Wstęp do programowania, część V

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

10 stycznia 2012

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.

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 $j = 1 \dots N$ oraz $k = 1 \dots M$.

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 $j = 1 \dots N$ oraz $k = 1 \dots M$.

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

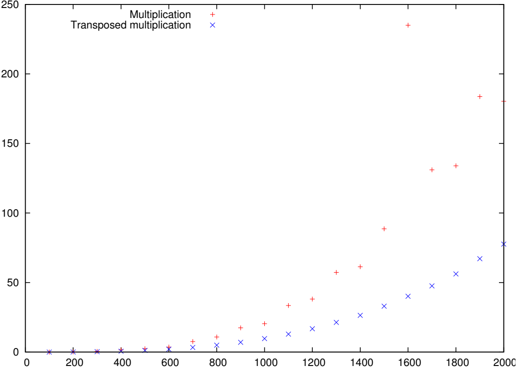
$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \,, \quad D_{ij} = \sum_{k=1}^{n} A_{ik} B_{jk}$$

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)

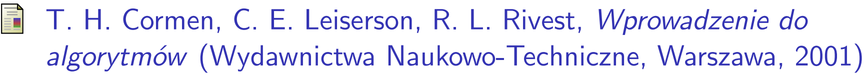
Obliczenia zajmują znacznie mniej czasu w przypadku, gdy w najbardziej wewnętrznej pętli indeks k przebiega elementy macierzowe w jednym wierszu $\widehat{B}$ (a nie w jednej kolumnie).

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.

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).