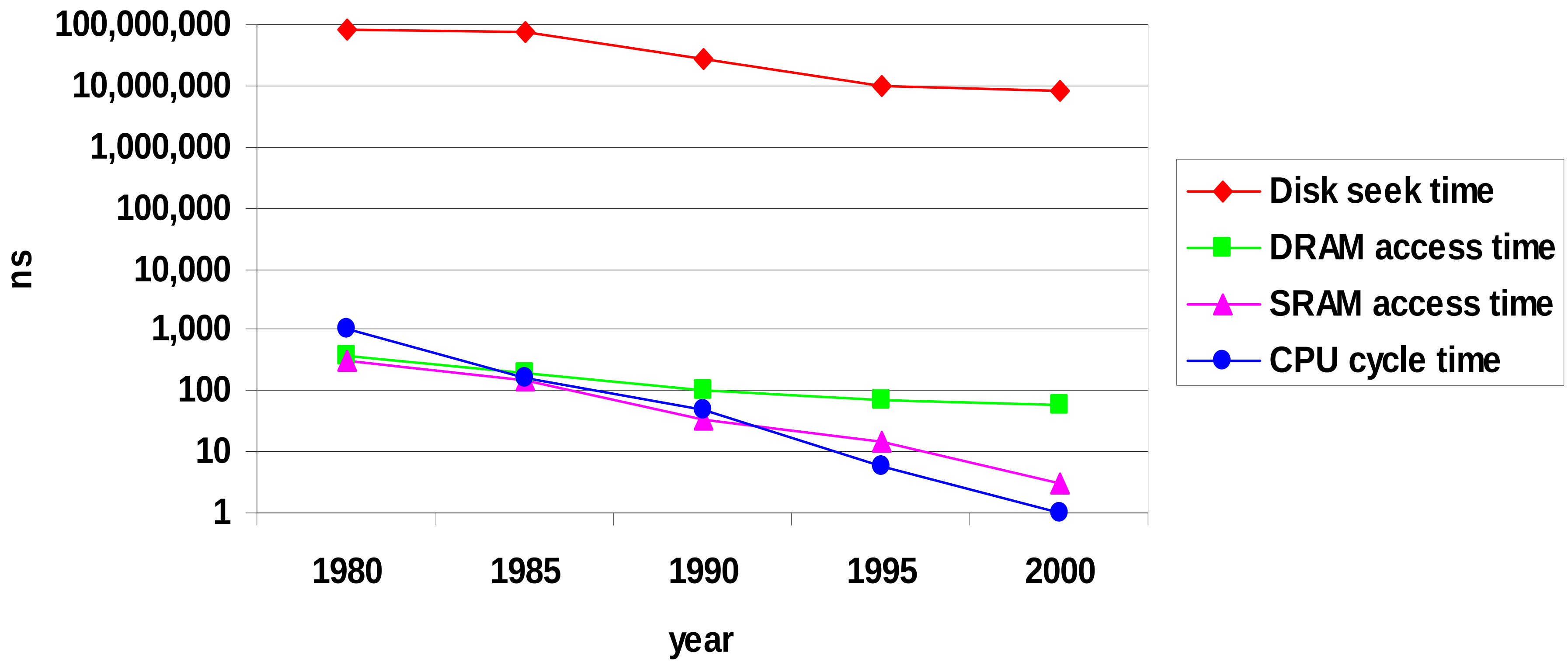
# Memory Hierarchy

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

# The CPU-Memory Gap

- The increasing gap between DRAM, disk, and CPU speeds



# Principle of Locality (1)

- **Temporal locality**
  - Recently referenced items are likely to be referenced in the near future.
- **Spatial locality**
  - Items with nearby addresses tend to be referenced close together in time.

# Principle of Locality (2)

- **Locality example:**

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

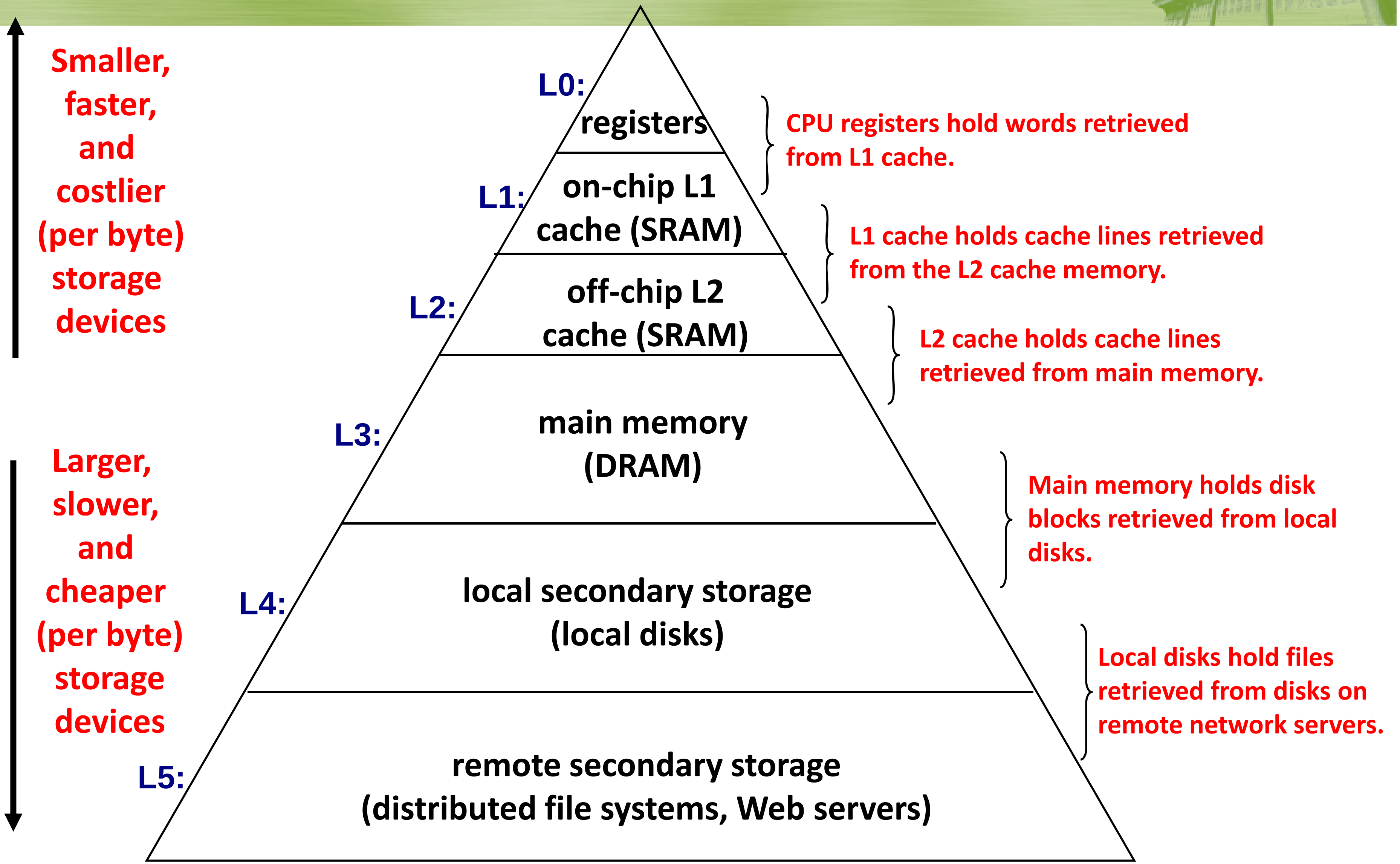
- Data
  - Reference array elements in succession — **Spatial locality**
  - Reference sum each iteration — **Temporal locality**
- Instructions
  - Reference instructions in sequence — **Spatial locality**
  - Cycle through loop repeatedly — **Temporal locality**

# Principle of Locality (3)

- **How to exploit temporal locality?**
  - Speed up data accesses by caching data in faster storage.
  - Caching in multiple levels: form a memory hierarchy:
    - The lower levels of the memory hierarchy tend to be slower, but larger and cheaper.

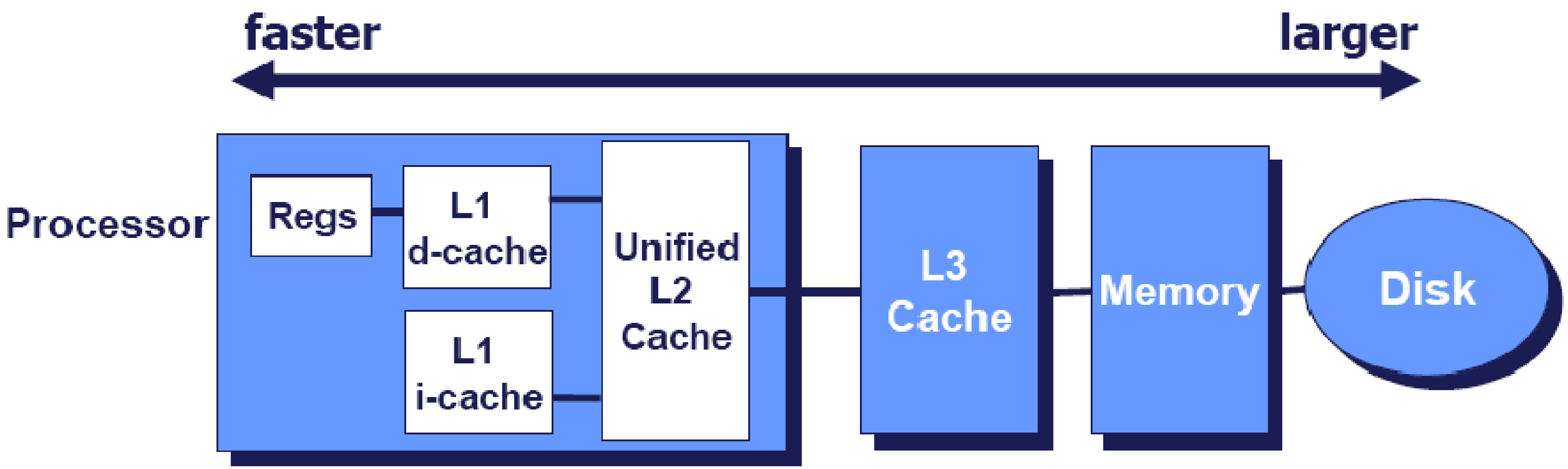
- **How to exploit spatial locality?**
  - Larger cache line size
    - Cache nearby data together.

# Memory Hierarchy



# Caching (1)

- **Cache**
  - A smaller, faster storage
  - Improves the average access time
  - Exploits both temporal and spatial locality



# Caching (2)

## Cache performance metrics

- Average memory access time = $T_{hit} + R_{miss} * T_{miss}$
- Hit time ($T_{hit}$)
  - Time to deliver a line in the cache to the processor
  - Includes time to determine whether the line is in the cache
  - 1 clock cycle for L1, 3 ~ 8 clock cycles for L2
- Miss rate ($R_{miss}$)
  - Fraction of memory references not found in cache (misses/references)
  - 3 ~ 10% for L1, < 1% for L2
- Miss penalty ($T_{miss}$)
  - Additional time required because of a miss
  - Typically 25 ~ 100 cycles for main memory

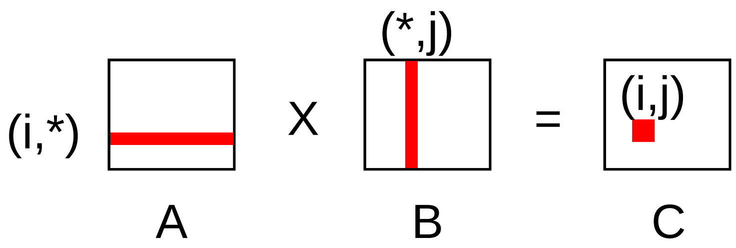
# Caching (3)

- **Cache design issues**
  - Cache size
    - 8KB ~ 64KB for L1
  - Cache line size
    - Typically, 32B or 64B for L1
  - Lookup
    - Fully associative
    - Set associative: 2-way, 4-way, 8-way, 16-way, etc.
    - Direct mapped
  - Replacement
    - LRU (Least Recently Used)
    - FIFO (First-In First-Out), etc.

# Matrix Multiplication (1)

## ▪ Description

- Multiply N x N matrices
- O($N^3$) total operations




```
/* 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*

## ▪ Assumptions

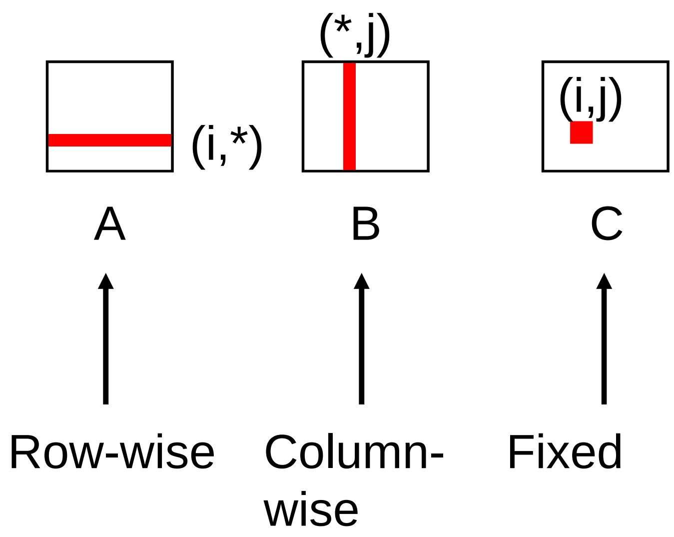
- Line size = 32B (big enough for 4 64-bit words)
- Matrix dimension (N) is very large

# Matrix Multiplication (2)

- **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;
  }
}
```

Inner loop:




**Misses per Inner Loop Iteration:**

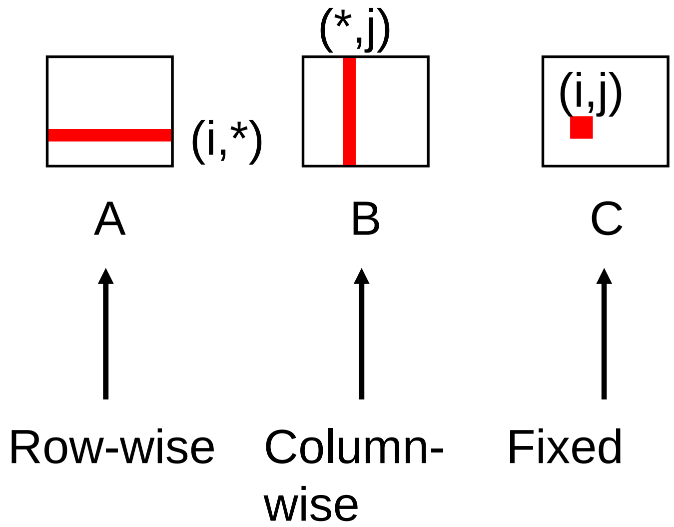
| A | B | C |
|---|---|---|
| 0.25 | 1.0 | 0.0 |

# Matrix Multiplication (3)

- **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
  }
}
```

Inner loop:



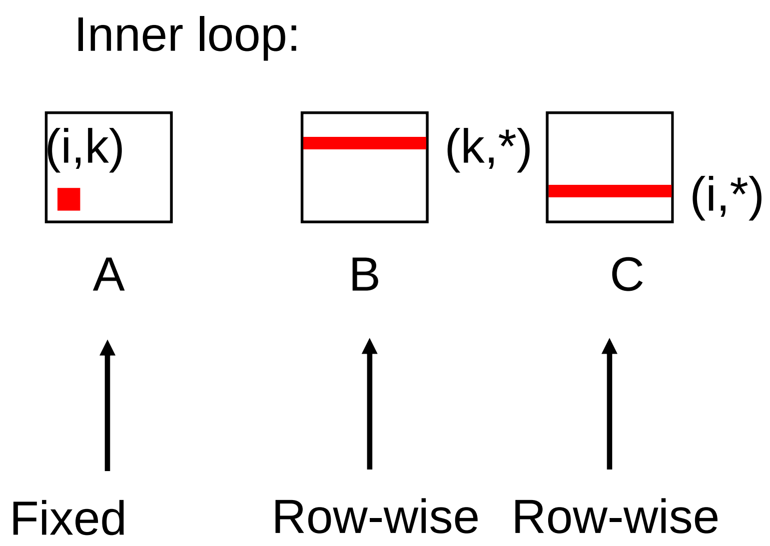

**Misses per Inner Loop Iteration:**

| A | B | C |
|---|---|---|
| 0.25 | 1.0 | 0.0 |

# Matrix Multiplication (4)

- **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:**

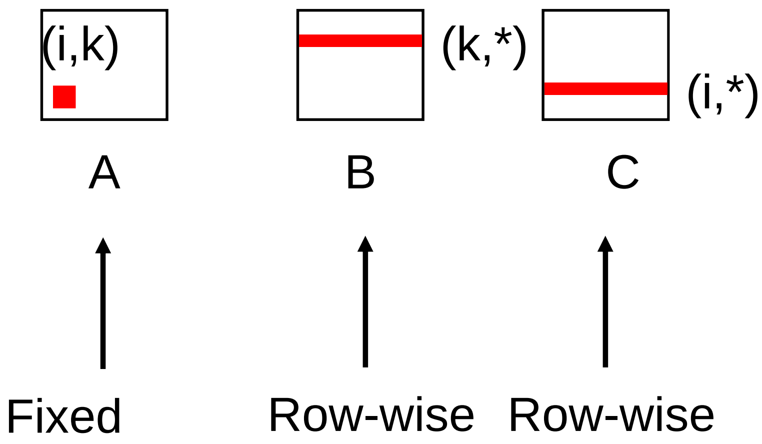
| A | B | C |
|---|---|---|
| 0.0 | 0.25 | 0.25 |

# Matrix Multiplication (5)

- **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];
  }
}
```

Inner loop:




**Misses per Inner Loop Iteration:**

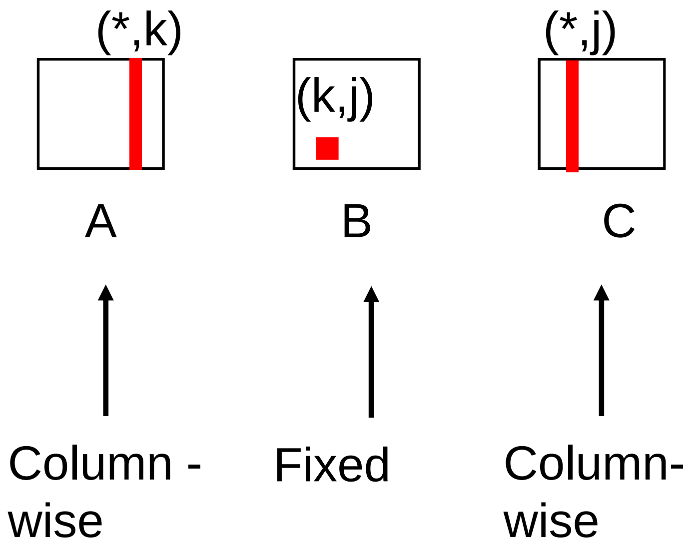
| A | B | C |
|---|---|---|
| 0.0 | 0.25 | 0.25 |

# Matrix Multiplication (6)

- 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;
  }
}
```

Inner loop:




**Misses per Inner Loop Iteration:**

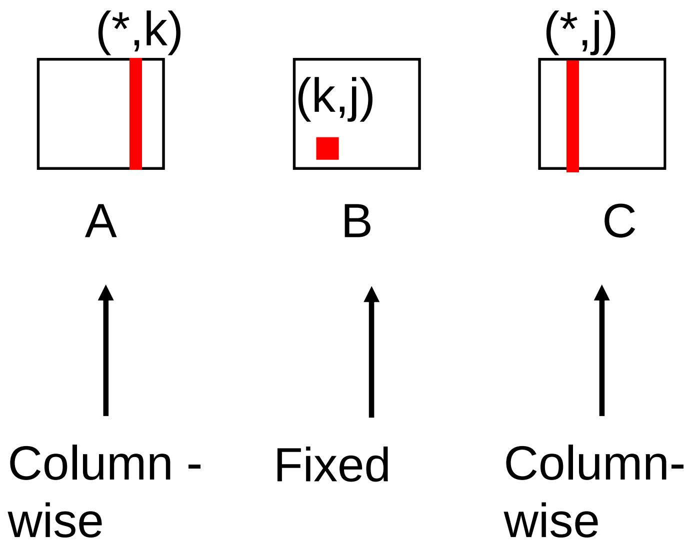
| A | B | C |
|---|---|---|
| 1.0 | 0.0 | 1.0 |

# Matrix Multiplication (7)

- **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;
  }
}
```

Inner loop:




**Misses per Inner Loop Iteration:**

| A | B | C |
|---|---|---|
| 1.0 | 0.0 | 1.0 |

# Matrix Multiplication (8)

- **Summary**

**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;
   }
}
```

| A | B | C |
|---|---|---|
| 0.25 | 1.0 | 0.0 |

**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];
   }
}
```

| A | B | C |
|---|---|---|
| 0.0 | 0.25 | 0.25 |

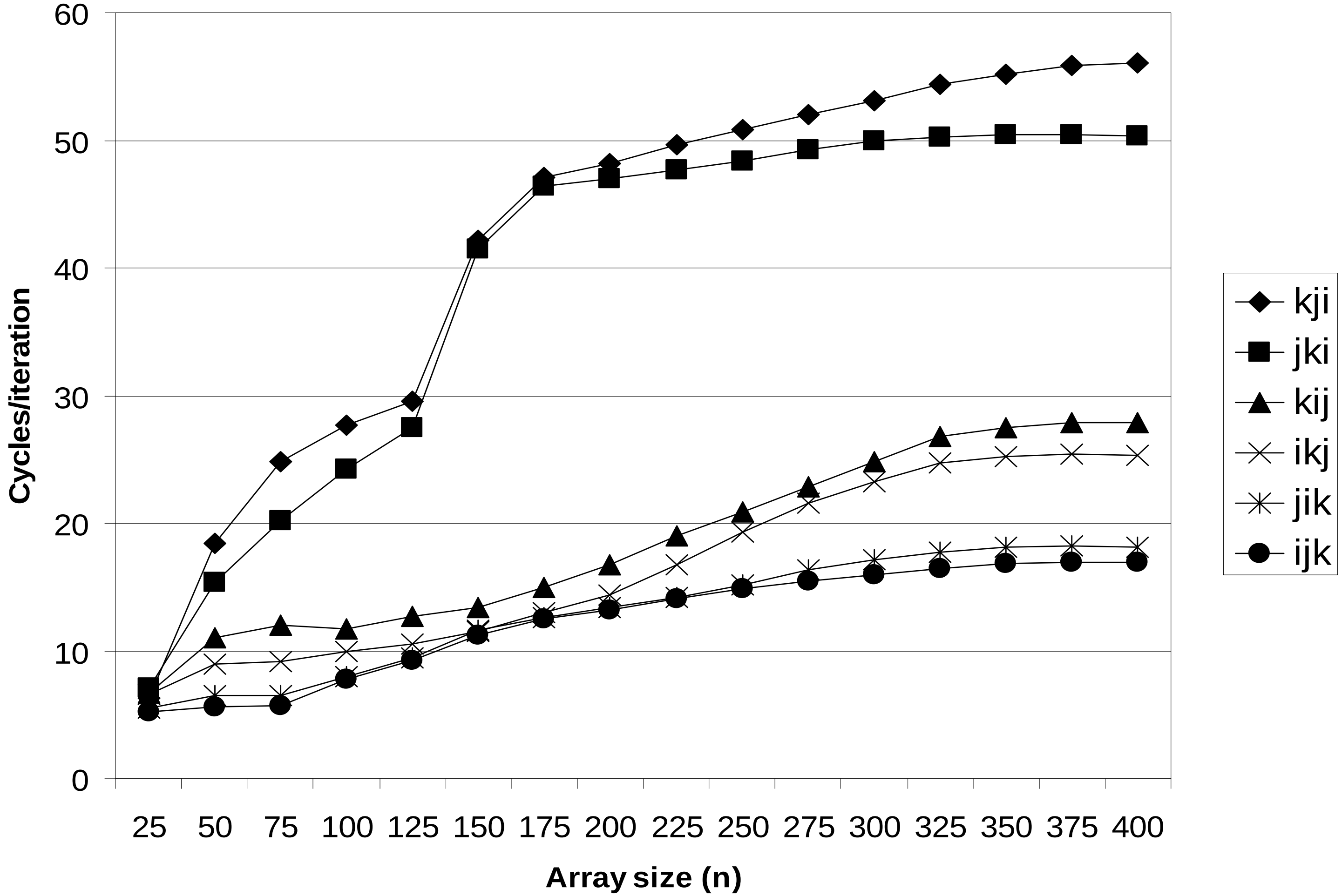
**jki (& kji):**
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;
   }
}
```

| A | B | C |
|---|---|---|
| 1.0 | 0.0 | 1.0 |

# Matrix Multiplication (9)

- **Performance in Pentium**



# Blocked Matrix Multiplication (1)

- **Improving temporal locality by blocking**
  - "Block" means a sub-block within the matrix.
  - Example: N = 8, sub-block size = 4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \text{X} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

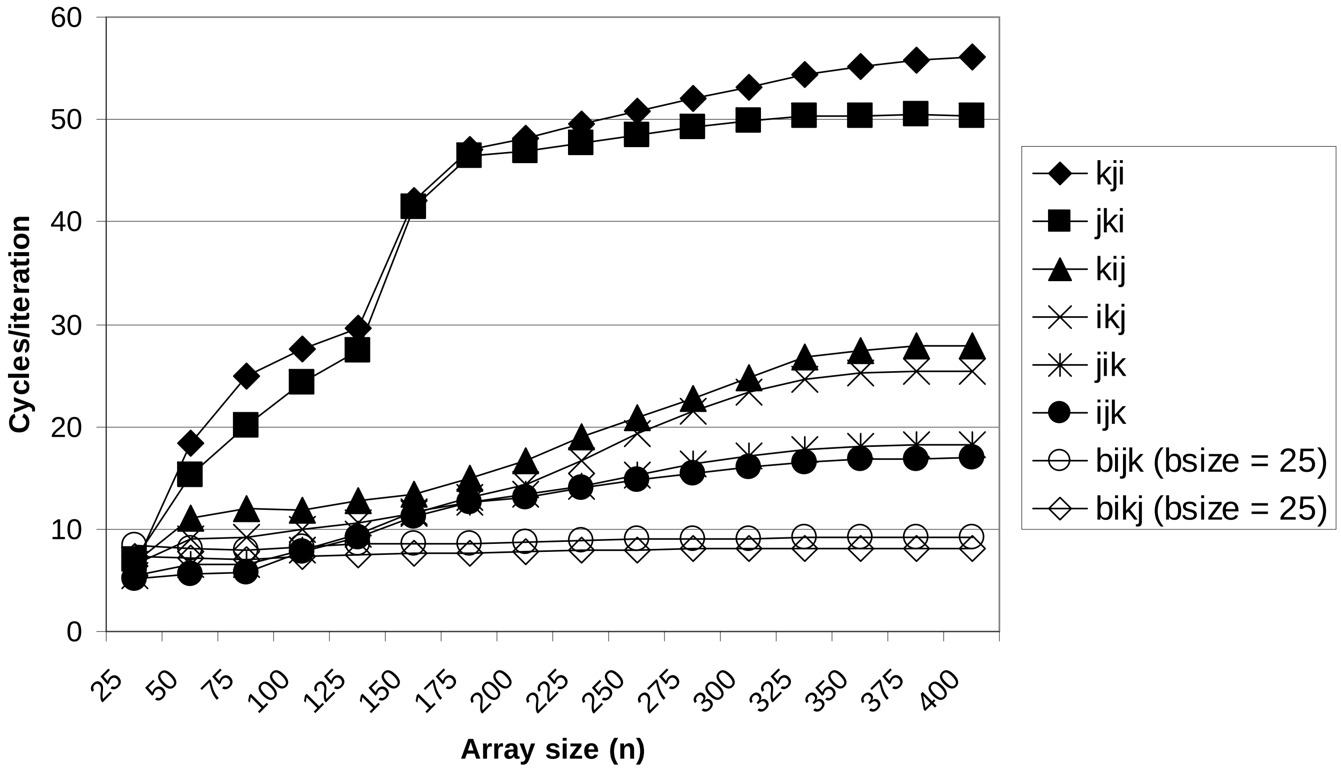
  - Key idea: Sub-blocks (i.e., $A_{xy}$) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

# Blocked Matrix Multiplication (2)

- **Performance in Pentium**
  - Blocking improves performance by a factor of two over unblocked version (ijk and jik)

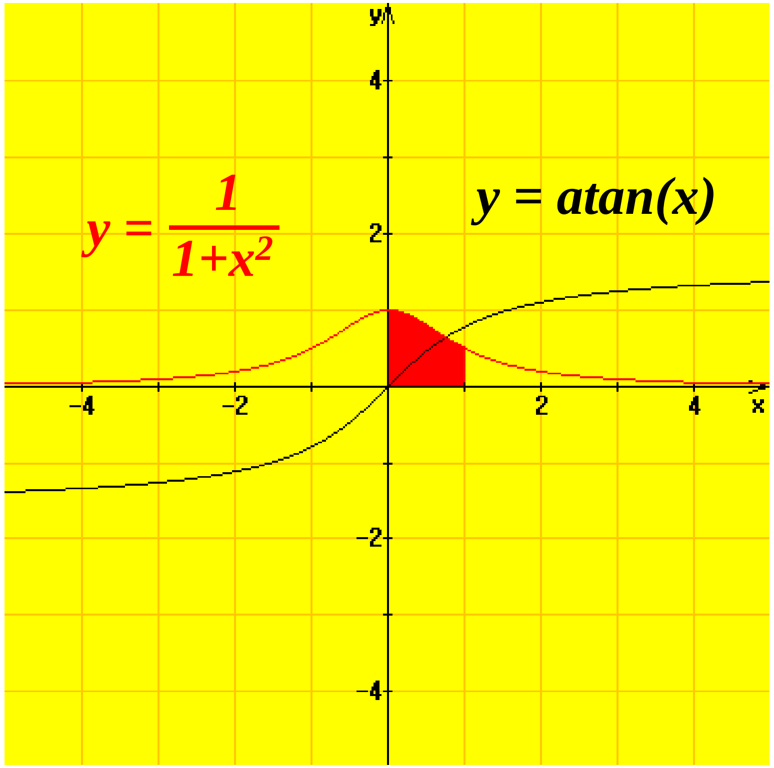


# Calculating Pi (1)

- $\tan \frac{\pi}{4} = 1 \Rightarrow \operatorname{atan}(1) = \frac{\pi}{4}$
- $4 \int_0^1 \frac{1}{1+x^2}\, dx$

  $= 4\ (\operatorname{atan}(1) - \operatorname{atan}(0))$

  $= 4\ (\frac{\pi}{4} - 0) = \pi$



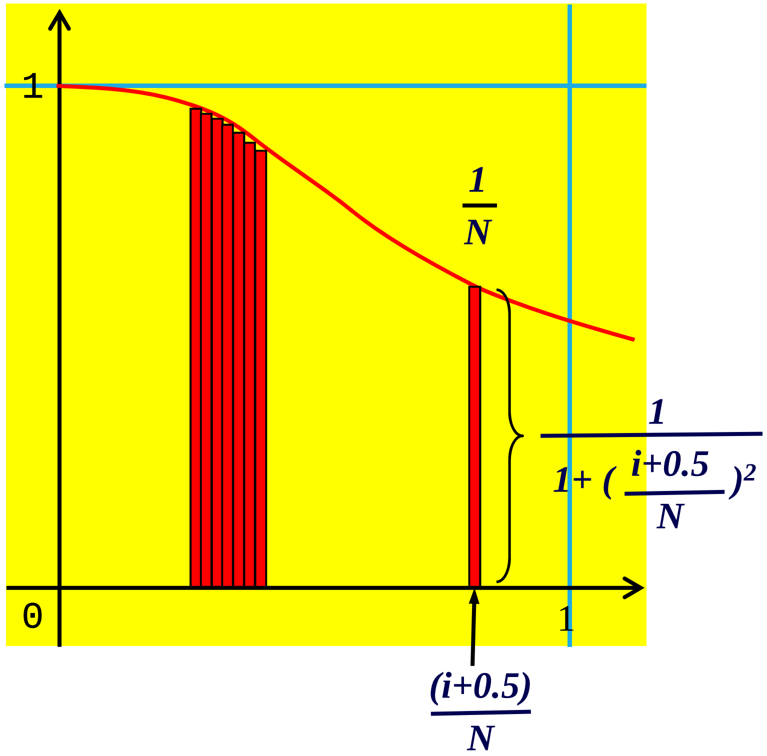
# Calculating Pi (2)

- **Pi: Sequential version**

```
#define N         20000000
#define STEP      (1.0 / (double) N)

double compute ()
{
    int i;
    double x;
    double sum = 0.0;

    for (i = 0; i < N; i++)
    {
        x = (i + 0.5) * STEP;
        sum += 4.0 / (1.0 + x * x);
    }
    return sum * STEP;
}
```



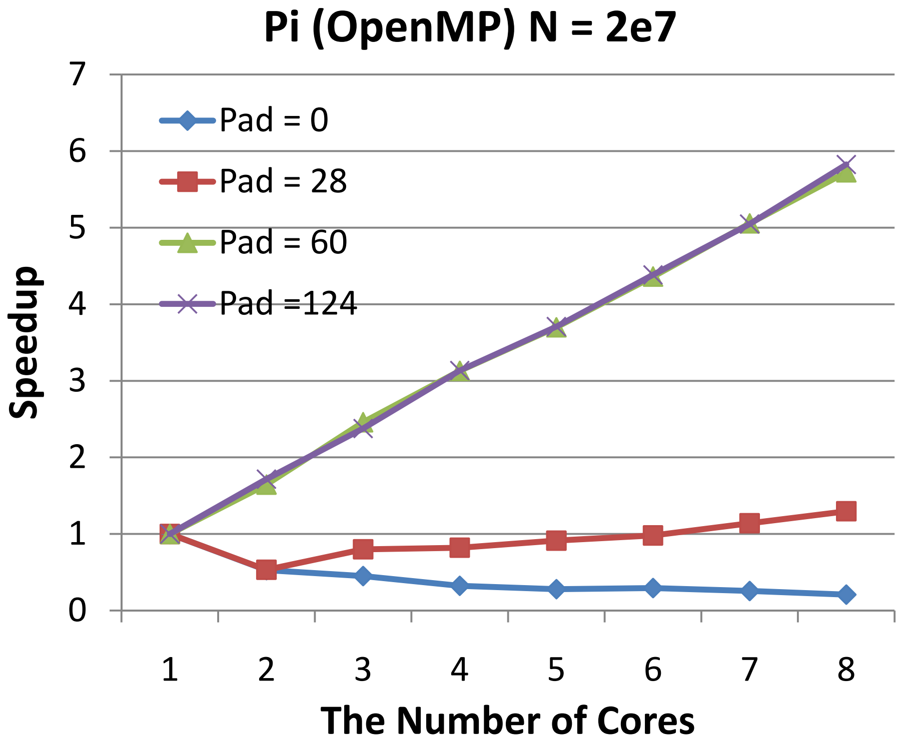
# Calculating Pi (3)

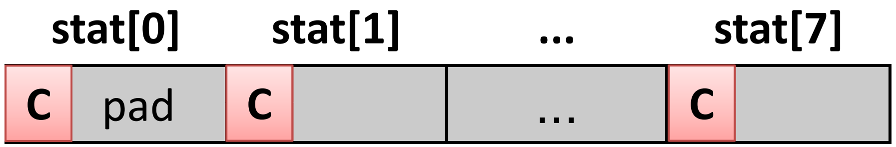
- False sharing

```
struct thread_stat {
  int count;
  char pad[PADS];
} stat[8];

double compute ()  {
  int i, id;
  double x, sum = 0.0;

#pragma omp parallel for
private(x) reduction(+:sum)
  for (i = 0; i < N; i ++)  {
    int id=omp_get_thread_num();
    stat[id].count++;
    x = (i + 0.5) * STEP;
    sum += 4.0 / (1.0 + x * x);
  }
  return sum * STEP;
}
```



# Observations

- **Programmer can optimize for cache performance.**
  - How data structures are organized.
  - How data are accessed.
    - Nested loop structure
    - Blocking is a general technique.
- **All systems favor "cache friendly code"**
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)