Lecture 1: Single processor performance

Why parallel computing

- Solving an $n \times n$ linear system *Ax*=*b* by using Gaussian elimination takes $\approx \frac{1}{3}n^3$ flops.
- On **Core i7** 975 @ 4.0 GHz, which is capable of about 60-70 Gigaflops

n	flops	time
1000	3.3×10^{8}	0.006 seconds
1000000	3.3×10^{17}	57.9 days

www.top500.org

TOP500® SUPERCOMPUTER SITES

PROJECT | LISTS | STATISTICS | RESOURCES | NEWS

TOP 10 Systems - 11/2011

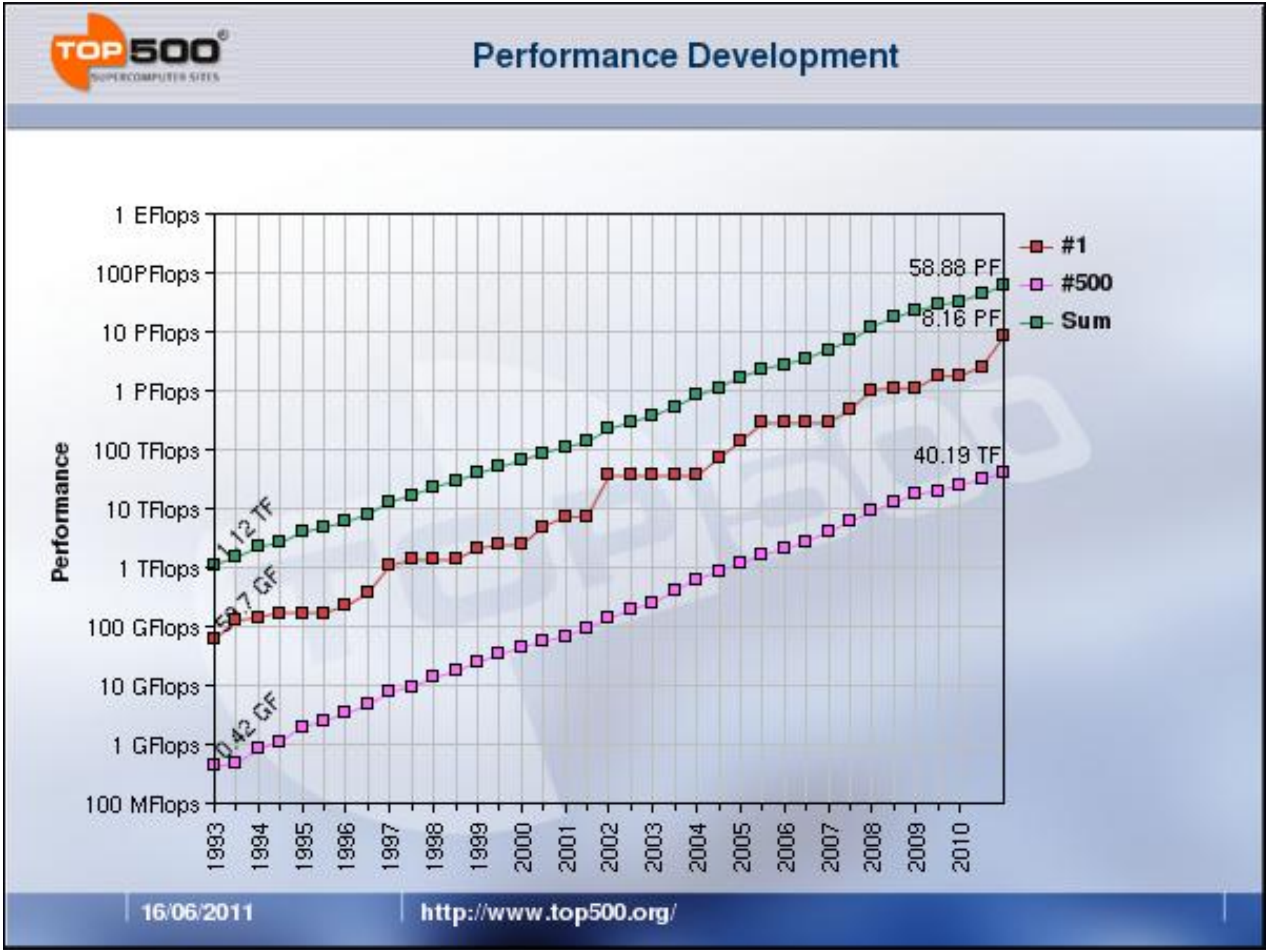
1	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect
2	NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050
3	Cray XT5-HE Opteron 6-core 2.6 GHz
4	Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU
5	HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows
6	Cray XE6, Opteron 6136 8C 2.40GHz, Custom
7	SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband
8	Cray XE6, Opteron 6172 12C 2.10GHz, Custom
9	Bull bullx super-node S6010/S6030
10	BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband

Japan's K Computer Tops 10 Petaflop/s to Stay Atop TOP500 List

Fri, 2011-11-11 11:11

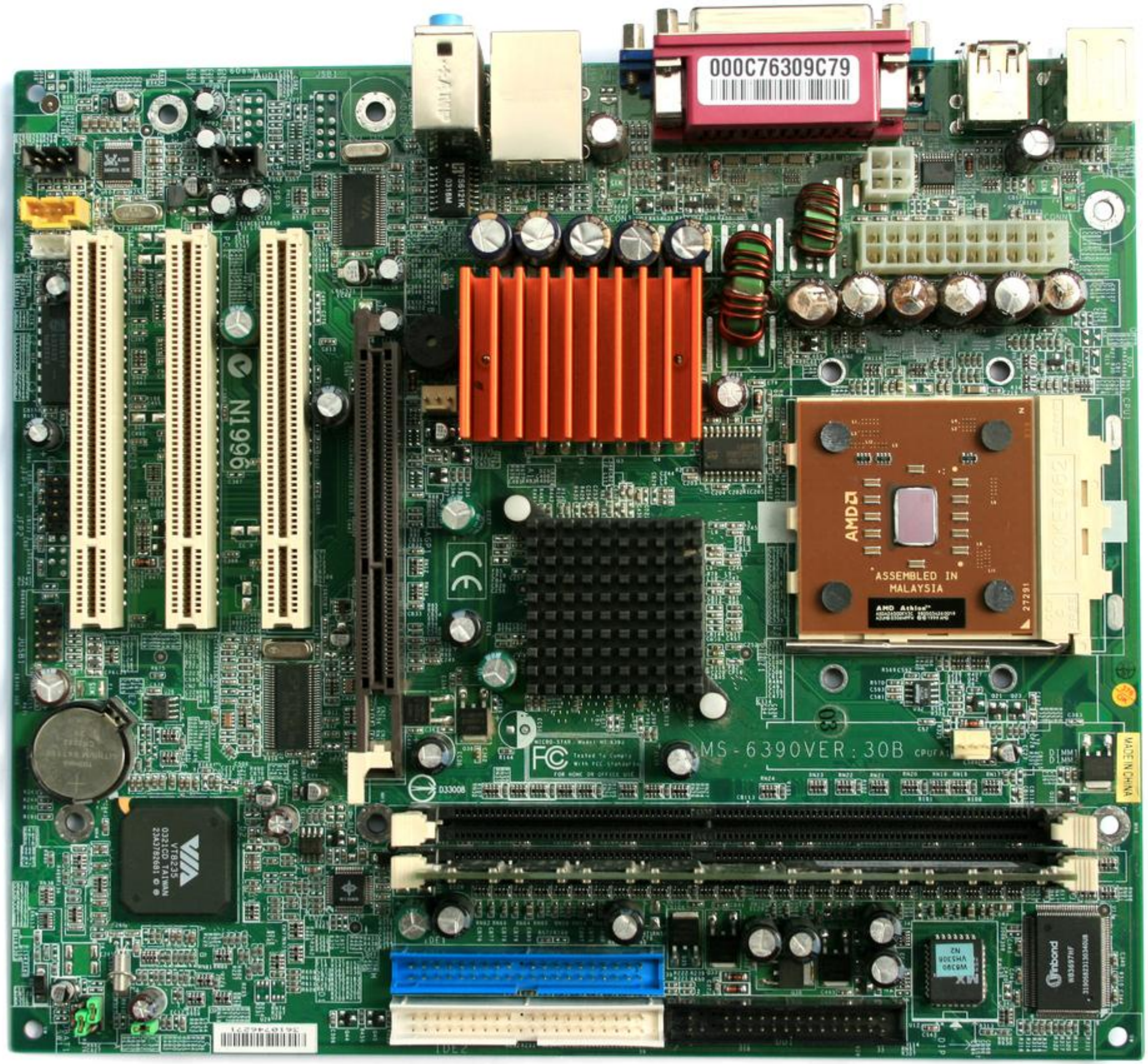
BERKELEY, Calif.; KNOXVILLE, Tenn.; and MANNHEIM, Germany (Nov. 14, 2011)—Japan's "K Computer" maintained its position atop the newest edition of the TOP500 List of the world's most powerful supercomputers, thanks to a full build-out that makes it four times as powerful as its nearest competitor. Installed at the RIKEN Advanced Institute for Computational Science (AICS) in Kobe, Japan, the K Computer it achieved an impressive 10.51 Petaflop/s on the Linpack benchmark using 705,024 SPARC64 processing cores.

» Read more

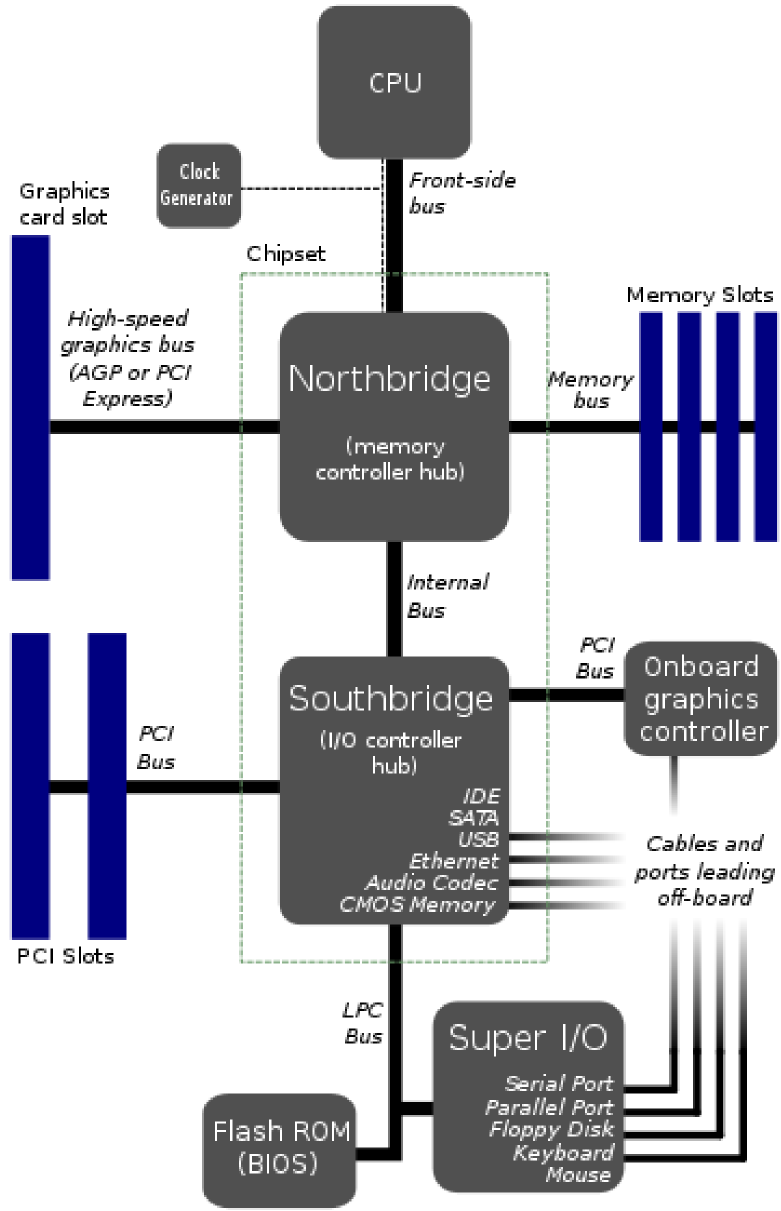


Over 17 years, 10000-fold increases.

What’s in a computer



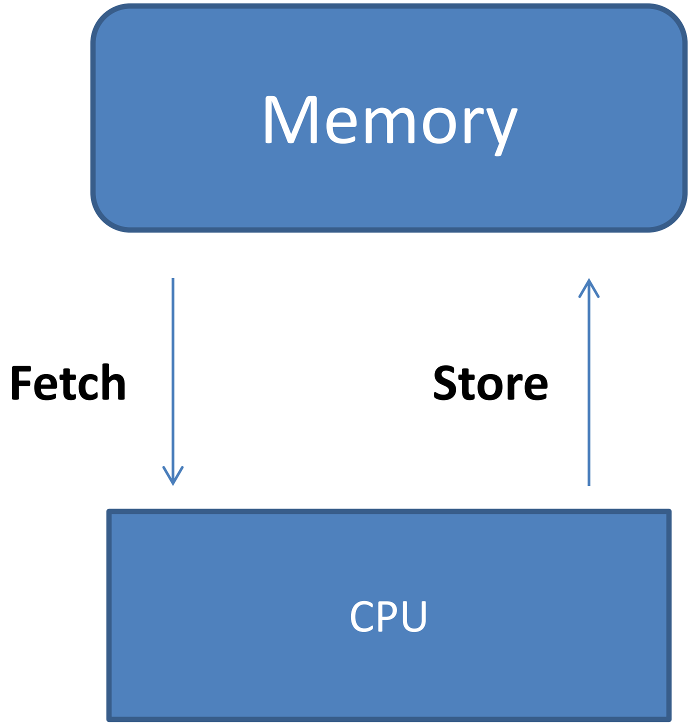
Motherboard diagram



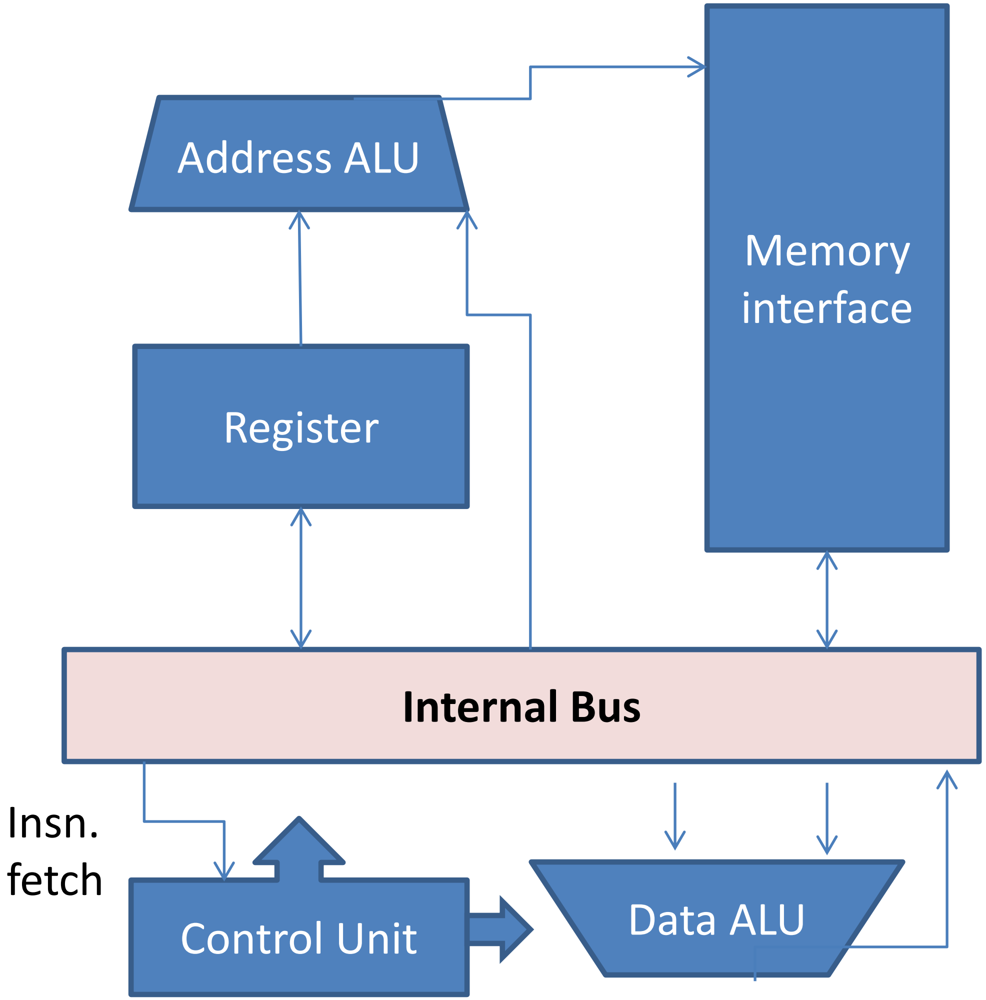
http://en.wikipedia.org/wiki/Front-side_bus

von Neumann machine

- Common machine model for many years
- Stored-program concept
- CPU executes a stored program
- Machine is divided into a CPU and main memory



16-bit Intel 8086 processor



First available in 1978

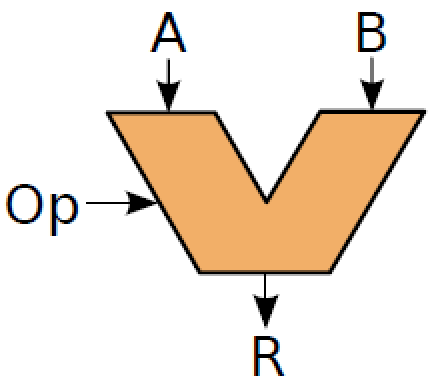
ALU

Arithmetic Logic Unit (**ALU**)
ALU takes one or two operands A,B
Operation:

1. Addition, Subtraction (integer)
2. Multiplication, Division (integer)
3. And, Or, Not (logical operation)
4. Bitwise operation (shifts, equivalent to multiplication by power of 2)

Specialized ALUs:

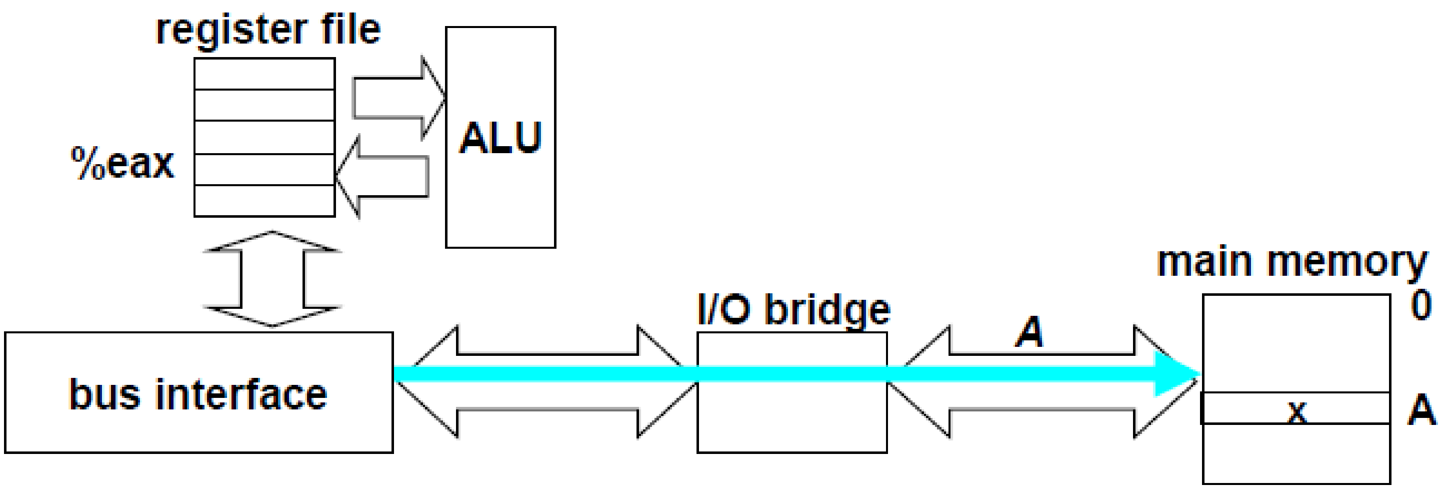
- Floating Point Unit (FPU)
- Address ALU



Memory read transaction (1)

Load operation: movl A, %eax

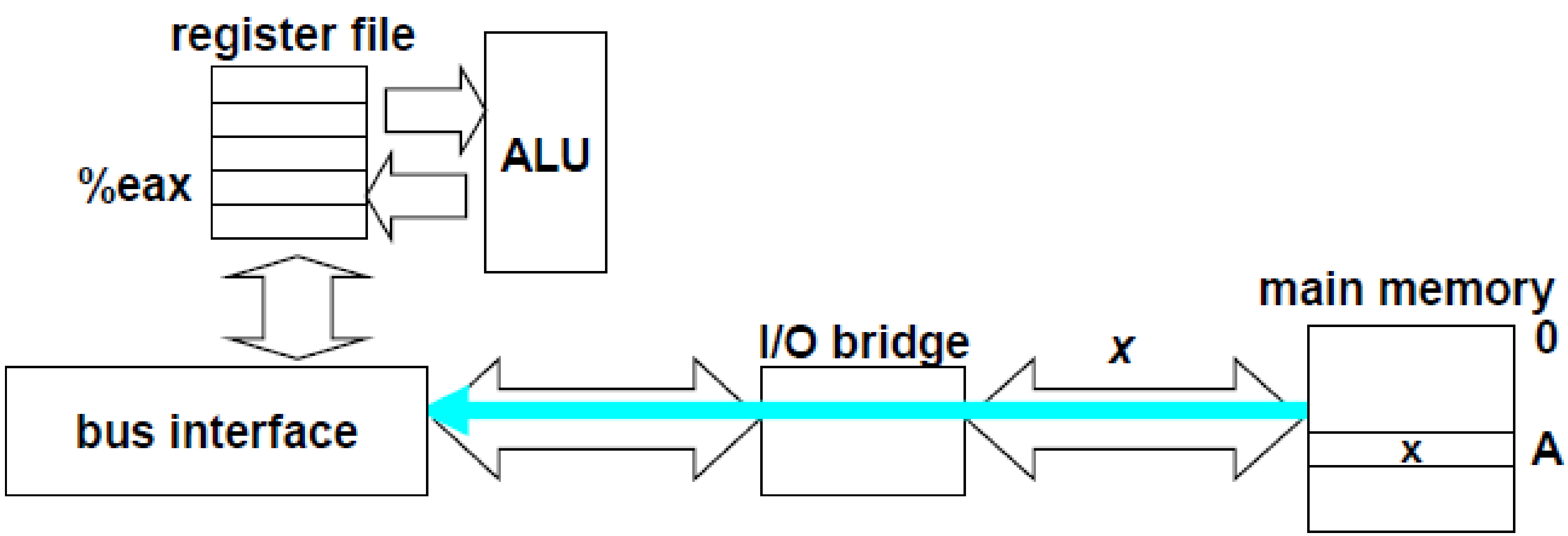
- Load content of address A into a register
- CPU places address A on the system bus, I/O bridge passes it onto the memory bus



Memory read transaction (2)

Load operation: movl A, %eax

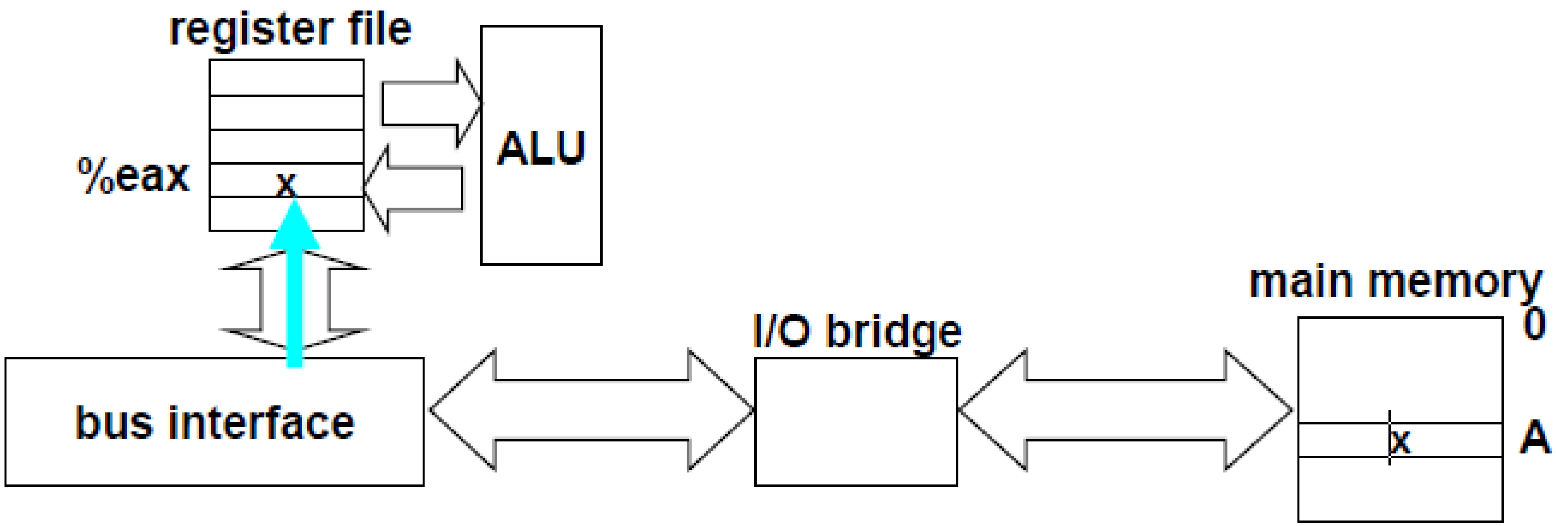
- Main memory reads A from memory bus, retrieve word x, and places x on the bus; I/O bridge passes it along to the system bus



Memory read transaction (3)

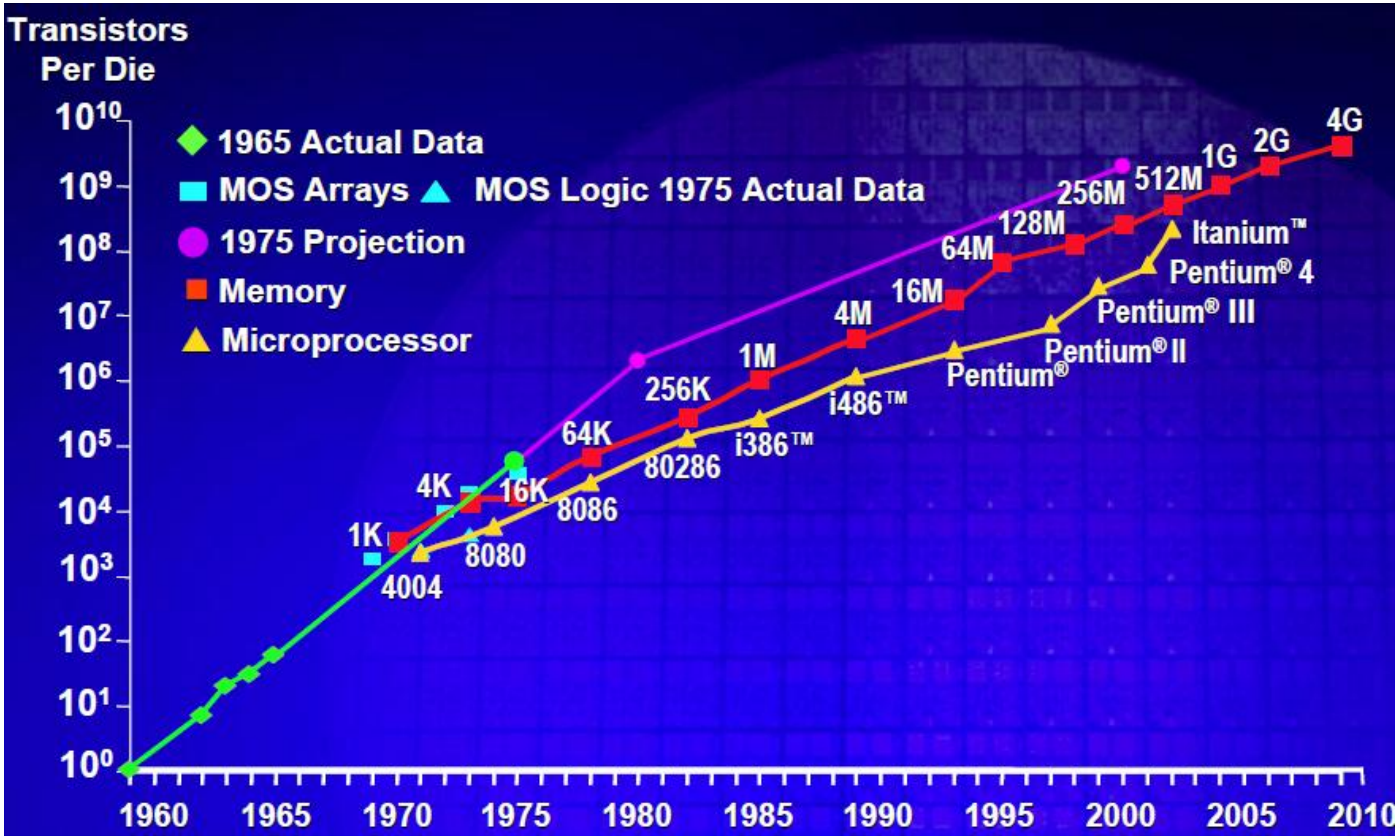
Load operation: movl A, %eax

- CPU read word x from the bus and copies it into register %eax



Moore's law

- Gordon Moore's observation in 1965: the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented (often interpreted as **Computer performance doubles every two years (same cost)**)



(Gordon_Moore_ISSCC_021003.pdf)

Moore's law

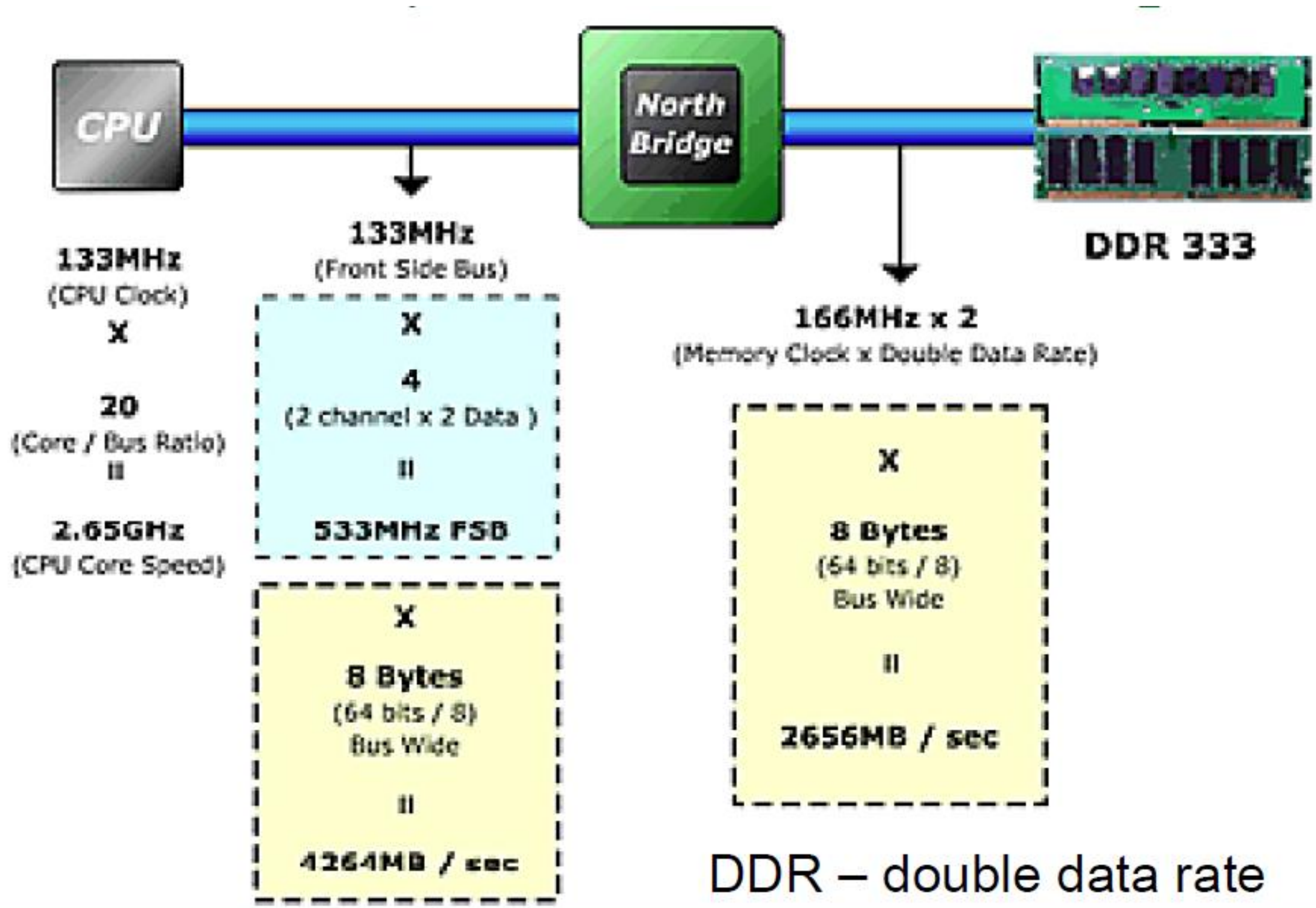
- Moore's revised observation in 1975: the pace slowed down a bit, but data density had doubled approximately every 18 months
- Moore's law is dead

Gordon Moore quote from 2005: "in terms of size [of transistor] ..we're approaching the size of atoms which is a fundamental barrier..."

Date	Intel Transistors CPU (x1000)	Technology
1971	4004 2.3	
1978	8086 31	2.0 micron
1982	80286 110	HMOS
1985	80386 280	0.8 micron CMOS
1989	80486 1200	
1993	Pentium 3100	0.8 micron biCMOS
1995	Pentium Pro 5500	0.6 micron – 0.25

Effect of memory latency on performance (1)

von Neumann Bottleneck: the transfer of data and instructions between memory and the CPU

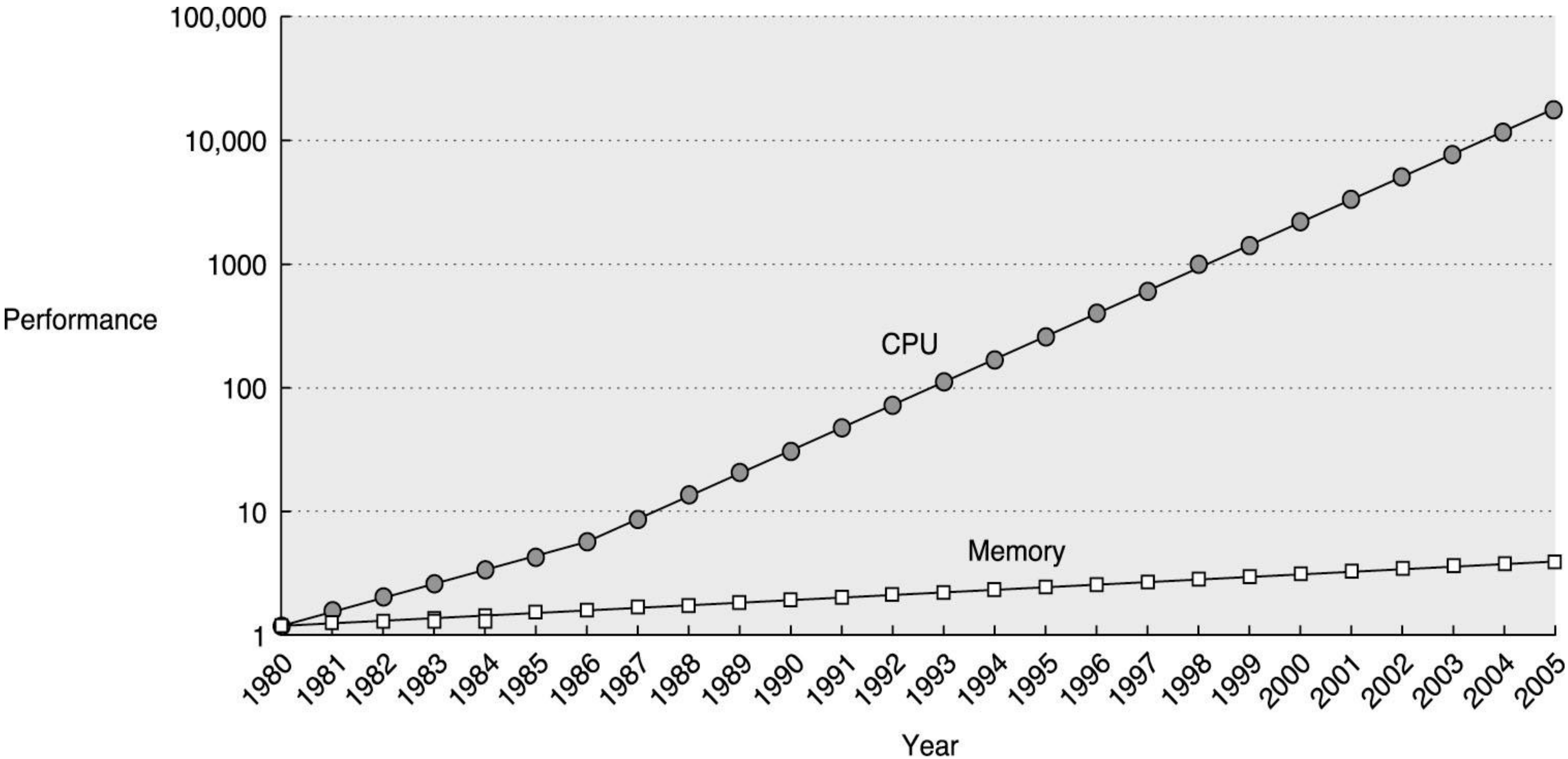


Effect of memory latency on performance (2)

Example. Assume a CPU operates at 1GHz (1 ns clock) and is connected to a DRAM with a latency of 100 ns. Assume the CPU has 2 multiply/add units and is capable of executing 4 instructions in each cycle of 1 ns. The peak CPU rating is 4GFLOPS (floating-point operations per second).

Since the memory latency is 100 cycles, CPU must wait 100 cycles before it can process data. Therefore, the peak speed of computation is 10MFLOPS.

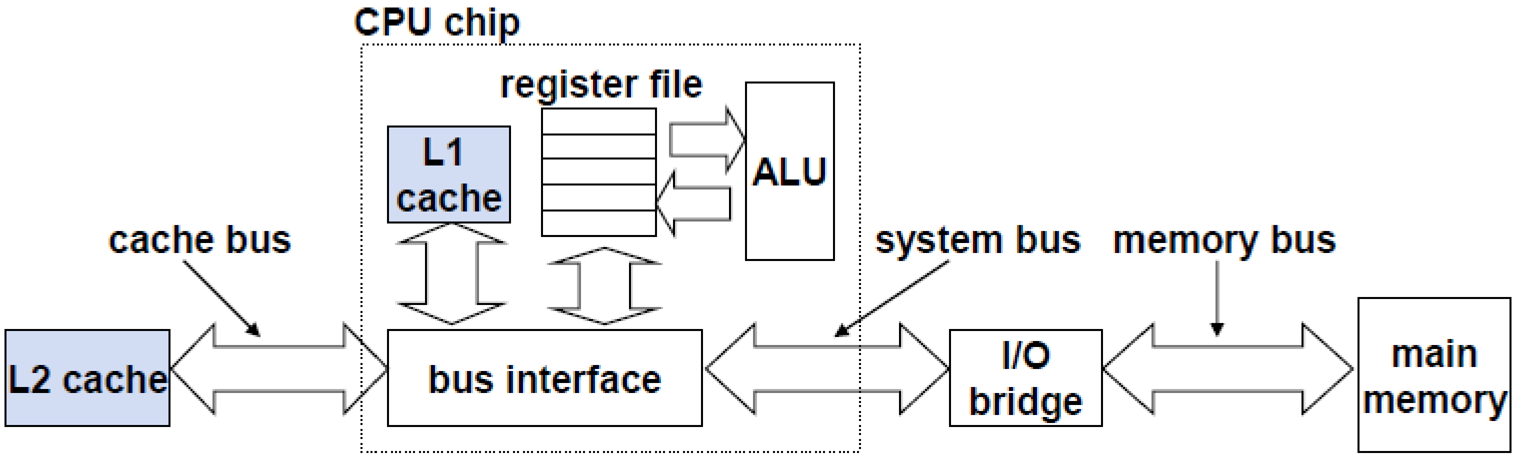
Source of slowness: CPU and memory speed



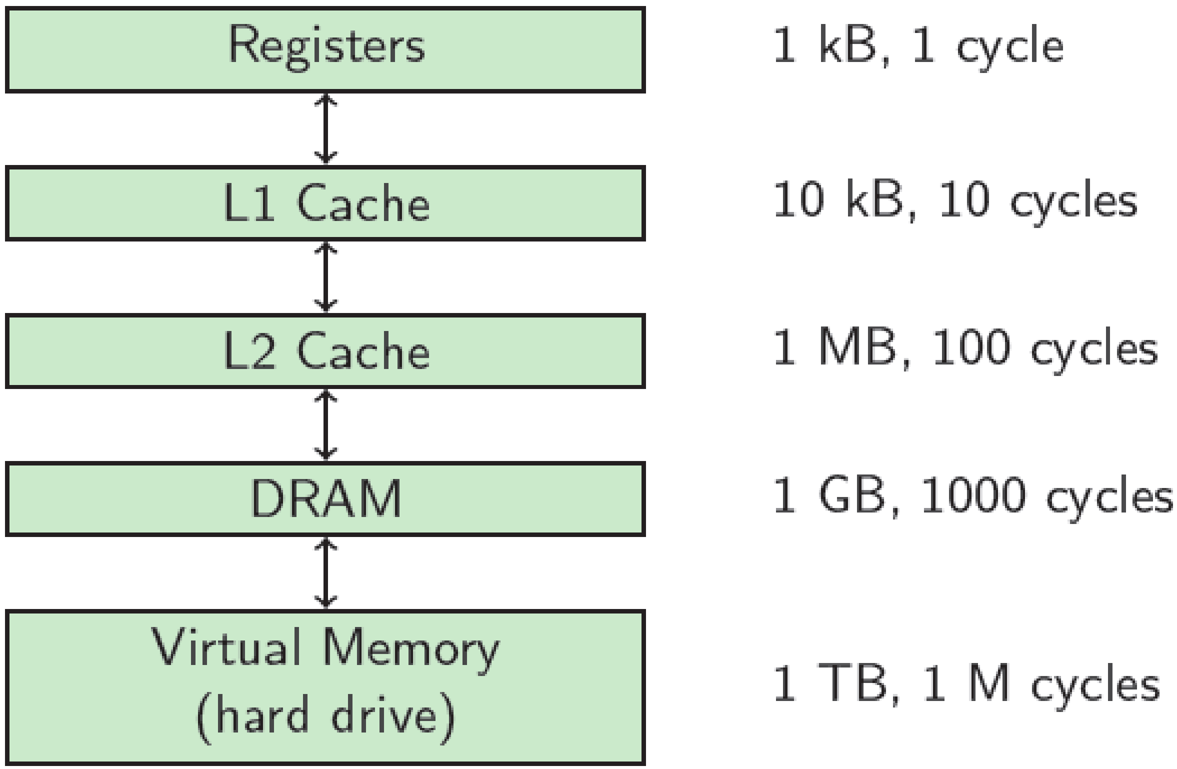
From Hennessy and Patterson,"Computer Architecture:
A Quantitative Approach," 3rd Edition, 2003, Morgan Kaufman Publishers.

Improving effective memory latency using cache memories (1)

- Put a look-up table of recently used data onto the CPU chip.
- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
- CPU look first for data in L1, then in L2,..., then in main memory



Hierarchy of increasingly bigger, slower memories



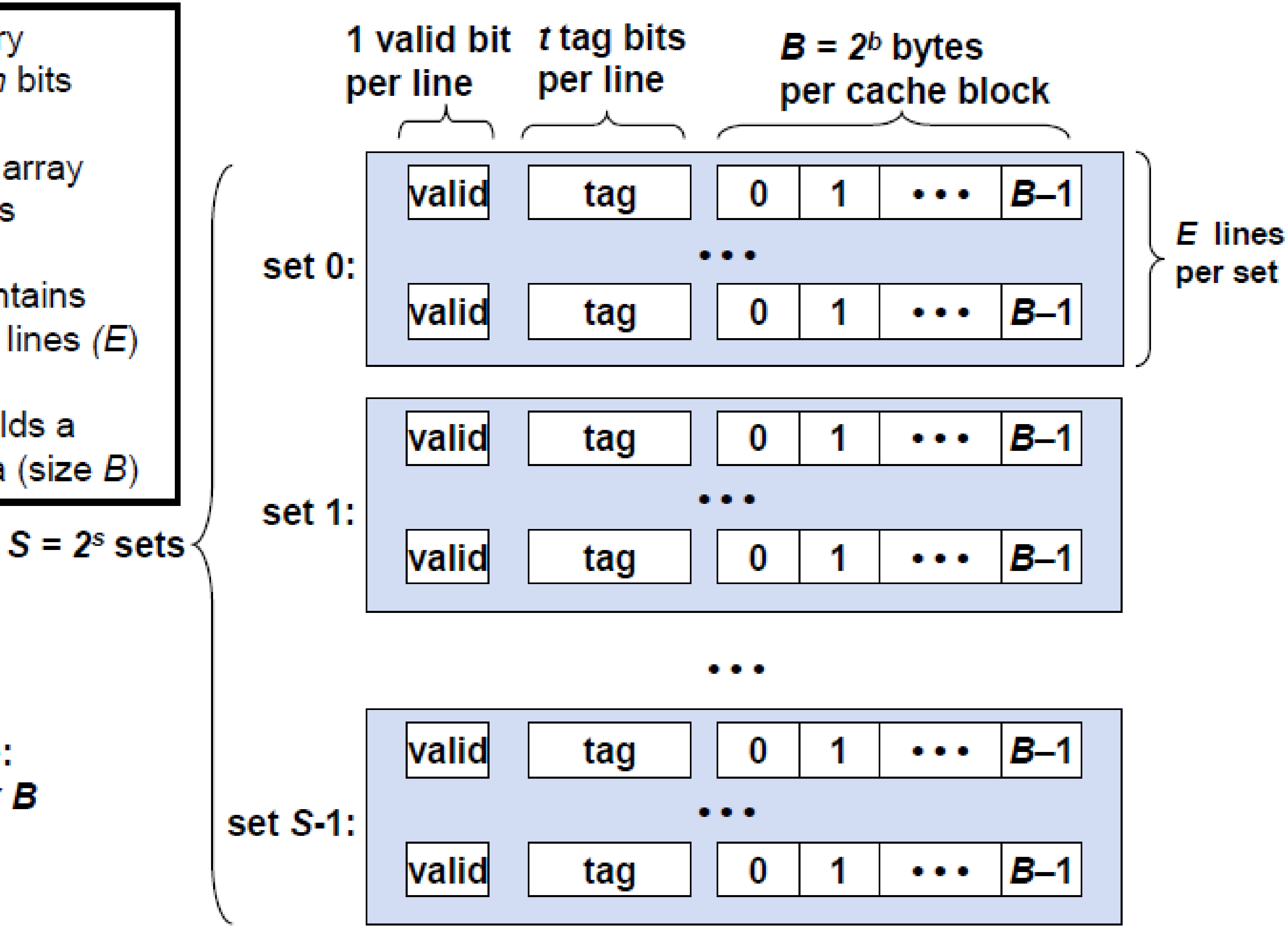
Organization of a cache memory

Each memory address is *m* bits

Cache is an array of $S = 2^s$ sets

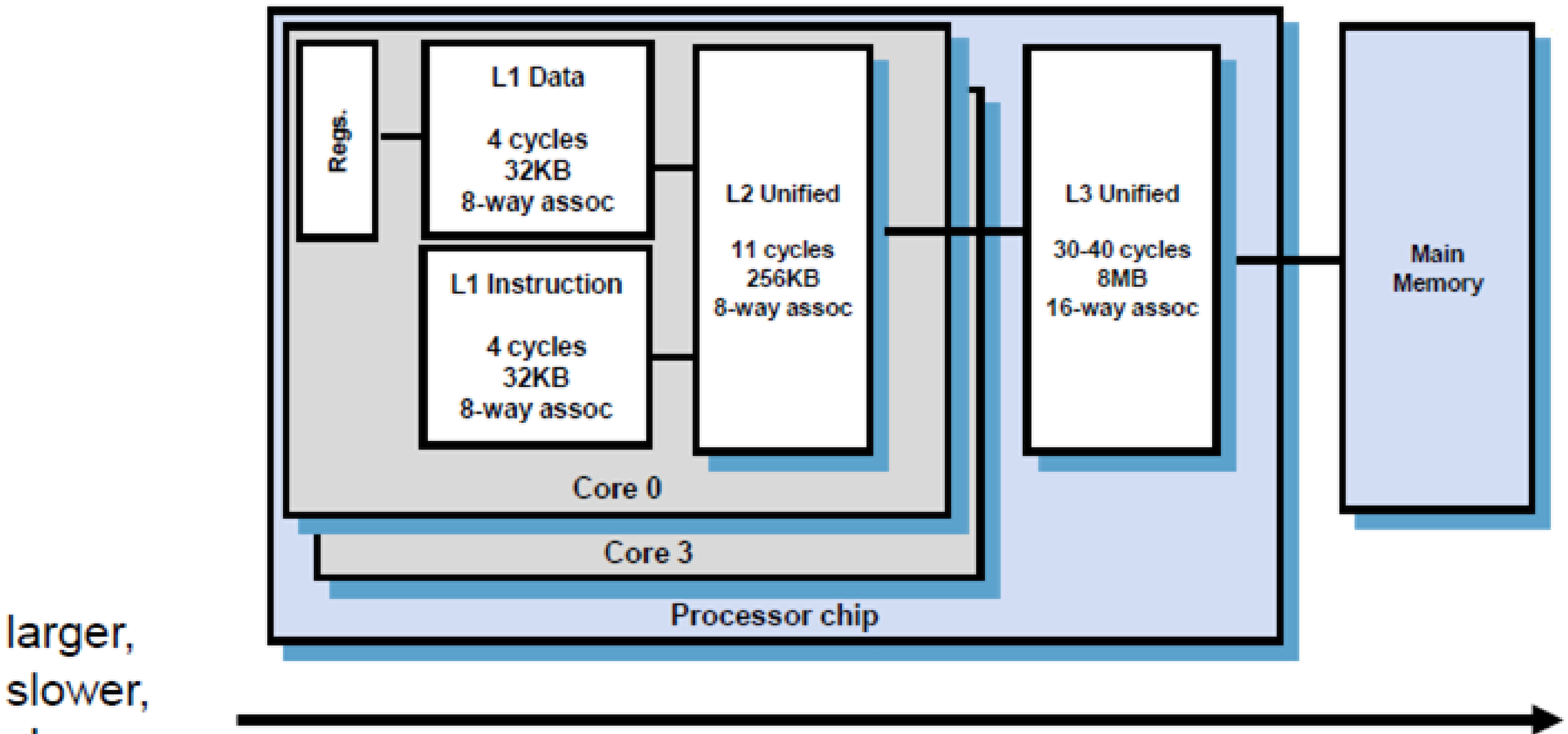
Each set contains one or more lines (E)

Each line holds a block of data (size B)



Cache size:
$C = S \times E \times B$
data bytes

Core i7 cache hierarchies



Size:	32KB	256KB	8MB
E:	8-way	8-way	16-way
Access:	4 cycles	11 cycles	30-40 cycles

Improving effective memory latency using cache memories (2)

Example. Consider to use a 1GHz CPU with a latency of 100 ns DRAM, and a cache of size 32KB with a latency of 1 ns to multiply two matrices A and B of dimensions 32 × 32.

Fetching A and B into cache corresponds to fetching **2K** words, taking 200 μs. Multiplying A and B takes $2n^3$ operations = **64K** operations, which can be performed in 16K cycles (or 16 μs) at 4 instructions per cycle.

The total time for computing = 200 + 16 μs.

Peak computing rate = **64K**/216 μs = 303 MFLOPS.

Cache performance measurements (1)

- Miss rate

  -- Fraction of memory references not found in cache

- Hit time

  -- Time to deliver a line in the cache to the processor, including time to determine whether the line is in the cache

- Missing penalty

  -- Additional time required because of a miss

Cache performance measurements (2)

- Big difference between a hit and a miss

Example. Assume that cache hit time is 1 cycle, and miss penalty is 100 cycles. A 99% hit rate is twice as good as 97% rate.

-- Average access time

1. 97% hit rate: 0.97* 1 + 0.03*(1+100) = 4 cycles
2. 99% hit rate: 0.99*1 + 0.01*(1+100) = 2 cycles

Writing cache-friendly code (1)

- Principle of **locality**:

  -- programs tend to reuse/use data items recently used or nearby those recently used

  -- *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

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

Data

-- Reference array elements in succession: spatial locality

-- Reference “sum” in each iteration: temporal locality

Instructions

-- Reference instructions in sequence: Spatial locality

-- Cycle through loop repeatedly: Temporal locality

How caches take advantage of temporal locality

- The first time the CPU reads from an address in main memory, a copy of that data is also stored in the cache.

  -- The next time that same address is read, the copy of the data in the cache is used instead of accessing the slower DRAM

- Commonly accessed data is stored in the faster cache memory

How caches take advantage of spatial locality

- When the CPU reads location *i* from main memory, a copy of that data is placed in the cache.
- Instead of just copying the contents of location *i*, we can copy several values into the cache at once, such as the four words from locations *i* through *i*+3.
  - If the CPU does need to read from locations *i*+1, *i*+2 or *i*+3, it can access that data from the cache.

Writing cache-friendly code (2)

In C/C++ language, array is stored in row-major order in memory

```
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
}
```

```
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
}
```

Assume that there is a cache with size of 4-byte words, 4-words cache blocks.

Left code has miss rate = ¼ = 25%

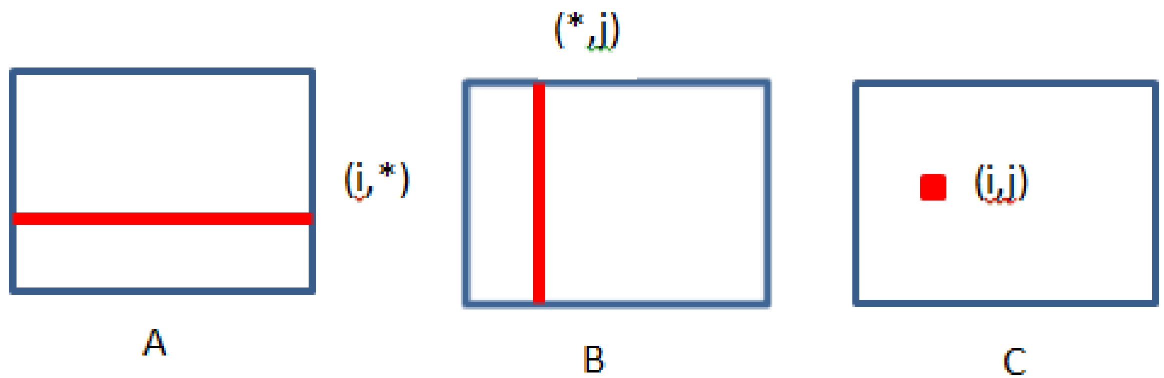
Right code has miss rate = 100%

Rearranging loops to improve locality

Miss rate analysis for matrix-matrix multiplication

- Assume a single matrix row does not fit in L1, each cache block holds 4 elements, and compiler stores local variables in registers.

```
/* 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;
    }
}
```

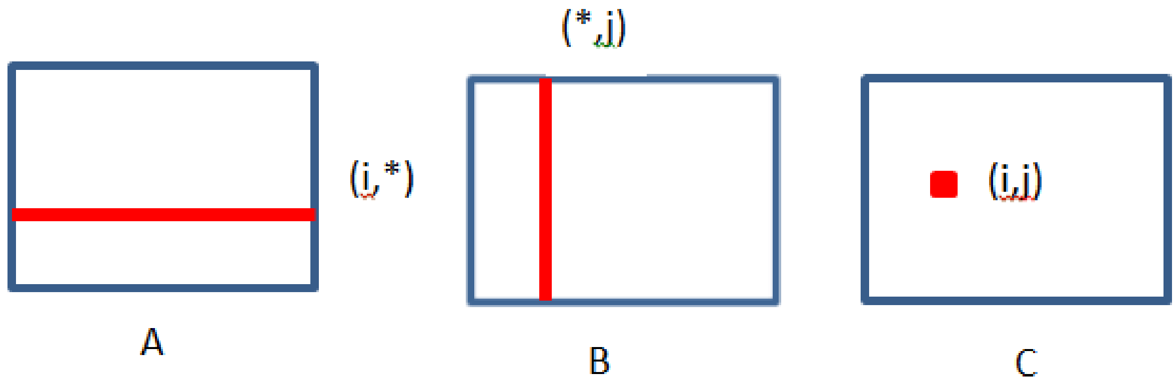


Per iteration

Loads	Stores	A misses	B misses	C misses	**Total misses**
2	0	0.25	1.00	0.00	1.25

/* 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;
    }
}
```

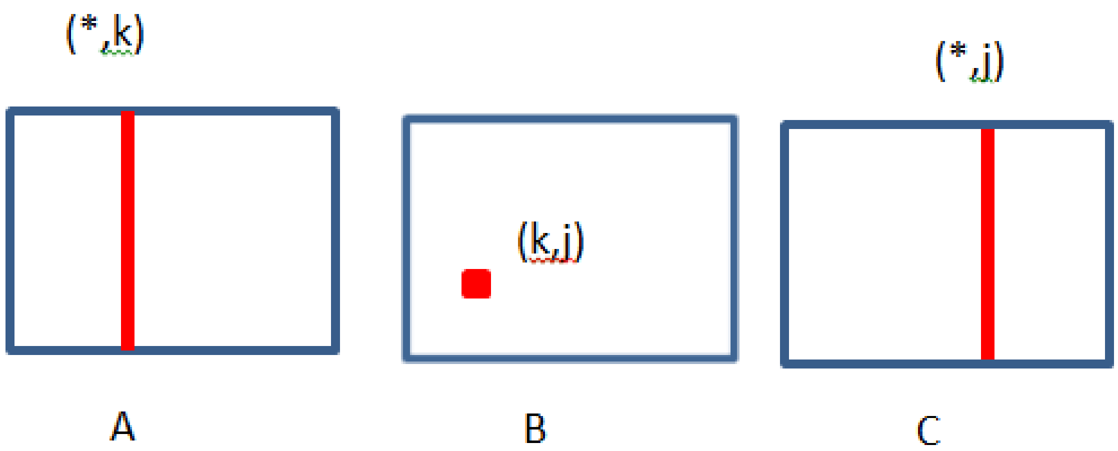


Per iteration

Loads	Stores	A misses	B misses	C misses	**Total misses**
2	0	0.25	1.00	0.00	1.25

/* 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;
     }
}
```



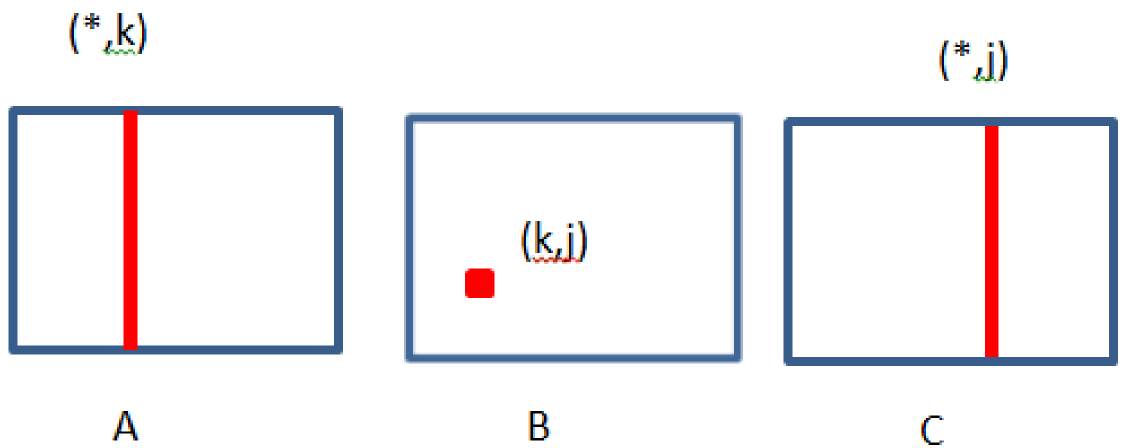
Per iteration

Loads	Stores	A misses	B misses	C misses	Total misses
2	1	1.00	0.00	1.00	2.00

- Scan A and C with stride of n
- 1 more memory operation

/* 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;
    }
}
```

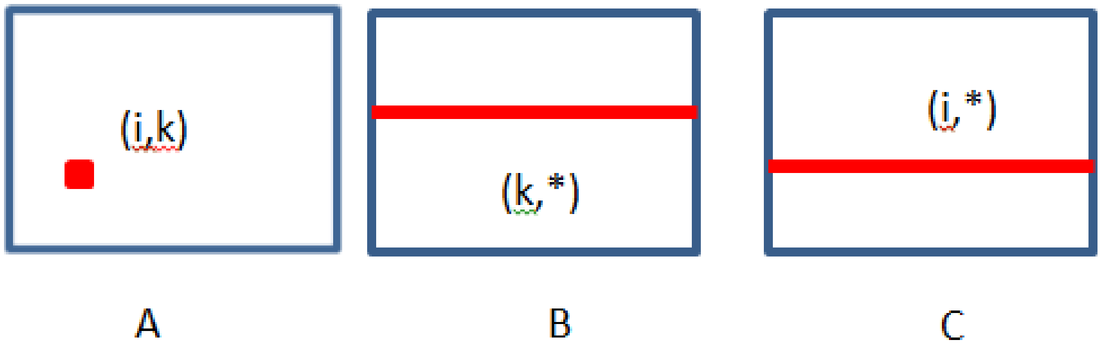


Per iteration

Loads	Stores	A misses	B misses	C misses	Total misses
2	1	1.00	0.00	1.00	2.00

/* 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];
    }
}
```

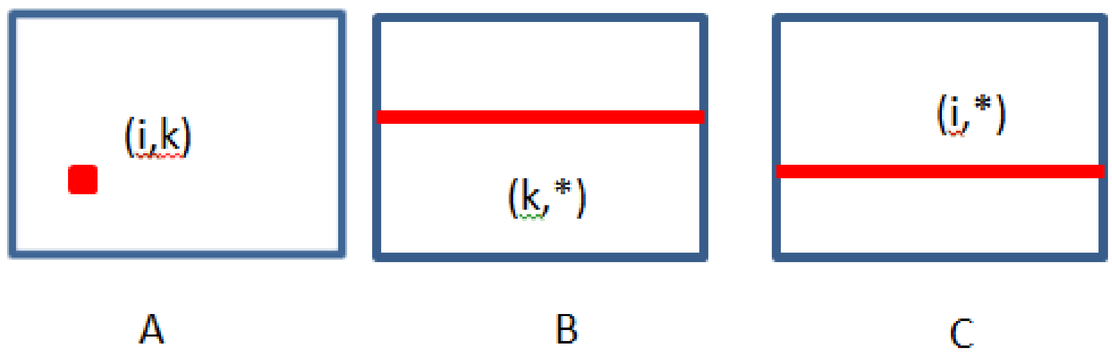


Per iteration

Loads	Stores	A misses	B misses	C misses	Total misses
2	1	0.00	0.25	0.25	0.50

Trade-off: one memory operation – fewer misses

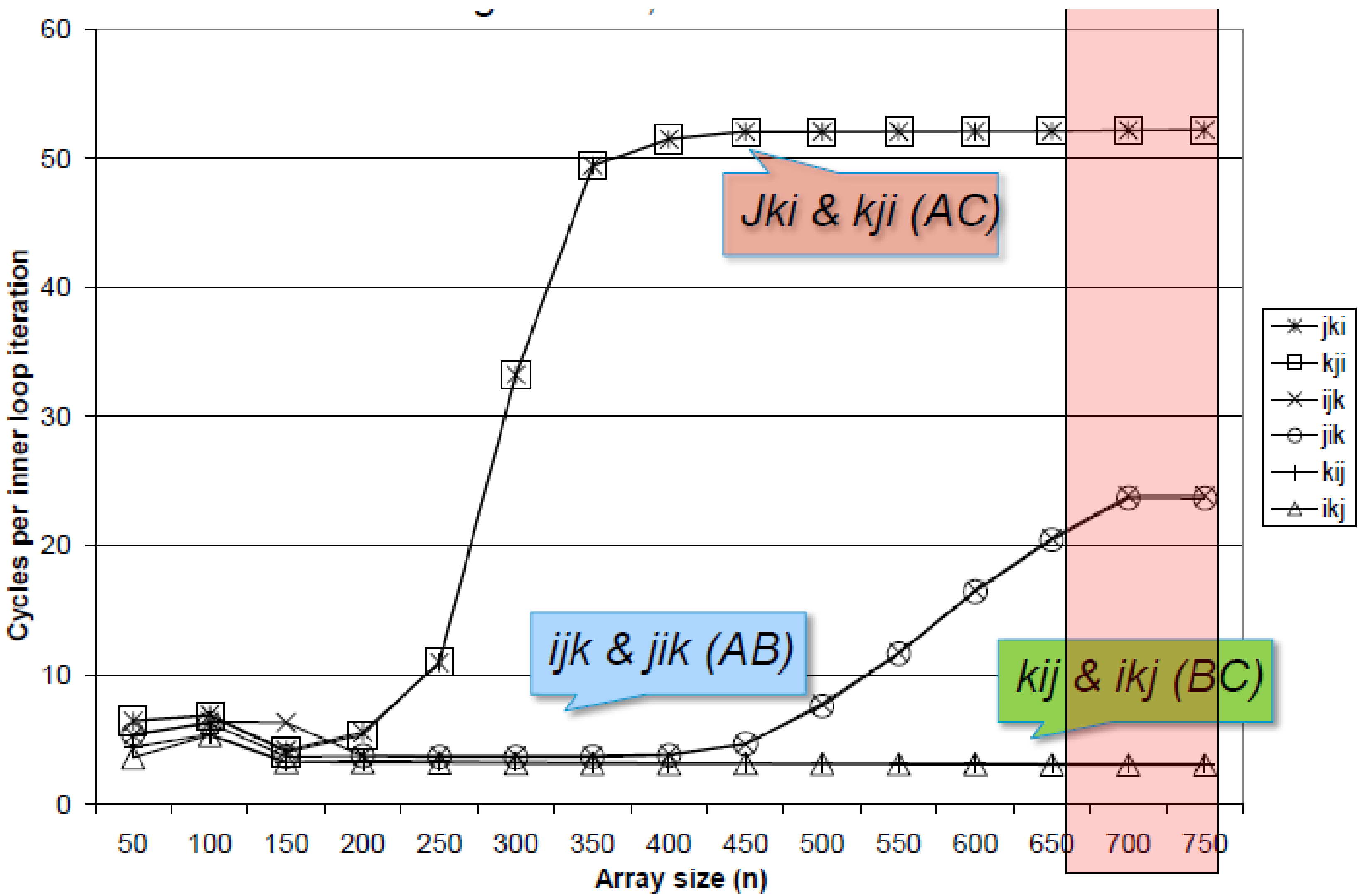
```
/* ikj */
for (i = 0; i < n; i++)
{
    for(k = 0; k < n; k++)
    {
        r = a[i][k];
        for(j = 0; j < n; j++)
            c[i][j] += r * b[k][j];
    }
}
```



Per iteration

Loads	Stores	A misses	B misses	C misses	**Total misses**
2	1	0.00	0.25	0.25	0.50

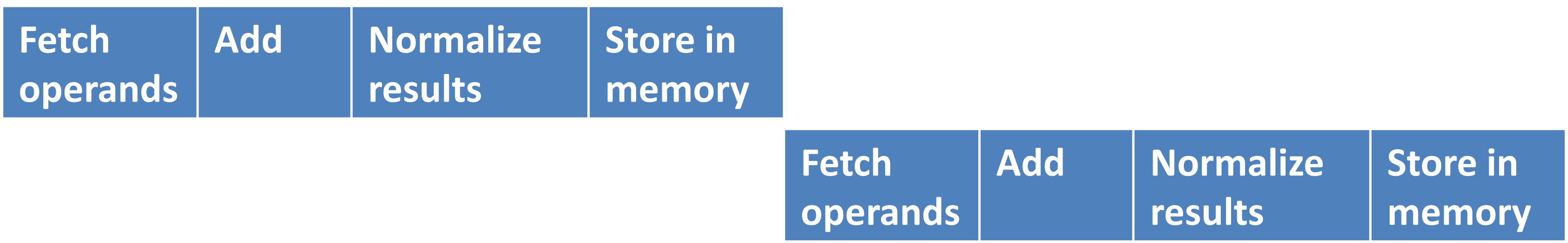
Matrix-matrix multiplication performance



From EECS213 Northwestern University

Sequential Operation

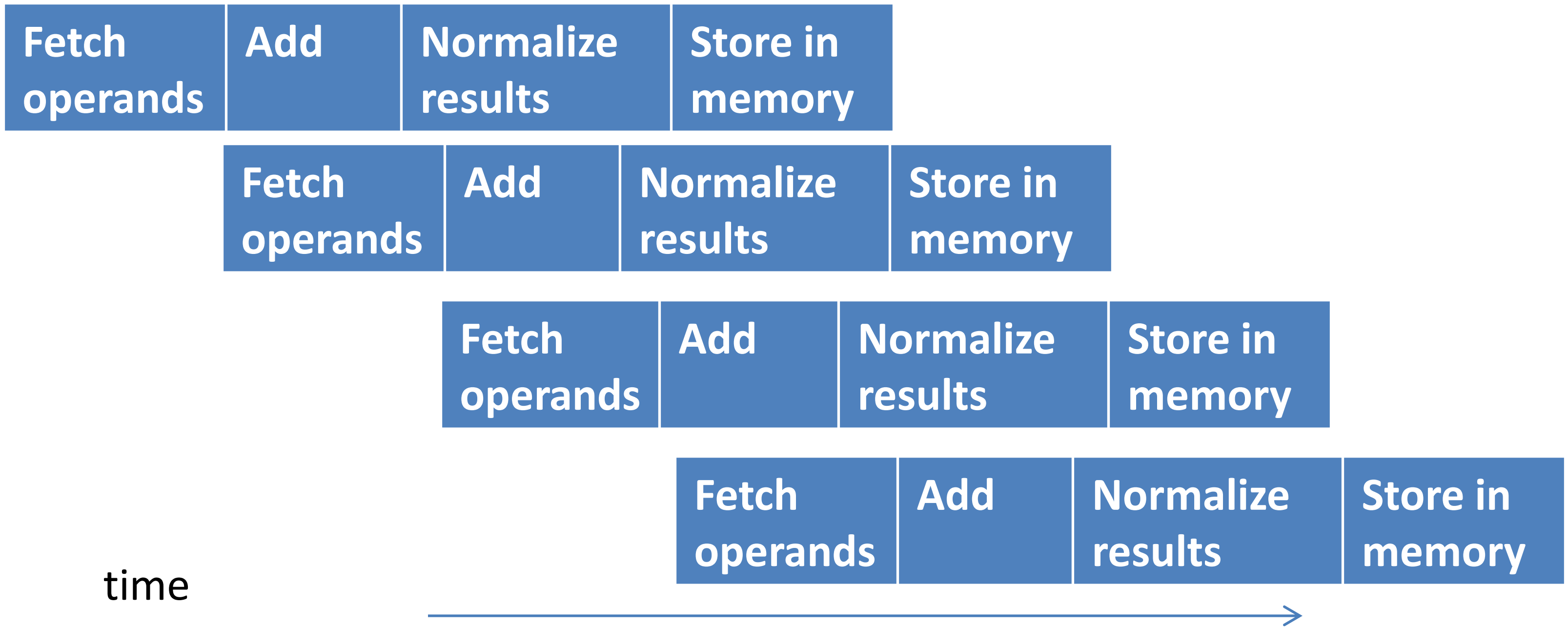
```
Double x[100], y[100], z[100];
 for (i = 0; i < 100; i++)
     z[i] = x[i] + y[i];
```



Solution: Pipelining

Divide a computation into stages that can support concurrency.

```
Double x[100], y[100], z[100];
 for (i = 0; i < 100; i++)
     z[i] = x[i] + y[i];
```



Another improvement: Vector processor pipeline.

Example: Cray 90

Loop unrolling:

```
for (i = 0; i < 100; i++)
    do_a(i);
```

```
for (i = 0; i < 50; i+=2)
{
    do_a(i);
    do_a(i+1);
}
```

Software pipelining

```
for (i = 0; i < 100; i++)
{
    do_a(i);
    do_b(i);
}
```

```
for (i = 0; i < 50; i+=2)
{
    do_a(i); do_a(i+1);
    do_b(i); do_b(i+1);
}
```