# Distributed Computing and Introduction to High Performance Computing

Ahmed Ratnani[1]

[1]Mohammed VI Polytechnic University, Benguerir, Morocco

# 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`

## 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.

## 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`.

# 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

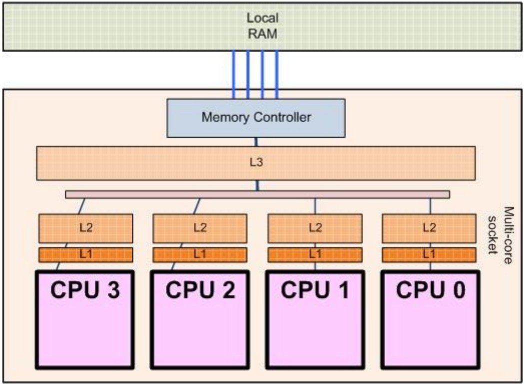
It is the ratio between arithmetic complexity (or cost) and memory complexity (cost).

- 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**.

# Computational Intensity

## Modern architecture (CPU)




**Typical sizes**

- `RAM` $\sim 4\ GB - 128\ GB$ even higher on servers
- `L3` $\sim 4\ MB - 50\ MB$
- `L2` $\sim 256\ KB - 8\ MB$
- ➠ Holds data that is likely to be accessed by the CPU
- `L1` $\sim 256\ KB$
- ➠ Instruction and Data cache

**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

## Avoiding communication

- Running time of an algorithm is sum of 3 terms:
  - `N_flops * time_per_flop`
  - `N_words / bandwidth`
  - `N_messages * latency`
- It important to notice that
  `time_per_flop << 1/ bandwidth << latency`
- **Avoiding communication algorithms** come with a significant speedup
- Some examples
  - Up to 12x faster for 2.5D matmul on 64K core IBM BG/P
  - Up to 3x faster for tensor contractions on 2K core Cray XE/6
  - Up to 6.2x faster for All-Pairs-Shortest-Path on 24K core Cray CE6
  - Up to 2.1x faster for 2.5D LU on 64K core IBM BG/P
  - Up to 11.8x faster for direct N-body on 32K core IBM BG/P
  - Up to 13x faster for Tall Skinny QR on Tesla C2050 Fermi NVIDIA GPU

# Computational Intensity

## mxm example: naive version

```
for i in range(0, n):
    for j in range(0, n):
        for k in range(0, n):
            z[i,j] = z[i,j] + x[i,k]*y[k,j]
```

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

# Two Memory level model

# Two Memory level model

## mxm example: Using block version (cache optimization)

```
p = n/b
x = zeros((n,n))
y = zeros((n,n))
z = zeros((n,n))

r = zeros((b,b))
u = zeros((b,b))
v = zeros((b,b))

for i in range(0, n, b):
    for j in range(0, n, b):
        for k1 in range(0, b):
            for k2 in range(0, b):
                r[k1,k2] = z[i+k1,j+k2]
        for k in range(0, n, b):
            for k1 in range(0, b):
                for k2 in range(0, b):
                    u[k1,k2] = x[i+k1,k+k2]
                    v[k1,k2] = y[k+k1,j+k2]
            for ii in range(0, b):
                for jj in range(0, b):
                    for kk in range(0, b):
                        r[ii,jj] = r[ii,jj] + u[ii,kk]*v[kk,jj]
        for k1 in range(0, b):
            for k2 in range(0, b):
                z[i+k1,j+k2] = r[k1,k2]
```

# Two Memory level model

## mxm example: Using block version (cache optimization)

**Without any additional information**

```
arithmetic cost :: DIV + b**3*p**3*(ADD + MUL)
memory cost :: 2*READ + 3*WRITE + b**2*p**2*(2*READ + 2*WRITE + p*(2*READ + 2*WRITE + b*(3*READ + WRITE)))
computational intensity :: (ADD + MUL)/(3*READ + WRITE)
```

**Assuming two level of memories**:

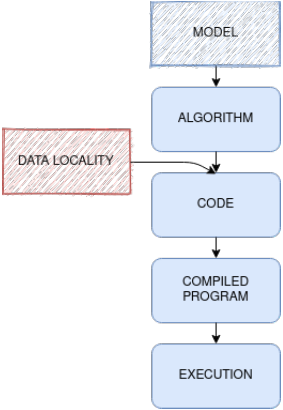
- the fast memory represents the L2 cache
- variables that are supposed to live in the cache (`r, u, v`)

```
arithmetic cost :: DIV + b**3*p**3*(ADD + MUL)
memory cost :: 2*READ + 3*WRITE + b**2*p**2*(2*READ*p + READ + WRITE)
computational intensity :: b*(ADD + MUL)/(2*READ)
```

# 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
- In the previous example, we saw that it is possible to increase the computational intensity by rewriting the `mxm` using a blocking version
- ➠ Let's understand how things are actually handled



**How do we access Data?**

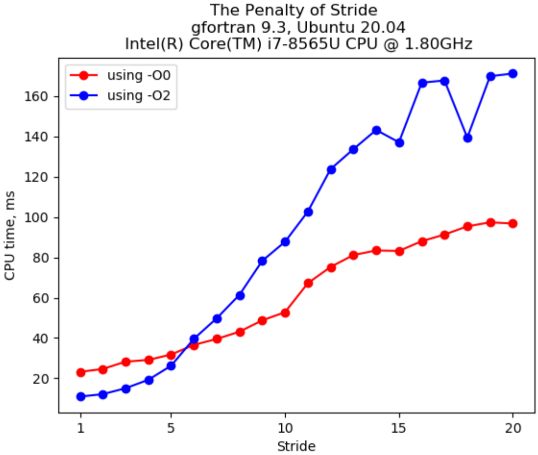
- Not only there are different hierarchies of memories, each one, with a specific memory cost
- We need also to think about how do we access to Data
- We should always arrange your data so that the elements are accessed with unit (1) stride
- The following **example** will convince you that the penalty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

```
do i=1, N*i_stride,i_stride
  mean = mean + a(i)
end do
```

- 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

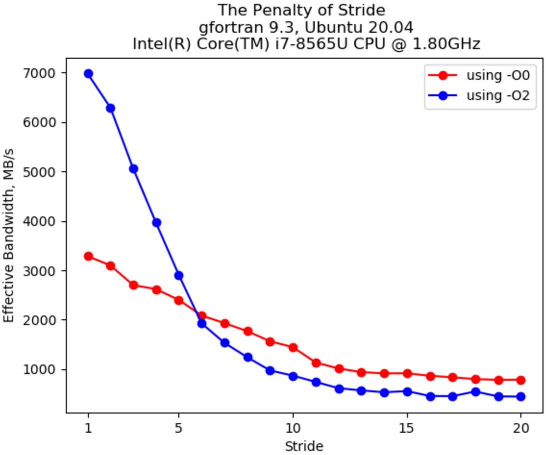
# Data Locality

## The Penalty of Stride: CPU time



# Data Locality

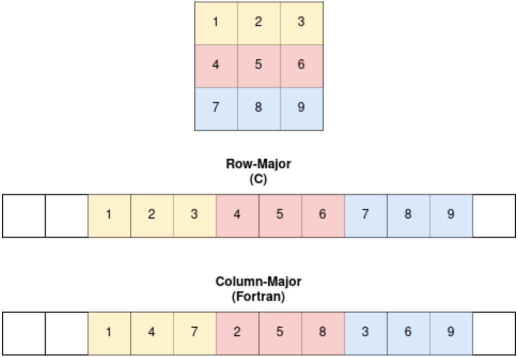
## The Penalty of Stride: Bandwidth



- High Dimensional Arrays are stored as a contiguous sequence of elements
  - `Fortran` uses Column-Major ordering
  - `C` uses Row-Major ordering

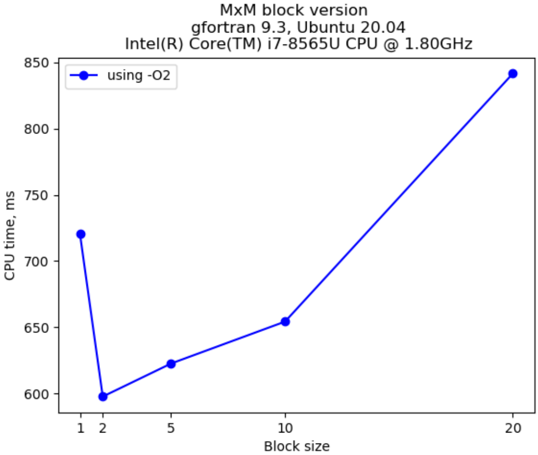
**mxm in Fortran** N = 1000

- Naive version: CPU-time 1660.6 (msec)
- Transpose version: CPU-time 1139.8 (msec)



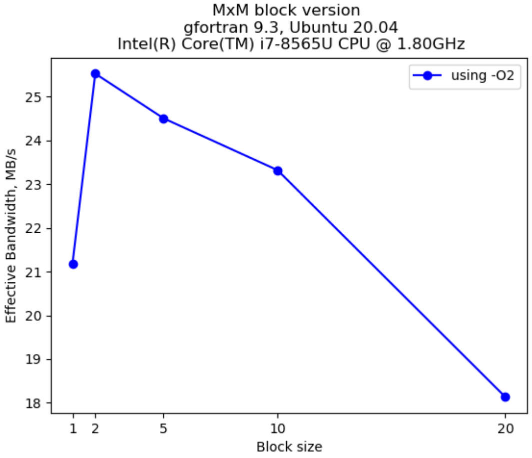
# Data Locality

mxm block version: CPU time



# Data Locality

## mxm block version: Bandwidth



# Data Locality

## 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.

# Assignments

- Look for the cost (FLOPS) of the usual arithmetic operations `ADD`, `MUL`, `SUB`, `DIV` on different CPUs
- Look for the cost (FLOPS) of the usual math functions `SIN`, `COS`, `EXP`, `...`
- Compute the Computational Intensity of the `mxv` Matrix-Vector product using the naive version and a blocking version. Implement both verions
- Compute the cost (FLOPS) of evaluating a Polynomial of degree $p$ using a naive version and the Horner algorithm. Implement both verions