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

Lecture 3: Flop counts and high-performance matrix commutations

Lecturer: Bo Y.-C. Ning *January 13, 2022*

Disclaimer: *My notes may contain errors, distribution outside this class is allowed only with the permission of the Instructor.*

Announcement

- Think about group members?
- Google survey
- Homework due Jan 26
- Final report due March 17 @ 11:59 pm
- Discussion sessions?

Last time

- Computer storage
- Fixed-point and floating-point number systems
- Catastrophic cancellation

Today

- Flop counts and big-O notation
- BLAS and LAPACK
- High-performance matrix commutations
- (maybe) Gaussian elimination and LU decomposition

1 Flops and Big-O notations

Algorithm is loosely defined as a set of instructions for doing something (Input $\rightarrow$ Output). Basic unit for measuring efficiency of an algorithm is flops (floating point operation). A *flop* serves as a basic unit of computation, which consists of a floating point addition ($+$), subtraction ($-$), multiplication ($\times$), and division ($/$). Some books also consider $\sqrt{}$ and $>$ ($<$) as a flop, although, in general, addition is faster than $\sqrt{}$, in practice, since we only interested in rough (not exact) flop counts, their difference is minor.

How to (roughly) measure efficiency of an algorithm? We use Big-O notation. If n is the size of a problem, we say a function $f(n) = O(g(n))$ if $f(n) \leq ag(n)$, a is a constant. This means that $f(n)$ grows at most in the order of $g(n)$. For example,

- $2n^2 - n/2 - 1 = O(n^2)$
- $n = O(n)$, $n = O(n^2)$, $n = O(n \log n)$
- $n^2 \neq O(n \log n)$

More examples:

1. The vector-vector operations: for $a, b \in \mathbb{R}^n$, c is a scalar
   - $a + b$: n flops, $O(n)$
   - $c * a$: n flops, $O(n)$
   - $a'b$: n multiplications and $n - 1$ additions, total $2n - 1$ flops, $O(n)$
   - what about ab' (outer product)?
2. Matrix-vector multiplication: Ab, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^n$: mn multiplications and $m(n-1)$ additions, total $2mn - m$ flops, $O(mn)$
3. Matrix-matrix multiplication: AB, $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$: mnp multiplications and $mp(n-1)$ additions, total $2mnp - mp$ flops, $O(mnp)$

Read more examples: `https://mediatum.ub.tum.de/doc/625604/625604`

Big-O notation is not the only one used to measure efficiency of an algorithm, there are other notations, such as Θ and Ω. We will introduce them during next week's discussion session.

Knowing how to count flops is important. Compare flops of the two expressions:

$$G \times X \times y \qquad \text{and} \qquad G \times (X \times y)$$

for $G \in \mathbb{R}^{p \times p}$, $X \in \mathbb{R}^{p \times n}$, and $y \in \mathbb{R}^n$. We will get familiar with the flop counts for common numerical tasks in statistics.

2 BLAS, LINPACK and LAPACK

2.1 BLAS

BLAS stands for basic linear algebra subroutines (see [this] for a complete listing of BLAS functions and [this] explains why it matters). BLAS is the low-level part of the system that is responsible for efficiently performing numerical linear algebra.

```
# find out my BLAS library in numpy
import numpy as np
np.show_config()

blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/Users/yuchien/opt/anaconda3/lib']
```

```
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/Users/yuchien/opt/anaconda3/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/Users/yuchien/opt/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/Users/yuchien/opt/anaconda3/include']
lapack_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/Users/yuchien/opt/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/Users/yuchien/opt/anaconda3/include']
lapack_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/Users/yuchien/opt/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/Users/yuchien/opt/anaconda3/include']
```

R default uses BLAS, see [tutorial] to install optimized BLAS/LAPACK libraries.

Table 1.1: Flop counts for BLAS functions

Level	Operation	Name	Dimension	Flops
1	$\alpha \leftarrow x'y$	dot product	$x, y \in \mathbb{R}^n$	$O(n)$
	$y \leftarrow y + ax$	saxpy	$a \in \mathbb{R}, x, y \in \mathbb{R}^n$	$O(n)$
2	$y \rightarrow y + Ax$	gaxpy	$A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$	$O(mn)$
	$A \leftarrow A + yx'$	rank one update	$A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$	$O(mn)$
3	$C \leftarrow C + AB$	matrix multiplication	$A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times p}$, $C \in \mathbb{R}^{m\times p}$	

Note: saxpy is short for *scalar a x plus y*; gaxpy is short for *generalized saxpy*. Generally speaking, Level 1 BLAS subroutine does vector-vector operations; Level 2 BLAS subroutine does matrix-vector operations; and Level 3 BLAS subroutine does matrix-matrix operations.

Some operations appear as level-3 but indeed are level-2, for example,

- Column scaling: $A \leftarrow AD$, $A \in \mathbb{R}^{m\times n}$, $D = \text{diag}(d_1, \ldots, d_n)$ is $O(mn)$
- Row scaling: $A \leftarrow DA$, $A \in \mathbb{R}^{m\times n}$, $D = \text{diag}(d_1, \ldots, d_m)$ is $O(mn)$

Matrix inversion is more complicate, we will learn that $E \leftarrow E^{-1}$, $E \in \mathbb{R}^{n\times n}$ takes $O(n^3)$.

Example: Computing $X'W^{-1}X$ (e.g., the weighted least square), where $X \in \mathbb{R}^{n\times p}$ and $W = \text{diag}(w_1, \ldots, w_n) \in \mathbb{R}^{n\times n}$:

1. X' %∗% solve(W) %∗% X (or np.linalg.solve(W)): takes $O(n^3 + n^2p + np^2)$ flops, matrix inversion is expensive! (transpose requires no flops)
2. X' %∗% diag$(1/w)$ %∗% X (or np.diag$(1/w)$): takes $O(n^2p + np^2)$ flops, why? Can we improve?
3. X' %∗% (X/w): takes $O(np^2 + np) = O(np^2)$ flops. In numpy, do X.T ⋆ (X/w) and in R, do crossprod(X, X/w). This is fine for R, not for numpy. Why?

   More about the crossprod (and tcrossprod) function: [see here]

Bottom line: Always be aware flop counts when writing code!

```r
# clean up workspace
rm( list=ls() )

# simple linear regression design matrix X
n <- 100          # dim of matrix
X <- cbind( rep(1,n), (1:n) )
X

# weight vector w
w <- sqrt( (1:n) )           # just so they're different
w

# weight matrix
W <- diag(w)
# now calculate (X' * inv(W) * X) in four ways:
#   first: Correct, but slow, W takes lots of space
ptw1 <- proc.time()
W <- diag(w)
xwx1 <- t(X) %*% solve(W) %*% X
xwx1
proc.time() - ptw1

#   second: Correct, less slow, takes lots of space
ptw2 <- proc.time()
xwx2 <- t(X) %*% diag(1/w) %*% X
xwx2
proc.time() - ptw2

#   third: Wrong, but faster -- recycles w wrong
ptw3 <- proc.time()
xwx3 <- ( t(X) / w ) %*% X           # *****wrong*****
xwx3
proc.time() - ptw3

#   fourth: Correct, but looks wrong
ptw4 <- proc.time()
xwx4 <- t(X) %*% (X/w)               # uses recycling correctly, fast
xwx4
proc.time() - ptw4

# fourth: looks different but same execution
ptw5 <- proc.time()
crossprod(X,X/w)                     # correct, fast
proc.time() - ptw5

# done
rm( list=ls() )

# In numpy, use
import time
import numpy as np
t = time.time()

print np.round_(time.time() - t, 3), 'sec elapsed'
```

3 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.

3.1 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, ...

3.2 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:

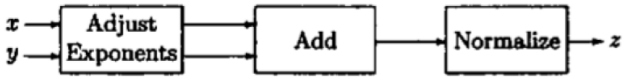

FIG. 1.4.1 *A 3-Cycle Adder*

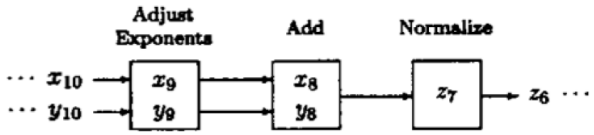

FIG. 1.4.2 *Pipelined Addition*

3.3 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.

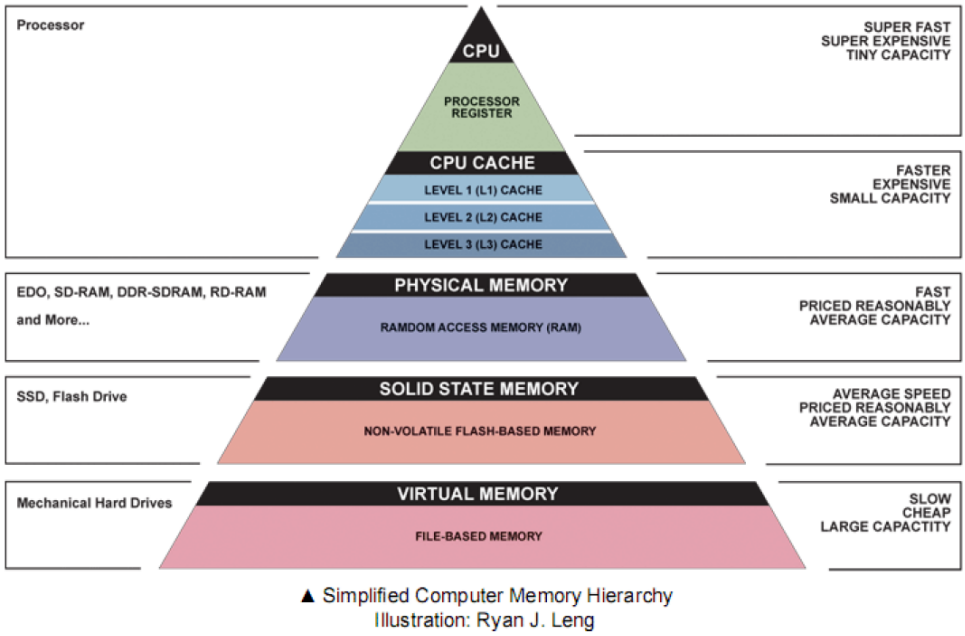

Figure 1.1: Computer memory hierarchy

Table 1.2: 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)

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 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/`).