Parallel and Distributed Programming

Imad Kissami[1]

[1]Mohammed VI Polytechnic University, Benguerir, Morocco

October 27, 2025

Outline of this lecture

- Computational Intensity
- Two Memory level model
- Data Locality
  - The Penalty of Stride
  - High Dimensional Arrays
  - Principles of good data locality

SOME DEFINITIONS

FLOPS

FLOPS is thefloating point operations per second. It is expressed as `FLOPS` or `flop/s`

- Let's consider the multiplication of two $N \times N$ matrices.
- For each element in the result, we perform N multiplications and $N - 1$ additions.
- Total floating point operations (FLOPs) = $2 \times N^3$
- If this operation takes T seconds, then the achieved performance is:

$$\text{FLOPS} = \frac{2 \times N^3}{T}$$

- Example: For $N = 1000$, the operation involves 2×10^9 FLOPs.
- If the computation takes 1 second, the performance is 2×10^9 FLOPS (i.e., 2 GFLOPS).

SOME DEFINITIONS

Memory Latency

Memory Latency is the time between initiating a request for a word in memory until it is retrieved by the CPU. It is expressed in `clock cycles` or in time.

- Clock Cycle Duration: The time of one CPU cycle is inversely proportional to its frequency:

$$\text{Cycle time} = \frac{1}{\text{Frequency}}$$ (e.g., at 2 GHz: 1 cycle = 0.5 ns)

- Latency (in time) is computed from latency in cycles and CPU frequency:

$$\text{Latency (s)} = \frac{\text{Latency (cycles)}}{\text{CPU Frequency (Hz)}}$$

Example: If access takes `100 cycles`

- CPU frequency = `2 GHz` = 2×10^9 cycles/s
- Time = $\frac{100}{2 \times 10^9}$ = 50 nanoseconds (ns)

Access	Latency
Register	≈1 cycle
L1 cache	≈3–5 cycles
L2 cache	≈10–15 cycles
Main memory (DRAM)	≈50–150 cycles

SOME DEFINITIONS

Memory Bandwidth

Memory Bandwidth or Throughput of Memory is the rate at which data can be (read from) or (stored into) a semiconductor memory by the CPU. It is expressed in units of `bytes/second`.

- Suppose a CPU reads a block of data of size `800 MB` from RAM.
- The transfer takes `0.2 seconds`.
- Then, the achieved bandwidth is:

$$\text{Bandwidth} = \frac{800 \times 10^6 \text{ Bytes}}{0.2 \text{ s}} = 4 \times 10^9 \text{ Bytes/s} = 4 \text{ GB/s}$$

- Interpretation:
  - A bandwidth of 4 GB/s means the CPU can continuously transfer 4 gigabytes of data from memory every second.
  - High bandwidth is crucial for data-intensive applications like matrix operations or streaming.

COMPUTATIONAL INTENSITY

Definition

Algorithms have two costs (measured in time or energy):

- Arithmetic (FLOPS)
- Communication: moving data between
  - levels of a memory hierarchy (sequential case)
  - processors over a network (parallel case)

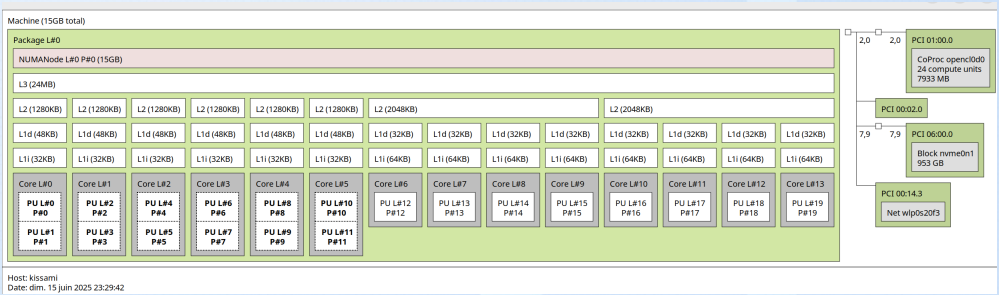
Computational Intensity

It is the ratio between arithmetic complexity (or cost) and memory complexity (cost).

- The cost of arithmetic operations (e.g. floating-point `add` and `mul`) is related to the frequency,
- The cost of memory operations is the cost of moving data

➠ Because moving a word of data is much slower than doing an operation on it, we want to use algorithms with high computational intensity.

Modern architecture (CPU)

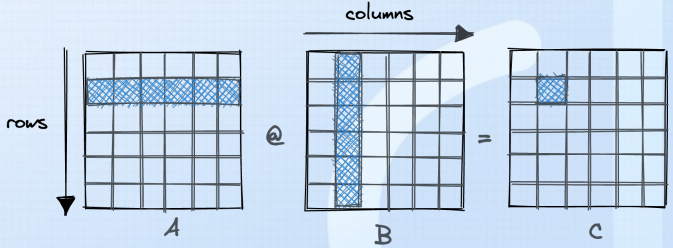


Cache Hit or Miss

- `Cache Hit`: if the CPU is able to find the Data in `L1`
- `Cache Miss`: if the CPU is able to find the Data in `L1-L2-L3` and must retrieve it from `RAM`

COMPUTATIONAL INTENSITY

mxm example: naive version



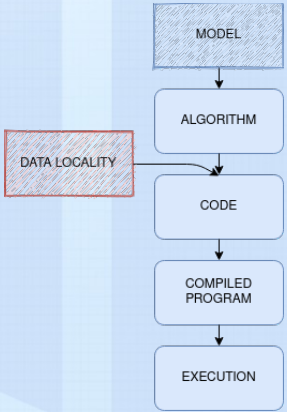

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++)
      C[i][j] += A[i][k] * B[k][j];
  }
}
```

```
arithmetic cost          :: n**3*(ADD + MUL) = 2n**3 arithmetic operations
memory cost              :: WRITE + n**3*(3*READ + WRITE) = n**3 + n**2
computational intensity  :: (ADD + MUL)/(3*READ + WRITE)  ~=2
```

DATA LOCALITY

Introduction

- Data locality is often the most important issue to address for improving per-core performance.
- We've seen that we have 4 levels of memory
- Where in this hierarchy will the processor actually find the data that are needed at any given moment?
- We can gain a speedup of $\sim 10 - 100$ even higher by some simple or more complex manipulations



The Penalty of Stride

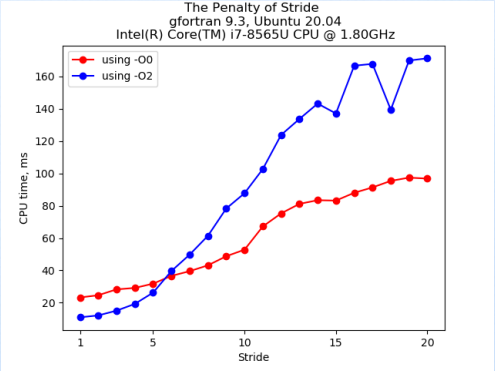
How do we access Data?

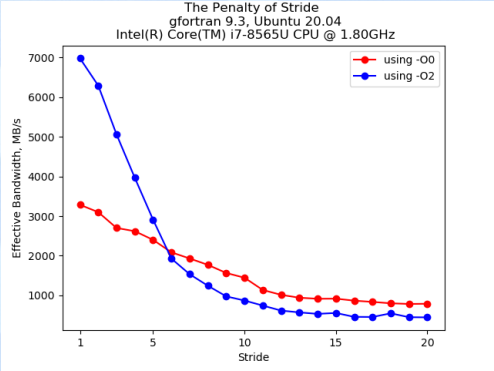
- Not only there are different hierarchies of memories, each one, with a specific memory cost
- We need also to think about how do we access to Data
- We should always arrange your data so that the elements are accessed with unit (1) stride
- The following example will convince you that the penalty for not doing so can be pretty severe

```
for (int i = 0; i < N * i_stride; i += i_stride) {
  mean += a[i];
}
```

- We compile the above Fortran code with all optimization and vectorization disabled (`-O0`) and we run it for different strides
- We do the same thing, with (`-O2`) that activates some optimizations

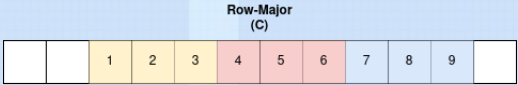
The Penalty of Stride: CPU time vs. Bandwidth

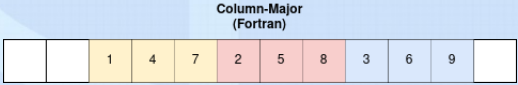




High Dimensional Arrays

- High Dimensional Arrays are stored as a contiguous sequence of elements
  - `Fortran` uses Column-Major ordering
  - `C` uses Row-Major ordering
- In row-major ordering
  - Incrementing the column gives sequential memory access.
  - Iterating over rows of A is cache-friendly: one cache miss followed by 15 contiguous accesses.
  - Iterating over columns of B causes a cache miss at every access.
  - Example: 1024 rows $\times$ 64 bytes = 64KB loaded, exceeding typical 32KB L1d cache $\Rightarrow$ eviction of A and B.
  - Result: each new dot-product requires full reloading from memory.

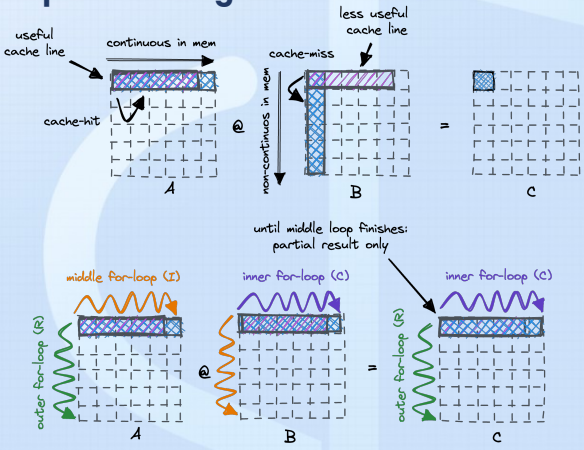




Cache-aware Dot Product: Loop Reordering

- Reordering the inner loops improves memory access patterns.
- We iterate sequentially over B and C, making better use of the cache.

```
void matmul_loop_reordered(int n,
float* A, float* B, float* C) {
  for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
      for (int j = 0; j < n; j++)
        C[i * n + j] += A[i * n + k] * B[k↩
              * n + j];
    }
  }
}
```



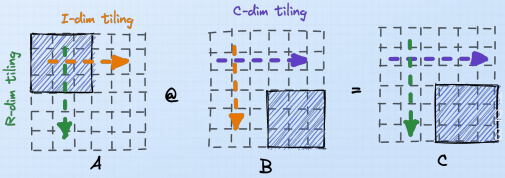
Cache Tiling (Blocking)

- Tiling improves data locality and reuse within the cache.
- Instead of processing full rows, split the computation into tiles.
- Tiling the inner loop ensures previously loaded values stay in cache.
- Example: 6×6 matrices, L1d cache fits 36 floats.
- We split the inner dimension into tiles of size 3.

```
for (int row = 0; row < 6; row++) {
  for (int inner = 0; inner < 3; inner++) {
    for (int col = 0; col < 6; col++) {
      C[row * 6 + col] += A[row * 6 + inner] * B[inner * 6 + col];
}}}
// Second tile
for (int row = 0; row < 6; row++) {
  for (int inner = 3; inner < 6; inner++) {
    for (int col = 0; col < 6; col++) {
      C[row * 6 + col] += A[row * 6 + inner] * B[inner * 6 + col];
}}}
```

Triple Tiling: Optimizing All Matrix Dimensions

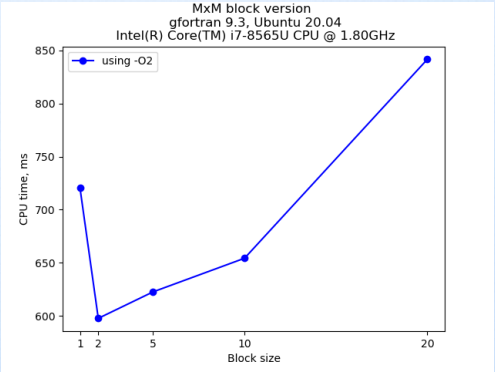
- Tile all three loops to maximize cache reuse.
- Improves locality of A, B, and C simultaneously.
- Essential for large matrix multiplication in HPC.

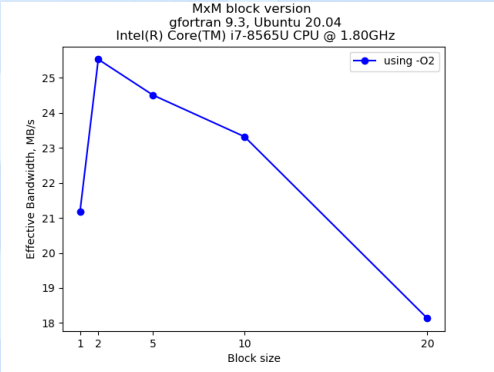



```
for (int i0 = 0; i0 < n; i0 += tileSize) {
  for (int j0 = 0; j0 < n; j0 += tileSize) {
    for (int k0 = 0; k0 < n; k0 += tileSize) {
      for (int i = i0; i < i0 + tileSize; i++) {
        for (int k = k0; k < k0 + tileSize; k++) {
          for (int j = j0; j < j0 + tileSize; j++) {
            C[i * n + j] += A[i * n + k] * B[k * n + j];
}}}}}}
```

```
arithmetic cost  :: b**3*p**3*(ADD + MUL)
memory cost      :: b**2*p**2*(2*READ*p + WRITE)
CI               :: b*(ADD + MUL)/(2*READ) ~= b
```

mxm block version: CPU time vs. Bandwidth





Principles of good data locality

- Code accesses contiguous, stride-one memory addresses
  - data are always fetched in cache lines which include neighbors
  - inner loops are instantly vectorizable via `SSE`, `AVX`, or `AVX-512`
- Code emphasizes cache reuse
  - when multiple operations on a data item are grouped together, the item remains in cache, where access is much faster than `RAM`
- Data are aligned on important boundaries (e.g., doublewords)
  - items don't straddle boundaries, so efficient one-shot access is possible
  - it is a precondition for fetching data as a single vector, single cache line, etc.