Fundamentals of MCS
Computer Architecture Part 1

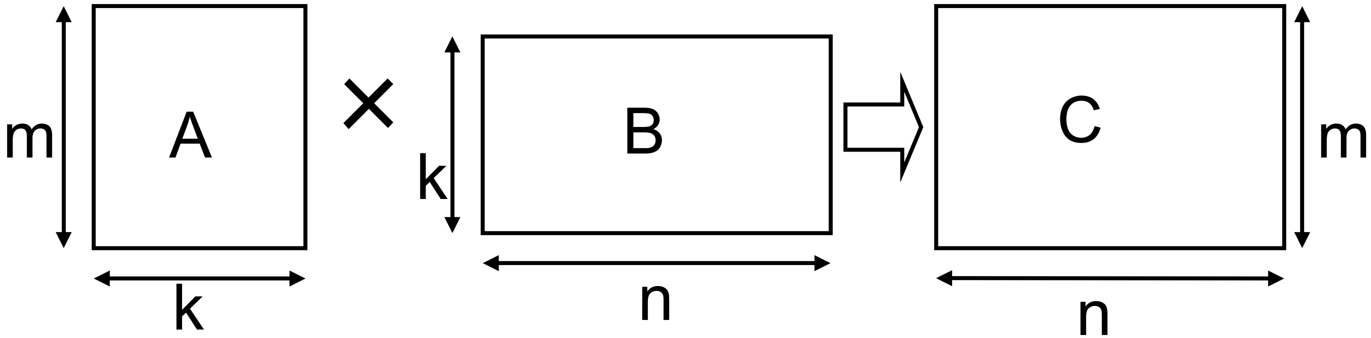
Toshio Endo
endo@is.titech.ac.jp
www.el.gsic.titech.ac.jp

Why Computer Architecture is Important for Algorithm/Software?

- Understanding CPU/memory architecture is important for “speed” of computation
  - Multi-core
  - SIMD
  - Cache, memory system
  - Network
- Improvement of algorithm complexity is (of course) important, but architecture-aware approach is becoming more important

Example Computation: Matrix Multiply (matmul)

Multiplying a (m × k) matrix and a (k × n) matrix




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

Complexity : O(mnk)

Here, we assume C_{ij} is represented as C[j][i] (column-major)

Variants in matmul Implementation

```
for (i = …) {
  for (j = …) {
    for (l = …) {
      …
    }
  }
}
```

```
for (j = …) {
  for (i = …) {
    for (l = …) {
      …
    }
  }
}
```

```
for (l = …) {
  for (i = …) {
    for (j = …) {
      …
    }
  }
}
```

- What happens if we exchange the sequence of for loop?
  - We have 6 implementations: IJL, ILJ, JIL, JLI, LIJ, LJI
  - This change does affects neither computed results nor compute complexity of O(mnk)
  - Only the sequence of operations are changed

Effects of Software Implementation

- Performance of different 6 implementations of matmul
  - Written in C language, not parallelized, gcc 4.3.4, -O2
  - Elements are "double" type
  - m=n=k=1024
  - On a single node of TSUBAME2 supercomputer

Imple	IJL	ILJ	JIL	JLI	LIJ	LJI
Time (sec)	8.51	17.5 Slow!	8.52	1.11 Fast!	17.5 Slow!	1.30

Although all implementations have same complexity, but largely different in the computation speed

What is the cause of the difference?
→ We should learn computer architecture

Speed of “matmul”

- Actual “Flops” achieved by the software is calculated by

  (*The number of FP operations / Elapsed time)*

- In “matmul”, the number of FP operations is

  2mnk = 2 x 1024 x 1024 x 1024

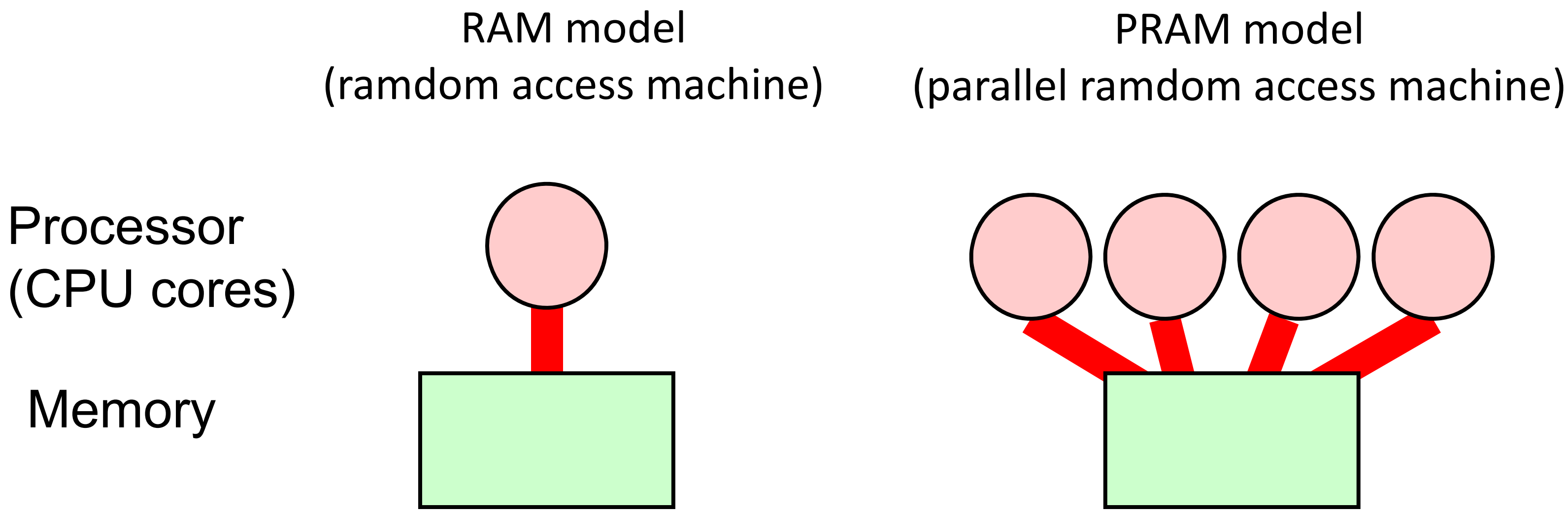
Imple	IJL	ILJ	JIL	JLI	LIJ	LJI
Time(sec)	8.51	17.5	8.52	1.11	17.5	1.30
Speed (GFlops)	0.25	0.12	0.25	1.92	0.12	1.65

What are the reasons of the difference?
Is the fastest speed, 1.92GFlops, sufficient?
← Knowledge of architecture is required

Keywords in Recent Architecture

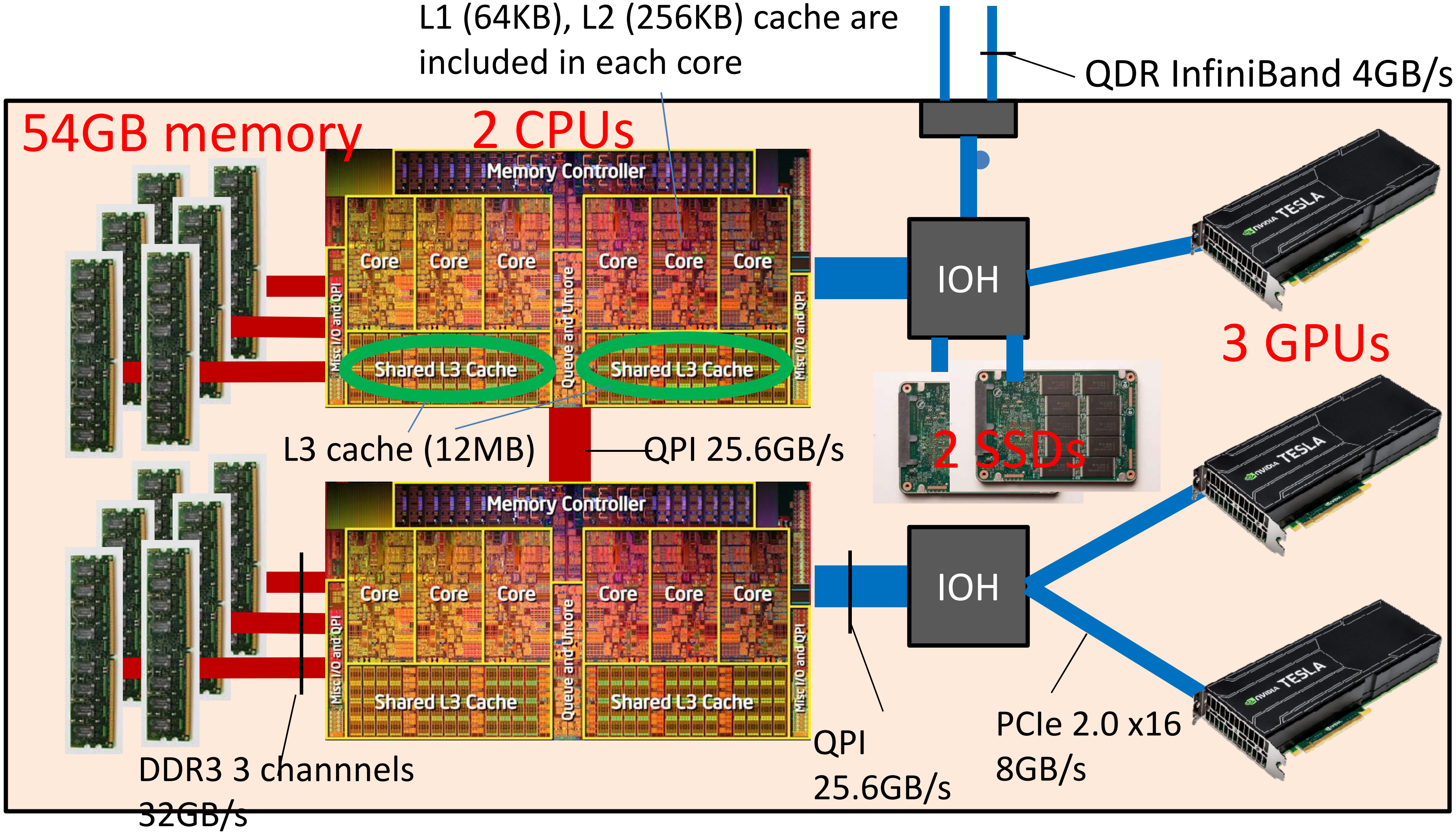
- Multi-core
- SIMD
- Cache, memory system
- Network

What are Components of Computers? Very Simplified Models



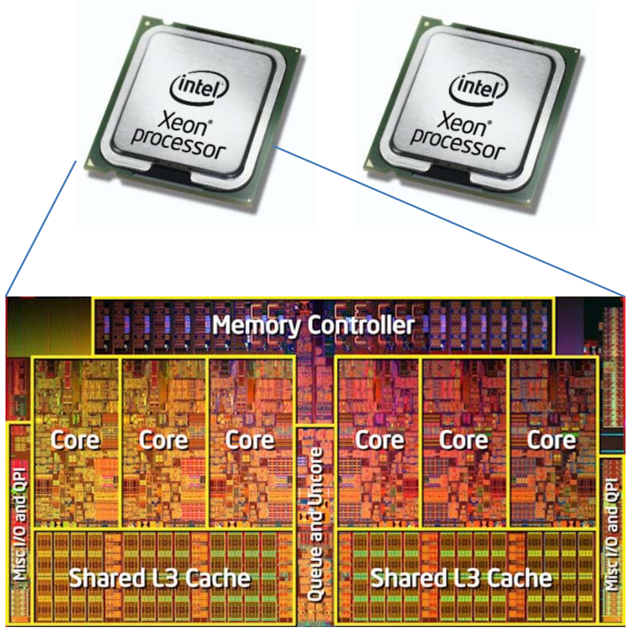
- These model does not explain the difference in 6 programs…

An Example of Real Computers:
A Node in TSUBAME2 Supercomputer
(still simplified)



TSUBAME2 Node Architecture
(GPUs are omitted)

- A node has 2 CPUs
  - Intel Xeon X5670
- Each CPU has 6 CPU cores (multi-core)
  - 12 CPU cores share 54GB node memory
- Each core works at 2.93GHz
- On each clock, each core can execute 4 FP operations
  - By using SIMD instructions called SSE
    - SIMD: Single Instruction Multiple Data
  - Latest CPUs can execute 8 per clock



4 x 2.93G(1/sec) x 6 x 2 = 140.8 (Gflops) — Theoretical Performance

Even in the fastest matmul implementation, 1.92GFlops is far lower than 140.8GFlops. Why?

Limitation of “matmul”

- Only 1 core is used
- SIMD instructions are not used
  - Recently clever compilers can use them, but it is not the case now

→Considering above, 1.92GFlops is still lower than 2.93GFlops (about 65%)

→This is mainly due to inefficiency in cache and memory usage

Performance of Optimized Library

- BLAS (Basic Linear Algebra Subprograms)
  - An API for matrix operations
- Implementation: GotoBLAS, MKL, ACML…
  - Highly optimized for each CPU architecture

Imple	IJL	ILJ	JIL	JLI	LIJ	LJI	GotoBLAS (1core)	GotoBLAS (12cores)
Time(sec)	8.51	17.5	8.52	1.11	17.5	1.30	0.182	0.0181
Speed (GFlops)	0.25	0.12	0.25	1.92	0.12	1.65	11.8	118.8

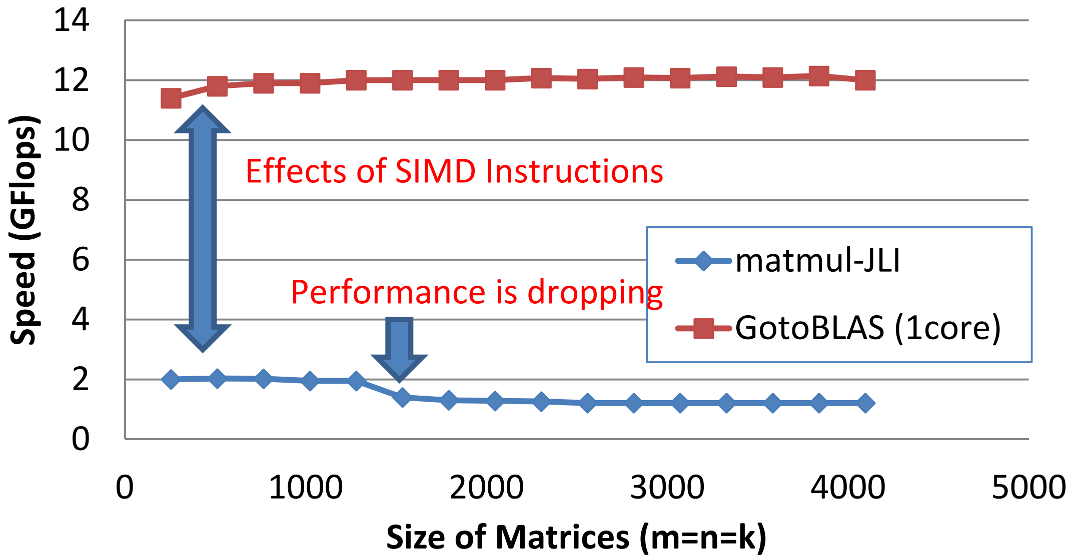
Very Fast!

Discussion of Performance (1)

- 1 core performance (11.8GFlops) is almost the same as theoretical one (11.7GFlops)
  - This is *too good*??. Possibly “Intel turbo boost” is working
  - Turbo boost: if node load is sufficiently low, working core is boosted (up to 3.2GHz here)
- 12 core performance is x10.07 faster than 1 core
  - x10.07 speed up is fairly good, but less than 12
  - Effects of turbo boost?
  - Memory contention?

Discussion of Performance (2)

- Changing matrix sizes

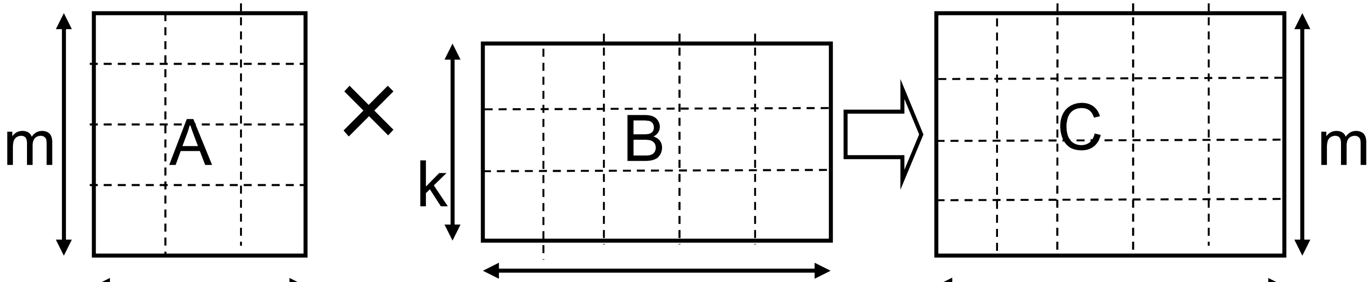


- (Naïve) Simple matmul suffers from more “cache-misses” when problem gets larger
- Optimized GotoBLAS is not only fast, but stable toward the change of problem size

Optimizations in GotoBLAS

- Effectively use multi-core
- Effectively use SIMD instructions
- Effectively use memory system

  Cache-blocking:

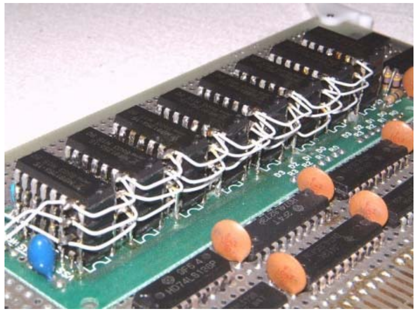


  – Matrices are broken into "blocks", each of which are smaller than cache size
  – Sometimes data replacement occurs
  – Also optimized to reduce TLB misses

For details, please refer:

Cf: K. Goto, R. Geijn: Anatomy of high-performance Matrix Multiplication, ACM TOMS 2008

CPU and Memory: Past and Present

	Around 1980	Present
CPU	2MHz → 1clock = 500ns 	**x1000** 2GHz → 1clock = 0.5ns 
Memory	Access time = 2000ns(?) 	**x40(?)** Access time = 50ns or more 

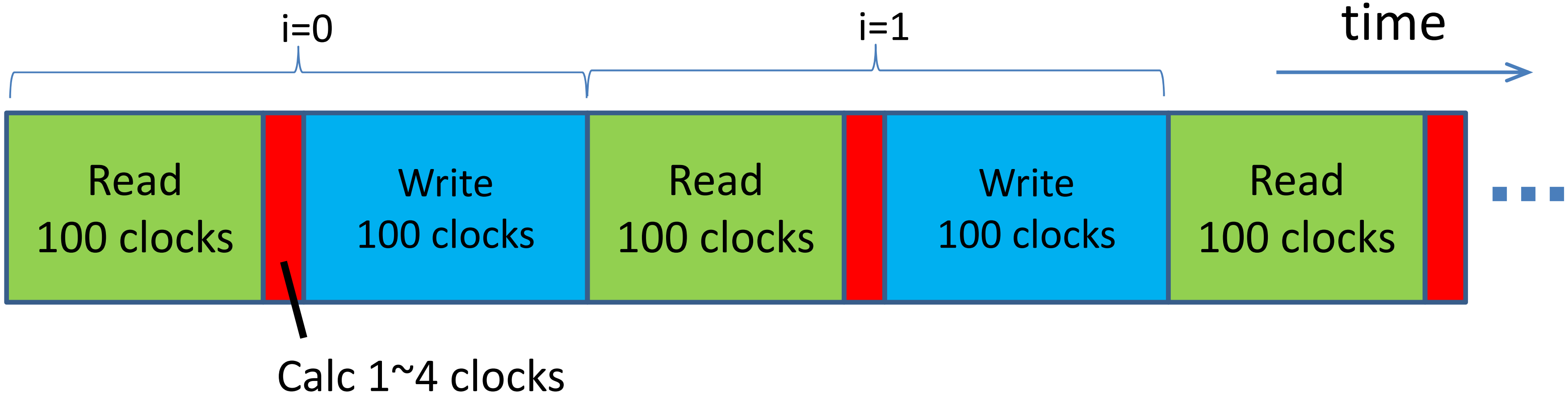
Memory Access Time

How long does a memory “read” instruction take?

Around 1980	Present
2MHz → 1clock = 500ns	2GHz → 1clock = 0.5ns
Access time = 2000ns(?)	Access time = 50ns or more
↓	↓
4 clocks	>100 clocks!!

What happens If Every Memory Access Takes 100 clocks?

```
for (i = 0; i < n; i++) A[i] = A[i]*2.0;
```



This is very insufficient!

Computation speed would be only 10MFlops

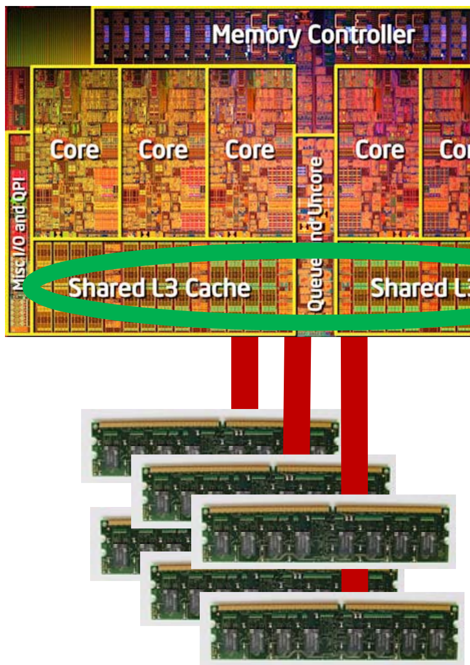
To alleviate this problem,

cache memory has been invented in 1968.

It became popular around 1985

Cache Memory

- Fast and small memory (usually) included in CPU
- Used to store data that have been recently accessed
- Used automatically --- Sometimes programmers do not know existence of cache memory



Memory Hierarchy of TSUBAME2 Node

CPU: Intel Xeon X5670 (Westmere)

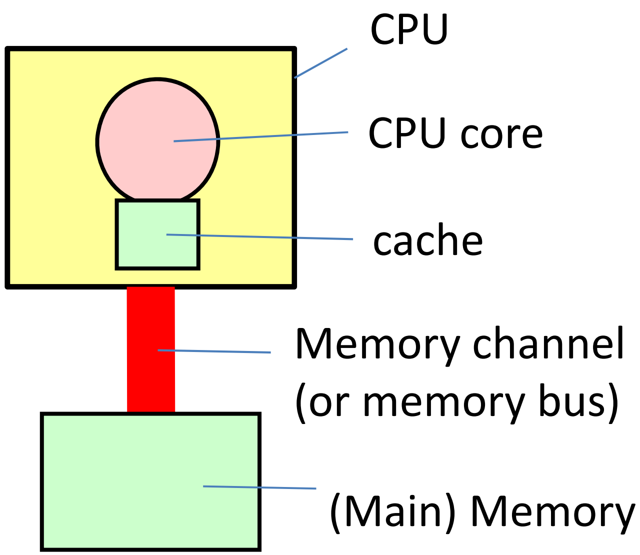
	Capacity	Access Time
Level1 cache	64KB	~4 clocks
Level2 cache	256KB	~10 clocks
Level3 cache	12MB (shared by 6 cores)	~25 clocks or more
Main Memory	54GB (shared by 12 cores)	100 clocks or more

corrected

Figures for access time are from Web, and may be inaccurate

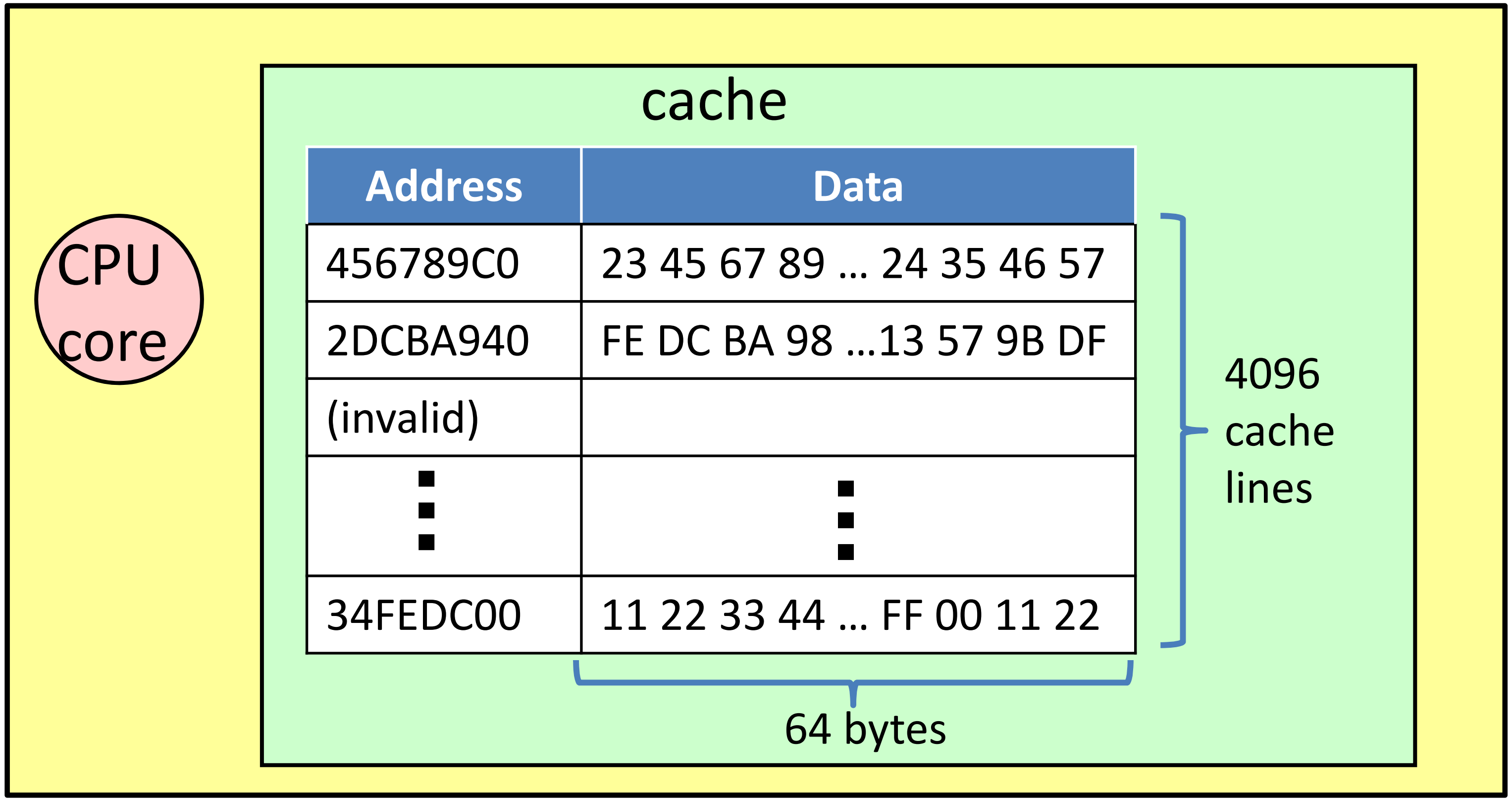
Assumption in This Lecture (at First)

- Modern CPUs has hierarchical cache memory (Level 1 cache, Level 2 cache…)
- For simplicity, we consider
  - A single level cache
    - Capacity: 256KB
    - Cache line size: 64B
  - 32 bit addresses
    - Though recent CPUs have 64 bit addresses
  - Single core CPU at First, and Multicore CPU later



Example Cache and Cache Line

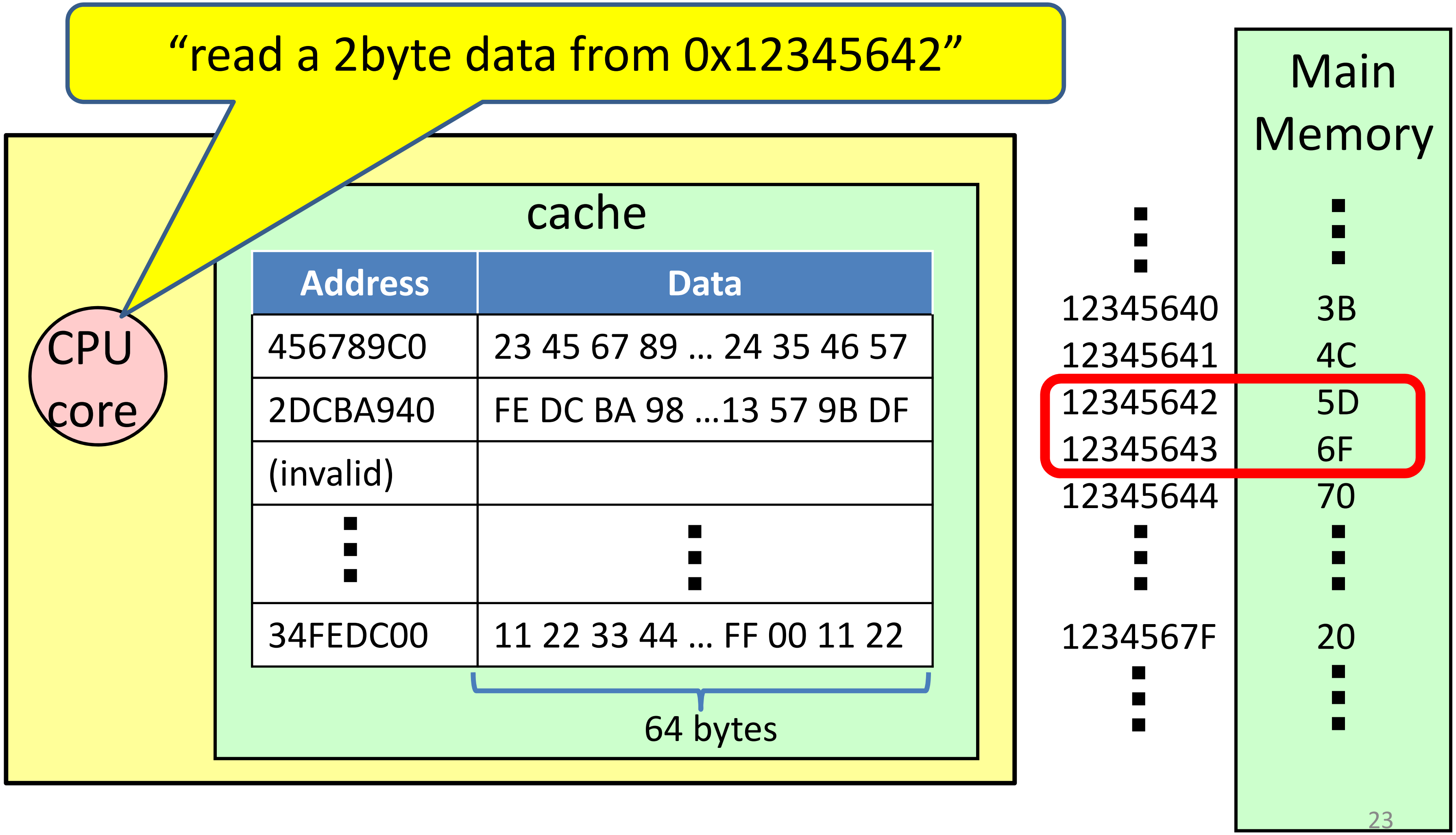
- There are “units” for data movement, called cache lines
  - We assume each cache line has 64bytes
  - 256KB cache holds 4096 (=256K/64) cache lines



Address	Data
456789C0	23 45 67 89 … 24 35 46 57
2DCBA940	FE DC BA 98 …13 57 9B DF
(invalid)	
⋮	⋮
34FEDC00	11 22 33 44 … FF 00 11 22

Memory Access with Cache (1)

- When CPU core executes a read instruction



Address	Data
456789C0	23 45 67 89 … 24 35 46 57
2DCBA940	FE DC BA 98 …13 57 9B DF
(invalid)	
⋮	⋮
34FEDC00	11 22 33 44 … FF 00 11 22

Memory Access with Cache (2)

1. Calculate the start address of cache line that includes target address
   - 0x12345642 & 0xFFFFFFC0 → 0x12345640
   - Cache line to be accessed is [0x12345640, 0x1234567F] (64=0x40bytes)
2. Search address 0x12345640 in cache

   2-1: If found, cache hit (We go to Step 5.)

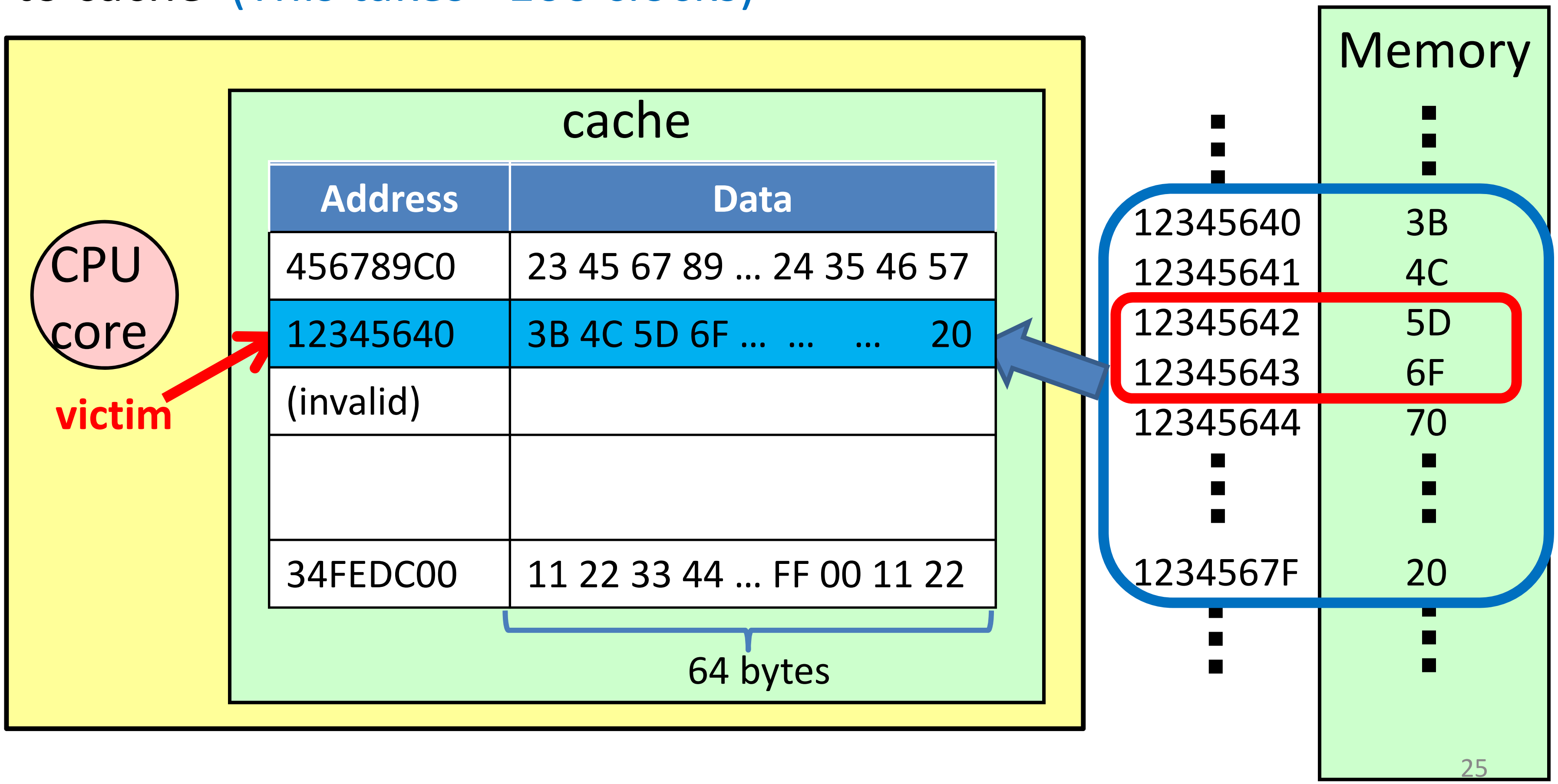
   2-2: If not found, cache miss (This is the case now)

Memory Access with Cache (3)

Cache Miss Case

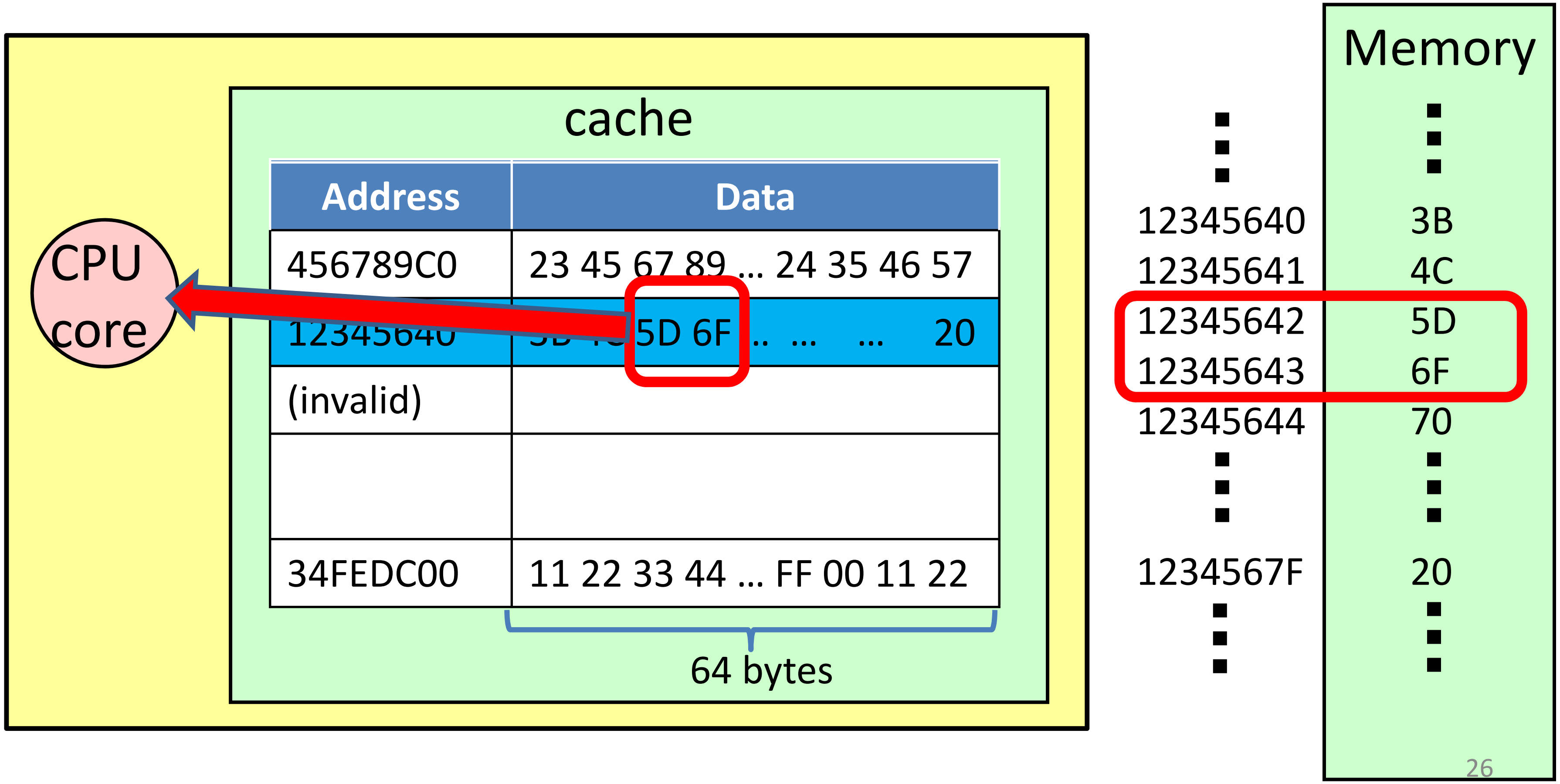
3. Select a “victim” line in cache, to be deleted

4. Copy 64byte data from [0x12345640, 0x1234567F] in memory to cache (This takes >100 clocks)



Memory Access with Cache

5. Deliver the desired data to CPU core



Characteristics of CPU with Cache

- Time to execute a memory access instruction is not constant
  - In cache hit cases, a few clocks
  - In cache miss cases, >100 clocks
- Due to existence of cache lines, sequential memory access tends to raise higher cache hit ratio
  - Program A accesses to 12345642, 12345644, 12345646…

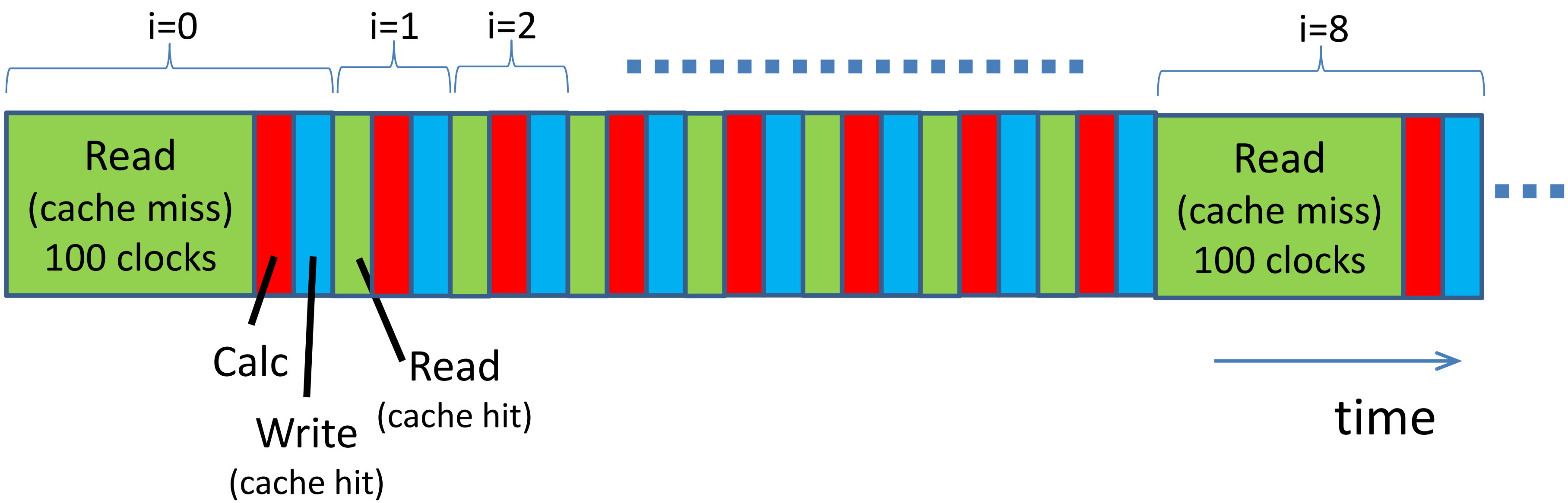
  → Good locality
  - Program B accesses to 12345642, 1234A000, 23456780…

  → Bad locality

Example of Sequential Access

```
for (i = 0; i < n; i++) A[i] = A[i]*2.0;
```

- We assume that cache is empty, when the programs begins
- We assume A[i] has double type (8Byte)



- Much more efficient than “No cache” CPU in p.20
- Actual CPU is even more efficient, due to pipelined execution

Deeper Insights: Cache Policy

- How the “victim” line is selected?
  - Direct mapping, or set associative or full associative
- “Write” is more complex than read!
  - Write through, or Write back

- These policies are implemented by processor makers (Intel, AMD, NVIDIA…), so users cannot change it basically
- Memory access is done by hardware (not software), too complex method is impractical
  - For example, there is no commercial CPU with full associative cache

Agenda of Architecture Part

- Memory System
  - Cache line, associativity, replacement algorithm
- Parallelism
  - Multi-core
  - SIMD
- Memory system and parallelism
  - Maintaining consistency of cache
- Network communication

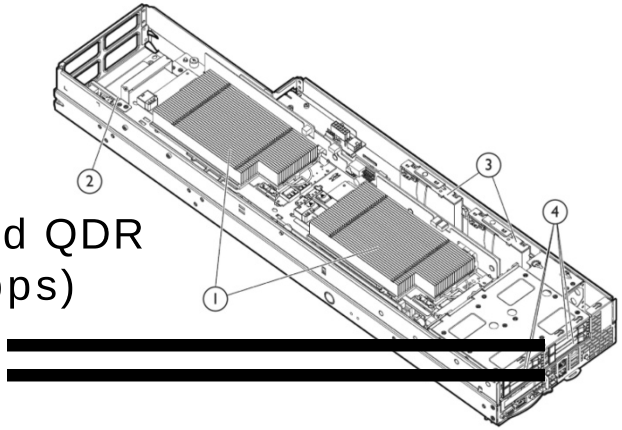
Next lecture is on Jan 12 (Tue)

Jan 18 will be cancelled

Appendix

TSUBAME2.5 Compute Node

Thin Node



Infiniband QDR x2 (80Gbps)

HP SL390z

CPU: Intel Xeon X5670 (Westmere-EP)
2.93GHz x2 (12cores/node)
GPU: NVIDIA Fermi K20X x 3
1.31TFlops, 6GByte memory /GPU
Memory: 54GB DDR3-1333
SSD:60GBx2

Theoretical performance per node is

- CPU: 140.8 GFlops
- GPU: 3.93 TFlops
- CPU+GPU: 4.07 TFlops

96.5% of performance is contributed by GPUs

TSUBAME2.5 Supercomputer

- TSUBAME2.5 (mainly) consists of 1408 compute nodes
- Total Theoretical Performance (Double precision):

  5.7PFlops = 4.07TFlops x 1408
- Currently 25th supercomputer in the world
  - See http://www.top500.org