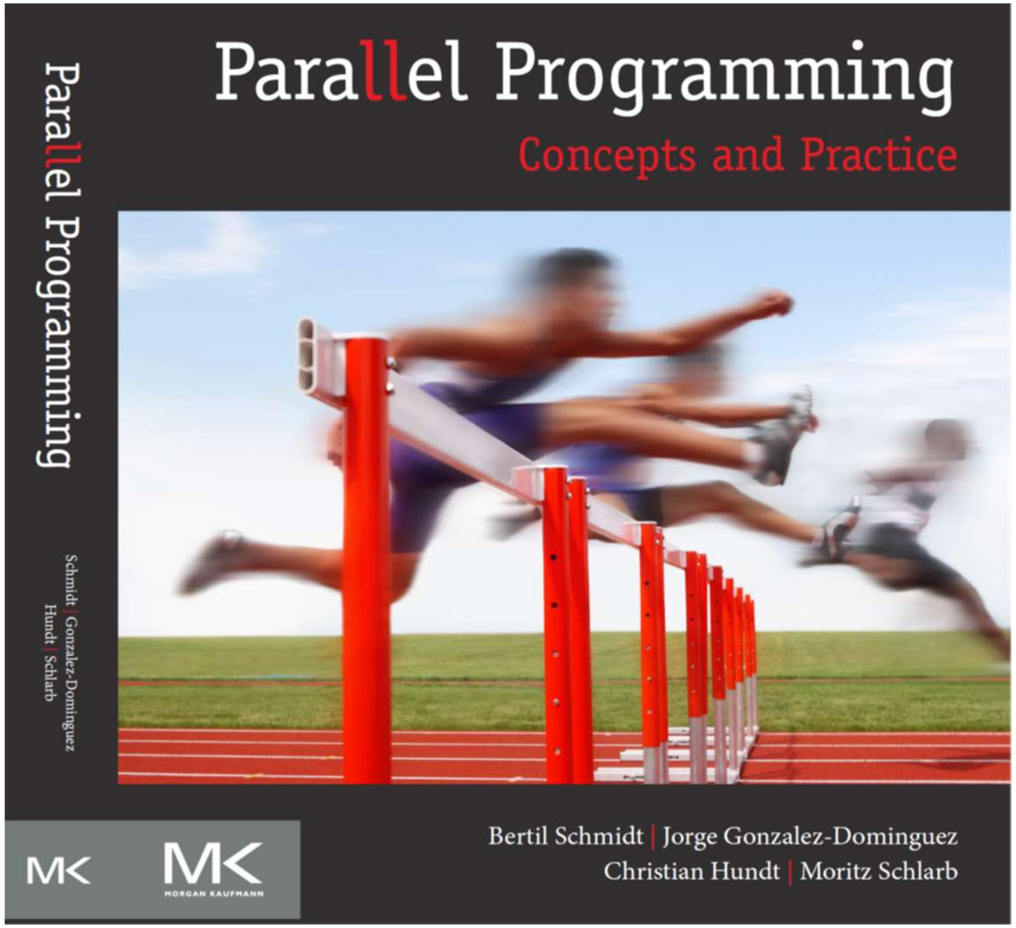
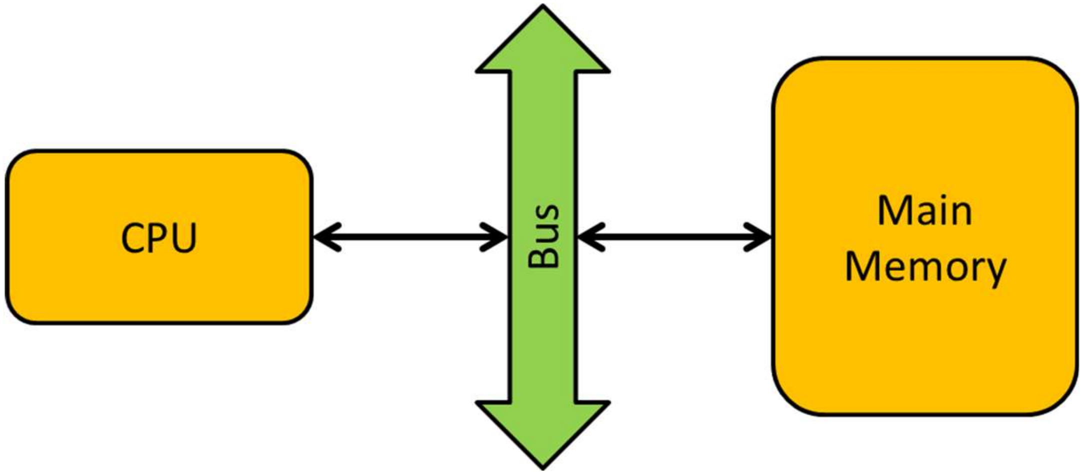
Chapter 03: Modern Architectures



Learning Outcomes

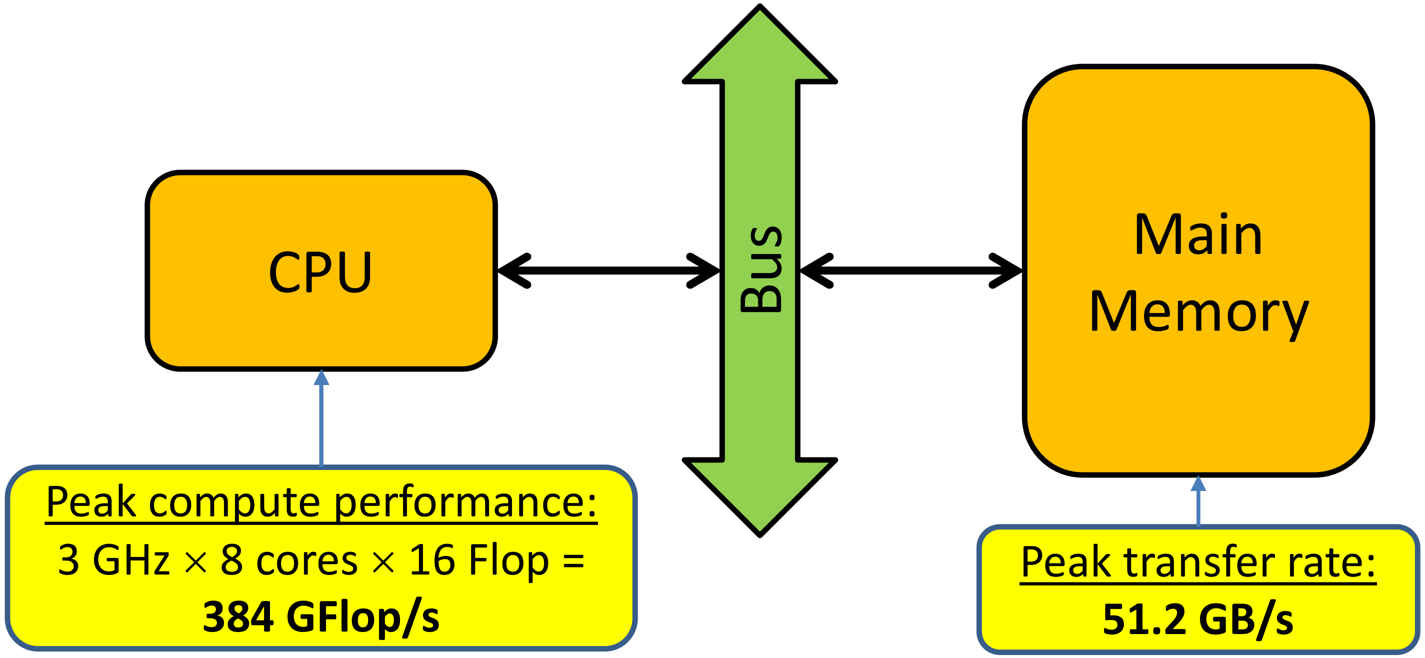
- Nowadays, single-threaded CPU performance is stagnating
- Taking full advantage of modern architectures requires not only sophisticated (parallel) algorithm design but also knowledge of features such as the **memory system**
- Learn about the memory hierarchy with fast caches located in-between CPU and main memory to mitigate the ***von Neumann bottleneck***
- Write programs making effective use of the available memory system
- Understand Cache Coherency and False Sharing in multi-core CPU systems
- Study the basics of SIMD parallelism and Flynn's taxonomy

Basic Structure of a Classical von Neumann Architecture



- In early computer systems timings for accessing main memory and for computation were reasonably well balanced
- During the past few decades computation speed grew at a much faster rate compared to main memory access speed resulting in a significant performance gap.
- **von Neumann Bottleneck:** Discrepancy between CPU compute speed and main memory (DRAM) speed

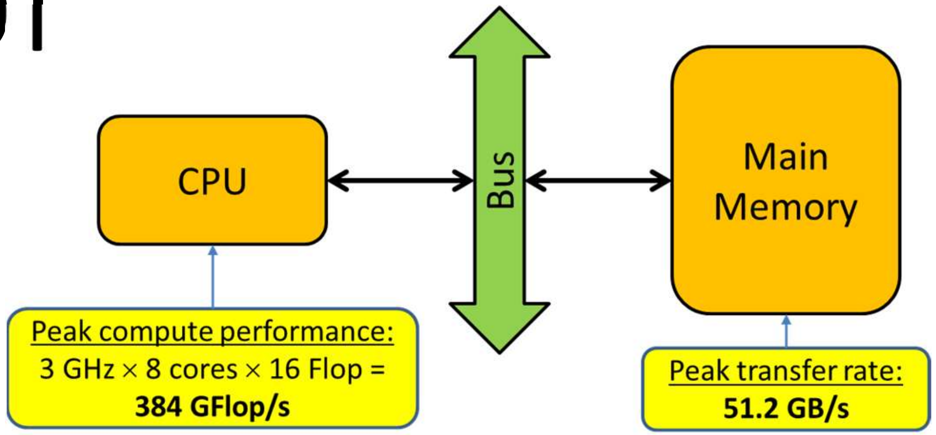
Von Neumann Bottleneck – Example



- Simplified model in order to establish an **upper bound on performance** for computing a **dot product** of two vectors *u* and *v* containing *n* double precision numbers stored in main memory
  - i.e. we will never go faster than what the model predicts

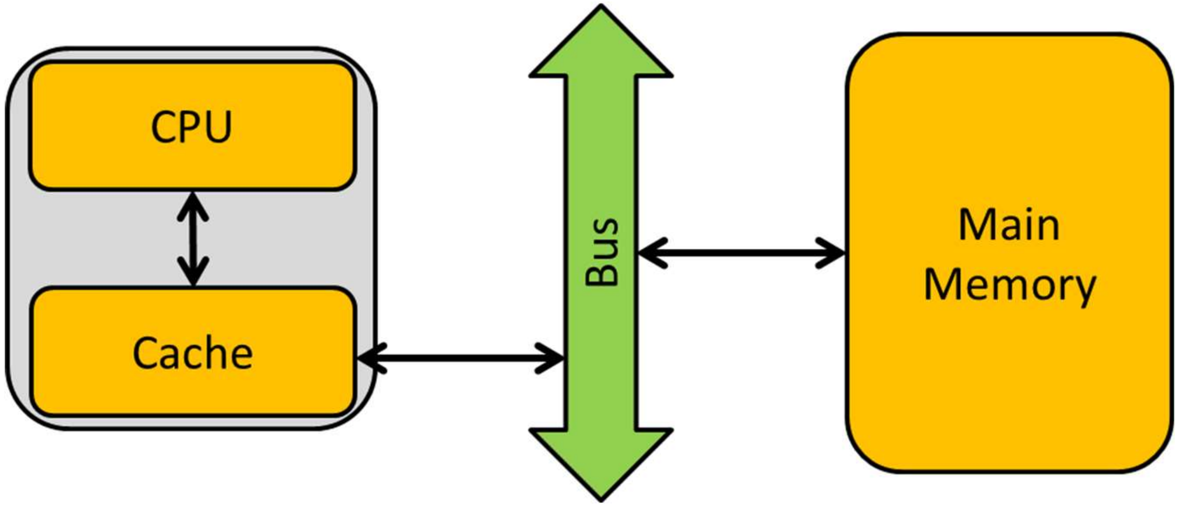
Performance of DOT

```c
// Dot Product
double dotp = 0.0;
for (int i = 0; i<n; i++)
    dotp += u[i] * v[i];
```



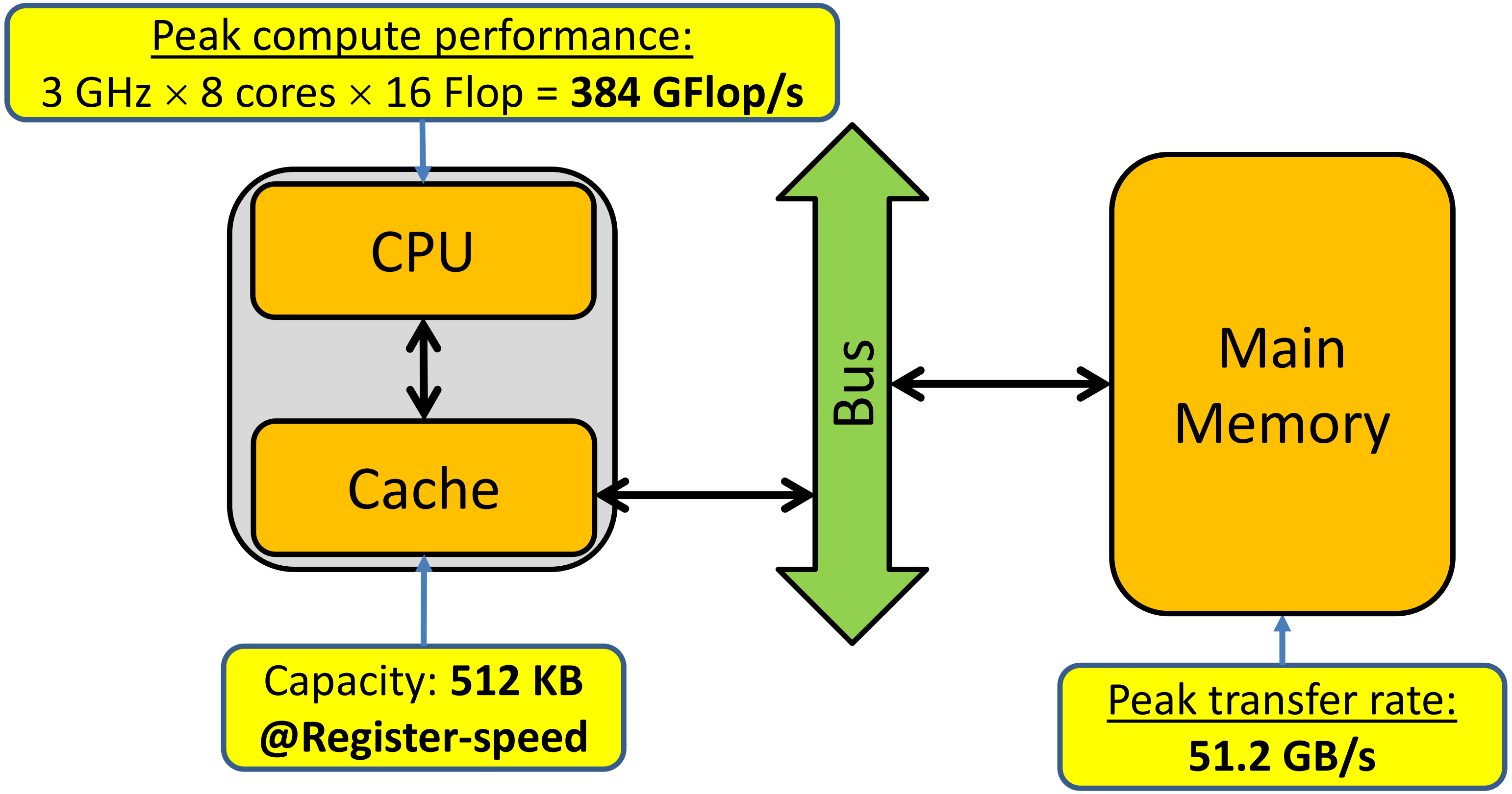
- Example: $n = 2^{30}$
- Computation time: $t_{\text{comp}} = \frac{2\ GFlop}{384\ GFlop/s} = 5.2\ ms$
  - Total operations: $2 \cdot n = 2^{31}$ Flops = 2 GFlops
- Data transfer time: $t_{\text{mem}} = \frac{16\ GB}{51.2\ GB/s} = 312.5\ ms$
  - Amount of data to be transferred: $2 \cdot 2^{30} \cdot 8$ B = 16 GB
- Execution time: $t_{\text{exec}} \geq max(5.2ms, 312.5ms) = 312.5ms$
  - Achievable performance: $\frac{2\ GFlop}{312.5\ ms} = 6.4\ GFlop/s$ (<2% of peak)
- $\Rightarrow$ Dot product is **<u>memory bound</u>** (no reuse of data)

Basic Structure of a CPU with a single Cache



- CPUs typically contain a hierarchy of three levels of cache (L1, L2, L3)
  - Current CUDA-enabled GPUs contain two levels
- Higher bandwidth and lower latency compared to main memory but much smaller capacity
- Trade-off between capacity and speed
  - e.g. L1-cache is small but fast and the L3-cache is relatively big but slow.
- Caches could be private for a single core or shared between several cores

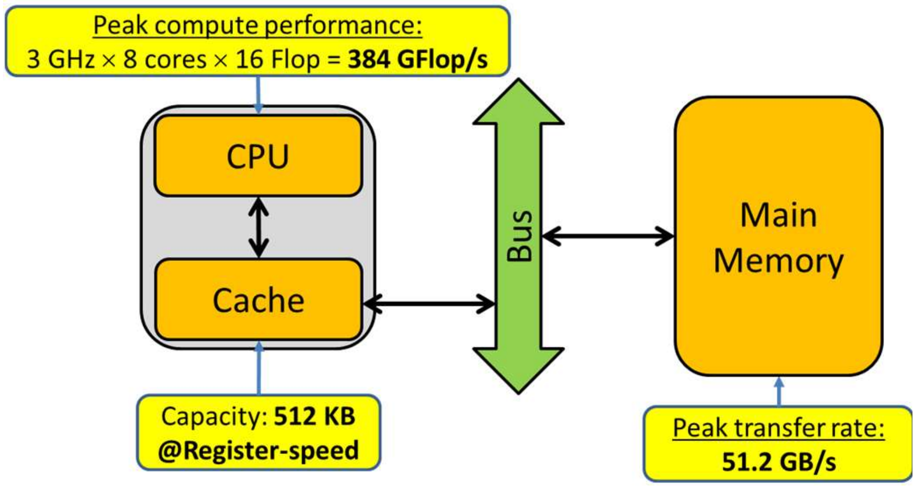
Cache Memory – Example



- Simplified model in order to establish an **upper bound on performance** for computing a matrix $W = U \times V$ each of size $n \times n$ stored in main memory
  - i.e. we will never go faster than what the model predicts

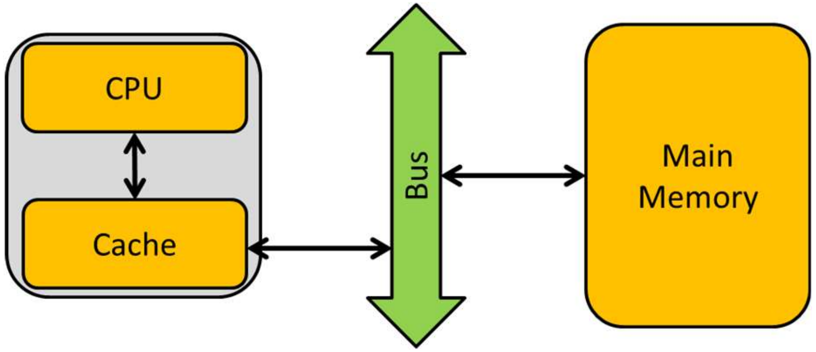
Performance of MM

```
//Matrix Multiplication
for (int i = 0; i<n; i++)
    for (int j = 0; j < n; j++) {
        double dotp = 0;
        for (int k = 0; k<n; k++)
            dotp += U[i][k]*V[k][j];
        W[i][j] = dotp;
    }
```



- Example: $n = 128$
- Data transfer time: $\boldsymbol{t_{\text{mem}} = \frac{384\ KB}{51.2\ GB/s} = 7.5\ \mu s}$
  - Data transfer (from/to Cache): $n = 128$: $128^2 \times 3 \times 8\text{B} = 384$ KB (fits in Cache)
- Computation time: $\boldsymbol{t_{\text{comp}} = \frac{2^{22}\ Flop}{384\ GFlop/s} = 10.4\ \mu s}$
  - Total operations: $2 \cdot n^3 = 2 \cdot 128^3 = 2^{22}$ Flops
- Execution time: $\boldsymbol{t_{\text{exec}} \geq 7.5\ \mu s + 10.4\ \mu s = 17.9\ \mu s}$
  - Achievable performance: $\boldsymbol{\frac{2^{22}\ Flop}{17.9\ \mu s} = 223 GFlop/s}$ (≈60% of peak)
- ⇒ Lot of data reuse in MM! What if matrices are bigger than cache?

Cache Algorithms



Which data do we load from main memory?

Where in the cache do we store it?

If cache is already full, which data do we evict?

- Cache does not need to be explicitly managed by the user
- Managed by a set of caching policies (*cache algorithms*) that determine which data is cached during program execution
- **Cache hit:** Data request can be serviced by reading from the cache without the need for a main memory transfer
- **Cach miss:** Otherwise
- **Hit ratio:** Percentage of data requests resulting in a cache hit

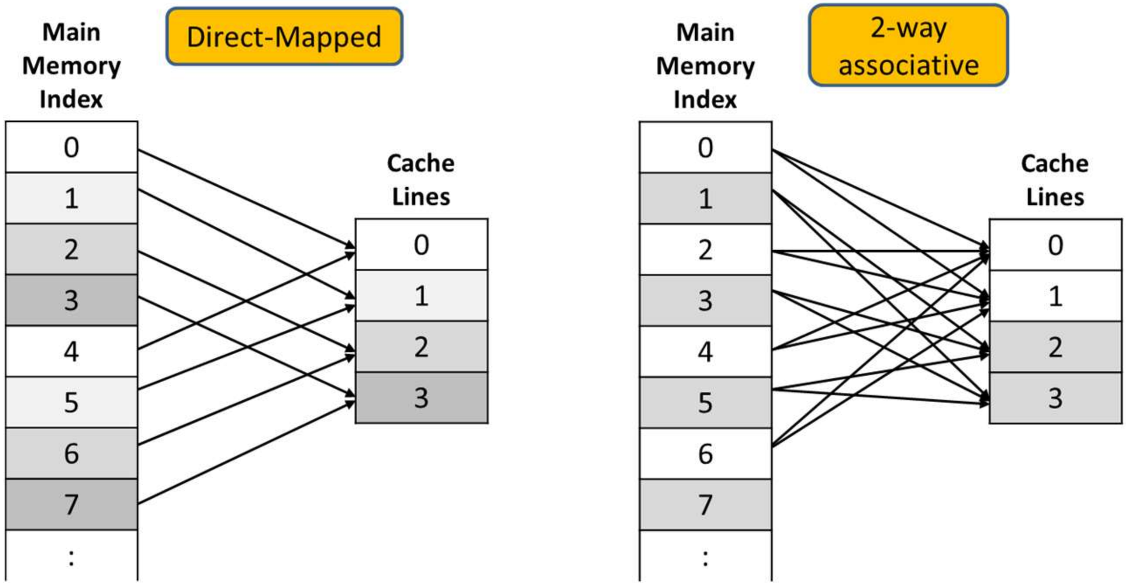
Caching Algorithms – Spatial Locality

Which data do we load from main memory?

```
//maximum of an array (elements stored contiguously)
for (i = 0; i<n; i++)
    maximum = max(a[i], maximum);
```

- **Cache Line:** several items of information as a single memory location
- Instead of requesting only a single value, an entire cache line is loaded with values from neighboring addresses.
- **Example:** Cache line size of 64 B and double precision values
  - First iteration: `a[0]`is requested resulting in a cache miss
  - Eight consecutive values `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`, `a[5]`, `a[6]`, `a[7]` loaded into the same cache line
  - Next seven iterations will then result in cache hits
  - Subsequent request `a[8]` resulting again in a cache miss, an so on
  - Overall, the hit ratio in our example is as high as 87.5%

Caching Algorithms – Temporal Locality



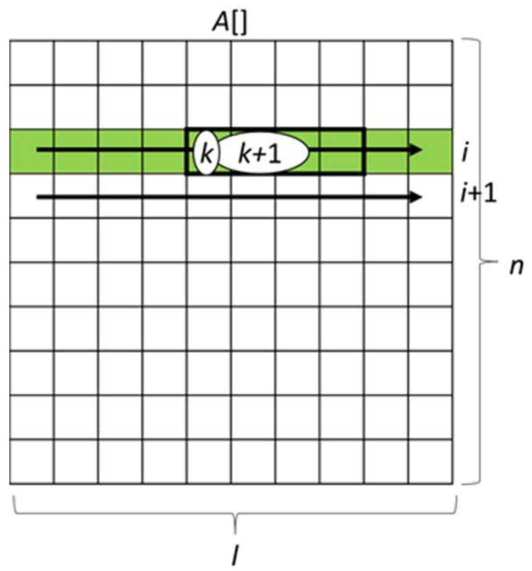
Where in the cache do we store it?

If cache is already full, which data do we evict?

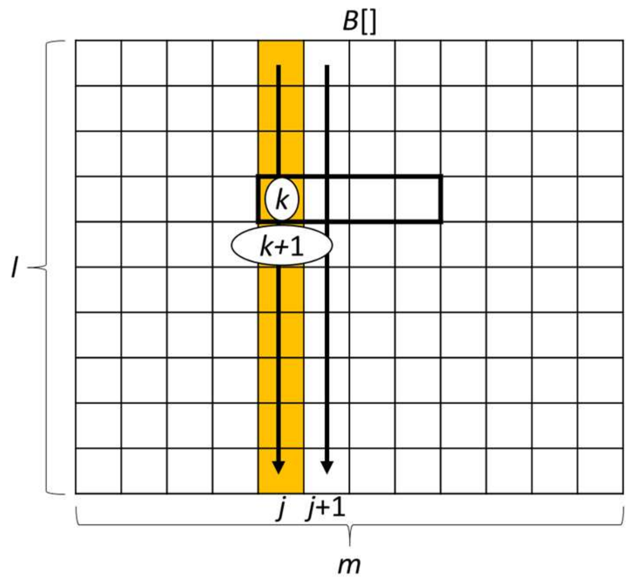
- Cache organized into a number of **Cache Lines**
- Cache mapping strategy decides in which location in the cache a copy of a particular entry of main memory will be stored
- **Direct-Mapped Cache:** Each block from main memory can be stored in exactly one cache line (high miss rates)
- ***n*-way Set Associative Cache:** Each block from main memory can be stored in one of *n* possible cache lines (higher hit rate at increased complexity)
- **Least Recently Used (LRU):** Commonly used policy to decide which of several possible locations to choose is based on temporal locality (LRU)

Optimizing Cache Accesses

```
//Naive Matrix Multiplication
for (int i = 0; i<n; i++)
    for (int j = 0; j < m; j++) {
        float accum = 0;
        for (k = 0; k < l; k++)
            accum += A[i*l+k]*B[k*n+j];
        C[i*m+j] = accum;
    }
```

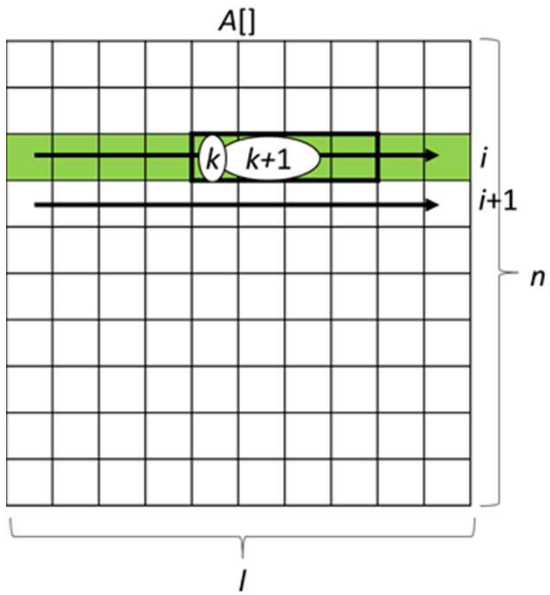


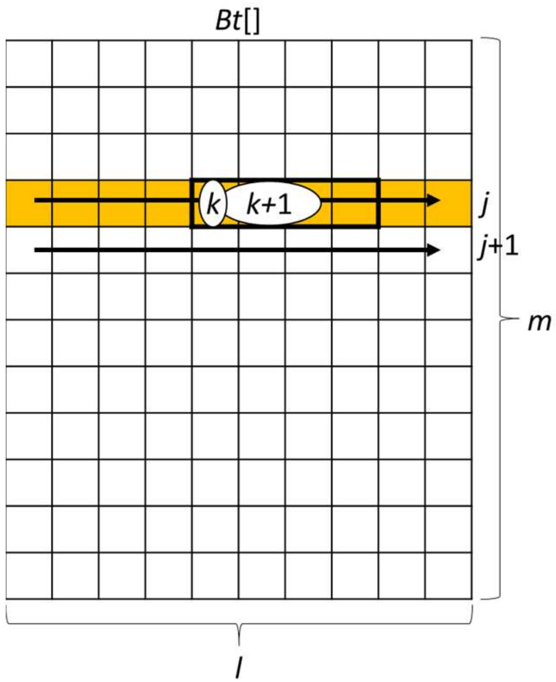
- Matrix multiplication: $A_{n\times l} \cdot B_{l\times m} = C_{n\times m}$
- Stored in linear arrays in row-major order
- Access pattern of A contiguously: $(i,k) \Rightarrow (i,k+1)$
- Accesses pattern of B non-contiguously: $(k,j) \Rightarrow (k+1,j)$
  - $l\times$sizeof(float) apart in main memory $\Rightarrow$ not stored in same cache line
  - Cache line possibly evicted from L1-cache $\Rightarrow$ low hit-rate for large l



Optimizing Cache Accesses

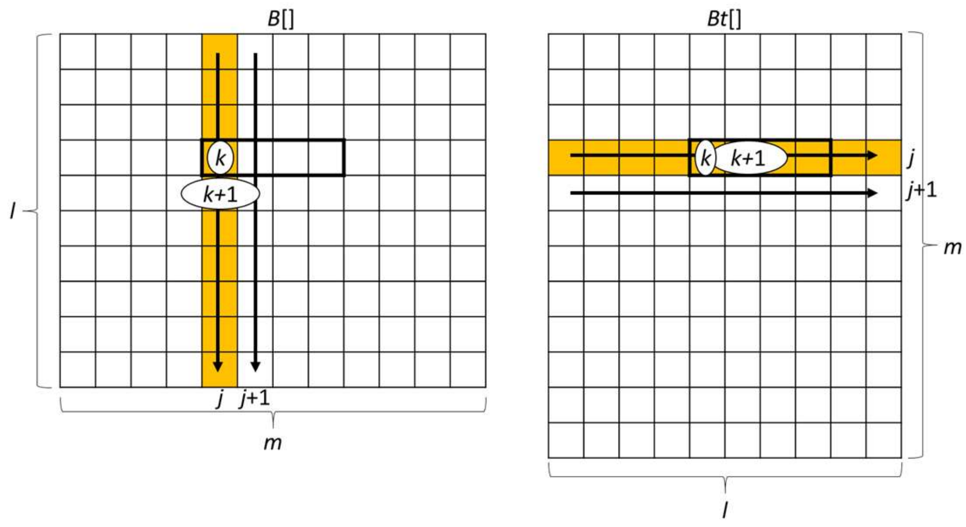
```
//Transpose-and-Multiply
for (k=0; k<l; k++)
    for (j = 0; j<m; j++)
        Bt[i*l+k] = B[k*n+j];
for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
        float accum = 0;
        for (k=0; k<l; k++)
            accum += A[i*l+k] * B[j*l+k];
        C[i*m + j] = accum;
    }
```





- Transpose-and-Multiply:
  1. $Bt_{m\times l} = (B_{l\times m})^T$
  2. $A_{n\times l} \cdot Bt_{m\times l} = C_{n\times m}$
- Access pattern of A contiguously: $(i,k) \Rightarrow (i,k+1)$
- Accesses pattern of Bt contiguously: $(j,k) \Rightarrow (j,k+1)$

Optimizing Cache Accesses



Execution on an i7-6800K using $m = n = l = 2^{13}$
#elapsed time (naive_mult): 5559.5s
#elapsed time (transpose): 0.8s
#elapsed time (transpose_mult): 497.9s

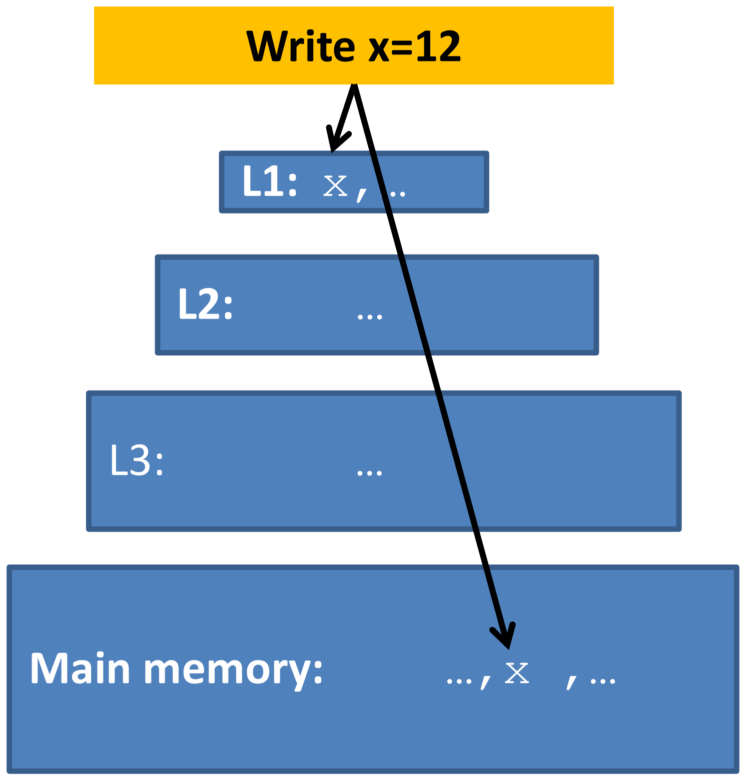
Speedup: 11.1

Execution on an i7-6800K using $m = n = 2^{13}$, $l = 2^{8}$
#elapsed time (naive_mult): 28.1s
#elapsed time (transpose): 0.01s
#elapsed time (transpose_mult): 12.9s

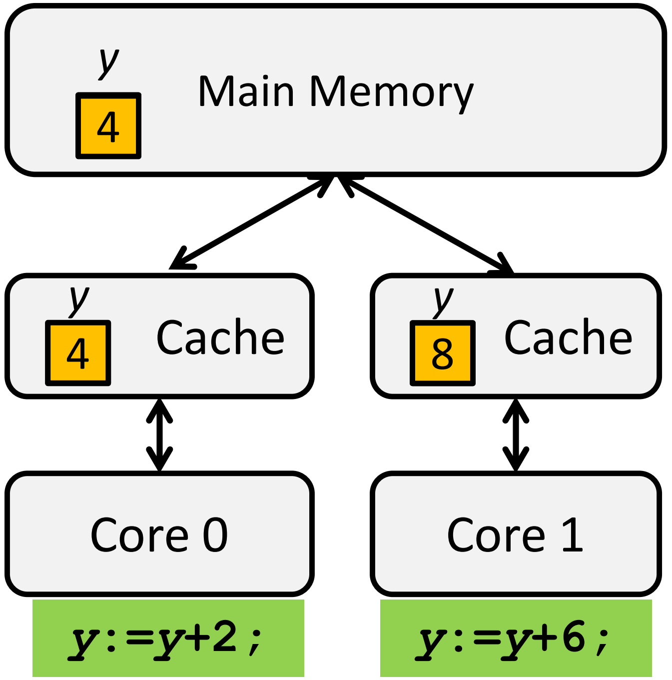
Speedup: 2.2

Cache Write Policies

- When a CPU writes data to cache, the value in cache may be **inconsistent** with the value in main memory.
- **Write-through**
  - Caches handle this by updating the data in main memory at the time it is written to cache
- **Write-back**
  - Caches mark data in the cache as dirty
  - When the cache line is replaced by a new cache line from memory, the dirty line is written to memory

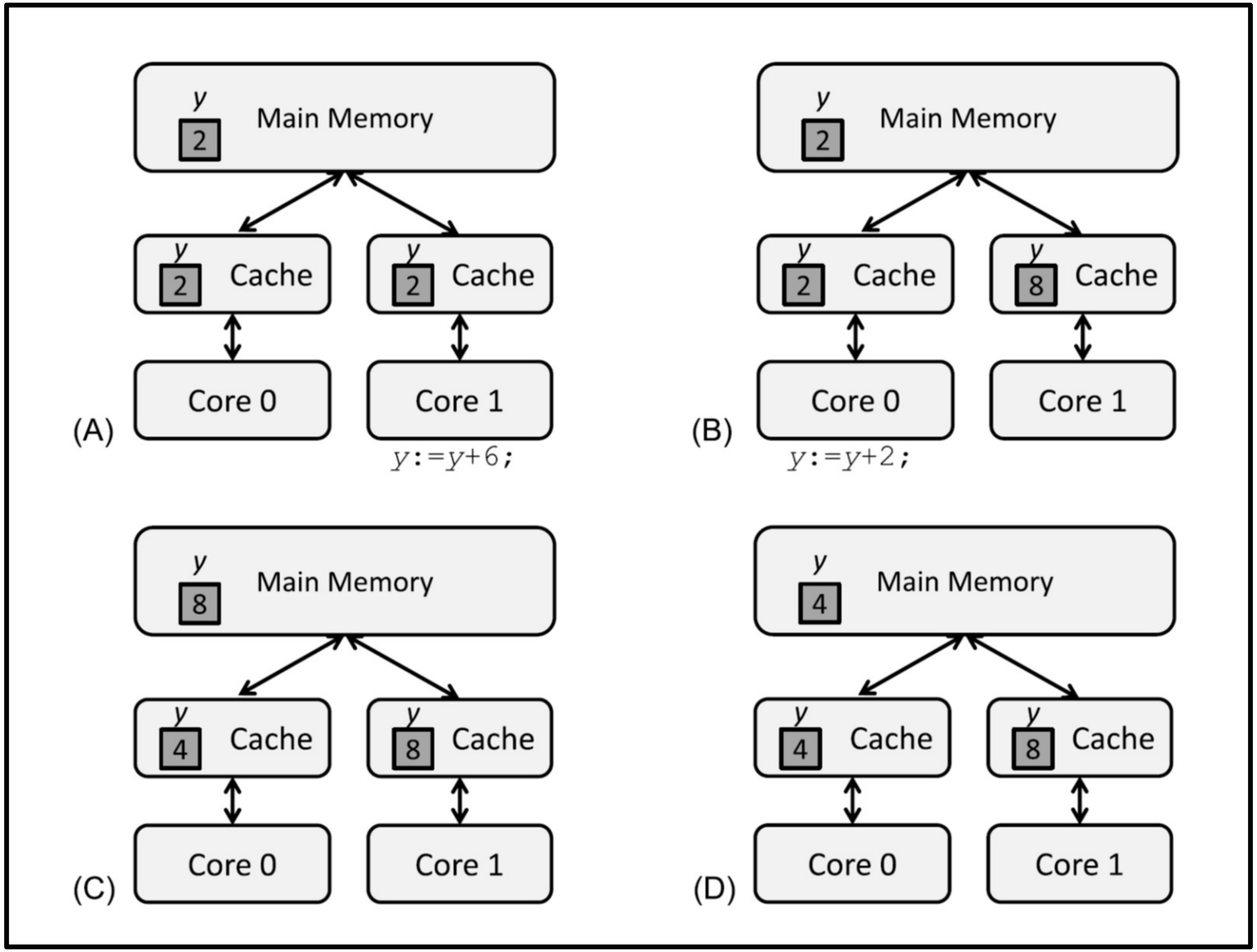


The Cache Coherence Problem – Example

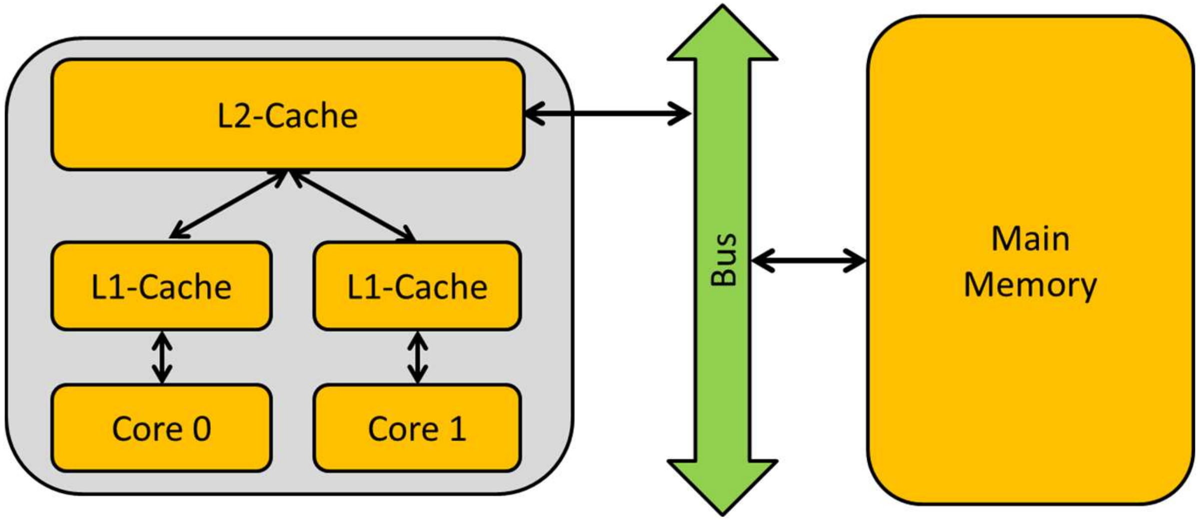


Cache inconsistency: the two caches store different values for the same variable

The Cache Coherence Problem – Example

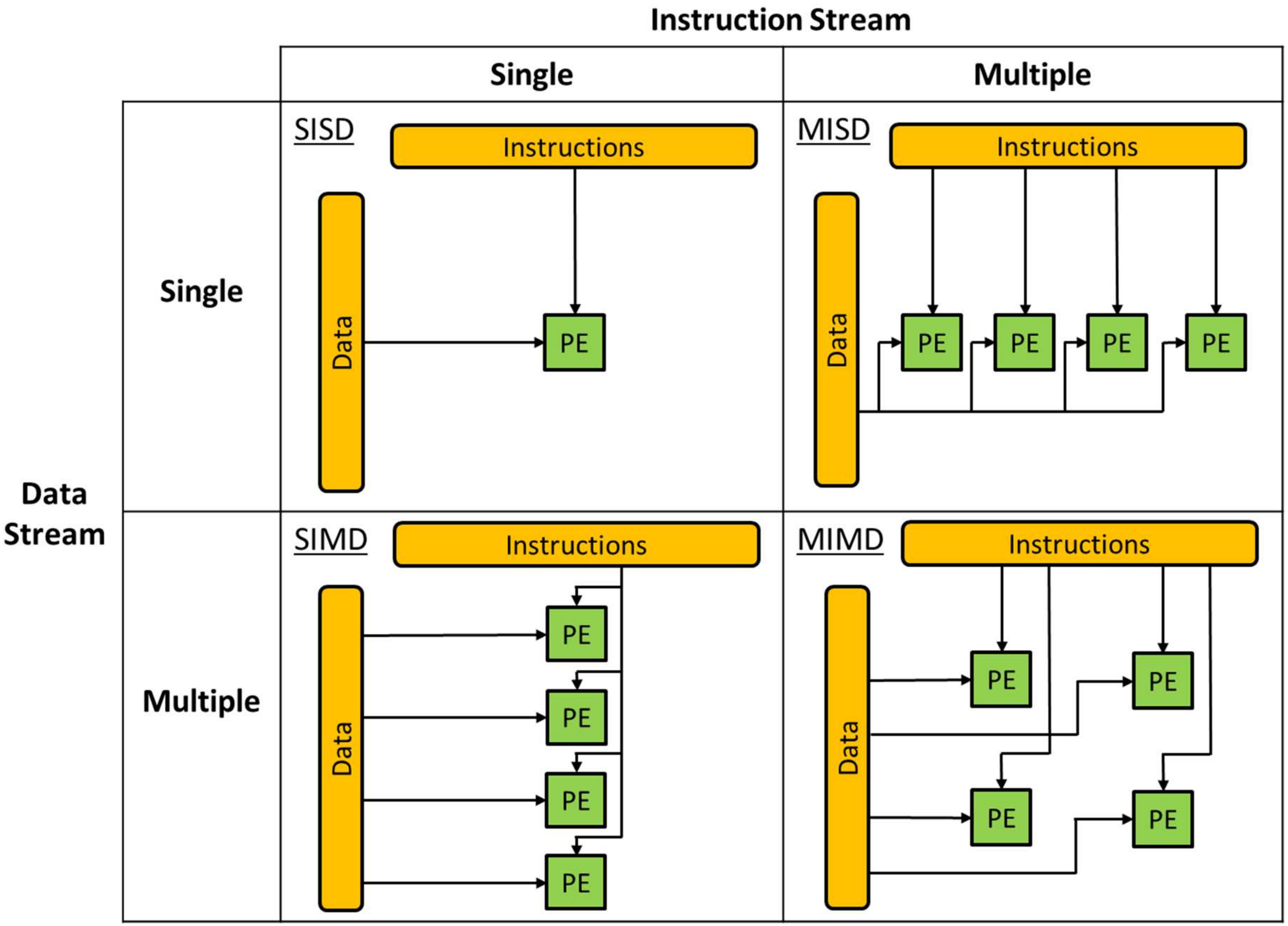


Cache Coherence



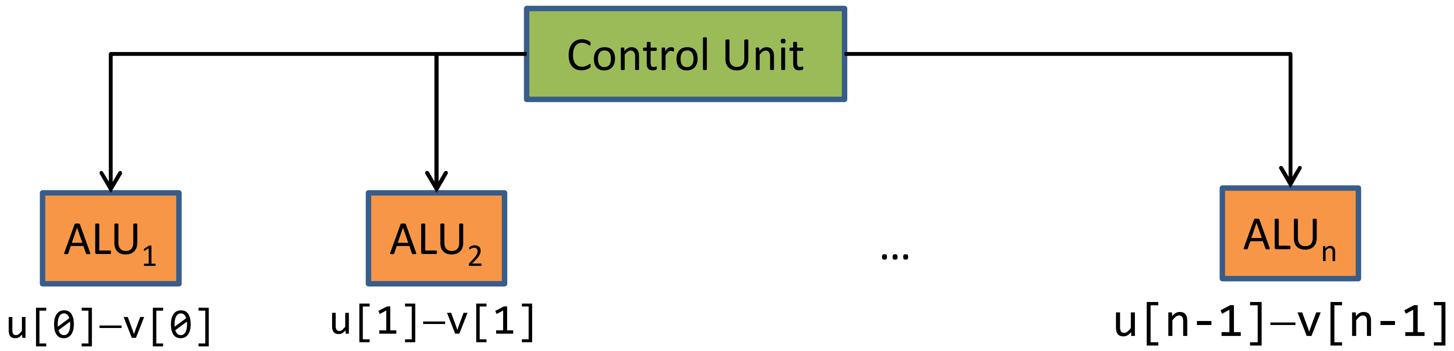
- Modern multicore CPUs often contain several cache levels
  - each core has a private (small but fast) lower-level cache
  - all cores share a common (larger but slower) higher-level cache
- Possible to have several copies of shared data
  - one copy stored in L1-cache of Core 0 & one stored in L1-cache of Core 1
- **Cache Inconsistency**
  - If Core 0 now writes to the associated cache line, only the value in the L1-cache of Core 0 is updated but not the value in the L1-cache of Core 1
- $\Rightarrow$ **Cache coherency** protocols are required

Flynn‘s Taxonomy (1966)



SIMD (Single Instruction, Multiple Data)

```
//Mapping element-wise subtraction onto SIMD
for (i = 0; i<n; i++) w[i] = u[i]-v[i];
```

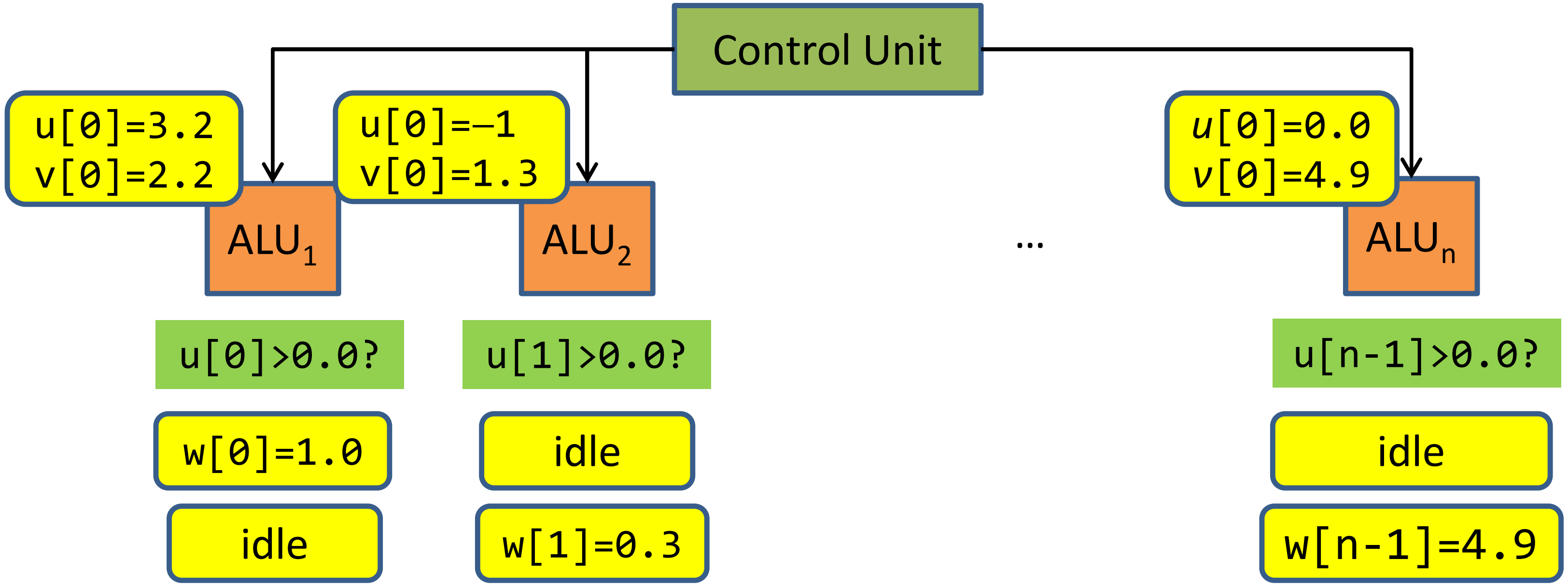


- What if we don't have as many ALUs as data items?
  - Divide the work and process iteratively

SIMD

```
//Mapping a Conditional Statement onto SIMD
for (i = 0; i<n; i++)
    if (u[i] > 0)
        w[i] = u[i]-v[i];
    else
        w[i] = u[i]+v[i];
```

All ALUs required to execute the same instruction (synchronously) or idle



- Modern CPU cores typically contains a vector unit that can operate a number of data items in parallel (which we discuss in the subsequent subsection).
- On CUDA-enabled GPUs threads within a so-called warp operate in SIMD fashion

Review Questions

- Can you explain Cache Algorithms?
- How does Cache Coherence relate to False Sharing?
- How can you optimize cache accesses in matrix multiplication?
- Can you name some concrete examples of SIMD, MIMD, and MISD machines?