %XMM15*.

Wzorując się na funkcji *dgemm_unroll4* rozwinąć pętlę 16 razy – czyli tak, aby w każdej iteracji wewnętrznej pętli (*k*) mnożone były 4 kolumny i 4 wiersze, a wynikiem był blok 4 x 4.

Sprawdzić poprawność wyniku, czas obliczeń oraz wydajność modyfikacji w porównaniu z metodą naiwną, *dgemm_unroll4* i algorytmem blokowym o rozmiarze 4.

Usunąć „skoki” w przestrzeni adresowej, występujące przy odczycie kolejnych elementów macierzy **A**. Sprawdzić wpływ modyfikacji na czas obliczeń i wydajność algorytmu.

Zadanie 4 - dla chętnych.

Przeanalizować wygenerowany kod w asemblerze funkcji z rozwinięciem 16x z zadania 3. (`gcc -S mat_mat.c -O3`). W jaki sposób kompilator sobie poradził z przechowywaniem szesnastu elementów? Czy liczba rejestrów *%XMM* jest wystarczająca, czy wartości były jednak zapisywane do pamięci (odkładane na stos)?
Porównać czas obliczeń i kod asemblera funkcji *dgemm_blocked*, np. liczbę dostępów do pamięci dla różnych poziomów automatycznej optymalizacji np. –O0 i –O3. „Co się stało” z funkcją *block*?