# Flops & Memory Cache Hierarchy

To assess the performance of an algorithm we have been counting arithmetic operations.

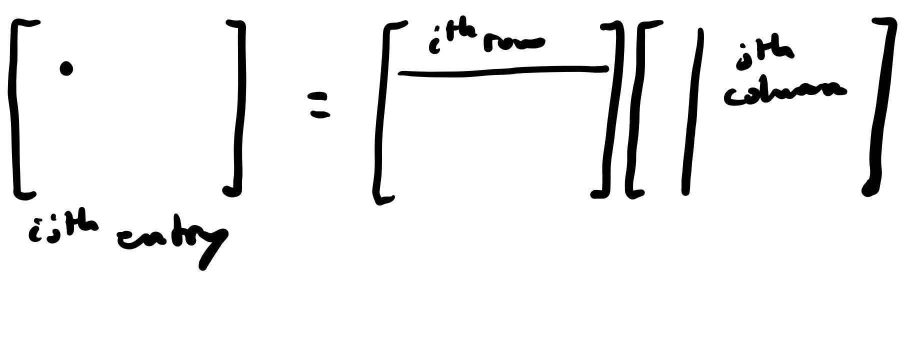
Around the 1980s, CPUs became fast enough that arithmetic costs were not the primary bottleneck. Instead, the costs of storing and manipulating numbers in memory became increasingly important.

## Matrix Multiplication

$$\underset{}{C} = \overset{m \times m}{A} B \quad \Rightarrow \quad \sim 2m^3 \text{ flops}$$

for $i = 1:m$
  for $j = 1:m$
    $C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$

"Naive Algorithm"



If flops are the dominant cost on a 2.6 GHz processor, we might expect

$$P_{mm} = \left[\frac{2m^3 \text{ flops}}{t \text{ ns}}\right] \approx \left[\frac{2 \text{ flops}}{1 \text{ cycle}}\right]\left[\frac{2.6 \text{ cycles}}{1 \text{ ns}}\right]$$

time for mat-mat product in nanoseconds

$= 5.2$ flops/ns (Gigaflops)

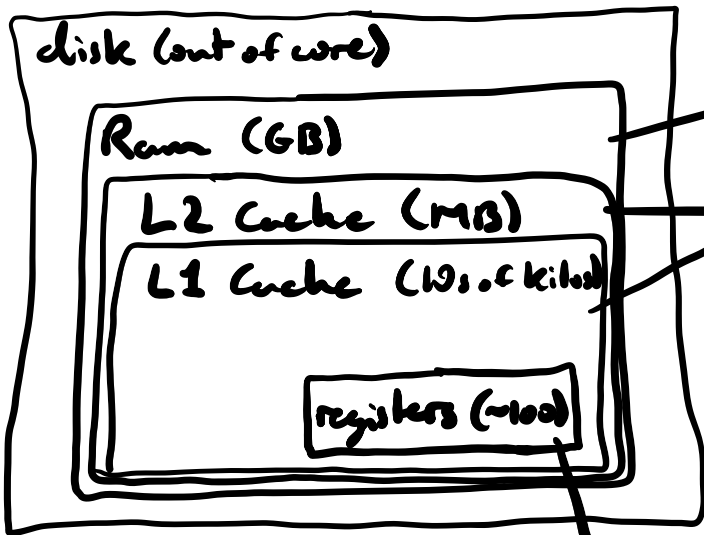
$\Rightarrow$ See Matmults.pdf for actual experiment.

$\Rightarrow$ Observe that $P_{mm}$ is $<1$ gigaflop and decreases as $m$ increases!

$\Rightarrow$ Highly optimized BLAS mat-mat product does achieve near peak theoretical flop rate using same # operations and mathematically equivalent algorithm.

How do we understand the difference in performance if flops are the same?

# Memory Hierarchy




10s of ns to access (~30 cpu cycles!)

local caches closer to CPU for faster access

wired directly into arithmetic (essentially "free")

*Some also have L3, even L4 cache

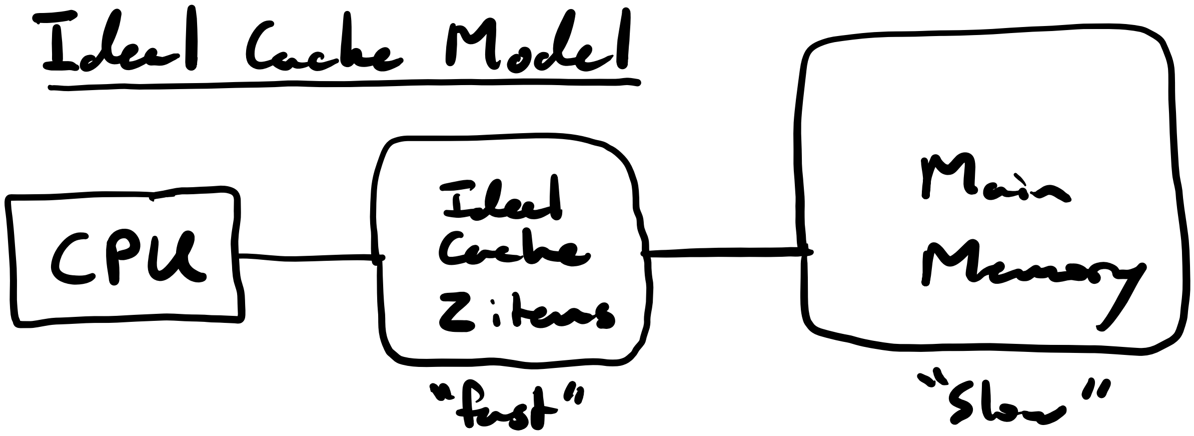
Not all flops are created equal!

⇒ Depends <u>where</u> & <u>when</u> data is accessed

<u>Idea:</u> do as <u>much</u> work as possible with data in "fast" memory before loading new data into local cache.

In practice, local caches are managed at hardware level, not by typical programmer. They replace data which has not been used recently with data needed currently.

# Ideal Cache Model




Cache "hit": CPU needs item from cache

Cache "miss": CPU needs item not in cache, item then loaded into cache for future use.

<u>Idea</u>: Loaded item replaces item that will not be used for longest time in future.

$\Rightarrow$ Analyze algorithms by counting cache misses $Q$.

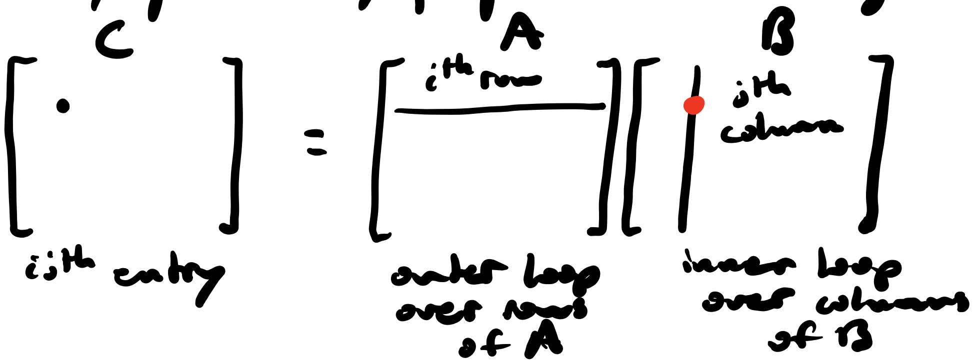
Idealized model but easier to analyze and reveals basic ideas needed to understand memory-efficient algorithms. Gets within a constant of optimal!

# Cache performance of naive Mat. Mult.

Notation: $f = \Theta(g)$ if constants $\alpha, \beta > 0$ s.t.

$$\alpha g(x) \le f(x) \le \beta g(x) \quad \text{as } x \to +\infty.$$

" $f$ Asymptotically proportional to $g$ "




Cache size $Z$

Cache misses $Q(n, Z) = n(n^2 - Z)$

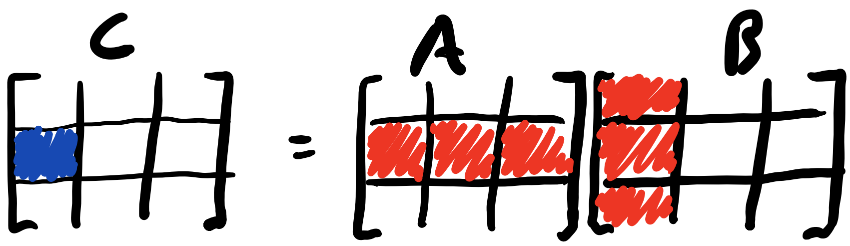
loop through <u>all</u> entries of B for every row of A.

$$Q(n, Z) = \Theta(n^3)$$

$\Rightarrow$ makes no use of Cache asymptotically

# Cache-Aware Matrix Mult.

Idea: Load in blocks of A, B and do all ops with these blocks before loading new data into cache.




$\left(\frac{n}{b}\right)^2$ blocks

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31}$$

block size $= b \Rightarrow 3b^2 = Z$

store $C_{21}, A_{2j}, B_{j1}$

Now, $Q(n, Z) = \Theta\Big(\underbrace{2b^2}_{\text{block size}}\underbrace{\left(\frac{n}{b}\right)^3}_{\text{\# of blocks}}\Big)$

$$= \Theta\left(n^3/\sqrt{Z}\right) \quad \text{as } n \to \infty$$

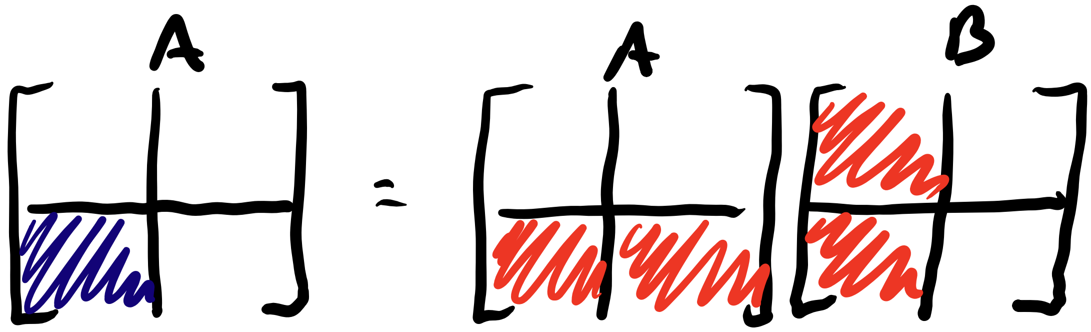
$\Rightarrow$ benefit from Cache

$\Rightarrow$ Actually need to do this hierarchically.

# Cache-Oblivious

→ Divide recursively

$$Q(n) = \begin{cases} 8Q(\frac{n}{2}) & \text{otherwise} \\ 3n^2 & 3n^2 \le Z \end{cases}$$




$$C_{21} = \underbrace{A_{21}B_{11}}_{\substack{\text{call} \\ \text{recursively} \\ \text{untill} \\ n \text{ suff. small}}} + A_{22}B_{21}$$

for $n^2 >> Z$

$$Q(n) = 8Q(\frac{n}{2})$$

$$= 8 \cdot 8 \cdot Q(n/4)$$

$= 8Z$

$$= \underbrace{8 \cdots 8}_{k\text{ times}} \left[ 3\left(\frac{n}{2^{2k}}\right)^2 \right] \qquad \frac{1}{4} < f \leq 1$$

$$\Rightarrow \quad = (2^3)^k \, 3\left(\frac{n}{2^k}\right)^2 = 2^k \, 3n^2$$

$2^k = n\sqrt{\frac{3}{fZ}}$

$$= \Theta\left(\frac{n^3}{\sqrt{Z}}\right)$$

=> Same asymptotic cache benefit,
but w/out explicit knowledge of cache size!