Plan for today

- Need for fast computers in engineering
- Barriers to improving serial processing
- Caches
- Arrays
- Single CPU simple performance model
- Matrix-vector multiply performance analysis
- Matrix-matrix multiply performance analysis

Need for fast computers in engineering

(Old) design process:

block diagram → physical prototytpe → experimments → analysis → modify prototype → ...

- repeating experiments is too expensive - airplane
- too slow or too dangerous - nuclear

Current design process:

block diagram → parametrized software model → simulations → analysis → adjust parameters → simulatons → ... → prototype →...

Need for fast(er) processing of large data also in

- data minimg, information retrieval, computational finance
- video games, medical imaging, drug design, etc.

Algorithms costs

How can one speed-up serial computations?

Algorithms have two costs (measured in seconds):

- cost of arithmetic/logic operations
- cost of moving data - read/write

For a ”traditional” serial CPU these are difficult to improve.

Some barriers to serial processing

”Walls”:

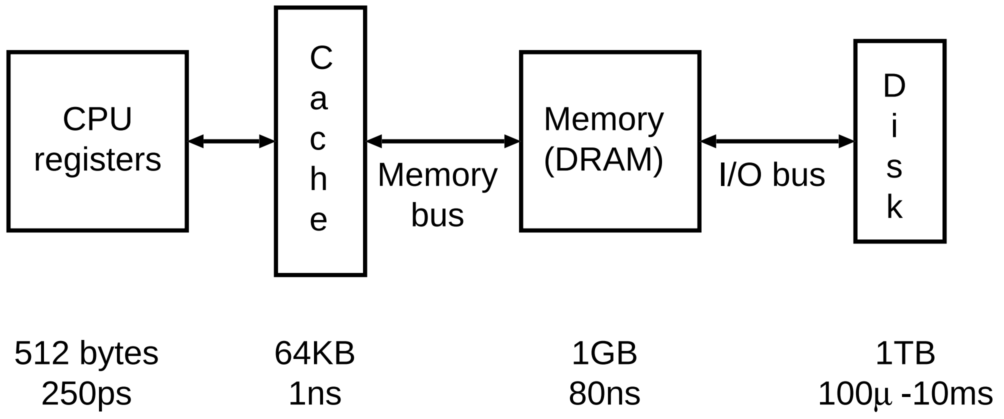
- Memory Wall
- Power Wall
- ILP Wall

Barriers to Serial Performance

Memory Wall

- Time (and energy) to fetch data from outside the CPU chip is much greater than time per flop
- This is due to long latency and limited communication bandwidth beyond the CPU chip.
- Multi-level caches improve the average memory reference time.

Barriers to Serial Performance



Improvement per year (1995-2010)			
time/flop		bandwidth	latency
1.5	network	1.25	1.15
	DRAM	1.22	1.06

Ref: H& P (2012)

CPU is easily starved of data.

Barriers to Serial Performance

Power Wall

- for faster speed increase the clock frequency
- in past decades we have gone from KHz to GHz
- power dissipation is proportional to clock frequency and feature length
- further increase in clock speed without significant (expensive) cooling is not possible
- in the last several years the clock speed has remained flat

Instruction Level Parallelism (ILP) Wall

- concurrent execution of independent instructions on duplicate hardware
- speculative execution (branches)
- one needs to examin large subblocks of instructions to find independent ones (done mostly in hardware)
- large and complex execution units with diminishing returns

Transistors are cheap

Make use of more and more transistors

- the number of transistors/inch^2 in circuits roughly doubles every 2 years
- 1970's - 20K transistors, 2018 - 20–30B transistors (AMD,NVIDIA)
- duplicate resources
  - multicores (10's)
  - graphic processing units (GPUs) (1000's)

Also, we can have many interconnected (multicore) processors each equipped with GPUs.

Challenge: parallel thinking

(We are observing) very little improvement in serial performance of general purpose CPUs.

So if we are to continue to enjoy improvements in software capability at the rate we have become accustomed to, we **must use parallel computing**.

Parallelism

Parallelism is not a new idea

- 1960's - theory of cellular machines (Ulam and von Neuman)
  - find computational cost $c(n)$, n is size of data.
  - for N cells, what is $c(n)$ when $N \to \infty$?
  - cost of moving data neglected
  - "systolic" model (early 80's) includes cost of communication
- Early parallel computers: Illiac IV in 60's, Cmmp, Cm* in 70's
- VLSI revolution of 80's, theory of area$\times$time complexity
- parallelism in CPU invisible to the programmer
  - bit level - array multiplier
  - pipelined execution and multiple functional units

Parallelism

Parallelism is to stay.

- all major processor vendors are producing multicore chips

It is not difficult to write parallel programs (?)

- a problem is broken into (quasi) independent subproblems
- subproblems are executed simultaneously on different CPUs
- partial results are merged
- (hopefully) solved in less time than on a single CPU.

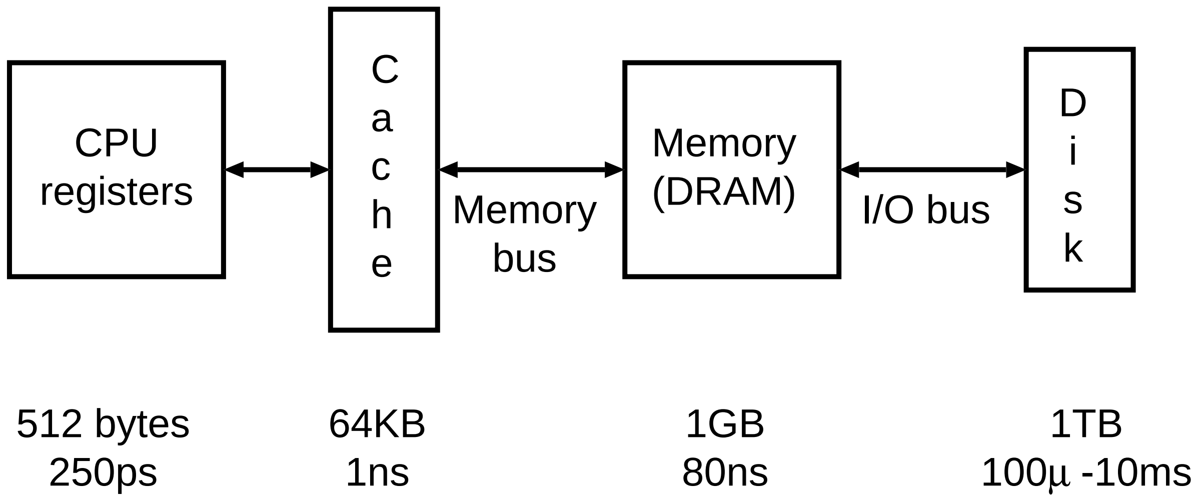
Making parallel programs effcient is complicated!

- Can (sequential) production codes run in parallel?
  Needs parallelizing compiler (CS).
  Needs rethinking of algorithm (area experts).

Single CPU - memory issus

Achieving peak performance for a single CPU is difficult.

Memory operations are not all the same.



- limited bandwidth beyond chip boundary
- we are often interested in ”average memory access” time per application.

Latency

$$\text{time to move data} = \text{latency} + (\#\ \text{bytes})\cdot\ (1/\text{bandwidth})$$

Latency: delay from start to response.

Bandwidth: # bytes transfered per unit of time.

Main memory latency is high
Main memory bandwidth a bit better (more lanes)

For off chip communication

$$\text{time/ops} << \text{time/(unit of data moved)} << \text{time/message}$$

Locality of memory access

Caches to the rescue!

Important observation: Programs usually have locality

- **spatial locality** (?)
- **temporal locality** (?)

Can improve "average memory access time" by introducing an intermediate store (cache) between registers and main memory.

Already known to von Neumann in 50's.

Fast memory acces by cheating

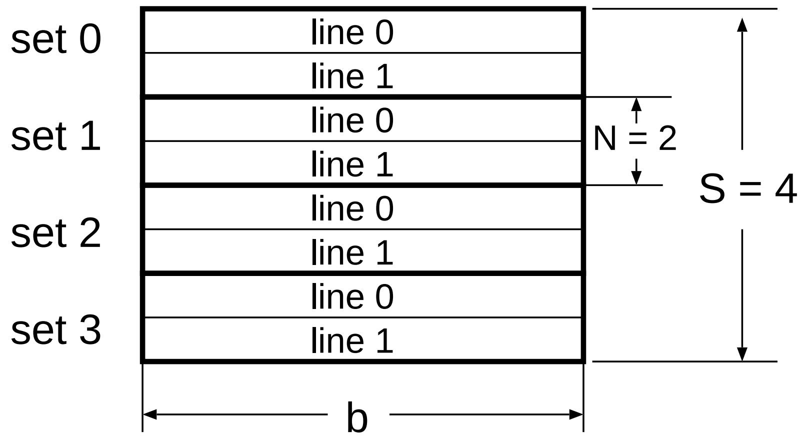
How caches help:

- spatial locality - fetch multiple bytes (a cache line) all at once
  - need high memory bandwidth
- temporal locality - hide memory costs by reusing data
- prefetching - overlap computation and communication with memory (very important!)

This is mostly automatic and implicit (but programmers can help).

Cache organization

Cache is an SRAM organized into a collection of same size sets:



Capacity C (in words)

Line size b (in words)

Degree of associativity N

# lines = C/b, # sets = C/N

Each (main) memory address maps to exactly one cache set.

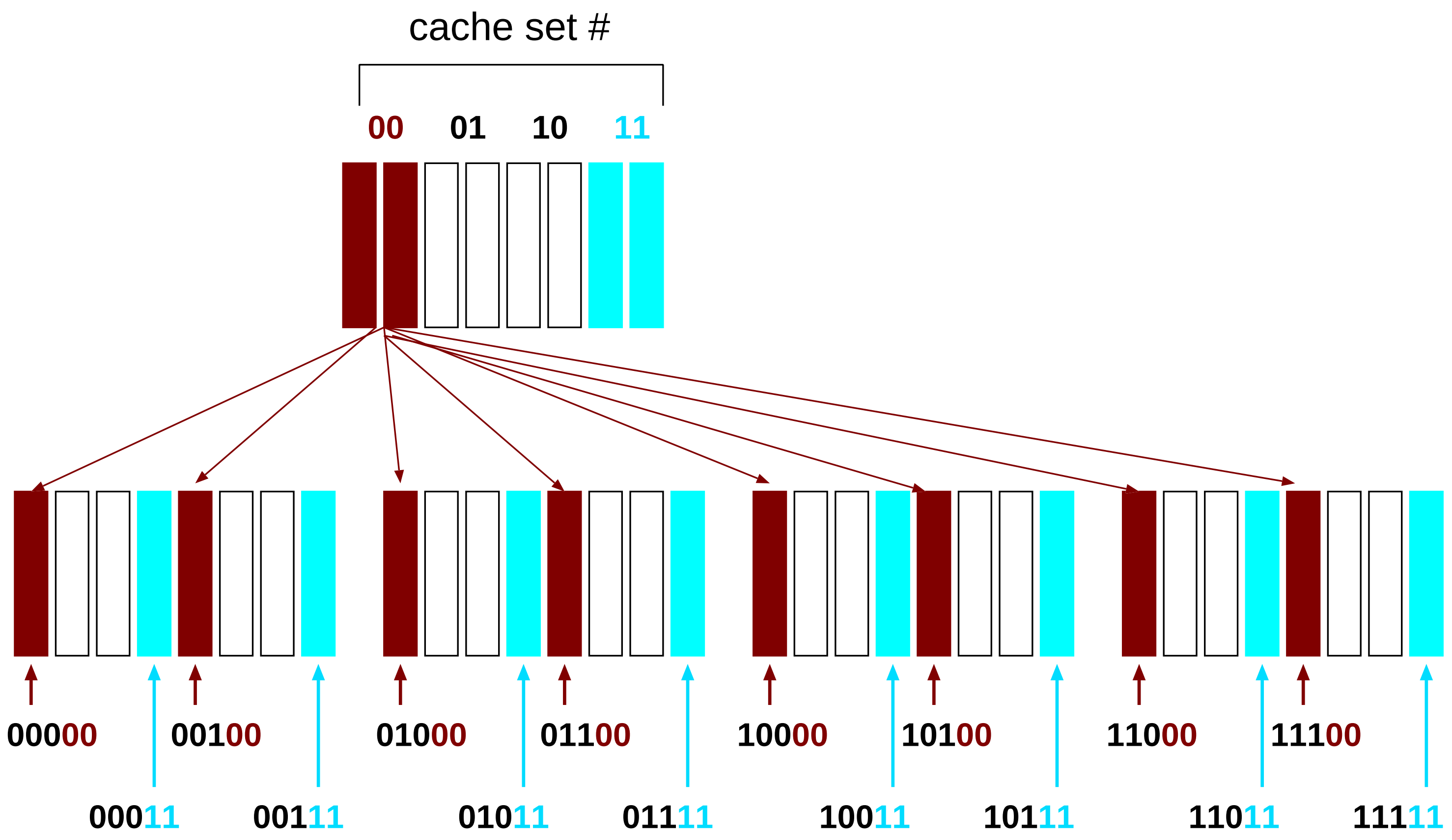
Direct mapped cache

There is only one line per set ($N = 1,\ C = 8$).

1 word per line, $b = 1$		2 words per line, $b = 2$	
M_addrs	C_addrs	M_addrs	C_addrs
XXX...XXX**000**00	000	XXX...XXX**00**000	00
XXX...XXX**001**00	001	XXX...XXX**00**100	00
XXX...XXX**010**00	010	XXX...XXX**01**000	01
XXX...XXX**011**00	011	XXX...XXX**01**100	01
XXX...XXX**100**00	100	XXX...XXX**10**000	10
XXX...XXX**101**00	101	XXX...XXX**10**100	10
XXX...XXX**110**00	110	XXX...XXX**11**000	11
XXX...XXX**111**00	111	XXX...XXX**11**100	11

XXX...XXX	**10**	**1**
tag	set	offset

Associative cache

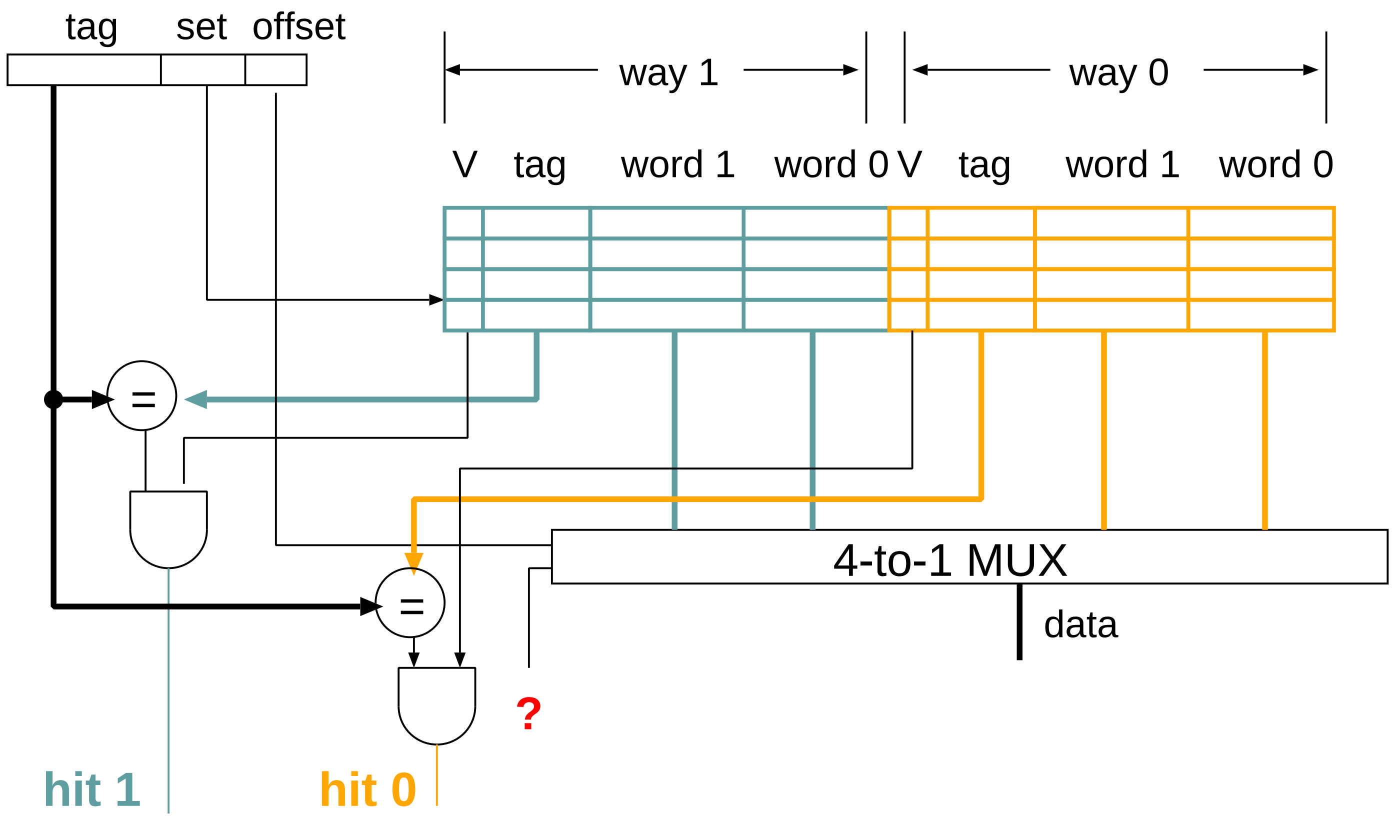


Less sets, more slots per set (longer tags).

Higher associavity higher cost (8 is common).

Associative cache

- Cache **hit** when copy of needed data in cache
- Cache **miss** otherwise.



From Harris& Harris, ? = hit 1

Caches

Miss types:

- **Compulsory** (?)
- **Capacity** (?)
- **Conflict** (?)

Caches

- Compulsory: first use of data
- Capacity: need more data than can fit in cache
- Conflict: insufficient associativity

AMAT

Estimating average memroy access time

AMAT = Hit_Time_L1+Miss_Rate*MIss_Penalty_L1

MissPenalty_L1 = Hit_Time_L2+Miss_Rate_L2*Miss_Penalty_L2

Hit time L1	1 cycle
Miss rate L1	0.05
Hit time L2	5 cycle
Miss rate L1	0.20
Miss penalty L2	50 cycles

AMAT = 1+0.05*(5+0.2*50) = 1.75

Arrays

An n-dim array is stored as a 1-dim array.

A 2-dimensional array (in C) is stored by row

- $A(i, j) = a((i - 1) \cdot n + j)$

Makes a difference when operating on arrays.

Fill-in a 2D array

columnwise:

```
main () {
  int i,j;
  static int x[4][4];
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++) {
      x[j][i] = x[j][i] + a[j][i]; }
  }
}
```

rowwise:

```
main () {
  int i,j;
  static int x[4][4];
  for (j = 0; j < 4; j++) {
     for (i = 0; i < 4; i++) {
       x[j][i] = x[j][i] + a[j][i]; }
   }
}
```

Arrays

columnwise (consecutive elements in a column are not nearby):

```
x[0][0]
          ....3....   x[1][0]
                              ....3....  x[2][0]
                                                   ....3....   x[3][0]
    x[0][1]
                ....3....   x[1][1] etc...
```

rowwise (consecutive elements in a row are next to each other):

```
x[0][0]
    x[0][1]
        x[0][2]
            x[0][3]
                x[1][0] etc...
```

Cache exercises

Program execution time depends on memory access pattern.

Experiment 1: (finding the centroid) Given points $x_i = (x_{i,1}, \ldots, x_{i,k}) \in R^k$, $i = 1, ..., N$, find the centroid $c = (c_1, \ldots c_k)$,

$$c = \frac{1}{N} \sum_{i=1}^{N} x_i$$

by the following methods.

Cache exercises

1. Store the 2D array $X = (x_{i,1}, \ldots, x_{i,k})_{i=1}^{N}$. In a single `for` loop over `i`, sum the coordinates $x_{i,1}, \ldots x_{i,k}$.

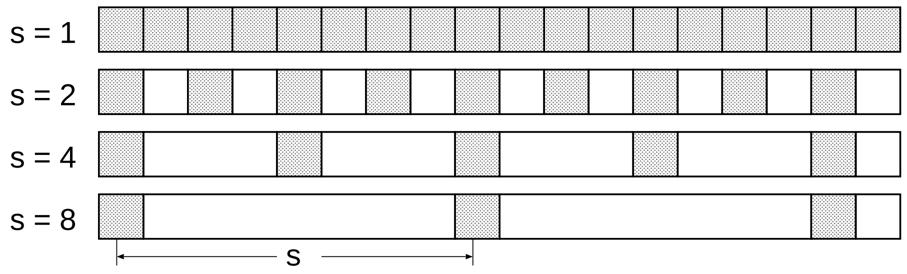
2. Store the 2D array $X = (x_{i,1}, \ldots x_{i,k})_{i=1}^{N}$. Sum $x_{i,1}$ in its own loop, sum $x_{i,k}$ in its own loop, etc.

3. Store $x_{i,1}$, $x_{i,2}$, etc., in separate 1D arrays. Sum $x_{i,1}$ in a single loop, then sum $x_{i,2}$ in another single loop, etc.

Which strategy is best and why?

Cache banchmark

Exercise 2: (memory benchmark) Time to access an array A with different strides



For $A[0 : n - 1]$

- want to have the same number n of accesses for each stride $s = 1, 2, 4, ...$
- to ”warm” cache repeat each experiment K times

Cache banchmark

```
for ($n \leftarrow 2 \cdot n$)                                          double size
    for array $A[0 : n-1]$
        for ($s \leftarrow 2 \cdot s$)                                  double stepsize (stride)
            Tstart = gettime()                                      start timer
            for ($k = 0 : 1 : K \cdot s$)                           repeat K times
                for ($i = 0 : s : n-1$)                             load A[0], A[s], A[2s],..
                    load $A(i)$
            $T(n, s)$ = (gettime() − Tstart)/($n \cdot K$)     average access time
```

Cache exercises

From time measurements, try to find:

1. L1 cache size,
2. L1 cache line length,
3. L1 latency and associativity,
4. L2 latency,
5. L2 cache size,
6. is there anything else that you can find out?

Performance is a complicated function of the architecture

Programmer can help

Compilers can help (use correct flags)

Single CPU simple performance model

(Accurate) modelling of performance is difficult but simple models can provide some insight.

Simple model (after J. Demmel):

K	- # words read from slow memory
t_{mem}	- slow memory access time
N	- # number of flops
t_{fl}	- time per flop
$q = \frac{N}{K}$	- average flops / slow memory access

Time : $N \cdot t_{fl} + K \cdot t_{mem} = N \cdot t_{fl} \left(1 + \frac{K}{N} \cdot \frac{t_{mem}}{t_{fl}}\right) = N \cdot t_{fl} \left(1 + \frac{1}{q} \cdot \frac{t_{mem}}{t_{fl}}\right)$

Larger $q = \frac{N}{K}$ is better.

Single CPU simple performance model

$$t_{total} = N t_{fl}(1 + \frac{t_{mem}}{t_{fl}} \cdot \frac{1}{q})$$

Example 1:

$t_{mem} = 80ns,\ t_{fl} = 0.4ns,\ N = 10^6,\ q = \frac{N}{K} = 2$

$t_{total} = (10^6 \cdot 0.4ns) \cdot (1 + 100) \approx 10^8 \cdot 0.4ns,\ \ t_{ideal} \approx 10^6 \cdot 0.4ns$

Potential **2.5Gflop** to realized **25Mflops**

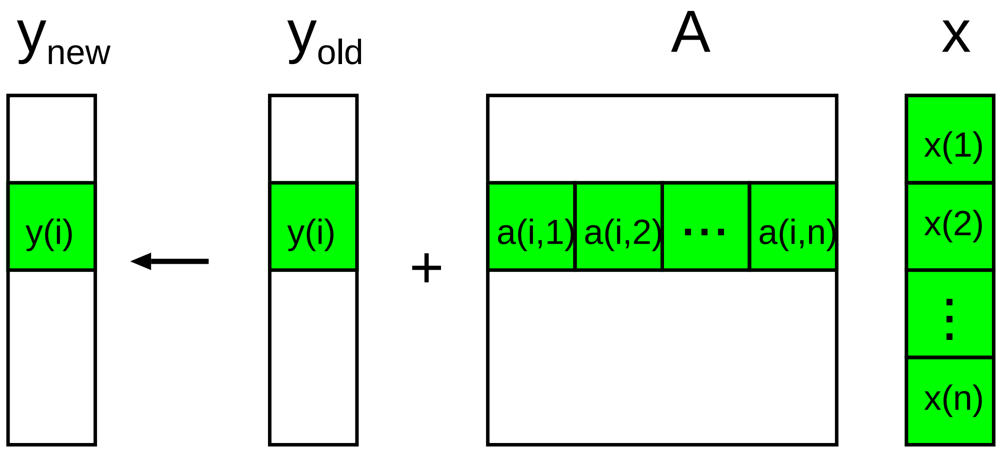
- $q = \frac{N}{K}$ - the larger q the faster the algorithm (total time closer to minimum $N \cdot t_{fl}$) - algorithm designers
- the smaller $\frac{t_{mem}}{t_{fl}}$ the faster the machine is - computer engineers

Analysis of matrix-vector multiply

$y \leftarrow y + Ax$, or $y(i) \leftarrow y(i) + \sum_{j=1}^{n} A(i,j)x(j)$

Classical algorithm (row-by-column)

```
for $i = 1 : n$
    for $j = 1 : n$
        $y(i) \leftarrow y(i) + A(i,j)x(j)$
```



Simplified analysis

```
load $x(1:n)$ and $y(1:n)$ into fast memory
for $i = 1:n$
    store $A(i,:)$ into fast memory
    for $j = 1:n$
        $y(i) \leftarrow y(i) + A(i,j) \cdot x(j)$
store $y(1:n)$ to slow memory
```

K (# slow memory ops, load x, y, A store y) $= 3n + n^2$

N (# flops) $\approx 2n^2$

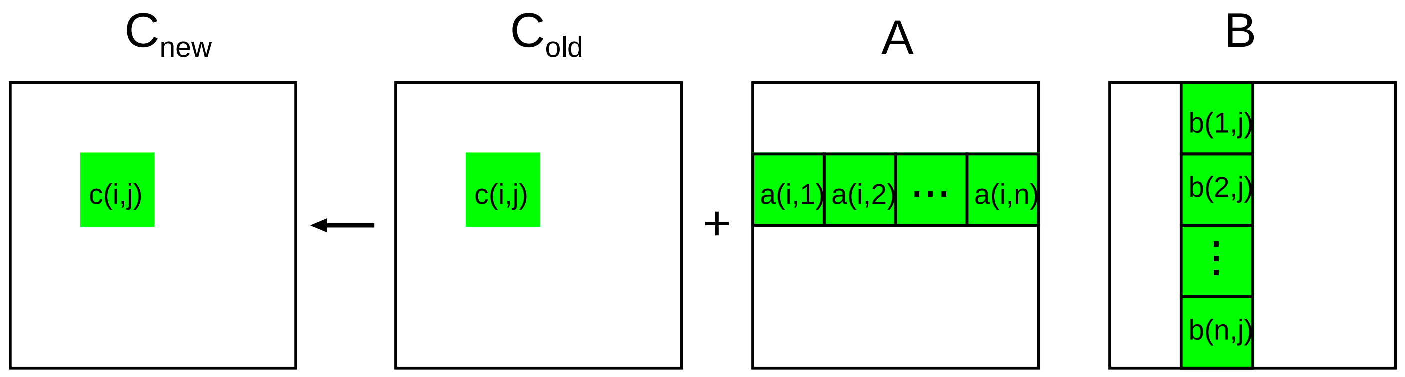
$q = \frac{N}{K} \approx 2$

Speed of mat-vec mult limited by slow memory (1% - 2% of peak).

Matrix-matrix multiply

$C \leftarrow C + A \cdot B,$

(or $c(i,j) = c(i,j) + \sum_k a(i,k) \cdot b(k,j)$) - row-by-column



$N \approx 2n^3$ flops

$K \approx 3n^2$ slow memory access

$q = \frac{N}{K}$ potentially as large as $\mathbf{\frac{2}{3}n}$

In Example 1: $(1 + \frac{t_{mem}}{t_{fl}} \cdot \frac{1}{q}) \approx \frac{150}{n}$, great for large n

Matrix-matrix multiply

$c(i,j) = c(i,j) + \sum_k a(i,k) \cdot b(k,j))$

```
for $i = 1 : n$ (loop over all rows)
    load $A(i,:)$ into fast memory
    for $j = 1 : n$ (loop over columns)
        load $c(i,j)$ into fast memory
        load $B(:,j)$ into fast memory
        for $k = 1 : n$ (accumulate dot product)
            $c(i,j) \leftarrow c(i,j) + a(i,k) \cdot b(k,j)$
        store $c(i,j)$ back to slow memory
```

Load $B(:,j)$ n times for total n^3 loads

Laod $A(i,:)$ only one time for total n^2 loads

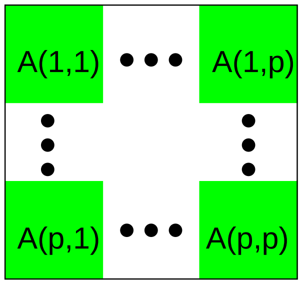
Load and store $c(i,j)$ once for total $2n^2$ slow memory ops

Total slow memory ops are $n^3 + 2n^2$, $2n^3$ flops

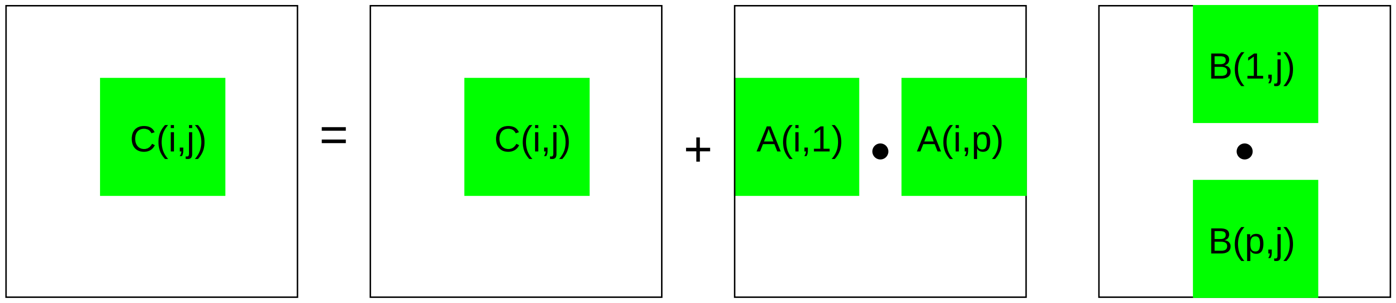
$\mathbf{q} \approx \mathbf{2}$, what did go wrong? Weak reuse of B.

Block matrix-matrix multiply

Partition A, B, C into $p \times p$ block matrices, with blocks $C(i,j),\ b(i,j),\ B(i,j)$ being $b \times b$ (sub)matrices (thus $n = bp$).



$C(i,j) \leftarrow C(i,j) + \sum_{k=1}^{p} A(i,k)B(k,j)$ still holds.



Block matrix-matrix multiply

$C(i,j) \leftarrow C(i,j) + \sum_{k=1}^{p} A(i,k)B(k,j)$

```
for $i = 1 : p$ % there are $p$ block rows
    for $j = 1 : p$ % there are $p$ block columns
        load block $C(i,j)$ into fast memory
        for $k = 1 : p$
            load block $A(i,k)$ into fast memory
            load block $B(k,j)$ into fast memory
            $C(i,j) \leftarrow C(i,j) + A(i,k)B(k,j)$
        store $C(i,j)$ into slow memory
```

load $A(i,k)$ and $B(k,j)$ p^3 time for total of $2p^3b^2 = 2pn^2$ loads

load and store each $C(i,j)$ once for total of $2n^2$ memory ops

total memory ops is $(2p+2)n^2$

$q = \frac{2n^3}{(2p+2)n^2} \approx \frac{n}{p} = b$

How large the block size b should be?

Block matrix-matrix multiply

All 3 blocks from A, B and C must fit in fast memory.

Say, the fast memory has size M. Then we need

$$3b^2 \leq M \ \Rightarrow \ q \approx b \leq \left(\frac{M}{3}\right)^{\frac{1}{2}}$$

Is it the best possible?

TH (Kung, 1981) Any reorganization of this matrix-matrix multiply algorithm is bounded by $q = O(M^{\frac{1}{2}})$.

Important observation for a single CPU with a cache:

Matrix-matrix like operations are very efficient.

Implications

Critical lesson for performance

- **data locality**
  - accessing (remote) data is the most expensive operation.
- reuse of recent data

Next time...

- Hardware models
- Programming models