ECS231

Intro to High Performance Computing

April 13, 2019

Algorithm design and complexity - as we know

Example. Computing the nth Fibonacci number:

$$F(n) = F(n-1) + F(n-2), \quad \text{for} \quad n = 2, 3, \ldots$$
$$F(0) = 0, \quad F(1) = 1$$

Algorithms and complexity:

1. Recursive
2. Iterative
3. Divide-and-conquer
4. Approximation

Algorithms design and communication

Examples:

- Matrix-vector multiplication $y \leftarrow y + A \cdot x$
  1. Row-oriented
  2. Column-oriented
- Solving triangular linear system $Tx = b$
  1. Row-oriented
  2. Column-oriented

Matrix storage

- A matrix is a 2-D array of elements, but memory addresses are "1-D".
- Conventions for matrix layout
  - by column, or "column major" – Fortran default
  - by row, or "row major" – C default

Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

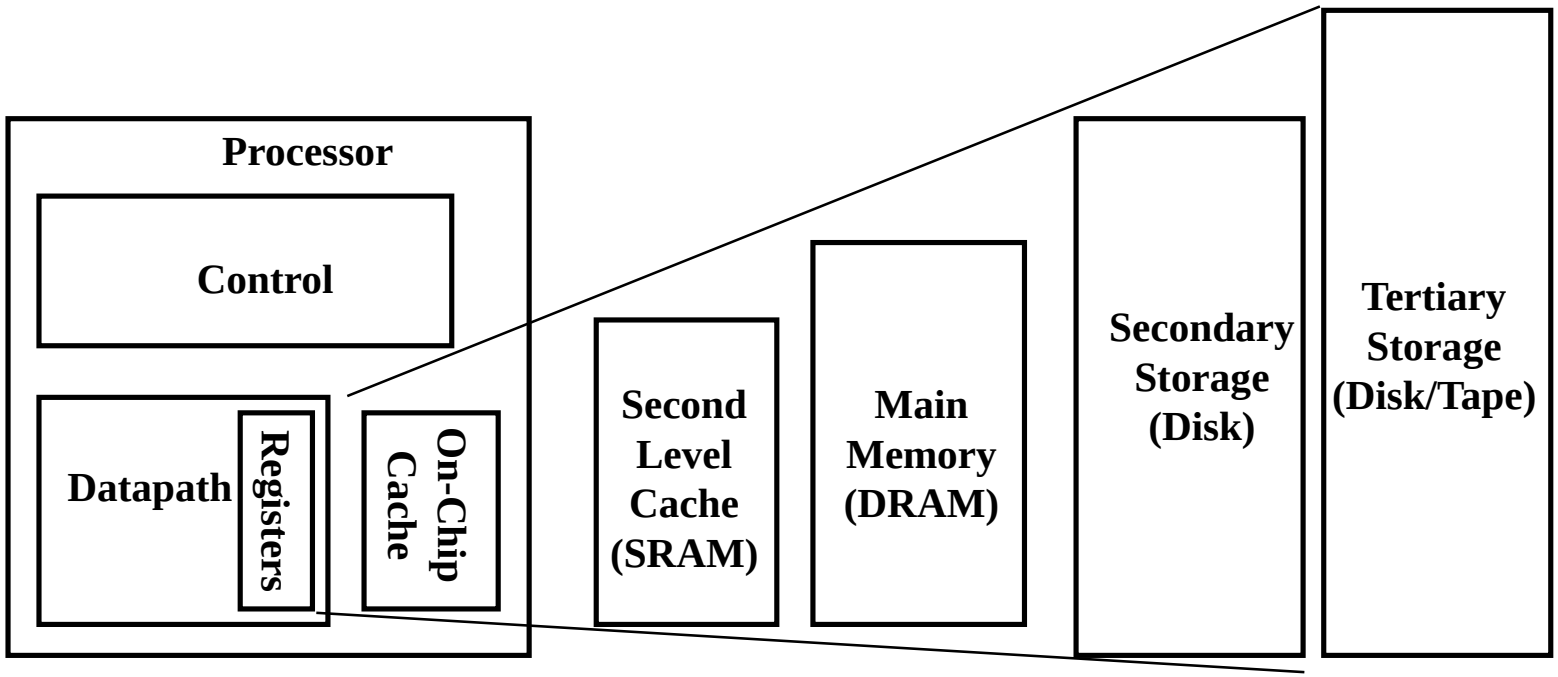
Row major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Memory hierarchy

- Most programs have a high degree of locality in their memory accesses:
  - spatial locality
    accessing things nearby previous accesses
  - temporal locality
    reusing an item that was previously accessed
- Memory hierarchy tries to exploit locality
- By taking advantage of the principle of locality:
  - present the user with as much memory as is available in the cheapest technology
  - provide access at the speed offered by the fastest technology

Memory hierarchy



Speed (ns):	1s	10s	100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks	Ms	Gs	Ts

Idealized processor model

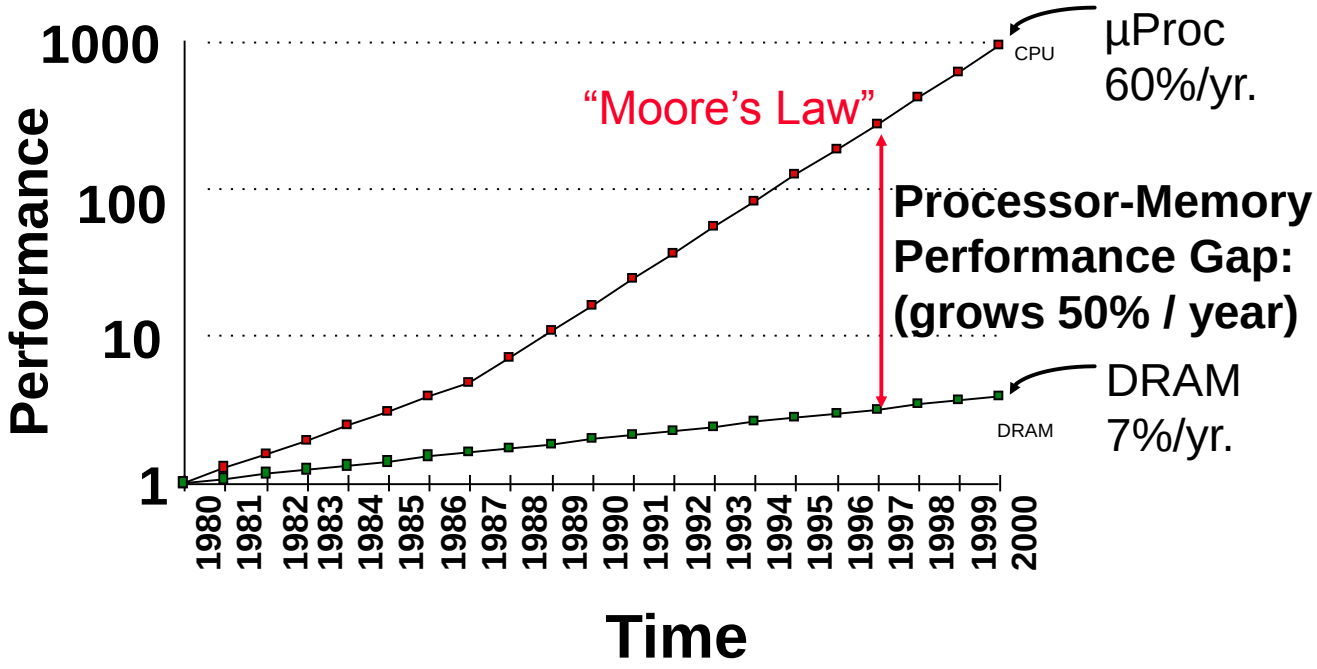
- Processor names bytes, words, etc. in its address space
  - these represent integers, floats, pointers, arrays, etc
  - exist in the program stack, static region, or heap
- Operations include
  - read and write (given an address/pointer)
  - arithmetic and other logical operations
- Order specified by program
  - read returns the most recently written data
  - compiler and architecture translate high level expressions into "obvious" lower level instructions
  - Hardware executes instructions in order specified by compiler
- Cost
  - Each operations has roughly the same cost (read, write, add, multiply, etc.)

Processor in the real world

- Processors have
  - registers and caches
    - small amounts of fast memory
    - store values of recently used or nearby data
    - different memory ops can have very different costs
  - parallelism
    - multiple "functional units" that can run in parallel
    - different orders, instruction mixes have different costs
  - pipelining
    - a form of parallelism, like an assembly line in a factory
- Why is this your program?
  - In theory, compilers understand all of this and can optimize your program, in practice, they don't.

Processor-DRAM gap (latency)

Memory hierarchies are getting deeper, processors get faster more quickly than memory access.



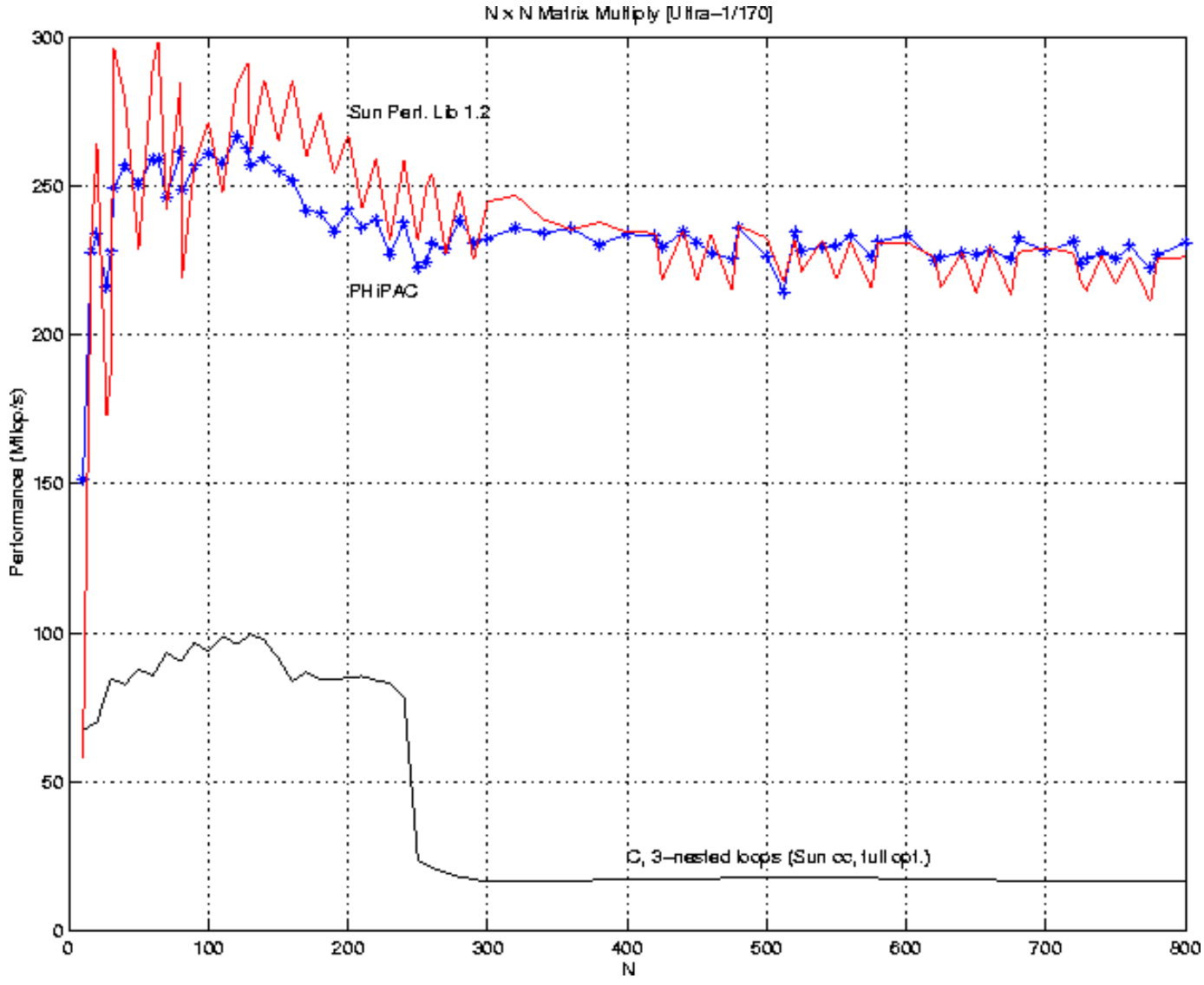
Communication is the bottleneck!

Communication bottleneck

- Time to run code = clock cycles running code + clock cycles waiting for memory
- For many years, CPU's have sped up an average of 50% per year over memory chip speed ups.
- Hence, memory access is the computing bottleneck. The communication cost of an algorithm has already exceed arithmetic cost by orders of magnitude, and the gap is growing.

Example: matrix-matrix multiply

Otimized vs. naïve triple-loop algorithms for matrix multiply



Cache and its importance in performance

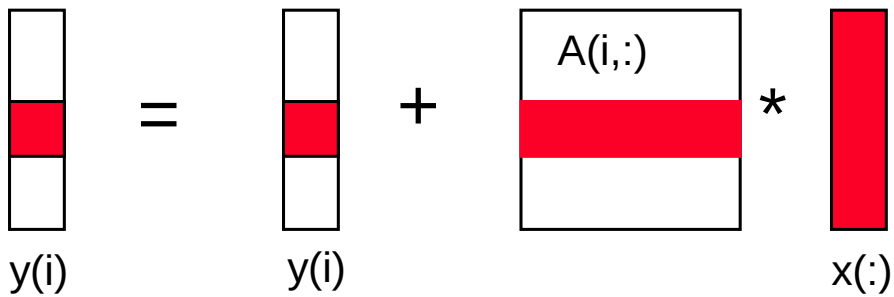
- Data cache was designed with two key concepts in mind
  - Spatial locality
    - when an element is referenced, its neighbors will be referenced too,
    - cache lines are fetched together,
    - work on consecutive data elements in the same cache line.
  - Temporal locality
    - when an element is referenced, it might be referenced again soon,
    - arrange code so that data in cache is reused often.
- Actual performance of a program can be complicated function of the architecture. We will use a simple model to help us design efficient algorithm.
- Is this possible? we will illustrate with a common technique for improving catch performance, called blocking or tiling.

A simple model of memory

- Assume just 2 levels in the hierachy: fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation
  - $\mathbf{q} = f/m$ average number of flops per slow element access
- Minimum possible time $= f \cdot t_f$ when all data in fast
- Total time $= f \cdot t_f + m \cdot t_m$
  $= f \cdot t_f(1 + t_m/t_f \cdot 1/\mathbf{q})$
- Larger $\mathbf{q}$ means "Total time" closer to minimum $f \cdot t_f$
- t_m/t_f = key to machine efficiency
- $\mathbf{q}$ = key to algorithm efficiency

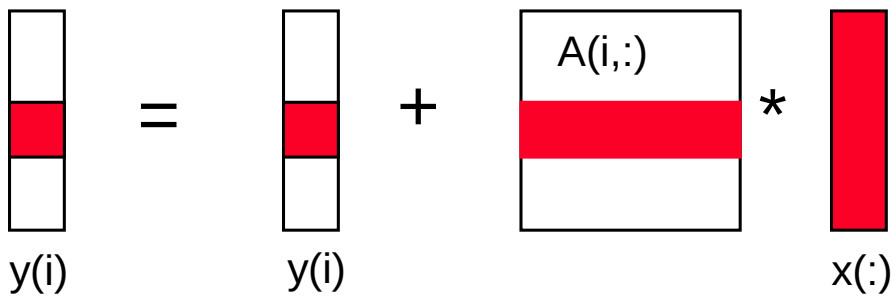
Matrix-vector multiply $y \leftarrow y + Ax$

```
for i = 1:n
    for j = 1:n
        y(i) = y(i) + A(i,j)*x(j)
```



Matrix-vector multiply $y \leftarrow y + Ax$

```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
    {read row i of A into fast memory}
    for j = 1:n
        y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```



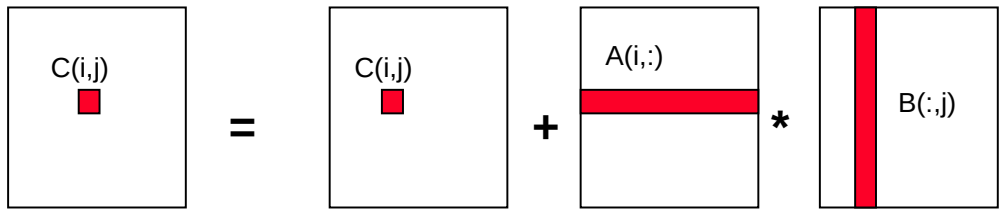
Matrix-vector multiply $y \leftarrow y + Ax$

```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
    {read row i of A into fast memory}
    for j = 1:n
        y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

- $m =$ number of slow memory refs $= 3n + n^2$
- $f =$ number of arithm ops $= 2n^2$
- $\mathbf{q} = f/m \approx 2$
- Matrix-vector multiplication limited by slow memory speed!

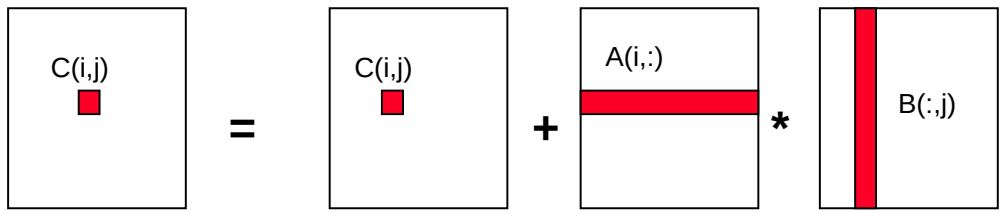
Naïve matrix-matrix multiply $C \leftarrow C + AB$

```
for i = 1:n
    for j = 1:n
        for k = 1:n
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
```



Naïve matrix-matrix multiply $C \leftarrow C + AB$

```
for i = 1:n
    {read row i of A into fast memory}
    for j = 1:n
        {read C(i,j) into fast memory}
        {read column j of B into fast memory}
        for k = 1:n
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
        {write C(i,j) back to slow memory}
```



Naïve matrix-matrix multiply $C \leftarrow C + AB$

```
for i = 1:n
    {read row i of A into fast memory}
    for j = 1:n
        {read C(i,j) into fast memory}
        {read column j of B into fast memory}
        for k = 1:n
            C(i,j) = C(i,j) + A(i,k)*B(k,j)
        {write C(i,j) back to slow memory}
```

Number of slow memory references:

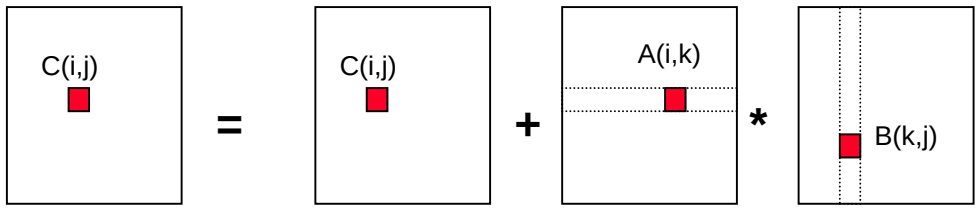
$$m = n^2 (\text{read each row of } A \text{ once})$$
$$+ n^3 (\text{read each column of } B \ n \text{ times})$$
$$+ 2n^2 (\text{read and write each element of } C \text{ once}) = n^3 + 2n^2$$

Therefore, $\mathbf{q} = f/m = 2n^3/(n^3 + 3n^2) \approx 2$. There is no improvement over matrix-vector multiply!

Block matrix-matrix multiply

Consider A, B, C to be $N \times N$ matrices of $b \times b$ subblocks, where $b = n/N$ is called the blocksize

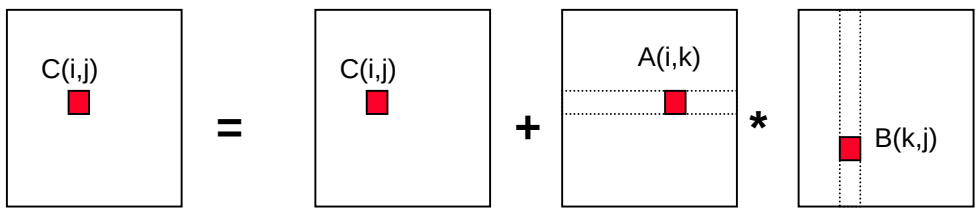
```
for i = 1:N
    for j = 1:N
        for k = 1:N
            C(i,j) = C(i,j) + A(i,k)*B(k,j) {on blocks}
```



Block matrix-matrix multiply

Consider A, B, C to be $N \times N$ matrices of $b \times b$ subblocks, where $b = n/N$ is called the blocksize

```
for i = 1:N
    for j = 1:N
        {read block C(i,j) into fast memory}
        for k = 1:N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
            C(i,j) = C(i,j) + A(i,k)*B(k,j) {on blocks}
        {read block C(i,j) back to slow memory}
```



Block matrix-matrix multiply

```
for i = 1:N
    for j = 1:N
        {read block C(i,j) into fast memory}
        for k = 1:N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
            C(i,j) = C(i,j) + A(i,k)*B(k,j) {on blocks}
        {read block C(i,j) back to slow memory}
```

Number of slow memory references:

$$m = N^3 \cdot n/N \cdot n/N (\text{read each block of } A\ N^3 \text{ times})$$
$$+ N^3 \cdot n/N \cdot n/N (\text{read each block of } B\ N^3 \text{ times})$$
$$+ 2n^2 (\text{read and write each block of } C \text{ once}) = (2N+2)n^2$$

and average number of flops per slow memory access

$$\mathbf{q} = \frac{f}{m} = \frac{2n^3}{(2N+2)n^2} \approx \frac{n}{N} = b.$$

Hence, we can improve performance by increasing the blocksize b!

Limits to optimizing matrix multiply

The blocked algorithm has the ratio $\mathbf{q} \approx b$:

- The larger the blocksize, the more efficient the blocked algorithm will be.
- Limit: all three blocks from A, B, C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large:

$$3b^2 \leq M \quad \Longrightarrow \quad \mathbf{q} \approx b \leq \sqrt{M/3}.$$

Fast linear algebra kernels: BLAS

- Simple linear algebra kernels such as matrix multiply
- More complicated algorithm can be built from these kernels
- The interface of these kernels havebeen standardized as the Basic Linear Algebra Subroutines (BLAS).

BLAS: advantages

- Clarity: code is shorter and easier to read
- Modularity: gives programmer larger building blocks
- Performance: manufacturers provide tuned machine-specific BLAS
- Portability: machine dependencies are confined to the BLAS

Level 1 BLAS

- Operate on vectors or pairs of vectors
  perform $O(n)$ operations
  return either a vector or a scalar
- `xAXPY`
  $y \leftarrow ax + y$
- `xSCAL`
  $y = ax$
- `xDOT`
  $s = x^T y$
- ...

Level 2 BLAS

- Operate on a matrix and a vector:
  perform $O(n^2)$ operations
  return a matrix or a vector
- `xGEMV`
  $y \leftarrow y + Ax$
- `xGER`
  $A \leftarrow A + yx^T$ rank-one update
- `xTRSV`
  Solves $Tx = b$ for x, where T is triangular
- ...

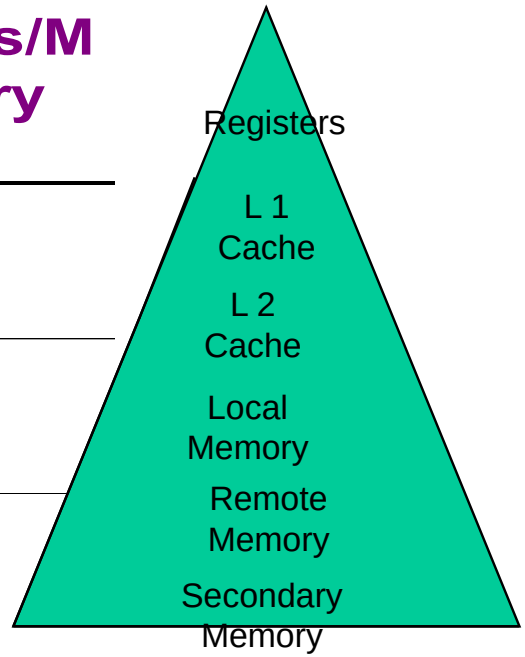
Level 3 BLAS

- Operate on a pair or triple of matrices
  perform $O(n^3)$ operations
  return a matrix
- `xGEMM`
  $C \leftarrow \alpha C + \beta AB$
- `xTRSM`
  solves $TX = B$ for X, where T is trianglar
- ...

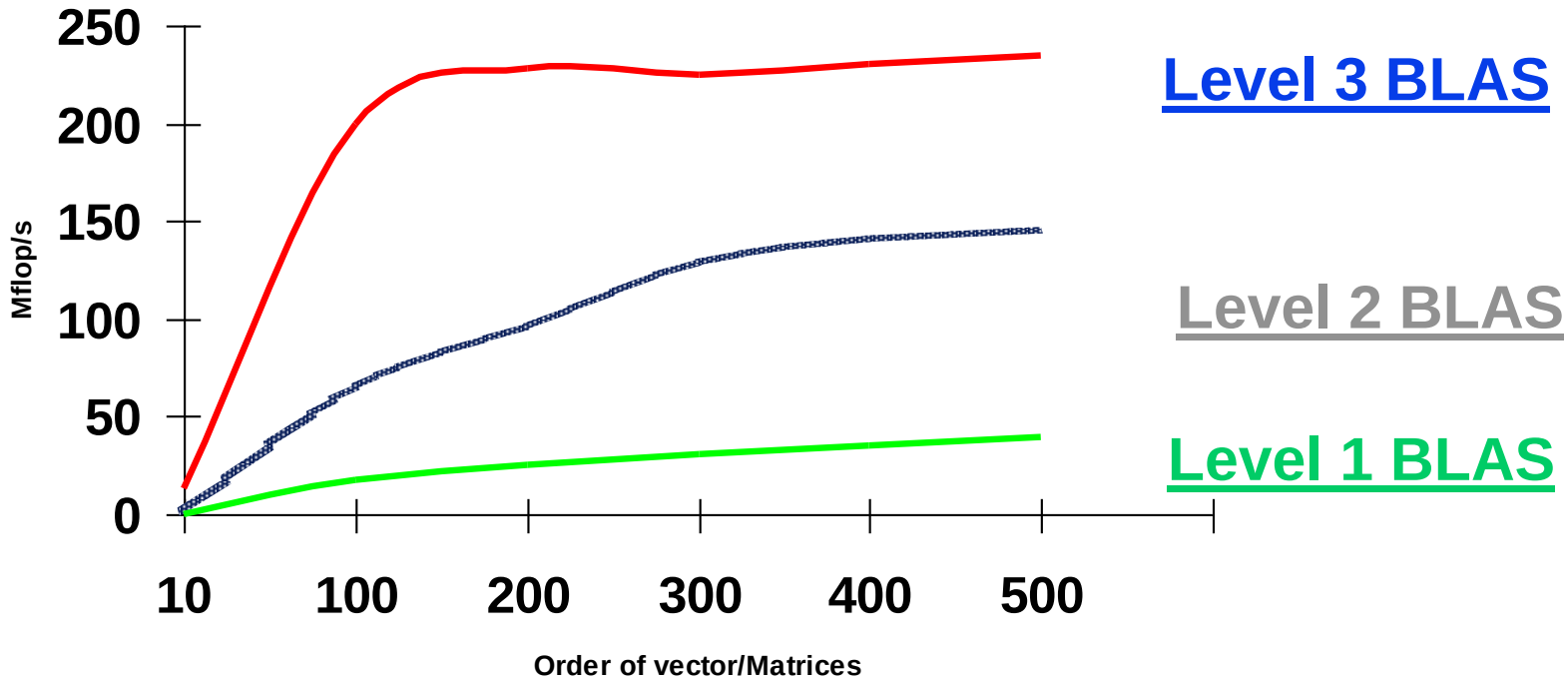
Why higher level BLAS?

- Can only do arithmetic on data at the top of hierarchy
- Higher level BLAS let us do this

BLAS	Memory Refs	Flops	Flops/Memory Refs
Level 1 $y=y+\alpha x$	$3n$	$2n$	$2/3$
Level 2 $y=y+Ax$	n^2	$2n^2$	2
Level 3 $C=C+AB$	$4n^2$	$2n^3$	$n/2$



Typical BLAS Performance



Further reading:

https://github.com/flame/how-to-optimize-gemm/wiki/

Mini project – Homework

Algorithms for the matrix multiply $C = C + A \cdot B$ with different BLAS-type operation kernels:

1. triple-loop
2. dot product (inner product), i.e., the inner loop use the vector inner product $x^T y$.
3. saxpy, i.e., the inner loop use Level 1 BLAS: $y := a * x + y$
4. matrix-vector, i.e., the inner loop use Level 2 BLAS: $y := y + A * x$
5. Outer product, i.e., the inner loop use Level 2 BLAS: $C := C + xy^T$.

What we learned here

1. The weakness of flop counting: methods for the same problem that involve the same number of flops can perform very differently.
2. The nature of the kernel operations (BLAS 1, 2, 3) is more important than the amount of arithmetic involved.

Numerical software engineering

- documentation
  an integral part of programming
- Software design
  modular design
- Validation and debugging
  write a program to validate a function that you have written
- Efficiency
  array (matrix)-level computing
  make use of BLAS, and high-performance libraries such as Intel's MKL

Further Reading

- Berkeley CS267 Lecture on "Single Processor Machines: Memory Hierarchies and Processor Features by J. Demmel