STA 141C - Big Data & High Performance Statistical Computing **Spring 2022**

Week 3-1: High performance matrix communications

Lecturer: Bo Y.-C. Ning *April 12, 2022*

Announcement

- Lab this week, Big-O notation and time complexity - two-sum problem and sorting

Last time

- Flop counts

Today

- High-performance matrix commutations
- Effects of data layout: column-major and row-major
- Gaussian elimination
- LU decomposition

1 High-performance matrix commutations

For high-performance matrix commutations, it is not enough to just minimize flops. Programming languages, pipelining, effective use of memory hierarchy, data layout in memory,..., all play important roles.

1.1 Compiled and interpreted languages

Compiled languages are directly compiled to machine code that is executed by CPU. Examples include C/C++, Fortran, and Julia. Pros: fast and take less memory; cons: relatively longer development time, hard to debug (except for Julia).

Interpreted languages are programming languages whose implementations execute instructions directly and freely. They are fast for prototyping but can be excruciatingly slow for loops (we will talk about this later). Examples include R, Matlab, SAS...

Python is a mixed (complied and then interpreted by virtual machine) language. Pros: extremely convenient for data preprocessing and manipulation and need relatively short development time. Cons: not as fast as compiled language (e.g., dealing with loops).

Thus for python and R users, avoid using loops as much as you can. In R, use lapply() instead of for loop; in python, use compression and generators. If the loop is unavoidable (e.g., writing MCMC code), consider write it in C/C++ or Fortran, and call it from R/Python.

Some suggestions are: 1) to be versatile in dealing with big data, master at least one language in each category and take advantage of the resource online; 2) Don't reinvent wheels. Make good use of libraries BLAS, LAPACK/LINPACK, SciPy, NumPy, ...

1.2 Vector computer and pipelining technology

Most modern computers are vector machines, which perform vector calculations fast (e.g., saxpy, inner product). One implication of the pipelining technology is that we need to ship vectors to the pipeline fast enough to keep the arithmetic units (ALU) busy and maintain high throughput. Taking vector addition $z = x + y$ as an example:

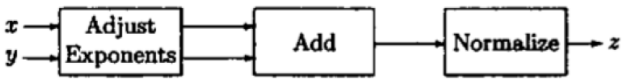


FIG. 1.4.1 *A 3-Cycle Adder*

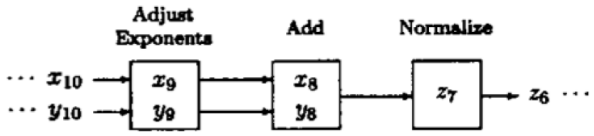


FIG. 1.4.2 *Pipelined Addition*

1.3 Memory hierarchy

In computer architecture, the *memory hierarchy* separates computer storage into a hierarchy based on response time. Upper the hierarchy, faster the memory accessing speed, and more expensive the memory units. Knowing the memory hierarchy is important. A key to high performance is effectively use of memory hierarchy. We need to use high-level BLAS as much as possible to keep the super fast arithmetic units busy with enough deliveries of matrix data and ship the results to memory fast enough to avoid backlog.

BLAS level 1 tend to be memory bandwidth-limited. Higher level BLAS (Level 2 or Level 3) make more effective use of ALUs (arithmetic logic units) to keep them busy. More about ALU, see wikipedia.

In addition to BLAS, LINPACK (**Lin**ear Equations Software **Pack**age) and LAPACK (**L**inear **A**lgebra **Pack**age) are both software packages that provide routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems

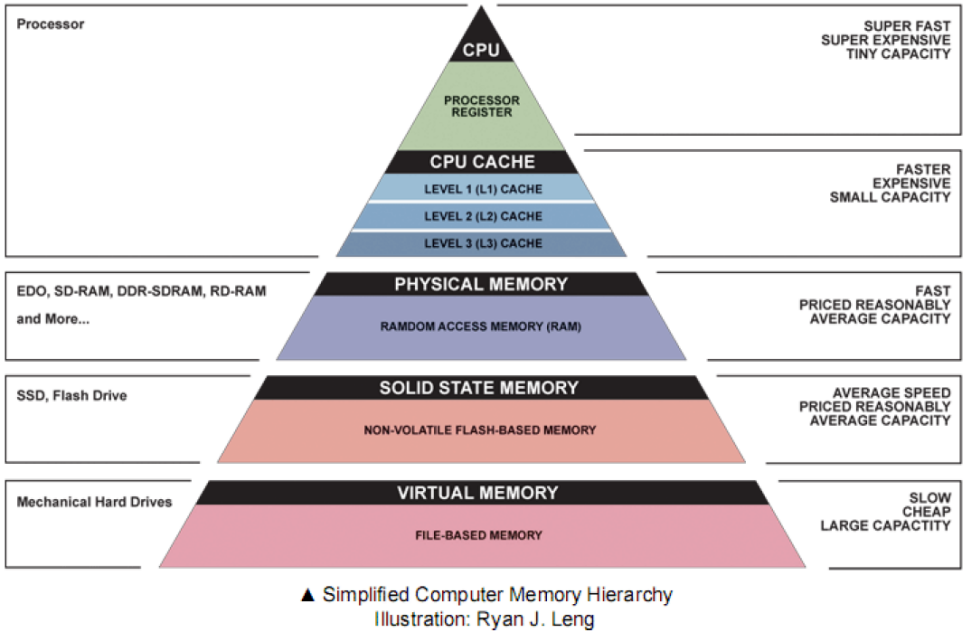


Figure 4.1: Computer memory hierarchy

Table 4.1: High-level BLAS

BLAS	Dimension	Mem ReFs	flops	ratio
Lv1: $y \leftarrow y + ax$	$x, y \in \mathbb{R}^n$	$3n$	n	$3:1$
Lv2: $y \leftarrow y + Ax$	$x, y \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$	n^2	n^2	$1:1$
Lv3: $C \leftarrow C + AB$	$A, B, C \in \mathbb{R}^{n \times n}$	$4n^2$	n^3	$4:n$

(ReFs: Resilient File System)

A distinction between LAPACK and LINPACK is that LAPACK makes use of higher level BLAS as much as possible (usually by smart partitioning) to increase the so-called level-3 fraction. numpy and R automatically uses LAPACK and BLAS libraries. HW 1 asks you to check the BLAS and LAPACK libraries in your computer. One can also install OpenBLAS library, which is an optimized BLAS library in your local machine (see `https://www.openblas.net/`).

2 Effects of data layout

Data layout also has an effect on computation speed. It is much faster to move chunks of data in memory than retrieving/writing scattered data.

There are two types of storage model:

1) *Column-major*, including Fortran, Matlab, and R;

2) *Row-major*, including C/C++, Python.

2.1 Stride

The time it takes to load a vector into a vector register may depend greatly on how the vector is laid out in memory. A vector is said to have *unit stride* if its components are continuous in memory. A matrix is said to be stored in *row-major order* if its rows have unit stride. If column-major order, then its columns have unit stride.

To understand the difference between column-major and row-major, take matrix multiplication as an example. One wishes to compute $C \leftarrow C + AB$, where $A \in \mathbb{R}^{m\times p}, B \in \mathbb{R}^{p\times m}, C \in \mathbb{R}^{m\times n}$. There are six variants of the algorithms according to the order in the triple loops:

jki or kji:
```
for i = 1:m
    C(i,j) = C(i,j) + A(i,k)B(k,j)
end
```

ikj or kij:
```
for j = 1:n
    C(i,j) = C(i,j) + A(i,k)B(k,j)
end
```

ijk or jik:
```
for k = 1:p
    C(i,j) = C(i,j) + A(i,k)B(k,j)
end
```

Pay attention to the innermost loop, where the vector calculation occurs, consider the associated stride when accessing the three matrices in memory (assuming column-major storage).

Suppose the loading of a unit-stride vector proceeds faster than non-unit, clearly, jki or kji is preferred.

Variant	A Stride	B Stride	C Stride
jki or kji	Unit	0	Unit
ikj or kij	0	Non-unit	Non-unit
ijk or jik	Non-unit	Unit	0

Now, what if row-major storage?

To summarize:

- Be aware of flop counts
- Be careful when computing A^{-1} (We will discuss more on this in the next few lectures)
- For python and R users, avoid using loop if possible. In R, use $\mathsf{lapply}(), \mathsf{sapply}(), ...$; in python, use compression and generators. If you must use loops (e.g., writing MCMC), consider write it in C/C++, and call it from R/Python.
- Notice the different between row-major and column-major.

3 Back to solving linear equations

Many statistical methods eventually require to solve linear equations such as

$$Ax = b,\ A \in \mathbb{R}^{m\times m}, b \in \mathbb{R}^{m\times 1}$$

How do you get x in R and in Python? How about we calculate A^{-1}, then do $A^{-1} \times b$ (of course, we assume A is invertible)? Do you have other suggestions?

Indeed, the problem is relatively easy when m is small but needs to be careful when m is large.

3.1 Triangular system

We start with A is a triangular matrix, if $A = L$ is a lower triangular matrix, we solve $Lx = b$ using *forward substitution*; if $A = U$ is an upper triangular matrix, we solve $UX = b$ using *backward substitution*. Eventually, we will show that solving a dense matrix A can be done using forward substitution and backward substitution.

3.1.1 Forward substitution

Forward substitution to solve $Lx = b$, where $L \in \mathbb{R}^{m \times m}$ is a lower triangular matrix:

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$\begin{aligned} x_1 &= b_1/a_{11} \\ x_2 &= (b_2 - a_{21}x_1)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \\ &\vdots \\ x_m &= b_m - a_{m1}x_1 - a_{m2}x_2 - \dots - a_{m,m-1}x_{m-1})/a_{mm} \end{aligned}$$

Figure 4.2: Forward substitution

How many flops does it take?

$\rightarrow m^2$ flops (m divisions, $O(m^2/2)$ additions, $O(m^2/2)$ multiplications).

3.1.2 Backward substitution

Backward substitution to solve $Ux = b$, where U is a upper triangular matrix.

How many flops does it take?

$\rightarrow m^2$ flops (m divisions, $O(m^2/2)$ additions, $O(m^2/2)$ multiplications)

Note that L and U are both accessed by row, what about column-major language like R?

Forward and backward substitution in software:

- BLAS Level 2 function: dtrsv (triangular solve with one right hand side)

$$\begin{bmatrix} a_{11} & \dots & a_{1,m-1} & a_{1m} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & a_{m-1,m-1} & a_{m-1,m} \\ 0 & \dots & 0 & a_{mm} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{m-1} \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_{m-1} \\ b_m \end{bmatrix}$$

$$\begin{aligned} x_m &= b_m/a_{mm} \\ x_{m-1} &= (b_{m-1} - a_{m-1,m}x_m)/a_{m-1,m-1} \\ x_{m-2} &= (b_{m-2} - a_{m-2,m-1}x_{m-1} - a_{m-2,m}x_m)/a_{m-2,m-2} \\ &\vdots \\ x_1 &= b_1 - a_{12}x_2 - a_{13}x_3 - \dots - a_{1m}x_m)/a_{11} \end{aligned}$$

Figure 4.3: Backward substitution

- BLAS Level 3 function: dtrsm (matrix triangular solve, i.e., multiple right hand sides)
- In R, use forwardsolve() and backsolve() (wrappers of dtrsm).
- In python, use scipy.linalg.solve_triangular() (wrapper of trtrs, dtrsm) (see [Link])

3.2 Gaussian elimination and LU decomposition

Recall that we want to solve the linear equation

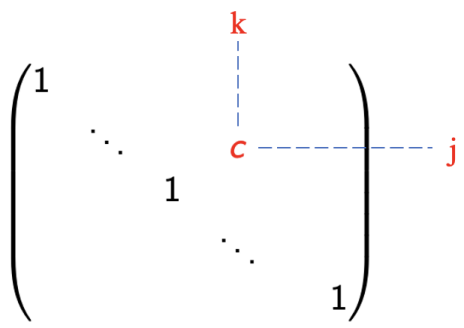
$$Ax = b,$$

where A is a dense matrix (not necessarily symmetric), $A \in \mathbb{R}^{n\times n}$, $x, b \in \mathbb{R}^n$.

The idea is to use a series of elementary operations called *Gaussian elimination* (proposed by Carl Friedrich Gauss in 1800s) to turn A into a triangular system and then apply forward and backward substitutions to solve x.

3.2.1 Gaussian elimination

Introducing the elementary operator matrix $E_{jk}(c)$, an identity matrix with 0 in the position (j, k) replaced by c.



Mathematically, for any **vector** $x = (x_1, \ldots, x_n)'$, we denote

$$E_{jk}(c)x = (x_1, \ldots, x_{j-1}, x_j + cx_k, x_{j+1}, \ldots, x_n)'$$

We apply $E_{jk}(c)$ on both sides of the system $Ax = b$. Then the j-th equation

$$a'_{j\cdot}x = b_j$$

is replaced by

$$a'_{j\cdot}x + ca'_{k\cdot}x = b_j + cb_k.$$

The value of c depends on j and k, for the first column shown below, $c_j = -a_{j1}/a_{11}$.

Given a system of linear algebraic equations

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Step 1: Each row times a_{11}/a_{k1},
then use row one to subtract other rows.

$$\Rightarrow \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & \tilde{a}_{22} & \cdots & \tilde{a}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \tilde{a}_{n2} & \cdots & \tilde{a}_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \tilde{b}_2 \\ \vdots \\ \tilde{b}_n \end{bmatrix}$$

Step 2: The second row and down multiply by $\tilde{a}_{22}/\tilde{a}_{k2}$,
then use row two to subtract every row below.

$$\Rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} & \cdots & \tilde{a}_{2n} \\ 0 & 0 & \tilde{a}_{33} & \cdots & \tilde{a}_{3n} \\ \vdots & \vdots & \cdots & \ddots & \vdots \\ 0 & 0 & \tilde{a}_{n3} & \cdots & \tilde{a}_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \vdots \\ \tilde{b}_n \end{bmatrix}$$

Step 3: Similar to the previous two steps, repeat until all
elements in the lower triangle of the matrix A
become zeros.

$$\Rightarrow \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & \tilde{a}_{22} & \cdots & \tilde{a}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ & & & \vdots \\ 0 & 0 & \cdots & \tilde{a}_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \tilde{b}_2 \\ \vdots \\ \vdots \\ \tilde{b}_n \end{bmatrix}$$

3.2.2 Mathematical representations for GE

First, zeroing the first column

$$\begin{aligned} E_{21}(c_2^{(1)})Ax &= E_{21}(c_2^{(1)})b \\ E_{31}(c_3^{(1)})E_{21}(c_2^{(1)})Ax &= E_{31}(c_3^{(1)})E_{21}(c_2^{(1)})b \\ &\vdots \\ E_{n1}(c_n^{(1)})\cdots E_{31}(c_3^{(1)})E_{21}(c_2^{(1)})Ax &= E_{n1}(c_n^{(1)})\cdots E_{31}(c_3^{(1)})E_{21}(c_2^{(1)})b, \end{aligned}$$

where $c_j^{(1)} = -a_{j1}/a_{11}$. Denote

$$M_1 = E_{n1}(c_n^{(1)})\cdots E_{31}(c_3^{(1)})E_{21}(c_2^{(1)}).$$

Note that for $j > k$, $E_{jk}(c) = I + ce_je_k'$ is unit lower triangular and full rank and $E_{jk}^{-1}(c) = E_{jk}(-c)$, hence M_1 is a lower triangular matrix (homework).

We then apply the similar strategy to zero the k-th column for $k = 2, \ldots, n-1$ sequentially. Finally, we obtain $Ux = \tilde{b}$, where

$$\begin{aligned} U &= M_{n-1}\ldots M_1 A, \\ \tilde{b} &= M_{n-1}\ldots M_1 b, \end{aligned}$$

and each M_k has the shape

$$M_k = E_{n,k}(c_n^{(k)})\ldots E_{k+1,k}(c_{k+1}^{(k)}) = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & c_{k+1}^{(k)} & 1 & \\ & & \vdots & & \ddots \\ & & c_n^{(k)} & & 1 \end{pmatrix},$$

where $c_i^{(k)} = -\tilde{a}_{ik}^{(k-1)}/\tilde{a}_{kk}^{(k-1)}$. Note that U an upper triangular matrix and M_k is unit lower triangular and full rank. The matrices M_ks are called the *Gauss transformations*.

3.2.3 LU decomposition

Let $L = M_1^{-1}\ldots M_{n-1}^{-1}$, we have the decomposition

$$A = LU,$$

where M_k is lower triangular, so does M_k^{-1} and thus L is a lower triangular matrix and U is an upper triangular matrix.

Furthermore, by the Sherman-Morrison formula (homework) if

$$M_k = I + (0, \ldots, 0, c_{k+1}^{(k)}, \ldots, c_n^{(k)})'e_k',$$

then

$$M_k^{-1} = I - (0, \ldots, 0, c_{k+1}^{(k)}, \ldots, c_n^{(k)})'e_k'.$$

So the entries of L are simply $l_{jk} = -c_j^{(k)}$, $j > k$.

3.2.4 Flop counts for solving the linear system $Ax = b$

Now we can calculate the flop counts for solving the linear system $Ax = b$ ($A \in \mathbb{R}^{n\times n}$):

Step 1: The LU decomposition

- Each multiplier is computed with one division, the total cost is $\sum_{i=1}^{n} i = (n-1)n/2$
- At stage k, we need to modify a $(n-i)\times(n-i)$ matrix, each entry is modified by one subtraction and one multiplication, the total cost of the row operations is $2\sum_{i=1}^{n-1} i^2 = 2n^3/3 - n^2 + n/3$
- Total cost for LU is $\approx 2n^3/3$

Step 2: Given LU, forward substitution and backward substitution costs $2n^2$ flops

So, the total flops for solving linear equation $Ax = b$ costs $2n^3/3 + O(n^2)$ flops.

A few comments:

- LU decomposition exists if the principal sub-matrix $A[1:k, 1:k]$ is non-singular for $k = 1, \ldots, n-1$.
- If the LU decomposition exists and A is non-singular, then the LU decomposition is unique and $det(A) = \prod_{i=1}^{n} u_{ii}$.
- For non-square matrix (rectangular matrix) $A \in \mathbb{R}^{m\times n}$. LU decomposition exists if $A[1:k, 1:k]$ is nonsingular for $k = \min\{m, n\}$. Then we can write it as

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 3 & 1 \\ 5 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 0 & -2 \end{pmatrix}$$

  and

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 4 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \end{pmatrix}$$

  Then, one can slightly modify the original algorithm.
- In numpy: `scipy.linalg.lu_factor()` and `scipy.linalg.lu_solve()` is equivalent to numpy's `numpy.linalg.solve()`
- In R, LU is used in `solve()` function

In conclusion, avoid computing matrix inverse unless: 1) it is absolutely necessary to compute (e.g., obtain inverse of the covariance matrix); 2) n is small.