High-Performance Computing

Algorithms and techniques for matrix computations

Overview

- Matrices – a quick review
  - Matrix addition
  - Matrix multiplication
  - Matrix times vector
- BLAS – routines for matrices/vectors
  - different levels
  - naming conventions
  - calling BLAS from C programs

Matrices and Linear Equations

A close relationship:

A system of linear equations can be written in matrix form:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

Matrix **A** holds the *a* constants

x is a vector of the unknowns

b is a vector of the *b* constants.

Systems of linear equations appear in almost all engineering problems

Matrices — a review

An *n* x *m* matrix:

$$
\begin{array}{cc}
 & \text{Column} \longrightarrow \\
\text{Row} \downarrow &
\begin{bmatrix}
a_{0,0} & a_{0,1} & \cdots & a_{0,m-2} & a_{0,m-1} \\
a_{1,0} & a_{1,1} & \cdots & a_{1,m-2} & a_{1,m-1} \\
\vdots & \vdots & & \vdots & \vdots \\
a_{n-2,0} & a_{n-2,1} & \cdots & a_{n-2,m-2} & a_{n-2,m-1} \\
a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-2} & a_{n-1,m-1}
\end{bmatrix}
\end{array}
$$

indices shown in
C/C++ notation

Matrix Addition

Involves adding corresponding elements of each matrix to form the result matrix:

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

Given the elements of **A** as *a(i,j)* and the elements of **B** as *b(i,j)*, each element of **C** is computed as

$$c_{i,j} = a_{i,j} + b_{i,j}$$

$$(0 \leq i < n,\ 0 \leq j < m)$$

Matrix Multiplication

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

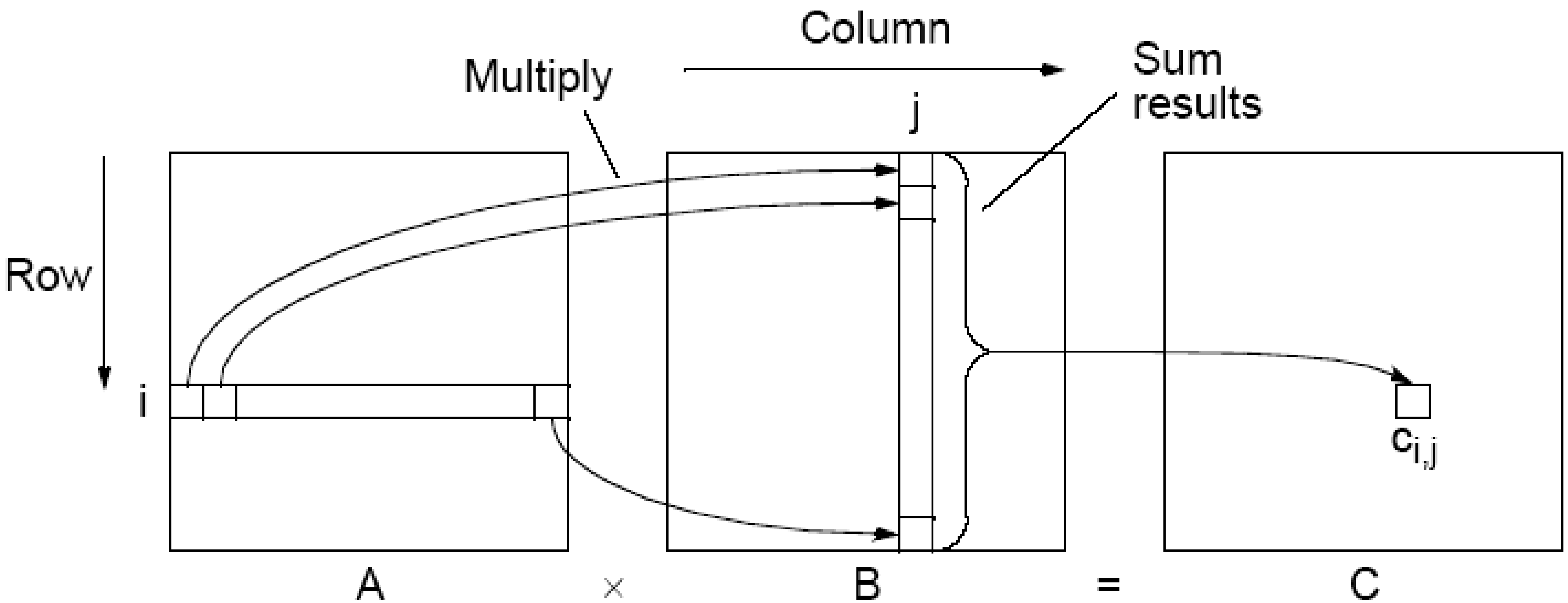
Multiplication of two matrices, **A** and **B**, produces the matrix **C** whose elements, *c(i,j)* (0 <= *i* < *n*, 0 <= *j* < *m*), are computed as follows:

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}$$

where **A** is an *n* x *l* matrix and **B** is an *l* x *m* matrix.

Matrix multiplication

$$C = A \cdot B$$

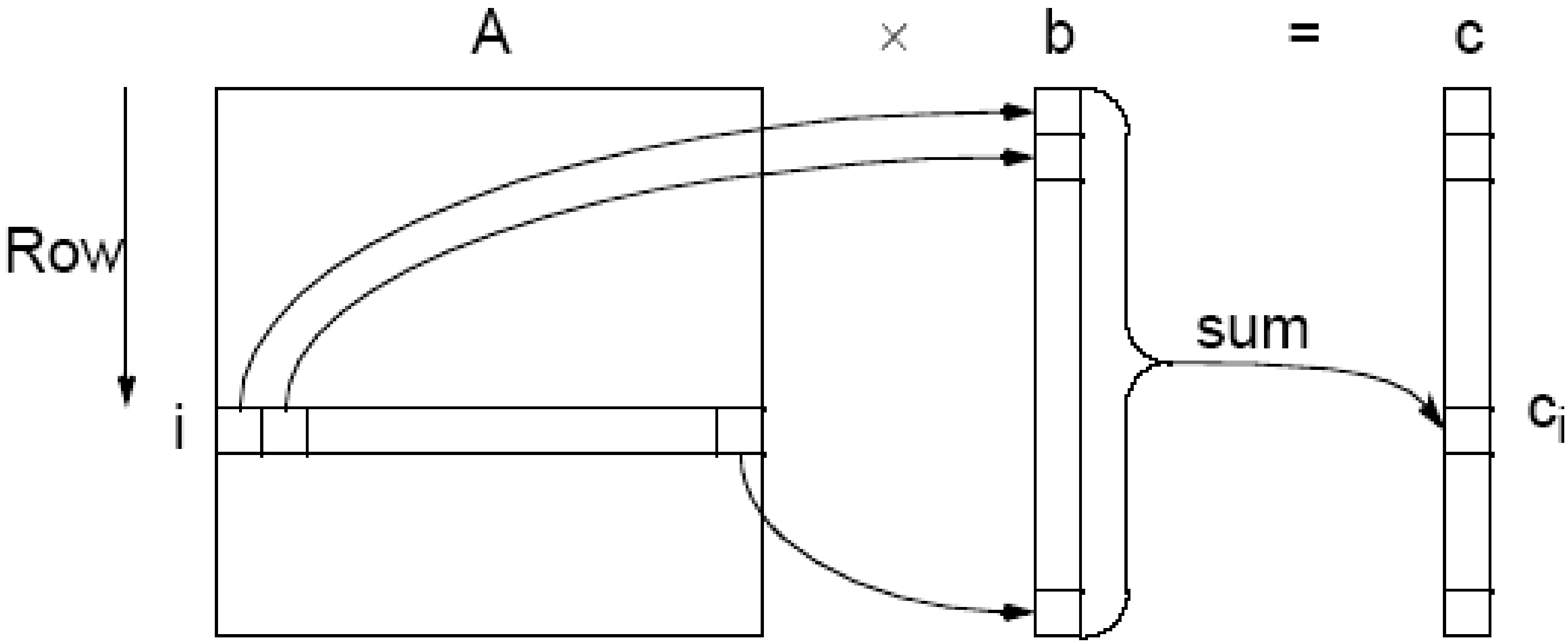


in vector notation: $c(i,j) = a(i) \cdot b(j)$

Matrix-Vector Multiplication

$$\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$$

Matrix-vector multiplication follows directly from the definition of matrix-matrix multiplication by making **B** an *n* x1 matrix (vector). Result an *n* x 1 matrix (vector).



Using a library for matrices/vectors

Basic Linear Algebra Subroutines (BLAS)

- building blocks for linear algebra (de facto standard)
- started as a FORTRAN library (late 1970s)
- linear algebra engine in MATLAB, Python, R, Mathematica, . . .
- high performance when optimized for a specific system/architecture

BLAS levels

BLAS level 1 routines (1970s)

- vector operations, e.g.,

$$x^T y, \quad \|x\|_2, \quad x \leftarrow \alpha x, \quad y \leftarrow \alpha x + y$$

- use $O(n)$ operations for vectors of length n

BLAS level 2 routines (1980s)

- matrix-vector operations, e.g.,

$$y \leftarrow \alpha Ax + \beta y, \quad A \leftarrow \alpha xx^T + A, \quad x \leftarrow T^{-1}b, \ T \text{ triangular}$$

- use $O(mn)$ operations for matrices of size $m \times n$

BLAS levels

BLAS level 3 routines (1980s)

- matrix-matrix routines, e.g.,

$$C \leftarrow \alpha AB + \beta C, \quad X \leftarrow T^{-1}B, \; T \text{ triangular}$$

- use $O(n^3)$ operations for matrices of size $n \times n$

BLAS – what's in a name?

BLAS naming scheme

XYYZZ

- First character X indicates data type (S, D, C, Z)
- BLAS level 1: letters YYZZ indicate mathematical operation
- BLAS level 2+3: letters YY indicate matrix type
- BLAS level 2+3: letters ZZ indicate mathematical operation

Examples

- `dscal` — *double scale* ($x \leftarrow \alpha x$)
- `saxpy` — *single a x plus y* ($y \leftarrow \alpha x + y$)
- `dgemv` — *double general matrix-vector* ($y \leftarrow \alpha A x + \beta y$)
- `dtrsv` — *double triangular solve vector* ($x \leftarrow T^{-1} x$)
- `ssymm` — *single symmetric matrix-matrix* ($C \leftarrow \alpha S B + \beta C$)

BLAS – memory & notations

- vectors and matrices are contiguous arrays
- matrices are stored in column-major ordering
- *stride* refers to distance between *consecutive* elements
- *leading dimension* (LDA) refers to distance between columns

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \end{bmatrix}, \qquad \begin{bmatrix} * & * & * & * & * \\ * & * & A_{23} & A_{24} & A_{25} \\ * & * & A_{33} & A_{34} & A_{35} \\ * & * & * & * & * \end{bmatrix}$$

- ith column of A is a vector of length 4 with stride 1
- ith row of A is a vector of length 5 with stride 4
- $(A_{11}, A_{22}, A_{33}, A_{44})$ is a vector of length 4 with stride 5
- A is a matrix with 4 rows, 5 columns, stride 1, LDA 4
- *slice* (submatrix to the right) has 2 rows, 3 columns, stride 1, LDA 4

Calling (FORTRAN) BLAS from C

```
/* Prototype for BLAS dscal */
void dscal_(
    const int * n,          /* length of array    */
    const double * a,       /* scalar a           */
    double * x,             /* array x            */
    const int * incx        /* array x, stride    */
);

int main(void) {
     int i,incx,n;
     double a, x[5] = {2.0,2.0,2.0,2.0,2.0};

     /* Scale the vector x by 3.0 */
     n = 5; a = 3.0; incx = 1;
     dscal_(&n, &a, x, &incx);

     return 0;
}
```

CBLAS – BLAS in C

```c
#include <stdio.h>
#if defined(__MACH__) && defined(__APPLE__)
#include <Accelerate/Accelerate.h>
#else
#include <cblas.h>
#endif

int main(void) {

    int i,incx,n;
    double a, x[5] = {2.0,2.0,2.0,2.0,2.0};

    /* Scale the vector x by 3.0 */
    n = 5; a = 3.0; incx = 1;
    cblas_dscal(n, a, x, incx);

    return 0;
}
```

BLAS or CBLAS – what to use?

- Calling (FORTRAN)-BLAS from C/C++ can be cumbersome
  - add a trailing "_" to routine name
  - all arguments have to be passed by address
- CBLAS is more convenient
  - just add a "cblas_" prefix to the routine name
  - all arguments have their natural type
  - there might be extra arguments, though
  - many CBLAS implementations call BLAS "under the hood"

BLAS or CBLAS – what to use?

- Some libraries implement a C interface with the original BLAS names – but C-style arguments
  - Intel MKL
  - Oracle Studio Performance Library
  - ...
- They might provide a CBLAS interface as well

Calling BLAS/CBLAS: some hints

Important things to have in mind:

- memory for matrices and vectors is expected to be contiguous, i.e. one large block, no holes – important when allocating memory
- check the access order of matrices, i.e. row-wise or column-wise, and adapt the corresponding parameters
- look carefully at parameters like 'leading dimension', etc, especially for non-square matrices

Dynamic allocation of matrices in C

- Many libraries that can handle matrices, like BLAS, require, that the memory is contiguous, i.e. allocated in one large block.
- On the next slides, you can find an implementation that does exactly that.

Allocating a matrix in C

```
// allocate a double-prec m x n matrix
double **
dmalloc_2d(int m, int n) {
    if (m <= 0 || n <= 0) return NULL;
    double **A = malloc(m * sizeof(double *));
    if (A == NULL) return NULL;
    A[0] = malloc(m*n*sizeof(double));
    if (A[0] == NULL) {
        free(A);
        return NULL;
    }
    for (i = 1; i < m; i++)
        A[i] = A[0] + i * n;
    return A;
}
```

De-allocating a matrix in C

```
// de-allocting memory, allocated with
// dmalloc_2d

void
dfree_2d(double **A) {
    free(A[0]);
    free(A);
}
```