Wydajność obliczeń sekwencyjnych. Osobliwości pracy z cache - liniami.

- **Typowe algorytmy algebry liniowej i oszacowanie możliwości ich przyspieszenia**

$$\mathbf{y} = \alpha \cdot \mathbf{x} + \mathbf{y}$$ **- saxpy (scalar α X plus Y)**

$$\mathbf{y} = \mathbf{y} + \mathbf{A} \cdot \mathbf{x}$$

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

- **Przypomnieć reguły algebry liniowej: mnożenia macierzy przez wektor i macierzy przez macierz**

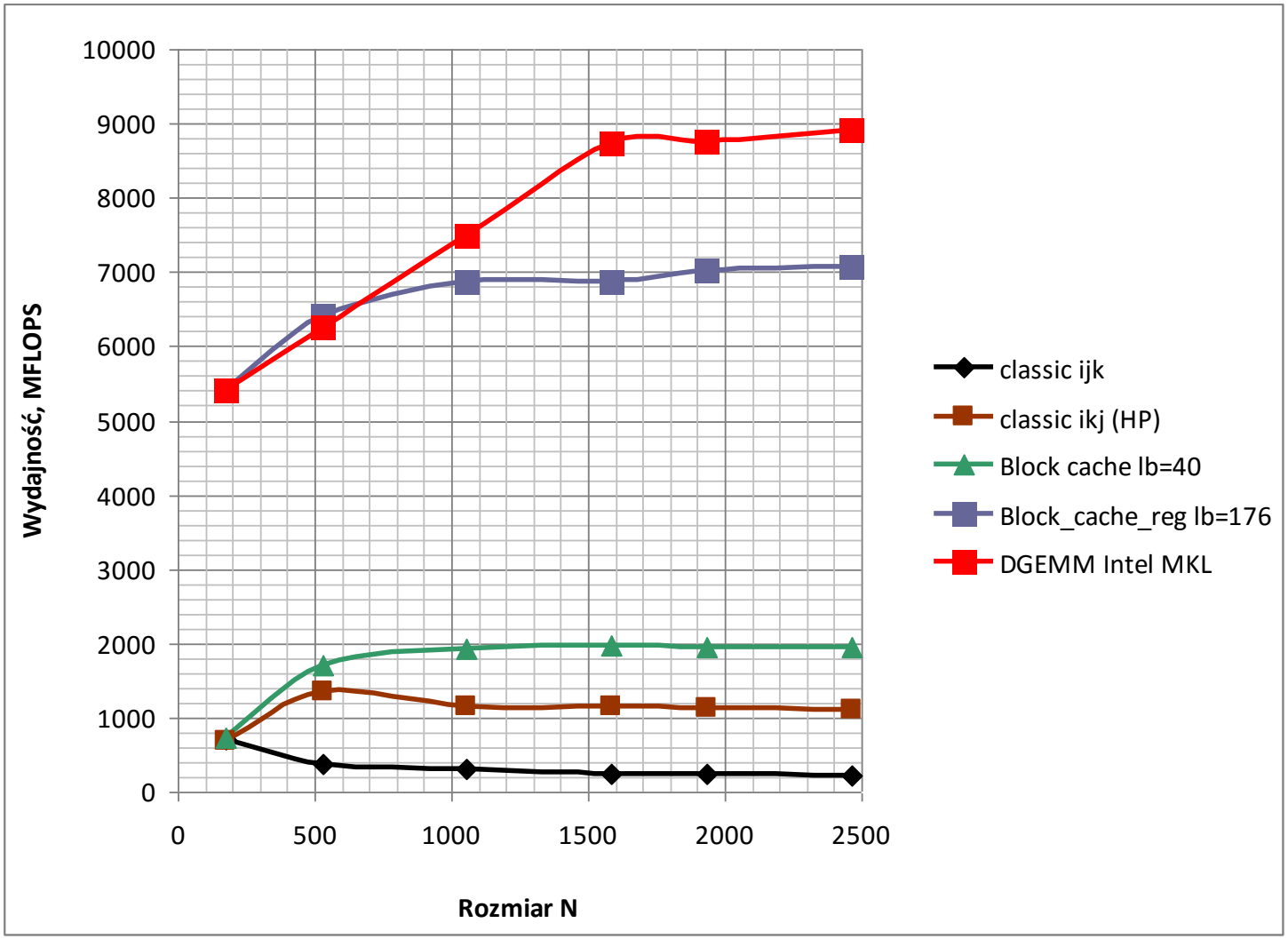
- **Demonstracja programu Multm: ta sama ilość operacji – różna wydajność**

Rozmiar zagadnienia: 2 000 równań

Charakterystyki komputera:

Procesor – Intel® Core™2 Quad CPU Q6600 @2.40 GHz
Cache - L1: 32 KB, L2:4096 KB
Pamięć: DDR2 800 MHz 4 GB

Wydajność różnych algorytmów C=C+A*B, MFLOPS



Wydajność dla metody mnożenia block_cache_reg w zależności od rozmiaru bloku C=A*B (Mflops), Nrównan = 2 000,

Rozmiar bloku	Wydajność, MFLOPS
8x8	3 615
16x16	4 714
32x32	5 284
64x64	5 944
128x128	6 703
256x256	6 828
512x512	3 765

Istnieje optymalny rozmiar błoku lb, przy którym osiągamy wydajność maksymalną. Jak będzie podano dalej, ten rozmiar zależy od sprzętu, mianowicie od rozmiaru cache L1 i rozmiaru TLB (Translation look-aside) bufora.

Oszacowanie wydajności algorytmów

Założenia

- **Odczyt i zapis (cache ↔ RAM) i operacje arytmetyczne są wykonywane w czasie sekwencyjne (rezygnujemy z równoległości operacji arytmetycznych i we\wy)**

- **Pamięć podręczna (cache) jest rozdzieloną pomiędzy danymi w najbardziej skuteczny sposób (procesor i system operacyjny są „intelektualne" – dane załadowane w cache tak, żeby zminimalizować ilość przesyłek cache ↔ RAM)**

- **Komputer ma tylko rejestry (najbardziej szybką pamięć), jeden poziom cache (szybka pamięć – rezygnujemy z hierarchicznej struktury pamięci podręcznej) i pamięć główną (dostęp do adresów której jest wykonany z szybkością magistrali).**

Przyjmujemy:

- **Wszystkie tablicy danych są umieszczone w pamięci głównej**
- **n - rozmiar zagadnienia**
- **M - objętość pamięci podręcznej (cache), przy czym M << n**

Będziemy rozważali takie przypadki:

- **M ≥ 2n – w pamięci podręcznej można umieścić 2 wiersze macierzy (małe zagadnienie)**
- **M < 2n – dwa wiersze macierzy nie udaje się umieścić w cache (duże zagadnienie)**

Czas wykonania programu:

$$t = f \cdot t_{arith} + m \cdot t_{bus} = f \cdot t_{arith} \cdot \left(1 + \frac{m}{f} \cdot \frac{t_{bus}}{t_{arith}}\right)$$

f, m **- ilość arytmetycznych operacji i ilość przesłań danych cache ↔ RAM**

t_{arith}, t_{bus} **- trwałość jednej operacji arytmetycznej i trwałość przesłania jednego słowa**

Wskaźnik wydajności wykorzystania pamięci szybkiej:

$$q = \frac{f}{m}$$

Umieszczenie danych w pamięci:

Macierze

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Wektory

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Podział macierzy na bloki

$$A = \begin{pmatrix} \{a\}_{1,1} & \{a\}_{1,2} \\ \{a\}_{2,1} & \{a\}_{2,2} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & : & a_{13} & a_{14} \\ a_{21} & a_{22} & : & a_{23} & a_{24} \\ . & . & . & . & . \\ a_{31} & a_{32} & : & a_{33} & a_{34} \\ a_{41} & a_{42} & : & a_{43} & a_{44} \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & : & 3 & 4 \\ 5 & 6 & : & 7 & 8 \\ . & . & . & . & . \\ 9 & 10 & : & 11 & 12 \\ 13 & 14 & : & 15 & 16 \end{pmatrix}$$

SAXPY: y = y+αx

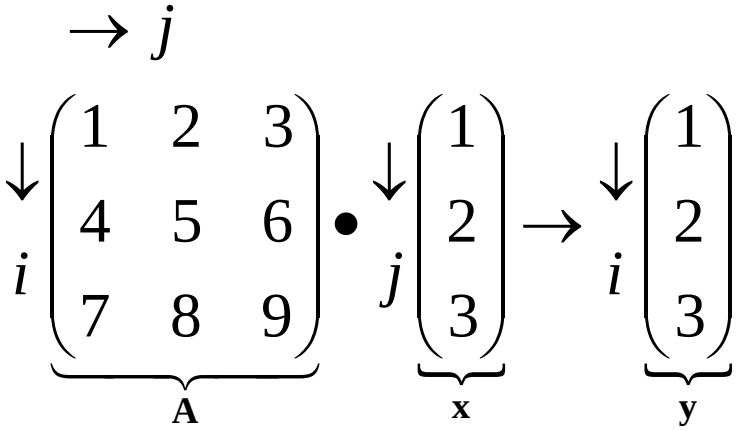
```
//odczyt alpha i umieszczenie w rejestrze
for(i=1; i<=n; i++)
{
        //odczyt y[i], x[i]
        y[i] = y[i] + alpha*x[i];
        //zapis y[i]
}
```

Ilość odczytów: α – 1; x – n; y – n
Ilość zapisów : y - n
Razem : m = 3n+1

Ilość mnożeń : n
Ilość dodawań : n
Razem : f = 2n

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

Mnożenie macierzy przez wektor: y = y+Ax




```
for(i=1; i<=n; i++)
{
        //odczyt y_i  i umieszczenie w rejestr
        for(j=1; j<=n; j++)
        {
                //odczyt A_ij , x_j
                y_i = y_i + A_ij*x_j;
        }
        //zapis y_i
}
```

Ilość odczytów:

y – n;

x – n, jeśli M ≥ 2n,

n^2, jeśli M < 2n;

A – n^2;

Ilość zapisów : y – n

Razem : m = n^2+3n (M ≥ 2n)

m = $2n^2$+2n (M < 2n)

Ilość mnożeń : n^2

Ilość dodawań : n^2

Razem : f = $2n^2$

$$q = \frac{f}{m} = \frac{2n^2}{n^2 + 3n} = \frac{2}{1 + \frac{3}{n}} \approx 2, \quad M \geq 2n$$

$$q = \frac{f}{m} = \frac{2n^2}{2n^2 + 2n} = \frac{2}{2 + \frac{2}{n}} \approx 1, \quad M < 2n$$

Wniosek

- **Wydajność drugiego algorytmu jest bardziej wysoką od pierwszego zagadnienia, ale jako w pierwszym zagadnieniu, tak i w drugim wskaźnik wydajności nie zależy od rozmiaru pamięci podręcznej. To oznacza, że te algorytmy są skazane na wykonanie z szybkością wolnej magistrali, a nie szybkiego procesora. Zwiększenie częstości zegara procesora, objętości pamięci podręcznej nie powoduje istotnego przyspieszenia obliczeń.**

- **Przykład: przesłanie jednego słowa trwa ~40 cyklów procesora. Dla algorytmu q = 1 oznacza, że na jedną operacje arytmetyczną przypada przesłanie jednego słowa. Jeden cykl procesor wytraca na operacje arytmetyczną. A pozostałe 39 cyklów? Są puste.**

- **Tu nie są uwzględnione prefetch i ten fakt, że ładowanie danych do cache jest wykonywane nie słowo po słowie, a grupami słów, umieszczonych w cache-linie. Przecież nie zmienia to sytuacji w całości.**

Mnożenie macierzy przez macierz: C = C + A*B

1. Metoda klasyczna (ijk)

$$\underset{\downarrow i}{\underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}}}_{\mathbf{C}} \; \begin{matrix} \rightarrow j \end{matrix} \quad += \quad \underset{\downarrow i}{\underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}}}_{\mathbf{A}} \; \begin{matrix} \rightarrow k \end{matrix} \quad \bullet \quad \underset{\downarrow k}{\underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}}}_{\mathbf{B}} \; \begin{matrix} \rightarrow j \end{matrix}$$

Macierze są umieszczone w pamięci głównej wiersz po wiersze

Wzór matematyczny:

$$c_{ij} += \sum_{k=1}^{n} a_{ik} b_{kj}, \quad i, j \in [1, n]$$

Algorytm:

```
for(i=1; i<=n; i++)
{
        for(j=1; j<=n; j++)
        {
                //odczyt Cij  i umieszczenie w rejestrze
                for(k=1; k<=n; k++)
                {
                        //odczyt Aik , Bkj
                        Cij = Cij + Aik*Bkj ;
                }
                //zapis Cij
        }
}
```

UWAGA! Przy każdej zmianie indeksu k będzie pobrany element macierzy B_{kj} z następnego wiersza → jeśli macierz jest dość dużą, bez sensu przydzielać cache dla ej elementów → wyniknie przeładowanie cache przy każdej zmianie indeksu k. Intelektualny procesor i system operacyjny będą pobierali elementy macierzy B bez buforowania.

Ilość odczytów:
C – n^2;
A – n^2, jeśli M ≥ 2n,
n^3, jeśli M < 2n;
B – n^3;
Ilość zapisów : C – n^2
Razem : m = n^3+3n^2 (M ≥ 2n)
m = $2n^3+2n^2$ (M < 2n)

Ilość mnożeń : n^3
Ilość dodawań : n^3
Razem : f = $2n^3$

$$q = \frac{f}{m} = \frac{2n^3}{n^3+3n^2} = \frac{2}{1+\frac{3}{n}} \approx 2, \quad M \geq 2n$$

$$q = \frac{f}{m} = \frac{2n^3}{2n^3+2n^2} = \frac{1}{1+\frac{1}{n}} \approx 1, \quad M < 2n$$

Metoda klasyczna (ikj) - HP

```
for(i=1; i<=n; i++)
{
        for(k=1; k<=n; k++)
        {
                //odczyt A_ik  i umieszczenie w rejestrze
                for(j=1; j<=n; j++)
                {
                        //odczyt C_ij , B_kj
                        C_ij = C_ij + A_ik*B_kj ;
                         //zapis C_ij , jak tylko bufor będzie wyczerpany
                }

        }
}
```

- **Model nie uwzględnia pobierania danych z pamięci: wskaźnik wydajności dla macierzy dużych (M < 2n) okazuje się nawet gorzej, niż w przypadku poprzednim – to się nie odpowiada rzeczywistości.**

- **Na praktyce ten model liczy szybszej od poprzedniego dla tego, że pozostałe usunięte skoki w danych**

Ilość odczytów:

C – n^2 , jeśli M ≥ 2n,

n^3, jeśli M < 2n;

A – n^2 ;

B – n^3 ;

Ilość zapisów :

C – n^2 , jeśli M ≥ 2n,

n^3, jeśli M < 2n;

Razem : m = n^3+3n^2 (M ≥ 2n)

m = $3n^3+n^2$ (M < 2n)

Ilość operacji arytmetycznych: f = $2n^3$

$$q = \frac{f}{m} = \frac{2n^3}{n^3 + 3n^2} = \frac{2}{1 + \frac{3}{n}} \approx 2, \quad M \geq 2n$$

$$q = \frac{f}{m} = \frac{2n^3}{3n^3 + n^2} = \frac{2}{3 + \frac{1}{n}} \approx \frac{2}{3}, \quad M < 2n$$

Metoda blokowa

Macierz jest podzieloną na bloki o takim rozmiarze, że trzy bloki jednocześnie mogą być umieszczone w pamięć podręczną (cache)

$$\underbrace{\begin{array}{c} \\ \downarrow \\ ib \end{array}\begin{pmatrix} 1 & 2 & : & 3 & 4 \\ \underbrace{5}_{\mathbf{C}_{11}} & \underbrace{6}_{\mathbf{C}_{11}} & : & \underbrace{7}_{\mathbf{C}_{12}} & \underbrace{8}_{\mathbf{C}_{12}} \\ \cdots\cdots & & : & \cdots\cdots & \\ 9 & 10 & : & 11 & 12 \\ \underbrace{13}_{\mathbf{C}_{21}} & \underbrace{14}_{\mathbf{C}_{21}} & : & \underbrace{15}_{\mathbf{C}_{22}} & \underbrace{16}_{\mathbf{C}_{22}} \end{pmatrix}}_{\mathbf{C}} \; += \; \underbrace{\begin{array}{c} \\ \downarrow \\ ib \end{array}\begin{pmatrix} 1 & 2 & : & 3 & 4 \\ \underbrace{5}_{\mathbf{A}_{11}} & \underbrace{6}_{\mathbf{A}_{11}} & : & \underbrace{7}_{\mathbf{A}_{12}} & \underbrace{8}_{\mathbf{A}_{12}} \\ \cdots\cdots & & : & \cdots\cdots & \\ 9 & 10 & : & 11 & 12 \\ \underbrace{13}_{\mathbf{A}_{21}} & \underbrace{14}_{\mathbf{A}_{21}} & : & \underbrace{15}_{\mathbf{A}_{22}} & \underbrace{16}_{\mathbf{A}_{22}} \end{pmatrix}}_{\mathbf{A}} \; \bullet \; \underbrace{\begin{array}{c} \downarrow \\ kb \end{array}\begin{pmatrix} 1 & 2 & : & 3 & 4 \\ \underbrace{5}_{\mathbf{B}_{11}} & \underbrace{6}_{\mathbf{B}_{11}} & : & \underbrace{7}_{\mathbf{B}_{12}} & \underbrace{8}_{\mathbf{B}_{12}} \\ \cdots\cdots & & : & \cdots\cdots & \\ 9 & 10 & : & 11 & 12 \\ \underbrace{13}_{\mathbf{B}_{21}} & \underbrace{14}_{\mathbf{B}_{21}} & : & \underbrace{15}_{\mathbf{B}_{22}} & \underbrace{16}_{\mathbf{B}_{22}} \end{pmatrix}}_{\mathbf{B}}$$

$\rightarrow jb$ $\qquad$ $\rightarrow kb$ $\qquad$ $\rightarrow jb$

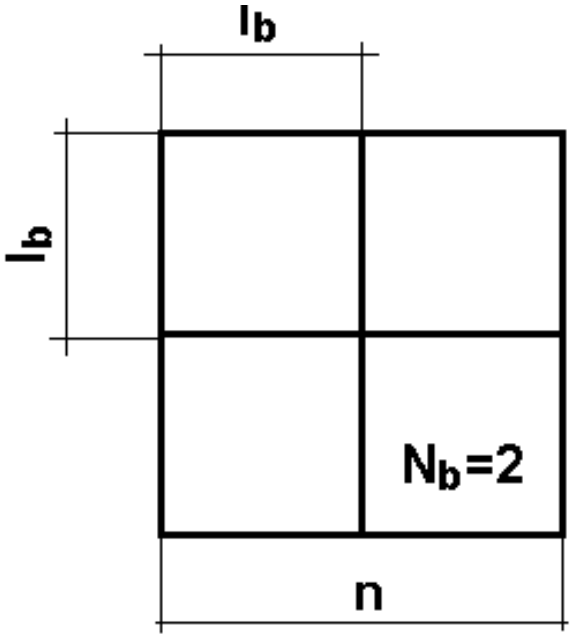
Dla macierzy blokowej zastosowujemy działania nad blokami:

$$\mathbf{C}_{ib,jb} = \mathbf{C}_{ib,jb} + \sum_{kb=1}^{Nb} \mathbf{A}_{ib,kb} \cdot \mathbf{B}_{kb,jb}, \quad ib, jb \in [1, Nb]$$

W obszarze każdego bloku używamy zwykłe działania nad macierzami

Algorytm (*)

```
for(ib=1; ib<=Nb; ib++)
{
        for(jb=1; jb<= Nb; jb++)
        {
                //odczyt  bloku Cib,jb  i umieszczenie w
                //cache
                for(kb=1; kb<= Nb; kb++)
                {
                        //odczyt bloków Aib,kb , Bkb, jb
                        Cib,jb = Cib,jb + Aib,kb*Bkb,jb ;
                }
                 //zapis bloku Cib,jb
        }
}
```



- Rozmiar każdego bloku wynosi $l_b = n/N_b$, gdzie n – rozmiar macierzy, N_b - ilość podziałów na bloki wzdłuż jednej iż stron (przypuszczamy, że n jest wielokrotne N_b)
- Ilość słów w każdym bloku jest równa $l_b^2 = n^2/N_b^2$

Ilość odczytów:

C – $N_b^2 \cdot l_b^2 = N_b^2 \cdot n^2/N_b^2 = n^2$;

A – $N_b^3 \cdot l_b^2 = N_b^3 \cdot n^2/N_b^2 = N_b \cdot n^2$;

B – $N_b^3 \cdot l_b^2 = N_b \cdot n^2$;

Ilość zapisów :

C – n^2 ;

Razem : $m = 2 \cdot N_b \cdot n^2 + 2n^2 = 2 \cdot (N_b + 1) \cdot n^2 \approx 2 \cdot N_b \cdot n^2$;

Ilość operacji arytmetycznych: $f = 2n^3$

$$q = \frac{f}{m} = \frac{2n^3}{2N_b n^2} = \frac{n}{N_b} = l_b;$$

$$3\,bloki\,muszą\,byc\,w\,cache \rightarrow 3l_b^{\,2} \le M$$

$$3l_b^{\,2} \le M \rightarrow l_b = \sqrt{\frac{M}{3}}$$

$$q = \sqrt{\frac{M}{3}};$$

Wydajność algorytmów algebry liniowej jest przyjęte odnosić do jednego z poziomów BLAS (Basis Linear Algebra Subroutines) zgodnie z rzędem operacji arytmetycznych $O(n^k)$. BLAS1 – $O(n^1)$, BLAS2 – $O(n^2)$, BLAS3 – $O(n^3)$

Algorytm	Wzór matematyczny	$q = \frac{f}{m}$	Poziom BLAS
saxpy	$\mathbf{y} = \mathbf{y} + \alpha \cdot \mathbf{x}$	*2/3*	1 – O(n)
Mnożenie macierzy przez wektor	$\mathbf{y} = \mathbf{y} + \mathbf{A} \cdot \mathbf{x}$	*2 (M ≥ 2n)* *1 (M<2n)*	2 – $O(n^2)$
Mnożenie macierzy przez macierz (Classic)	$\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$	*2 (M ≥ 2n)* *1 (M < 2n)*	3 – $O(n^3)$
Mnożenie macierzy przez macierz (HP)	$\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$	*2 (M>=2n)* *2/3 (M<2n)*	3 – $O(n^3)$
Mnożenie macierzy przez macierz (metoda blokowa)	$\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$	$\sqrt{\frac{M}{3}}$	3 – $O(n^3)$

Wnioski

- **Algorytmy poziomu BLAS 1, 2 - $O(n^1)$, $O(n^2)$ – nie wykazują oszacowania wydajności lepiej od q = 2. Dla nich podstawowo nie jest możliwe podniesienie wydajności w skutek utrzymania danych w cache – ilość operacji arytmetycznych jest tego samego rzędu, jak i ilość przesłań danych. Oni są przyrzeczone pracować z prędkością wolnej magistrali.**
- **Algorytmy poziomu BLAS 3 - $O(n^3)$ – mogą być zorganizowane tak, że część danych, jeden raz umieszczoną w cache, pozostaje wykorzystana wielokrotnie (cache reuse). Powoduje to zmniejszenie rzędu ilości przesłań danych w stosunku do rzędu operacji arytmetycznych – oszacowanie wydajności jest lepsze od q = 2. Podstawą dla tworzenia takich algorytmów jest podział danych na bloki.**
- **Przy oszacowaniu $q \sim \sqrt{M}$ można przynajmniej teoretycznie tak dobrać rozmiar cache że procesor będzie miał minimalną ilość pustych cykli.**
- **Im więcej jest l_b ($3l_b^2 < M$), tym większa wydajność.**

Podział na bloki na poziomie rejestrów. (Register's blocking)

- **Przedstawiona powyżej procedura podziału na bloki w taki sposób, żeby jednocześnie w pamięci podręcznej mieścili się 3 bloki, okazała się podstawą dla podniesienia wydajności algorytmu mnożenia macierzy przez macierz rzędu ~O(n^3) działań arytmetycznych.**

- **Procedura ta zmniejsza ilość przesłań danych pamięć główna – pamięć podręczna – pamięć główna i dostała nazwę cache blocking.**

- **Okazuje się że przy przesyłaniu danych pamięć podręczna – rejestry – pamięć podręczna zastosowanie techniki blokowej zmniejsza ilość ładowań rejestrów (register's reuse) i też służy do podniesienia wydajności algorytmu.**

- **Rozważmy algorytm $C_{ib,jb} = C_{ib,jb} + A_{ib,kb}*B_{kb,jb}$ - pętla wewnętrzna algorytmu (*)**

Algorytm naiwny

(mnożenie bloków macierzy Cib,jb = Cib,jb + Aib,kb*Bkb,jb)

```
for(i = 0; i < l_b; i++)
{
   for( j = 0; j < l_b; j++)
   {
      r = 0;
      for(k = 0; k < l_b; k++)
      {
         r = r + a_ik · b_kj;
      }
      c_ij = c_ij + r;
   }
}
```

W pętle wewnętrznej:

Ilość ładowań rejestrów: 2

1 odczyt a_{ik} + 1 odczyt b_{ik}

Ilość operacji arytmetycznych: 2

1 mnożenie + 1 dodawanie

---

q = f/m = 2/2 = 1

Ilość potrzebnych rejestrów typu double: 4

***r*, a_{ik}, b_{ik} , *tmp* ← $a_{ik}b_{ik}$**

Żadnego blokowania rejestrów w tym algorytmie nie ma

Tu *r* – zmienna klasy register, która ma być przechowywana w rejestrze w ciągu wykonania całej pętli wewnętrznej, a_{ik}, b_{ik} – elementy bloków $A_{ib,kb}$, $B_{kb,jb}$ odpowiednio.

Po zakończeniu pętli wewnętrznej wartość zmiennej *r* będzie przypisana c_{ij} - elementowi bloku $C_{ib,jb}$.

Schemat blokowania rejestrów 2x2

```
for(i = 0; i < l_b; i+ = 2)
{
   for( j = 0; j < l_b; j+ = 2)
   {
      r0 = 0;   r1 = 0;   r2 = 0;   r3 = 0;
      for(k = 0; k < l_b; k + +)
      {
         // Ładujemy w rejestry A_0 ← a_ik;   A_1 ← a_{i+1,k};
         // B_0 ← b_kj;
         r = A_0 · B_0;   // r ← a_ik · b_kj
         r0 = r0 + r;   // r0 ← r0 + a_ik · b_kj
         r = A_1 · B_0;   // r ← a_{i+1,k} · b_kj
         r2 = r2 + r;   // r2 ← r2 + a_{i+1,k} · b_kj
         // Ładujemy w rejestr B_0 ← b_{k,j+1};
         r = A_0 · B_0;   // r ← a_ik · b_{k,j+1}
         r1 = r1 + r;   // r1 ← r1 + a_ik · b_{k,j+1}
         r = A_1 · B_0;   // r ← a_{i+1,k} · b_{k,j+1}
         r3 = r3 + r;   // r3 ← r3 + a_{i+1,k} · b_{k,j+1}
      }// zakonczenie petli k
      c_ij = c_ij + r0;   c_{i,j+1} = c_{i,j+1} + r1;
      c_{i+1,j} = c_{i+1,j} + r2;   c_{i+1,j+1} = c_{i+1,j+1} + r3;
   }// zakonczenie petli j
}// zakonczenie petli i
```

Rejestry: *r0, r1, r2, r3* służą dla przechowywania poprawek do elementów c_{ij} , $c_{i,j+1}$, $c_{i+1,j}$, $c_{i+1,j+1}$ w ciągu iterowania całej pętli wewnętrznej.

A_0, A_1 – do przechowywania a_{ik}, $a_{i+1,k}$; B_0 – b_{kj}, $b_{k,j+1}$; *r* – pomocniczy rejestr do przechowywania wyników mnożenia.

Razem: 8 rejestrów typu double

Fragment kodu dgemm_c

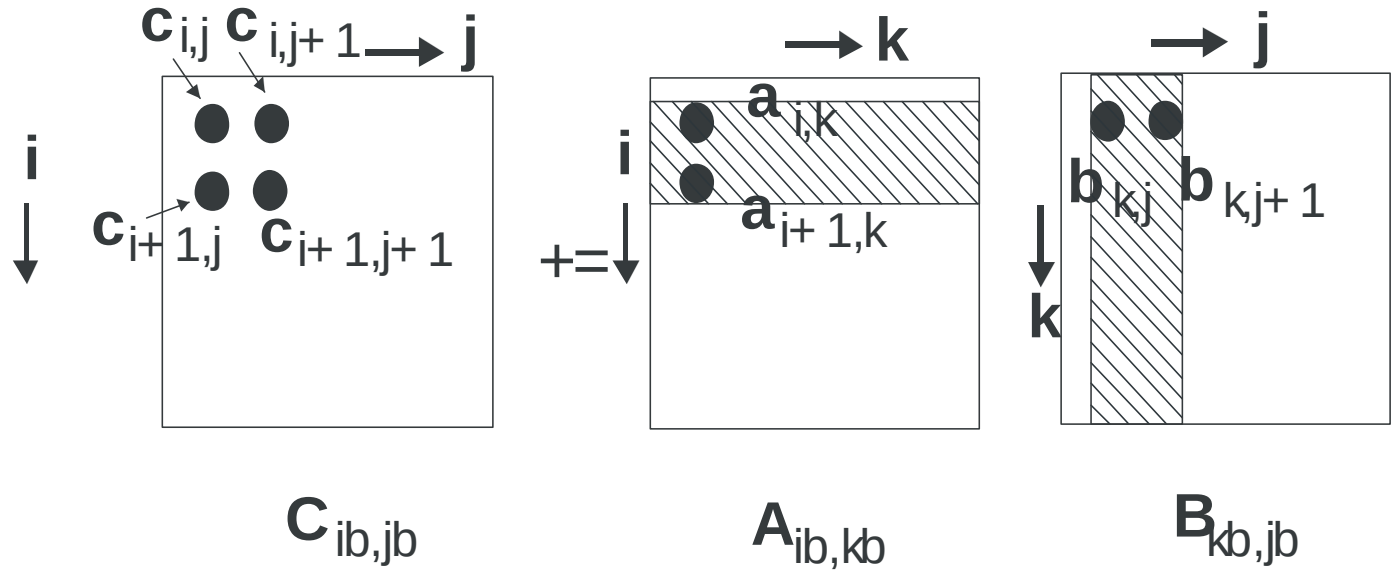
W pętle wewnętrznej:

Ilość ładowań w rejestry: 4
Ilość operacji arytmetycznych: 8 (4 mnożenia + 4 dodawania)

---

q = f/m = 8/4 = 2

To jest znacznie lepiej, niż dla poprzedniego algorytmu, przecież potrzebuje 8 rejestrów typu double.



Rozmieszczenie rejestrów double w schemacie blokowania 2x2

Blokowanie 3x3

Można udowodnić, że blokowanie 3x3 będzie jeszcze lepiej:
w pętle wewnętrznej - 6 odczytów w rejestry, 18 operacji arytmetycznych,

$q = f/m = 18/6 = 3$

Przecież potrzebuje 15 rejestrów typu double: 9 rejestrów – dla poprawek do elementów macierzy $C_{ib,kb}$, 3 rejestry – dla elementów macierzy $A_{ib,kb}$, 1 rejestr – dla elementów macierzy $B_{ik,jk}$ i 2 rejestry – pomocnicze.

Wnioski

- **Blokowanie rejestrów zmniejsza ilość przesłań danych cache-rejestr-cache w skutek utrzymania części danych w rejestrach. To powoduje zwiększenie współczynnika ilość arytmetycznych operacji/ilość przesłań danych na poziomie pamięć podręczna – rejestr – pamięć podręczna .**

- **Im więcej rejestrów double zawiera procesor, tym większy rozmiar bloku uda się zrealizować, tym bardziej wysoką będzie wydajność.**

- **Dla schematu blokowania n×n: $q = 2n^2/(2n) = n^2/n$, ilość potrzebnych rejestrów - n^2+2n**

Przykład:

Rozmiar macierzy N = 1 000

Algorytm naiwny ijk:	**300 MFLOPS**
Blokowanie rejestrów 2x2:	**2 600 MFLOPS**
Blokowanie rejestrów 2x2, blokowanie cache 64×64 i repack (BLAS 1989, J. Dongarra):	**3 121 MFLOPS**

Fragment kodu dgemm_dong

Komputer: Procesor – Intel® Core™2 Quad CPU Q6600 @2.40 GHz
Cache - L1: 32 KB, L2:4096 KB; Pamięć: DDR2 800 MHz 4 GB

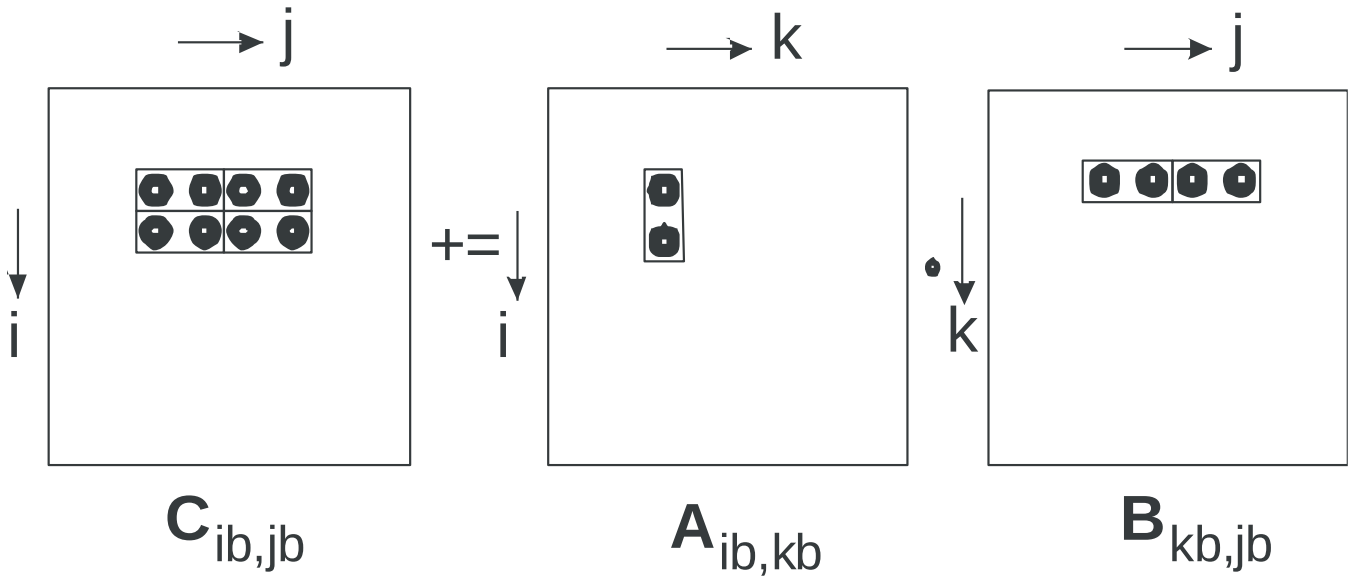
Kompilator: Intel C/C++ 10.1.021
/O3 /QaxT /QxT /Qunroll:10 /Qparallel

Zastosowanie rejestrów wektorowych

- Następnym etapem podniesienia wydajności jest zastosowanie rejestrów wektorowych. Procesory Pentium IV zawierają 8 128-bitowych rejestrów XMM (Streaming SIMD Extensions SSE2 technologii), każdy z których umieszcza dwa słowa typu double. Takie rejestry nadają możliwość za jeden cykl procesora wykonywać dwa mnożenia lub dwa dodawania.

- W porównaniu ze zwykłymi rejestrami typu double, które mogą wykonywać tylko jedną operacje (mnożenie lub dodawanie) za jeden cykl procesora, użycie rejestrów wektorowych teoretycznie zezwala na podniesianie wydajności algorytmów obliczeniowych, jeśli nie powstaje przy tym zatrzymanie przesłań danych RAM – cache – RAM, cache – rejestry - cache.

- To jest element architektury SIMD (single instruction stream – multiple data stream). Wcześniej zastosowanie rejestrów XMM było możliwe tylko w języku assemblera. Teraz technologii SSE2, SSE3 wspierają możliwość oprogramowania na poziomie języka C/C++.



Rozmieszczenie rejestrów wektorowych w schemacie blokowania 2x4, używanego dla procesorów Pentium IV architektury Prescott и Opteron

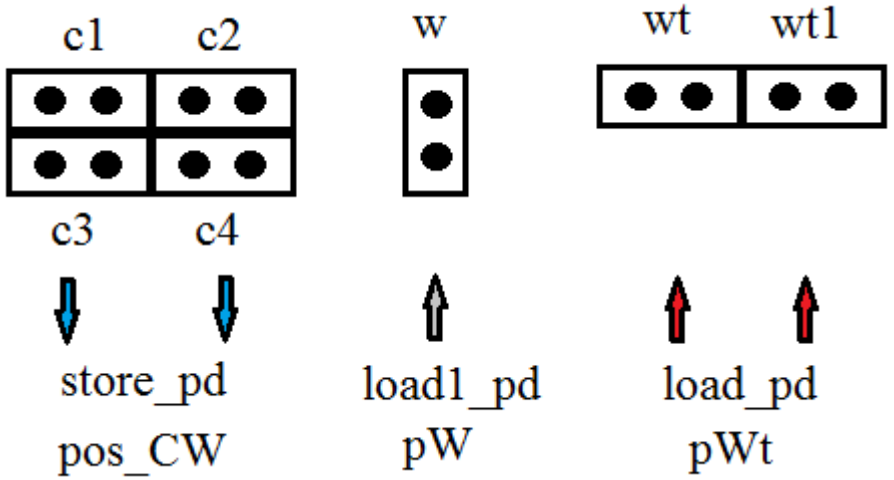
Przy jednej iteracji pętli wewnętrznej mamy:
6 ładowań danych w rejestry
16 operacji arytmetycznych
----
q = 16/6 = 2.67

W3

```
#include <emmintrin.h>
#include <intrin.h>
__m128d c1, c2, c3, c4, w, w1, wt, wt1;   //8 registers MMX are needed
const int mr = 2;
const int nr = 4;
for(i=0; i<lbv; i+=mr)
{
   for(j=0; j<lbh; j+=nr)
   {
        pWt = WT+M*j;          //point to block B_kb,jb ;  M – ilość wierszy w macierzy B_kb,jb
        pW = W+i*M;            //point to block A_ib,kb  ;  M – ilość kolumn w macierzy A_ib,kb
        //cleaning of registers for C_ib,kb
        c1 = _mm_setzero_pd();
        c2 = _mm_setzero_pd();
        c3 = _mm_setzero_pd();
        c4 = _mm_setzero_pd();

        for(k=0; k<M; k++)
        {
              //----------------------- 0
              w   = _mm_load1_pd(pW);        //load w <- a_ik, a_i,k
              wt  = _mm_load_pd(pWt);        //load wt <- b_kj, b_k,j+1
              wt1 = _mm_load_pd(pWt+2);      //load wt1 <- b_k,j+2, b_k,j+3
              w1 = w;                        //w1 <- w

              w1 = _mm_mul_pd(w1, wt);       //{a_ik b_kj, a_ik,b_k,j+1}: w1<-w*wt
              w   = _mm_mul_pd(w, wt1);      //{a_ik b_k,j+2, a_ik,b_k,j+3}: w1<-w*wt1

              c2  = _mm_add_pd(c2, w);       //{c_i,j+2, c_i,j+3} <- {c_i,j+2, c_i,j+3}+{ a_ik b_k,j+2, a_ik,b_k,j+3}
              c1  = _mm_add_pd(c1, w1);      //{c_ij, c_i,j+1} <- {c_ij, c_i,j+1}+{a_ik b_kj, a_ik,b_k,j+1}
```

W3

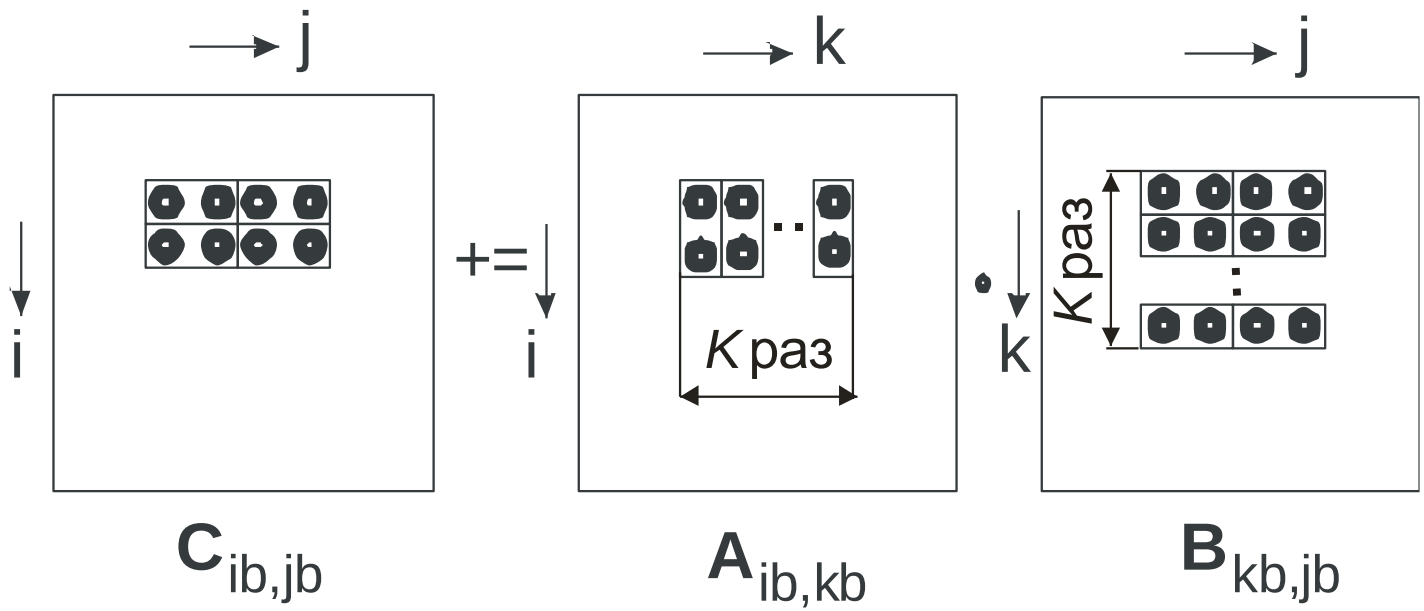
```
            w = _mm_load1_pd(pW+1);      // load w <- a_{i+1,k}, a_{i+1,k}
            w1 = w;                      //w1 <- w

            w1 = _mm_mul_pd(w1, wt);     //{a_{i+1,k}b_{kj}, a_{i+1,k},b_{k,j+1}}: w1<-w*wt
            w  = _mm_mul_pd(w, wt1);     //{a_{i+1,k}b_{kj}, a_{i+1,k},b_{k,j+1}}: w1<-w*wt

            c4 = _mm_add_pd(c4, w);      //{c_{i+1,j+2}, c_{i+1,j+3}} <- {c_{i+1,j+2}, c_{i+1,j+3}}+{a_{i+1,k}b_{k,j+2}, a_{i+1,k},b_{k,j+3}}
            c3 = _mm_add_pd(c3, w1);     //{c_{i+1,j}, c_{i+1,j+1}} <- {c_{i+1,j}, c_{i+1,j+1}}+{a_{i+1,k}b_{kj}, a_{i+1,k},b_{k,j+1}}

            pW += mr;
            pWt+= nr;
        }

        pos_CW = mr*j;
        //unload registers to C
        _mm_store_pd(CW+pos_CW, c1);
        _mm_store_pd(CW+pos_CW+2, c2);
        _mm_store_pd(CW+pos_CW+4, c3);
        _mm_store_pd(CW+pos_CW+6, c4);

    }//j loop
} //i loop
```



Uwaga! Dane tablic $B_{kb,jb}$, $C_{ib,jb}$ mają być wyrównane po granice 16 bajtów!

```
A = (double *)_aligned_malloc((number_of_items)*sizeof(double), 16);
…….
_aligned_free(A);
```

Dla wciągnięcia potokowych technologii procesora rozwijamy pętle wewnętrzne K razy (ilość używanych rejestrów MMX nie zależy od K):



W zależności od typu procesora K może osiągnąć 90 – 120 razy. Napisana w taki sposób pętla wewnętrzna dostała nazwę mikrojądra (Microkernel) .

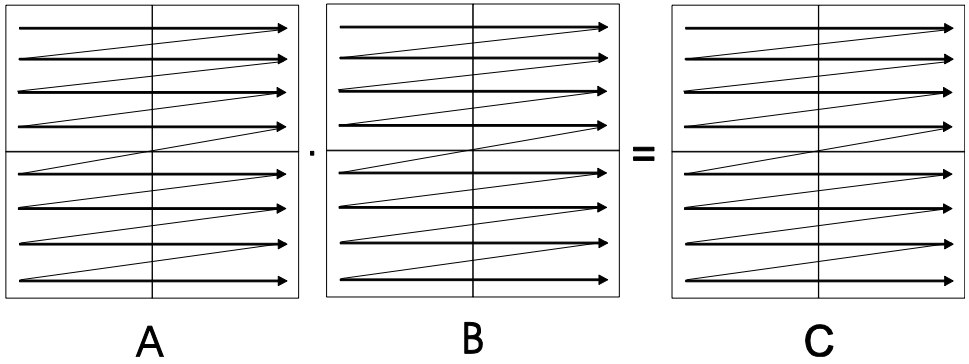
Pakowanie danych

- **Dla tego, żeby zminimalizować read miss (cache-chybienie przy odczycie), dane mają być umieszczone w pamięci głównej w kolejności ich pobrania → będziemy nazywali to kolejnością optymalną.**

- **Dane, które pozostałe umieszczone w cache, mają pozostawać tam jak najdłużej.**

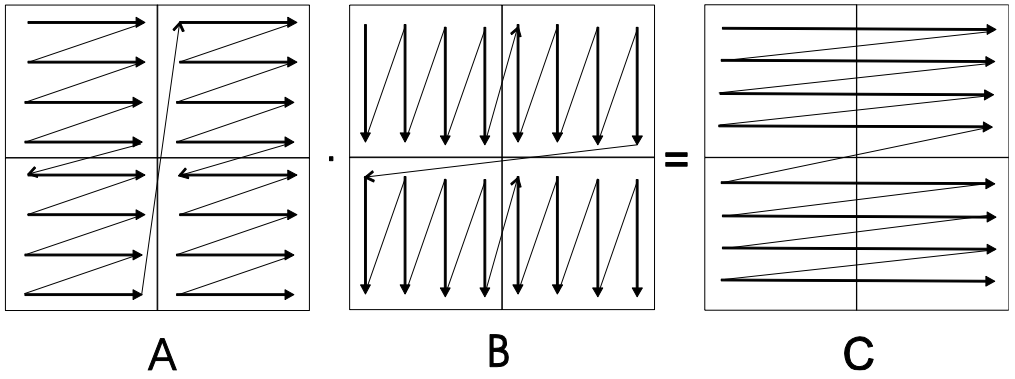
- **Przecież pierwotne umieszczenie danych w każdej z macierzy i w blokach $C_{ib,jb}$, $A_{ib,kb}$, $B_{kb,jb}$ nie odpowiada takiej kolejności → wynika konieczność przepakowania danych.**

- **Istnieje kilka różnych schematów przepakowania.**



Początkowe umieszczenie danych w macierzy A, B, C

```
kb=1,….,Nb
      //Pack A *,kb
      jb=1,….,Nb
            //Pack  Bkb,jb
            ib=1, Nb
                  Cib,jb+=Aib,kb*Bkb,jb
            end ib
      end jb
end kb
```



Mnożenie w blokach:

```
i=1, lb, 2
   j=1, lb, 2
      k=1, lb
         C^ij ib,jb+=A^ik ib,kb*B^kj kb,jb
         C^i+1,j ib,jb+=A^i+1,k ib,kb*B^kj kb,jb
         ……
      end k
   end j
end i
```

Pakowanie BLAS, ATLAS

Pakowanie BLAS, ATLAS:

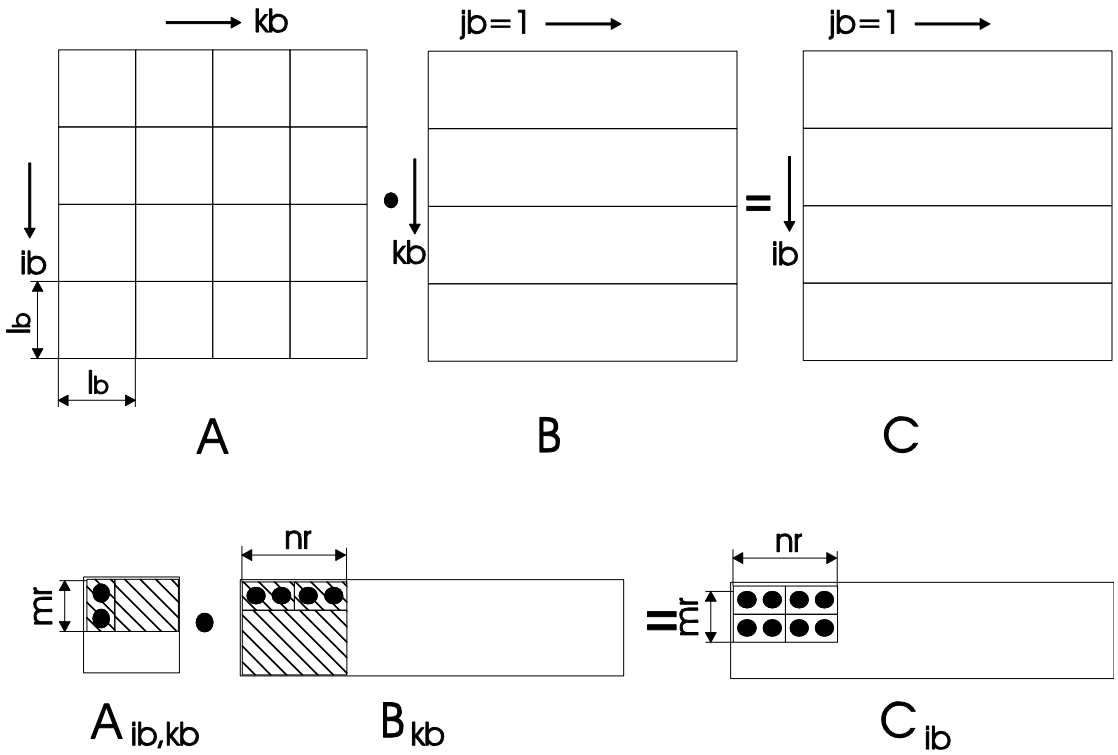
W cache L1 mają być umieszczone trzy bloki: $A_{ib,kb}$, $B_{kb,jb}$, $C_{ib,kb}$.

Z tego wynika:

$$3 \cdot l_b^{\,2} = L_1 \rightarrow \quad l_b = \sqrt{\frac{L_1}{3}}$$

Przykład: L1=32K = 32*1024 = 32 768 bajtów
albo 32 768/8 = 4 096 słów double.
l_b = 36.95 → 36 – 40

To jest optymalny rozmiar bloku dla takiego algorytmu

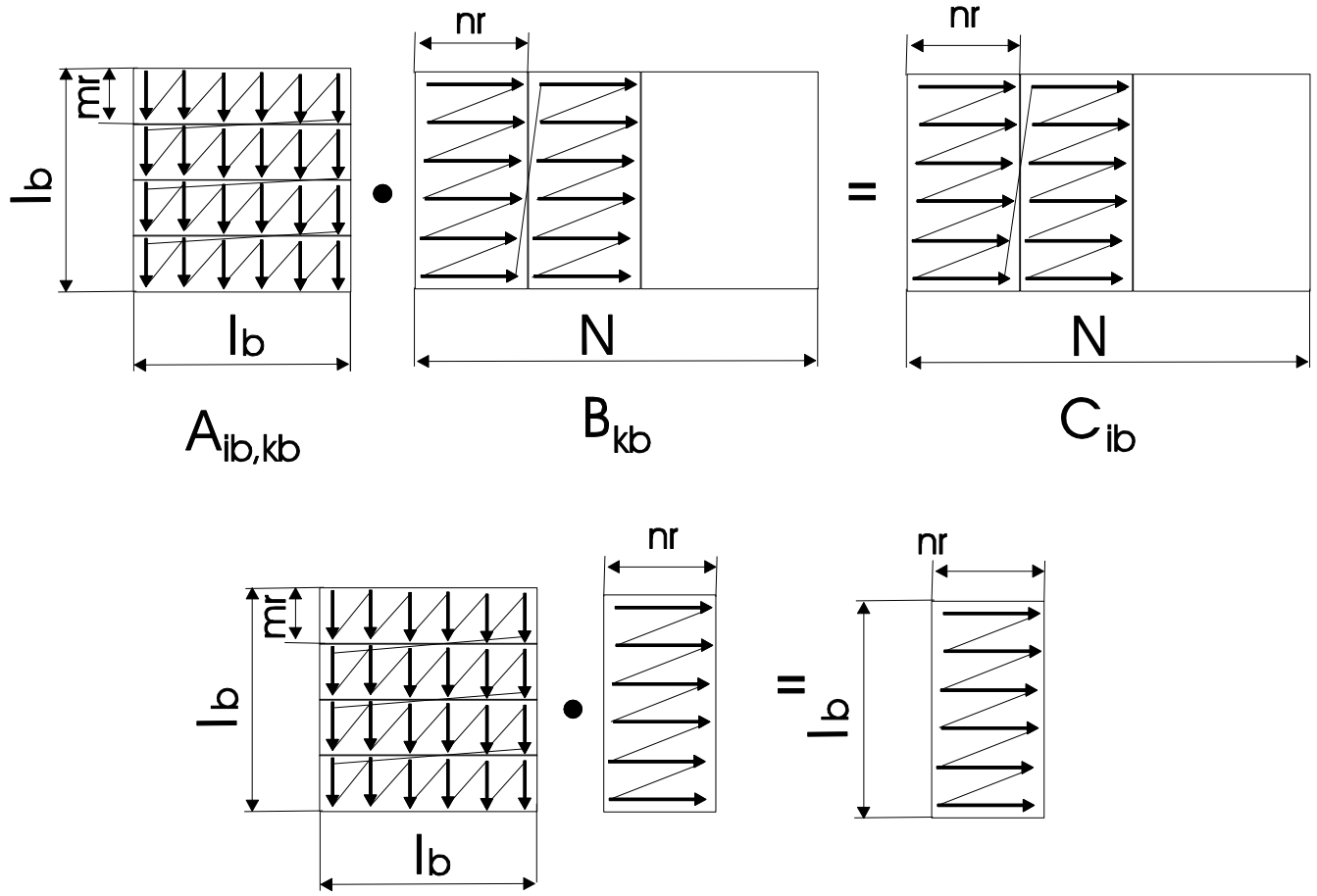


Intel MKL

```
kb=1,….,Nb
      //Pack B_kb
      ib=1,….,Nb
            //Pack A_ib,kb
            jb=1
                  C_ib+=A_ib,kb*B_kb
            end ib
      end jb
end kb
```

- **Pakowanie Intel MKL: góra – poziom macierzy; dół – poziom bloków $m_r \times n_r$ - schemat blokowania rejestrów; l_b – szerokość paneli. Z punktu widzenia wydajności obliczeń przy wykonaniu pętli wewnętrznej w cache L1 mają być umieszczone bloki danych, zakreskowane na rys, plus blok $m_r x n_r$.**

- **To jest bardziej oszczędny sposób użycia ograniczonej pamięci podręcznej, niż przedstawiony w projektach BLAS, ATLAS**



- **Pakowanie danych Intel MKL ?gemm dla schematu blokowania rejestrów wektorowych $m_r \times n_r$. Góra : mnożenie panelu macierzy przez blok; Dół – bloki, które mają być umieszczone w TLB buforze.**

Pakowanie Intel MKL:

W pętle wewnętrznej {k=0; k<lb;…} są potrzebne: jeden blok $m_r \times l_b$, jeden blok $n_r \times l_b$ i jeden blok $m_r \times n_r$. Z tego wynika:

$$l_b(m_r + n_r) + m_r \cdot n_r = L_1 \quad \rightarrow \quad l_b = \frac{L_1 - m_r \cdot n_r}{m_r + n_r} \qquad (1)$$

Z innej strony rozmiar danych, umieszczonych w cache L2, nie powinien przekroczyć rozmiaru TLB – Translation Lookaside Buffer (Dodatek 1). To jest cache CPU, używany narzędziem sterowania pamięci dla przyspieszenia translacji adresów wirtualnych w trakcie mapowania adresów wirtualnych na pamięć fizyczną. Wielu procesorów, a również procesory x86, używają tego urządzenia. Jeśli obszar danych przekracza rozmiar TLB, dostęp do tych danych idzie znacznie wolniej. Więc drugi warunek (dane, oznaczone na poprzednim rys, dół):

$$l_b^{\,2} + 2n_r l_b = TLB \quad \rightarrow \quad l_b = \sqrt{n_r^{\,2} + TLB} - n_r \qquad (2)$$

Z tych dwóch warunków trzeba wybrać ten, który doprowadzi do mniejszego rozmiaru l_b .

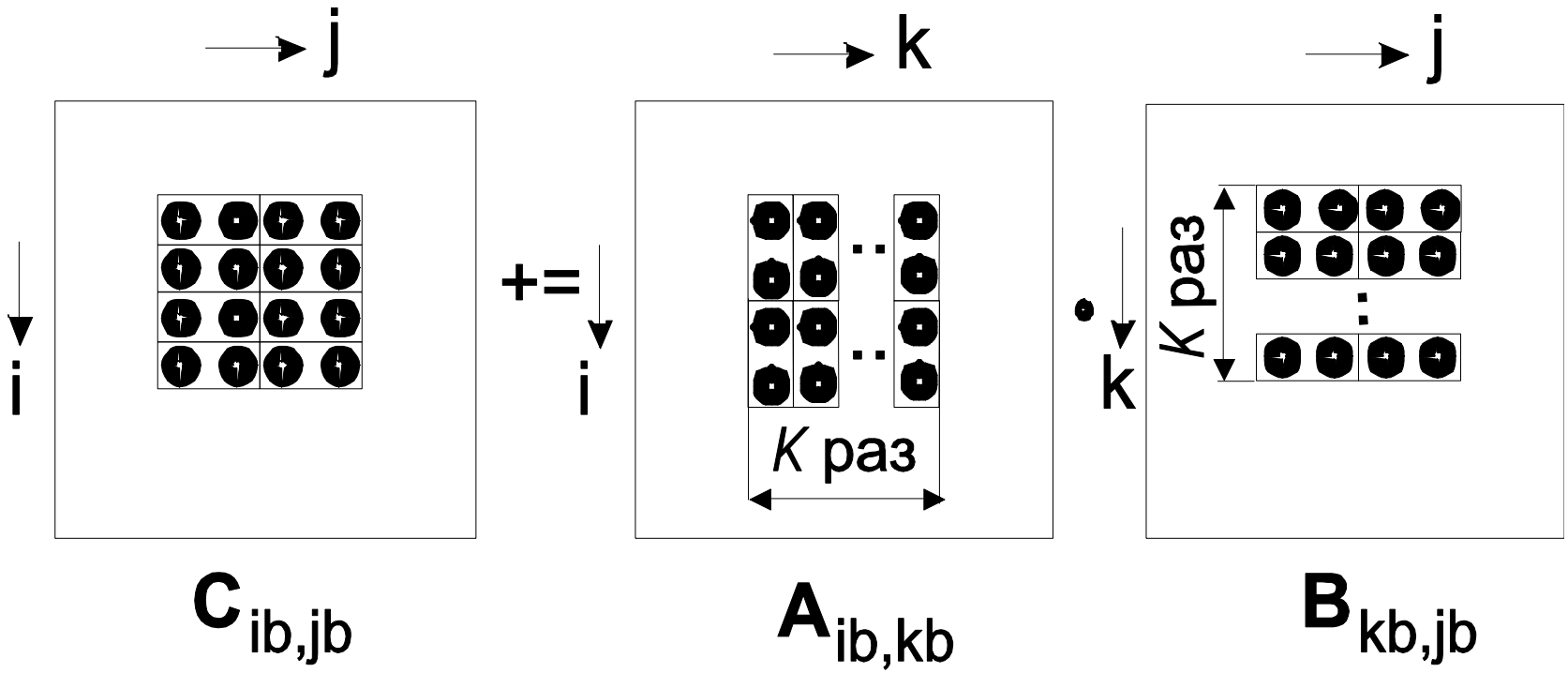
Przykład: $m_r = 2$, $n_r = 4$, $TLB = 256$ K (Pentium IV, Core Duo), $L1 = 32$ K. To oznacza: L1 = 4096 słów double, TLB = 32 768 słów double.

Ze wzoru (1) : $l_b = (4\,096 - 2*4)/(2+4) = 681$

Ze wzoru (2): $l_b = \sqrt{[2^2 + 32\,768]} - 2 = 179$

Optymalna wartość: $l_b = 176$ (musi być wielokrotna do $n_r = 4$)

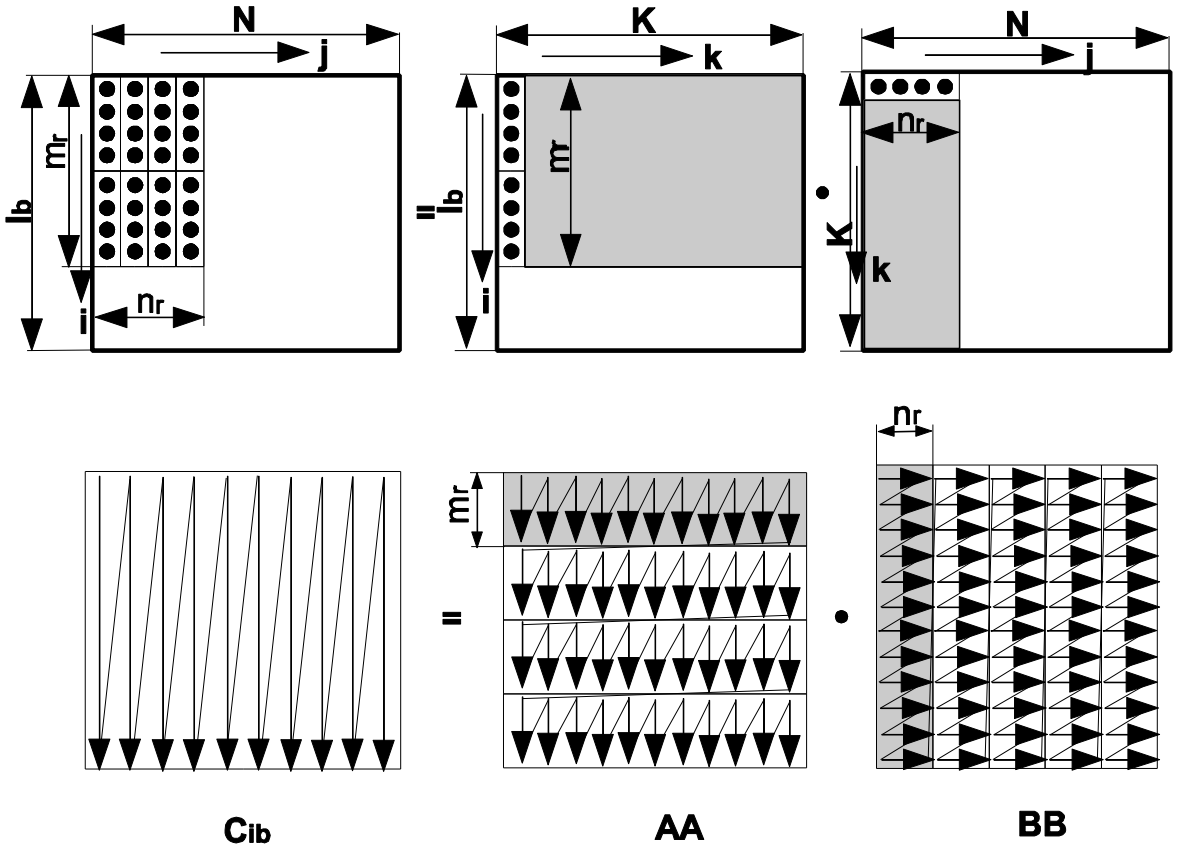
- **Technologie EM64t (system operacyjny 64 bit) dopuszczają obecność 16 128-bit rejestrów XMM. Więc register's blocking, a również i pakowanie danych myszą odpowiadać schematowi 4×4 (m_r=n_r=4) :**



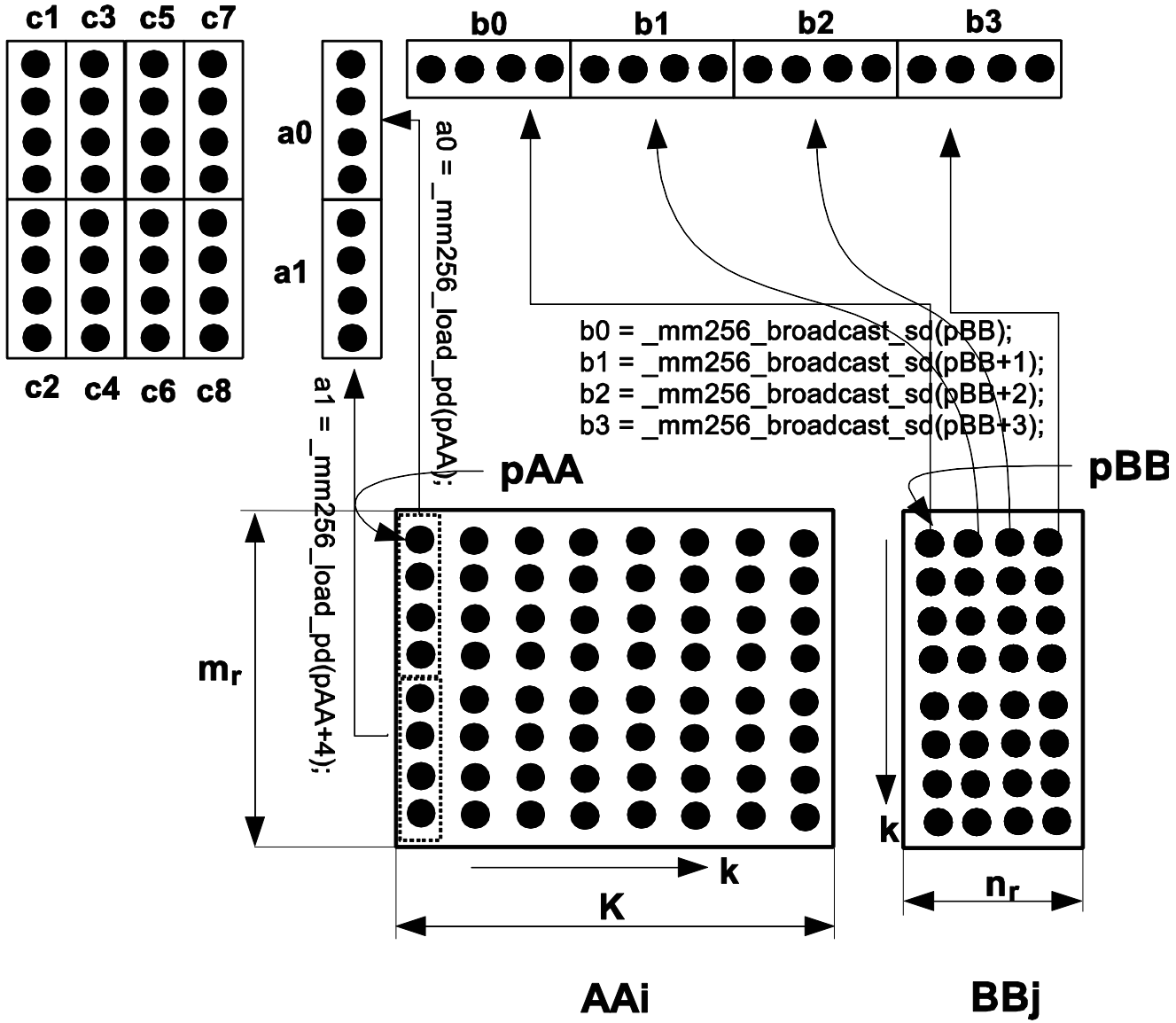
Użycie rejestrów: 8 – dla elementów macierzy $C_{ib,jb}$, 4 – dla elementów macierzy $A_{ib,kb}$, 4 – dla elementów macierzy $B_{kb,\ jb}$.

- Na każdej iteracji pętli wewnętrznej mamy: 8 ładowań w rejestry i 32 operacji arytmetyczne – q = 32/8 = 4

Advanced Vector Extension – AVX



Podział macierzy na bloki i odwzorowanie danych na rejestry YMM



Ładowanie elementów macierzy w rejestry YMM

Mikrojądro wykonane w pętle wewnętrznej po indeksu k:

```
a0 = _mm256_load_pd(pAA);
b0 = _mm256_broadcast_sd(pBB);
a1 = _mm256_load_pd(pAA+4);
b1 = _mm256_broadcast_sd(pBB+1);
b2 = _mm256_broadcast_sd(pBB+2);
b3 = _mm256_broadcast_sd(pBB+3);

mul  = _mm256_mul_pd(a0, b0);
mul1 = _mm256_mul_pd(a1, b0);
c2 = _mm256_add_pd(c2, mul1);
c1 = _mm256_add_pd(c1, mul);

mul = _mm256_mul_pd(a0, b1);
mul1 = _mm256_mul_pd(a1, b1);
c4 = _mm256_add_pd(c4, mul1);
c3 = _mm256_add_pd(c3, mul);
```

```
mul  = _mm256_mul_pd(a0, b2);
mul1 = _mm256_mul_pd(a1, b2);
c6 = _mm256_add_pd(c6, mul1);
c5 = _mm256_add_pd(c5, mul);

mul  = _mm256_mul_pd(a0, b3);
mul1 = _mm256_mul_pd(a1, b3);
c8 = _mm256_add_pd(c8, mul1);
c7 = _mm256_add_pd(c7, mul);
```

FMA: w jednej instrukcji są wykonane mnożenie i dodawanie.

Mikrojądro wykonane w pętle wewnętrznej po indeksu k:

```
a0 = _mm256_load_pd(pAA);
a1 = _mm256_load_pd(pAA+4);
b0 = _mm256_broadcast_sd(pBB);
b1 = _mm256_broadcast_sd(pBB+1);
b2 = _mm256_broadcast_sd(pBB + 2);
b3 = _mm256_broadcast_sd(pBB + 3);

c1 = _mm256_fmadd_pd(a0, b0, c1);    //c1 = c1 + a0*b0
c2 = _mm256_fmadd_pd(a1, b0, c2);    //c2 = c2 + a1*b0
c3 = _mm256_fmadd_pd(a0, b1, c3);    //c3 = c3 + a0*b1
c4 = _mm256_fmadd_pd(a1, b1, c4);    //……….
c5 = _mm256_fmadd_pd(a0, b2, c5);
c6 = _mm256_fmadd_pd(a1, b2, c6);
c7 = _mm256_fmadd_pd(a0, b3, c7);
c8 = _mm256_fmadd_pd(a1, b3, c8);
```

AVX : mnożenie + dodawanie - 8 cykli procesora
FMA: _fmadd_pd – 5 cykli procesora

Przykład: mnożenie macierzy kwadratowych o rozmiarze 1 960 × 1 960 .

Komputer: Procesor Intel Core i3 4010U (Haswell ULT) CPU@1.7 GHz
Cache: L1 = 2 × 32 KB, L2 = 2 × 256 KB, L3 = 3 MB
RAM: DDR3 4.0 GB

Wydajność, MFLOPS

SSE2	AVX	AVX FMA
5 841	12 047	17 530

Dla porównania.
Komputer z procesorem Intel Core™ i7-2760QM@2.4/3.5 GHz (Sandy Bridge) cache L1= 4 × 32 KB, L2 = 4 × 256 KB, L3 = 6 MB, RAM DDR3 8.0 GB wspiera tylko zestaw instrukcji AVX i wykazuje wydajność 19 350 MFLOPS.
To jest nie wielu lepiej od procesora i3, który na tym teście osiąga trochę mniejszej wydajności przy częstości taktowania ponad 2 razy mniejszej od i7.

Literatura do wykładu:

1. Demmel J. W., *Applied Numerical Linear Algebra*, SIAM, Philadellphia, 1997, Russian edition, Moscow, Mir, 2001.
2. Golub G.H., van Loan C. F., Matrix Computations. Third edition, The Johns Hopkins University Press, 1996.
3. Goto K, Van De Geijn R A (2008). Anatomy of High-Performance Matrix Multiplication. ACM Transactions on Mathematical Software. V 34, 3, 1–25.
4. Yotov K., Roeder T., Pingali K., Gunnels J., Gustavson F., An Experimental Comparison of Cache-oblivious and Cache-conscious Programs. SPAA'07, June 9–11, 2007, San Diego, California, USA.
5. Fialko S. Blokowa wielofrontalna metoda podstruktur do rozwiązywania dużych układów równań MES. Czasopismo techniczne, 1-NP/2009, 175 – 188.
6. Fialko S. Application of AVX (Advanced Vector Extensions) for Improved Performance of the PARFES – Finite Element Parallel Direct Solver. Federated Conference on Computer Science and Information Systems, September 8–11, 2013. Kraków, Poland. IEEE Xplore Digital Library, pp. 447 – 454. https://fedcsis.org/proceedings/2013/pliks/98.pdf

Dodatek 1

TLB (Goto K, Van De Geijn R A (2008). Anatomy of High-Performance Matrix Multiplication. ACM Transactions on Mathematical Software. V 34, 3, 1–25)

4.2.2 *TLB considerations. A second architectural consideration relates to the* page management system. For our discussion it suffices to consider that a typical modern architecture uses virtual memory so that the size of usable memory is not constrained by the size of the physical memory: Memory is partitioned into pages of some (often fixed) prescribed size. A table, referred to as the *page table maps* virtual addresses to physical addresses and keeps track of whether a page is in memory or on disk. The problem is that this table itself is stored in memory, which adds additional memory access costs to perform virtual to physical translations. To overcome this, a smaller table, the Translation Look-aside Buffer (TLB), that stores information about the most recently used pages, is kept. Whenever a virtual address is found in the TLB, the translation is fast. Whenever it is not found (a TLB miss occurs), the page table is consulted and the resulting entry is moved from the page table to the TLB. In other words, the TLB is a cache for the page table. More recently, a level 2 TLB has been introduced into some architectures for reasons similar to those that motivated the introduction of an L2 cache.

The most significant difference between a cache miss and a TLB miss is that a cache miss does not necessarily stall the CPU. A small number of cache misses can be tolerated by using algorithmic prefetching techniques as long as the data can be read fast enough from the memory where it does exist and arrives at the CPU by the time it is needed for computation. A TLB miss, by contrast, causes the CPU to stall until the TLB has been updated with the new address. In other words, prefetching can mask a cache miss but not a TLB miss.