
innovate
achieve
lead
BITS Pilani
WILP
AIML CLZG516
ML System Optimization

innovate achieve lead

BITS Pilani

WILP

AIML CLZG516
ML System Optimization

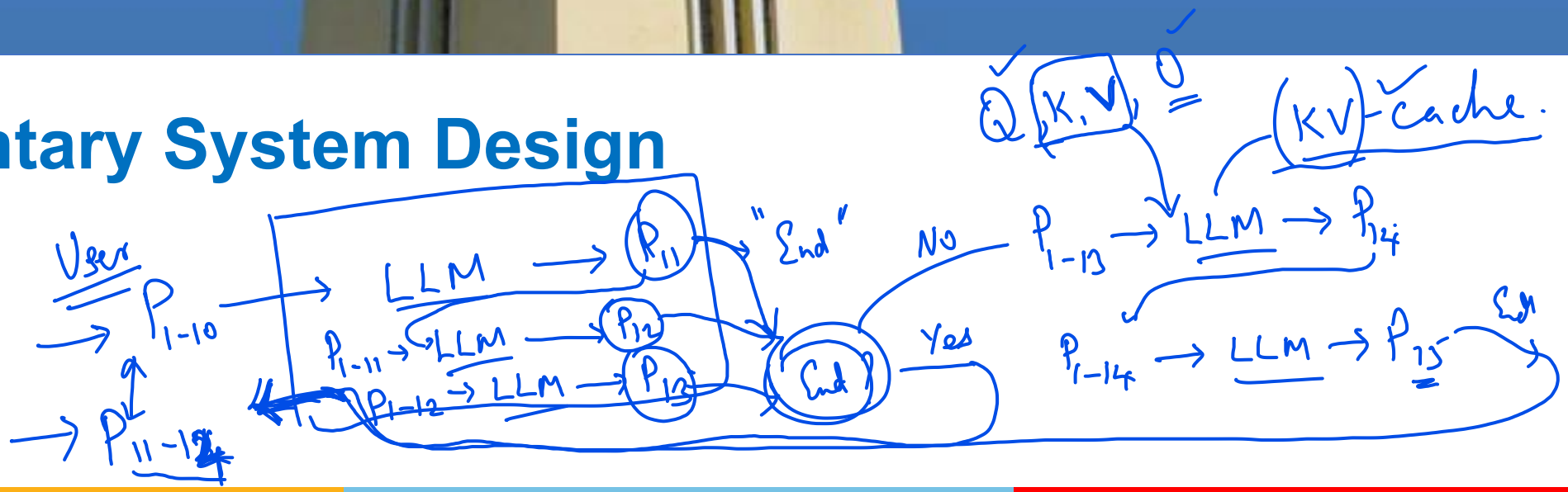
Session 9

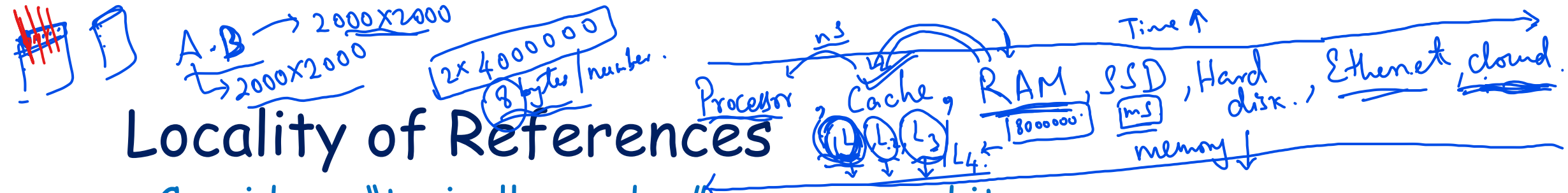
Towards Optimizing Neural Nets

- System / Program Optimization Techniques
- Locality-aware Programs

Speculative decoding.

Elementary System Design





Locality of References

- Consider a "typically random" program and its memory accesses:
  - *Locations* (or *variables* or *objects*) that are accessed are ***likely to be accessed*** repeatedly
    - e.g. instructions and data *in the body of a loop*
    - e.g. instructions and data in a function / procedure

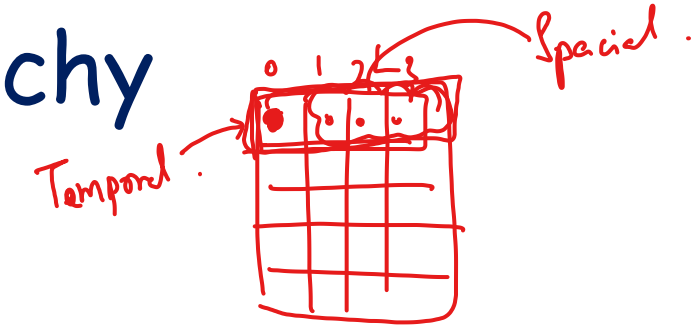
    } Temporal locality
  - Locations nearby - other locations that are accessed - are likely to be accessed
    - e.g. elements of an array (or a matrix)
    - e.g. instructions in a sequence

    } Spatial locality
- In short, the locus of memory accesses (*by a program*) *is* small:
  - the locus refers to any small window of time during program execution.

Locality and System Design

- Locality of Reference is an observation about "typical programs".
  - Exercise:
    - Design counter-example programs (that violate this observation).
- (Computing) System Designers use this a heuristic in designing and optimizing systems:
  - Memory hierarchy and Data Movement techniques

Locality and Memory hierarchy



- Different levels of memory
  - Large (, cheap, and slow)

  to
  - Small (, costly, and fast):
    - Registers (inside the processor)
    - Cache (between the processor and RAM)
    - RAM addressable but volatile
    - Hard Disk (non-volatile, semi-sequential access)
    - Remote (over the network)/ cloud

L1, L2, L3 | L4

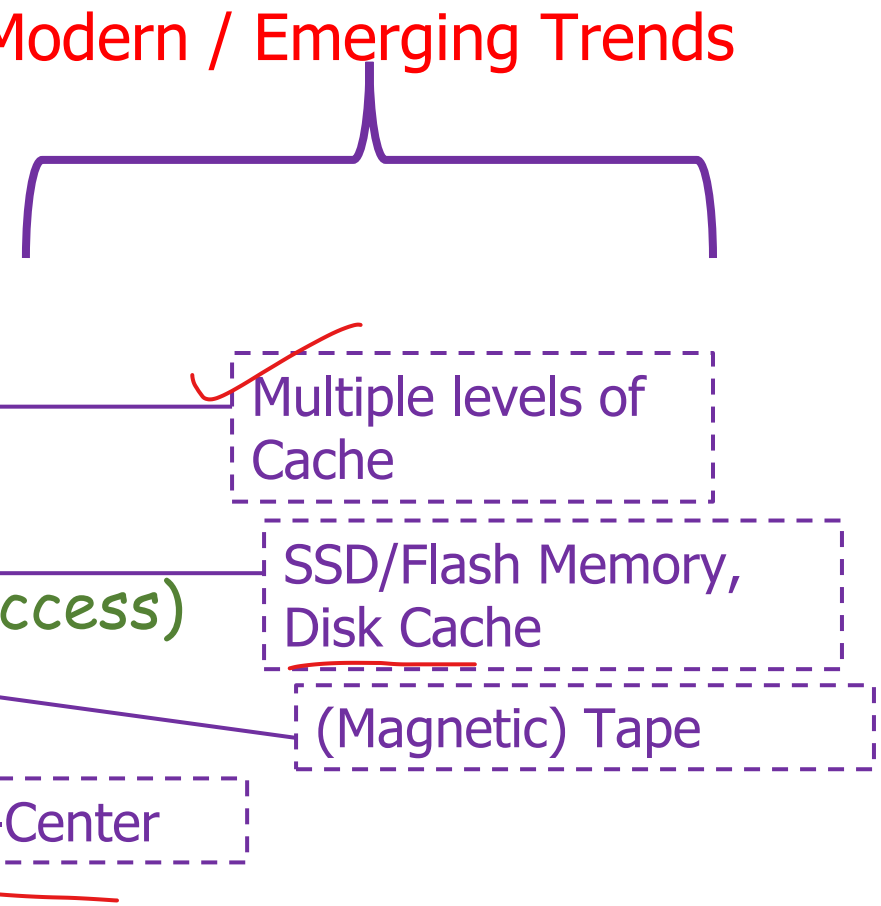
Cost Amortization

- Exercise:
  - Find out and understand the access times and ratios

Locality and Memory hierarchy

- Different levels of memory
  - Large (, cheap, and slow)

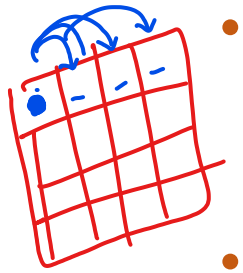
  to
  - Small (, costly, and fast):
    - Registers (inside the processor)
    - Cache (between the processor and RAM)
    - RAM addressable but volatile
    - Hard Disk (non-volatile, semi-sequential access)
    - Remote (over the network)



Exercise: Understand and extrapolate the trends: what issues are they addressing and how?

Locality and Data Movement (up the Hierarchy)

- (Computing) System Designers design and optimize Data Movement techniques



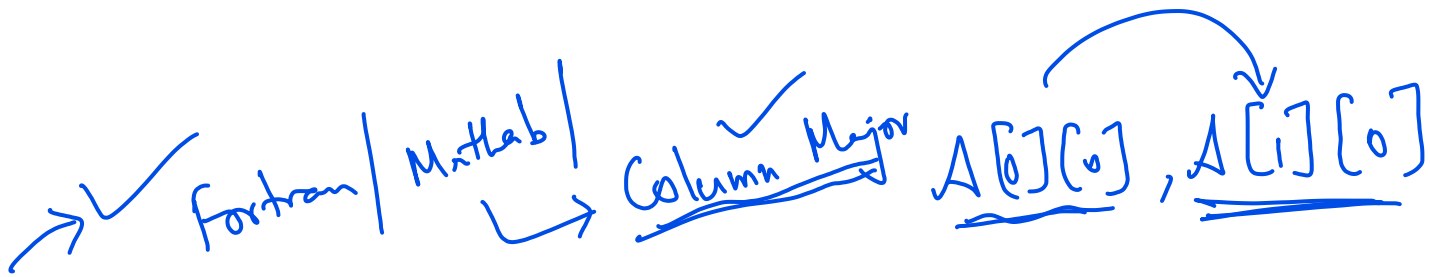
  - Caching (addresses temporal locality):
    - "Retain" data – *close to the processor* - that have been accessed recently.
  - Pre-fetching (addresses spatial locality):
    - "Move data" – *closer to the processor* - by anticipating their access
  - Blocked data access and buffering
    - Access data in bulk to amortize (setup) cost in sequential accesses:
      - e.g. seek time, disk rotation time in hard disk
      - e.g. spooling time in tapes
      - e.g. latency in network access

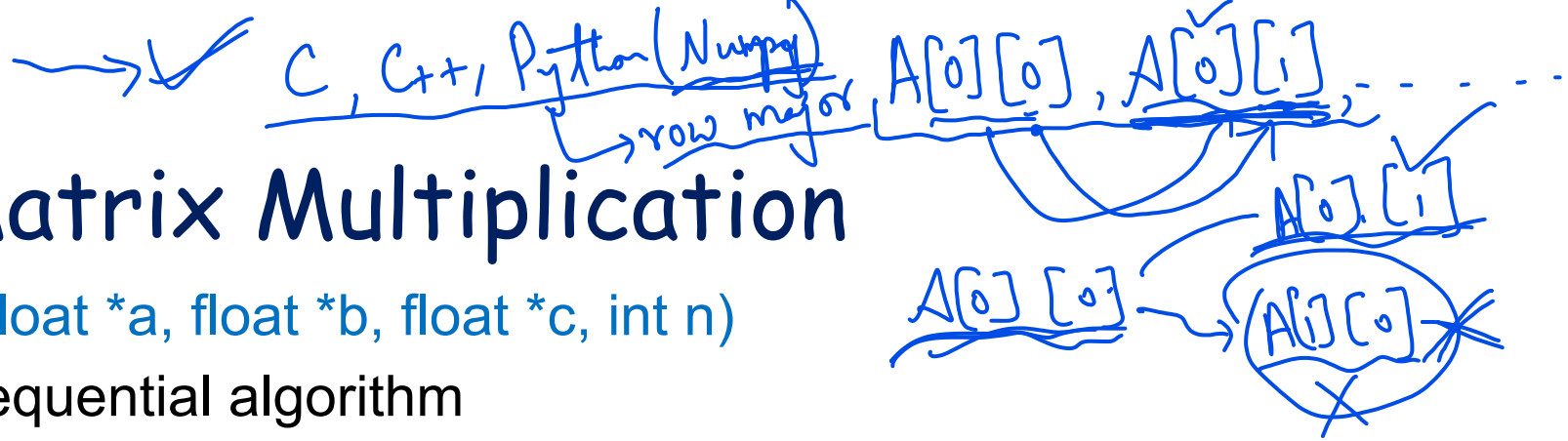
innovate achieve lead

BITS Pilani

WILP

Program Design and Optimization





Example: Matrix Multiplication

```
void matMult_IJK (float *a, float *b, float *c, int n)
{       // Classic Sequential algorithm
        // n x n row-major matrices a and b.
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    {
        float temp = 0;
        for (int k = 0; k < n; k++)
        temp += a[i*n+k]*b[k*n+j];
        c[i*n+j] = temp;
    }
}
```

Inner product of i^{th} Row of a with k^{th} column of B

a[i*n+0],a[i*n+1],a[i*n+2],..

b[0*n+j],b[1*n+j],a[2*n+j],..

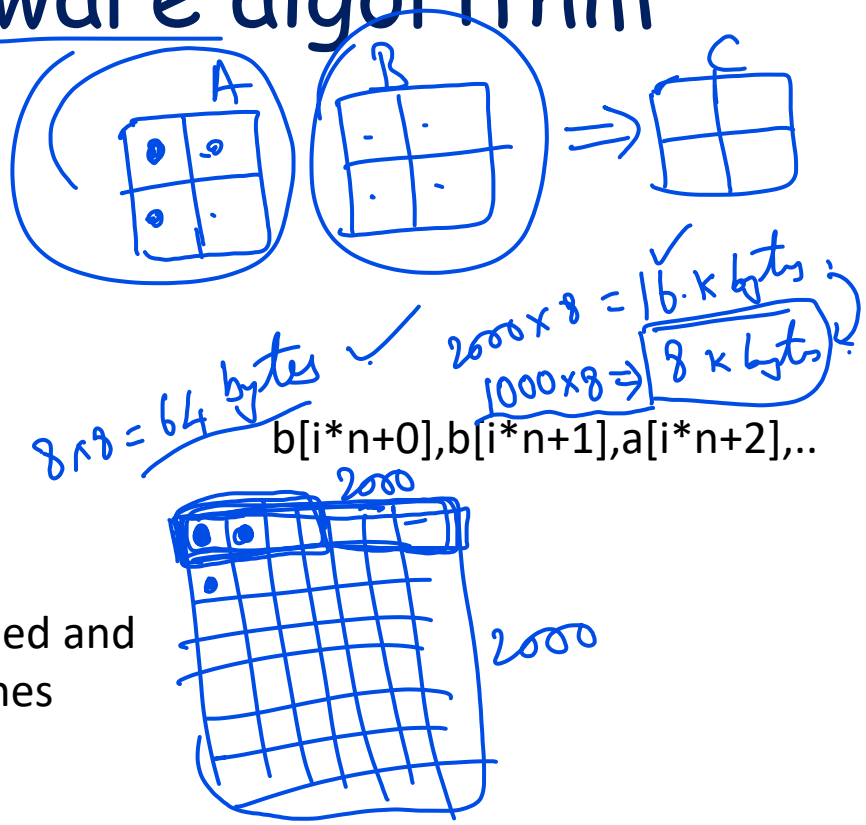
c[i*n+j],b[i*n+j],a[i*n+j],..

Consider the sequence of memory accesses: for **a**, **b**, and **c**

Matrix-multiplication: Cache-aware algorithm

```
void matMult_IKJ (float *a, float *b, float *c, int n)
{   // Sequential cache-aware algorithm
    // n x n row-major matrices a and b
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j < n; j++) c[i*n+j] = 0;
      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];
    }
```

Each c[i,j] is cached and accessed n*n times



How does the sequence of memory accesses: (for **a**, **b**, and **c**) compare with those from the previous version?

Running time Comparison

Running Time of the Matrix Multiplication methods in seconds (size n*n)
[on an Intel Xeon Quad-Core using only one core]

Method	n=256	n=512	n=1024	n=2048	n=4096
ijk	0.11	0.93	10.41	~450	~4026
ikj	0.14	1.12	8.98	~73	~581
ijk / ikj	0.80	0.83	1.16	6.19	6.93

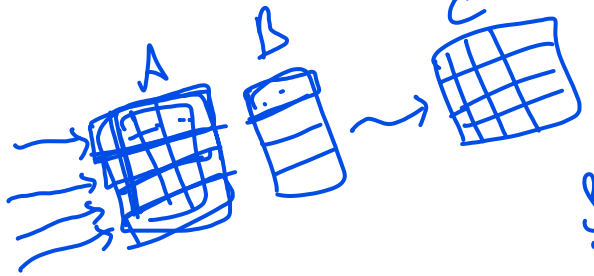
Ratio of the running times

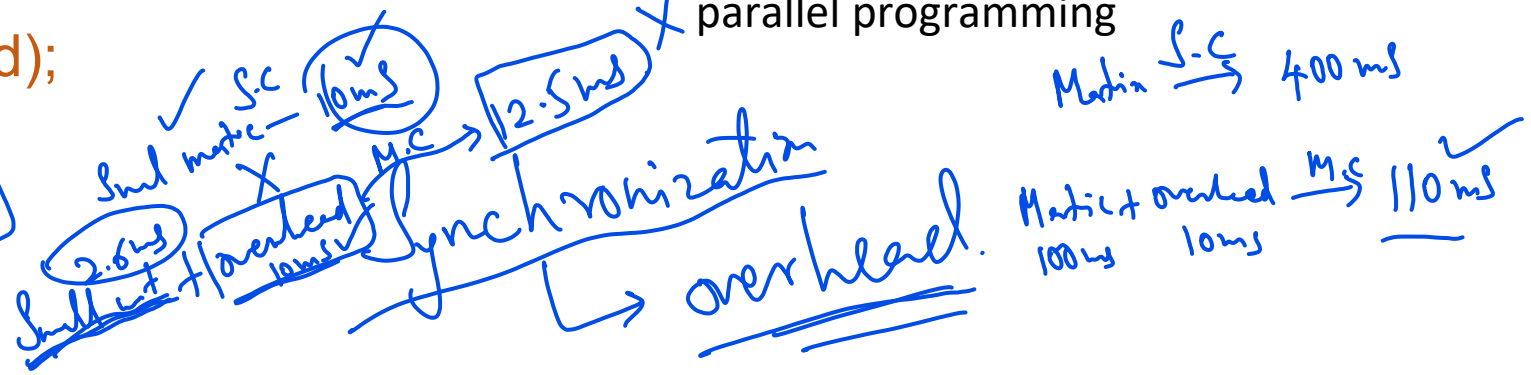
Matrix-multiplication: Multi-core version

```
void MultiCore_MM (float *a, float *b, float *c, int n, int t)
{       // Multicore algorithm to multiply two
        // n x n row-major matrices a and b.
    // t is the number of threads
#pragma omp parallel shared (a, b, c, n, t)
{
int tid = omp_get_thread_num(); // thread ID
matMult_IKJ_sc(a, b, c, n, t, tid);
}
}
```

#pragma instructs the compiler to use OpenMP library

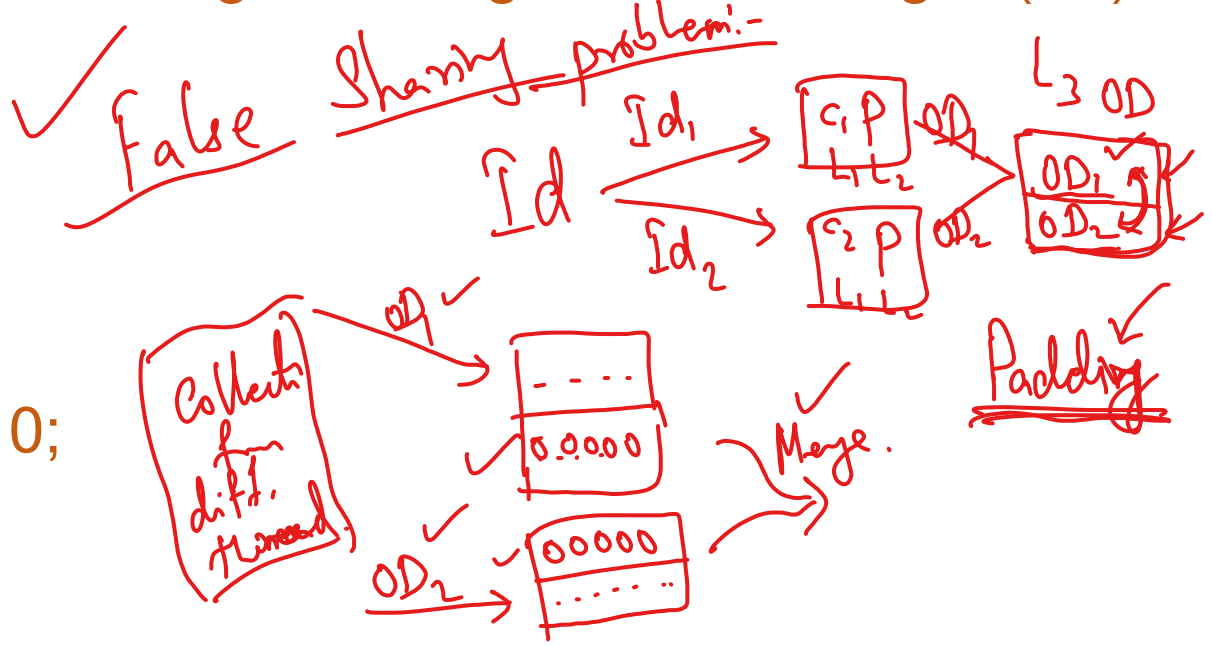
SPMD programming i.e. data parallel programming





```
void matMult_IKJ_SC(float *a, float *b, float *c, int n, int t, int tid)
{           // compute an (n/t) x n sub-matrix of c; t is #threads; thread ID (0 <= tid < t )
    int height = n/t;
    int offset = height*n*tid; // offset = 0, height*n, height*2*n, … height*(t-1)*n
    a += offset;
    c += offset;

    for (int i = 0; i < height; i++)
    {
    for (int j = 0; j < n; j++) c[i*n+j] = 0;
    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];
    }
}
```



Running Time of the Matrix Multiplication method in seconds (size n*n)

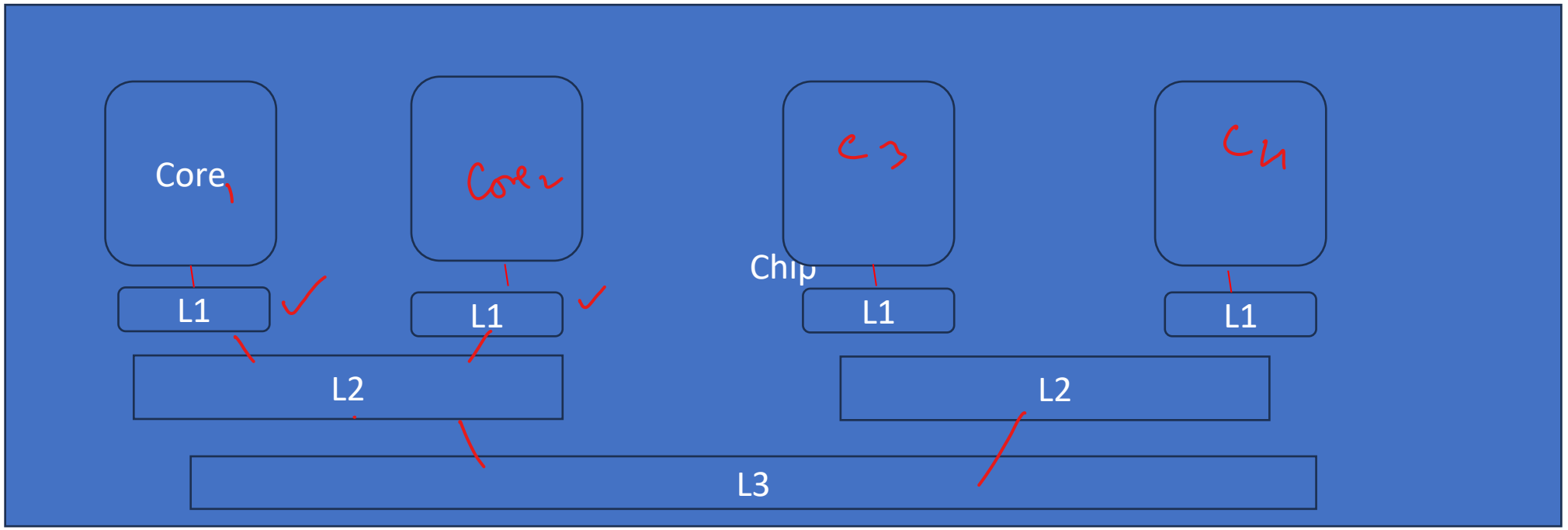
[on an Intel Xeon Quad-Core using t threads]

t | n --> | | \/	256	512	1024	2048	4096
1	0.14	1.12	8.98	72.67	~581
2	0.07	0.56	4.50	36.39	~291
4	0.04	0.28	2.26	18.21	~146
8	0.04	0.28	2.28	18.42	~146

T1/T2	2	2	2	2	2
T1/T4	4	3.97	3.98	3.99	3.97
T1/T8	3.89	4.0	3.94	3.95	3.95

Speedups: Tj – Running time with j threads

Typical *Intel Multi-core Architecture*



L1 has I-cache and D-cache