Laboratorium 13. Wektoryzacja i pomiar wydajności algorytmów mnożenia macierzy.

Podstawowe algorytmy algebry liniowej wykonują powtarzające się operacje (np. mnożenie, dodawanie) na kolejnych, sąsiadujących elementach tablic (macierzy, wektorów) umieszczonych w pamięci komputera.

Wpływ sposobu przechowywania danych w pamięci (*row major order*, *column major order*), sposobu dostępu do nich (skokowy, sekwencyjny, blokowy) – i tym samym wpływ wykorzystania pamięci podręcznej na wydajność obliczeń był badany w ćwiczeniu nr 12.

Operacje macierzowe są powszechnie wykorzystywane w różnych wielu dziedzinach: grafika 3D, cyfrowe przetwarzanie sygnałów – np. dźwięku (nie utożsamiać obliczeń macierzowych tylko z przybliżonym rozwiązywaniem równań np. modeli zjawisk fizycznych). Projektanci procesorów zaczęli zatem wyposażać je w elementy architektury SIMD (*Single Instruction Multiple Data*) – umożliwiającej jednoczesne przetwarzanie pojedynczą instrukcją (lub jednym programem – jednym ciągiem instrukcji) wielu „strumieni" danych*.

Model przetwarzania SIMD stosowany był od dawna – początkowo w sprzęcie profesjonalnym (węzły obliczeniowe komputerów np. masowo równoległych), lub specjalizowanym: do wykonywania określonych zadań – np. akceleratory graficzne. Później trafił również do tanich procesorów ogólnego zastosowania.

Pierwsze, jedynie całkowitoliczbowe, implementacje SIMD w architekturze x86 (np. MMX – *MultiMedia eXtension*) tutaj pominiemy. Obecnie, każdy procesor zgodny z x86-64 posiada rozszerzenia wektorowe minimum SSE2 (*Streaming SIMD Extension*), na które składają się, m.in.:

- szesnaście 128 bitowych rejestrów *%XMM0* - *%XMM15*,
- zestaw instrukcji: do transferu i konwersji danych, arytmetyczno-logicznych, porównujących itp.
- odpowiednia liczba (zależna do mikroarchitektury procesora) jednostek wykonawczych,
- rejestr kontrolny z flagami wyjątków (podobnie jak w FPU x87),
- możliwość wykonywania obliczeń na typach zarówno zmiennoprzecinkowych i całkowitoliczbowych.

- Idea wektoryzacji polega na jednoczesnym pobieraniu z pamięci nie jednej liczby (skalar), lecz **całego wektora** liczb: w rejestrze 128-bitowym pomieszczą się np. dwie zmienne typu *double* lub cztery typu *float*.

- Następnie, procesor wykonując **jedną instrukcję** np. arytmetyczną przeprowadza stosowne operacje na **wszystkich elementach** wektorów liczb przechowywanych w „długich" rejestrach.

- Zapis wyników przebiega podobnie do ładowania: **jedna instrukcja zapisu** przenosi do pamięci **wiele elementów** (ew. jedną liczbę typu: suma elementów wektora, minimalny, maksymalny element wektora).

Przykład:

Każdy z rejestrów *%XMM0* i *%XMM1* (o długości 128 bitów) zawiera cztery liczby typu *float* (32 bitowe):

127...96	95...64	63...32	31...0		127...96	95...64	63...32	31...0	*bity*
1.24	10.03	7.24	15.60	,	2.01	14.23	2.50	10.20	.

Wykonując jedną operację dodawania: *ADDPS %XMM0* , *%XMM1* otrzymamy w *%XMM1*:

1.24	10.03	7.24	15.60	*XMM*0
2.01	14.23	2.50	10.20	*XMM*1
+ - - - - - - - - - - - - - - -				.
3.25	24.26	9.74	25.80	*XMM*1

* zajmujemy się tutaj jednym rdzeniem procesora. Współczesny, złożony komputer z procesorem wielordzeniowym (komp. wieloprocesorowy) może realizować różne modele przetwarzania. Wg klasyfikacji Flynna jako „całość" powinien być rozpatrywany jako maszyna MIMD (*Multiple Instruction Multiple Data*) z pamięcią wspólną.

Uwaga: końcówka *PS* rozkazu *ADD* oznacza: P – *Packed* – „spakowane” w jednym rejestrze cztery liczby pojedynczej precyzji (*Single Precsion*), analogicznie dla dwóch sPakowanych liczb typu *Double* rozkaz miałby końcówkę *PD*. Dotyczy to również omówionych w dalszej części rozkazów typu *intrinsic*.

Wektorowe jednostki wykonawcze najczęściej wykonują podstawowe operacje arytmetyczne: + - * /, 1/x, obliczania min., maks. i pierwiastka kwadratowego. W przeciwieństwie do FPU x87, nie mają implementowanych funkcji trygonometrycznych lub logarytmicznych.

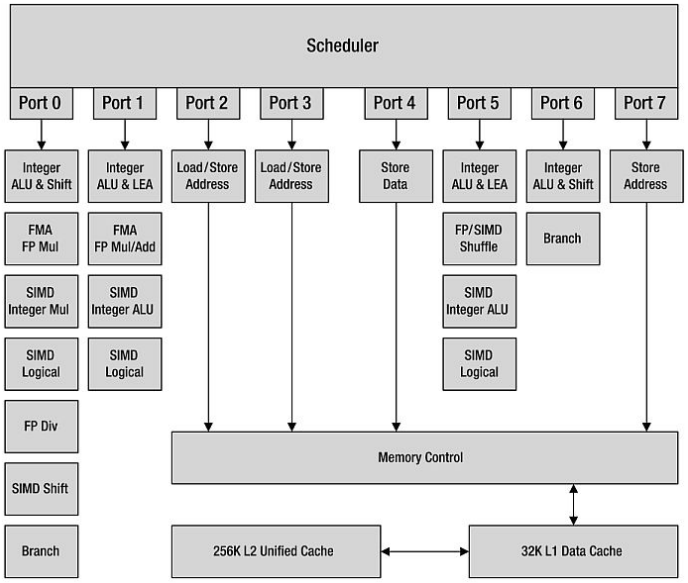
Zaletą jednostek wektorowych, wbudowanych we współczesne procesory jest natomiast ich wysoka optymalizacja, szybkość oraz sama ich liczba: z reguły kilka w każdym rdzeniu.

Przykładowo:

- możliwość wykonywania dominujących w algorytmach macierzowych operacji mnożenie-akumulacja: „*Fused Multiply-Add*” (FMA): $a = (b * c) + d$, ew. $a += b * c$ itp.

Standardowe wykonanie tego działania wymaga dwóch operacji zaokrąglenia – po obliczeniu iloczynu oraz po dodawaniu. FMA nie zaokrągla wyniku częściowego (jest trzymany w wewnętrznym rejestrze o zwiększonej precyzji), a zaokrąglana jest dopiero końcowa suma. Dodatkowo: jedna operacja FMA jest z reguły szybsza od dwóch osobnych oraz zajmuje mniej miejsca w pamięci *cache* itp.

- W zależności od mikroarchitektury procesora (liczby potoków, typu jednostek wykonawczych, układów sterowania i np. przewidywania skoków) jednostek SIMD może być kilka – poniżej schemat blokowy jednostek wykonawczych mikroarchitektury Intel *Haswell*:



- Część prostych instrukcji wektorowych wykonywana jest w jednym cyklu potoku (typowe wyjątki to dzielenie i pierwiastkowanie), ich parametry dla różnych generacji procesorów zestawione zostały na liście:

  `https://www.agner.org/optimize/instruction_tables.pdf`

Większość instrukcji z danego rozszerzenia wektorowego ma swój skalarny odpowiednik – np. rozkazy *MOVSS*/*MOVSD* ładują jedną liczbę (skalar) odpowiednio pojedynczej i podwójnej precyzji, a *MULSS*/*MULD* wykonują mnożenie skalarnych argumentów. Starsze części rejestrów *%XMM* pozostają niewykorzystane.

Obecnie, nowoczesne kompilatory często używają zoptymalizowanych jednostek wektorowych do wykonania typowych, skalarnych obliczeń zmiennoprzecinkowych zamiast „odziedziczonego” po poprzednikach FPU x87.

Ładowanie danych do „długich” rejestrów wektorowych może odbywać się na dwa sposoby:

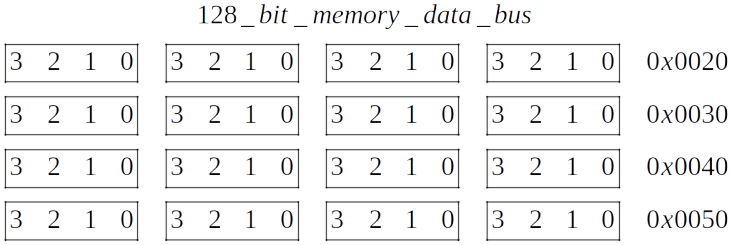
Aligned memory access.

Dane są wyrównane do granicy będącej długością (w bajtach) pobieranego/zapisywanego bloku danych.
Inaczej: adres początku pobieranego bloku danych jest wielokrotnością długości rejestru wektorowego.

Przykład.

Procesor dysponujący 128 bitową szyną danych pamięci RAM może za jednym razem pobrać z niej 16 bajtów.

Początek tablicy z szesnastoma liczbami typu *float* został wyrównany do granicy 16 bajtów (czyli każdy 16 bajtowy blok zawierający cztery liczby zaczyna się od adresu będącego wielokrotnością „16”: 32, 48, 64 i 80). W zapisie szesnastkowym od razu widać to wyrównanie po zerowym najmłodszym półbajcie.



Ładowanie czterech, wyrównanych liczb typu *float* do 128 bitowego (16 bajtowego) rejestru *%XMM* może zostać zatem przeprowadzone jednoetapowo.

Dostęp do danych wyrównanych przebiega szybciej. Procesor posiada dedykowane instrukcje przeznaczone specjalnie do tego celu (*MOV**A**PS*, *MOV**A**PD*, gdzie: **A** oznacza *aligned*). Podczas próby ich użycia w przypadku danych niewyrównanych – zgłoszony zostanie wyjątek (*segmentation fault*).

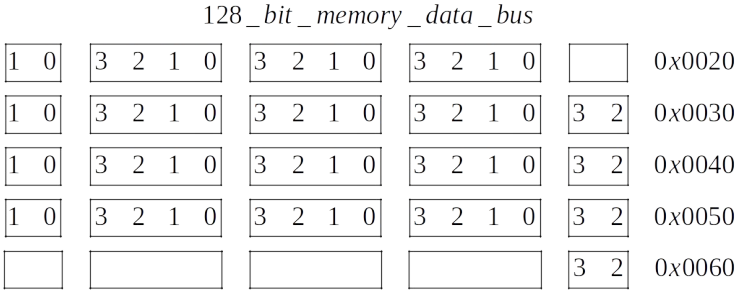
W większości przypadków kompilatory same wyrównują dane (i części kodu!) do odpowiedniej granicy dodając stosowny tzw. *padding* – zerowe bajty lub puste operacje.

Odpowiednie wyrównanie danych można również kompilatorowi narzucić, np.:

```
static double x[SIZE] __attribute__((aligned(32)));
```

Unaligned memory access: dostęp niewyrównany – nieoptymalny.

Przykład - tablica z 16 *floatami* została wyrównana do granicy dwóch bajtów (pierwszy element zaczyna się pod adresem 0x22):



Załadowanie całego wektora danych (zawierającego cztery liczby pojedynczej precyzji) w powyższym przypadku będzie wymagało dodatkowego cyklu dostępu do pamięci i „poskładania” wektora z dwóch części.

Dostęp do danych niewyrównanych (do granicy odpowiadającej długości rejestru) możliwy jest przy za pomocą instrukcji *MOV**U**PS*, *MOV**U**PD* (gdzie: **U** oznacza *unaligned*), wolniejszych i zmniejszających wydajność programu.

Rejestry i składnia instrukcji wektorowych.

W przeciwieństwie do FPU x87, rejestry rozszerzeń wektorowych nie są zorganizowane w strukturę stosu. Szesnastu 128-bitowych rejestrów *%XMM0* - *%XMM15* używa się podobnie jak rejestrów ogólnego przeznaczenia x86 (z pewnymi wyjątkami: np. nie mogą służyć do adresowania pamięci!).

Wprowadzone przez Intela (zaczynając od mikroarchitektury *SandyBridge*) rozszerzenie AVX (*Advanced Vector Extension*) „wydłużyło" rejestry wektorowe do 256-bitów (*%YMM0* - *%YMM15*). Kolejne rozszerzenie, AVX512, zwiększyło m.in. liczbę rejestrów do 32, a ich długość do 512 bitów (*%ZMM0* - *%ZMM31*).

W AVX dołożone zostały np. instrukcje FMA oraz dużo innych np. usprawniających manipulację spakowanymi liczbami w rejestrach (*broadcast*, *shuffle*, *permute*, *gather*, *extract* itp.). Zmieniona została również składnia rozkazów: z typowej x86 na wykorzystującą trzy operandy (podobnie jak w procesorach MIPS i ARM) np.:

VADDPD *%YMM0* , *%YMM1* , *%YMM2* # *%YMM2 = %YMM0 + %YMM1*

Pomijając wewnętrzny sposób przetwarzania takich rozkazów (przemianowanie rejestrów, eliminacja zależności *WaW*, *WaR*), taka składnia jest bardziej uniwersalna od dwuargumentowej - w pewnych sytuacjach można zapobiec nadpisaniu jednego z argumentów wynikiem.

Najnowsze rozwinięcia w/w koncepcji – rozszerzenie wektorów na tablice dwuwymiarowe - AMX *Advanced Matrix Extension* - w tej instrukcji pominiemy…

Nie będziemy w tym ćwiczeniu również analizować ani pełnej listy rozkazów wektorowych, ani pisać funkcji wykorzystujących instrukcje wektorowe w asemblerze.

Oprócz w/w niskopoziomowego rozwiązania dostępne kompilatory umożliwiają automatyczną wektoryzację obliczeń (również ją pominiemy) albo wykorzystanie instrukcji-funkcji wewnętrznych/wbudowanych (***intrinsic instructions***) kompilatora.

W niniejszym ćwiczeniu, do wektoryzacji algorytmu mnożenia macierzy (DGEMM - wszystkie założenia jak w lab. 12.) wykorzystane zostaną funkcje *intrinsic*.

Zasada działania najprostszej wersji wektorowej jest podobna (technicznie to nie jest to samo!) do dwukrotnego rozwinięcia pętli – obliczane są jednocześnie dwa elementy macierzy **C**: **C**[*i*][*j*] i **C**[*i*+1][*j*].

```
#include <x86intrin.h>

void dgemm_sse_2x1(int n, double* A, double* B, double* C)
{
register int i,j,k;
__m128d cv;

for (i=0; i<n; i+=2)
    for (j=0; j<n; ++j)
    {
       cv = _mm_load_pd(C+i+j*n);

       for(k=0; k<n; ++k)
       cv=_mm_add_pd(cv,_mm_mul_pd(_mm_load_pd(A+i+k*n),_mm_load1_pd(B+k+j*n)));

       _mm_store_pd(C+i+j*n, cv);
    }
}
```

Przy założeniu, że elementy macierzy przechowywane są w pamięci RAM rzędami (*row major order*):

- w pętli środkowej (*j*) jedna instrukcja `_mm_load_pd` pobiera dwie sąsiadujące liczby podwójnej precyzji (*j*-ty wiersz, *i*-ta oraz *i*+1-wsza kolumna): **C**[*i*][*j*] i **C**[*i*+1][*j*] z pamięci do rejestru *%XMM* (zmienna **cv**)*,

- w pętli wewnętrznej (*k*) obliczane są jednocześnie dwa iloczyny skalarne:
przeprowadzane są mnożenia (`_mm_mul_pd`)*k*-tych elementów dwóch kolumn (*i* i *i*+1) macierzy **A** (`_mm_load_pd`) przez *k*-te elementy *j*-tego wiersza macierzy **B** (`_mm_load1_pd`),

- akumulacja iloczynów cząstkowych (`_mm_add_pd`) następuje w dwuelementowym wektorze **cv** (rejestr *%XMM*), który po wyjściu z pętli (k) zapisywany jest w pamięci (`_mm_store_pd`) – jako elementy **C**[*i*][*j*] i **C**[*i*+1][*j*],

- całkowita liczba iteracji jest dwukrotnie mniejsza (pętla *i*) w porównaniu z wersją naiwną.

* Typ danych `__m128d` oznacza 128-bitowy wektor z dwiema zmiennymi typu *double*. W zależności od dostępności wolnych rejestrów, kompilator będzie starał się przechowywać taki wektor w rejestrze *%XMM* albo, w ostateczności, w pamięci o adresie wyrównanym do granicy 16 bajtów. Podobnie oznaczenia noszą typy wektorów całkowitoliczbowych (np. `__m128i`) lub AVX (np. `__m256d`).

Funkcje wewnętrzne – w zależności od konkretnego typu – albo są „odpowiednikiem” rozkazów procesora, np.:

`_mm_load/store_pd -> movapd` (ładuj/zapisz wektor 2 x *double*, adres wyrównany do wielokrotności 16 B)
`_mm_mul_pd -> mulpd,`

albo są tłumaczone na sekwencję odpowiednich instrukcji, np.

funkcja: `_mm_load1_pd` ładuje tę samą liczbę (typu *double*) do obu elementów docelowego wektora (tutaj: z **B**[*k*][*j*] do drugiego argumentu rozkazu `mulpd`) wykorzystując rozkaz `unpcklpd`.

W efekcie, w powyższym przykładzie, wykonywane jest działanie:

```
A[i][k]             A[i+1][k]
B[k][j]             B[k][j]
*-----------------------------------
A[i][k]* B[k][j]    A[i+1][k]* B[k][j]
```

Rozszerzenie AVX posiada już rozkazy typu *broadcast* (rozgłaszanie) – jak sama nazwa wskazuje pozwalające szybko skopiować **jedną** liczbę **do wszystkich** elementów rejestru wektorowego.

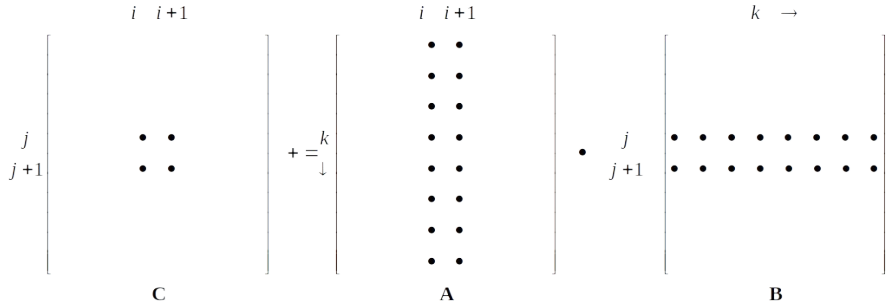
Wywołując funkcje *intrinsic* używamy argumentów-zmiennych języka C. Kompilator dba o poprawną ich zamianę na odpowiednie rejestry lub adresy komórek pamięci. Obliczanie adresu-wskaźnika przy dostępie do pamięci ma postać: baza (adres pierwszego elementu) + przesunięcie (jak w kodzie procedury `dgemm_sse`).

Plik nagłówkowy `x86intrin.h` jest używany przez kompilatory *gcc* i *icc*. W przypadku np. Visual Studio może mieć inną nazwę np. `emmintrin.h` .

Odpowiednie funkcje *intrinsic* można wyszukać na stronie Intela:

`https://software.intel.com/sites/landingpage/IntrinsicsGuide`

Wersja z czterokrotnym rozwinięciem pętli (przypomnienie):




```
void dgemm_unroll_2x2(int n, double* A, double* B, double* C)
{
register int i,j,k;
register double ci0j0,ci1j0,ci0j1,ci1j1;

for(i=0; i<n; i+=2)
    for(j=0; j<n; j+=2)
    {
       ci0j0=C[i+0+(j+0)*n];
       ci1j0=C[i+1+(j+0)*n];
       ci0j1=C[i+0+(j+1)*n];
       ci1j1=C[i+1+(j+1)*n];

       for(k=0; k<n; ++k)
       {
           ci0j0+=A[i+0+k*n]*B[k+(j+0)*n];
           ci1j0+=A[i+1+k*n]*B[k+(j+0)*n];
           ci0j1+=A[i+0+k*n]*B[k+(j+1)*n];
           ci1j1+=A[i+1+k*n]*B[k+(j+1)*n];
       }
       C[i+0+(j+0)*n]=ci0j0;
       C[i+1+(j+0)*n]=ci1j0;
       C[i+0+(j+1)*n]=ci0j1;
       C[i+1+(j+1)*n]=ci1j1;
    }
}
```

i jej **wersja wektorowa** (bez komentarza):

```
void dgemm_sse_unroll_2x2 (int n, double *A, double *B, double *C)
{
register int i,j,k;

__m128d Ci0j0,Ci0j1,bj0k0,bj1k0;

for (i=0; i<n; i+=2)
    for (j=0; j<n; j+=2)
    {
       Ci0j0 = _mm_load_pd(C+i+(j+0)*n);
       Ci0j1 = _mm_load_pd(C+i+(j+1)*n);

       for(k=0; k<n; ++k)
       {
           bj0k0 = _mm_load1_pd(B+k+(j+0)*n);
           bj1k0 = _mm_load1_pd(B+k+(j+1)*n);

           Ci0j0 = _mm_add_pd(Ci0j0,_mm_mul_pd(_mm_load_pd(A+n*k+i), bj0k0));
           Ci0j1 = _mm_add_pd(Ci0j1,_mm_mul_pd(_mm_load_pd(A+n*k+i), bj1k0));
       }
       _mm_store_pd(C+i+(j+0)*n, Ci0j0);
       _mm_store_pd(C+i+(j+1)*n, Ci0j1);
    }
}
```

Zadanie - wzorując się na powyższych przykładach oraz wykorzystując wiedzę oraz „szkielet" programu z laboratoriów 12. zmierzyć czasy i wydajność (GFLOPS) następujących wersji algorytmu DGEMM z wektoryzacją SSE:

- naiwnego,

- z rozwinięciem pętli: 2x – 8x,

- blokowego,

- blokowego z rozwinięciem pętli: 2x – 8x.

Sprawdzić typ i rozszerzenia wektorowe procesora komputera, na którym prowadzone są obliczenia. Informację o wykorzystanym procesorze zawrzeć w sprawozdaniu.

W miarę możliwości:

- w miejsce SSE zastosować rozszerzenie AVX (256-bitowe, odpowiednio zmieniając skok pętli, typy danych, i wykorzystując dedykowane funkcje intrinsic np. *_mm256_broadcast_sd*),

- jeżeli procesor dysponuje rozszerzeniem FMA lub AVX-512 – spróbować je wykorzystać w w/w algorytmach;

(np. zastąpić sekwencję funkcji *_mm256_add_pd* i *_mm256_mul_pd* jedną: *_mm256_fmadd_pd*).

Pomiary i prezentację graficzną wykonać podobnie jak w ćw. 12. - dla macierzy o rozmiarach np. 512, 1024, 2048 (ew. blokach 8, 16, 32, 64, 128),

Wybrać najszybsze wersje algorytmów mnożących dla danego rozmiaru macierzy (przy założeniu, że lab. 12. i 13. wykonywane są na tym samym komputerze…),

Zadanie dla chętnych:

Spróbować zwektoryzować algorytm w wersji z usuniętymi „skokami" w przestrzeni adresowej (przy odczycie kolejnych elementów macierzy **A**, analogicznie jak w ćw. 12).

Uwaga:

Włączyć w kompilatorze *gcc* autooptymalizację np. `-O3` (również przy kompilacji biblioteki do pomiaru czasu).

Gdyby pojawiły się problemy z kompilacją programu zawierającego funkcje *intrinsics* rozszerzenia AVX/AVX2/AVX-512 (starsze wersje gcc, np. 4) należy… zaktualizować.

Gdyby błędy pojawiały się również w nowszej wersji (np. >= 7) można wymusić kompilację dostosowaną do konkretnego typu mikroarchitektury (`-march`), np.:

CPU Xeon E5 2630v4 to mikroarchitektura Broadwell.

```
gcc –c proc.c –O3 –march=broadwell
```

Jeżeli podana zostanie błędna nazwa mikroarchitektury – *gcc* wyświetli listę dostępnych-wspieranych w danej wersji.

Mikroarchitekturę procesora można sprawdzić na stronie:

`https://ark.intel.com/content/www/pl/pl/ark.html#@Processors`