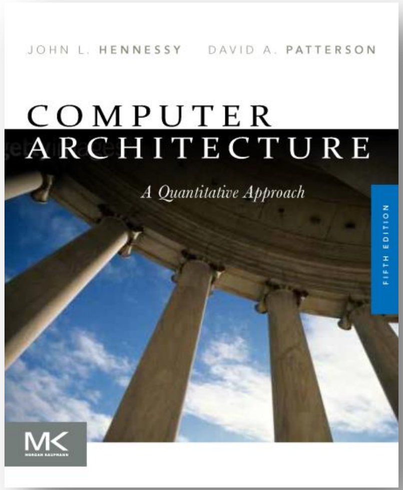
# PADP(18CS73) Unit 3

Dr. Minal Moharir



# Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

## Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors

- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices

- SIMD allows programmer to continue to think sequentially

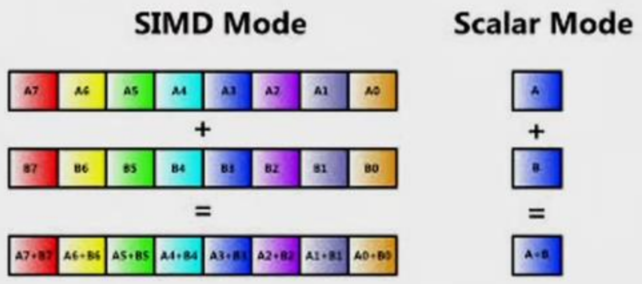
## SIMD Parallelism

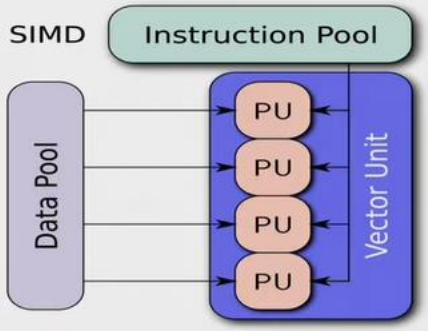
- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

## SIMD Parallelism

- Vector Processors (VPs)
- Graphics Processor Units (GPUs)





## Vector Architectures

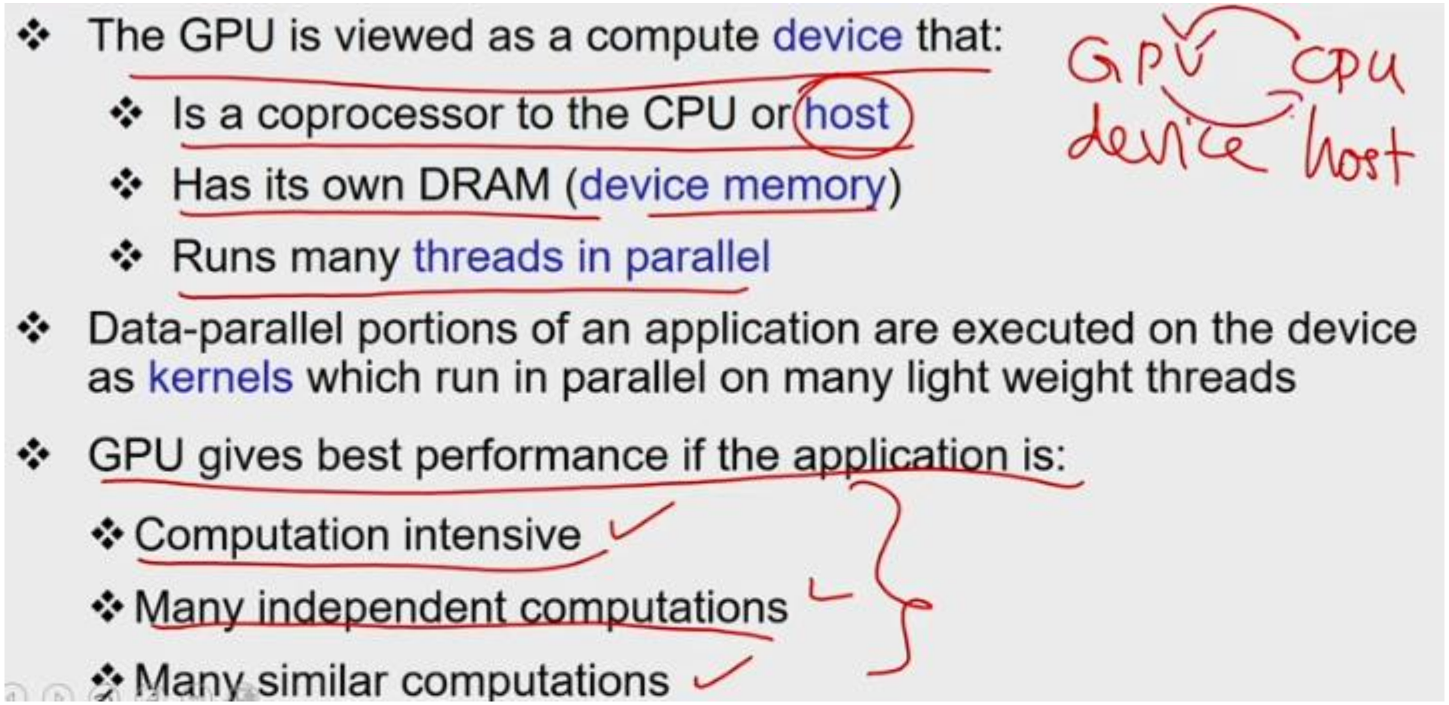
- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

# Graphical Processing Units

- Given the hardware invested to do graphics well, how can be supplement it to improve performance of a wider range of applications?

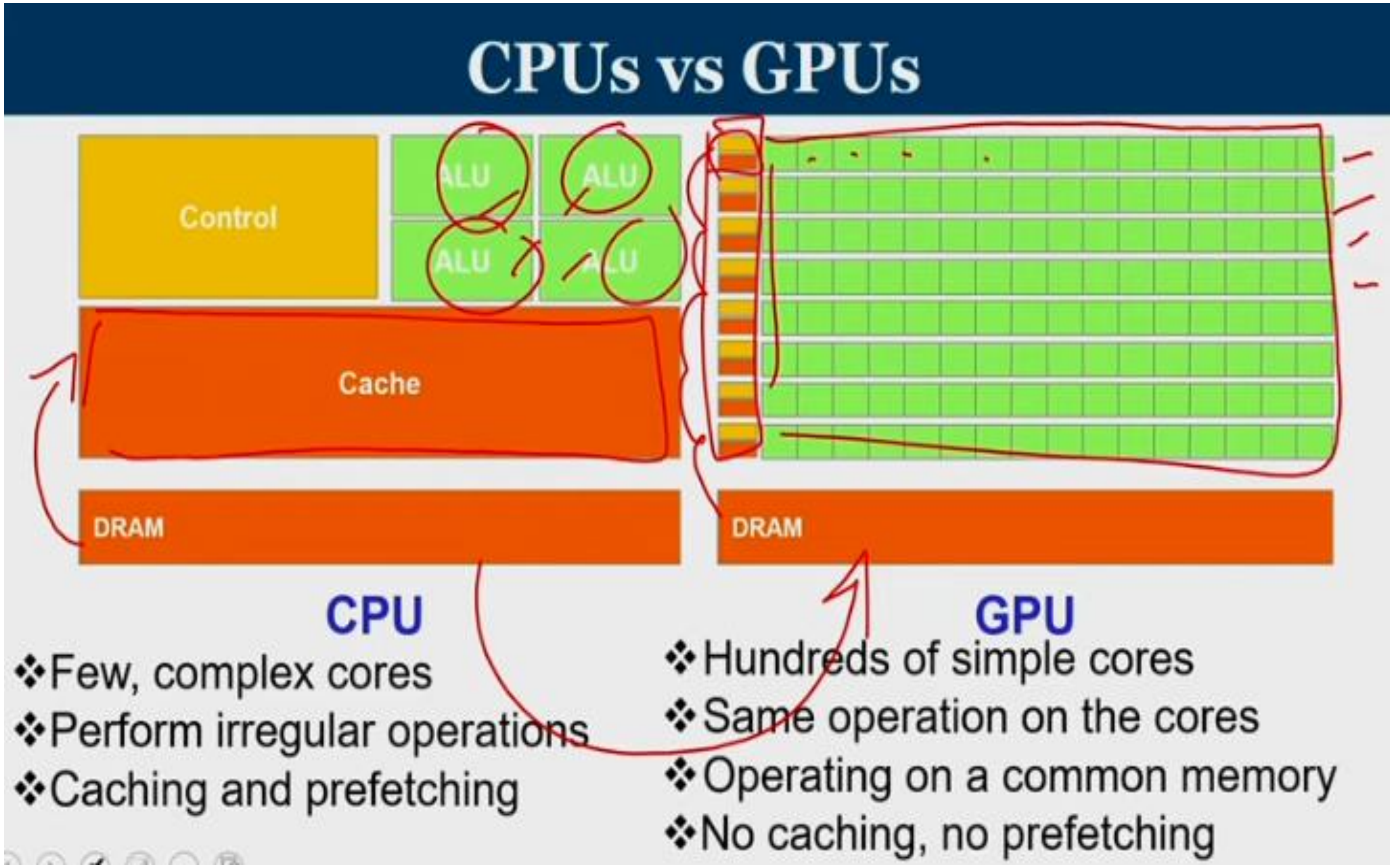
- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is “Single Instruction Multiple Thread”

# Graphical Processing Units



- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many light weight threads
- GPU gives best performance if the application is:
  - Computation intensive ✓
  - Many independent computations
  - Many similar computations ✓

## Graphical Processing Units



CPUs vs GPUs

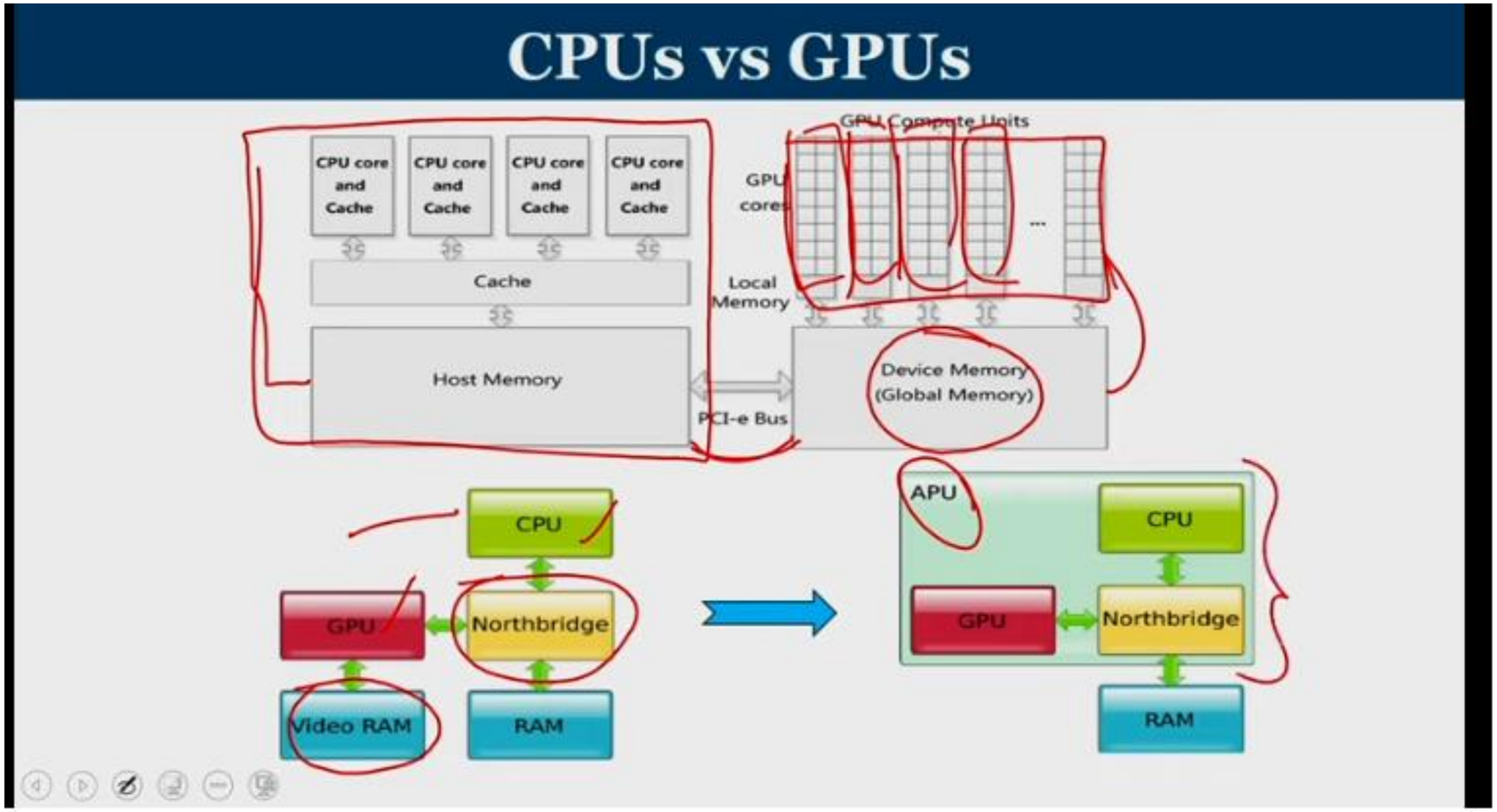
**CPU**

- Few, complex cores
- Perform irregular operations
- Caching and prefetching

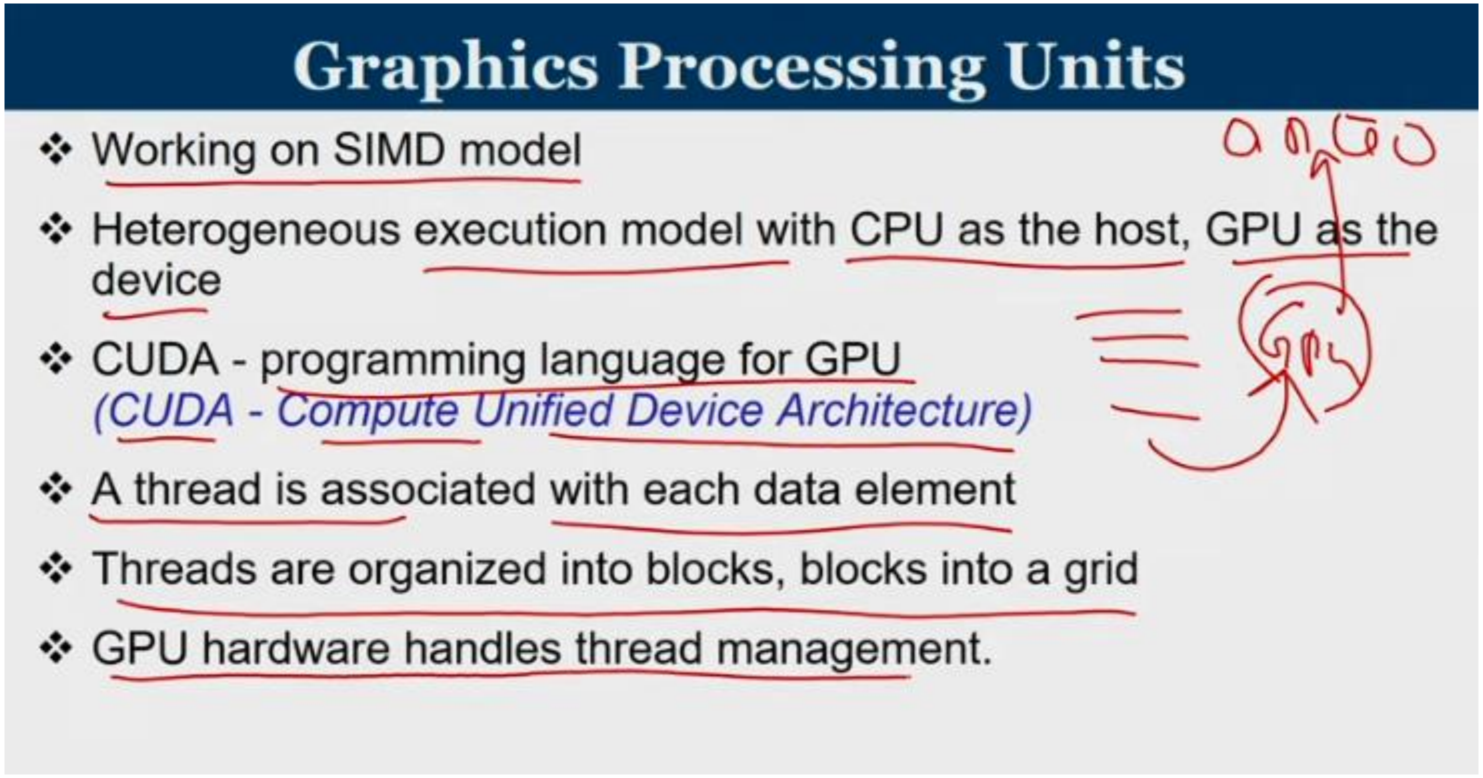
**GPU**

- Hundreds of simple cores
- Same operation on the cores
- Operating on a common memory
- No caching, no prefetching

# Graphical Processing Units



# Graphical Processing Units



## Graphics Processing Units

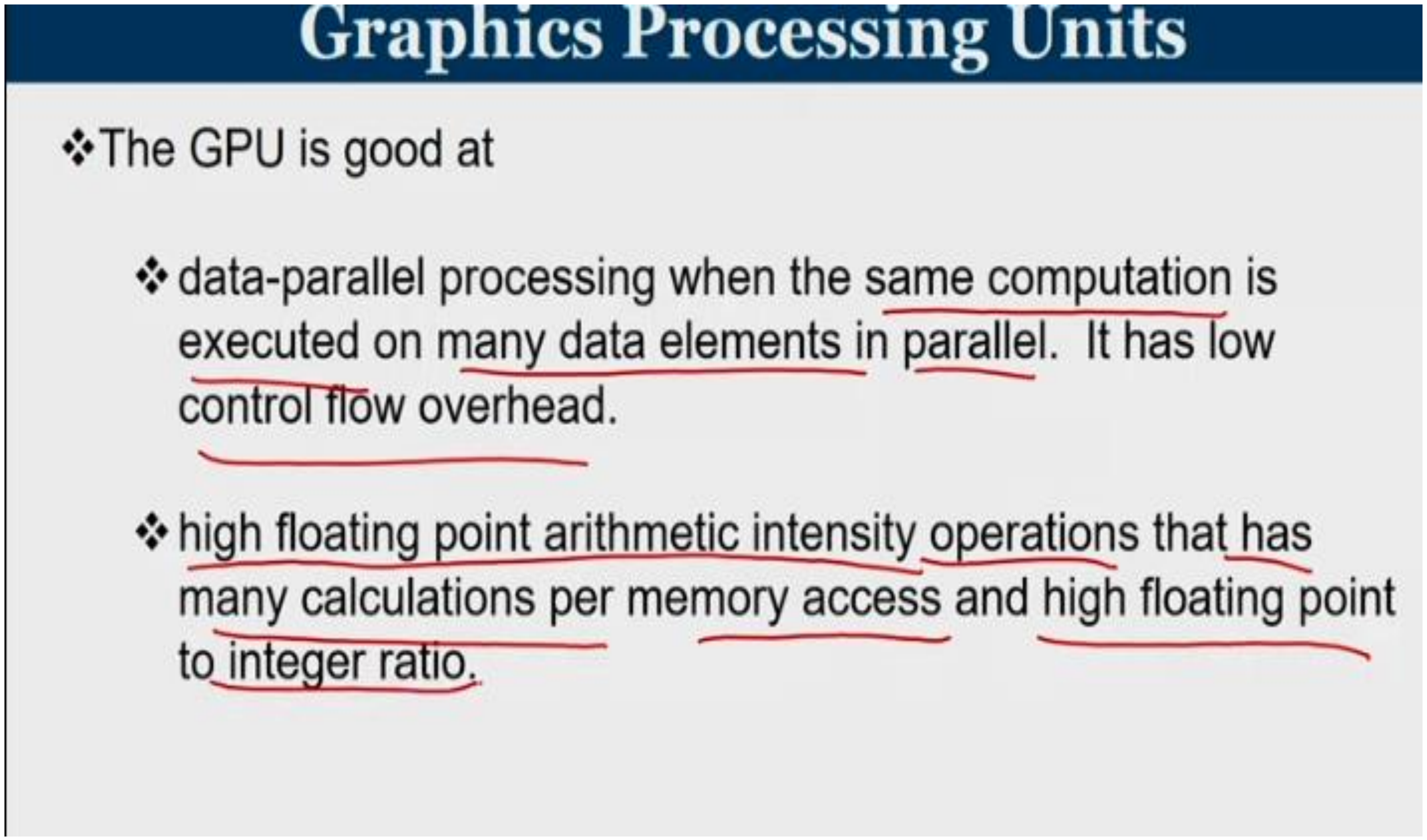
- Working on SIMD model
- Heterogeneous execution model with CPU as the host, GPU as the device
- CUDA - programming language for GPU *(CUDA - Compute Unified Device Architecture)*
- A thread is associated with each data element
- Threads are organized into blocks, blocks into a grid
- GPU hardware handles thread management.

## Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid

- GPU hardware handles thread management, not applications or OS

# Threads and Blocks



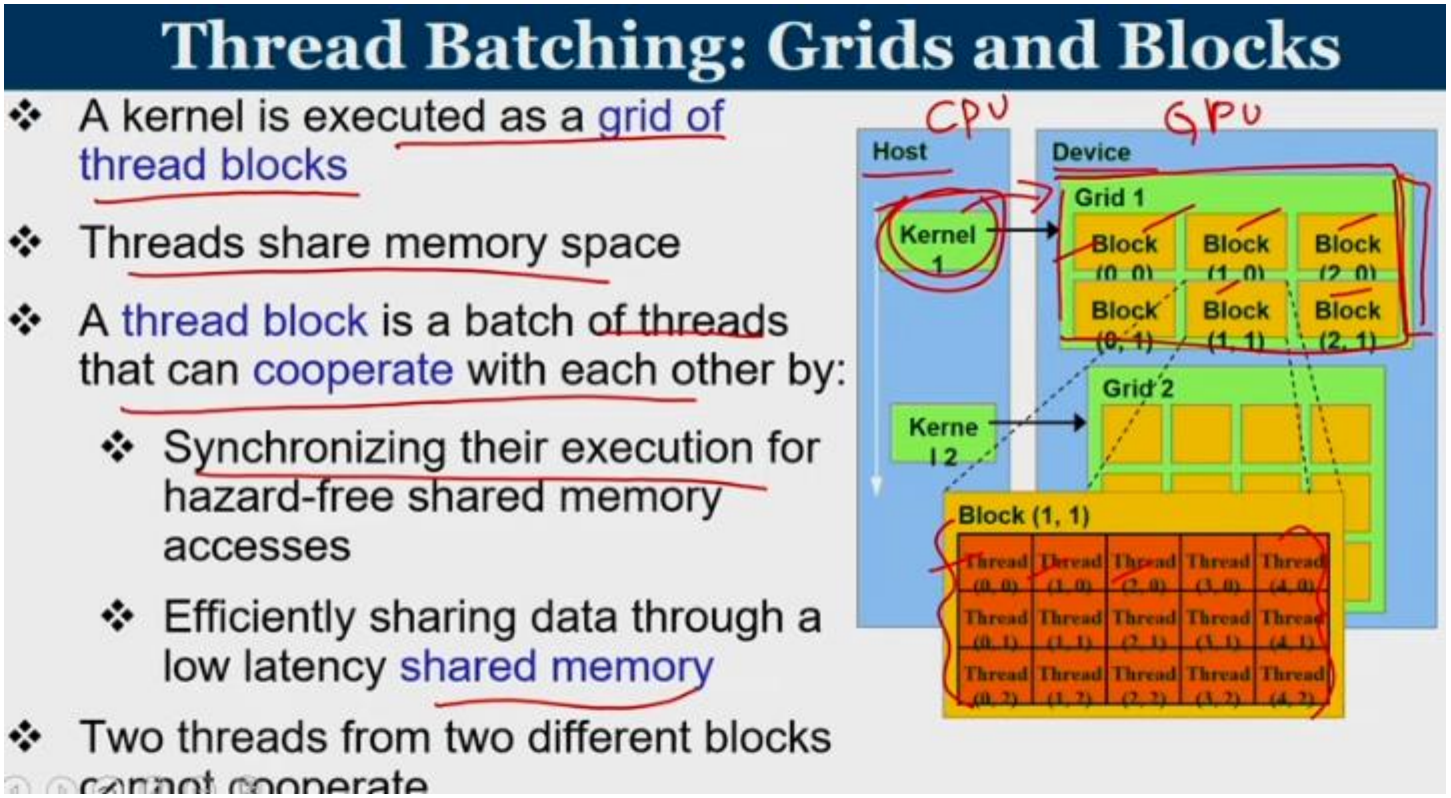
Graphics Processing Units

- The GPU is good at
  - data-parallel processing when the same computation is executed on many data elements in parallel. It has low control flow overhead.
  - high floating point arithmetic intensity operations that has many calculations per memory access and high floating point to integer ratio.

# NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

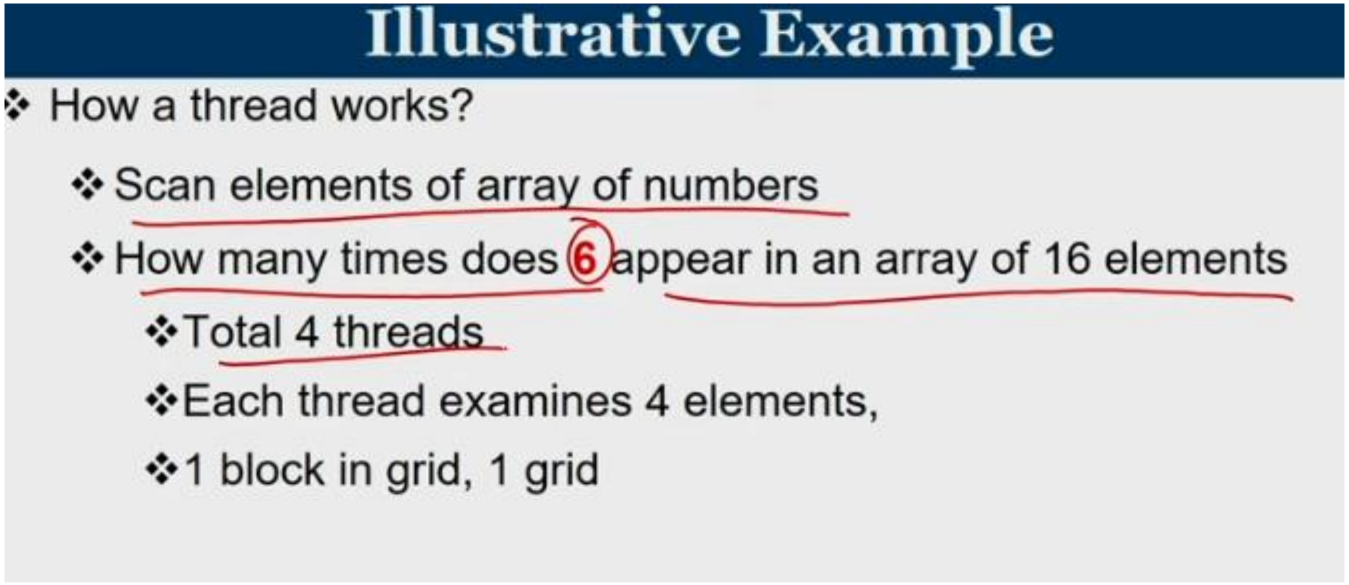
# NVIDIA GPU Architecture



## Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
- Threads share memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution for hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

# NVIDIA GPU Architecture



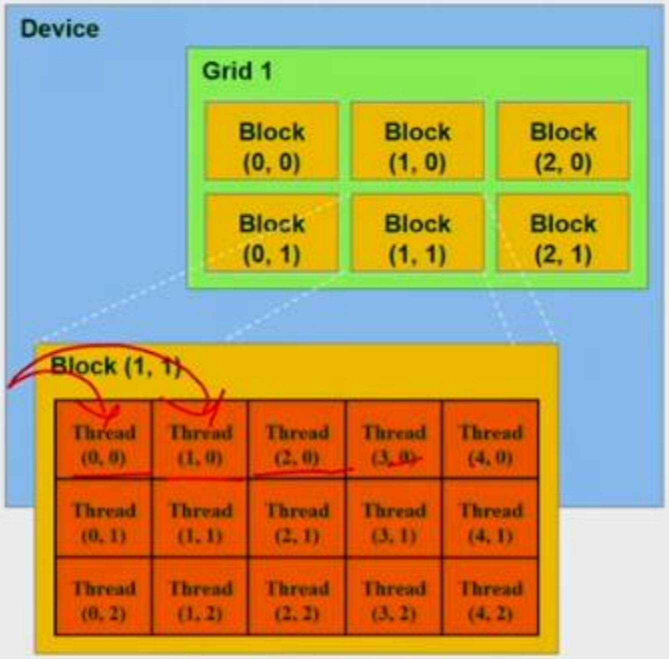
## Illustrative Example

- How a thread works?
  - Scan elements of array of numbers
  - How many times does $\bigcirc\!\!\!\!\!6$ appear in an array of 16 elements
    - Total 4 threads
    - Each thread examines 4 elements,
    - 1 block in grid, 1 grid

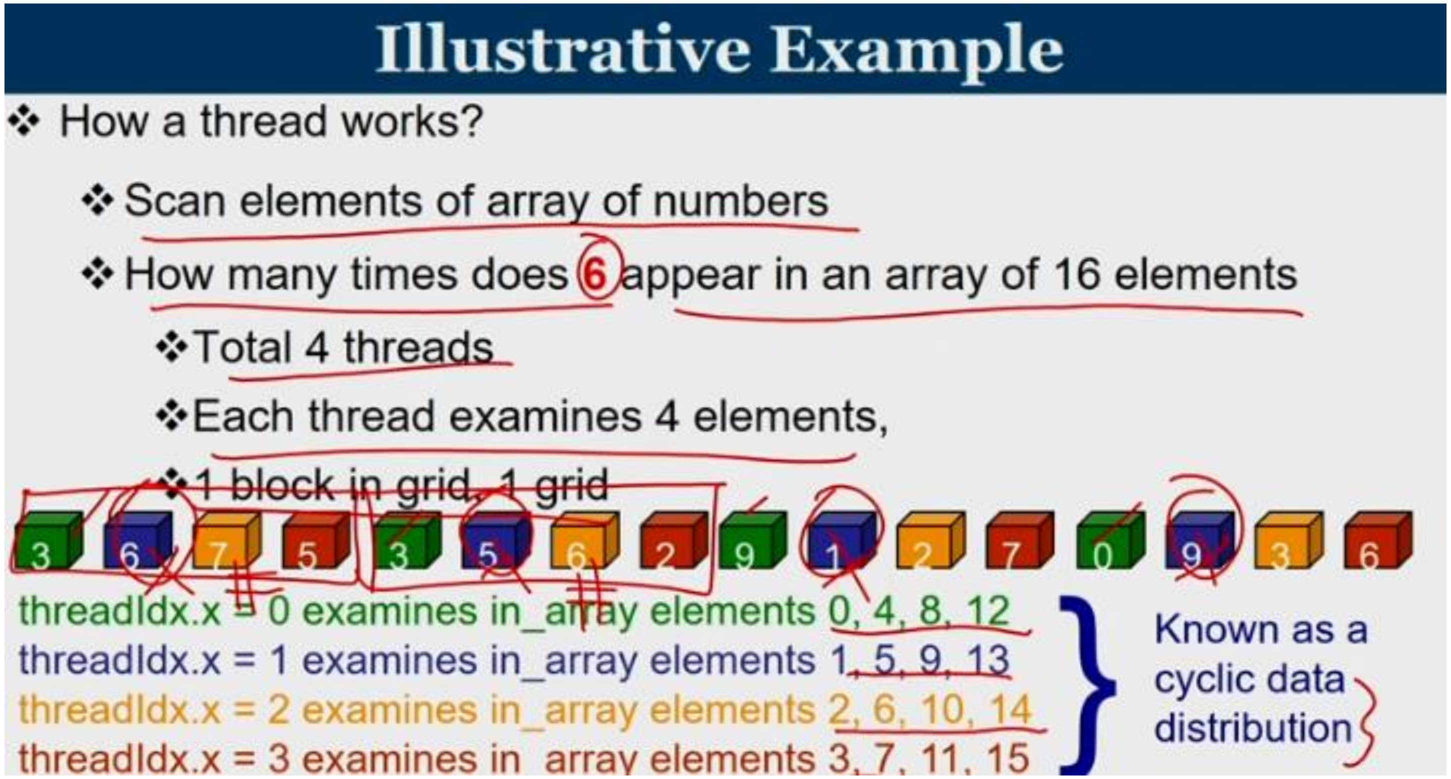
# NVIDIA GPU Architecture

## Block and Thread IDs

- Threads and blocks have IDs
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})
- Simplifies memory addressing when processing multidimensional data



# NVIDIA GPU Architecture



## Illustrative Example

- How a thread works?
  - Scan elements of array of numbers
  - How many times does 6 appear in an array of 16 elements
    - Total 4 threads
    - Each thread examines 4 elements,
    - 1 block in grid, 1 grid

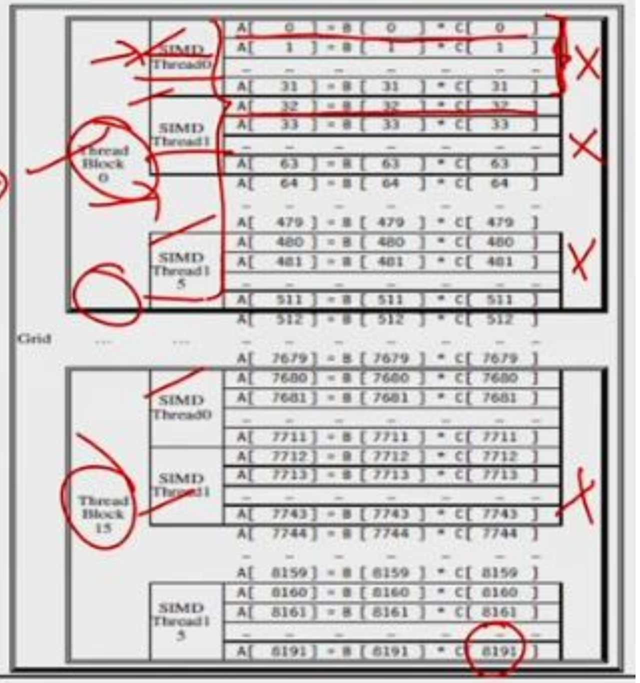
threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Known as a cyclic data distribution

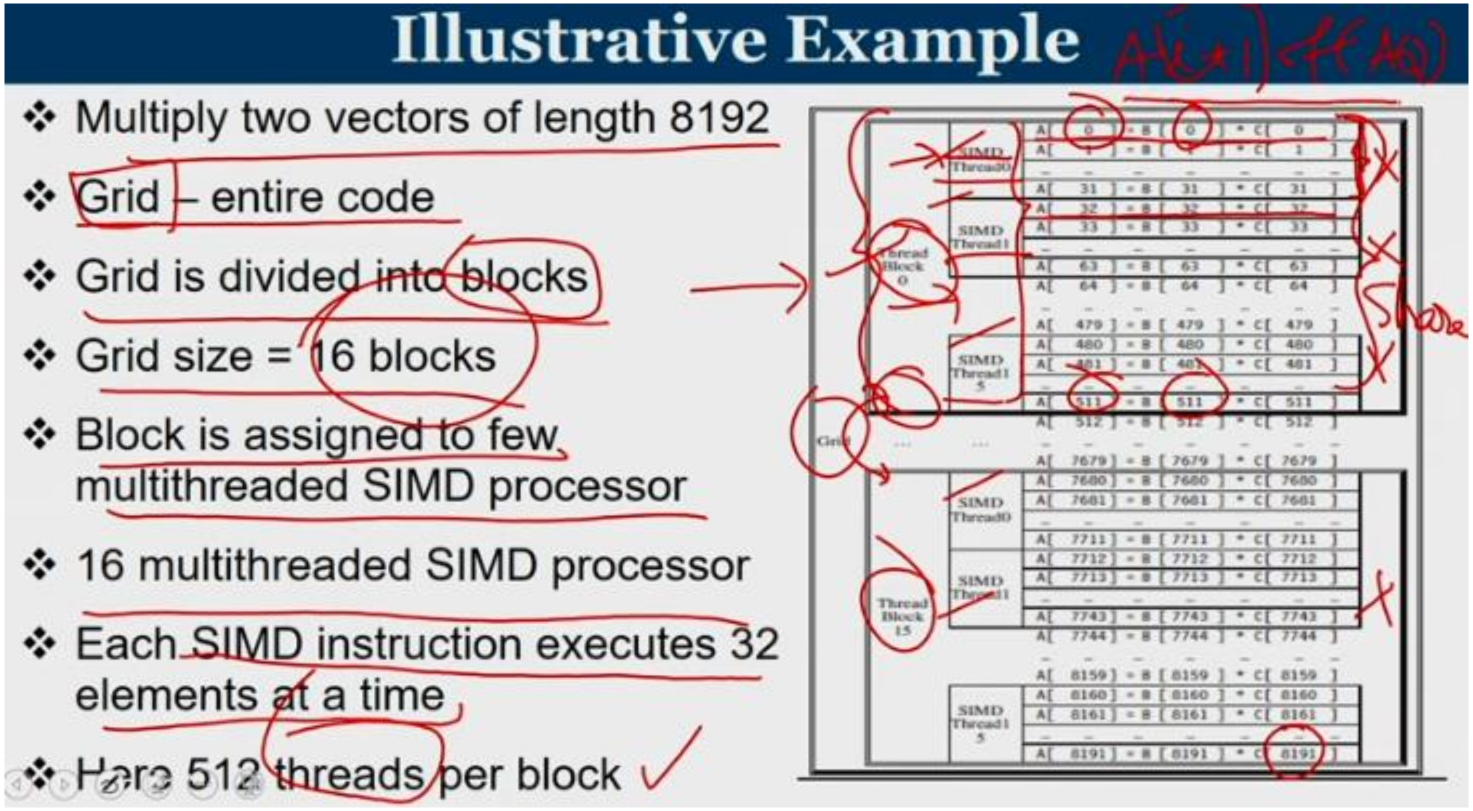
# NVIDIA GPU Architecture

## Illustrative Example

- Multiply two vectors of length 8192
- Grid – entire code
- Grid is divided into blocks
- Grid size = 16 blocks
- Block is assigned to few multithreaded SIMD processor
- 16 multithreaded SIMD processor
- Each SIMD instruction executes 32 elements at a time
- Here 512 threads per block ✓


Grid
Thread Block 0
SIMD Thread0
SIMD Thread1
SIMD Thread15
Thread Block 15
A[ 0 ] = B [ 0 ] * C[ 0 ]
A[ 1 ] = B [ 1 ] * C[ 1 ]
A[ 31 ] = B [ 31 ] * C[ 31 ]
A[ 32 ] = B [ 32 ] * C[ 32 ]
A[ 33 ] = B [ 33 ] * C[ 33 ]
A[ 63 ] = B [ 63 ] * C[ 63 ]
A[ 64 ] = B [ 64 ] * C[ 64 ]
A[ 479 ] = B [ 479 ] * C[ 479 ]
A[ 480 ] = B [ 480 ] * C[ 480 ]
A[ 481 ] = B [ 481 ] * C[ 481 ]
A[ 511 ] = B [ 511 ] * C[ 511 ]
A[ 512 ] = B [ 512 ] * C[ 512 ]
A[ 7679 ] = B [ 7679 ] * C[ 7679 ]
A[ 7680 ] = B [ 7680 ] * C[ 7680 ]
A[ 7681 ] = B [ 7681 ] * C[ 7681 ]
A[ 7711 ] = B [ 7711 ] * C[ 7711 ]
A[ 7712 ] = B [ 7712 ] * C[ 7712 ]
A[ 7713 ] = B [ 7713 ] * C[ 7713 ]
A[ 7743 ] = B [ 7743 ] * C[ 7743 ]
A[ 7744 ] = B [ 7744 ] * C[ 7744 ]
A[ 8159 ] = B [ 8159 ] * C[ 8159 ]
A[ 8160 ] = B [ 8160 ] * C[ 8160 ]
A[ 8161 ] = B [ 8161 ] * C[ 8161 ]
A[ 8191 ] = B [ 8191 ] * C[ 8191 ]

## NVIDIA GPU Architecture

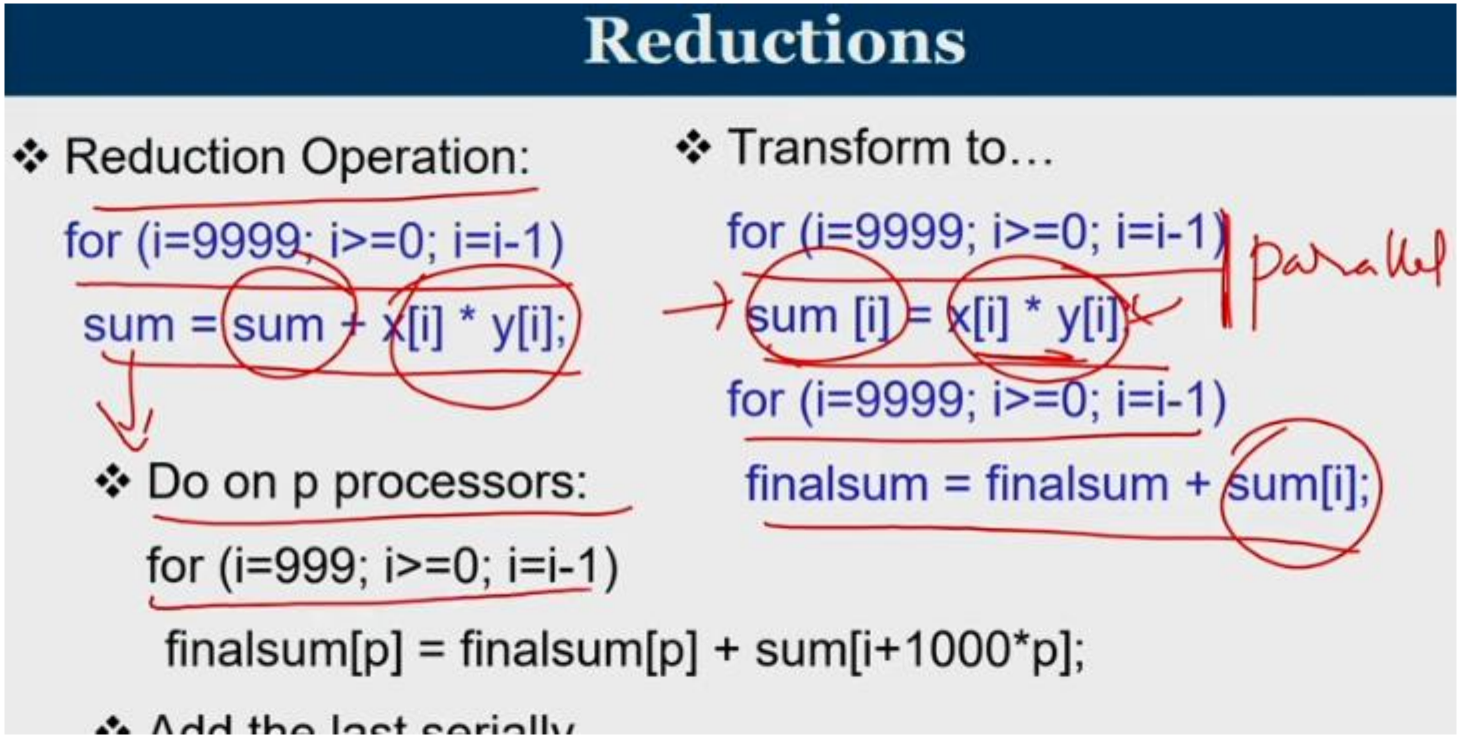


### Illustrative Example

- Multiply two vectors of length 8192
- Grid – entire code
- Grid is divided into blocks
- Grid size = 16 blocks
- Block is assigned to few multithreaded SIMD processor
- 16 multithreaded SIMD processor
- Each SIMD instruction executes 32 elements at a time
- Here 512 threads per block ✓

```
A[ 0 ] = B [ 0 ] * C[ 0 ]
A[ 1 ] = B [ 1 ] * C[ 1 ]
 …
A[ 31 ] = B [ 31 ] * C[ 31 ]
A[ 32 ] = B [ 32 ] * C[ 32 ]
A[ 33 ] = B [ 33 ] * C[ 33 ]
 …
A[ 63 ] = B [ 63 ] * C[ 63 ]
A[ 64 ] = B [ 64 ] * C[ 64 ]
 …
A[ 479 ] = B [ 479 ] * C[ 479 ]
A[ 480 ] = B [ 480 ] * C[ 480 ]
A[ 481 ] = B [ 481 ] * C[ 481 ]
 …
A[ 511 ] = B [ 511 ] * C[ 511 ]
A[ 512 ] = B [ 512 ] * C[ 512 ]
 …
A[ 7679 ] = B [ 7679 ] * C[ 7679 ]
A[ 7680 ] = B [ 7680 ] * C[ 7680 ]
A[ 7681 ] = B [ 7681 ] * C[ 7681 ]
 …
A[ 7711 ] = B [ 7711 ] * C[ 7711 ]
A[ 7712 ] = B [ 7712 ] * C[ 7712 ]
A[ 7713 ] = B [ 7713 ] * C[ 7713 ]
 …
A[ 7743 ] = B [ 7743 ] * C[ 7743 ]
A[ 7744 ] = B [ 7744 ] * C[ 7744 ]
 …
A[ 8159 ] = B [ 8159 ] * C[ 8159 ]
A[ 8160 ] = B [ 8160 ] * C[ 8160 ]
A[ 8161 ] = B [ 8161 ] * C[ 8161 ]
 …
A[ 8191 ] = B [ 8191 ] * C[ 8191 ]
```

## NVIDIA GPU Architecture


Reductions

❖ Reduction Operation:

```
for (i=9999; i>=0; i=i-1)
  sum = sum + x[i] * y[i];
```

❖ Transform to…

```
for (i=9999; i>=0; i=i-1)
  sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
  finalsum = finalsum + sum[i];
```

❖ Do on p processors:

```
for (i=999; i>=0; i=i-1)
  finalsum[p] = finalsum[p] + sum[i+1000*p];
```

❖ Add the last serially

## Example

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 threads per block
  - SIMD instruction executes 32 elements at a time
  - Thus grid size = 16 blocks
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

## Terminology

- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

## Example

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers

## NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - "Parallel Thread Execution (PTX)"
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Example:

```
shl.s32         R8, blockIdx, 9      ; Thread Block ID * Block size (512 or 29)
add.s32         R8, R8, threadIdx    ; R8 = i = my CUDA thread ID
ld.global.f64   RD0, [X+R8]          ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]          ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4                ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2                ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0            ; Y[i] = sum (X[i]*a + Y[i])
```

# Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

## Example

```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];

     ld.global.f64    RD0, [X+R8]                 ; RD0 = X[i]
     setp.neq.s32     P1, RD0, #0                 ; P1 is predicate register 1
     @!P1, bra        ELSE1, *Push                ; Push old mask, set new mask bits
                                                  ; if P1 false, go to ELSE1
     ld.global.f64    RD2, [Y+R8]                 ; RD2 = Y[i]
     sub.f64          RD0, RD0, RD2               ; Difference in RD0
     st.global.f64    [X+R8], RD0                 ; X[i] = RD0
     @P1, bra         ENDIF1, *Comp               ; complement mask bits
                                                  ; if P1 true, go to ENDIF1
ELSE1:                ld.global.f64 RD0, [Z+R8]   ; RD0 = Z[i]
                      st.global.f64 [X+R8], RD0   ; X[i] = RD0
ENDIF1:  <next instruction>, *Pop           ; pop to restore old mask
```

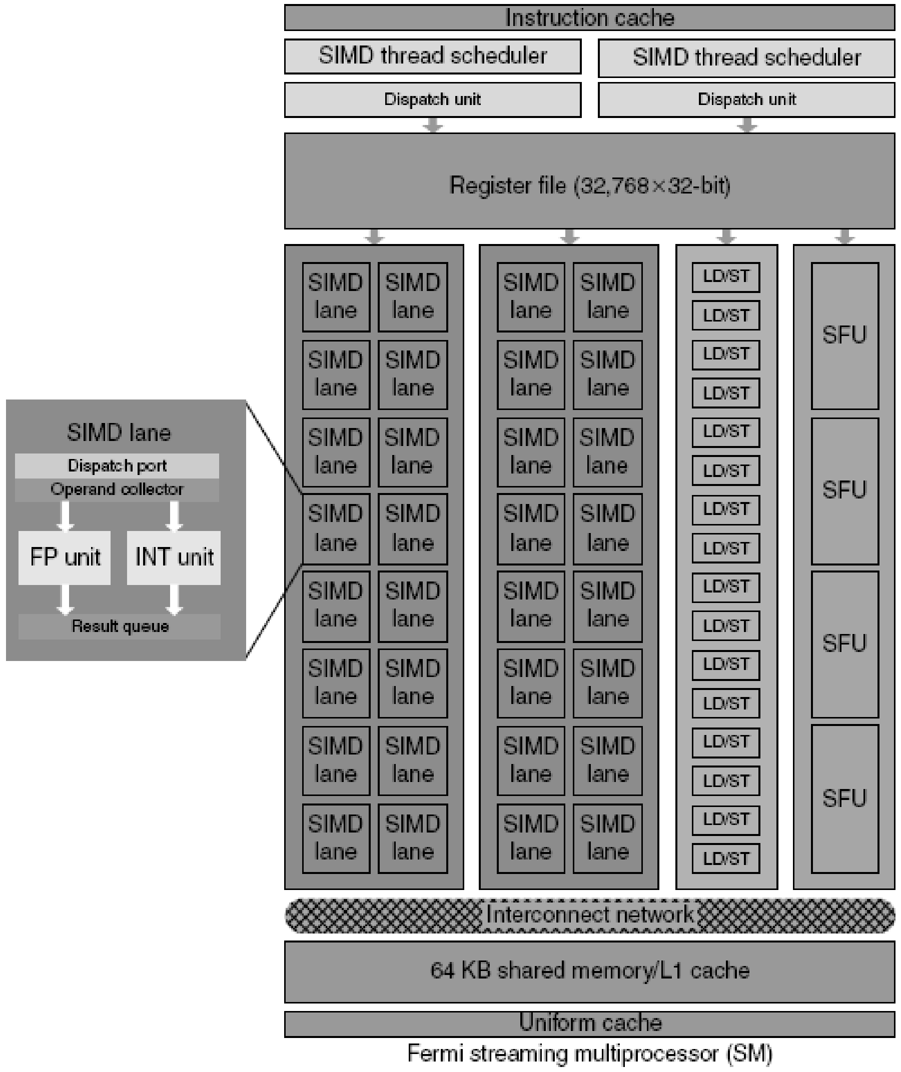
## NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - “Private memory”
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

## Fermi Architecture Innovations

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision
- Caches for GPU memory
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
- Faster atomic instructions

# Fermi Multithreaded SIMD Proc.



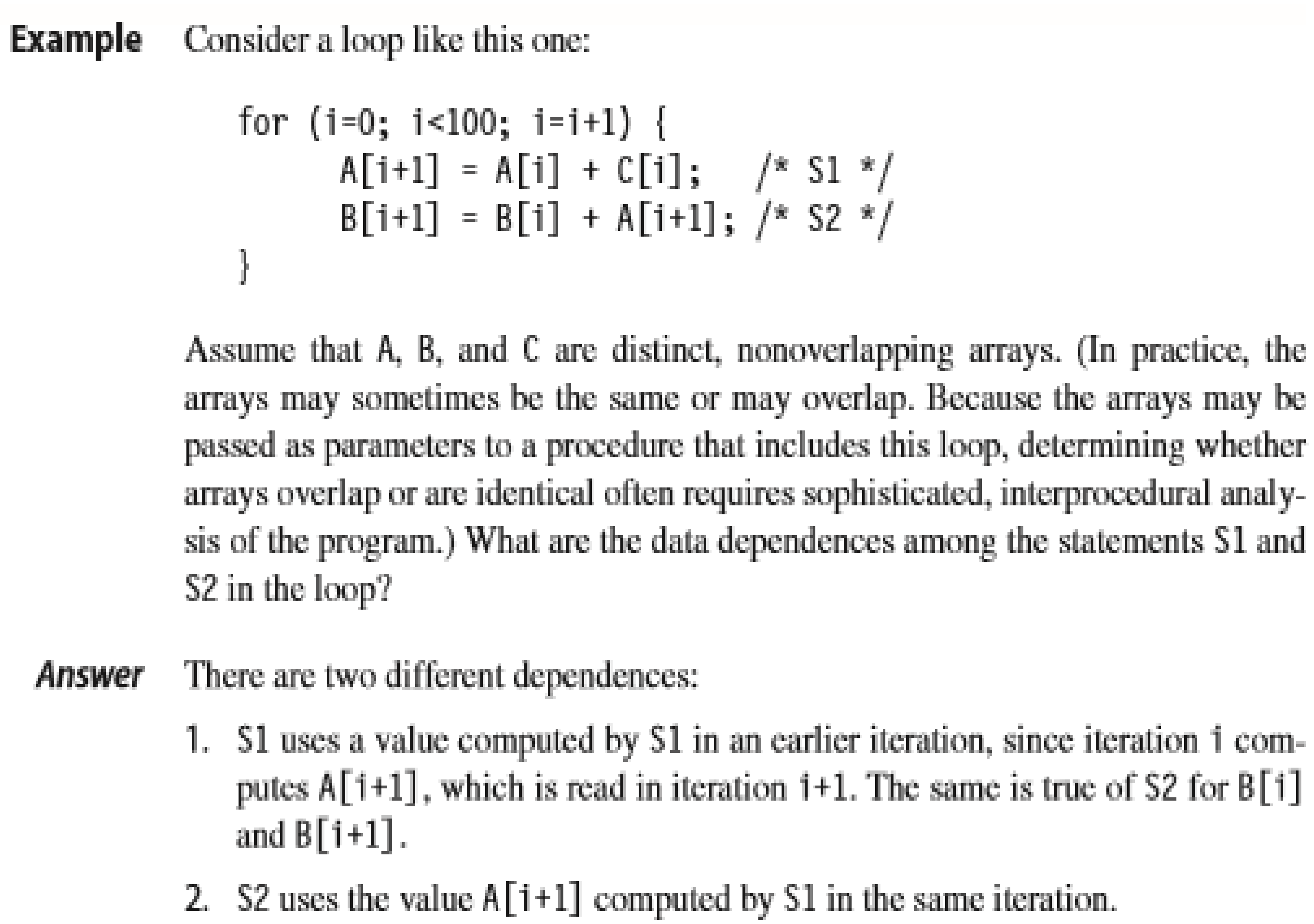
## Loop-Level Parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence

- Example 1:

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

- No loop-carried dependence



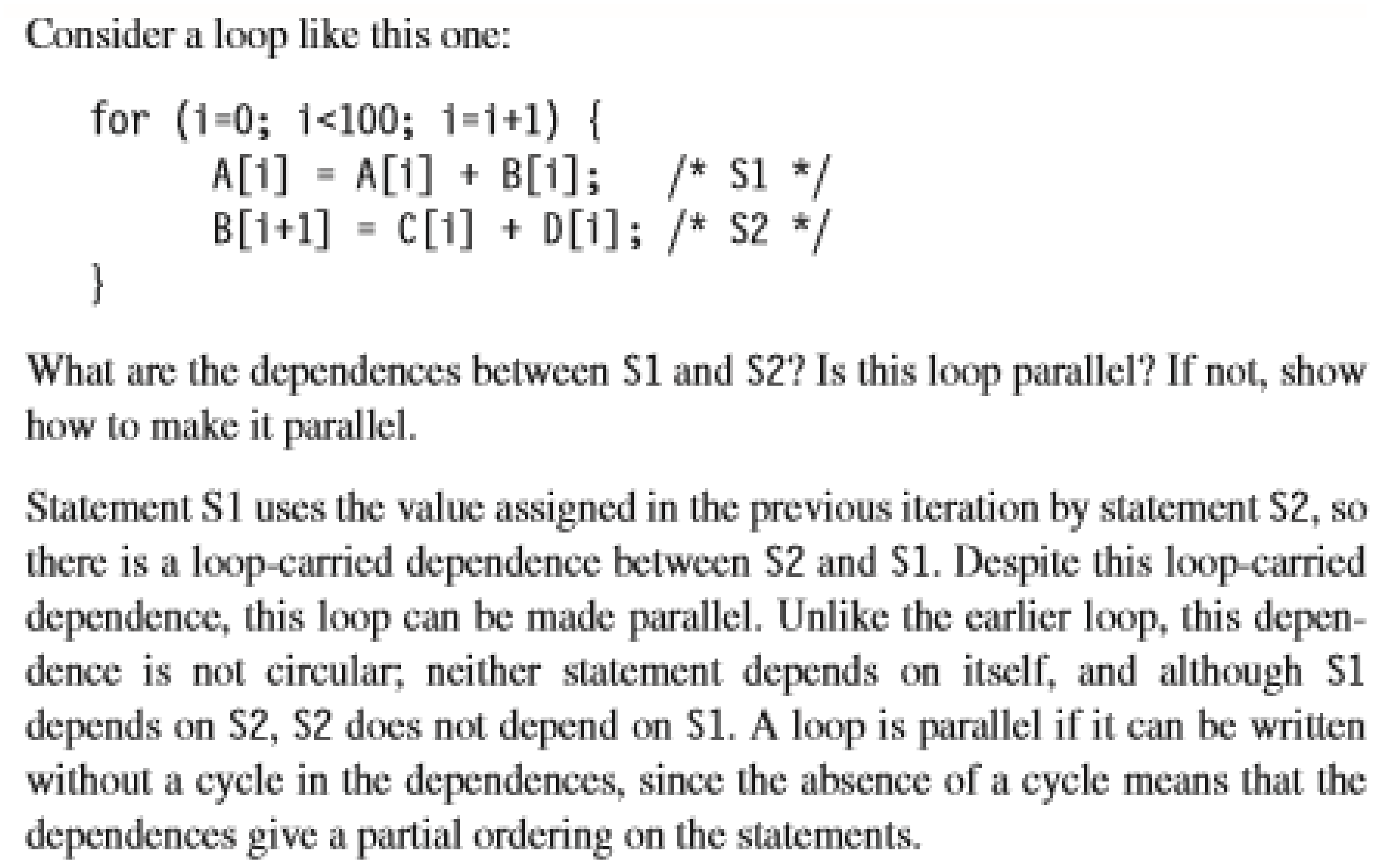
**Example** Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Assume that A, B, and C are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure that includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements S1 and S2 in the loop?

***Answer*** There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1], which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
2. S2 uses the value A[i+1] computed by S1 in the same iteration.

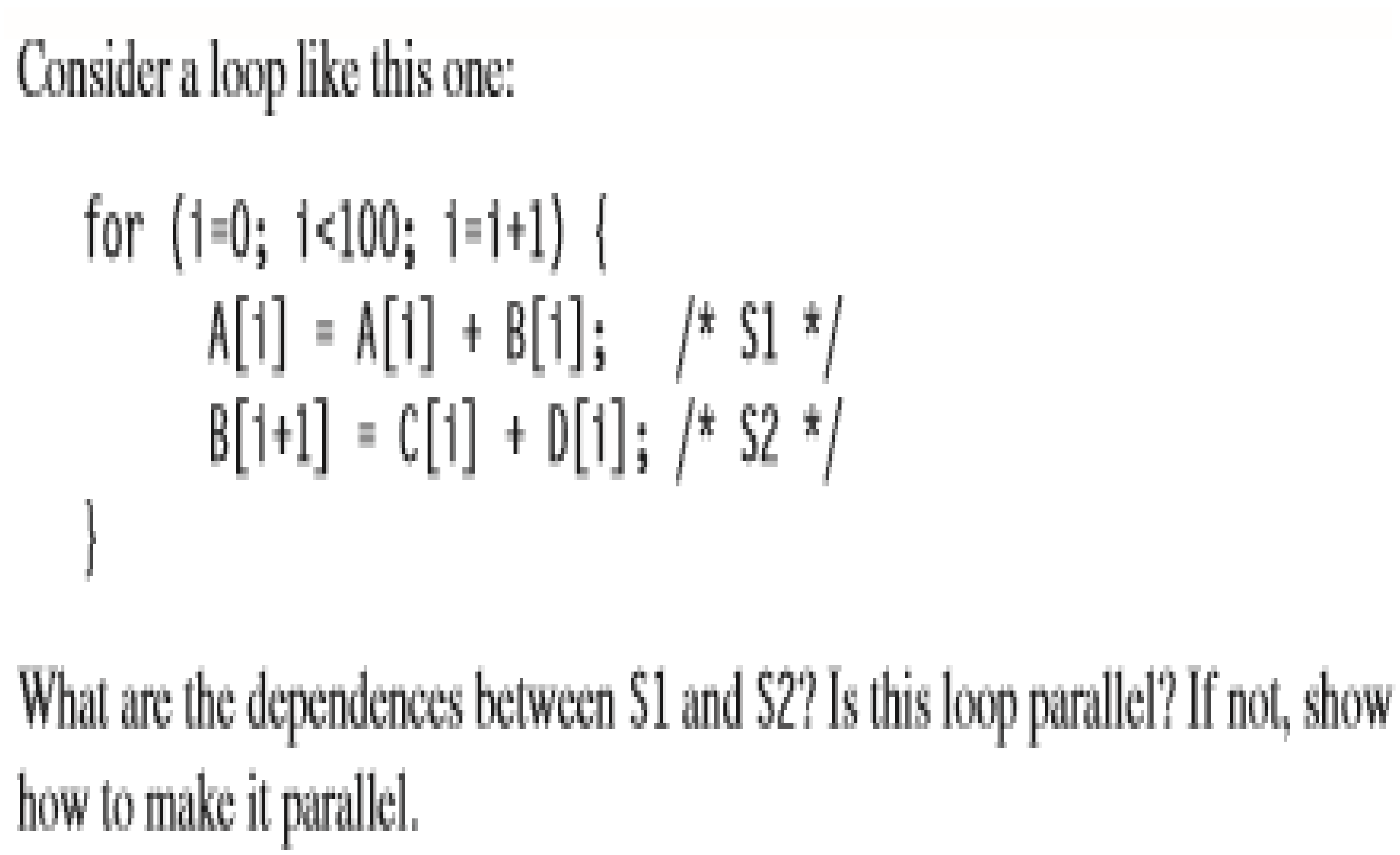


Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
      A[i] = A[i] + B[i];   /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

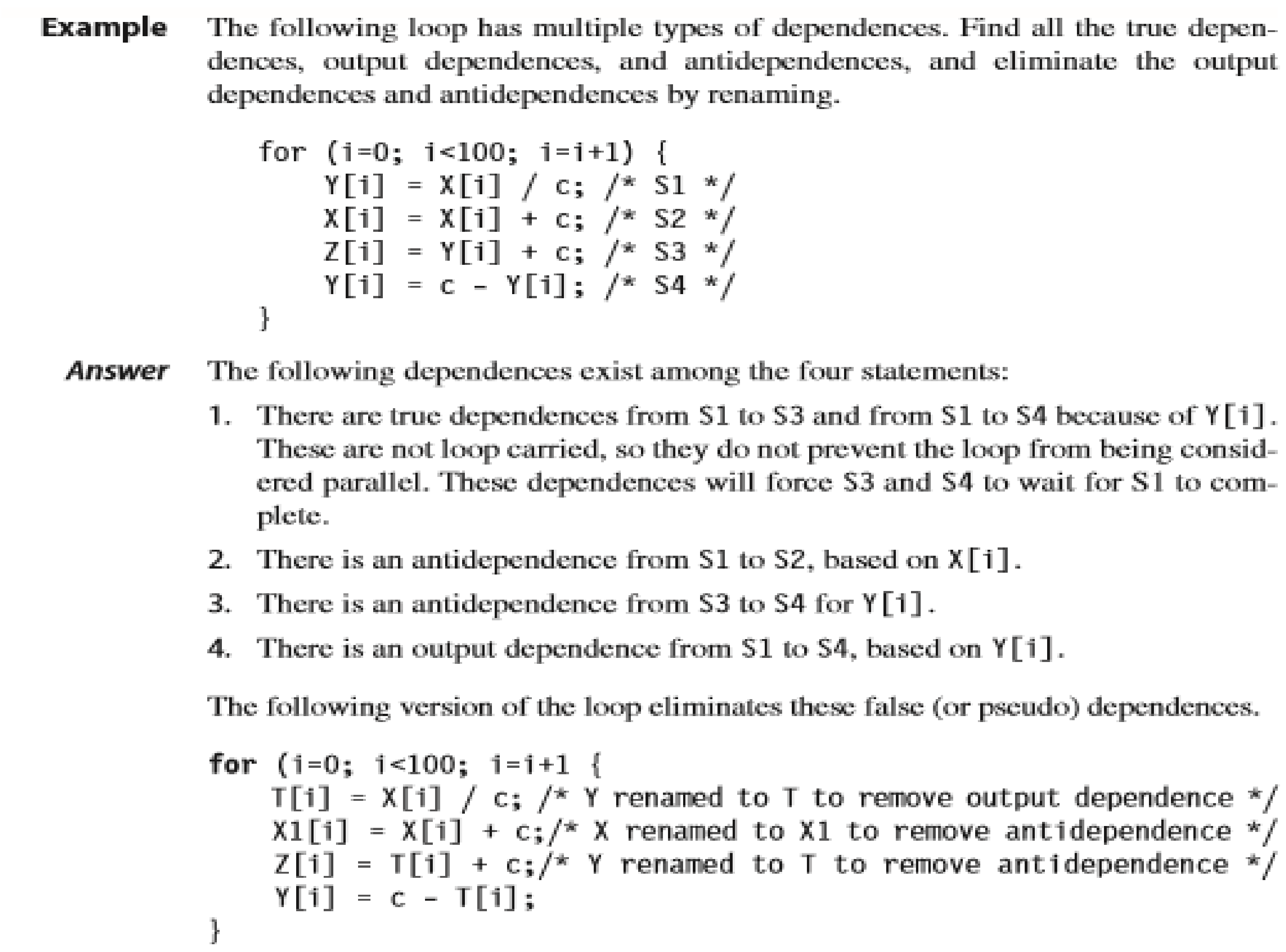
Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular; neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements.



Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i];    /* S1 */
        B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.



**Example** The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

***Answer*** The following dependences exist among the four statements:

1. There are true dependences from S1 to S3 and from S1 to S4 because of `Y[i]`. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.
2. There is an antidependence from S1 to S2, based on `X[i]`.
3. There is an antidependence from S3 to S4 for `Y[i]`.
4. There is an output dependence from S1 to S4, based on `Y[i]`.

The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=0; i<100; i=i+1 {
    T[i] = X[i] / c; /* Y renamed to T to remove output dependence */
    X1[i] = X[i] + c;/* X renamed to X1 to remove antidependence */
    Z[i] = T[i] + c;/* Y renamed to T to remove antidependence */
    Y[i] = c - T[i];
}
```

# Loop-Level Parallelism

- Example 2:

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i]; /* S1 */
        B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration

## Loop-Level Parallelism

- Example 3:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i]; /* S1 */
        B[i+1] = C[i] + D[i]; /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

# Loop-Level Parallelism

- Example 4:

```
for (i=0;i<100;i=i+1)  {
        A[i] = B[i] + C[i];
        D[i] = A[i] * E[i];
}
```

- Example 5:

```
for (i=1;i<100;i=i+1)  {
        Y[i] = Y[i-1] + Y[i];
}
```

## Finding dependencies

- Assume indices are affine:
  - $a \times i + b$ (i is loop index)
- Assume:
  - Store to $a \times i + b$, then
  - Load from $c \times i + d$
  - $i$ runs from $m$ to $n$
  - Dependence exists if:
    - Given $j$, $k$ such that $m \le j \le n$, $m \le k \le n$
    - Store to $a \times j + b$, load from $a \times k + d$, and $a \times j + b = c \times k + d$

## Finding dependencies

- Generally cannot determine at compile time
- Test for absence of a dependence:
  - GCD test:
    - If a dependency exists, GCD(*c*,*a*) must evenly divide (*d*-*b*)
- Example:

```
for (i=0; i<100; i=i+1) {
    X[2*i+3] = X[2*i] * 5.0;
}
```

# Finding dependencies

- Example 2:

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

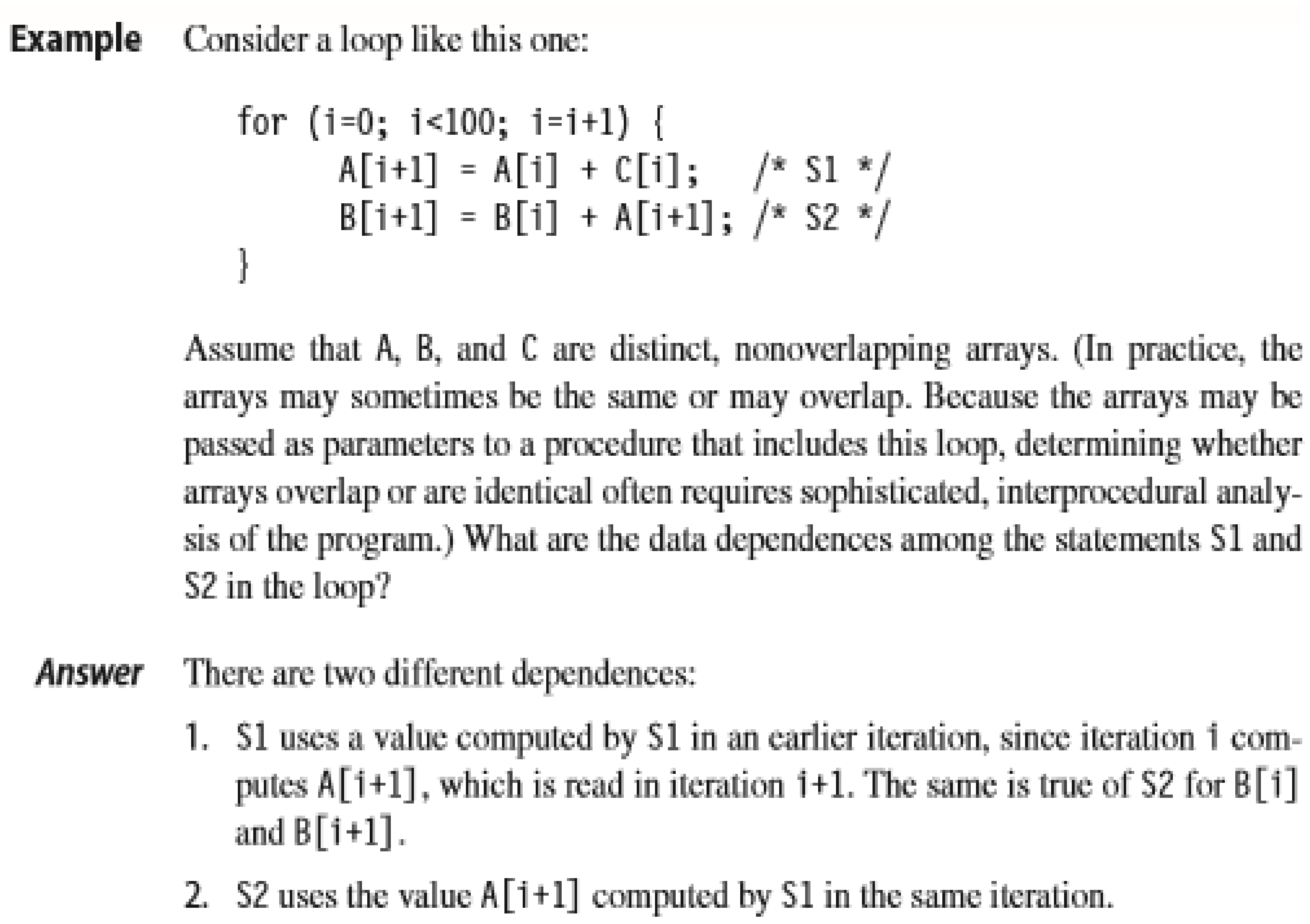
- Watch for antidependencies and output dependencies

## Finding dependencies

- Example 2:

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

- Watch for antidependencies and output dependencies



**Example** Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Assume that A, B, and C are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure that includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements S1 and S2 in the loop?

***Answer*** There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration `i` computes `A[i+1]`, which is read in iteration `i+1`. The same is true of S2 for `B[i]` and `B[i+1]`.
2. S2 uses the value `A[i+1]` computed by S1 in the same iteration.

Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
      A[i] = A[i] + B[i];   /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

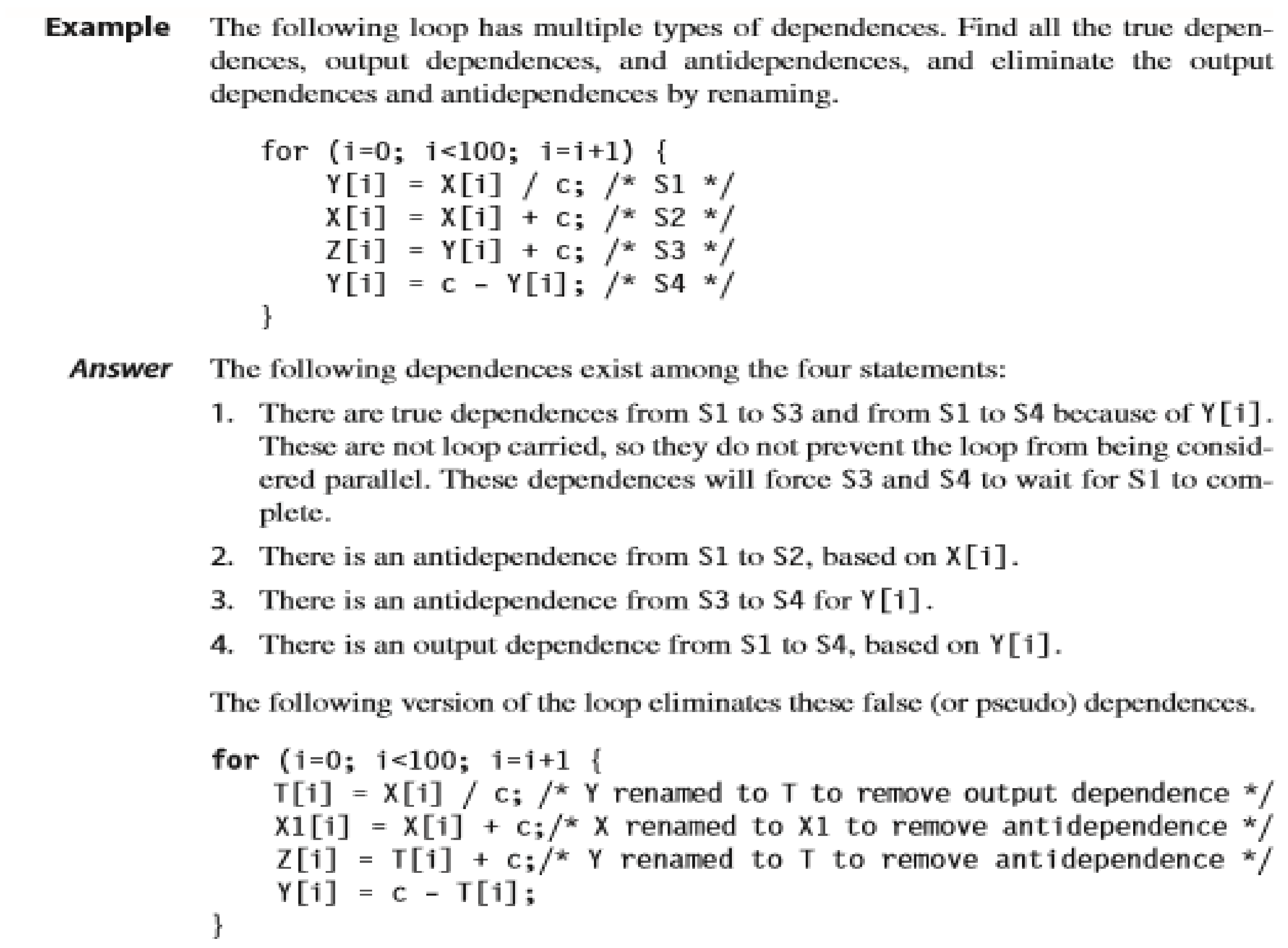
What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular; neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements.

Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i];    /* S1 */
        B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.



**Example** The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

***Answer*** The following dependences exist among the four statements:

1. There are true dependences from S1 to S3 and from S1 to S4 because of `Y[i]`. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.
2. There is an antidependence from S1 to S2, based on `X[i]`.
3. There is an antidependence from S3 to S4 for `Y[i]`.
4. There is an output dependence from S1 to S4, based on `Y[i]`.

The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=0; i<100; i=i+1 {
    T[i] = X[i] / c; /* Y renamed to T to remove output dependence */
    X1[i] = X[i] + c;/* X renamed to X1 to remove antidependence */
    Z[i] = T[i] + c;/* Y renamed to T to remove antidependence */
    Y[i] = c - T[i];
}
```

## Reductions

- Reduction Operation:

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];
```

- Transform to…

```
for (i=9999; i>=0; i=i-1)
        sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
        finalsum = finalsum + sum[i];
```

- Do on p processors:

```
for (i=999; i>=0; i=i-1)
        finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- Note:  assumes associativity!