Lecture 10: Memory Hierarchy
-- Memory Technology and Principal of Locality

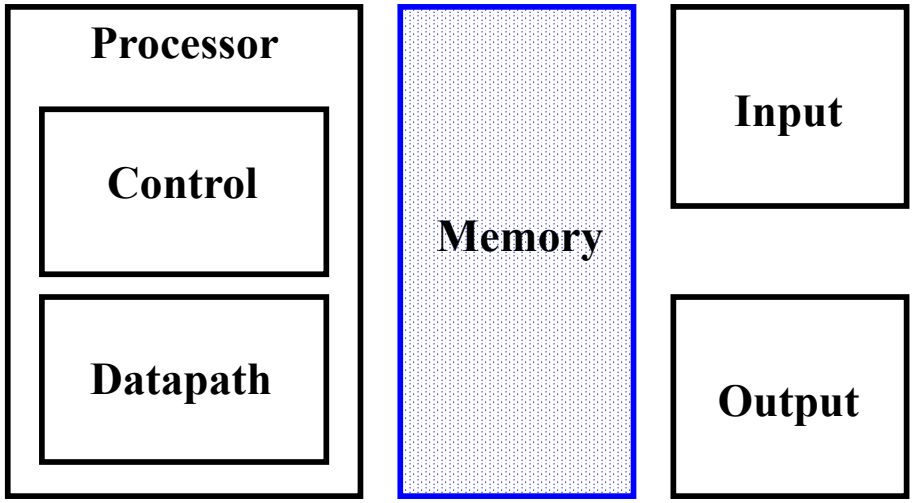
CSCE 513 Computer Architecture

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
https://passlab.github.io/CSCE513

Topics for Memory Hierarchy

- Memory Technology and Principal of Locality
  - CAQA: 2.1, 2.2, B.1
  - COD: 5.1, 5.2
- Cache Organization and Performance
  - CAQA: B.1, B.2
  - COD: 5.2, 5.3
- Cache Optimization
  - 6 Basic Cache Optimization Techniques
    - CAQA: B.3
  - 10 Advanced Optimization Techniques
    - CAQA: 2.3
- Virtual Memory and Virtual Machine
  - CAQA: B.4, 2.4; COD: 5.6, 5.7
  - Skip for this course

The Big Picture: Where are We Now?

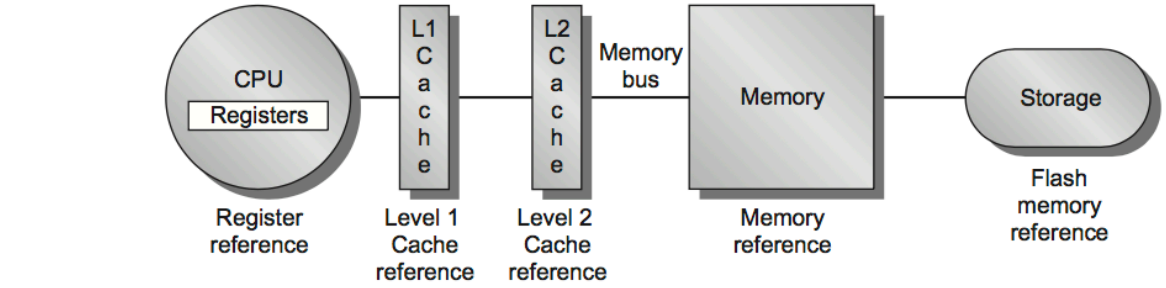


- Memory system
  - Supplying data on time for computation (speed)
  - Large enough to hold everything needed (capacity)

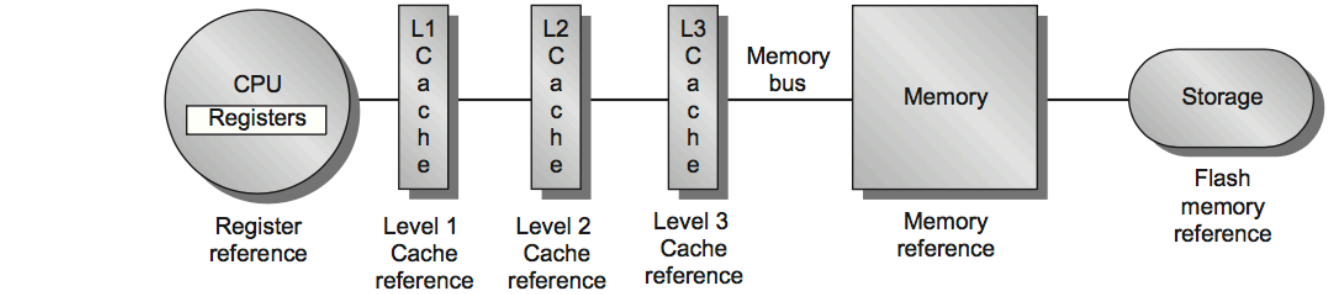
Overview

- Programmers want unlimited amounts of memory with low latency
- Fast memory technology is more expensive per bit than slower memory
- Solution: organize memory system into a hierarchy
  - Entire addressable memory space available in largest, slowest memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Temporal and spatial locality insures that nearly all references can be found in smaller memories
  - Gives the allusion of a large, fast memory being presented to the processor

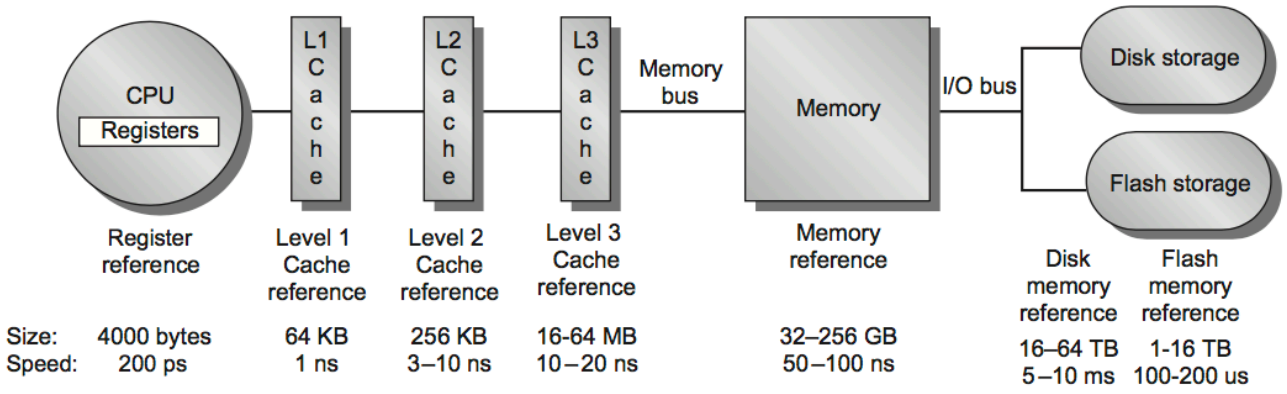
Memory Hierarchy



(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

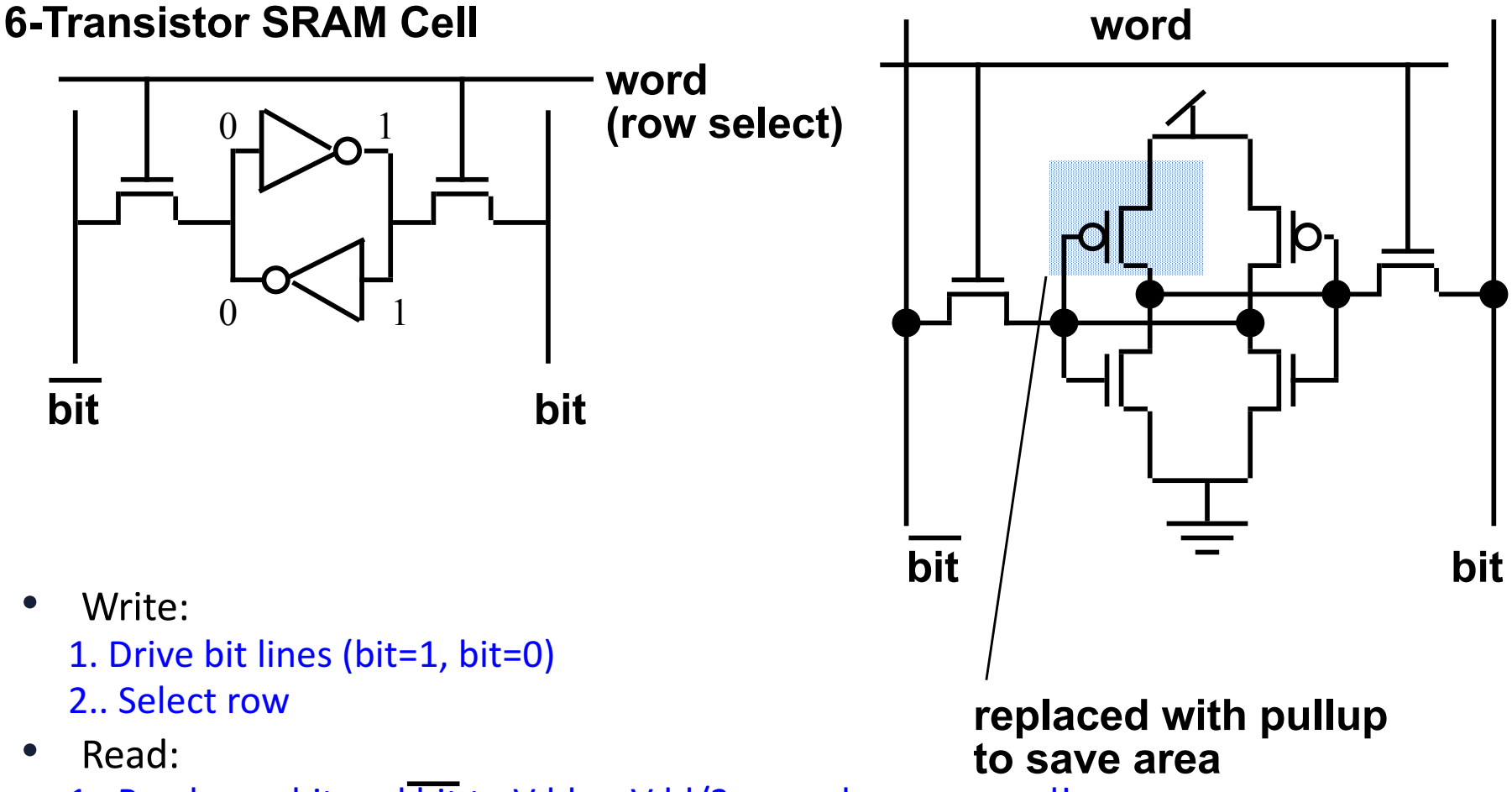
Memory Technology

- **Random Access: access time is the same for all locations**
- **DRAM: Dynamic Random Access Memory**
  - **High density, low power, cheap, slow**
  - **Dynamic: need to be "refreshed" regularly**
  - **50ns – 70ns, $20 – $75 per GB**
- **SRAM: Static Random Access Memory**
  - **Low density, high power, expensive, fast**
  - **Static: content will last "forever"(until lose power)**
  - **0.5ns – 2.5ns, $2000 – $5000 per GB**
- Magnetic disk
  - 5ms – 20ms, $0.20 – $2 per GB

Ideal memory:

- **Access time of SRAM**
- **Capacity and cost/GB of disk**

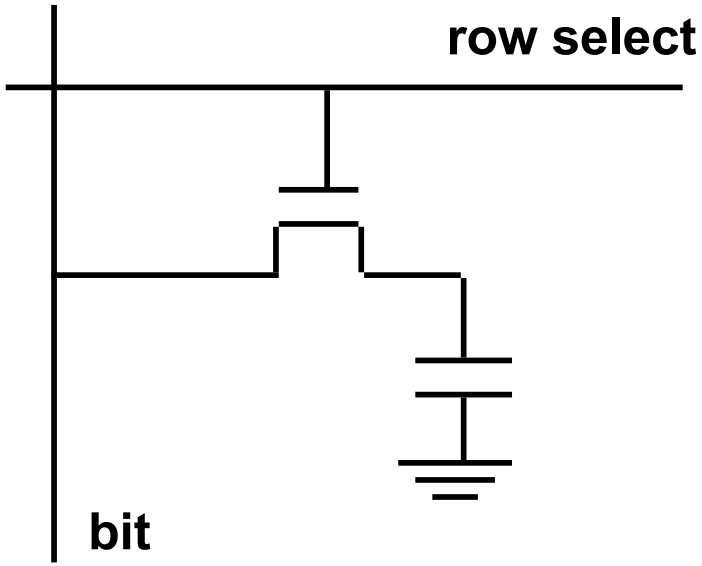
Static RAM (SRAM) 6-Transistor Cell – 1 Bit



- Write:
  1. Drive bit lines (bit=1, bit=0)
  2.. Select row
- Read:
  1. Precharge bit and $\overline{bit}$ to Vdd or Vdd/2 => make sure equal!
  2.. Select row
  3. Cell pulls one line low
  4. Sense amp on column detects difference between bit and $\overline{bit}$

Dynamic RAM (DRAM) 1-Transistor Memory Cell

- Write:
  - 1. Drive bit line
  - 2. Select row
- Read:
  - 1. Precharge bit line to Vdd
  - 2. Select row
  - 3. Cell and bit line share charges
    - Very small voltage changes on the bit line
  - 4. Sense (fancy sense amp)
    - Can detect changes of ~1 million electrons
  - 5. Write: restore the value
- Refresh
  - 1. Just do a dummy read to every cell.



Performance: Latency and Bandwidth

- Performance of Main Memory:
  - Latency: Cache Miss Penalty
    - *Access Time*: time between request and word arrives
    - *Cycle Time*: time between requests
  - Bandwidth: I/O & Large Block Miss Penalty (L2)
- Main Memory is *DRAM :* Dynamic Random Access Memory
  - Needs to be refreshed periodically (8 ms)
  - Addresses divided into 2 halves (Memory as a 2D matrix):
    - *RAS* or *Row Access Strobe* and *CAS* or *Column Access Strobe*
- Cache uses *SRAM :* Static Random Access Memory
  - No refresh (6 transistors/bit vs. 1 transistor)
    Size: DRAM/SRAM *4-8*
    Cost/Cycle time: SRAM/DRAM *8-16*

Stacked/Embedded DRAMs

- Stacked DRAMs in same package as processor
  - High Bandwidth Memory (HBM)

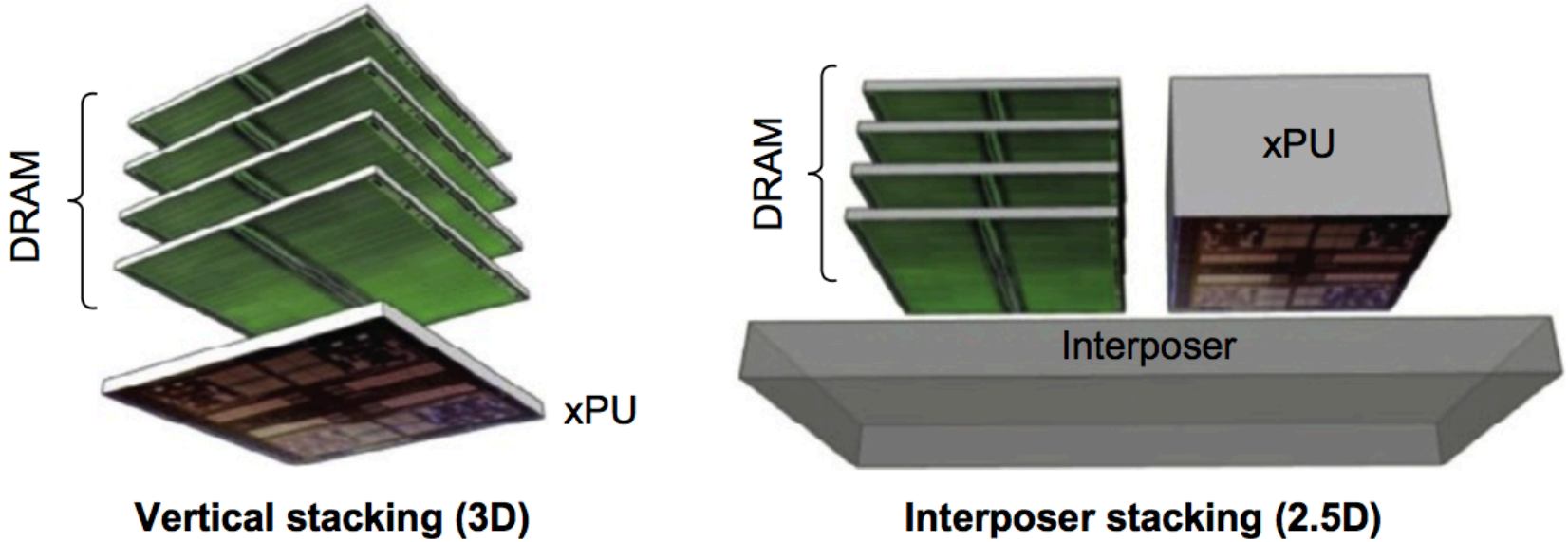


Figure 2.7 Two forms of die stacking. The 2.5D form is available now. 3D stacking is under development and faces heat management challenges due to the CPU.

Flash Memory

- Type of EEPROM
- Types: NAND (denser) and NOR (faster)
- NAND Flash:
  - Reads are sequential, reads entire page (.5 to 4 KiB)
  - 25 us for first byte, 40 MiB/s for subsequent bytes
  - SDRAM: 40 ns for first byte, 4.8 GB/s for subsequent bytes
  - 2 KiB transfer: 75 uS vs 500 ns for SDRAM, 150X slower
  - 300 to 500X faster than magnetic disk

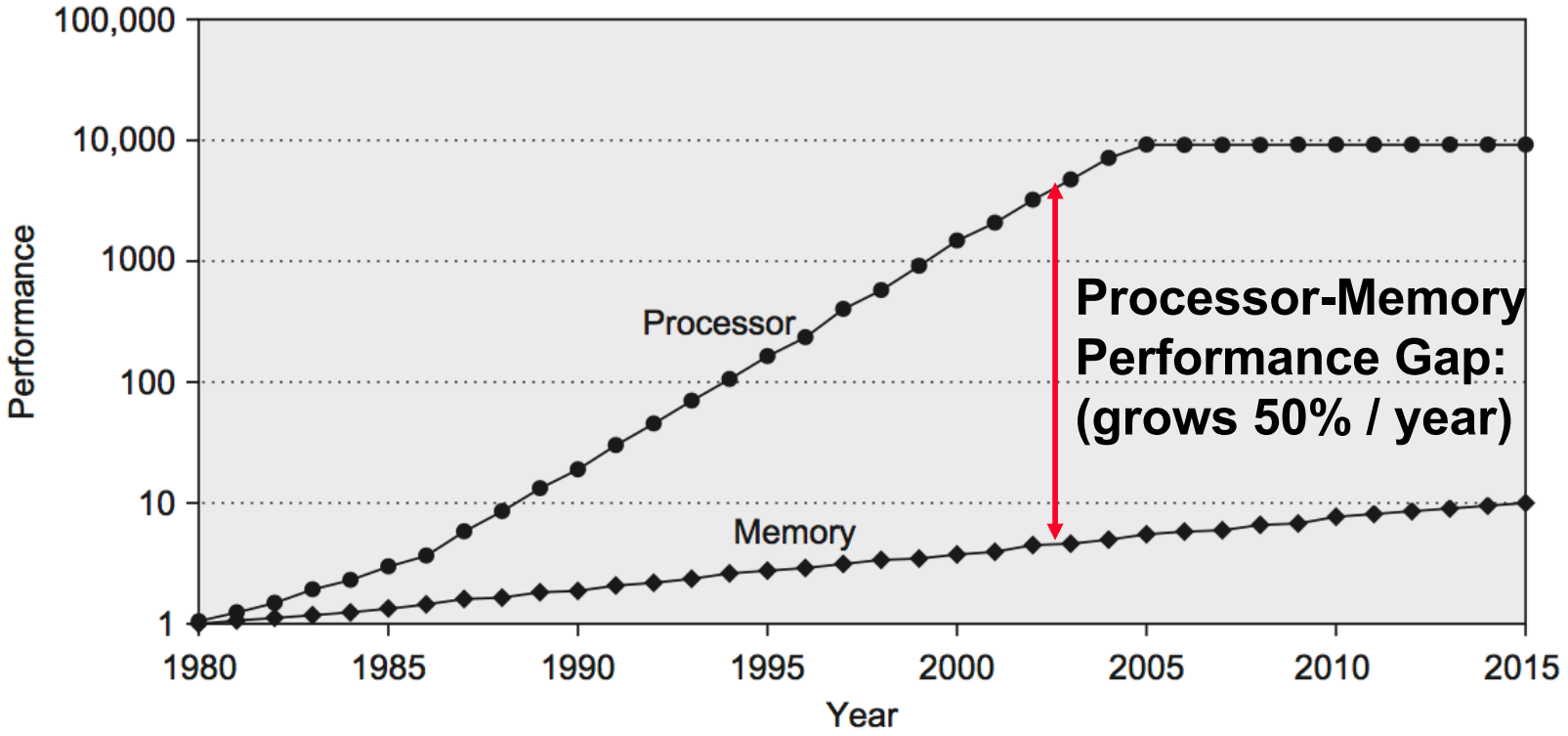
NAND Flash Memory

- Must be erased (in blocks) before being overwritten
- Nonvolatile, can use as little as zero power
- Limited number of write cycles (~100,000)
- $2/GiB, compared to $20-40/GiB for SDRAM and $0.09 GiB for magnetic disk

- Phase-Change/Memrister Memory
  - Possibly 10X improvement in write performance and 2X improvement in read performance

CPU-Memory Performance Gap: Latency

CPU-DRAM Memory Latency Gap → Memory Wall



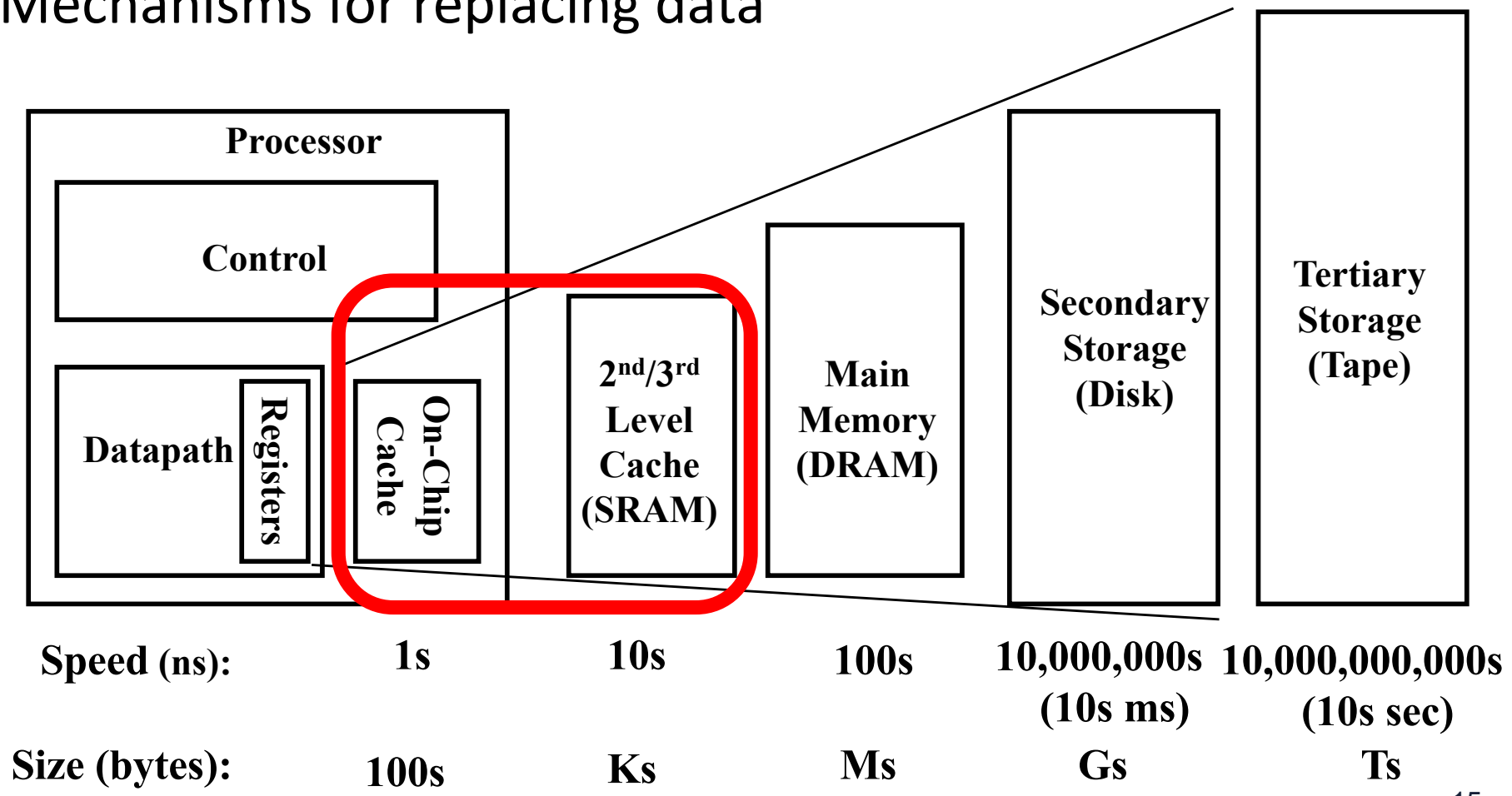
of the processor-DRAM performance gap. The memory baseline is 64 KiB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.4 on page 88). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and only small improvements in processor performance (on a per-core basis) between 2005 and 2015. As you

CPU-Memory Performance Gap: Bandwidth

- Memory hierarchy design becomes **more crucial with recent multi-core processors:**
- Aggregate peak bandwidth grows with # cores:
  - Intel Core i7 can generate two references per core per clock
  - Four cores and 3.2 GHz clock
    - 25.6 billion 64-bit data references/second +
    - 12.8 billion 128-bit instruction references/second
    - = 409.6 GB/s!
  - DRAM bandwidth is only 8% of this (34.1 GB/s)
- Requires:
  - Multi-port, pipelined caches
  - Two levels of cache per core
  - Shared third-level cache on chip

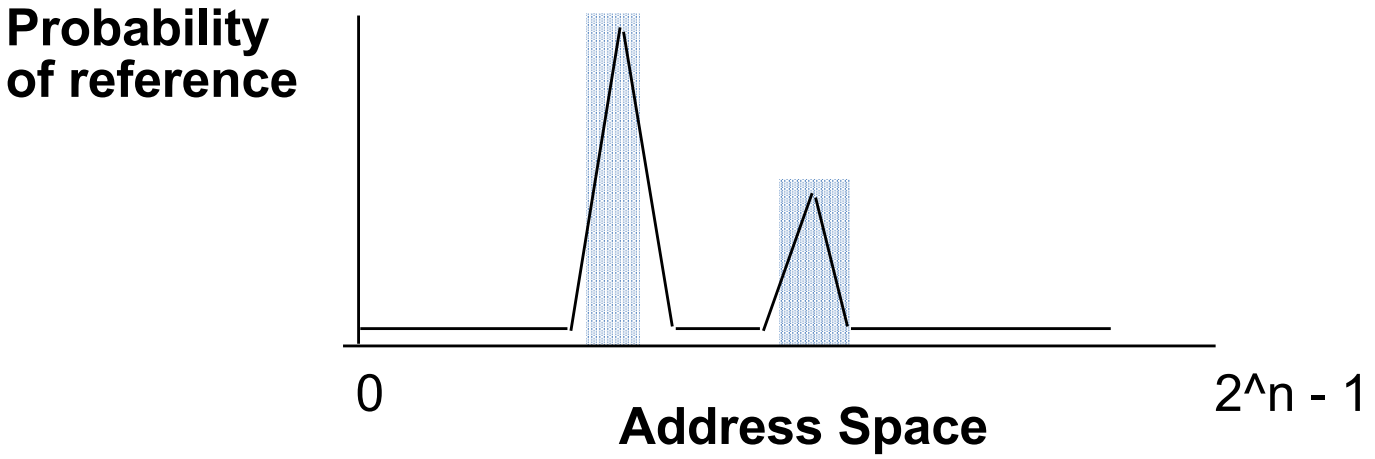
Memory Hierarchy

- **Keep most recent accessed data and its adjacent data in the smaller/faster caches that are closer to processor**
- Mechanisms for replacing data



Why Hierarchy Works

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.



The Principle of Locality

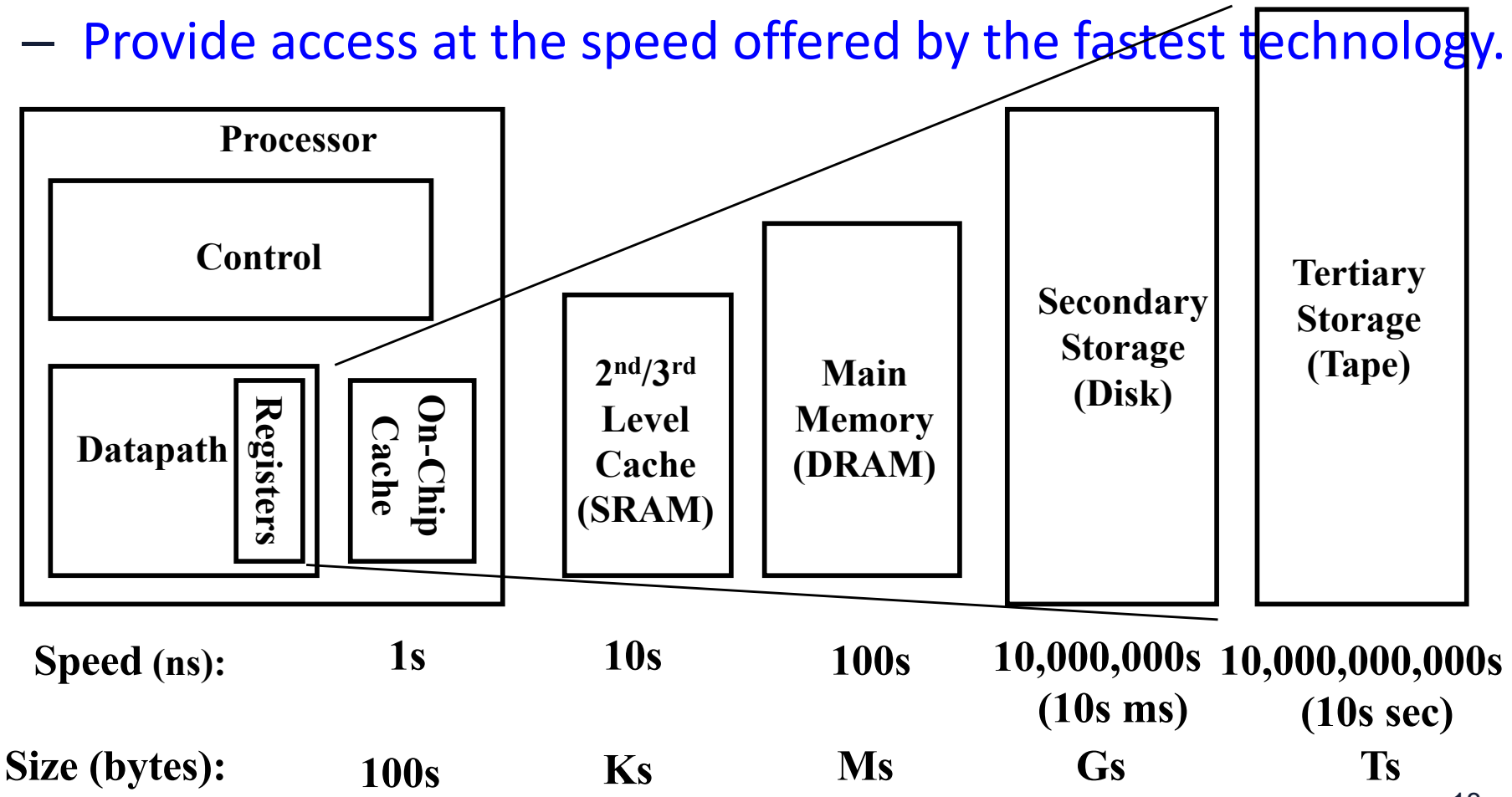
- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves
- **Spatial locality:** Items with nearby addresses tend to be referenced close together in time
- **Temporal locality:** Recently referenced items are likely to be referenced in the near future

```
sum = 0;
for(i=0; i<n; i++)
  sum += a[i];
return sum;
```

- **Data**
  - **Reference array elements in succession (stride-1 reference pattern): Spatial Locality**
  - **Reference sum each iteration: Temporal Locality**
- **Instructions**
  - **Reference instructions in sequence: Spatial Locality**
  - **Cycle through loop repeatedly: Temporal Locality**

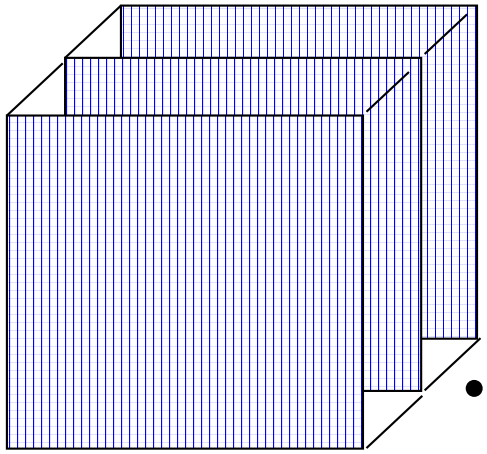
Memory Hierarchy of a Computer System

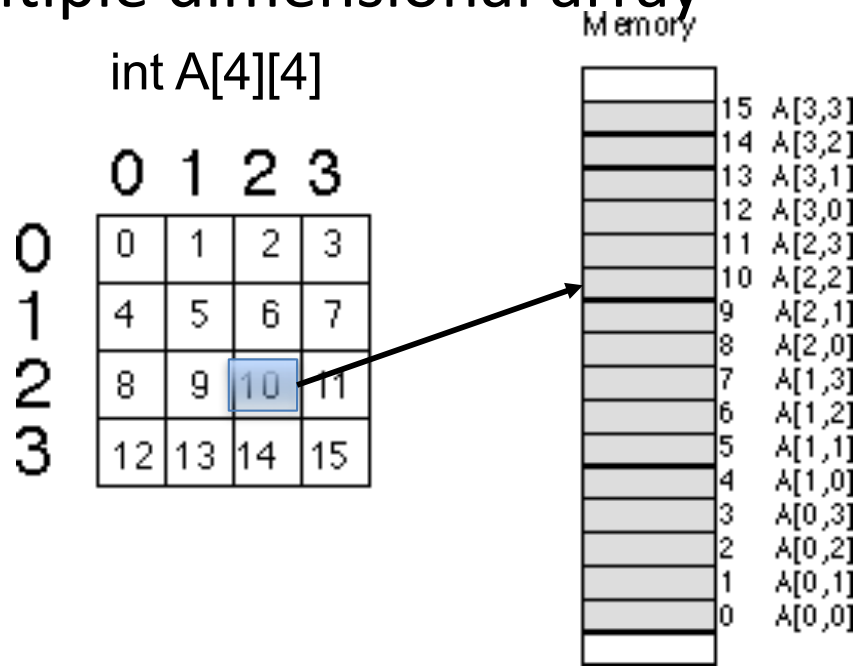
- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



Vector/Matrix and Array in C

- C has row-major storage for multiple dimensional array
  - A[2,2] is followed by A[2,3]
- 3-dimensional array
  - B[3][100][100]





- **Stepping through columns in one row:**
  for (i=0; i<4; i++) sum += A[0][i];
  accesses successive elements
- **Stepping through rows in one column:**
  for (i=0; i<4; i++) sum += A[i][0];
  Stride-4 access

Locality Example

- **Claim: Being able to look at code and *get qualitative sense* of its locality is key skill for professional programmer**

- Question: Does this function have good locality?

```
int sumarrayrows(int a[M][N]){
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- Question: Does this function have good locality?

```
int sumarraycols(int a[M][N]){
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

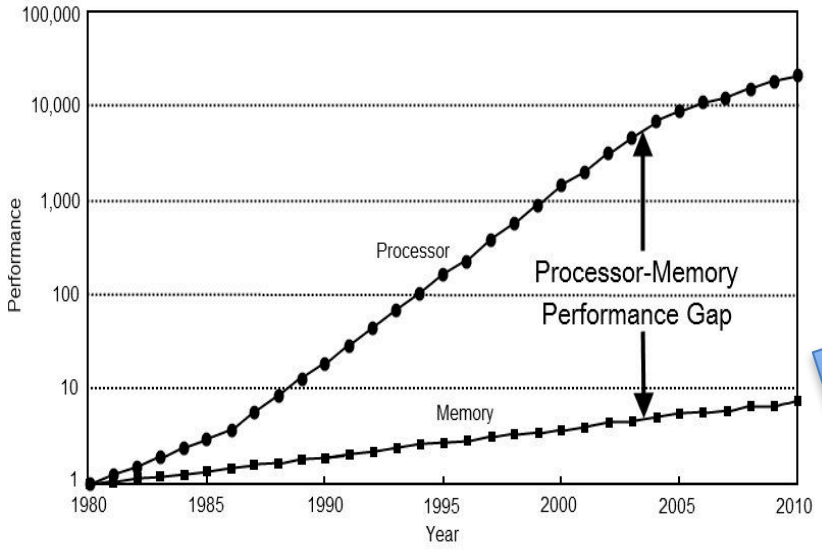
Locality Example

- Question: Can you permute the loops so that the function scans the 3-d array `a[]` with a stride-1 reference pattern (and thus has good spatial locality)?

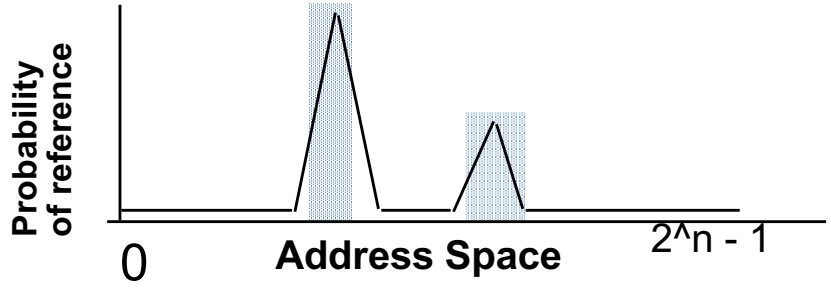
```
int sumarray3d(int a[M][N][N]) {
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

Review: Memory Technology and Hierarchy

Technology Challenge: *Memory Wall*



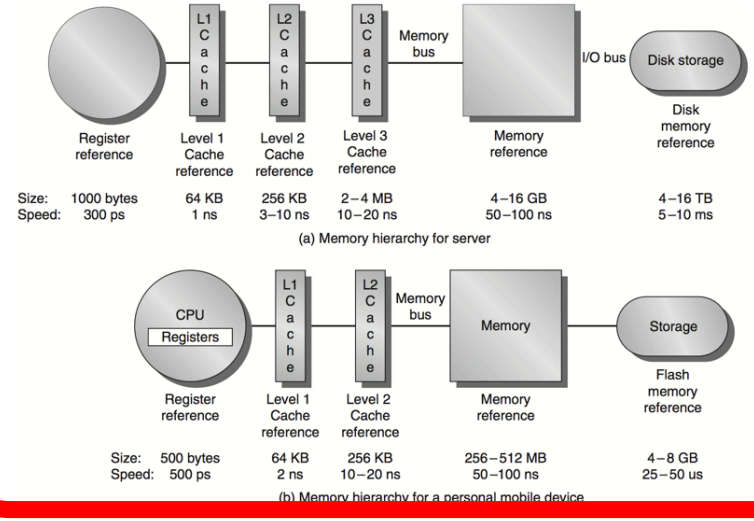
Program Behavior: *Principle of Locality*



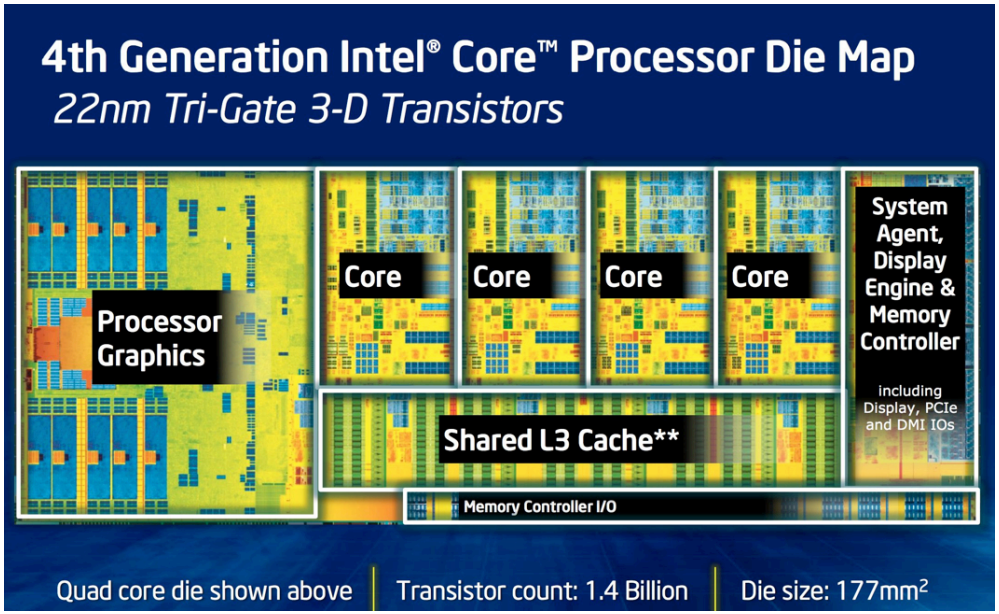
Your Code: *Exploit Locality, and Work With Memory Storage Type*

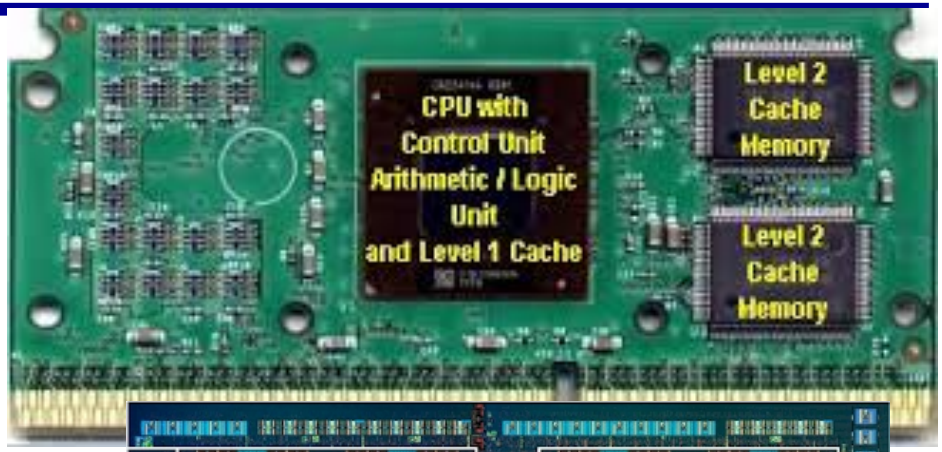
```
int sumarrayrows(int a[M][N]){
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];
  return sum;
}
```

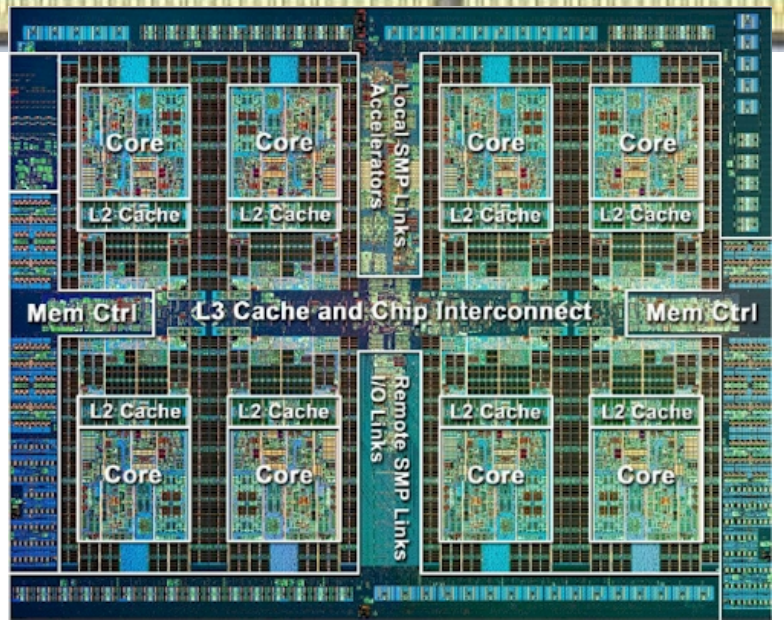
Architecture Approach: *Memory Hierarchy*



Memory Hierarchy







- *capacity*: Register << SRAM << DRAM
- *latency*: Register << SRAM << DRAM
- *bandwidth:* on-chip >> off-chip

On a data access:

if data $\in$ fast memory $\Rightarrow$ low latency access *(SRAM)*

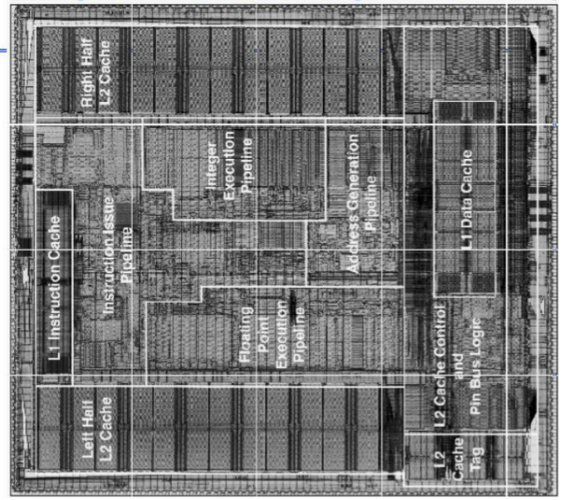
if data $\notin$ fast memory $\Rightarrow$ high latency access *(DRAM)*

History: Alpha 21164 (1994)

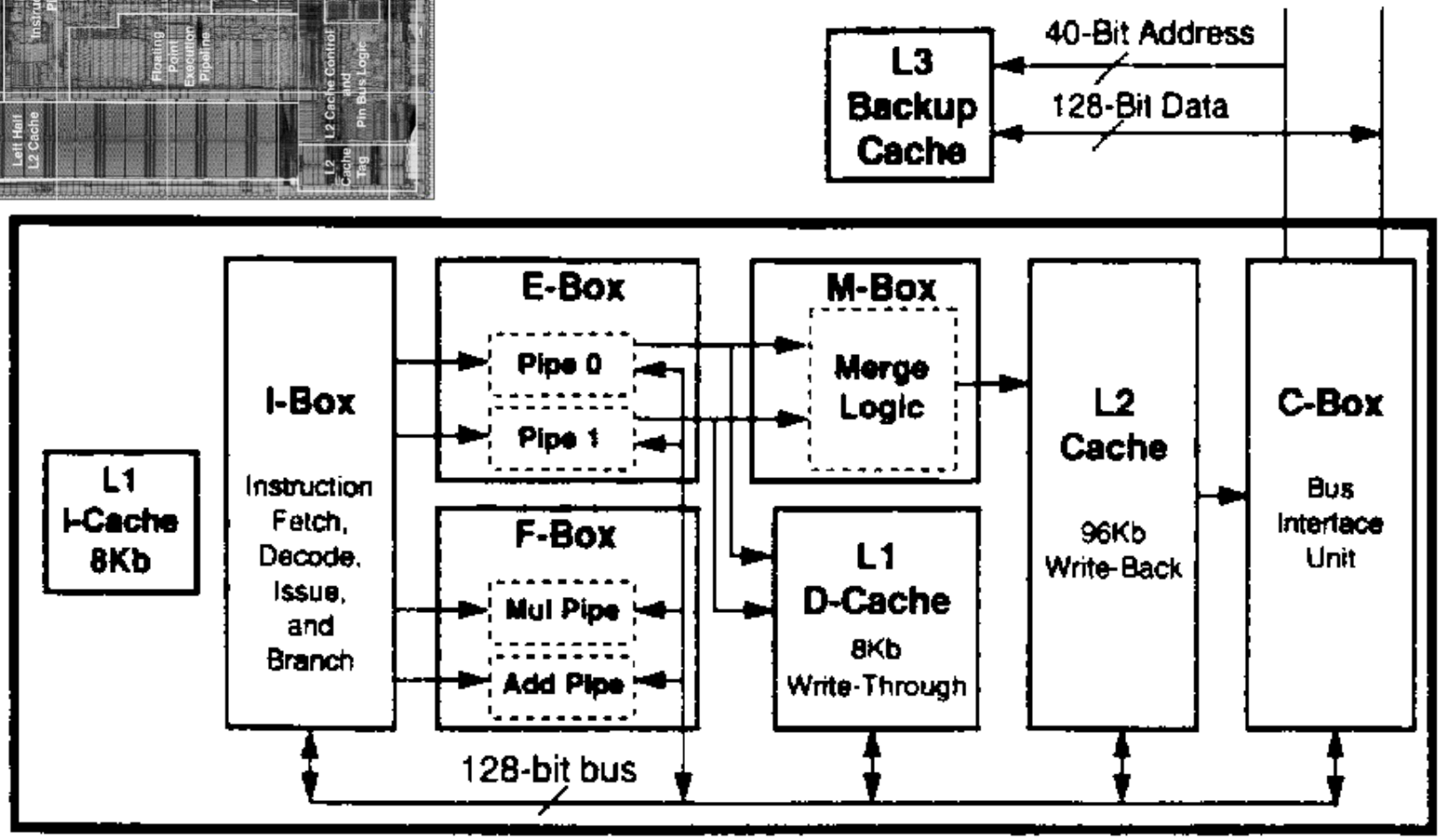
Alpha 21164 Chip Photo

Microprocessor Report 9/12/94

Caches:
- L1 data
- L1 instruction
- L2 unified
- + L3 off-chip



https://en.wikipedia.org/wiki/Alpha_21164



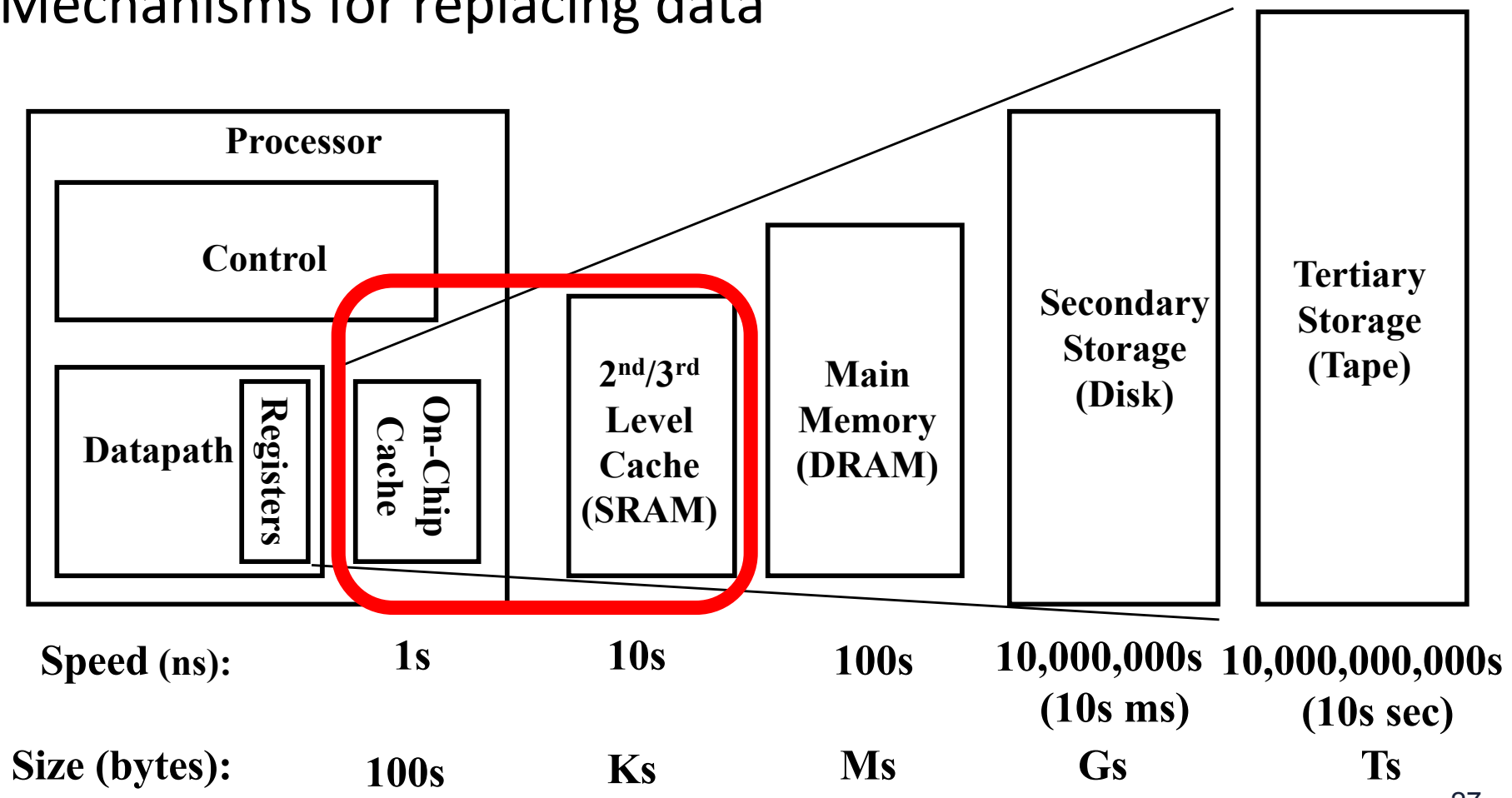
History: Further Back

Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a ***hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.***

A. W. Burks, H. H. Goldstine, and J. von Neumann
***Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,* 1946**

Next: Memory Hierarchy - Cache Organization

- Keep most recent accessed data and its adjacent data in the smaller/faster caches that are closer to processor
- Mechanisms for replacing data



More Examples for Locality Discussion

Sources of locality

- Temporal locality
  - Code within a loop
  - Same instructions fetched repeatedly
- Spatial locality
  - Data arrays
  - Local variables in stack
  - Data allocated in chunks (contiguous bytes)

```
for (i=0; i<N; i++) {
   A[i] = B[i] + C[i] * a;
}
```

Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

Matrix Multiplication Example

- Major cache effects to consider
  - Total cache size
    - Exploit temporal locality and blocking)
  - Block size
    - Exploit spatial locality

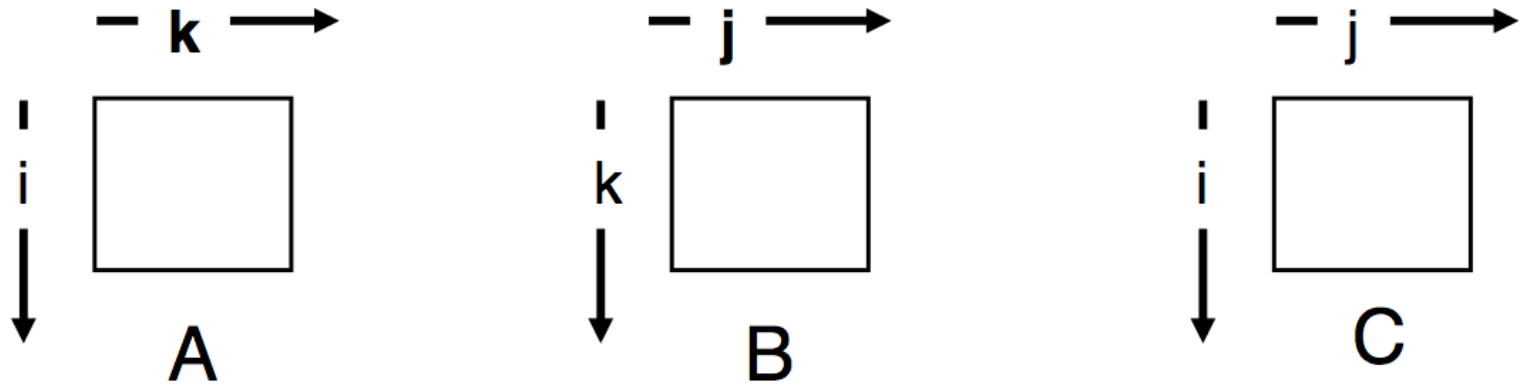
- Description:
  - Multiply N x N matrices
  - $O(N^3)$ total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable ***sum*** *held in register*

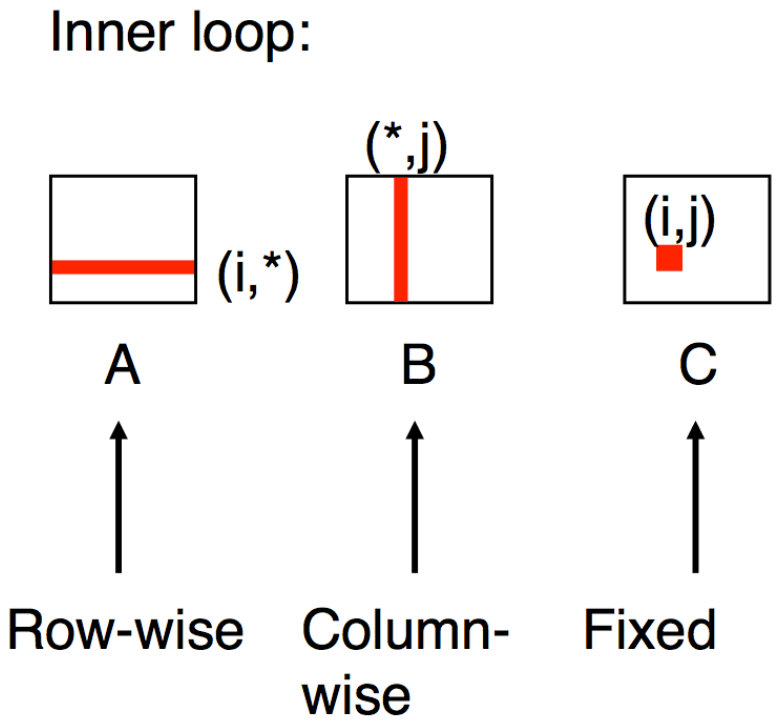
Miss Rate Analysis for Matrix Multiply

- Assume:
  - Cache line size = 32 Bytes (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis method:
  - Look at access pattern of inner loop



Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

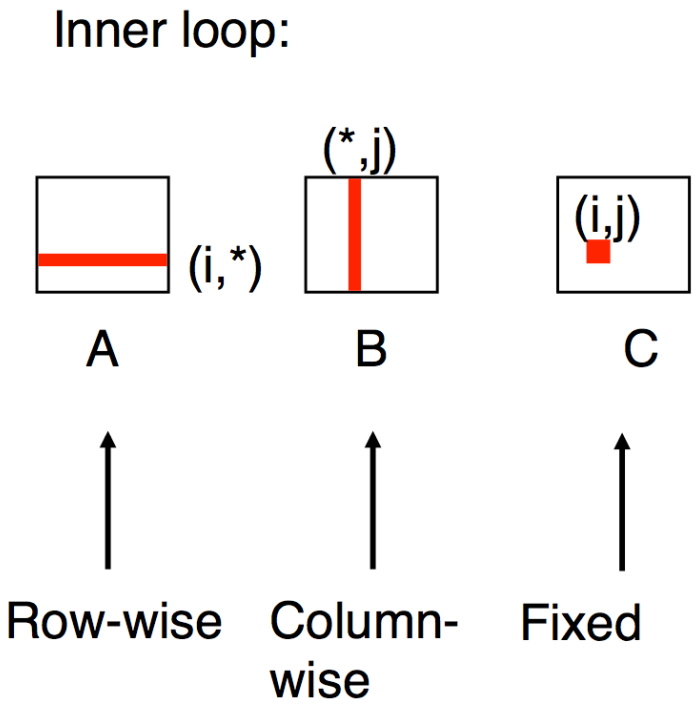


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

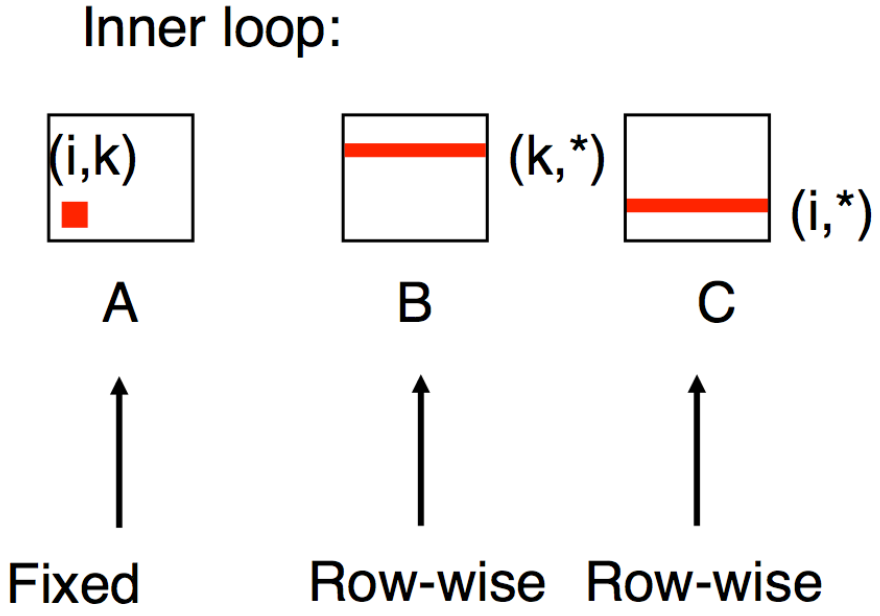


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

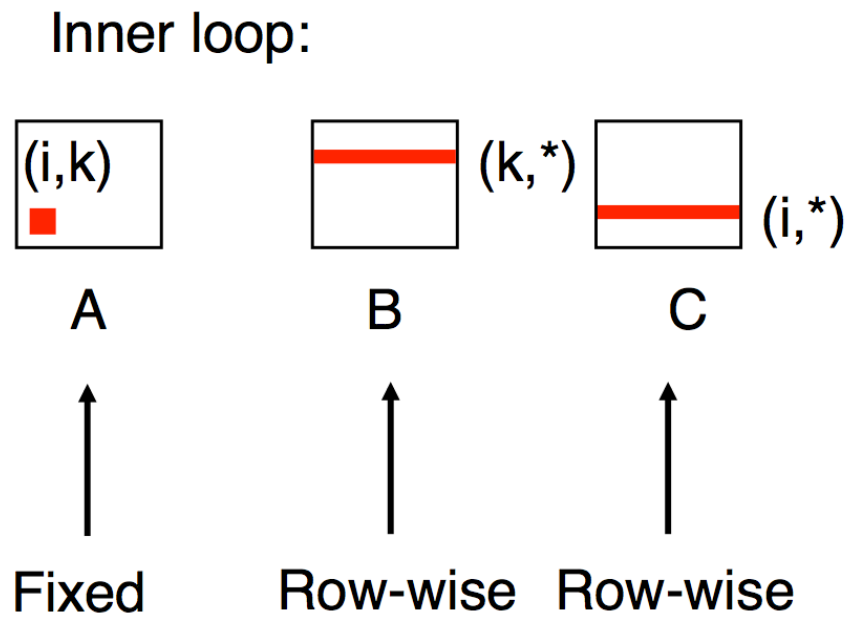


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

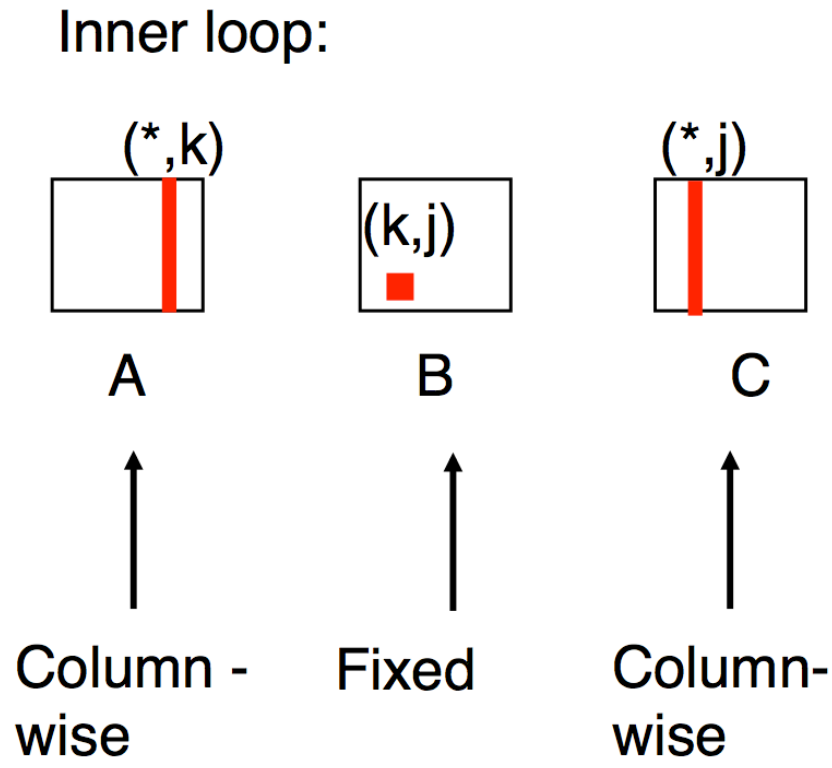


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

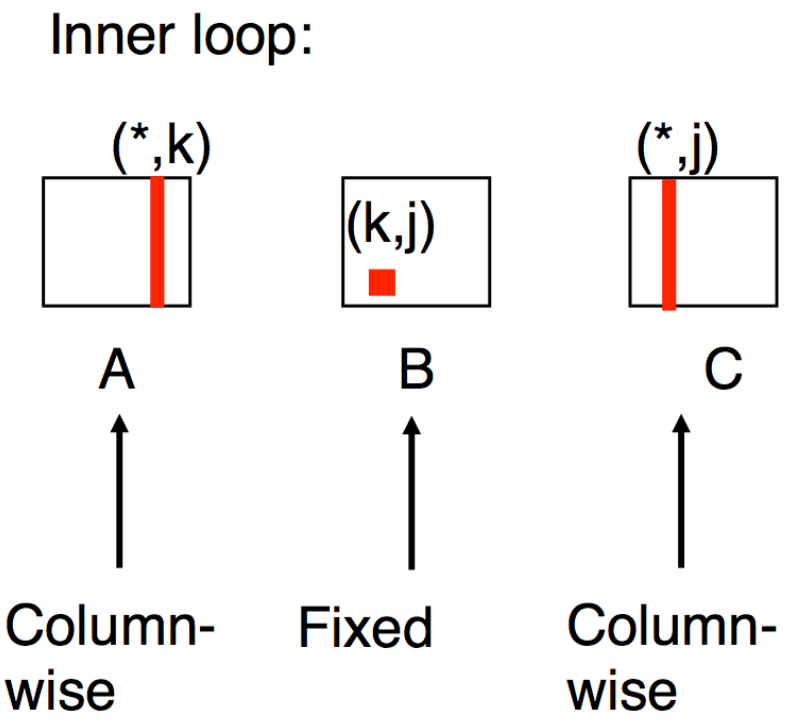


- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (i=0; i<n; i++)  {
   for (j=0; j<n; j++) {
      sum = 0.0;
      for (k=0; k<n; k++)
         sum += a[i][k] *
      b[k][j]; c[i][j] = sum;
   }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
for (k=0; k<n; k++) {
   for (i=0; i<n; i++) {
      r = a[i][k];
      for (j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++) {
      r = b[k][j];
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * r;
   }
}
```