Memory Cache Hierarchy

- To assess the performance of an algorithm. we have been counting arithmetic operations. Around the 1980s, CPUs become fast enough that arithmetic cost were not the primary bottleneck. Instead, the costs of storing and manipulating numbers in memory become increasingly important.
- In many problems, especially problems accessing lots of data and doing relatively simple computations on each datum, the performance bottleneck is memory rather than computational speed. Because memory is arranged into a memory hierarchy of larger/slower and smaller/faster memories, it turns out that changing the order of memory access can have a huge impact on performance.

CPU Architecture

- A modern CPU can be understood as a giant synchronous finite-state machine. Inside the CPU, there is a clock generator built from transistors that produces periodic signal and sync all transistors. The cycle is the smallest unit of time at which the CPU's state is permitted to change. (Period of the signal) Inside each cycle, signals must physically settle into stable configurations. The frequency (GHz) tells you how quickly this signal ocsillates. (Faster clock → physically unstable/too hot)

- During each cycle, CPU implements instructions. Usually implement one instruction take $1 \sim 15$ cycles, depending on the complexity of the operation and the clock frequency.
- On modern CPUs, a "vector unit" (also called SIMD: Single Instruction, Multiple Data) can apply one instruction to several numbers simultaneously. For example, an AVX-512 instruction can operate on 8 doubles at once (or 16 floats). Not only that, but there are also "fused-multiply add" (FMA) that perform $x + ay$ in a single instruction.
- For example, on a 2.6 GHz CPU with 2 flops/cycle via SSE instructions, we might expect the maximal flop/ns

$$\frac{2 \text{ flops}}{1 \text{ cycle}} \times \frac{2.6 \text{ cycle}}{1 \text{ ns}} = 5.2 \text{ flops/ns (Gigaflops)}$$

We can test this on an "obvious" C code for matrix-matrix multiplication:

$A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times n}$, $C = AB$

$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ $\quad 2n^3$ flops (adds + mults)

"Naive" matrix multiplication:

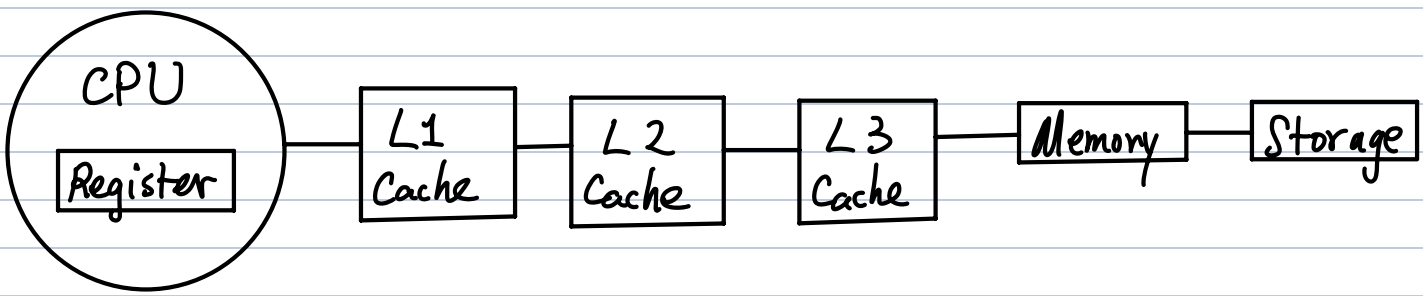
> Given A, B as arrays, and initialize C as an array
> For $i = 1, \cdots, n$
> For $j = 1, \cdots, n$
> $c_{ij} = 0$

```
        For k = 1, ..., n
            c_ij += A[i,k] * B[k,j]
        end
        C[i,j] = c_ij
    end
end
```

Yet if we compare $2n^3$/ time for mat-mat product. it is 10 times slower. (See matmuls.pdf)

- Highly optimized BLAS mat-mat product does achieve near peak theoretical flop rate using mathematically equivalent algorithms. But w/ 130,000+ (ATLAS)

Memory hierarchy (Desktop)

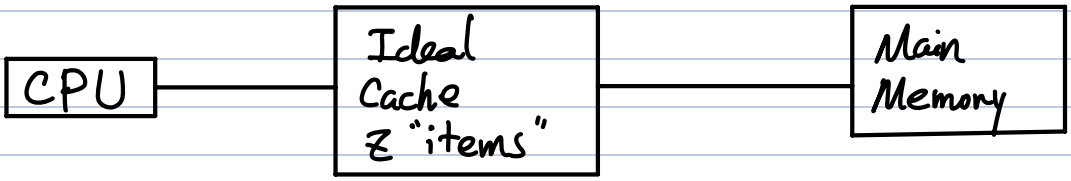


Size:	4,000 bytes	64-128 KB	.5-1 MB	16-64 MB	16-128 GB	.5-4 TB
Speed:	200 ps	1 ns	3-10 ns	10-20 ns	50-100 ns	50-100 μs

- Modern computers are made with fast but small and expensive (on-chip) cache. and slow but large and cheap (off-chip) memory.

- Because the trade-off between the two types of memories is so extreme, it makes sense to have some of each. Carefully combining memories of different speeds can have a huge impact on the performance of an algorithm.

Ideal Cache Model



- Cache hit: CPU needs and finds an item in cache (fast)
- Cache miss: CPU needs an item which is not in cache.
  In this case, the item will be loaded into cache for future use, and replace some other item if no space available in cache or a prescribed space for the item is not available (set-associative cache).
- Fully associative cache: Any item in memory can go anywhere in the cache. (Impractical because we have to check all items on every access to the cache)
- Optimal replacement: On cache miss, load item replaces item that will not be needed for the longest time in the future. (More realistic sheme: Least recently used (LRU) replacement, provably within small constant factor of optimal, but much harder to analyze)

- Cache complexity: the number of cache misses $Q(n; Z)$ required for a given algorithm running on a problem

  (It also takes time to search for target item in the cache for each cache hit / miss. But the time is ~1ns while accessing memory takes ~50ns.)

Remark: To make cache efficient, we need a principle that helps predict which data is likely to be accessed frequently in the future. Such principle is called locality. Both of them are assumptions about how "normal" program are likely to behave:

- Temporal locality: If a program accesses a given value, it is likely to need to access the same value again sometime soon.
- Spatial locality: If a program accesses a given value, it is likely to access nearby values in main memory (i.e. addresses that are numerically close to the original address) sometime soon.

  The spatial locality can be exploited by cache lines (i.e. load L nearby values from main memory upon each cache miss).

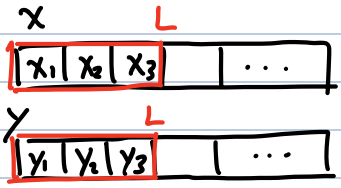
Ex. Consider a dot product $x^T y = \sum_{i=1}^{n} x_i y_i$

- Uses each input exactly once
- no reuse $\Rightarrow$ no benefit from cache?

Real caches load L consecutive in memory values on every miss, L = cache line

- Optimized for consecutive memory access

Naive dot product (x, y) :




```
s = 0
for i = 1 to n
    s += x_i * y_i
end
```

has $Q(n; z) = \Theta(n/L)$ misses - still benefit from cache as long as access is consecutive (spatial locality)

Ex 2. Consider matrix addition:

$$\underset{n\times n}{C} = \underset{n\times n}{A} + \underset{n\times n}{B}$$

Naive matrix addition

```
For i = 1, ..., n
    For j = 1, ..., n
        c_ij = a_ij + b_ij
```

Swap for column-major

$Q = \Theta(n^2)$

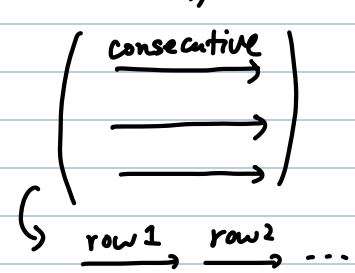
good for row-major (consecutive)

$Q = \Theta(\frac{n^2}{L})$

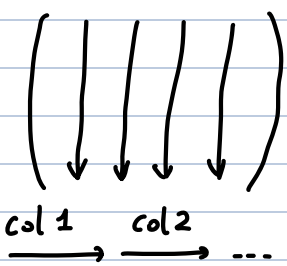
Matrix storage: $A = (a_{ij})_{n\times n}$ ⟶ "flattened" into "1d" memory

Two common orderings

row-major (C, Numpy default)



column-major (Fortran, Matlab, Julia)



Cache complexity for mat-mat multiplication

We take cache line $L = 1$ in the analysis (otherwise Q

will depend on how the matrix is stored)

- For naive mat-mat multiplication, the cache misses

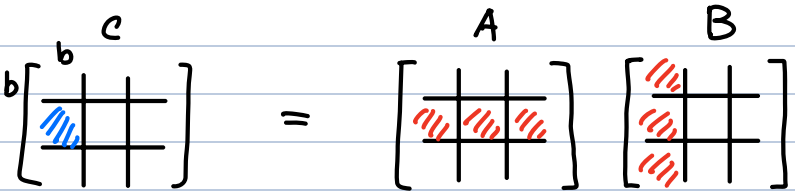
$$Q(n, Z) \approx m(m^2 + m - Z) = \Theta(m^3)$$

↑ loop through B for every row of A

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. Sometimes the technique is called tiling / blocking.



$(\frac{m}{b})^2$ 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}

$b \times b$ multiplication
A_{ij}, B_{jk} are complete in cache after loading ($2b^2$ misses)

The cache miss is now

block size ↑ , # of blocks →

$$Q(n, Z) = \Theta\left(2b^2 \left(\frac{m}{b}\right)^3\right)$$

$$= \Theta\left(\frac{m^3}{\sqrt{Z}}\right) \leftarrow \text{benefit from cache}$$

Intuition: blocks are good because they have $\Theta(b^3)$ flops done for $\Theta(b^2)$ misses. Since there are many flops per miss, arithmetic dominate the cost.

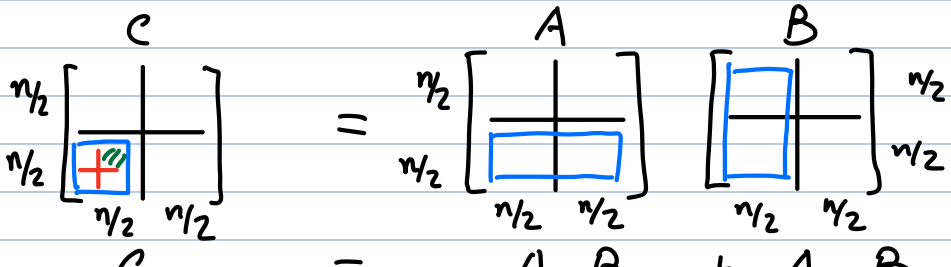
- Cache-oblivious matrix mult.
  - Cache size is an explicit parameter in cache-aware mat. mult.

    pro: highly optimizable

    con: cache-size dependent, on modern CPU we have multi-levels of caches (L1, L2, L3, ...). need to do nested blocking, leading to complex code.
  - In cache-oblivious algorithms, cache size Z is not an explicit parameter. Yet there exists optimal cache-oblivious algorithms that achieves best case Q in asymtotic sense regardless of Z. for many algorithms: mat. mult., sorting, FFTs, backsubstitution, ...

    pro: write program once for all machines (exploits all levels of cache!

    con: only optimal up to constant factors
  - General idea of optimal cache-oblivious algorithms: recursive divide-and-conquer, eventually fit into any cache ...



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

call program recursively until n small so that $3n^2$ fits into cache

misses

$$Q(m) = \begin{cases} \beta\, Q\left(\frac{m}{2}\right), & \text{otherwise} \\ 3m^2, & 3m^2 \le Z \end{cases}$$

Asymptotically for $m^2 \gg Z$.

$$Q(m) = \beta\, Q\left(\frac{m}{2}\right)$$

$$= \dots = \beta^k\, Q\left(\frac{m}{2^k}\right) = \beta^k \cdot 3 \cdot \frac{m}{2^k}$$

$$= 3 \cdot 2^k \cdot m = \Theta\left(\frac{m^3}{\sqrt{Z}}\right)$$

$\Rightarrow$ Same asymptotic cache benefit,

but w/o explicit knowledge of cache size!

$3 \cdot \left(\frac{m}{2^k}\right)^2 = f Z, \quad \left(\frac{1}{4} < f \le 1\right)$

$\Rightarrow 2^k = m\sqrt{\frac{3}{fZ}} = \Theta\left(\frac{m}{\sqrt{Z}}\right)$