Programowanie, część V

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

24 maja 2011

Algorytm eliminacji Gaussa

Przypuśćmy, że chcemy znaleźć rozwiązanie równania postaci $\widehat{A}\mathbf{x} = \mathbf{b}$, gdzie $\widehat{A}$ jest macierzą (o jednakowej liczbie wierszy i kolumn), a $\mathbf{b}$ jest wektorem o zadanych współrzędnych.

Algorytm eliminacji Gaussa

Przypuśćmy, że chcemy znaleźć rozwiązanie równania postaci $\widehat{A}\mathbf{x} = \mathbf{b}$, gdzie $\widehat{A}$ jest macierzą (o jednakowej liczbie wierszy i kolumn), a $\mathbf{b}$ jest wektorem o zadanych współrzędnych.

W ogólności można sprowadzić to równanie do równania $\widehat{U}\mathbf{x} = \mathbf{b}'$, gdzie $\widehat{U}$ jest macierzą trójkątną.

Algorytm eliminacji Gaussa

Przypuśćmy, że chcemy znaleźć rozwiązanie równania postaci $\widehat{A}\mathbf{x} = \mathbf{b}$, gdzie $\widehat{A}$ jest macierzą (o jednakowej liczbie wierszy i kolumn), a $\mathbf{b}$ jest wektorem o zadanych współrzędnych.

W ogólności można sprowadzić to równanie do równania $\widehat{U}\mathbf{x} = \mathbf{b}'$, gdzie $\widehat{U}$ jest macierzą trójkątną.

Wtedy rozwiązania oryginalnego równania można otrzymać poprzez tak zwane podstawienie wsteczne (*ang. backward substitution*).

Algorytm eliminacji Gaussa

Przypuśćmy, że chcemy znaleźć rozwiązanie równania postaci $\widehat{A}\mathbf{x} = \mathbf{b}$, gdzie $\widehat{A}$ jest macierzą (o jednakowej liczbie wierszy i kolumn), a $\mathbf{b}$ jest wektorem o zadanych współrzędnych.

W ogólności można sprowadzić to równanie do równania $\widehat{U}\mathbf{x} = \mathbf{b}'$, gdzie $\widehat{U}$ jest macierzą trójkątną.

Wtedy rozwiązania oryginalnego równania można otrzymać poprzez tak zwane podstawienie wsteczne (*ang. backward substitution*).

Algorytm Gaussa określa sposób wyznaczenia macierzy $\widehat{U}$ zwany eliminacją do przodu (*ang. forward elimination*).

Eliminacja do przodu

Aby możliwe było przeprowadzenie obliczeń tą metodą, macierz $\widehat{A}$ musi spełniać określone warunki. Załóżmy, że są one spełnione.

Eliminacja do przodu

Aby możliwe było przeprowadzenie obliczeń tą metodą, macierz $\widehat{A}$ musi spełniać określone warunki. Załóżmy, że są one spełnione.

W pierwszym kroku obliczamy macierz $\widehat{A}^{(1)}$, dla której:

1. Pierwszy wiersz jest identyczny z pierwszym wierszem macierzy $\widehat{A}$.
2. W każdym z pozostałych wierszy pierwszy element macierzowy jest zerem:

$$A_{ij}^{(1)} = A_{ij} - \frac{A_{i1}}{A_{11}} A_{1j}$$

Eliminacja do przodu

Aby możliwe było przeprowadzenie obliczeń tą metodą, macierz $\widehat{A}$ musi spełniać określone warunki. Załóżmy, że są one spełnione.

W pierwszym kroku obliczamy macierz $\widehat{A}^{(1)}$, dla której:

1. Pierwszy wiersz jest identyczny z pierwszym wierszem macierzy $\widehat{A}$.
2. W każdym z pozostałych wierszy pierwszy element macierzowy jest zerem:

$$A^{(1)}_{ij} = A_{ij} - \frac{A_{i1}}{A_{11}} A_{1j}$$

Jednocześnie obliczamy wektor $\mathbf{b}^{(1)}$ taki, że $b^{(1)}_1 = b_1$ oraz (dla $i > 1$)

$$b^{(1)}_i = b_i - \frac{A_{i1}}{A_{11}} b_1$$

Eliminacja do przodu (c. d.)

Inaczej mówiąc mnożymy równanie 1 w oryginalnym układzie równań przez A_{i1}/A_{11} i odejmujemy wynik od równania i (powtarzając dla wszystkich $i > 1$).

Eliminacja do przodu (c. d.)

Inaczej mówiąc mnożymy równanie 1 w oryginalnym układzie równań przez A_{i1}/A_{11} i odejmujemy wynik od równania i (powtarzając dla wszystkich $i > 1$).

W drugim kroku równania 1 i 2 pozostawiamy bez zmian, zaś dla $i > 2$ równanie 2 mnożymy przez $A^{(1)}_{i2}/A^{(1)}_{22}$ i wynik odejmujemy od równania i.

Eliminacja do przodu (c. d.)

Inaczej mówiąc mnożymy równanie 1 w oryginalnym układzie równań przez A_{i1}/A_{11} i odejmujemy wynik od równania i (powtarzając dla wszystkich $i > 1$).

W drugim kroku równania 1 i 2 pozostawiamy bez zmian, zaś dla $i > 2$ równanie 2 mnożymy przez $A^{(1)}_{i2}/A^{(1)}_{22}$ i wynik odejmujemy od równania i.

W ten sposób otrzymujemy macierz macierz $\widehat{A}^{(2)}$, dla której:

1. Pierwszy wiersz jest identyczny z pierwszym wierszem macierzy $\widehat{A}$.
2. Drugi wiersz jest identyczny z drugim wierszem macierzy $\widehat{A}^{(1)}$.
3. W każdym z pozostałych wierszy pierwszy i drugi element macierzowy są zerami.

Eliminacja do przodu (c. d.)

W rezultacie mamy (dla $i > 2$):

$$A_{ij}^{(2)} = A_{ij}^{(1)} - \frac{A_{i2}^{(1)}}{A_{22}^{(1)}} A_{2j}^{(1)}$$

$$b_i^{(2)} = b_i^{(1)} - \frac{A_{i2}^{(1)}}{A_{22}^{(1)}} b_2^{(1)}$$

Eliminacja do przodu (c. d.)

W rezultacie mamy (dla $i > 2$):

$$A_{ij}^{(2)} = A_{ij}^{(1)} - \frac{A_{i2}^{(1)}}{A_{22}^{(1)}} A_{2j}^{(1)}$$

$$b_{i}^{(2)} = b_{i}^{(1)} - \frac{A_{i2}^{(1)}}{A_{22}^{(1)}} b_{2}^{(1)}$$

Powtarzając analogiczną procedurę dla kolejnych kolumn macierzy po lewej stronie równania w kroku k otrzymujemy macierz $\widehat{A}^{(k)}$, dla której:

1. Wiersze od 1 do k są takie, jak w macierzy $\widehat{A}^{(k-1)}$.
2. W każdym z pozostałych wierszy elementy macierzowe $A_{ij}^{(k)}$ dla $j < k$ są zerami.

Eliminacja do przodu (c. d.)

W trakcie wykonywania obliczeń elementy macierzowe macierzy $\widehat{A}$ mogą być zastępowane przez odpowiednie elementy macierzowe macierzy $\widehat{A}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń (elementy macierzowe $\widehat{A}$, z wyjątkiem wiersza 1, nie są już potrzebne po przeprowadzeniu kroku 1, zaś elementy macierzowe $\widehat{A}^{(1)}$, z wyjątkiem wierszy 1 i 2, nie są potrzebne po przeprowadzeniu kroku 2 itd.).

Eliminacja do przodu (c. d.)

W trakcie wykonywania obliczeń elementy macierzowe macierzy $\widehat{A}$ mogą być zastępowane przez odpowiednie elementy macierzowe macierzy $\widehat{A}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń (elementy macierzowe $\widehat{A}$, z wyjątkiem wiersza 1, nie są już potrzebne po przeprowadzeniu kroku 1, zaś elementy macierzowe $\widehat{A}^{(1)}$, z wyjątkiem wierszy 1 i 2, nie są potrzebne po przeprowadzeniu kroku 2 itd.).

Podobnie współrzędne wektora $\mathbf{b}$ mogą być zastępowane przez odpowiednie współrzędne wektorów $\mathbf{b}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń.

Eliminacja do przodu (c. d.)

W trakcie wykonywania obliczeń elementy macierzowe macierzy $\widehat{A}$ mogą być zastępowane przez odpowiednie elementy macierzowe macierzy $\widehat{A}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń (elementy macierzowe $\widehat{A}$, z wyjątkiem wiersza 1, nie są już potrzebne po przeprowadzeniu kroku 1, zaś elementy macierzowe $\widehat{A}^{(1)}$, z wyjątkiem wierszy 1 i 2, nie są potrzebne po przeprowadzeniu kroku 2 itd.).

Podobnie współrzędne wektora $\mathbf{b}$ mogą być zastępowane przez odpowiednie współrzędne wektorów $\mathbf{b}^{(k)}$ otrzymywanych w kolejnych krokach obliczeń.

Po przeprowadzeniu obliczeń dla wszystkich kolumn macierzy po lewej stronie równania otrzymujemy poszukiwaną trójkątną macierz $\widehat{U}$.

Eliminacja do przodu – kod

Jeżeli miejsce przechowywania elementu macierzowego $A_{ij}^{(k)}$ oznaczymy przez `a[i][j]`, to algorytm obliczeń dla eliminacji do przodu można zapisać w następujący sposób (`n` jest wymiarem macierzy $\widehat{A}$):

```
for (k = 1; k < n; k++)
    for (i = k + 1; i <= n; i++) {
        double alfa = a[i][k] / a[k][k];

        for (j = k; j <= n; j++)
            a[i][j] -= alfa * a[k][j];

        b[i] -= alfa * b[k];
    }
```

Podstawienie wstecz – kod

Podstawienie wstecz można przeprowadzić w taki sposób, że współrzędne rozwiązania zostaną zapisane w miejscach służących do przechowywania współrzędnych wektora **b**:

```
for (i = n; i >= 1; i--) {
    // Dla j > i b[j] jest współrzędną poszukiwanego wektora x[].
    for (int j = i + 1; j <= n; j++)
        b[i] -= a[i][j] * b[j];

    b[i] /= a[i][i];
}
```

Podstawienie wstecz – kod

Podstawienie wstecz można przeprowadzić w taki sposób, że współrzędne rozwiązania zostaną zapisane w miejscach służących do przechowywania współrzędnych wektora **b**:

```
for (i = n; i >= 1; i--) {
    // Dla j > i b[j] jest współrzędną poszukiwanego wektora x[].
    for (int j = i + 1; j <= n; j++)
        b[i] -= a[i][j] * b[j];

    b[i] /= a[i][i];
}
```

Powyższa pętla jest zapisana z założeniem, że `a[i][j]` oznacza miejsce przechowywania elementu macierzowego U_{ij}, a `b[i]` – mejsce przechowywania współrzędnej b'_i wektora $\mathbf{b}'$.

Eliminacja Gaussa – warunki

Aby można było przeprowadzić eliminację Gaussa, zmienna `a[k][k]` musi być różna od zera na początku kroku k.

Eliminacja Gaussa – warunki

Aby można było przeprowadzić eliminację Gaussa, zmienna `a[k][k]` musi być różna od zera na początku kroku k.

W praktyce, aby uniknąć znaczących błędów zaokrąglenia, żąda się, aby jej wartość bezwzględna była większa od pewnej zadanej stałej ϵ.

Eliminacja Gaussa – warunki

Aby można było przeprowadzić eliminację Gaussa, zmienna `a[k][k]` musi być różna od zera na początku kroku k.

W praktyce, aby uniknąć znaczących błędów zaokrąglenia, żąda się, aby jej wartość bezwzględna była większa od pewnej zadanej stałej ϵ.

Najwygodniej jest sprawdzać spełnienie tego warunku już w trakcie trwania obliczeń, przerywając je w przypadku, gdy nie jest on spełniony dla pewnego k.

Eliminacja Gaussa – warunki

Aby można było przeprowadzić eliminację Gaussa, zmienna `a[k][k]` musi być różna od zera na początku kroku k.

W praktyce, aby uniknąć znaczących błędów zaokrąglenia, żąda się, aby jej wartość bezwzględna była większa od pewnej zadanej stałej ϵ.

Najwygodniej jest sprawdzać spełnienie tego warunku już w trakcie trwania obliczeń, przerywając je w przypadku, gdy nie jest on spełniony dla pewnego k.

Układ równań początkowo nie spełniający wymaganych warunków może być przekształcony do postaci, w której będą one spełnione.

Algorytm eliminacji Gaussa-Jordana

Algorytm eliminacji Gaussa-Jordana jest bardzo podobny do algorytmu eliminacji Gaussa, ale wykorzystuje obserwację, że eliminację można przeprowadzić także w „górnych" wierszach macierzy, otrzymując ostatecznie macierz diagonalną, dzięki czemu nie ma potrzeby przeprowadzania podstawienia wstecz.

Algorytm eliminacji Gaussa-Jordana

Algorytm eliminacji Gaussa-Jordana jest bardzo podobny do algorytmu eliminacji Gaussa, ale wykorzystuje obserwację, że eliminację można przeprowadzić także w „górnych" wierszach macierzy, otrzymując ostatecznie macierz diagonalną, dzięki czemu nie ma potrzeby przeprowadzania podstawienia wstecz.

```
for (k = 1; k <= n; k++) {
    for (i = 1; i <= n; i++) {
        if (i == k)
            continue;

        alfa = a[i][k] / a[k][k];
        for (j = k; j <= n; j++)
            a[i][j] -= alfa * a[k][j];

        b[i] -= alfa * b[k];
    }
    x[k] = b[k] / a[k][k];
}
```

Wady algorytmu eliminacji Gaussa

Istnieją układy równań liniowych, dla których istnieje rozwiązanie, chociaż warunki konieczne do przeprowadzenia eliminacji Gaussa nie są przez nie spełniane w wyjściowej postaci.

Wady algorytmu eliminacji Gaussa

Istnieją układy równań liniowych, dla których istnieje rozwiązanie, chociaż warunki konieczne do przeprowadzenia eliminacji Gaussa nie są przez nie spełniane w wyjściowej postaci.

Ponadto jeśli elementy macierzowe w różnych wierszach macierzy równania znacznie różnią się od siebie (np. elementy macierzowe w wierszu k są o wiele rzędów wielkości większe od odpowiadających im elementów macierzowych w wierszu j), w czasie obliczeń mogą powstać znaczące błędy zaokrąglenia.

Wady algorytmu eliminacji Gaussa

Istnieją układy równań liniowych, dla których istnieje rozwiązanie, chociaż warunki konieczne do przeprowadzenia eliminacji Gaussa nie są przez nie spełniane w wyjściowej postaci.

Ponadto jeśli elementy macierzowe w różnych wierszach macierzy równania znacznie różnią się od siebie (np. elementy macierzowe w wierszu k są o wiele rzędów wielkości większe od odpowiadających im elementów macierzowych w wierszu j), w czasie obliczeń mogą powstać znaczące błędy zaokrąglenia.

Problemów tych można uniknąć poprzez odpowiednie przekształcanie układu równań podczas przeprowadzania obliczeń.

Normalizacja elementów macierzowych

Pozwala na redukcję błędów zaokrąglenia.

Normalizacja elementów macierzowych

Pozwala na redukcję błędów zaokrąglenia.

Polega na tym, że w każdym kroku obliczeń:

1. Dla każdego wiersza $i \geqslant k$ macierzy $\widehat{A}^{(k)}$ wyznacza się element macierzowy o największej wartości bezwzględnej.
2. Wszystkie elementy macierzowe w wierszu i macierzy $\widehat{A}^{(k)}$ oraz współrzędna i wektora $\mathbf{b}^{(k)}$ są dzielone przez wartość bezwzględną tego elementu macierzowego.

Normalizacja elementów macierzowych

Pozwala na redukcję błędów zaokrąglenia.

Polega na tym, że w każdym kroku obliczeń:

1. Dla każdego wiersza $i \geqslant k$ macierzy $\widehat{A}^{(k)}$ wyznacza się element macierzowy o największej wartości bezwzględnej.
2. Wszystkie elementy macierzowe w wierszu i macierzy $\widehat{A}^{(k)}$ oraz współrzędna i wektora $\mathbf{b}^{(k)}$ są dzielone przez wartość bezwzględną tego elementu macierzowego.

Po przeprowadzeniu tej operacji wszystkie elementy macierzowe $\widehat{A}^{(k)}$ w wierszach $i \geqslant k$ oraz wszystkie współrzędne $\mathbf{b}^{(k)}$ o indeksach $i \geqslant k$ mają wartości bezwzględne nie przekraczające 1.

Wybieranie dzielnika o największym module

Wykorzystuje obserwację, iż zamiana miejscami dwóch równań w układzie równań liniowych nie ma wpływu na jego rozwiązanie.

Wybieranie dzielnika o największym module

Wykorzystuje obserwację, iż zamiana miejscami dwóch równań w układzie równań liniowych nie ma wpływu na jego rozwiązanie.

Polega na tym, że w każdym kroku obliczeń:

1. Spośród wierszy $i \geqslant k$ macierzy $\widehat{A}^{(k)}$ wyznacza się wiersz i_{max}, w którym element w kolumnie k ma największą wartość bezwzględną.
2. Zamienia się miejscami wiersze k oraz i_{max} macierzy $\widehat{A}^{(k)}$ i współrzędne k oraz i_{max} wektora $\mathbf{b}^{(k)}$.

Wybieranie dzielnika o największym module

Wykorzystuje obserwację, iż zamiana miejscami dwóch równań w układzie równań liniowych nie ma wpływu na jego rozwiązanie.

Polega na tym, że w każdym kroku obliczeń:

1. Spośród wierszy $i \geqslant k$ macierzy $\widehat{A}^{(k)}$ wyznacza się wiersz i_{max}, w którym element w kolumnie k ma największą wartość bezwzględną.
2. Zamienia się miejscami wiersze k oraz i_{max} macierzy $\widehat{A}^{(k)}$ i współrzędne k oraz i_{max} wektora $\mathbf{b}^{(k)}$.

Po przeprowadzeniu tej operacji element macierzowy $A_{kk}^{(k)}$ ma największą wartość bezwzględną ze wszystkich elementów macierzowych $\widehat{A}^{(k)}$ w kolumnie k. Jeśli jest on zerem, to wszystkie elementy macierzowe $\widehat{A}^{(k)}$ w kolumnie k są zerami.

Poprawiony algorytm eliminacji Gaussa

```
for (k = 1; k < n; k++) {
    int i, j;
    double r, w;

    // Normalizacja elementów macierzowych
    for (i = k; i <= n; i++) {
        r = fabs(a[i][k]);
        for (j = k + 1; j <= n; j++) {
            w = fabs(a[i][j]);
            if (w > r)
                r = w;
        }
        if (r == 0)
            throw "Dzielenie przez zero";

        b[i] /= r;
        for (j = k; j <= n; j++)
            a[i][j] /= r;
    }

    // Wybór dzielnika o największym module
    r = fabs(a[k][k]);
    i = k;
    for (j = k + 1; j <= n; j++) {
        w = fabs(a[j][k]);
        if (w > r) {
            r = w;
            i = j;
        }
    }

    if (i > k) {
        zamiana_wierszy(a[k], a[i]);
        zamiana(b[k], b[i]);
    }

    if (a[k][k] == 0)
        throw "Dzielenie przez zero";

    // Eliminacja
    for (i = k + 1; i <= n; i++) {
        r = a[i][k] / a[k][k];
        b[i] -= r * b[k];
        for (j = k; j <= n; j++)
            a[i][j] -= r * a[k][j];
    }
}
```

Układy równań nieliniowych

Układy równań nieliniowych w ogólności mają postać

$$\mathbf{F}(\mathbf{x}) = 0$$

gdzie $\mathbf{F}(\mathbf{x})$ jest dowolną funkcją $\mathbb{R}^n \longrightarrow \mathbb{R}^m$.

Układy równań nieliniowych

Układy równań nieliniowych w ogólności mają postać

$$\mathbf{F}(\mathbf{x}) = 0$$

gdzie $\mathbf{F}(\mathbf{x})$ jest dowolną funkcją $\mathbb{R}^n \longrightarrow \mathbb{R}^m$.

Aby obliczyć (przybliżone) rozwiązanie takiego układu równań, można posłużyć się rozwinięciem Taylora funkcji $\mathbf{F}(\mathbf{x})$ w otoczeniu pewnego punktu $\mathbf{x}_0$, który będzie stanowił początkowe przybliżenie rozwiązania:

$$\mathbf{F}(\mathbf{x}) \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

gdzie $\nabla\mathbf{F}(\mathbf{x}_0)$ oznacza macierz Jacobiego funkcji w punkcie $\mathbf{x}_0$.

Algorytm Newtona

Jeśli $\mathbf{x}_\infty$ jest ścisłym rozwiązaniem naszego równania, to mamy:

$$0 \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x}_\infty - \mathbf{x}_0) \tag{1}$$

Algorytm Newtona

Jeśli $\mathbf{x}_\infty$ jest ścisłym rozwiązaniem naszego równania, to mamy:

$$0 \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x}_\infty - \mathbf{x}_0) \tag{1}$$

To oznacza, że

$$\mathbf{x}_\infty \approx \mathbf{x}_0 - [\nabla\mathbf{F}(\mathbf{x}_0)]^{-1}\mathbf{F}(\mathbf{x}_0) \tag{2}$$

Algorytm Newtona

Jeśli $\mathbf{x}_\infty$ jest ścisłym rozwiązaniem naszego równania, to mamy:

$$0 \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x}_\infty - \mathbf{x}_0) \tag{1}$$

To oznacza, że

$$\mathbf{x}_\infty \approx \mathbf{x}_0 - [\nabla\mathbf{F}(\mathbf{x}_0)]^{-1}\mathbf{F}(\mathbf{x}_0) \tag{2}$$

Możemy zatem skonstruować ciąg punktów $\mathbf{x}_k$ taki, że

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\nabla\mathbf{F}(\mathbf{x}_k)]^{-1}\mathbf{F}(\mathbf{x}_k) \tag{3}$$

który będzie zbieżny do $\mathbf{x}_\infty$, o ile punkt $\mathbf{x}_0$ zostanie wybrany we właściwy sposób (musi on być „dostatecznie dobrym” przybliżeniem rozwiązania).

Algorytm Newtona (c. d.)

Zamiast obliczania macierzy $[\nabla \mathbf{F}(\mathbf{x}_k)]^{-1}$ w każdym punkcie można wykorzystać metody omówione wyżej i rozwiązywać równania liniowe

$$\nabla \mathbf{F}(\mathbf{x}_k)\mathbf{d}_k = -\mathbf{F}(\mathbf{x}_k) \tag{4}$$

a następnie wyznaczać $\mathbf{x}_{k+1}$ z wzoru

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k \tag{5}$$

Algorytm Newtona (c. d.)

Zamiast obliczania macierzy $[\nabla \mathbf{F}(\mathbf{x}_k)]^{-1}$ w każdym punkcie można wykorzystać metody omówione wyżej i rozwiązywać równania liniowe

$$\nabla \mathbf{F}(\mathbf{x}_k)\mathbf{d}_k = -\mathbf{F}(\mathbf{x}_k) \tag{4}$$

a następnie wyznaczać $\mathbf{x}_{k+1}$ z wzoru

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k \tag{5}$$

Zatem najpierw rozwiązujemy równanie (4) dla $\mathbf{x}_0$, podstawiamy wynik do równania (5), aby obliczyć $\mathbf{x}_1$ itd.

Algorytm Newtona (c. d.)

Zamiast obliczania macierzy $[\nabla \mathbf{F}(\mathbf{x}_k)]^{-1}$ w każdym punkcie można wykorzystać metody omówione wyżej i rozwiązywać równania liniowe

$$\nabla \mathbf{F}(\mathbf{x}_k)\mathbf{d}_k = -\mathbf{F}(\mathbf{x}_k) \tag{4}$$

a następnie wyznaczać $\mathbf{x}_{k+1}$ z wzoru

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k \tag{5}$$

Zatem najpierw rozwiązujemy równanie (4) dla $\mathbf{x}_0$, podstawiamy wynik do równania (5), aby obliczyć $\mathbf{x}_1$ itd.

Obliczenia można zakończyć np. wtedy, gdy $||\mathbf{x}_j - \mathbf{x}_{j-1}|| < \epsilon$ dla pewnego $k = j$ (i dla zadanej stałej ϵ).

Tablice dwuwymiarowe

Najprościej jest reprezentować macierz z wykorzystaniem tablicy dwuwymiarowej o elementach typu `double`.

Tablice dwuwymiarowe

Najprościej jest reprezentować macierz z wykorzystaniem tablicy dwuwymiarowej o elementach typu `double`.

Taką tablicę definiuje się określając liczbę jej wierszy i kolumn (`N` i `M`; mogą to być zmienne, jeżeli sama tablica jest zmienną lokalną):

```
double tablica[N][M];
```

Tablice dwuwymiarowe

Najprościej jest reprezentować macierz z wykorzystaniem tablicy dwuwymiarowej o elementach typu `double`.

Taką tablicę definiuje się określając liczbę jej wierszy i kolumn (`N` i `M`; mogą to być zmienne, jeżeli sama tablica jest zmienną lokalną):

```
double tablica[N][M];
```

Wtedy wiersze tablicy mają indeksy od 0 do $N-1$, a kolumny mają indeksy od 0 do $M-1$.

Tablice dwuwymiarowe

Najprościej jest reprezentować macierz z wykorzystaniem tablicy dwuwymiarowej o elementach typu `double`.

Taką tablicę definiuje się określając liczbę jej wierszy i kolumn (`N` i `M`; mogą to być zmienne, jeżeli sama tablica jest zmienną lokalną):

```
double tablica[N][M];
```

Wtedy wiersze tablicy mają indeksy od 0 do $N-1$, a kolumny mają indeksy od 0 do $M-1$.

W C++ nie ma możliwości zmiany sposobu indeksowania kolumn i wierszy tablicy dwuwymiarowej. Wynika to ze sposobu rozmieszczenia elementów tablicy w pamięci.

Struktura tablic dwuwymiarowych

W C++ (podobnie, jak w C) każdy wiersz tablicy dwuwymiarowej jest traktowany jako oddzielna tablica jednowymiarowa.

Struktura tablic dwuwymiarowych

W C++ (podobnie, jak w C) każdy wiersz tablicy dwuwymiarowej jest traktowany jako oddzielna tablica jednowymiarowa.

Zatem, jeżeli `tablica[][]` jest tablicą dwuwymiarową zdefiniowaną jak wyżej, to symbol `tablica[i]` oznacza wskaźnik zawierający adres elementu `tablica[i][0]`.

Struktura tablic dwuwymiarowych

W C++ (podobnie, jak w C) każdy wiersz tablicy dwuwymiarowej jest traktowany jako oddzielna tablica jednowymiarowa.

Zatem, jeżeli `tablica[][]` jest tablicą dwuwymiarową zdefiniowaną jak wyżej, to symbol `tablica[i]` oznacza wskaźnik zawierający adres elementu `tablica[i][0]`.

W związku z tym symbol `*(tablica[i])` oznacza element `tablica[i][0]`.

Struktura tablic dwuwymiarowych

W C++ (podobnie, jak w C) każdy wiersz tablicy dwuwymiarowej jest traktowany jako oddzielna tablica jednowymiarowa.

Zatem, jeżeli `tablica[][]` jest tablicą dwuwymiarową zdefiniowaną jak wyżej, to symbol `tablica[i]` oznacza wskaźnik zawierający adres elementu `tablica[i][0]`.

W związku z tym symbol `*(tablica[i])` oznacza element `tablica[i][0]`.

Po wykonaniu przypisania `ptr = tablica[i]`, gdzie `ptr` jest wskaźnikiem typu `double`, można posługiwać się wskaźnikiem `ptr` tak, jakby był on nazwą tablicy jednowymiarowej pokrywającej się z wierszem *i* tablicy dwuwymiarowej.

Algorytm Gaussa w wersji wskaźnikowej

```
for (k = 0; k < n - 1; k++) {
    double *a_k = a[k];

    for (i = k + 1; i < n; i++) {
        double *a_i = a[i];
        double alfa = a_i[k] / a_k[k];

        for (j = k; j < n; j++)
            a_i[j] -= alfa * a_k[j];

        b[i] -= alfa * b[k];
    }
}
```

```
for (k = 0; k < n - 1; k++) {
    double *a_k = a[k];

    for (i = k + 1; i < n; i++) {
        double alfa = a[i][k] / a_k[k];
        double *a_i_j = a[i] + j;
        double *a_k_j = a_k + j;

        for (j = k; j < n; j++)
            *a_i_j++ -= *a_k_j++ * alpha;

        b[i] -= b[k] * alfa;
    }
}
```

Algorytm Gaussa w wersji wskaźnikowej

```
for (k = 0; k < n - 1; k++) {
    double *a_k = a[k];

    for (i = k + 1; i < n; i++) {
        double *a_i = a[i];
        double alfa = a_i[k] / a_k[k];

        for (j = k; j < n; j++)
            a_i[j] -= alfa * a_k[j];

        b[i] -= alfa * b[k];
    }
}
```

```
for (k = 0; k < n - 1; k++) {
    double *a_k = a[k];

    for (i = k + 1; i < n; i++) {
        double alfa = a[i][k] / a_k[k];
        double *a_i_j = a[i] + j;
        double *a_k_j = a_k + j;

        for (j = k; j < n; j++)
            *a_i_j++ -= *a_k_j++ * alpha;

        b[i] -= b[k] * alfa;
    }
}
```

Niestety w tablicach dwuwymiarowych nie można zamieniać miejscami wskaźników zawierających adresy poszczególnych wierszy. To powoduje, że w poprawionym algorytmie eliminacji Gaussa zamianę wierszy macierzy trzeba przeprowadzać jako kopiowanie elementów macierzowych.

Problemy z rozmiarami tablic dwuwymiarowych

Z tablicamy dwuwymiarowymi w C++ wiąże się dodatkowy problem, polegający na tym, że mogą one być deklarowane jako zmienne statyczne lub jako zmienne lokalne (w funkcjach).

Problemy z rozmiarami tablic dwuwymiarowych

Z tablicamy dwuwymiarowymi w C++ wiąże się dodatkowy problem, polegający na tym, że mogą one być deklarowane jako zmienne statyczne lub jako zmienne lokalne (w funkcjach).

Jeżeli mają one być zmiennymi statycznymi, to ich rozmiary muszą być znane z wyprzedzeniem (tzn. przed skompilowaniem programu) lub trzeba nadawać im rozmiary „na zapas”.

Problemy z rozmiarami tablic dwuwymiarowych

Z tablicamy dwuwymiarowymi w C++ wiąże się dodatkowy problem, polegający na tym, że mogą one być deklarowane jako zmienne statyczne lub jako zmienne lokalne (w funkcjach).

Jeżeli mają one być zmiennymi statycznymi, to ich rozmiary muszą być znane z wyprzedzeniem (tzn. przed skompilowaniem programu) lub trzeba nadawać im rozmiary „na zapas".

Jeżeli mają one być zmiennymi lokalnymi, to pamięć na przechowywanie ich elementów macierzowych jest rezerwowana na stosie procesora. Powoduje to, że rozmiary takich tablic podlegają ograniczeniom.

Problemy z rozmiarami tablic dwuwymiarowych

Z tablicamy dwuwymiarowymi w C++ wiąże się dodatkowy problem, polegający na tym, że mogą one być deklarowane jako zmienne statyczne lub jako zmienne lokalne (w funkcjach).

Jeżeli mają one być zmiennymi statycznymi, to ich rozmiary muszą być znane z wyprzedzeniem (tzn. przed skompilowaniem programu) lub trzeba nadawać im rozmiary „na zapas”.

Jeżeli mają one być zmiennymi lokalnymi, to pamięć na przechowywanie ich elementów macierzowych jest rezerwowana na stosie procesora. Powoduje to, że rozmiary takich tablic podlegają ograniczeniom.

Zatem pamięć do przechowywania elementów macierzowych macierzy najlepiej jest rezerwować na żądanie.

Macierze i rezerwowanie pamięci na żądanie

Kompilator C++ zawsze rozmieszcza wiersze tablic dwuwymiarowych w pamięci kolejno jeden za drugim, zgodnie z numeracją.

Macierze i rezerwowanie pamięci na żądanie

Kompilator C++ zawsze rozmieszcza wiersze tablic dwuwymiarowych w pamięci kolejno jeden za drugim, zgodnie z numeracją.

Można wykorzystać tę obserwację rezerwując na żądanie zmienne, które wspólnie będą reprezentować macierz:

```
double **wiersze, *elementy;

elementy = new double[N*M];  // Elementy macierzowe.
wiersze = new double *[N];   // Wskaźniki do wierszy.
for (int j = 0; j < N; j++)
    wiersze[j] = elementy + j*M;
```

Macierze i rezerwowanie pamięci na żądanie

Kompilator C++ zawsze rozmieszcza wiersze tablic dwuwymiarowych w pamięci kolejno jeden za drugim, zgodnie z numeracją.

Można wykorzystać tę obserwację rezerwując na żądanie zmienne, które wspólnie będą reprezentować macierz:

```
double **wiersze, *elementy;

elementy = new double[N*M];  // Elementy macierzowe.
wiersze = new double *[N];   // Wskaźniki do wierszy.
for (int j = 0; j < N; j++)
    wiersze[j] = elementy + j*M;
```

Wtedy symbol `wiersze[j][k]` oznacza element o indeksie k z wiersza o indeksie j macierzy, gdzie $j = 0 \ldots N-1$ oraz $k = 0 \ldots M-1$.

Macierze i rezerwowanie pamięci na żądanie (c. d.)

Rezerwując pamięć na żądanie w sposób podobny do przedstawionego można tak zmodyfikować wskaźniki, aby wiersze i kolumny macierzy były indeksowane począwszy od 1.

Macierze i rezerwowanie pamięci na żądanie (c. d.)

Rezerwując pamięć na żądanie w sposób podobny do przedstawionego można tak zmodyfikować wskaźniki, aby wiersze i kolumny macierzy były indeksowane począwszy od 1.

```
double **wiersze, *elementy;

elementy = new double[N*M];
wiersze = new double *[N];
wiersze--;
for (int j = 1; j <= N; j++)
    wiersze[j] = elementy + (j-1)*M - 1;
```

Macierze i rezerwowanie pamięci na żądanie (c. d.)

Rezerwując pamięć na żądanie w sposób podobny do przedstawionego można tak zmodyfikować wskaźniki, aby wiersze i kolumny macierzy były indeksowane począwszy od 1.

```
double **wiersze, *elementy;

elementy = new double[N*M];
wiersze = new double *[N];
wiersze--;
for (int j = 1; j <= N; j++)
    wiersze[j] = elementy + (j-1)*M - 1;
```

Wtedy symbol `wiersze[j][k]` oznacza element o indeksie k z wiersza o indeksie j macierzy, gdzie $j = 1 \ldots N$ oraz $k = 1 \ldots M$.

Klasa reprezentująca macierz

Można także zdefiniować klasę reprezentującą macierz:

```
class Macierz {
    double *elementy;
    int n, m;
  public:
    Macierz(int a, int b);
    ~Macierz(void);
    ...
    double * operator [](int i) const;
};

Macierz::Macierz(int a, int b)
{
    elementy = new double[a*b];
    n = a;
    m = b;
}
```

```
Macierz::~Macierz(void)
{
    delete [] elementy;
}

double * Macierz::operator [](int i) const
{
    if (i < 1 || i > n)
        throw "Przekroczony zakres";

    return elementy + (i-1)*m - 1;
}
```

Klasa reprezentująca macierz

Można także zdefiniować klasę reprezentującą macierz:

```
class Macierz {
    double *elementy;
    int n, m;
  public:
    Macierz(int a, int b);
    ~Macierz(void);
    ...
    double * operator [](int i) const;
};

Macierz::Macierz(int a, int b)
{
    elementy = new double[a*b];
    n = a;
    m = b;
}
```

```
Macierz::~Macierz(void)
{
    delete [] elementy;
}

double * Macierz::operator [](int i) const
{
    if (i < 1 || i > n)
        throw "Przekroczony zakres";

    return elementy + (i-1)*m - 1;
}
```

Wtedy, dla obiektu `M` klasy `Macierz`, symbol `M[j][k]` oznacza element o indeksie k z wiersza o indeksie j macierzy, gdzie $j = 1 \ldots N$ oraz $k = 1 \ldots M$.

Macierze i klasy (c. d.)

Aby uniknąć niepotrzebnych obliczeń w czasie wykonywania programu, można użyć pomocniczej tablicy `wiersze[]`, w której będą zapisywane adresy poszczególnych wierszy macierzy:

```
class Macierz {
    double **wiersze;
  public:
    Macierz(unsigned int a, unsigned int b);
    ~Macierz(void);
    ...
    double * operator [](int i) const;
};

Macierz::~Macierz(void)
{
    delete [] (wiersze[1] + 1);
    wiersze++;
    delete [] wiersze;
}
```

```
Macierz::Macierz(unsigned int a, unsigned int b)
{
    double *wsk;

    wsk = new double[a*b];
    wiersze = new double *[a];
    wiersze--;
    for (int j = 1; j <= a; j++)
        wiersze[j] = wsk + (j-1)*b - 1;
}

double * Macierz::operator [](int i) const
{
    return wiersze[i];
}
```

Macierze i klasy (c. d.)

Aby uniknąć niepotrzebnych obliczeń w czasie wykonywania programu, można użyć pomocniczej tablicy `wiersze[]`, w której będą zapisywane adresy poszczególnych wierszy macierzy:

```
class Macierz {
    double **wiersze;
  public:
    Macierz(unsigned int a, unsigned int b);
    ~Macierz(void);
    ...
    double * operator [](int i) const;
};

Macierz::~Macierz(void)
{
    delete [] (wiersze[1] + 1);
    wiersze++;
    delete [] wiersze;
}
```

```
Macierz::Macierz(unsigned int a, unsigned int b)
{
    double *wsk;

    wsk = new double[a*b];
    wiersze = new double *[a];
    wiersze--;
    for (int j = 1; j <= a; j++)
        wiersze[j] = wsk + (j-1)*b - 1;
}

double * Macierz::operator [](int i) const
{
    return wiersze[i];
}
```

Można także dodać kod sprawdzający przekroczenie zakresu indeksów.

Wpływ sprzętu na szybkość obliczeń

Rzecz jasna szybkość działania programu komputerowego zależy od sprzętu, który go wykonuje.

Wpływ sprzętu na szybkość obliczeń

Rzecz jasna szybkość działania programu komputerowego zależy od sprzętu, który go wykonuje.

Na przykład czas wykonywania programu (po skompilowaniu) jest w ogólności krótszy dla procesorów o wyższej częstotliwości zegara.

Wpływ sprzętu na szybkość obliczeń

Rzecz jasna szybkość działania programu komputerowego zależy od sprzętu, który go wykonuje.

Na przykład czas wykonywania programu (po skompilowaniu) jest w ogólności krótszy dla procesorów o wyższej częstotliwości zegara.

Jednakże częstotliwość zegara procesora jest jednym z wielu parametrów wpływających na wydajność komputera (a przez to na szybkość obliczeń).

Wpływ sprzętu na szybkość obliczeń

Rzecz jasna szybkość działania programu komputerowego zależy od sprzętu, który go wykonuje.

Na przykład czas wykonywania programu (po skompilowaniu) jest w ogólności krótszy dla procesorów o wyższej częstotliwości zegara.

Jednakże częstotliwość zegara procesora jest jednym z wielu parametrów wpływających na wydajność komputera (a przez to na szybkość obliczeń).

Co więcej, szczególne własności sprzętu mogą powodować, że algorytmy o (teoretycznie) jednakowej złożoności obliczeniowej będą wykonywane z różną szybkością.

Przykład – mnożenie macierzy

Rozważmy dwie macierze kwadratowe $\widehat{A}$ i $\widehat{B}$ o wymiarze n oraz macierze $\widehat{C} = A \cdot B$ i $\widehat{D} = A \cdot B^T$. Mamy

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \,, \quad D_{ij} = \sum_{k=1}^{n} A_{ik} B_{jk}$$

Przykład – mnożenie macierzy

Rozważmy dwie macierze kwadratowe $\widehat{A}$ i $\widehat{B}$ o wymiarze n oraz macierze $\widehat{C} = A \cdot B$ i $\widehat{D} = A \cdot B^T$. Mamy

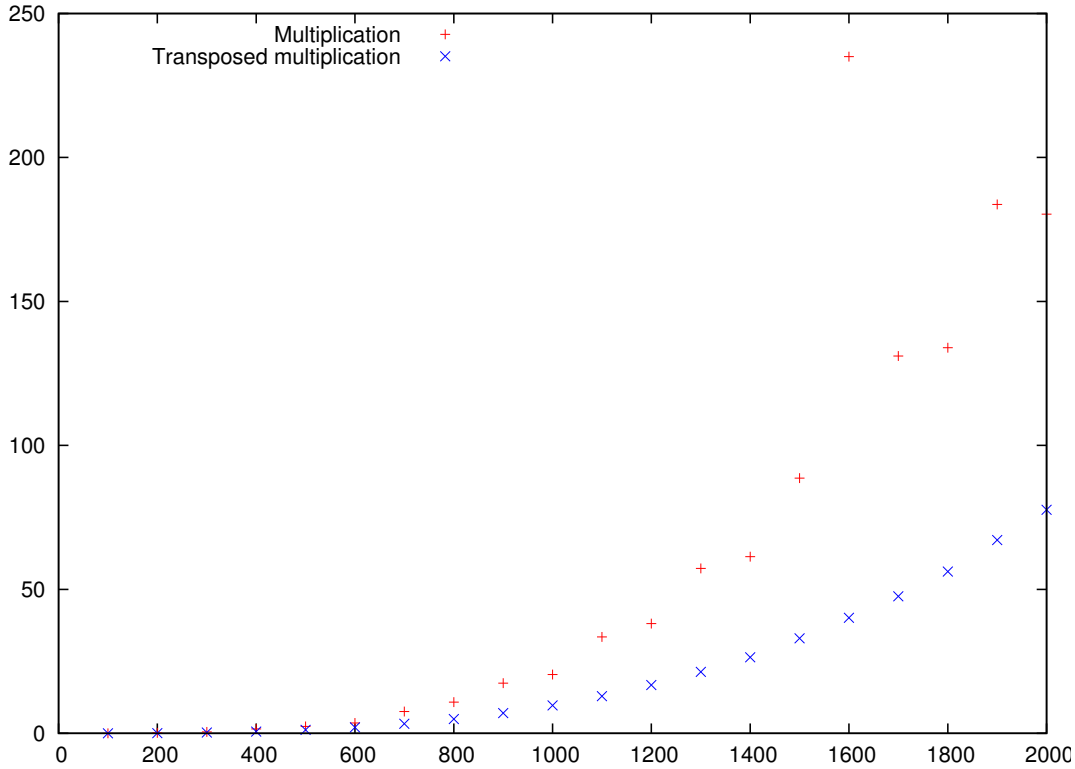
$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \,, \quad D_{ij} = \sum_{k=1}^{n} A_{ik} B_{jk}$$

Okazuje się, że obliczanie elementów macierzowych $\widehat{C}$ w pętli po lewej stronie zajmuje zwykle znacznie więcej czasu, niż obliczanie elementów macierzowych $\widehat{D}$ w pętli po prawej stronie:

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++) {
        c[i][j] = 0;
        for (k = 1; k <= n; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
```

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++) {
        d[i][j] = 0;
        for (k = 1; k <= n; k++)
            d[i][j] += a[i][k] * b[j][k];
    }
```

Wydajność kodu przy mnożeniu macierzy



Rysunek: Czas obliczeń w sekundach (oś pionowa) dla mnożenia macierzy (kolor czerwony) i mnożenia macierzy z transpozycją (kolor niebieski) w zależności od wymiaru macierzy (oś pozioma).

Przykład – mnożenie macierzy (wnioski)

Obliczenia zajmują znacznie mniej czasu w przypadku, gdy w najbardziej wewnętrznej pętli indeks k przebiega elementy macierzowe w jednym wierszu $\widehat{B}$ (a nie w jednej kolumnie).

Przykład – mnożenie macierzy (wnioski)

Obliczenia zajmują znacznie mniej czasu w przypadku, gdy w najbardziej wewnętrznej pętli indeks k przebiega elementy macierzowe w jednym wierszu $\widehat{B}$ (a nie w jednej kolumnie).

Ponadto czas trwania obliczeń dla elementów macierzowych $\widehat{D}$ jest znacznie bardziej przewidywalny, niż czas trwania obliczeń dla elementów macierzowych $\widehat{C}$.

Przykład – mnożenie macierzy (wnioski)

Obliczenia zajmują znacznie mniej czasu w przypadku, gdy w najbardziej wewnętrznej pętli indeks k przebiega elementy macierzowe w jednym wierszu $\widehat{B}$ (a nie w jednej kolumnie).

Ponadto czas trwania obliczeń dla elementów macierzowych $\widehat{D}$ jest znacznie bardziej przewidywalny, niż czas trwania obliczeń dla elementów macierzowych $\widehat{C}$.

Zatem w celu obliczenia elementów macierzowych $\widehat{C}$ korzystne może być transponowanie macierzy $\widehat{B}$ przed przeprowadzeniem obliczeń i zastosowanie mnożenia z transpozycją zamiast „naiwnego” algorytmu.

Przykład – mnożenie macierzy (wnioski)

Obliczenia zajmują znacznie mniej czasu w przypadku, gdy w najbardziej wewnętrznej pętli indeks k przebiega elementy macierzowe w jednym wierszu $\widehat{B}$ (a nie w jednej kolumnie).

Ponadto czas trwania obliczeń dla elementów macierzowych $\widehat{D}$ jest znacznie bardziej przewidywalny, niż czas trwania obliczeń dla elementów macierzowych $\widehat{C}$.

Zatem w celu obliczenia elementów macierzowych $\widehat{C}$ korzystne może być transponowanie macierzy $\widehat{B}$ przed przeprowadzeniem obliczeń i zastosowanie mnożenia z transpozycją zamiast „naiwnego” algorytmu.

Aby wyjaśnić te obserwacje, trzeba wziąć pod uwagę konstrukcję współczesnych komputerów oraz ich sposób działania.

Pamięć RAM

RAM (*ang. Random Access Memory*)

Pamięć o dostępie swobodnym pozwalająca na modyfikowanie zawartości dowolnej komórki pamięci (o pojemności 1 bitu) w dowolnym momencie.

Pamięć RAM

RAM (*ang. Random Access Memory*)

Pamięć o dostępie swobodnym pozwalająca na modyfikowanie zawartości dowolnej komórki pamięci (o pojemności 1 bitu) w dowolnym momencie.

Jest to pamięć ulotna (*ang. volatile*)

Wymaga ciągłego zasilania do zachowania zawartości.

Pamięć RAM

RAM (*ang. Random Access Memory*)

Pamięć o dostępie swobodnym pozwalająca na modyfikowanie zawartości dowolnej komórki pamięci (o pojemności 1 bitu) w dowolnym momencie.

Jest to pamięć ulotna (*ang. volatile*)

Wymaga ciągłego zasilania do zachowania zawartości.

Statyczna pamięć RAM (*ang. static RAM*)

Jest zbudowana z tak zwanych przerzutników (*ang. flip-flop*), które z kolei składają się z tranzystorów i nie wymaga odświeżania (*ang. refresh*).

Pamięć RAM

RAM (*ang. Random Access Memory*)

Pamięć o dostępie swobodnym pozwalająca na modyfikowanie zawartości dowolnej komórki pamięci (o pojemności 1 bitu) w dowolnym momencie.

Jest to pamięć ulotna (*ang. volatile*)

Wymaga ciągłego zasilania do zachowania zawartości.

Statyczna pamięć RAM (*ang. static RAM*)

Jest zbudowana z tak zwanych przerzutników (*ang. flip-flop*), które z kolei składają się z tranzystorów i nie wymaga odświeżania (*ang. refresh*).

Statyczna pamięć RAM charakteryzuje się krótkim czasem dostępu (*ang. access time*), czyli czasem potrzebnym do przeprowadzenia operacji zapisu bądź odczytu danych.

Dynamiczna pamięć RAM

DRAM (*ang. Dynamic RAM*)

Jest zbudowana z tranzystorów i kondensatorów (1 komórka składa się z 1 kondensatora i 1 tranzystora). Kondensatory służą do przechowywania bitów danych, w związku z czym taka pamięć wymaga odświeżania (kondensatory z czasem tracą ładunek).

Dynamiczna pamięć RAM

DRAM (*ang. Dynamic RAM*)

Jest zbudowana z tranzystorów i kondensatorów (1 komórka składa się z 1 kondensatora i 1 tranzystora). Kondensatory służą do przechowywania bitów danych, w związku z czym taka pamięć wymaga odświeżania (kondensatory z czasem tracą ładunek).

Odświeżanie polega na okresowym odczytywaniu danych zapisanych w pamięci.

Dynamiczna pamięć RAM

DRAM (*ang. Dynamic RAM*)

Jest zbudowana z tranzystorów i kondensatorów (1 komórka składa się z 1 kondensatora i 1 tranzystora). Kondensatory służą do przechowywania bitów danych, w związku z czym taka pamięć wymaga odświeżania (kondensatory z czasem tracą ładunek).

Odświeżanie polega na okresowym odczytywaniu danych zapisanych w pamięci.

Jak to działa?

W celu odczytania zawartości komórki pamięci trzeba zmierzyć ładunek na kondensatorze, więc trzeba pozwolić mu rozładować się poprzez układ pomiarowy (wzmacniający sygnał). Później (jeśli trzeba) kondensator jest ponownie ładowany na zasadzie sprzężenia zwrotnego (*ang. feedback*).

Zalety dynamicznych pamięci RAM

Zalety DRAM w stosunku do pamięci statycznych

1. Znacznie mniejsze rozmiary komórek.
2. Znacznie mniejsza liczba tranzystorów na komórkę (typowo 1:6).
3. Bardziej regularna struktura komórki (możliwe ciaśniejsze upakowanie).
4. Brak konieczności doprowadzania napięcia zasilającego do każdej komórki z osobna.

Zalety dynamicznych pamięci RAM

Zalety DRAM w stosunku do pamięci statycznych

1. Znacznie mniejsze rozmiary komórek.
2. Znacznie mniejsza liczba tranzystorów na komórkę (typowo 1:6).
3. Bardziej regularna struktura komórki (możliwe ciaśniejsze upakowanie).
4. Brak konieczności doprowadzania napięcia zasilającego do każdej komórki z osobna.

Wymienione cechy pamięci DRAM powodują, że ten rodzaj pamięci jest dramatycznie tańszy w produkcji, zwłaszcza dla dużych pojemności (im większa pojemność, tym pamięci DRAM są tańsze).

Zalety dynamicznych pamięci RAM

Zalety DRAM w stosunku do pamięci statycznych

1. Znacznie mniejsze rozmiary komórek.
2. Znacznie mniejsza liczba tranzystorów na komórkę (typowo 1:6).
3. Bardziej regularna struktura komórki (możliwe ciaśniejsze upakowanie).
4. Brak konieczności doprowadzania napięcia zasilającego do każdej komórki z osobna.

Wymienione cechy pamięci DRAM powodują, że ten rodzaj pamięci jest dramatycznie tańszy w produkcji, zwłaszcza dla dużych pojemności (im większa pojemność, tym pamięci DRAM są tańsze).

Dlatego w komputerach używa się głównie pamięci DRAM.

Wady dynamicznych pamięci RAM

Niestety technologia DRAM ma poważne wady

1. Znaczna upływność kondensatorów dla dużych pojemności pamięci (trzeba często odświeżać, typowo co 64 *ms*).
2. Konieczność wzmacniania sygnału przy odczycie (prąd pochodzący z rozładowania kondensatora w komórce pamięci jest bardzo słaby).
3. Konieczność ponownego ładowania kondensatora po odczycie (zwiększa zużycie energii i zabiera czas).
4. Ładowanie i rozładowywanie kondensatora nie jest natychmiastowe (przy odczycie dostatecznie wyraźny sygnał pojawia się z opóźnieniem).

Wady dynamicznych pamięci RAM

Niestety technologia DRAM ma poważne wady

1. Znaczna upływność kondensatorów dla dużych pojemności pamięci (trzeba często odświeżać, typowo co 64 *ms*).
2. Konieczność wzmacniania sygnału przy odczycie (prąd pochodzący z rozładowania kondensatora w komórce pamięci jest bardzo słaby).
3. Konieczność ponownego ładowania kondensatora po odczycie (zwiększa zużycie energii i zabiera czas).
4. Ładowanie i rozładowywanie kondensatora nie jest natychmiastowe (przy odczycie dostatecznie wyraźny sygnał pojawia się z opóźnieniem).

Powodują one, że operowanie pamięciami DRAM jest skomplikowane oraz istnieje fizyczne ograniczenie czasu dostępu dla nich, którego nie da się wyeliminować.

Organizacja pamięci DRAM

Komórki pamięci DRAM są rozmieszczane w prostokątnych matrycach (*ang. array*). Inny sposób uporządkowania ich byłby zbyt kosztowny.

Organizacja pamięci DRAM

Komórki pamięci DRAM są rozmieszczane w prostokątnych matrycach (*ang. array*). Inny sposób uporządkowania ich byłby zbyt kosztowny.

Przy odwołaniach do pamięci adres (fizyczny) lokacji jest dzielony na dwie części, z których jedna, $\overline{RAS}$ (*ang. Row Address Selection*), wybiera wiersz (*ang. row*) matrycy, a druga, $\overline{CAS}$ (*ang. Column Access Selection*), określa kolumnę (*ang. column*), z której mają być pobrane dane.

Organizacja pamięci DRAM

Komórki pamięci DRAM są rozmieszczane w prostokątnych matrycach (*ang. array*). Inny sposób uporządkowania ich byłby zbyt kosztowny.

Przy odwołaniach do pamięci adres (fizyczny) lokacji jest dzielony na dwie części, z których jedna, $\overline{RAS}$ (*ang. Row Address Selection*), wybiera wiersz (*ang. row*) matrycy, a druga, $\overline{CAS}$ (*ang. Column Access Selection*), określa kolumnę (*ang. column*), z której mają być pobrane dane.

$\overline{RAS}$ jest podawany na demultiplekser, który ma tyle wyjść, ile jest wierszy w matrycy. Generuje on sygnał „aktywujący" wszystkie komórki pamięci w danym wierszu.

Odczyt z pamięci DRAM

Wszystkie naładowane kondensatory komórek pamięci w wierszu matrycy „aktywowanym” przez $\overline{RAS}$ rozładowują się i pochodzące z nich sygnały (po wzmocnieniu) są podawane na multiplekser razem z $\overline{CAS}$.

Odczyt z pamięci DRAM

Wszystkie naładowane kondensatory komórek pamięci w wierszu matrycy „aktywowanym" przez $\overline{RAS}$ rozładowują się i pochodzące z nich sygnały (po wzmocnieniu) są podawane na multiplekser razem z $\overline{CAS}$.

Multiplekser „wypuszcza" sygnał z jednej kolumny jako odczytany bit danych. Kombinacja bitów $\overline{CAS}$ decyduje o tym, z której kolumny pochodzi ten sygnał.

Odczyt z pamięci DRAM

Wszystkie naładowane kondensatory komórek pamięci w wierszu matrycy „aktywowanym" przez $\overline{RAS}$ rozładowują się i pochodzące z nich sygnały (po wzmocnieniu) są podawane na multiplekser razem z $\overline{CAS}$.

Multiplekser „wypuszcza" sygnał z jednej kolumny jako odczytany bit danych. Kombinacja bitów $\overline{CAS}$ decyduje o tym, z której kolumny pochodzi ten sygnał.

Aby uzyskać liczbę bitów odpowiadającą szerokości magistrali pamięci (typowo 64), należy równolegle przeprowadzać operacje odczytu na odpowiedniej liczbie matryc DRAM.

Zapis do pamięci DRAM

Wszystkie naładowane kondensatory komórek pamięci w wierszu matrycy „aktywowanym” przez $\overline{RAS}$ rozładowują się.

Zapis do pamięci DRAM

Wszystkie naładowane kondensatory komórek pamięci w wierszu matrycy „aktywowanym” przez $\overline{RAS}$ rozładowują się.

$\overline{CAS}$ oraz bit danych do zapisania są podawane na demultiplekser, który (jeśli bit do zapisania odpowiada niezerowemu sygnałowi) generuje sygnał na linii odpowiadającej wybranej kolumnie. W komórce pamięci w tej kolumnie zapisywana jest wartość reprezentowana przez wejściowy bit.

Zapis do pamięci DRAM

Wszystkie naładowane kondensatory komórek pamięci w wierszu matrycy „aktywowanym" przez $\overline{RAS}$ rozładowują się.

$\overline{CAS}$ oraz bit danych do zapisania są podawane na demultiplekser, który (jeśli bit do zapisania odpowiada niezerowemu sygnałowi) generuje sygnał na linii odpowiadającej wybranej kolumnie. W komórce pamięci w tej kolumnie zapisywana jest wartość reprezentowana przez wejściowy bit.

W pozostałych komórkach w wierszu wybranym przez $\overline{RAS}$ zapisywana jest ich poprzednia zawartość.

Przesyłanie $\overline{RAS}$ i $\overline{CAS}$

Każda dodatkowa linia adresowa w magistrali pamięci zwiększa koszt produkcji komputera.

Przesyłanie $\overline{RAS}$ i $\overline{CAS}$

Każda dodatkowa linia adresowa w magistrali pamięci zwiększa koszt produkcji komputera.

W związku z tym, dla ograniczyenia liczby potrzebnych linii adresowych do niezbędnego minimum, słowa $\overline{RAS}$ i $\overline{CAS}$ są przesyłane oddzielnie tymi samymi liniami adresowymi.

Przesyłanie $\overline{RAS}$ i $\overline{CAS}$

Każda dodatkowa linia adresowa w magistrali pamięci zwiększa koszt produkcji komputera.

W związku z tym, dla ograniczyenia liczby potrzebnych linii adresowych do niezbędnego minimum, słowa $\overline{RAS}$ i $\overline{CAS}$ są przesyłane oddzielnie tymi samymi liniami adresowymi.

Odstęp czasu od wysłania $\overline{RAS}$ do wysłania $\overline{CAS}$ (*ang.* $\overline{RAS}$*-to-*$\overline{CAS}$ *delay*) musi być ściśle dostosowany do możliwości matryc pamięci w modułach DRAM.

Przesyłanie $\overline{RAS}$ i $\overline{CAS}$

Każda dodatkowa linia adresowa w magistrali pamięci zwiększa koszt produkcji komputera.

W związku z tym, dla ograniczyenia liczby potrzebnych linii adresowych do niezbędnego minimum, słowa $\overline{RAS}$ i $\overline{CAS}$ są przesyłane oddzielnie tymi samymi liniami adresowymi.

Odstęp czasu od wysłania $\overline{RAS}$ do wysłania $\overline{CAS}$ (*ang.* $\overline{RAS}$*-to-*$\overline{CAS}$ *delay*) musi być ściśle dostosowany do możliwości matryc pamięci w modułach DRAM.

Zapisanie (lub odczytanie) większej liczby bajtów wymaga przeprowadzenia ciągu takich operacji.

Protokół dostępu do pamięci DRAM

Synchroniczne pamięci DRAM (SDRAM)

Wszystkie operacje z udziałem magistrali pamięci są przeprowadzane zgodnie z dodatkowym sygnałem zegarowym (*ang. clock*).

Protokół dostępu do pamięci DRAM

Synchroniczne pamięci DRAM (SDRAM)

Wszystkie operacje z udziałem magistrali pamięci są przeprowadzane zgodnie z dodatkowym sygnałem zegarowym (*ang. clock*).

W najprostszej wersji protokołu wszystkie operacje zaczynają się przy narastającym zboczu (*ang. rising edge*) sygnału zegarowego.

Protokół dostępu do pamięci DRAM

Synchroniczne pamięci DRAM (SDRAM)

Wszystkie operacje z udziałem magistrali pamięci są przeprowadzane zgodnie z dodatkowym sygnałem zegarowym (*ang. clock*).

W najprostszej wersji protokołu wszystkie operacje zaczynają się przy narastającym zboczu (*ang. rising edge*) sygnału zegarowego.

Procedura odczytu

1. Numer wiersza na liniach adresowych, aktywny sygnał $\overline{RAS}$.
2. Odstęp czasu t_{RCD}.
3. Numer kolumny na liniach adresowych, aktywny sygnał $\overline{CAS}$.
4. Odstęp czasu t_{CL} (*ang.* $\overline{CAS}$ *Latency*).
5. Dane na liniach danych.

Protokół dostępu do pamięci DRAM (c. d.)

Po przesłaniu danych potrzebny jest czas na przygotowanie następnego numeru wiersza do zaadresowania, t_{RP} (*ang. Row Precharge*).

Protokół dostępu do pamięci DRAM (c. d.)

Po przesłaniu danych potrzebny jest czas na przygotowanie następnego numeru wiersza do zaadresowania, t_{RP} (*ang. Row Precharge*).

Dodatkowo, po aktywowaniu sygnału $\overline{RAS}$ musi upłynąć pewien czas, po którym może on być ponownie aktywowany, t_{RAS} (*ang.* $\overline{RAS}$ *active to precharge delay*).

Protokół dostępu do pamięci DRAM (c. d.)

Po przesłaniu danych potrzebny jest czas na przygotowanie następnego numeru wiersza do zaadresowania, t_{RP} (*ang. Row Precharge*).

Dodatkowo, po aktywowaniu sygnału $\overline{RAS}$ musi upłynąć pewien czas, po którym może on być ponownie aktywowany, t_{RAS} (*ang.* $\overline{RAS}$ *active to precharge delay*).

W niektórych przypadkach t_{RAS} może być dłuższy od sumy t_{RCD}, t_{CL} oraz czasu przesyłania adresów i danych.

Protokół dostępu do pamięci DRAM (c. d.)

Po przesłaniu danych potrzebny jest czas na przygotowanie następnego numeru wiersza do zaadresowania, t_{RP} (*ang. Row Precharge*).

Dodatkowo, po aktywowaniu sygnału $\overline{RAS}$ musi upłynąć pewien czas, po którym może on być ponownie aktywowany, t_{RAS} (*ang.* $\overline{RAS}$ *active to precharge delay*).

W niektórych przypadkach t_{RAS} może być dłuższy od sumy t_{RCD}, t_{CL} oraz czasu przesyłania adresów i danych.

Dlatego, jeżeli w następnej operacji dane mają być pobierane z tego samego wiersza, jego numer jest „zatrzaskiwany” (*ang. latch*) przez kontroler pamięci i opóźnienia t_{RAS} oraz t_{RP} można pominąć.

Protokół dostępu do pamięci DRAM (c. d.)

Sygnał $\overline{CAS}$ może być wielokrotnie aktywowany przy „zatrzaśniętym" numerze wiersza, jednak nie można tego robić zbyt często.

Protokół dostępu do pamięci DRAM (c. d.)

Sygnał $\overline{CAS}$ może być wielokrotnie aktywowany przy „zatrzaśniętym" numerze wiersza, jednak nie można tego robić zbyt często.

Decyduje o tym parametr zwany szybkością przetwarzania poleceń (*ang. Command Rate*) modułów pamięci.

Protokół dostępu do pamięci DRAM (c. d.)

Sygnał $\overline{CAS}$ może być wielokrotnie aktywowany przy „zatrzaśniętym" numerze wiersza, jednak nie można tego robić zbyt często.

Decyduje o tym parametr zwany szybkością przetwarzania poleceń (*ang. Command Rate*) modułów pamięci.

Dodatkowo, jeśli kolejne dostępy dotyczą danych z tego samego wiersza i wielu kolejnych kolumn, mogą one być przesłane w ciągu jednej operacji, w tak zwanym trybie nagłym (*ang. burst*).

Protokół dostępu do pamięci DRAM (c. d.)

Sygnał $\overline{CAS}$ może być wielokrotnie aktywowany przy „zatrzaśniętym" numerze wiersza, jednak nie można tego robić zbyt często.

Decyduje o tym parametr zwany szybkością przetwarzania poleceń (*ang. Command Rate*) modułów pamięci.

Dodatkowo, jeśli kolejne dostępy dotyczą danych z tego samego wiersza i wielu kolejnych kolumn, mogą one być przesłane w ciągu jednej operacji, w tak zwanym trybie nagłym (*ang. burst*).

Dlatego z punktu widzenia wydajności najlepiej jest programować dostępy do pamięci tak, aby dotyczyły one jak najczęściej danych przechowywanych w tych samych wierszach matryc DRAM.

Wpływ odświeżania pamięci DRAM

Odświeżanie pamięci DRAM jest przeprowadzane wiersz po wierszu.

Wpływ odświeżania pamięci DRAM

Odświeżanie pamięci DRAM jest przeprowadzane wiersz po wierszu.

W trakcie odświeżania jednego wiersza matryca pamięci nie jest dostępna dla użytecznego odczytu i zapisu danych.

Wpływ odświeżania pamięci DRAM

Odświeżanie pamięci DRAM jest przeprowadzane wiersz po wierszu.

W trakcie odświeżania jednego wiersza matryca pamięci nie jest dostępna dla użytecznego odczytu i zapisu danych.

Zgodnie z obowiązującymi standardami każda komórka DRAM powinna być odświeżana co 64 *ms*. Oznacza to, że w praktyce polecenie odświeżenia wiersza jest wysyłane do matrycy DRAM co około 7,8 μs.

Wpływ odświeżania pamięci DRAM

Odświeżanie pamięci DRAM jest przeprowadzane wiersz po wierszu.

W trakcie odświeżania jednego wiersza matryca pamięci nie jest dostępna dla użytecznego odczytu i zapisu danych.

Zgodnie z obowiązującymi standardami każda komórka DRAM powinna być odświeżana co 64 *ms*. Oznacza to, że w praktyce polecenie odświeżenia wiersza jest wysyłane do matrycy DRAM co około 7,8 μs.

Czas trwania jednej operacji odświeżania wiersza zależy od modułów DRAM i (jeśli jest zbyt długi) może powodować znaczne opóźnienia w dostępie do danych.

Różnice szybkości między procesorami i pamięciami DRAM

W latach 1990-2010 miał miejsce bardzo szybki rozwój technologii półprzewodnikowych, pozwalający na wielokrotne zmniejszenie rozmiarów i skrócenie czasu przełączania tranzystorów.

Różnice szybkości między procesorami i pamięciami DRAM

W latach 1990-2010 miał miejsce bardzo szybki rozwój technologii półprzewodnikowych, pozwalający na wielokrotne zmniejszenie rozmiarów i skrócenie czasu przełączania tranzystorów.

Niestety nie jest możliwe podobne przspieszenie matryc pamięci DRAM.

Różnice szybkości między procesorami i pamięciami DRAM

W latach 1990-2010 miał miejsce bardzo szybki rozwój technologii półprzewodnikowych, pozwalający na wielokrotne zmniejszenie rozmiarów i skrócenie czasu przełączania tranzystorów.

Niestety nie jest możliwe podobne przspieszenie matryc pamięci DRAM.

Stworzono technologie pozwalające korzystać z wielu matryc DRAM równolegle, ale pojawiła się kolejna przeszkoda w postaci trudności w konstruowaniu magistral pamięci (im większa jest częstotliwość zegara magistrali, tym trudniejsze jest poprowadzenie linii w taki sposób, aby magistrala nie była zbyt czuła na zakłócenia).

Różnice szybkości między procesorami i pamięciami DRAM

W latach 1990-2010 miał miejsce bardzo szybki rozwój technologii półprzewodnikowych, pozwalający na wielokrotne zmniejszenie rozmiarów i skrócenie czasu przełączania tranzystorów.

Niestety nie jest możliwe podobne przspieszenie matryc pamięci DRAM.

Stworzono technologie pozwalające korzystać z wielu matryc DRAM równolegle, ale pojawiła się kolejna przeszkoda w postaci trudności w konstruowaniu magistral pamięci (im większa jest częstotliwość zegara magistrali, tym trudniejsze jest poprowadzenie linii w taki sposób, aby magistrala nie była zbyt czuła na zakłócenia).

Nawet ostatnia generacja pamięci (DDR3 SDRAM) nie pozwala na zredukowanie różnicy szybkości działania między procesorami i DRAM.

Problemy z DRAM i schowki procesorów

Odwołania do pamięci DRAM z programu wykonywanego przez współczesny procesor mogą powodować opóźnienia wykonania tego programu rzędu 100-250 (i więcej) cykli zegara.

Problemy z DRAM i schowki procesorów

Odwołania do pamięci DRAM z programu wykonywanego przez współczesny procesor mogą powodować opóźnienia wykonania tego programu rzędu 100-250 (i więcej) cykli zegara.

W przypadku zastąpienia pamięci DRAM pamięciami statycznymi, opóźnienia te można by było znacznie zredukować (do ok. 90%).

Problemy z DRAM i schowki procesorów

Odwołania do pamięci DRAM z programu wykonywanego przez współczesny procesor mogą powodować opóźnienia wykonania tego programu rzędu 100-250 (i więcej) cykli zegara.

W przypadku zastąpienia pamięci DRAM pamięciami statycznymi, opóźnienia te można by było znacznie zredukować (do ok. 90%).

Niestety z ekonomicznego punktu widzenia jest to nierealne.

Problemy z DRAM i schowki procesorów

Odwołania do pamięci DRAM z programu wykonywanego przez współczesny procesor mogą powodować opóźnienia wykonania tego programu rzędu 100-250 (i więcej) cykli zegara.

W przypadku zastąpienia pamięci DRAM pamięciami statycznymi, opóźnienia te można by było znacznie zredukować (do ok. 90%).

Niestety z ekonomicznego punktu widzenia jest to nierealne.

W związku z tym pojawiła się koncepcja wprowadzenia tak zwanych schowków (*ang. cache*), czyli bloków statycznej pamięci RAM służących do tymczasowego przechowywania najczęściej używanych danych.

Strategia wykorzystania schowka

Pamięć główna (*ang. main memory*)

Pamięć (zwykle zbudowana z modułów DRAM), w której przechowywana jest większość danych (i rozkazów) wykorzystywanych przez procesory w trakcie wykonywania programów.

Strategia wykorzystania schowka

Pamięć główna (*ang. main memory*)

Pamięć (zwykle zbudowana z modułów DRAM), w której przechowywana jest większość danych (i rozkazów) wykorzystywanych przez procesory w trakcie wykonywania programów.

Dane odczytywane przez procesor z pamięci głównej najpierw trafiają do schowka. Dane, które już znajdują się w schowku, nie muszą być pobierane z pamięci głównej.

Strategia wykorzystania schowka

Pamięć główna (*ang. main memory*)

Pamięć (zwykle zbudowana z modułów DRAM), w której przechowywana jest większość danych (i rozkazów) wykorzystywanych przez procesory w trakcie wykonywania programów.

Dane odczytywane przez procesor z pamięci głównej najpierw trafiają do schowka. Dane, które już znajdują się w schowku, nie muszą być pobierane z pamięci głównej.

Strategia zapisu zwrotnego (*ang. write back*)

W przypadku zapisu, dane są początkowo zapisywane w schowku. Są one kopiowane do pamięci głównej w czasie, gdy w schowku trzeba je zastąpić innymi danymi (tzn. z opóźnieniem).

Zasada lokalności

Ponieważ częstotliwość zegara dla komórek pamięci, z których zbudowany jest schowek, może być taka, jak dla procesora, powinno być jasne, że zastosowanie schowka znacznie zwiększa wydajność przy wykonywaniu niewielkich programów.

Zasada lokalności

Ponieważ częstotliwość zegara dla komórek pamięci, z których zbudowany jest schowek, może być taka, jak dla procesora, powinno być jasne, że zastosowanie schowka znacznie zwiększa wydajność przy wykonywaniu niewielkich programów.

Okazuje się, że dzięki zastosowaniu schowków można zwiększyć szybkość wykonywania wszystkich programów, ponieważ większość programów wykazuje tendencję do „koncentrowania się" na stosunkowo małych blokach danych.

Zasada lokalności

Ponieważ częstotliwość zegara dla komórek pamięci, z których zbudowany jest schowek, może być taka, jak dla procesora, powinno być jasne, że zastosowanie schowka znacznie zwiększa wydajność przy wykonywaniu niewielkich programów.

Okazuje się, że dzięki zastosowaniu schowków można zwiększyć szybkość wykonywania wszystkich programów, ponieważ większość programów wykazuje tendencję do „koncentrowania się" na stosunkowo małych blokach danych.

Zasada lokalności (*ang. locality principle*)

Większość odwołań do pamięci w programach ma lokalny charakter (tzn. średnia różnica adresów przy kolejnych odwołaniach do pamięci jest stosunkowo niewielka).

Hierarchie schowków

Początkowo procesory dysponowały jednym schowkiem, jednak okazało się, że korzystne jest przechowywanie rozkazów i danych w różnych schowkach.

Hierarchie schowków

Początkowo procesory dysponowały jednym schowkiem, jednak okazało się, że korzystne jest przechowywanie rozkazów i danych w różnych schowkach.

Z przyczyn technicznych czas potrzebny na pobranie danych ze schowka lub zapis danych do schowka jest tym większy, im większe są rozmiary schowka.

Hierarchie schowków

Początkowo procesory dysponowały jednym schowkiem, jednak okazało się, że korzystne jest przechowywanie rozkazów i danych w różnych schowkach.

Z przyczyn technicznych czas potrzebny na pobranie danych ze schowka lub zapis danych do schowka jest tym większy, im większe są rozmiary schowka.

Dlatego zaczęto stosować niewielkie schowki I poziomu (*ang. level 1 cache*) oddzielne dla rozkazów i danych oraz znacznie większe schowki II poziomu (*ang. level 2 cache*).

Hierarchie schowków

Początkowo procesory dysponowały jednym schowkiem, jednak okazało się, że korzystne jest przechowywanie rozkazów i danych w różnych schowkach.

Z przyczyn technicznych czas potrzebny na pobranie danych ze schowka lub zapis danych do schowka jest tym większy, im większe są rozmiary schowka.

Dlatego zaczęto stosować niewielkie schowki I poziomu (*ang. level 1 cache*) oddzielne dla rozkazów i danych oraz znacznie większe schowki II poziomu (*ang. level 2 cache*).

Przy okazji wprowadzenia na rynek procesorów wielordzeniowych (*ang. multicore*) stwierdzono, że korzystne jest dodanie III poziomu schowków (wspólnych dla wielu rdzeni).

Wiersze schowka

Przy kopiowaniu danych z pamięci głównej do schowka (lub ze schowka do pamięci głównej przy zapisie zwrotnym) poszczególne bajty danych nie są kopiowane oddzielnie (byłoby to bardzo niewydajne).

Wiersze schowka

Przy kopiowaniu danych z pamięci głównej do schowka (lub ze schowka do pamięci głównej przy zapisie zwrotnym) poszczególne bajty danych nie są kopiowane oddzielnie (byłoby to bardzo niewydajne).

Pamięć główna jest podzielona na rozłączne bloki o jednakowych rozmiarach (zwykle 32 B lub 64 B), z których każdy jest kopiowany w całości.

Wiersze schowka

Przy kopiowaniu danych z pamięci głównej do schowka (lub ze schowka do pamięci głównej przy zapisie zwrotnym) poszczególne bajty danych nie są kopiowane oddzielnie (byłoby to bardzo niewydajne).

Pamięć główna jest podzielona na rozłączne bloki o jednakowych rozmiarach (zwykle 32 B lub 64 B), z których każdy jest kopiowany w całości.

Odwołanie (w programie) do jednej z lokacji w obrębie bloku powoduje skopiowanie całego bloku.

Wiersze schowka

Przy kopiowaniu danych z pamięci głównej do schowka (lub ze schowka do pamięci głównej przy zapisie zwrotnym) poszczególne bajty danych nie są kopiowane oddzielnie (byłoby to bardzo niewydajne).

Pamięć główna jest podzielona na rozłączne bloki o jednakowych rozmiarach (zwykle 32 B lub 64 B), z których każdy jest kopiowany w całości.

Odwołanie (w programie) do jednej z lokacji w obrębie bloku powoduje skopiowanie całego bloku.

Schowek jest podzielony na bloki o rozmiarach identycznych z rozmiarami bloków w pamięci głównej, zwane wierszami schowka (*ang. cache line*), w których można zapisywać dane pochodzące z bloków w pamięci głównej.

Pobieranie z wyprzedzeniem

Obserwacja I

Programy często odwołują się do sekwencji kolejnych lokacji w pamięci (tzn. np. po owołaniu do lokacji o adresie n następuje odwołanie do lokacji $n + 1$ itd.).

Pobieranie z wyprzedzeniem

Obserwacja I

Programy często odwołują się do sekwencji kolejnych lokacji w pamięci (tzn. np. po owołaniu do lokacji o adresie n następuje odwołanie do lokacji $n + 1$ itd.).

Obserwacja II

Protokół dostępu do pamięci DRAM sprzyja przeprowadzaniu operacji na ciągach lokacji o kolejnych adresach.

Pobieranie z wyprzedzeniem

Obserwacja I

Programy często odwołują się do sekwencji kolejnych lokacji w pamięci (tzn. np. po owołaniu do lokacji o adresie n następuje odwołanie do lokacji $n + 1$ itd.).

Obserwacja II

Protokół dostępu do pamięci DRAM sprzyja przeprowadzaniu operacji na ciągach lokacji o kolejnych adresach.

Konstruktorzy procesorów usiłują wykorzystać te obserwacje poprzez stosowanie spekulatywnego pobierania z wyprzedzeniem prefetch danych z bloków w pamięci głównej, do których prawdopodobnie nastąpi odwołanie w dalszej części programu.

Organizacja schowka

Schowek można wyobrażać sobie jako tablicę złożoną z 2^k wierszy o rozmiarach 2^w B każdy. Łączna pojemność schowka wynosi wtedy 2^{k+w} B.

Organizacja schowka

Schowek można wyobrażać sobie jako tablicę złożoną z 2^k wierszy o rozmiarach 2^w B każdy. Łączna pojemność schowka wynosi wtedy 2^{k+w} B.

Jeżeli pamięć główna ma pojemność $2^M = 2^m \times 2^w$ B, to na jeden wiersz schowka przypada 2^{m-k} bloków w pamięci głównej ($m > k$).

Organizacja schowka

Schowek można wyobrażać sobie jako tablicę złożoną z 2^k wierszy o rozmiarach 2^w B każdy. Łączna pojemność schowka wynosi wtedy 2^{k+w} B.

Jeżeli pamięć główna ma pojemność $2^M = 2^m \times 2^w$ B, to na jeden wiersz schowka przypada 2^{m-k} bloków w pamięci głównej ($m > k$).

Okazuje się, że najlepsze efekty dają zbiorowo-asocjatywne (*ang. set-associative*) sposoby przyporządkowywania bloków w pamięci głównej do wierszy schowka.

Organizacja schowka

Schowek można wyobrażać sobie jako tablicę złożoną z 2^k wierszy o rozmiarach 2^w B każdy. Łączna pojemność schowka wynosi wtedy 2^{k+w} B.

Jeżeli pamięć główna ma pojemność $2^M = 2^m \times 2^w$ B, to na jeden wiersz schowka przypada 2^{m-k} bloków w pamięci głównej ($m > k$).

Okazuje się, że najlepsze efekty dają zbiorowo-asocjatywne (*ang. set-associative*) sposoby przyporządkowywania bloków w pamięci głównej do wierszy schowka.

Schowek jest dzielony na 2^s części, zwanych zbiorami (*ang. set*), po 2^r wierszy w każdym. Wtedy $k = s + r$.

Organizacja schowka (c. d.)

Ponumerujmy bloki w pamięci głównej od 0 do $(2^m - 1)$ i wprowadźmy przyporządkowanie, w którym bloki o numerach $n + j \times 2^s$, gdzie $n < 2^s$, odpowiadają temu samemu zbiorowi wierszy w schowku.

Organizacja schowka (c. d.)

Ponumerujmy bloki w pamięci głównej od 0 do $(2^m - 1)$ i wprowadźmy przyporządkowanie, w którym bloki o numerach $n + j \times 2^s$, gdzie $n < 2^s$, odpowiadają temu samemu zbiorowi wierszy w schowku.

Przyjmijmy, że dla każdego z tych bloków w pamięci głównej dane będą mogły być przechowywane tylko w jednym z wierszy schowka należących do zbioru, który odpowiada temu blokowi.

Organizacja schowka (c. d.)

Ponumerujmy bloki w pamięci głównej od 0 do $(2^m - 1)$ i wprowadźmy przyporządkowanie, w którym bloki o numerach $n + j \times 2^s$, gdzie $n < 2^s$, odpowiadają temu samemu zbiorowi wierszy w schowku.

Przyjmijmy, że dla każdego z tych bloków w pamięci głównej dane będą mogły być przechowywane tylko w jednym z wierszy schowka należących do zbioru, który odpowiada temu blokowi.

Wtedy dla każdego bloku w pamięci głównej dane z tego bloku mogą być przechowywane w schowku na 2^r sposobów oraz istnieje $2^{m-s-r} - 1$ bloków, które „konkurują” z nim o „dostęp” do schowka.

Organizacja schowka (c. d.)

Ponumerujmy bloki w pamięci głównej od 0 do $(2^m - 1)$ i wprowadźmy przyporządkowanie, w którym bloki o numerach $n + j \times 2^s$, gdzie $n < 2^s$, odpowiadają temu samemu zbiorowi wierszy w schowku.

Przyjmijmy, że dla każdego z tych bloków w pamięci głównej dane będą mogły być przechowywane tylko w jednym z wierszy schowka należących do zbioru, który odpowiada temu blokowi.

Wtedy dla każdego bloku w pamięci głównej dane z tego bloku mogą być przechowywane w schowku na 2^r sposobów oraz istnieje $2^{m-s-r} - 1$ bloków, które „konkurują" z nim o „dostęp" do schowka.

Przy tym dane z 2^r bloków w każdej grupie 2^{m-s} bloków w pamięci głównej mogą jednocześnie być w schowku.

Organizacja schowka (c. d.)

Wtedy mówi się, że schowek ma 2^r-torową organizację zbiorowo-asocjatywną (*ang.* 2^r*-way set-associative*).

Organizacja schowka (c. d.)

Wtedy mówi się, że schowek ma 2^r-torową organizację zbiorowo-asocjatywną (*ang.* 2^r*-way set-associative*).

Okazuje się, że dla $r = 1$ wydajnośc rośnie blisko dwukrotnie w stosunku do $r = 0$. Jednak dla $r > 3$ wydajność jest praktycznie taka, jak dla $r = 3$.

Organizacja schowka (c. d.)

Wtedy mówi się, że schowek ma 2^r-torową organizację zbiorowo-asocjatywną (*ang.* 2^r*-way set-associative*).

Okazuje się, że dla $r = 1$ wydajnośc rośnie blisko dwukrotnie w stosunku do $r = 0$. Jednak dla $r > 3$ wydajność jest praktycznie taka, jak dla $r = 3$.

Ogólnie stopień skomplikowania elektroniki realizującej schowek rośnie wraz z r, natomiast czas dostępu do schowka rośnie wraz z s.

Organizacja schowka (c. d.)

Wtedy mówi się, że schowek ma 2^r-torową organizację zbiorowo-asocjatywną (*ang.* 2^r*-way set-associative*).

Okazuje się, że dla $r = 1$ wydajnośc rośnie blisko dwukrotnie w stosunku do $r = 0$. Jednak dla $r > 3$ wydajność jest praktycznie taka, jak dla $r = 3$.

Ogólnie stopień skomplikowania elektroniki realizującej schowek rośnie wraz z r, natomiast czas dostępu do schowka rośnie wraz z s.

Dlatego schowki I poziomu zwykle mają 2-torową organizację zbiorowo-asocjatywną, natomiast dla schowków II i III poziomu r może być w granicach od 2 do 4.

Adresy fizyczne i wirtualne

Adresy używane przez programy na ogół nie są adresami, których używa kontroler pamięci do dentyfikacji bitów w matrycach DRAM.

Adresy fizyczne i wirtualne

Adresy używane przez programy na ogół nie są adresami, których używa kontroler pamięci do dentyfikacji bitów w matrycach DRAM.

Adresy fizyczne (*ang. physical*)

Adresy używane do identyfikacji lokacji pamięci przy fizycznym przesyłaniu danych (np. z CPU do modułów DRAM).

Adresy fizyczne i wirtualne

Adresy używane przez programy na ogół nie są adresami, których używa kontroler pamięci do dentyfikacji bitów w matrycach DRAM.

Adresy fizyczne (*ang. physical*)

Adresy używane do identyfikacji lokacji pamięci przy fizycznym przesyłaniu danych (np. z CPU do modułów DRAM).

Adresy wirtualne (*ang. virtual*)

Adresy używane przez oprogramowanie przy odwołaniach do pamięci.

Adresy fizyczne i wirtualne

Adresy używane przez programy na ogół nie są adresami, których używa kontroler pamięci do dentyfikacji bitów w matrycach DRAM.

Adresy fizyczne (*ang. physical*)

Adresy używane do identyfikacji lokacji pamięci przy fizycznym przesyłaniu danych (np. z CPU do modułów DRAM).

Adresy wirtualne (*ang. virtual*)

Adresy używane przez oprogramowanie przy odwołaniach do pamięci.

Jednemu adresowi fizycznemu może odpowiadać wiele adresów wirtualnych.

Bloki stronicowe

Przyporządkowanie między adresami wirtualnymi i fizycznymi jest realizowane poprzez użycie stron pamięci (*ang. memory page*) oraz tabel translacji (*ang. translation table*).

Bloki stronicowe

Przyporządkowanie między adresami wirtualnymi i fizycznymi jest realizowane poprzez użycie stron pamięci (*ang. memory page*) oraz tabel translacji (*ang. translation table*).

Fizyczna przestrzeń adresowa (*ang. physical address space*)

Zbiór wszystkich możliwych fizycznych adresów, których może używać dany procesor (lub ogólnie rodzina procesorów). Jej rozmiar jest maksymalnym możliwym rozmiarem pamięci RAM dla tego procesora.

Bloki stronicowe

Przyporządkowanie między adresami wirtualnymi i fizycznymi jest realizowane poprzez użycie stron pamięci (*ang. memory page*) oraz tabel translacji (*ang. translation table*).

Fizyczna przestrzeń adresowa (*ang. physical address space*)

Zbiór wszystkich możliwych fizycznych adresów, których może używać dany procesor (lub ogólnie rodzina procesorów). Jej rozmiar jest maksymalnym możliwym rozmiarem pamięci RAM dla tego procesora.

Fizyczna przestrzeń adresowa jest dzielona na bloki stronicowe (*ang. page frame*) o rozmiarach 2^b B każdy ($b >= 12$). Jeżeli f jest numerem bloku stronicowego, to fizyczne adresy lokacji w obrębie tego bloku mają postać $f + o$, gdzie $o < 2^b$ nazywa się ofsetem (*ang. offset*).

Strony pamięci

Wirtualna przestrzeń adresowa (*ang. virtual address space*)

Zbiór wszystkich możliwych wirtualnych adresów, których może używać oprogramowanie wykonywane przez dany procesor (lub ogólnie rodzinę procesorów).

Strony pamięci

Wirtualna przestrzeń adresowa (*ang. virtual address space*)

Zbiór wszystkich możliwych wirtualnych adresów, których może używać oprogramowanie wykonywane przez dany procesor (lub ogólnie rodzinę procesorów).

Rozmiary wirtualnej przestrzeni adresowej na ogół są różne od rozmiarów fizycznej przestrzeni adresowej (i różne od rozmiarów pamięci RAM).

Strony pamięci

Wirtualna przestrzeń adresowa (*ang. virtual address space*)

Zbiór wszystkich możliwych wirtualnych adresów, których może używać oprogramowanie wykonywane przez dany procesor (lub ogólnie rodzinę procesorów).

Rozmiary wirtualnej przestrzeni adresowej na ogół są różne od rozmiarów fizycznej przestrzeni adresowej (i różne od rozmiarów pamięci RAM).

Potrzebna jest metoda wyznaczania adresu fizycznego dla danego adresu wirtualnego.

Strony pamięci

Wirtualna przestrzeń adresowa (*ang. virtual address space*)

Zbiór wszystkich możliwych wirtualnych adresów, których może używać oprogramowanie wykonywane przez dany procesor (lub ogólnie rodzinę procesorów).

Rozmiary wirtualnej przestrzeni adresowej na ogół są różne od rozmiarów fizycznej przestrzeni adresowej (i różne od rozmiarów pamięci RAM).

Potrzebna jest metoda wyznaczania adresu fizycznego dla danego adresu wirtualnego.

W tym celu bloki adresów w wirtualnej przestrzeni adresowej, zwane stronami pamięci (*ang. memory page*), mapuje się na odpowiednie bloki stronicowe.

Translacja adresów stron pamięci

Część adresu wirtualnego „ponad" ofsetem dzielona jest na ciągi bitów (zwykle o jednakowej długości). Nazwijmy je $k_1, \ldots, k_n$ tak, aby k_1 był ciągiem najmniej znaczących, a k_n – ciągiem najbardziej znaczących bitów.

Translacja adresów stron pamięci

Część adresu wirtualnego „ponad" ofsetem dzielona jest na ciągi bitów (zwykle o jednakowej długości). Nazwijmy je $k_1, \ldots, k_n$ tak, aby k_1 był ciągiem najmniej znaczących, a k_n – ciągiem najbardziej znaczących bitów.

1. k_n jest indeksem w tabeli, której adres fizyczny znajduje się w jednym z rejestrów (*ang. register*) procesora i wyznacza komórkę zawierającą adres fizyczny kolejnej tabeli.
2. k_{n-1} jest indeksem w tabeli, której adres fizyczny został wyznaczony w poprzednim kroku. Określa on komórkę zawierającą adres fizyczny kolejnej tabeli.
3. ...
4. k_1 jest indeksem w tabeli, której adres fizyczny został wyznaczony w poprzednim kroku. Określa on komórkę zawierającą adres fizyczny poszukiwanego bloku stronicowego.

Procesy i tabele translacji

Część adresu wirtualnego odpowiadającą ofsetowi dodaje się do fizycznego adresu bloku stronicowego odczytanego z tabel translacji adresów stron pamięci. W ten sposób otrzymuje się adres fizyczny lokacji, której dotyczny odwołanie do pamięci w programie.

Procesy i tabele translacji

Część adresu wirtualnego odpowiadającą ofsetowi dodaje się do fizycznego adresu bloku stronicowego odczytanego z tabel translacji adresów stron pamięci. W ten sposób otrzymuje się adres fizyczny lokacji, której dotyczny odwołanie do pamięci w programie.

Każdy proces w ogólności dysponuje własną strukturą tabel translacji adresów stron pamięci. Dlatego można powiedzieć, że każdy proces dysponuje własną przestrzenią adresową i „widzi" pamięć komputera w szczególny sposób.

Procesy i tabele translacji

Część adresu wirtualnego odpowiadającą ofsetowi dodaje się do fizycznego adresu bloku stronicowego odczytanego z tabel translacji adresów stron pamięci. W ten sposób otrzymuje się adres fizyczny lokacji, której dotyczny odwołanie do pamięci w programie.

Każdy proces w ogólności dysponuje własną strukturą tabel translacji adresów stron pamięci. Dlatego można powiedzieć, że każdy proces dysponuje własną przestrzenią adresową i „widzi" pamięć komputera w szczególny sposób.

Podczas przełączania zadań procesor jest „przełączany" z tabel translacji stron pamięci jednego procesu na tabele translacji stron pamięci drugiego procesu.

TLB

Mechanizm pamięci wirtualnej bez buforowania byłby bardzo kosztowny (wymaga n odwołań do pamięci w celu uzyskania 1 adresu fizycznego).

TLB

Mechanizm pamięci wirtualnej bez buforowania byłby bardzo kosztowny (wymaga n odwołań do pamięci w celu uzyskania 1 adresu fizycznego).

Dlatego wyniki pewnej liczby ostatnich operacji poszukiwania fizycznego adresu bloku stronicowego dla danego adresu wirtualnego są przechowywane w specjalnym schowku.

TLB

Mechanizm pamięci wirtualnej bez buforowania byłby bardzo kosztowny (wymaga n odwołań do pamięci w celu uzyskania 1 adresu fizycznego).

Dlatego wyniki pewnej liczby ostatnich operacji poszukiwania fizycznego adresu bloku stronicowego dla danego adresu wirtualnego są przechowywane w specjalnym schowku.

TLB (*ang. Translation Lookaside Buffer*)

Schowek, w którym procesor przechowuje fizyczne adresy dla pewnej liczby adresów wirtualnych, ostatnio używanych przez oprogramowanie przy odwołaniach do pamięci.

Wpływ TLB na wydajność

TLB ma strukturę w pełni asocjatywną (*ang. fully associative*), co oznacza, że pełny adres wirtualny strony pamięci jest używany do wyznaczenia wiersza przechowującego odpowiadający mu adres fizyczny.

Wpływ TLB na wydajność

TLB ma strukturę w pełni asocjatywną (*ang. fully associative*), co oznacza, że pełny adres wirtualny strony pamięci jest używany do wyznaczenia wiersza przechowującego odpowiadający mu adres fizyczny.

Dlatego rozmiary TLB są zazwyczaj bardzo ograniczone.

Wpływ TLB na wydajność

TLB ma strukturę w pełni asocjatywną (*ang. fully associative*), co oznacza, że pełny adres wirtualny strony pamięci jest używany do wyznaczenia wiersza przechowującego odpowiadający mu adres fizyczny.

Dlatego rozmiary TLB są zazwyczaj bardzo ograniczone.

Jednocześnie odwołania z użyciem adresów wirtualnych, dla których nie ma adresów fizycznych w TLB są bardzo kosztowne.

Wpływ TLB na wydajność

TLB ma strukturę w pełni asocjatywną (*ang. fully associative*), co oznacza, że pełny adres wirtualny strony pamięci jest używany do wyznaczenia wiersza przechowującego odpowiadający mu adres fizyczny.

Dlatego rozmiary TLB są zazwyczaj bardzo ograniczone.

Jednocześnie odwołania z użyciem adresów wirtualnych, dla których nie ma adresów fizycznych w TLB są bardzo kosztowne.

Dlatego program, który powoduje częste zmiany w TLB (odwołuje się ciągle do innych obszarów przestrzeni adresowej), będzie wykonywany bardzo wolno w porównaniu z programami, które tego nie robią.

Wpływ architektur klasy NUMA na wydajność

NUMA (*ang. Non-Uniform Memory Access*)

Sposób konstrukcji komputera, przy którym pamięć RAM jest podzielona na tak zwane węzły (*ang. node*) w taki sposób, że czas dostępu do pamięci w różnych węzłach z tego samego CPU jest w ogólności różny.

Wpływ architektur klasy NUMA na wydajność

NUMA (*ang. Non-Uniform Memory Access*)

Sposób konstrukcji komputera, przy którym pamięć RAM jest podzielona na tak zwane węzły (*ang. node*) w taki sposób, że czas dostępu do pamięci w różnych węzłach z tego samego CPU jest w ogólności różny.

Zwykle dla każdego procesora (lub rdzenia) istnieje jeden węzeł, do którego ma on „najszybszy” dostęp oraz wiele węzłów, czas dostępu do których jest dla niego znacznie dłuższy.

Wpływ architektur klasy NUMA na wydajność

NUMA (*ang. Non-Uniform Memory Access*)

Sposób konstrukcji komputera, przy którym pamięć RAM jest podzielona na tak zwane węzły (*ang. node*) w taki sposób, że czas dostępu do pamięci w różnych węzłach z tego samego CPU jest w ogólności różny.

Zwykle dla każdego procesora (lub rdzenia) istnieje jeden węzeł, do którego ma on „najszybszy" dostęp oraz wiele węzłów, czas dostępu do których jest dla niego znacznie dłuższy.

Wtedy, jeśli program wykonywany przez procesor odwołuje się do pamięci zlokalizowanej w „bardziej odległych" węzłach pamięci, to w ogólności będzie on wykonywany dużo wolniej od programu, który odwołuje się do pamięci zlokalizowanej w węźle „najbliższym" danemu procesorowi.

Literatura


T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Wprowadzenie do algorytmów* (Wydawnictwa Naukowo-Techniczne, Warszawa, 2001).


Pang Tao, *Metody obliczeniowe w fizyce* (Wydawnictwo Naukowe PWN, Warszawa 2001).


Praca zbiorowa, red. A. Karbowski, E. Niewiadmoska-Szynkiewicz, *Obliczenia równoległe i rozproszone* (Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2001).


U. Drepper, *What every programmer should know about memory* (`http://people.redhat.com/drepper/cpumemory.pdf`).