Lecture 23: End Semester Review

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

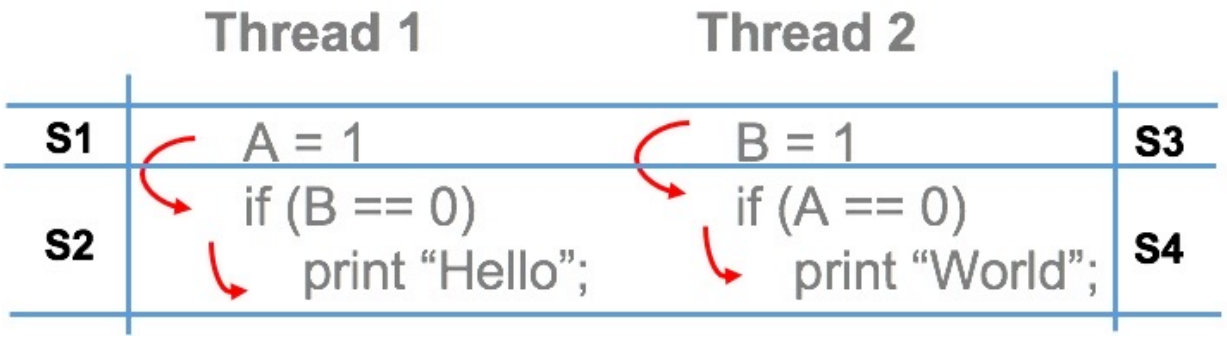
Today's Class

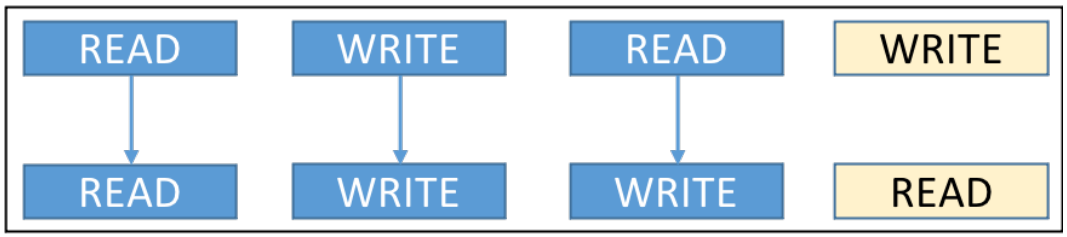
- End semester review

Hardware Memory Model (1/2)

- Memory latency continues to limit the performance of multicore processors
  - Several optimizations inside processors for hiding the load/store latency
    - As a side effect of these optimizations, load/store inside a program could be reordered, and hence may not happen in the source code order as expected by programmer
- Memory consistency model defines a set of rules for valid set of reordering of **two different memory accesses**
  - Both compiler and processor can perform reordering

Hardware Memory Model (2/2)

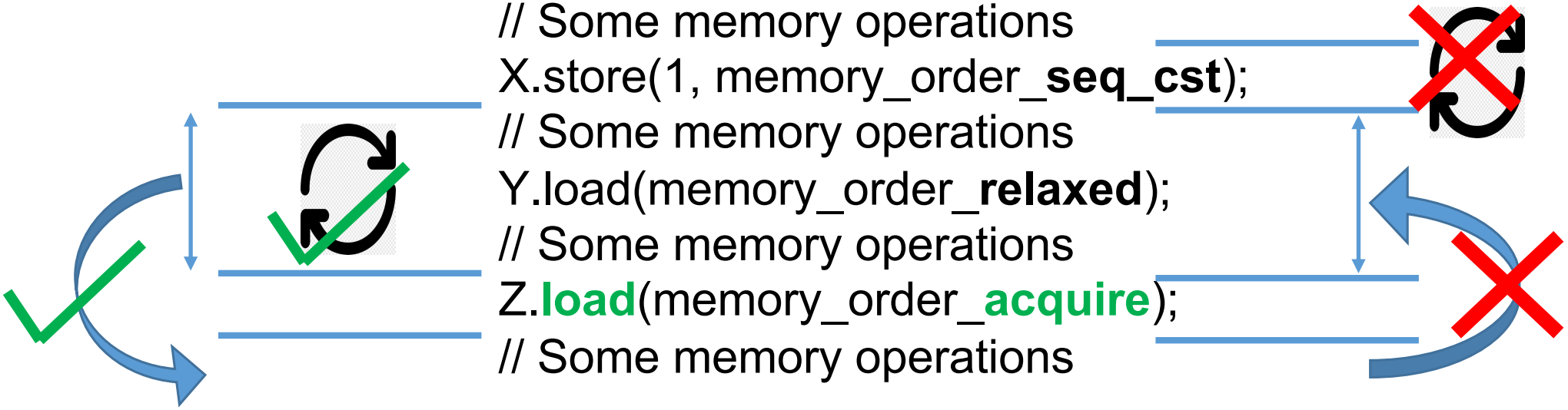




- x86-TSO memory model (Intel/AMD)
  - Write-Read reordering is only allowed
  - Due to presence of store buffers
- Store buffer
  - Temporarily holds write before they are committed to the cache
    - FIFO on x86
  - Size enough for ~50 stores on Intel processor (Skylake)

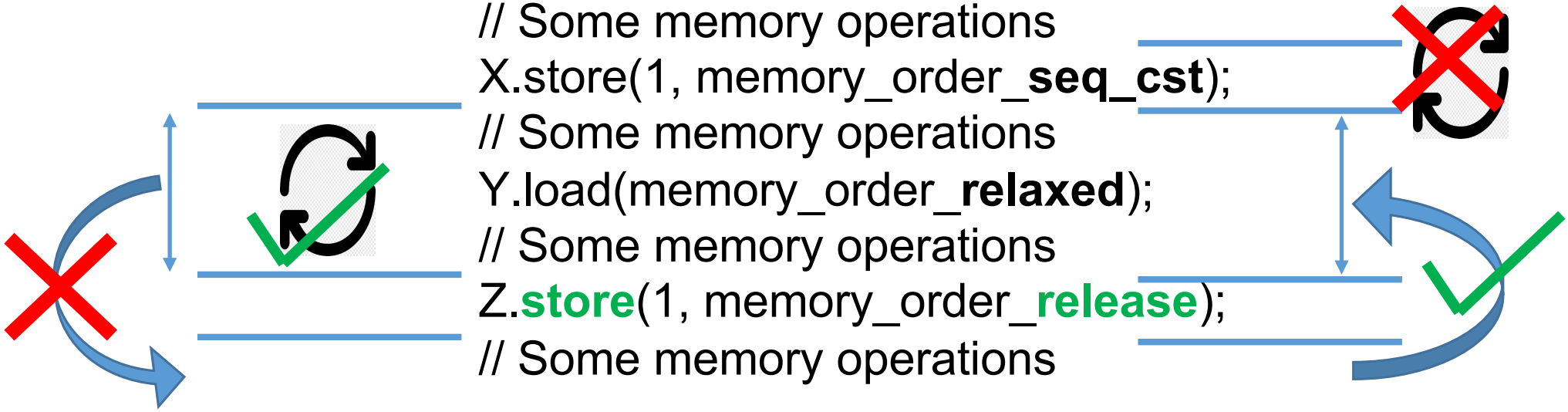
Language Memory Model (1/3)

```
std::atomic<int> X, Y, Z;

// Some memory operations
X.store(1, memory_order_seq_cst);
// Some memory operations
Y.load(memory_order_relaxed);
// Some memory operations
Z.load(memory_order_acquire);
// Some memory operations
```



Language Memory Model (2/3)

```
std::atomic<int> X, Y, Z;

// Some memory operations
X.store(1, memory_order_seq_cst);
// Some memory operations
Y.load(memory_order_relaxed);
// Some memory operations
Z.store(1, memory_order_release);
// Some memory operations
```



Language Memory Model (3/3)

std::atomic<bool> A(false), B(false);
int non_atomic = 0;

Thread 1

```
non_atomic = 10;
A.store(true, memory_order_release);
```

Synchronizes-with

Thread 2

```
if(A.load(memory_order_acquire) == true) {
    B.store(true, memory_order_release);
}
```

Synchronizes-with

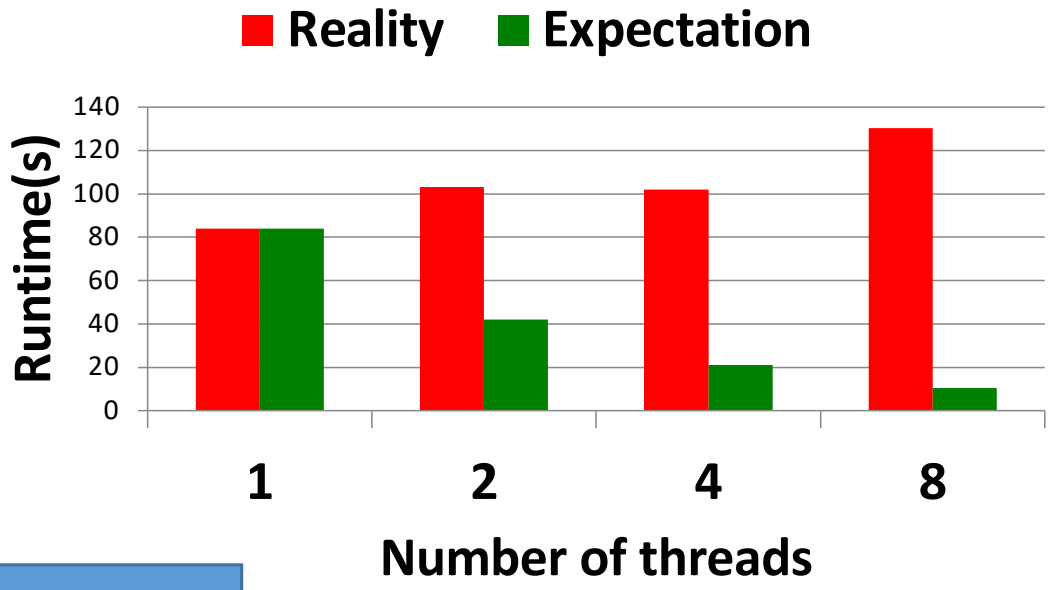
Thread 3

```
if(B.load(memory_order_acquire) == true) {
    assert(non_atomic == 10);
}
```

Acquire/Release ensures synchronization between threads that are storing and loading the same atomic object (also called as half-synchronization)

False Sharing (1/5)

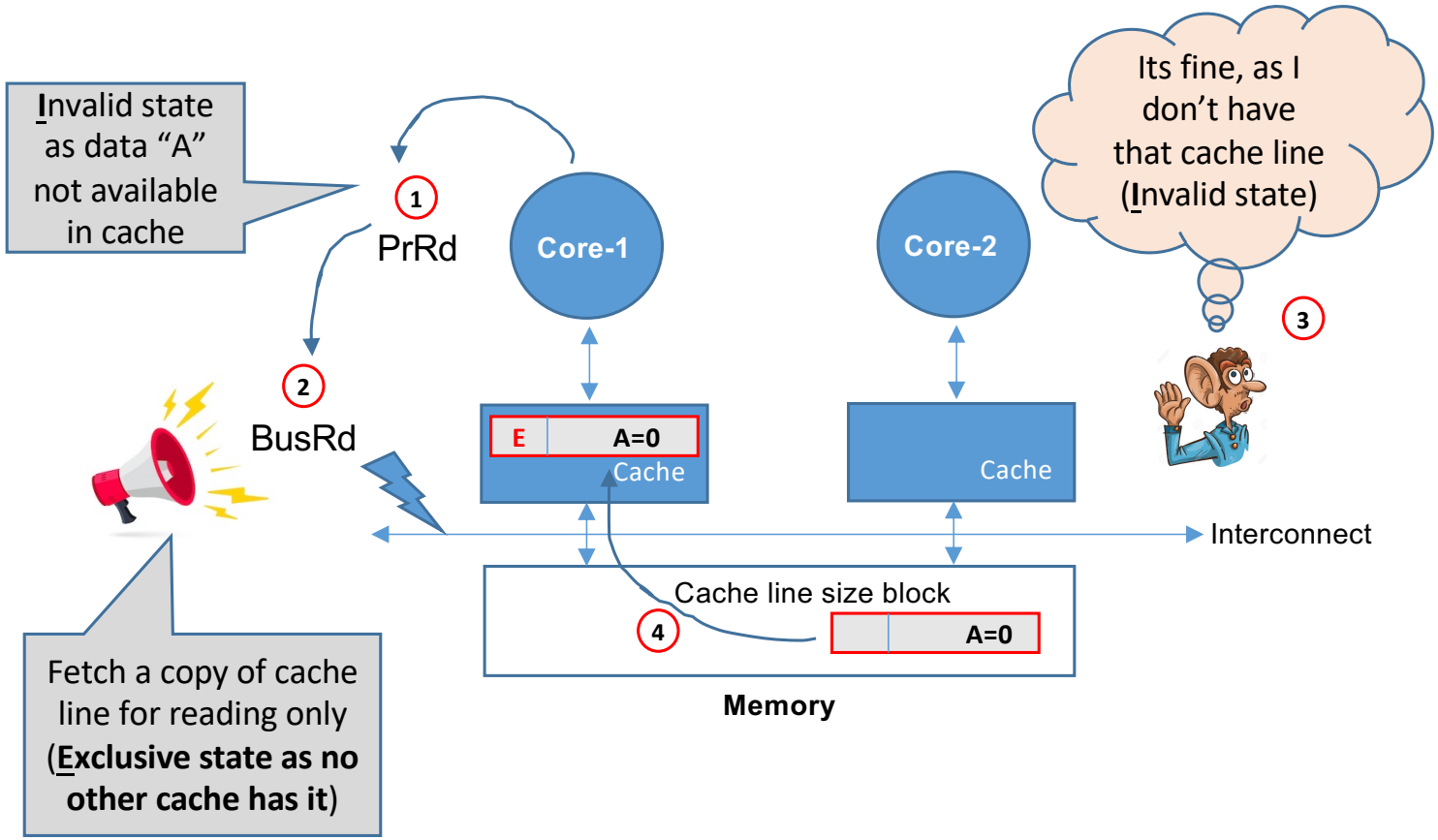
```
int A[8]; //Global array

thread_func(int id) {
    for(i = 0; i < M; i++)
        A[id]++; |++;
}
```



Let's try to understand the problem in this code using the MESI coherence protocol

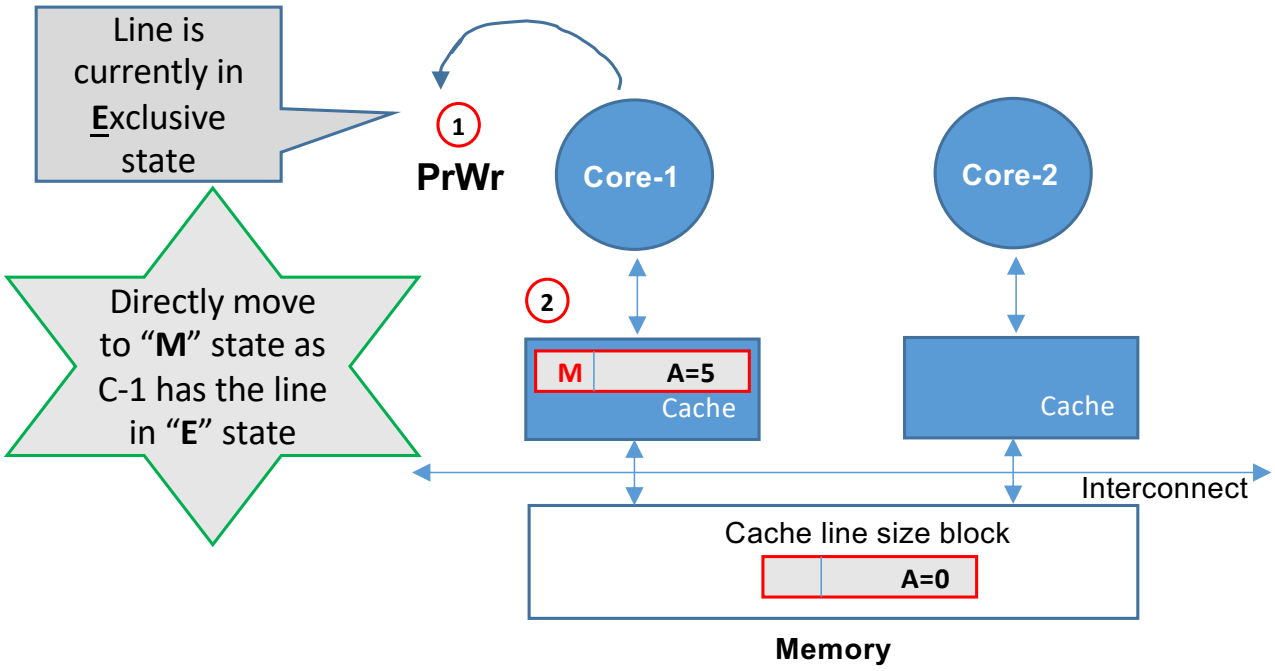
Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing (2/5)



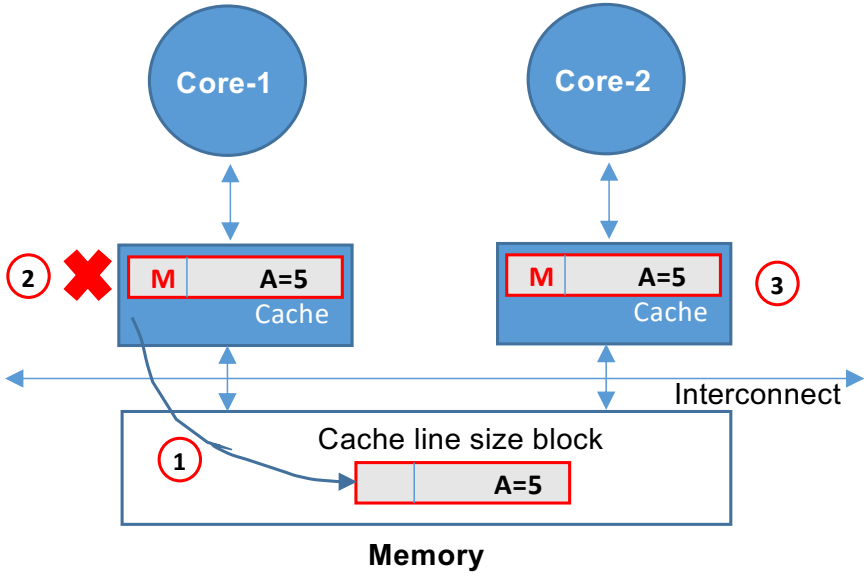
- An additional state **E**
  - Exclusive clean
  - Implies no other cache has a copy of this line

False Sharing (3/5)



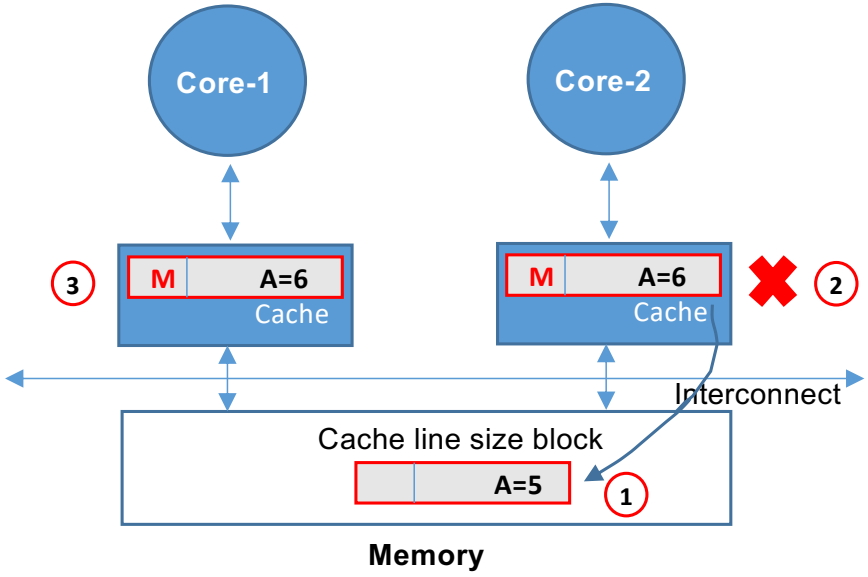
- Moving from **E** to **M** state
  - No action required to be performed on interconnect
  - Present **E** state implies the line is not in any other cache

False Sharing (4/5)



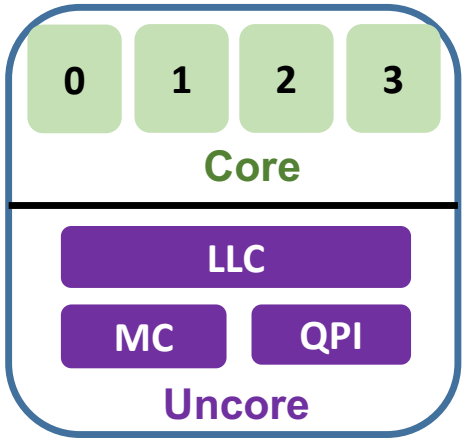
- Core-2 wants to read/write cache line A
  - Write back and invalidate cache line at Core-1

False Sharing (5/5)



- Core-1 wants to read/write cache line A
  - Write back and invalidate cache line at Core-2

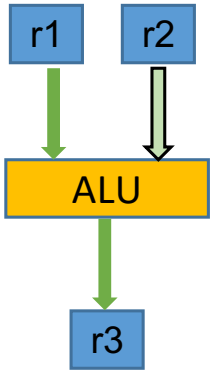
Power Management



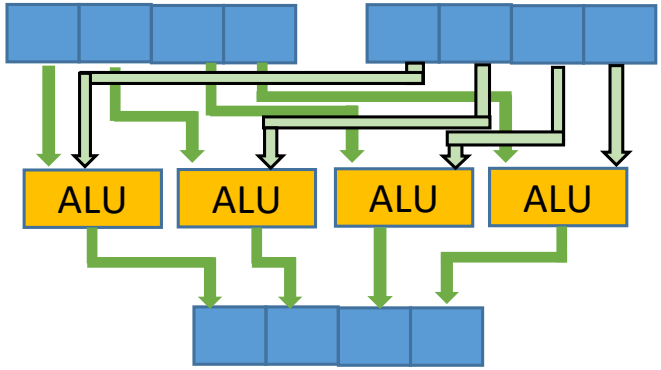
Multicore Processor

- P-states
  - Dynamic Voltage and Frequency Scaling (DVFS) is used by the processor to operate the core at a specific frequency and voltage
  - Each P-states have an associated frequency
- C-states
  - Power states used by the CPUs to reduce the power at **C**ore level or on a CPU **P**ackage **C**ore level (core, private caches, etc.)
  - Core goes to sleep
    - Used when some of the cores are not being used at all
- Uncore frequency scaling
  - Changes the frequency of the uncore elements in the processor
  - Can be set in the userspace similar to DVFS
  - Currently supported only by Intel processor

SIMD Vector Units (1/3)

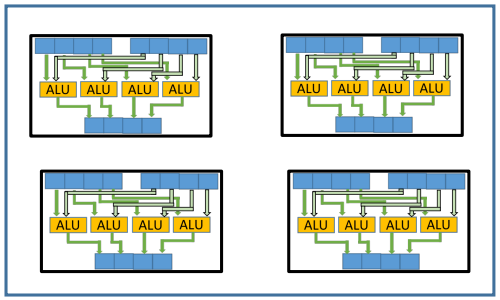


SISD operation on scalars



vr3

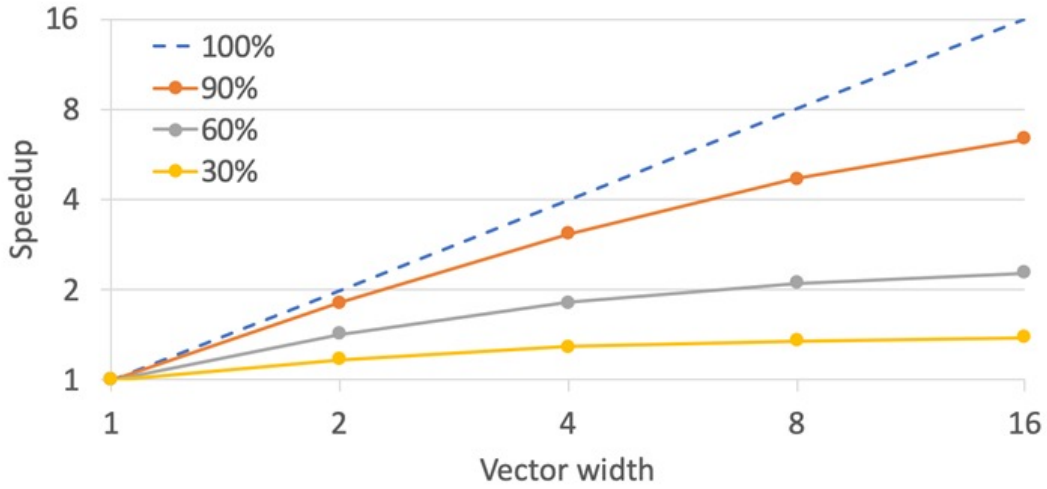
SIMD operation on vectors



Multicore processor supporting SIMD operations

- Special registers that support instructions to operate upon **vectors** than **scalar** values
- SIMD operation is supported on processors by adding more ALUs to each core, and by using wide registers (greater than 32 bit)
- Each core can operate on more than one 32-bit value in each cycle
  - Each core has its own SIMD unit
- Increasing vector register width require adding new instructions

SIMD Vector Units (2/3)



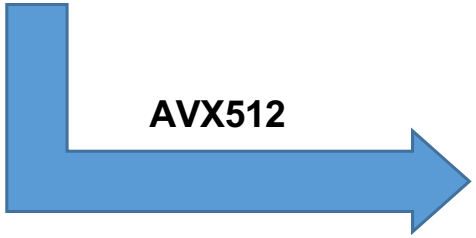
- Assume some work takes "W" time on a scalar CPU
- Time taken on a CPU with vector width "N" for total vectorized fraction "f" available in that work
  - $Time_{scalar} + Time_{vector}$ => (1-f)W + fW/N
- Hence, maximum possible speedup
  - W / {(1-f)W + fW/N} => 1 / { (1-f) + f/N }

Picture source: https://cvw.cac.cornell.edu/vector/performance_amdahl

- Linear speedup is possible only for perfectly parallel code
- The exact upper bound depends significantly on the percentage of code that is vectorized
  - At a vector width of 16, code that is 60% vectorized performs only twice as fast as non-vectorized code
- Sequential or scalar code would limit the performance
  - What about memory access pattern?

SIMD Vector Units (3/3)

```
double A[1024];

void sum() {
  for (int i=0; i<1024; i++) {
    A[i] = 1.0;
  }
}
```

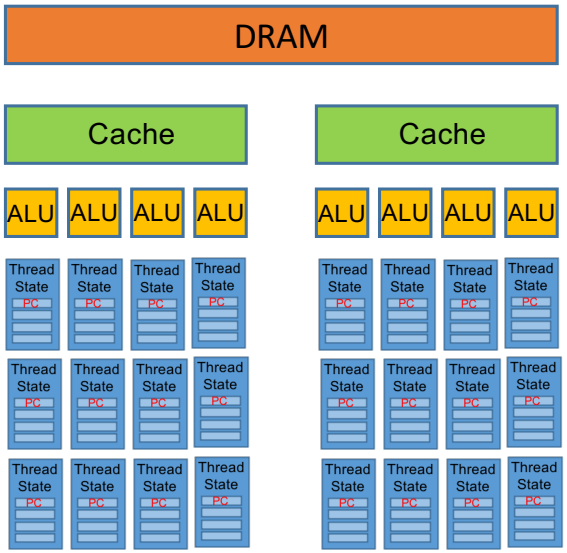



```
#include “vectorclass.h”
int A[1024];
void sum() {
  Vec8d Av(1.0);
  for (int i=0; i<1024; i+=8) {
    Av.store(A+i);
  }
}
```

- VCL program compilation

  g++ -std=c++17 -O3 -msse4 -fopt-info-vec -I/path_to/VCL/version2 sum.cpp

Design Goals for a GPU (1/2)



- Increase the number of hardware threads supported on each core
  - CPU stalls are significantly reduced
  - Improves the performance as the hardware schedule the threads instead of the OS
- Improve the memory bandwidth
  - As large chunks of memory addresses are being fetched from DRAM due to large number of threads
- But, won't these enhancements increase the complexity and cost of the multicore processor?

Design Goals for a GPU (2/2)

- If we only have to run SIMD applications on a processor, then how to cut down the complexity of the processor?
  - Reduce core frequency and increase the number of cores
  - Support large number of hardware threads at each core
    - Requires a large amount of data, but stalls are hidden due to large number of threads
  - Cores have smaller cache
    - Large number of threads per core would operate on large amount of data, thereby requiring frequent DRAM accesses
  - Increase the number of ALUs per core and the width of SIMD registers
  - Group of threads could share a single PC register
    - Single Instruction Multiple Thread (**SIMT**)
    - Shared instruction cache
  - Support high bandwidth data transfer

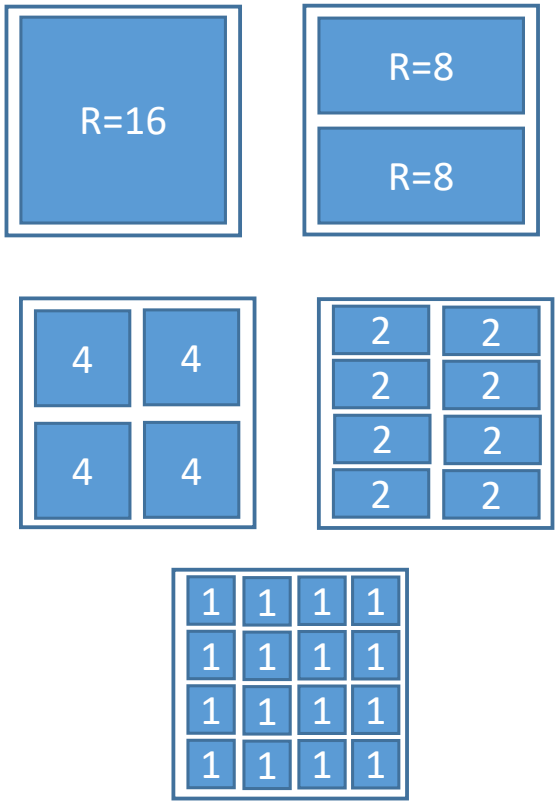
This is the design of a throughput oriented processor or a GPU

Heterogeneous Computing (1/3)

- Processor chip hardware roughly partitioned into
  - Multiple Cores (with L1 caches)
  - The Rest (L2/L3 cache banks, interconnect, etc.)
  - Changing Core Size/Number does **NOT** change **The Rest**
- Resources for multiple cores are bounded
  - Bound of **N** resources per chip for cores
  - Due to area, power, cost ($$$), or multiple factors
- Single core performance can be improved by adding more of the bounded resources (**N**)
  - E.g., **N** could be total number of transistors on a processor
  - Also known as Base Core Elements (BCE)

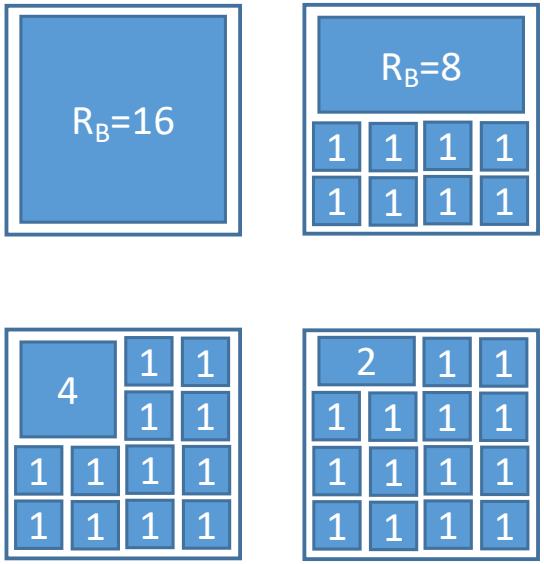
Hill and Marty 2008

Heterogeneous Computing (2/3)



- Amdahl's law for **symmetric** multicores
  - Serial fraction of the program would run on a single core with performance as:
    - (1-f) / $Perf_R$
      - $Perf_R$ is the performance of each of the single core of the processor type R shown below
  - Parallel fraction use N/R cores at the rate Perf(R) each
    - f / (Perf(R) * (N/R)) = f*R / Perf(R)*N
  - Speedup(f, R, N) = 1 / ({(1-f)/Perf(R)} + {f*R/(Perf(R)*N)})

Heterogeneous Computing (3/3)



- Amdahl's law for **asymmetric** multicores
  - Each processor is using the same number of total resources (**N=16**)
    - One **B**ig core with R_B resources would leave N-R_B resources for little cores
    - Assuming each little core has R=1, total number of little cores are N-R_B
  - Serial fraction would still be represented as in symmetric
    - (1-f) / Perf(R_B)
  - Parallel fraction using one big core with Perf(R_B) performance and N-R_B little cores with Perf(1)=1 performance would now be
    - f / (Perf(R_B) + N – R_B)
  - Speedup(f, R_B, N) = 1 / ({(1-f) / Perf(R_B)} + {f/(Perf(R_B) + N –R_B)})

Next Two Lectures

- Lectures 24-26 combined into two lectures of 2 hours each
- These remaining lectures will be student seminar
- Each project group will give a presentation as notified earlier
  - 15mins ppt per group