Struktura programu

Projekty złożone składają się zwykłe z różnych plików. Zawartość każdego pliku programista wyznacza zgodnie z jego przeznaczeniem.

W ostatnich latach najbardziej używanym stylem oprogramowania jest podejście obiektowe. Język C bezpośrednio nie ma narzędzi do oprogramowania obiektowego, przecież nadaje możliwość wytrzymać pewne cechę stylu oprogramowania obiektowego.

- **W podstawie tego stylu leżą dane i metodę (funkcji - C), które tworzą te dane, alokują pamięć, zmieniają dane, usuwają ich, zwalniają pamięć, a również wykonują dowolne działania nad danymi.**

- **Bezpośredni dostęp do danych jest możliwy wyłącznie z tego pliku, w którym te dane są zdefiniowane. Z innych plików programu bezpośredni dostęp do danych nie jest popierany.**

Do programisty należy:

• Podjęć decyzje – jak rozdzielić dane na grupy, przy czym każda grupa danych będzie umieszczona w swój plik i będzie obsługiwana swoimi funkcjami.

• Dla każdej grupy danych rozdzielić funkcji obsługi na funkcje wewnętrzne, wykonujące działania nad danymi tylko w obszarze pliku, w którym dane są zdefiniowane, i na funkcje interfejsowe, jakie udostępniają działania nad danymi w innych plikach.

Dla tego, żeby zabezpieczyć ten styl oprogramowania, należy (przykład W7_3 (st. zaoczne - samodzielnie):

• podstawowe dane w pliku definiować jako zmienne globalne z modyfikatorem *static*, który nie nadaje możliwości ich bezpośredniego udostępniania w innych plikach.

- **Prototypy funkcji interfejsowych umieścić w odpowiednim headerze (bez modyfikatora *static*).**

- **Prototypy funkcji wewnętrznych umieścić w pliku jako obiekty globalne z modyfikatorem *static*.**

- **Dane globalne, które nie są docelowo udostępniać przez funkcje, definiować bez modyfikatora *static*, a udostępnienie w innych plikach wprowadzić z specyfikatorem *extern*. Nie warto umieszczać te dane w headerze.**

- **Dane pomocnicze (indeksy tablic, tymczasowe zmienne i inne) używać jako dane lokalne (auto).**

• Headery (pliki nagłówkowe) są używane dla deklaracji typów danych (struktur, unij, typedef), pragm kompilatorów, makrosów, prototypów funkcji, które muszą być widoczne w różnych plikach projektu.

• Każdy header musi mieścić tak zwane straże, które w projektach złożonych chronią przed duplikacją definicji, umieszczonych w headerze.

Klasy pamięci

• *auto* – wszystkie zmienne i tablicy, zdefiniowane w dowolnym bloku. Rezerwacja miejsca w pamięci następuje przy wejściu w blok. Pamięć ta zostaje zwolniona przy wyjściu z bloku, w którym była zdefiniowana ta zmienna. Dane będą utracone. Obszar działania (widoczności, zasiąg deklaracji) – blok. Czas życia – od momentu definicji i do momentu wyjścia z bloku.
Domyślnie zmienne nie są inicjowane, zawierają „śmieci”.

- ***static*** **(definicja lokalna – wewnątrz bloku) – rezerwacja pamięci jest dokonywana na początku wykonania programu. Obszar widoczności zmiennych i tablic klasy *static* – od miejsca definicji w bloku, w którym oni są zdefiniowane, do końca bloku. Czas życia – czas wykonania całego programu. Oznacza to, że wartości zmiennych klasy pamięci *static* są chronione nawet po wyjściu z bloku i zostają aktualne po następnemu wejściu w blok. Przy braku jawnej inicjalizacji zmienne klasy *static* są inicjowane jako zero.**

Przykład:

```
void fun()
{
        static int a;
        a++;
}
pierwsze wejście – a = 0
            wyjście – a = 1
drugie wejście     – a = 1
            wyjście – a = 2
```

```
void fun()
{
        int a = 0;
        a++;
}
pierwsze wejście – a = 0
przed   wyjściem – a = 1
drugie wejście     – a = 0
przed wyjściem     – a = 1
```

Statyczna składowa struktury:

```
struct MY_STRUCT
{
        ……..
        static int a;
};

//konieczne jest inicjowanie jako zmiennej globalnej:

        int MY_STRUCT :: a = 0;

//Wszystkie obiekty MY_STRUCT rozdzielają tą samą zmienną a:

        MY_STRUCT A, B;
        A.a  i B.a mają tą samą wartość.
        A.a = 10;
        printf("%d\n", B.a);   //B.a = 10
```

```
MY_STRUCT tab[5];
tab[0].a, tab[1].a, …,  mają tą samą wartość.
tab[0].a = 10;
printf("%d\n", tab[1].a);   //10
```

Funkcja zwraca wskaźnik do obiektu:

```
int *fun()
{
        int i = 10;
        return &i;
}

//!OK
```

```
int *fun()
{
        static int i = 10;
        return &i;
}

//OK
```

```
int *fun()
{
      int *pi = (int *)malloc(sizeof(int));
      *pi = 10;
      return pi;
}   //OK
```

```
int i;

int *fun()
{
        i = 10;
        return &i;
}  //OK
```

- ***static*** **(definicja globalna – w pliku poza blokami) – rezerwacja pamięci jest dokonywana na początku wykonania programu. Obszar widoczności zmiennych i tablic klasy pamięci *static* – od miejsca definicji do końca pliku. Czas życia – czas wykonania całego programu. Przy braku jawnej inicjalizacji zmienne są inicjowane jako zero. Są dostępne tylko w podanym pliku.**

- **Zmienne globalne (bez specyfikatora *static*) - rezerwacja pamięci jest dokonywana na początku wykonania programu. Obszar widoczności zmiennych i tablic globalnych – od miejsca definicji do końca pliku. Czas życia – czas wykonania całego programu. Przy braku jawnej inicjalizacji zmienne są inicjowane jako zero. Mogą być udostępnione w innych plikach.**

- **Klasa pamięci register – oznacza, że dane, gdy to będzie możliwe, pozostaną umieszczone w rejestrze procesora.**

- **Specyfikator *static*, używany przy prototypie funkcji, ogranicza działalność tej funkcji obszarem pliku. Nie wolno umieszczać deklaracji funkcji statycznych w headerzach.**

Wektory i macierzy

Podstawowe działania algebry liniowej

1. **Iloczyn skalarny dwóch wektorów: scalar = $x^T y$. Przykład W7_4.**
2. **Mnożenie macierzy przez wektor: y=Ax. Przykład W7_5.**
3. **Mnożenie macierzy przez macierz: C = C + A*B. Przykład W7_6.**
4. **Faktoryzacja macierzy kwadratowej: A = L*U, A = $L*L^T$**

Zrównoleglenie i przetwarzanie potokowe

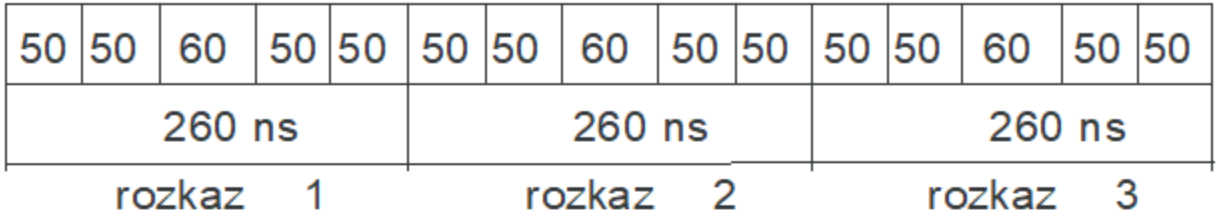
- **Zrównoleglenie – wysoka wydajność pozostaje osiągnięta w efekcie jednoczesnego wykonania różnych części zagadnienia.**

- **Przetwarzanie potokowe – proces jest rozdzielony na drobne części – stopni przebiegu. Każdy stopień będzie wykonany oddzielnym blokiem sprzętu. Przetwarzanie dowolnego rozkazu można rozdzielić na kilka stopni i zorganizować przekazanie danych od jednego etapu do innego. Przetwarzanie potokowe można wykorzystać dla skojarzenia etapów wykonania różnych rozkazów. Wydajność rośnie w efekcie tego, że na różnych stopniach przetwarzania potokowego jednocześnie są wykonane kilka rozkazów.**

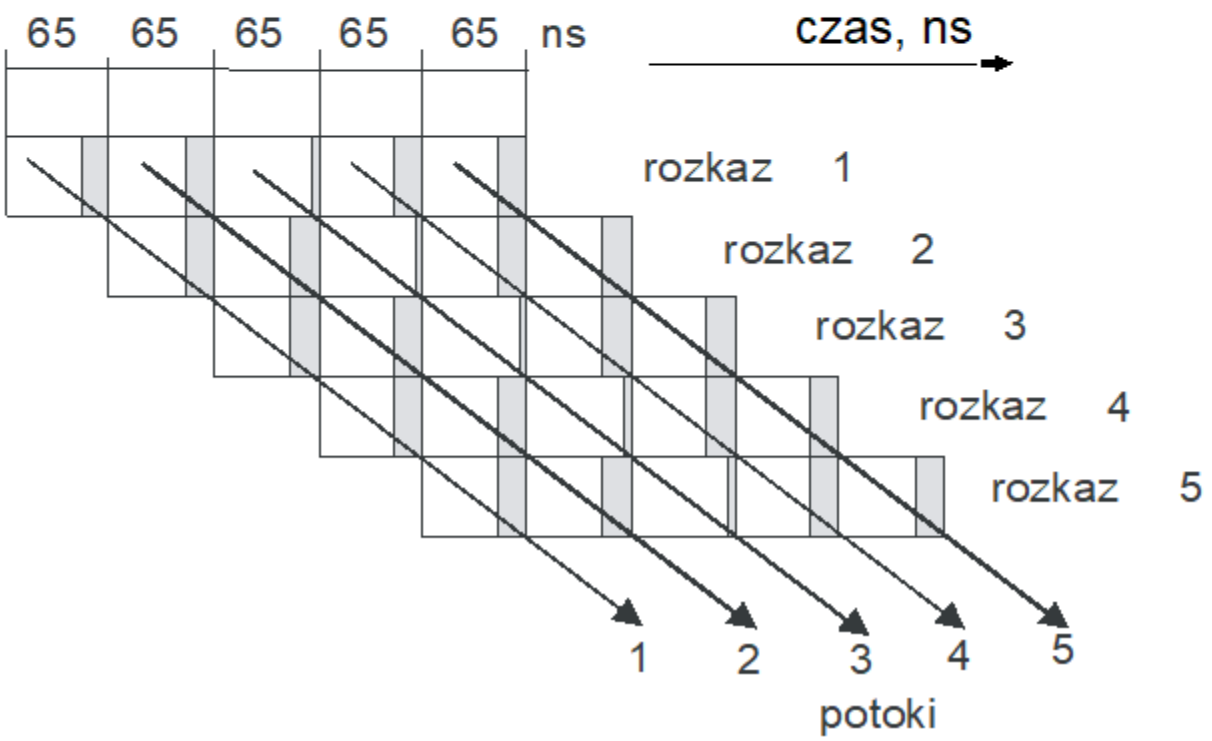
Przykład: mamy 5 etapów wykonania operacji trwałością 50, 50, 60, 50 i 50 ns odpowiednio, a 5 ns – nakład na organizacje przetwarzania potokowego. Średni czas wykonania rozkazu przy przetwarzani sekwencyjnym wynosi 260 ns, a przy przetwarzani potokowym (t_{sr}^{pot}) – trwałości najdłuższego taktu plus nakład na przetwarzanie potokowe – 60+5 = 65 ns [t_{sr}^{pot} = (maksymalna trwałość etapu)*(ilość etapów)/(ilość potoków) = 65*5/5 = 65 ns]. Więc przyspieszenie wynosi 260/65 = 4 razy.

Większość współczesnych procesorów używają przetwarzanie potokowe.

Sekwencyjne przetwarzanie rozkazów

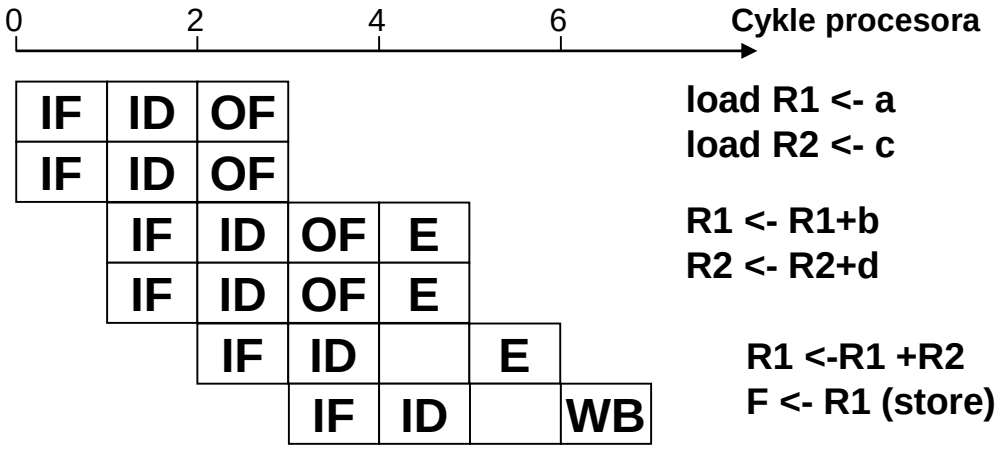


Potokowe przetwarzanie rozkazów



Superskalarność procesora – możliwość wykonywania wektoryzacji elementów obliczeń przy technologii potokowej i istnieniu rejestrów specjalnych.

- **Przykad: f = a+b+c+d;**



IF – instruction fetching

ID – decoding

OF – operand fetching

E – execution

WB – Write-back

Mnożenie macierzy przez wektor: y = Ax

$$\begin{matrix} & & \rightarrow & j \\ \downarrow & \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} & \cdot & \downarrow \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \downarrow \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \\ i & & & j \quad\quad\quad\quad i \end{matrix}$$

Ilość odczytów stron pamięci - *nPg*

$$columna - po - kolumnie$$

0	1	2	3	4	5	6	7	8
a_{11}	a_{21}	a_{31}	a_{12}	a_{22}	a_{32}	a_{13}	a_{23}	a_{33}

Algorithm 1 (row_by_row)

nPg = n^2

$i = 1, n$
  $r \leftarrow 0$
  $j = 1, n$
    $ij = n(j-1)+i-1;$
    $r = r + A_{ij} x_j$
  end do
  $y_j \leftarrow r$
end do

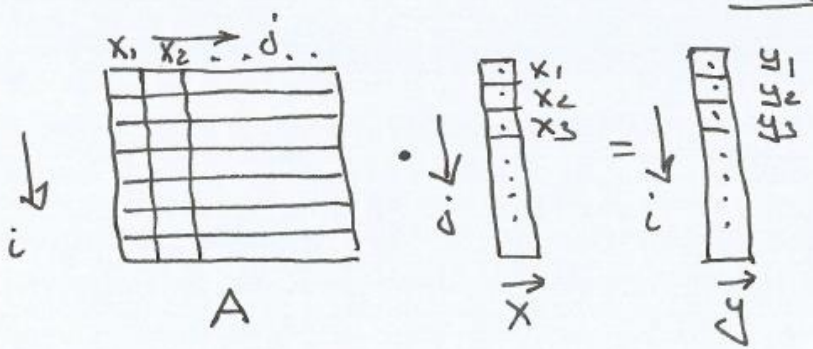
Algorithm 2 (col_by_col)

nPg = n^2/M,
M - Ilość słów *double* na jednej stronie

$\mathbf{y} \leftarrow 0$
$j = 1, n$
  $r \leftarrow x_j$
  $i = 1, n$
    $ij = n(j-1)+i-1;$
    $y_i = y_i + A_{ij} r$
  end do
end do

Mnożenie macierzy przez wektor: y = Ax

Algorytm naiwny



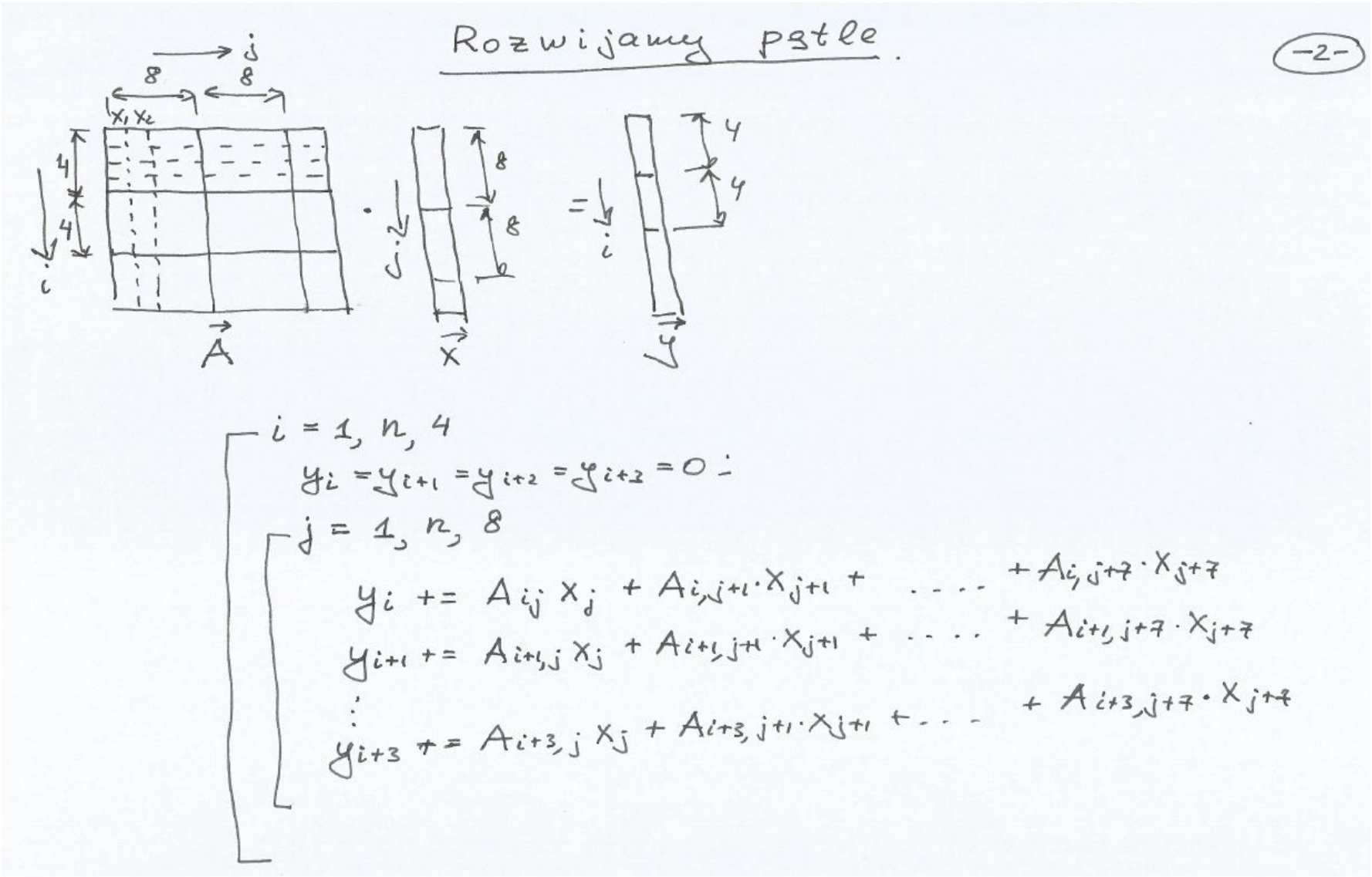

```
i = 1, n
  y_i ← 0
  j = 1, n
    y_i += A_ij · x_j
```

ładowanie do rejestrów elementów A: n^2

ładowanie do rejestrów elementów $\vec{x}$: n^2

ładowanie do rejestrów elementów $\vec{y}$: $2n$

Mnożenie macierzy przez wektor: y = Ax



$$
\begin{aligned}
&i = 1, n, 4 \\
&\quad y_i = y_{i+1} = y_{i+2} = y_{i+3} = 0 \\
&\quad j = 1, n, 8 \\
&\qquad y_i \mathrel{+}= A_{ij} x_j + A_{i,j+1} \cdot x_{j+1} + \ldots + A_{i,j+7} \cdot x_{j+7} \\
&\qquad y_{i+1} \mathrel{+}= A_{i+1,j} x_j + A_{i+1,j+1} \cdot x_{j+1} + \ldots + A_{i+1,j+7} \cdot x_{j+7} \\
&\qquad \vdots \\
&\qquad y_{i+3} \mathrel{+}= A_{i+3,j} x_j + A_{i+3,j+1} \cdot x_{j+1} + \ldots + A_{i+3,j+7} \cdot x_{j+7}
\end{aligned}
$$

Mnożenie macierzy przez wektor: y = Ax

Wady i zalety

- Bardziej skompikowany kod. – wada
- Rozwijanie pętli i – każdy element wektora $\vec{x}$ jeden raz ładuje się do rejestru i 4 razy uczestniczy w operacjach arytm. ilość ładowań – $n^2/4$;
- Rozwijanie pętli j – podnosi potokowe przetwarzanie danych i pomaga prefetch.

Przechowywanie macierzy (tablicy jednowymiarowej) „wierz-po-wiersze”

i – numer wiersza, j – numer kolumny, kolejność zmiany indeksów:

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
```

Indeks ij ⟶

ij=n*i+j

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \quad \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Przechowywanie macierzy (tablicy jednowymiarowej) „kolumna-po-kolumnie”

i – numer wiersza, j – numer kolumny, kolejność zmiany indeksów:

```
for(j=0; j<n; i++)
    for(i=0; i<n; i++)
```

Indeks ij ⟶

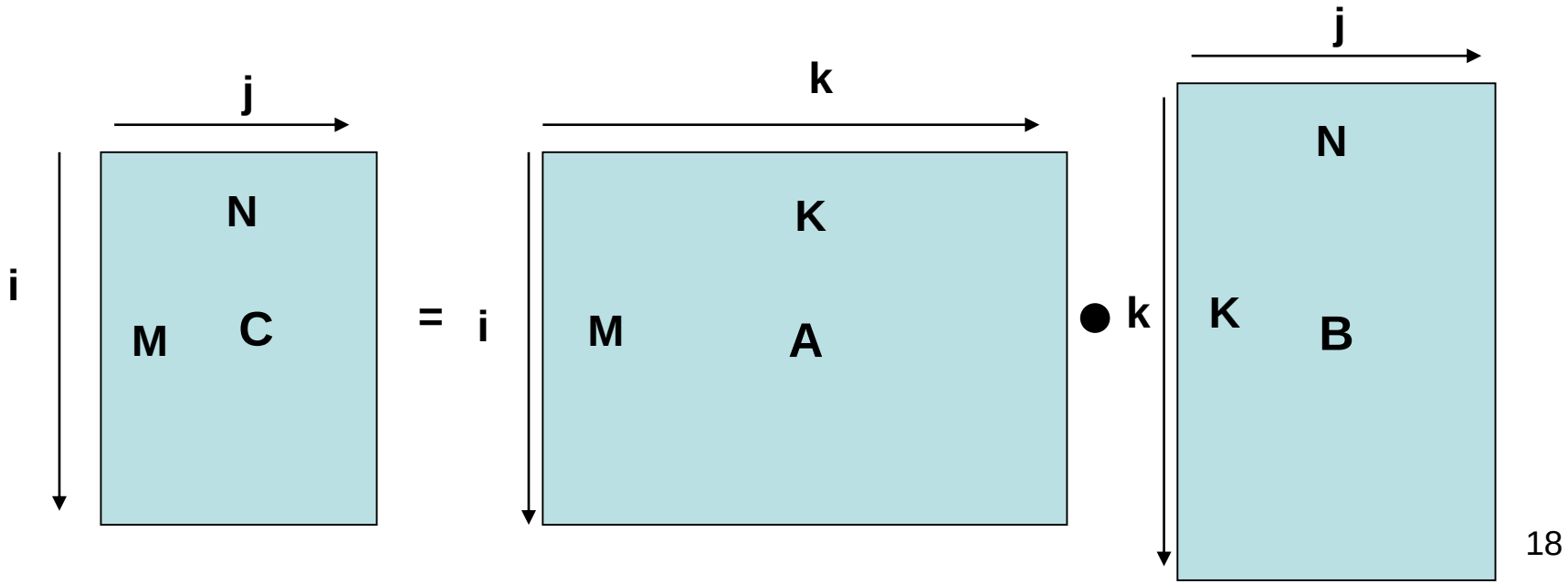
ij=n*j+i

$$\begin{pmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{pmatrix} \quad \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Mnożenie macierzy przez macierz

Wzór matematyczny: $c_{ij}=\sum_k a_{ik}\cdot b_{kj}$

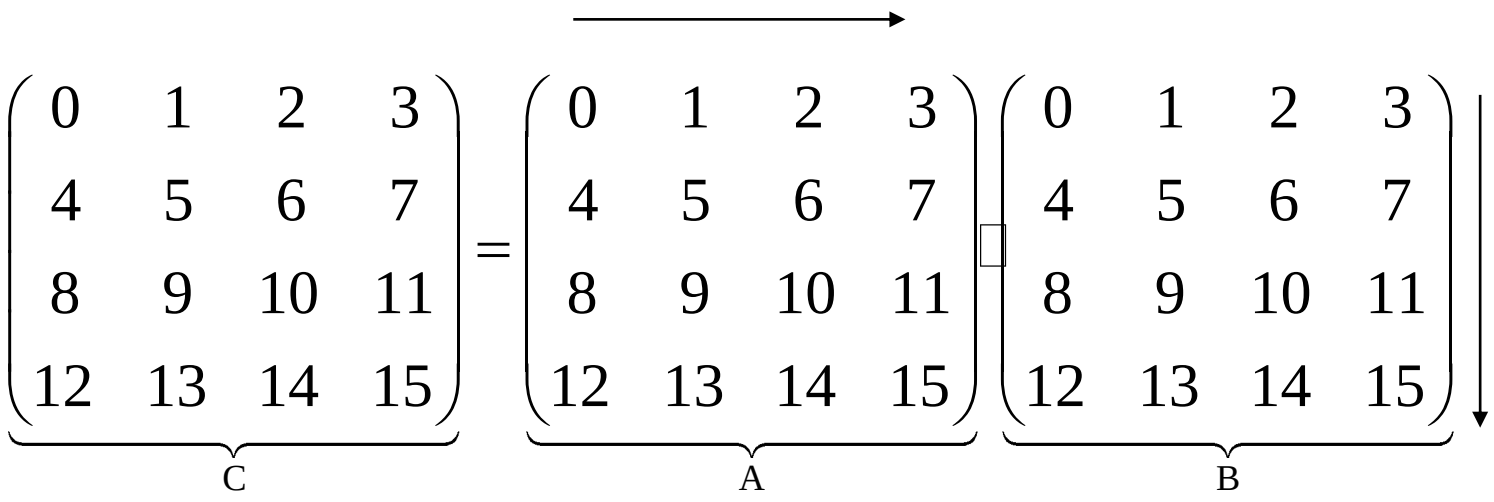
$$\begin{pmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \square \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{pmatrix}$$



Metoda klasyczna – przechowywanie macierzy wiersz po wiersze

```
for(i=0; i<M; i++)
    for(j=0; j<N; j++)
        for(k=0; k<K; k++)
            Cij = Cij + Aik * Bkj
```

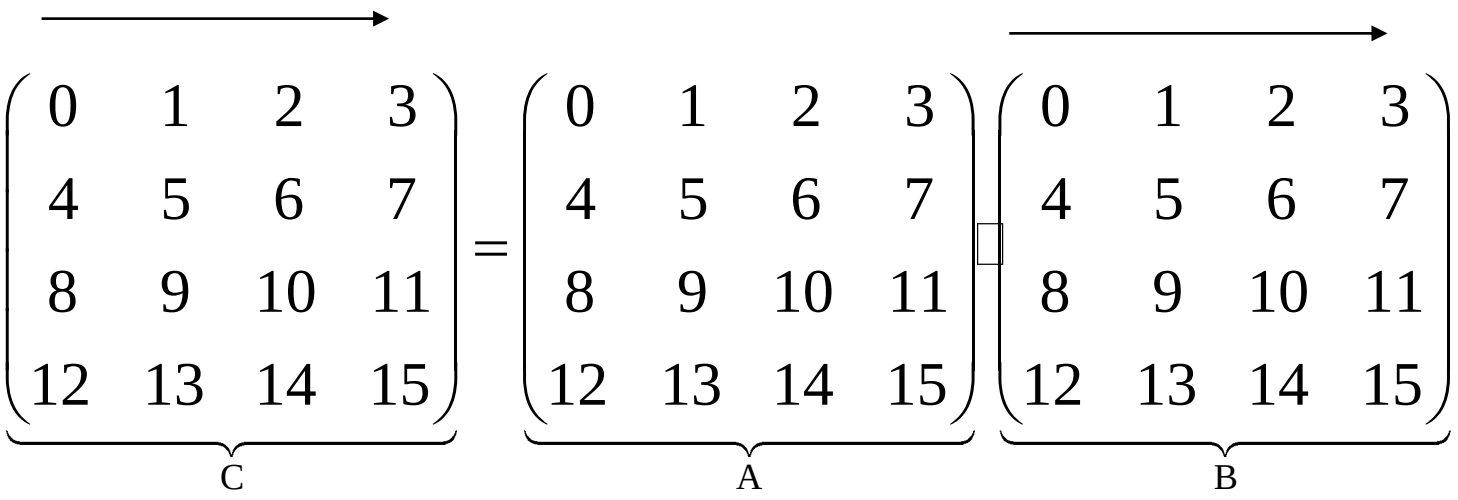
Element macierzy C w pętli wewnętrznej nie zależy od indeksu k.
Pobieranie elementów macierzy A odbywa się ciągłe, bez skoków.
Pobieranie elementów macierzy B odbywa się ze skokami.
Dla tego ta metoda nie skuteczna



Metoda przyspieszona – przechowywanie macierzy wiersz po wiersze

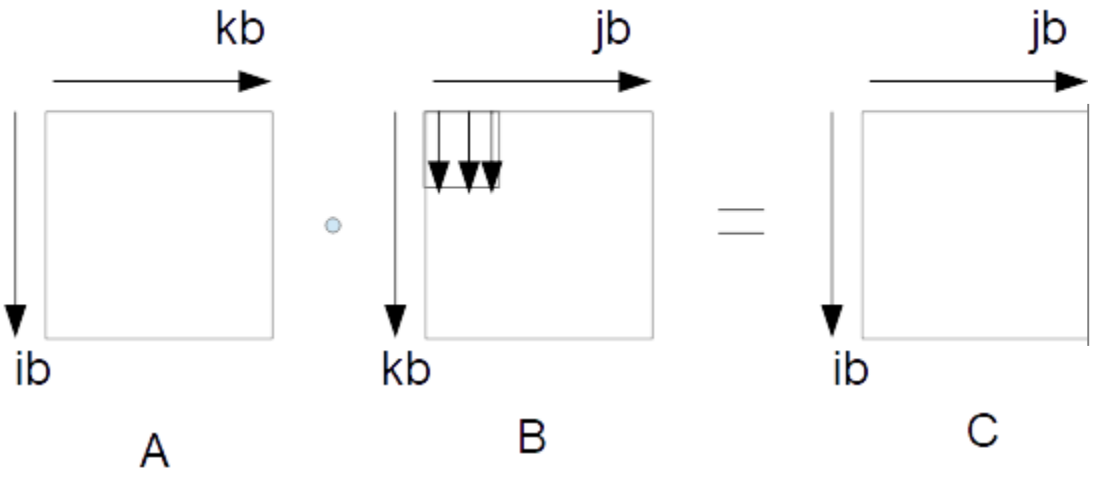
```
for(i=0; i<M; i++)
    for(k=0; k<K; k++)
        for(j=0; j<N; j++)
            Cij = Cij + Aik * Bkj
```

Elementy macierzy A nie zależą od indeksu j pętli wewnętrznej.
Pobieranie elementów macierzy C idzie ciągłe, bez skoków.
Pobieranie elementów macierzy B idzie ciągłe, bez skoków.
Dla tego ta metoda jest lepsza od poprzedniej



Blokowanie pamięci podręcznej, blokowanie rejestrów 2x2.

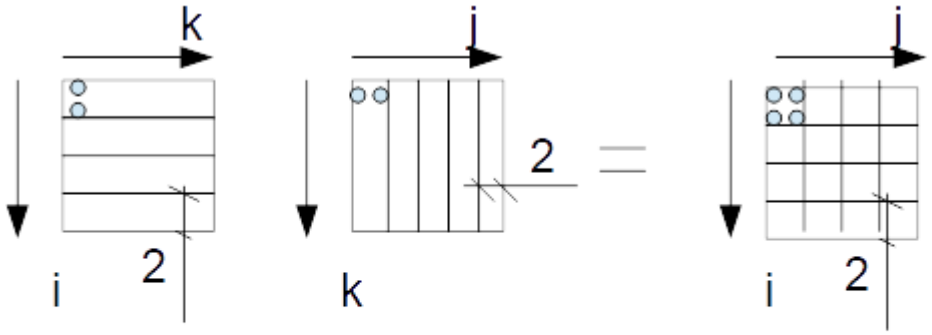
Blokowanie pamięci podręcznej




```
kb = 1, Nb
    jb = 1, Nb
        ib = 1, Nb
            Cib,jb += Aib,kb•Bkb,jb
```

Blokowanie pamięci podręcznej, blokowanie rejestrów 2x2.

Blokowanie rejestrów 2x2



Aib,kb Bkb,jb Cib,jb

```
i=1,Ib
   j=1,Ib
      k=1,Ib
         Ci,j += Ai,k•Bk,j

q=2/2=1
```

```
i=1,Ib,2
   j=1,Ib,2
      k=1,Ib
         Ci,j   += Ai,k•Bk,j
         Ci+1,j += Ai+1,k•Bk,j
         Ci,j+1 += Ai,k•Bk,j+1
         Ci+1,j+1 += Ai+1,k•Bk,j+1

q=8/4=2
```

Rozdzielenie macierzy na bloki

Mnożenie blokowe – przykład Mulm

$$A = \begin{pmatrix} \{a\}_{1,1} & \{a\}_{1,2} \\ \{a\}_{2,1} & \{a\}_{2,2} \end{pmatrix} = \left(\begin{array}{ccccc} a_{11} & a_{12} & : & a_{13} & a_{14} \\ a_{21} & a_{22} & : & a_{23} & a_{24} \\ . & . & . & . & . \\ a_{31} & a_{32} & : & a_{33} & a_{34} \\ a_{41} & a_{42} & : & a_{43} & a_{44} \end{array}\right) \rightarrow \left(\begin{array}{ccccc} 0 & 1 & : & 2 & 3 \\ 4 & 5 & : & 6 & 7 \\ . & . & . & . & . \\ 8 & 9 & : & 10 & 11 \\ 12 & 13 & : & 14 & 15 \end{array}\right)$$

#include <memory.h>

void *memcpy(void *_dest_, const void *_src_, size_t _count_);

Kopiuje _count_ bajtów z _src_ do _dest_. Zwraca wskaźnik do _dest_. Działanie nie jest przewidywane, jeśli obszary pamięci src i dest przekrywają się.

void *memmove(void *_dest_, const void *_src_, size_t _count_);

Kopiuje _count_ bajtów z _src_ do _dest_. Zwraca wskaźnik do _dest_. Jeśli jakieś obszary pamięci przekrywają się, funkcja gwarantuje, że oryginalne bajty źródłowe w przekrywającym się regionie będą skopiowany przed tym, jak pozostaną przepisane.

void *memset(void *_dest_, int _c_, size_t _count_);

Przypisuje symbol _c_ pierwszym count symbolom tablicy _dest._ Zwraca wskaźnik do _dest_.

```
#include <memory.h>
#include <string.h>
#include <stdio.h>

char str1[7] = "aabbcc";

int main( void )
{
    printf( "The string: %s\n", str1 );
    //Przekrywający się region! Kopiowanie może być nie poprawne
    memcpy( str1 + 2, str1, 4 );
    printf( "New string: %s\n", str1 );

    strcpy_s( str1, sizeof(str1), "aabbcc" ); // reset string
    printf( "The string: %s\n", str1 );
   //Przekrywający się region! Kopiowanie poprawne
    memmove( str1 + 2, str1, 4 );
    printf( "New string: %s\n", str1 );
 }
```

```
The string: aabbcc
New string: aaaabb
```

```
#include <memory.h>
#include <stdio.h>

int main( void )
{
    char buffer[] = "This is a test of the memset function";
    printf( "Before: %s\n", buffer );
    memset( buffer, '*', 4 );
    printf( "After: %s\n", buffer );
}

Before: This is a test of the memset function
After   : **** is a test of the memset function
```

Przykład 2.

```
char str[256];
double aa[200];
double cc[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
double dd[5] = { 20, 30, 40, 50, 60};

memset((void *)str, 0, 256*sizeof(char));          //OK
memset((void *)str, ' ', 256*sizeof(char));        //!OK – pisanie po pamięci
memset((void *)str, ' ', 255*sizeof(char));        //OK

memset((void *)aa, 0, 200*sizeof(double));         //OK

memcpy((void *)&cc[3], (const void *)&dd[2], 2*sizeof(double));

//cc: 1  2  3  40  50  6  7  8  9  10
```