

# Introduction to Parallel Computing

Sofien GANNOUNI  
Computer Science  
E-mail: [gnnosf@ksu.edu.sa](mailto:gnnosf@ksu.edu.sa) ; [gansof@yahoo.com](mailto:gansof@yahoo.com)

# Parallelism

## Parallel Computing:

- is a fundamental technique by which computations can be accelerated.
- is a form of computation in which many calculations are carried out simultaneously.

## Parallel Computers:

- classified according to the level at which the hardware supports parallelism:
  - **multi-core** and **multi-processor** computers having multiple processing elements within a single machine.
  - **clusters**, **Massively Parallel Processing**, and **grids** use multiple computers to work on the same task.

## Parallel Programming:

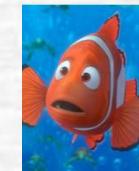
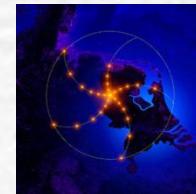
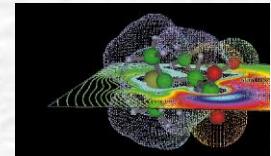
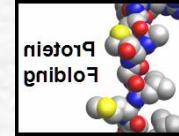
- Decomposing a programming problem into tasks
- Deploy the tasks on multiple processors and run them simultaneously
- Coordinating work and communications of those processors

# Why do we need Parallel Processing?

- sequential machines are reaching their speed limits.
- use multiple processors to solve large problems faster.
- microprocessors are getting cheaper and cheaper.
- Cheap multiprocessors and multi-core CPUs bring parallel processing to the desktop.
- Many applications need much faster machines

# Challenging Applications

- ☛ Modeling ozone layer, climate, ocean
- ☛ Quantum chemistry
- ☛ Protein folding
- ☛ General: *computational science*
- ☛ Aircraft modeling
- ☛ Using volumes of data from scientific instruments
  - ✿ astronomy
  - ✿ high-energy physics
- ☛ Computer chess
- ☛ Analyzing multimedia content



# History

- ☛ 1950s: first ideas (see Gill's quote)
- ☛ 1967 first parallel computer (ILLIAC IV)
- ☛ 1970s programming methods, experimental machines
- ☛ 1980s: parallel languages (SR, Linda, Orca), commercial supercomputers
- ☛ 1990s: software standardization (MPI), clusters, large-scale machines (Blue Gene)
- ☛ 2000s: grid computing: combining resources world-wide (Globus), General Purpose-GPU

# Opportunities to use parallelism

## Instruction Level Parallelism

- Hidden Parallelism in computer programs

## Single computer level

- Multi-core computers: Chip multi-processors
  - Dual-core, Quad-core, GP-GPU
- Multi-processor computers: Symmetric multi-processors
  - Super-computers

## Multiple computers level

- Clusters, Servers, Grid computing

# Parallelism in Computer Programs

- ➊ The main motivation for executing program instructions in parallel is to complete a computation faster.
- ➋ But programs are written assuming that:
  - Instructions are executed in **sequential**.
  - Most programming languages embed sequential execution.

# Instruction Level Parallelism (IPL)

## ☞ $(a+b) * (c+d)$

- could be computed simultaneously.

## ☞ Separation of instructions and data.

- Instructions and memory references execute in parallel without interfering.

## ☞ Instruction Execution is pipelined

## ☞ Processors initiate more than one instruction at a time.

# Parallel Computing vs. Distributed Computing

## Parallel computing

- Provides performance that a single processor can not give.
- Interaction among processors is frequent
  - Fine grained with low overhead
  - Assumed to be reliable.

## Distributed Computing

- Provides:
  - Availability, Reliability
  - Physical distribution, Heterogeneity
- Interaction is infrequent,
  - Coarse grained - heavier weight
  - Assumed to be unreliable.

# Aspects of Parallel Computing

## Parallel Computers Architecture

## Algorithms and applications

- Reasoning about performance
- Designing parallel algorithms.

## Parallel Programming

- Paradigms
- Programming Models
- Programming languages
- Frameworks
- Dedicated environments

# Understanding Parallel Computers

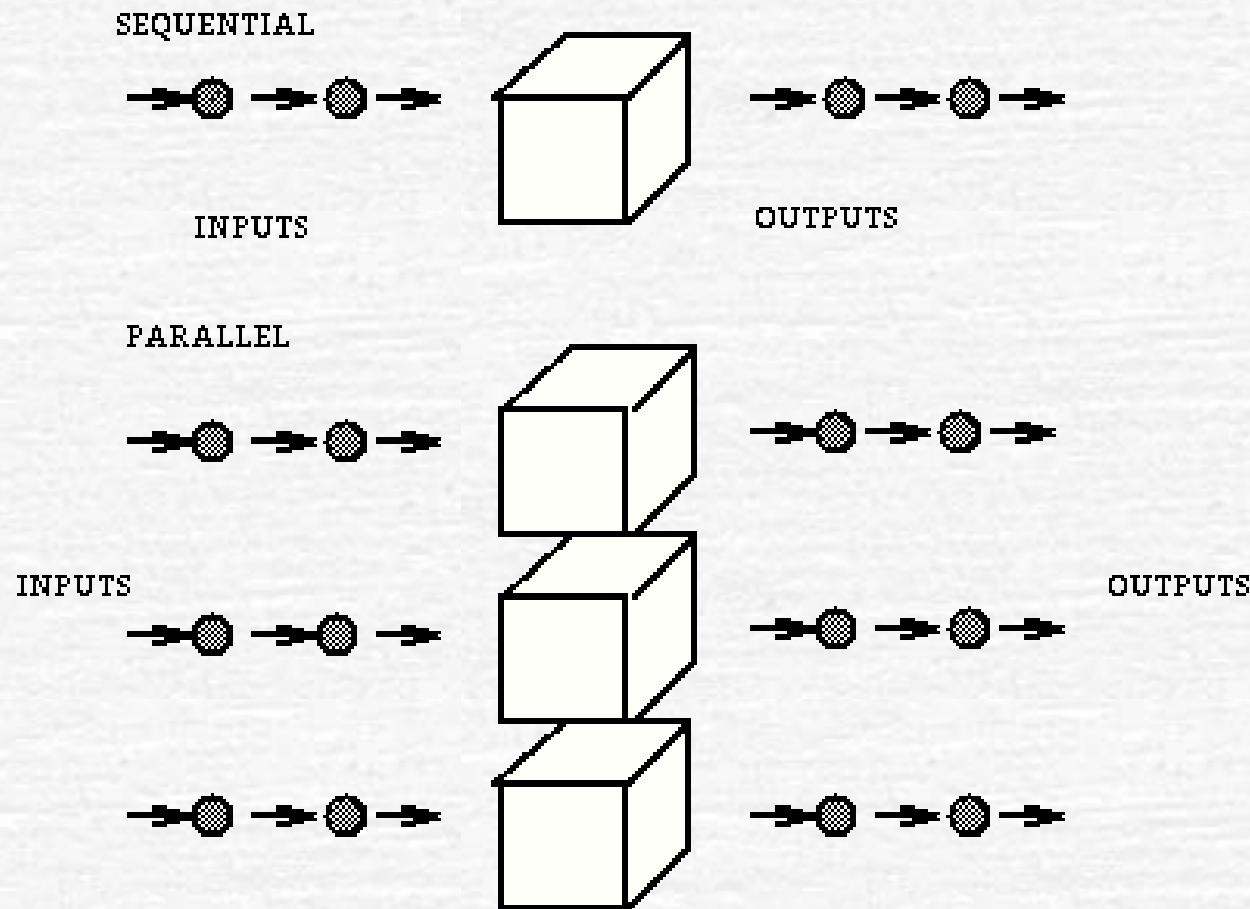
Sofien GANNOUNI  
Computer Science  
E-mail: [gnnosf@ksu.edu.sa](mailto:gnnosf@ksu.edu.sa) ; [gansof@yahoo.com](mailto:gansof@yahoo.com)

# Classification of Architectures – Flynn's classification

- ☛ Single Instruction Single Data (**SISD**): Serial Computers
- ☛ Single Instruction Multiple Data (**SIMD**)
  - Processor arrays
- ☛ Multiple Instruction Single Data (**MISD**): Not popular
- ☛ Multiple Instruction Multiple Data (**MIMD**)
  - Most popular
  - most supercomputers, clusters, computational Grids etc

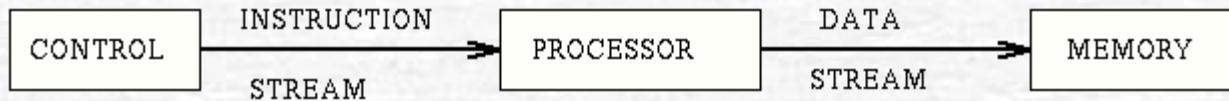
Single Instruction	Single Data	Multiple Data
Single Instruction	<b>SISD</b>	<b>SIMD</b>
<b>MISD</b>	<b>MIMD</b>	

# Sequential vs. Parallel Processing



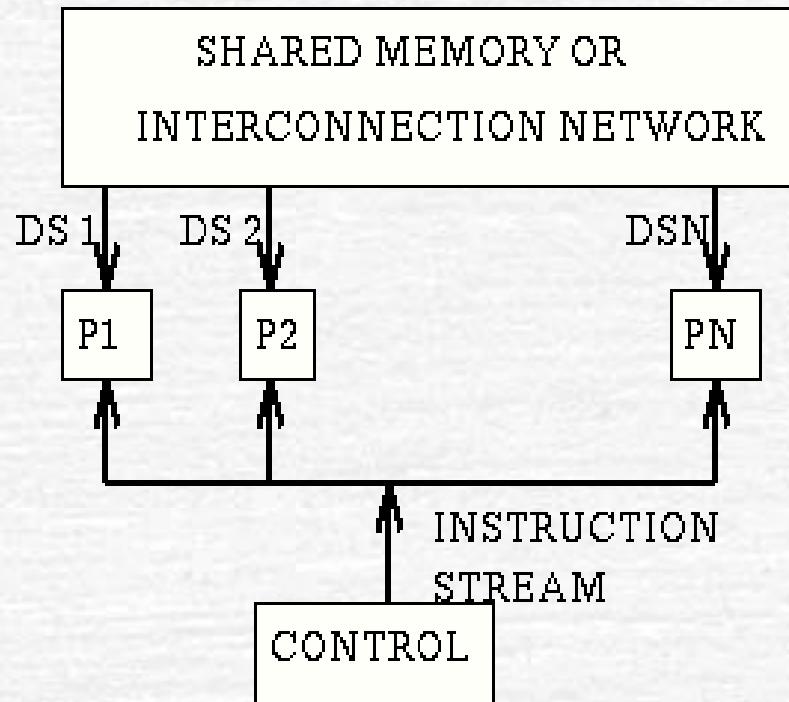
# SISD Computers

- ☞ a stream of instructions (the algorithm) tells the computer what to do.
- ☞ a stream of data (the input) is affected by these instructions.



# SIMD Computers

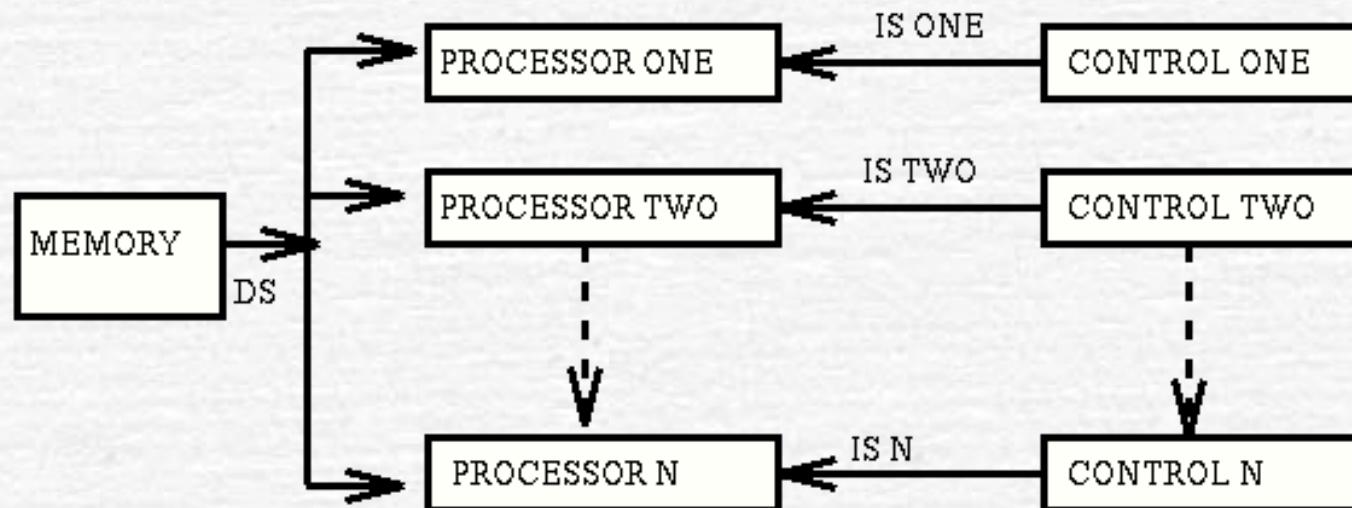
- A single instruction stream is broadcasted to multiple processors, each having its own data stream



P = PROCESSOR

DS = DATA STREAM

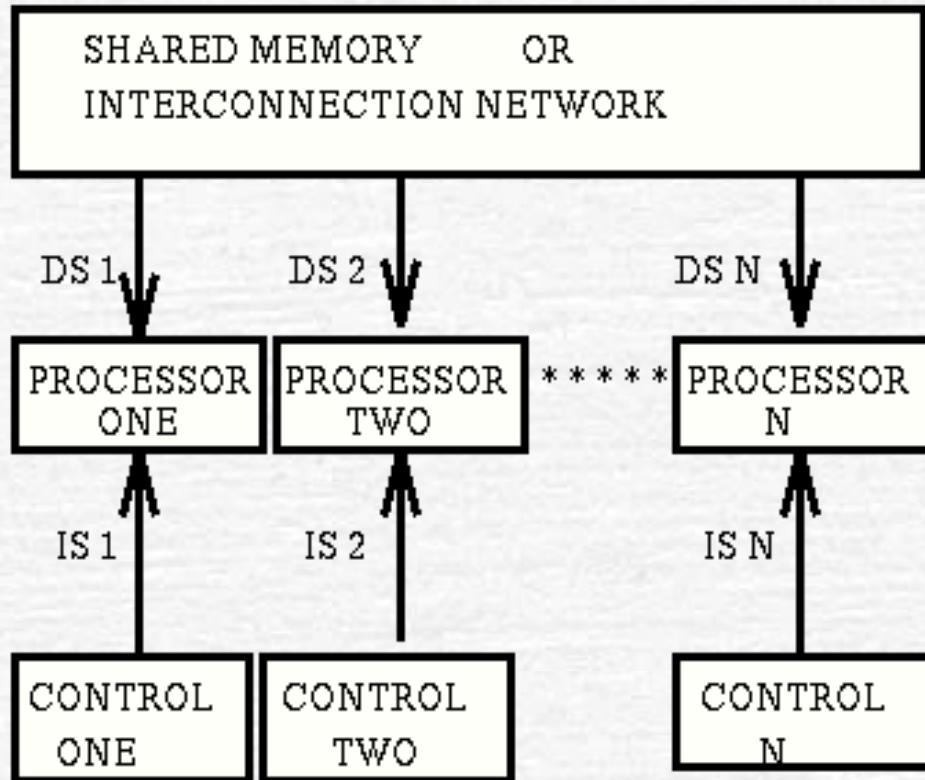
# MISD Computers



IS = INSTRUCTION STREAM

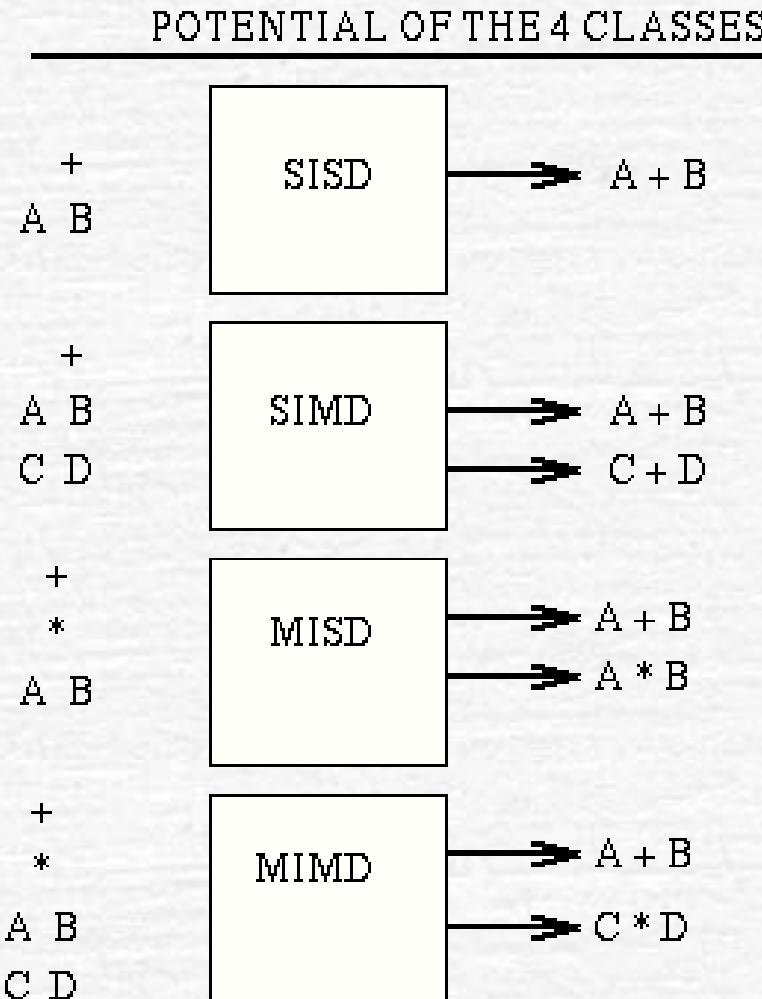
DS = DATA STREAM

# MIMD (multiprocessors / multiccomputers)



DS = DATA STREAM    IS = INSTRUCTION STREAM

# Potential of the 4 Classes



# Chip Multiprocessors

## Improvement in Silicon Technology (Moore's Law)

- Delivery of multiple instruction execution engines (called **cores**) on a single silicon chip.
- IBM PowerPC 970 in 2002 is the first multi-core.
- AMD introduced a Dual Core Opteron chip in 2005.
- Intel introduced Core Duo Pentium in 2006.

## Intel Core Duo:

- 2 32-bit Pentium processors on a single chip.
- Each processor has its own 32KB L1 data and instruction caches.
- Shared 2MB or 4MB L2 cache.
- Shared memory controller, I/O controller and son on.

## AMD Dual Core Opteron:

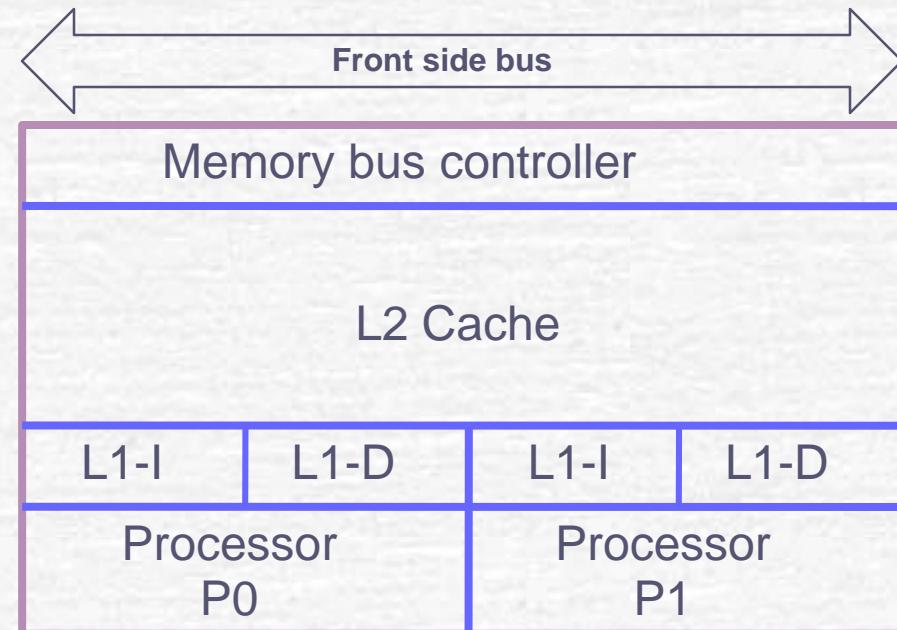
- 2 AMD64 processors on a single chip.
- Each processor has 64KB L1 data and instruction caches
- Separate 1 MB L2 cache per processor.

# Intel Core Duo

- The bus controller mediates transfer between the L2-cache and the RAM.

- When a processor references a location in the RAM,

- The Processor get a local copy of that location in its L1-cache passing by the L2-cache.
  - Fast access to their local copy of the location.



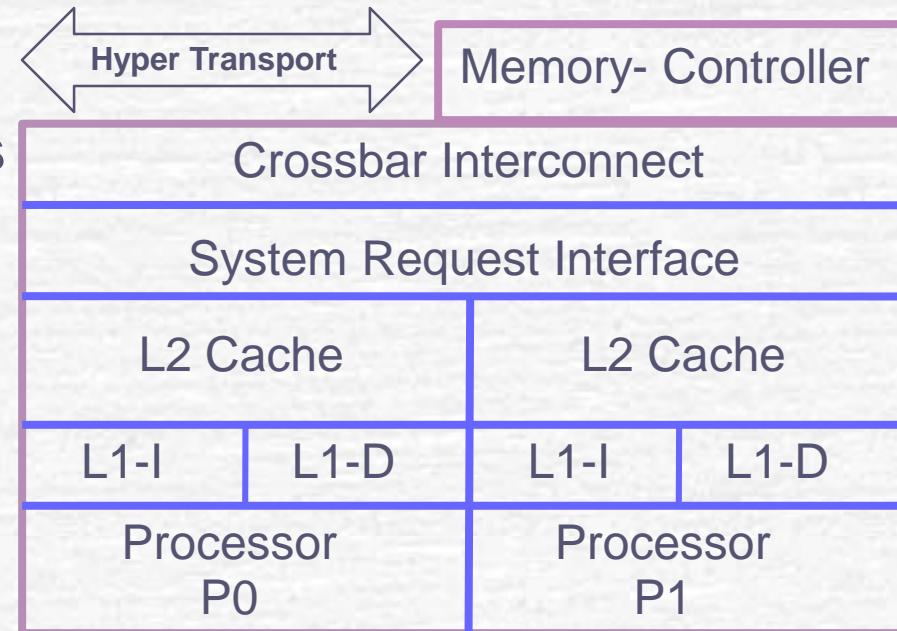
- A *cache coherency protocol* is used to avoid access to *stale* values.
  - *MESI* protocol: Modified, Exclusive, Shared and Invalid are the 4 possible states of a cache line.

# AMD Dual Core Opteron

- The System Request Interface (SRI) handles the memory coherency responsibilities.

- A *cache coherency protocol* is used:

- *MOESI* protocol: extends MESI by adding an “Owned” state.



# Symmetric Multiprocessor Arch.

## All processors access a single logical memory.

- A portion of the memory is sometimes near to each processor (on the same board).
- Sun Fire E25K

## To achieve consistency:

- The processors are connected to the bus.
  - Snoop the memory reference activity (the Cache control).

- Lets consider:

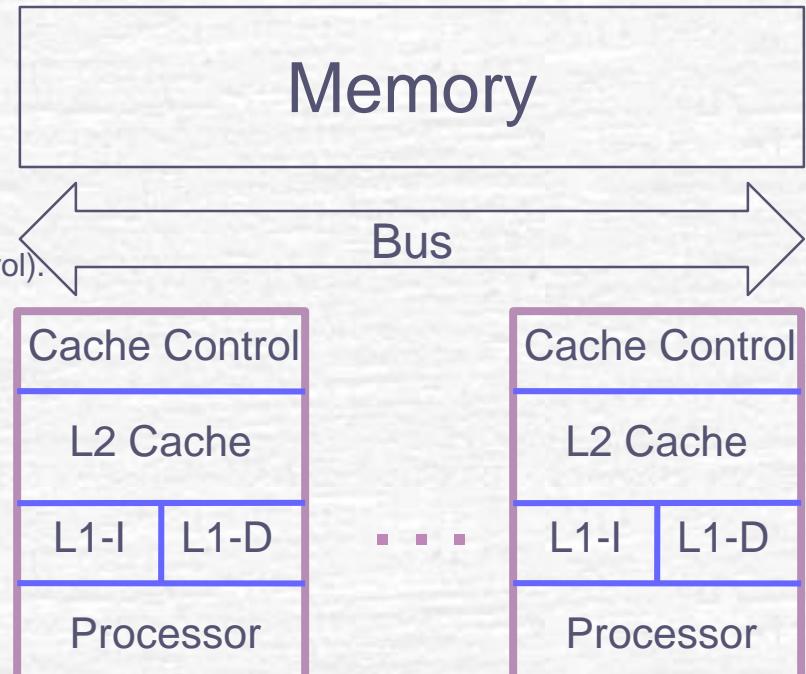
P0 request block x from memory.

P1 already has the block.

- P1 notices P0 and tags its copy as shared  
(The processor cache Control).

P2 requests writing for block x.

- P0 and P1 see the request.
- P0 and P1 invalidate their copies of x.
- P2 is owner of the block x and no other processor can use it until it is changed.



# Symmetric Multiprocessor Arch.

- ☞ **The common connection point, the bus, is a potential bottleneck.**
  - One memory operation takes place at a time.
  - The serial use of the bus limits the number of processors that can be connected.
- ☞ **SMPs achieve high performance in two ways:**
  - Being small and clustered near the bus.
  - Using sophisticated caching protocols
    - Tend to use the shared resource of the bus efficiently.
- ☞ **Sun Fire E25K is an example of SMP.**
  - Up to 72 processors, each is capable of executing two hardware threads.
  - 18 snoopy buses.
  - Access latency to shared memory is equal for all processors.
  - Shared memory of 1.15 TB

# Heterogeneous Chip Design

## ➤ An alternative to replicating a standard processor:

- Augment a standard processor with one or more specialized compute engines called attached processors.

## ➤ The idea is:

- The standard processor performs the hard-to-parallelize portion of computation.
- The attached processors perform the compute-intensive portion of the computation.

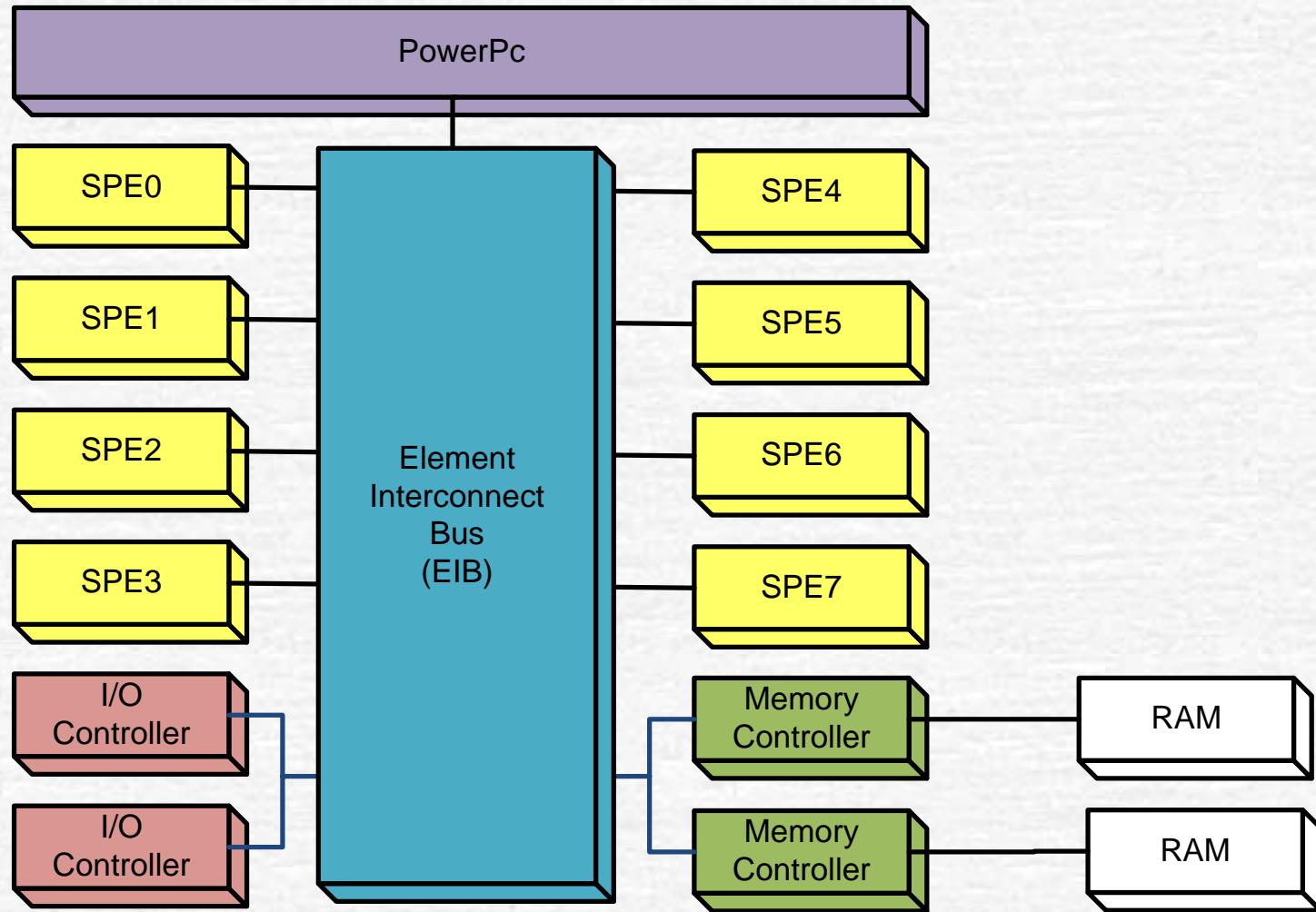
## ➤ Among the familiar variations on this design:

- Graphics Processing Units.
- Field Programmable Gate Arrays.
- Cell processor

# Cell Processor

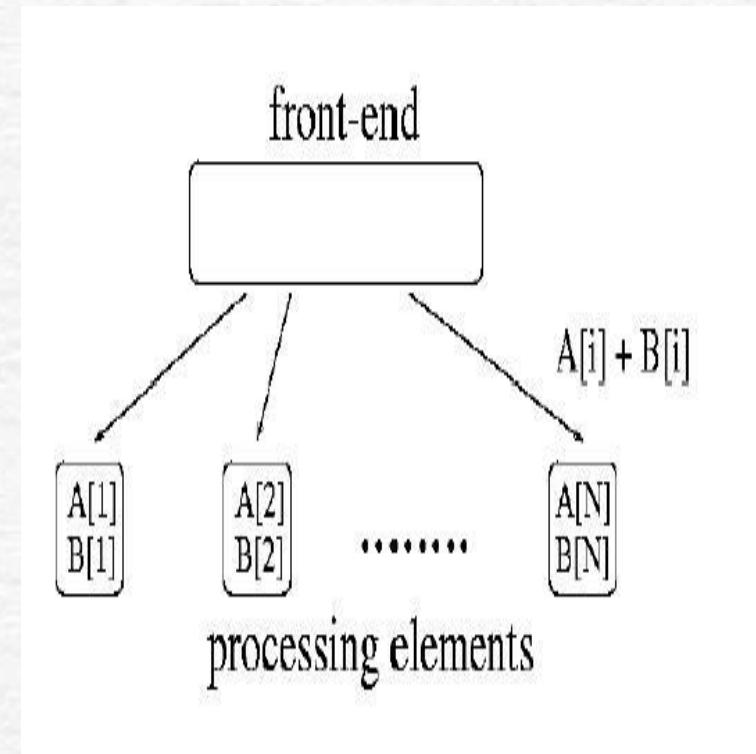
- Targeting the video market.
- A joint development by Sony, IBM and Toshiba.
- Cell:
  - 64-bit PowerPC core
  - 8 specialized cores called *synergistic processing elements*.
  - High communication bandwidth among processors 12.8 GB/s
- Cell do not provide coherent memory for the SPE

# Cell Architecture



# Processor Arrays

- Instructions operate on vectors
- Processor array = front-end + synchronized processing elements
- Front-end
  - Sequential machine that executes program
  - Vector operations are broadcast to PEs
- Processing element
  - Performs operation on its part of the vector
  - Communicates with other PEs through pipes



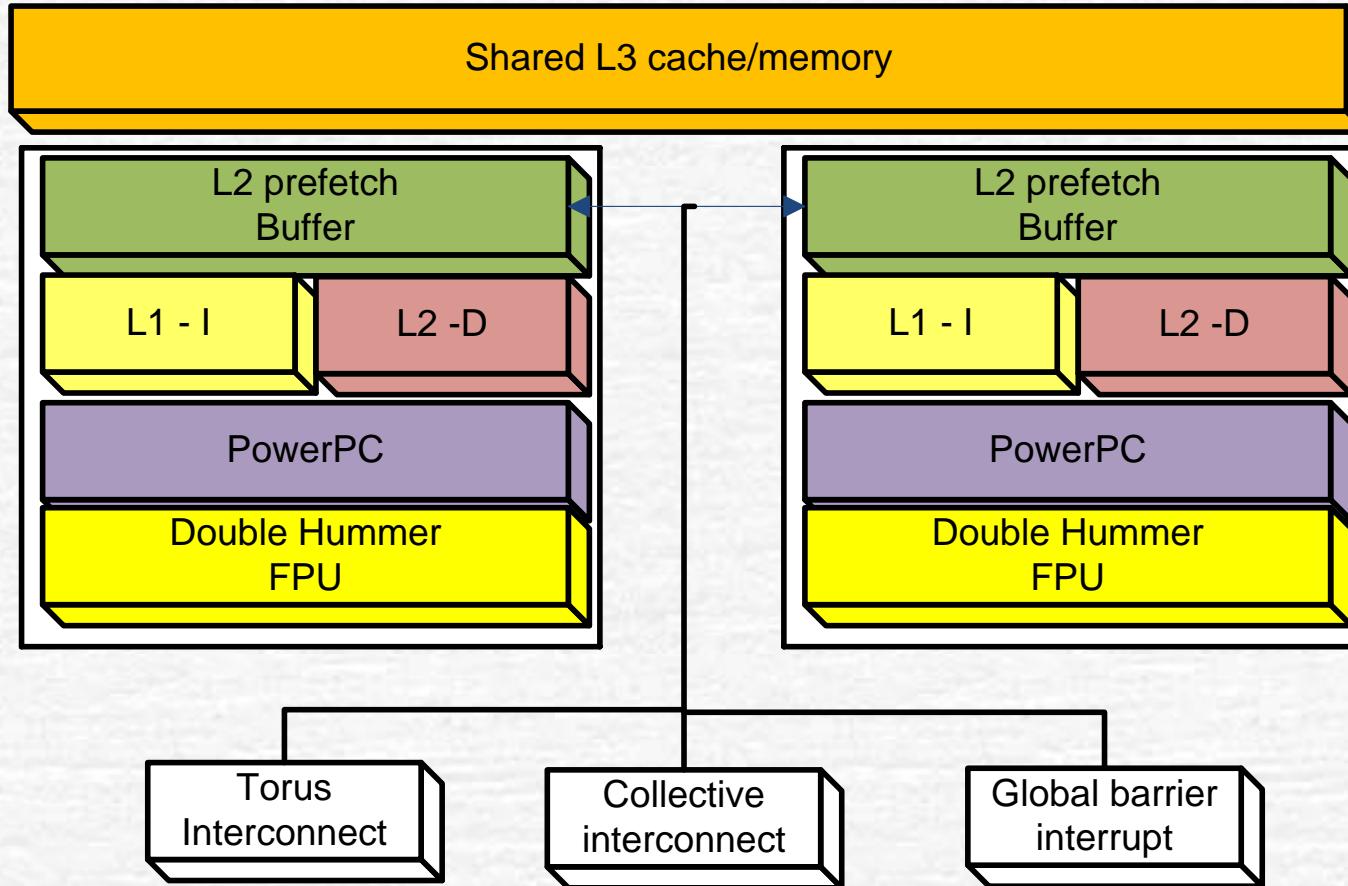
# Clusters

- Clusters are parallel computers made from commodity parts
- The nodes are boards containing
  - one or few processors
  - RAM memory
  - Disk Storage
- The nodes are interconnected
  - Ethernet, Myrinet,...
- The key property of clusters :
  - Memory is not shared
  - Processors communicates by message passing.

# Supercomputers

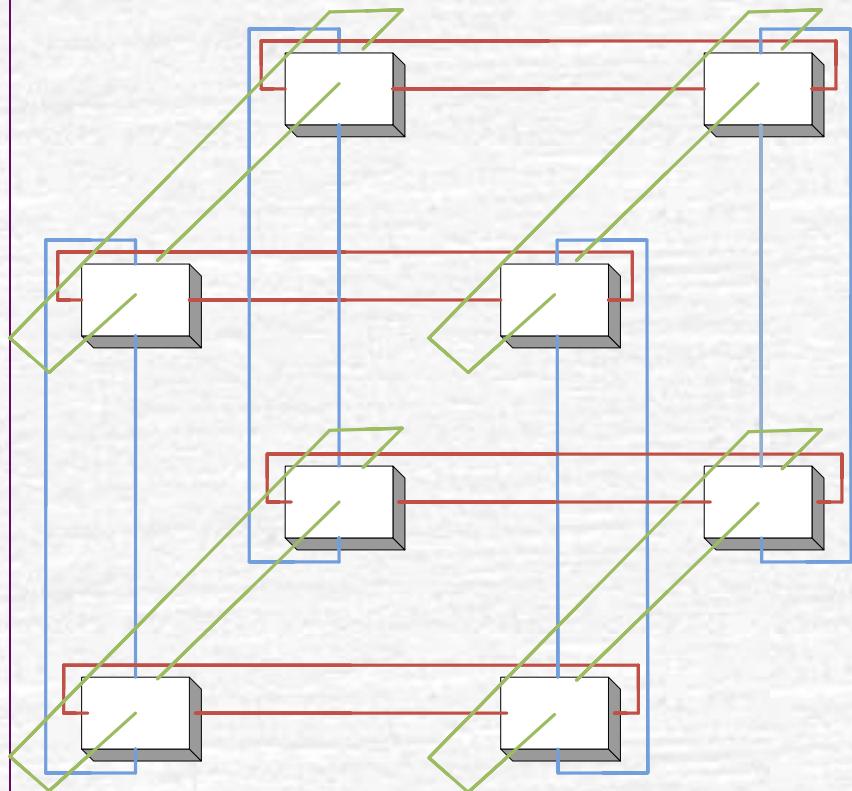
- Traditionally used for national labs and large companies
  - No standard architecture
  - BlueGene/L machine is modern supercomputer
- 
- BlueGene/L is configured:
    - 65536 dual core nodes
      - Each node is 440 PowerPC processor
    - Each node has a 32 KB L1 instruction and data
    - Each node has a 4MB L3 cache

# Logical Organization of a BlueGene/L node

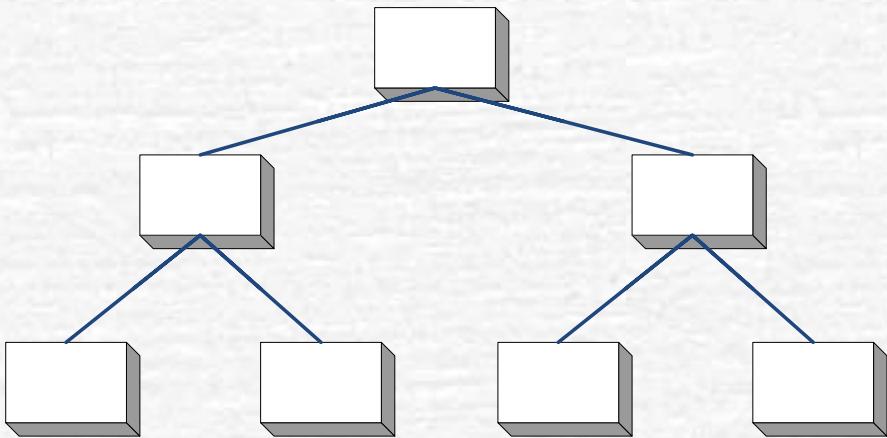


# BlueGene/L Communication Networks

- 3-D torus for data transfert



- Collective network

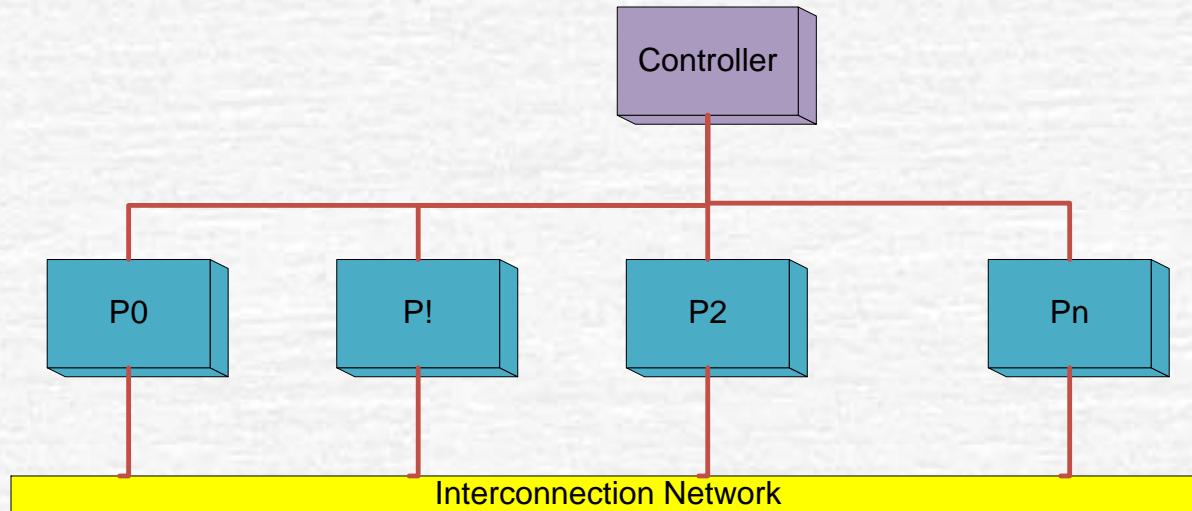


# PRAM: a Parallel Computer Model

- ☛ **Parallel Random Access Machine model**
- ☛ **Unspecified number of instruction execution units**
- ☛ **All execution units are connected to a single shared memory**
  - All reference the global memory
  - All observe a single sequence of memory state changes
- ☛ **Does not work well as a model for programmers.**
  - It fails by misrepresenting memory behavior

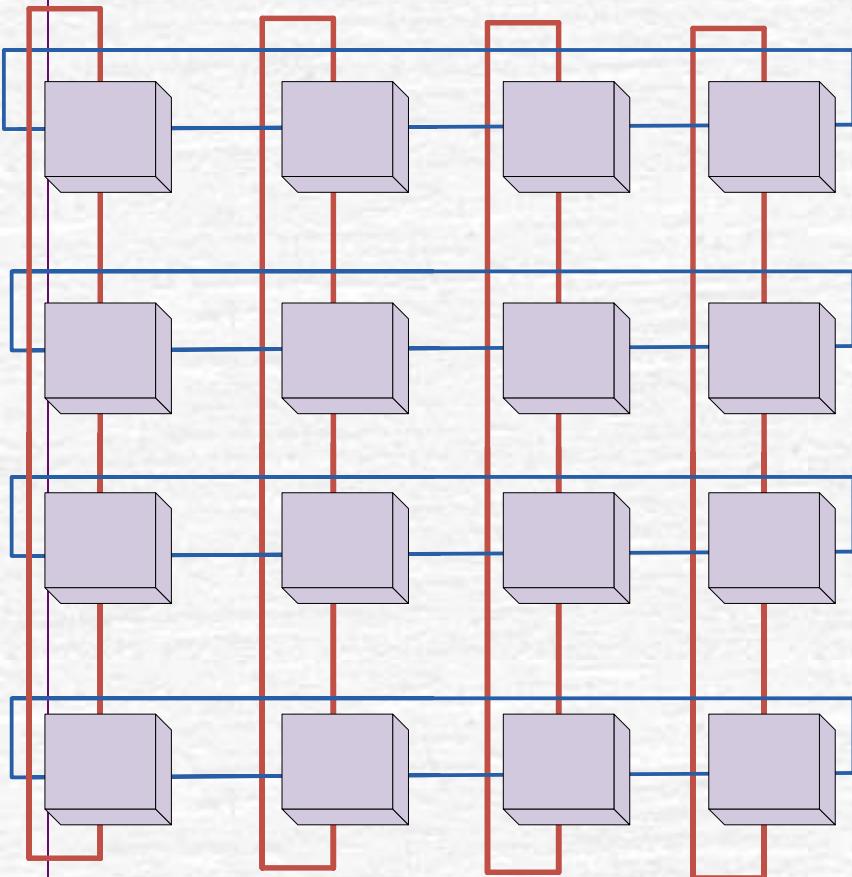
# CTA: a Practical Parallel Computer Model

- ✓ Addresses the shortcoming of the PRAM
  - ✓ CTA: Candidate Type Architecture
- 
- ✓ Two types of memory
    - Non expensive local references
    - Expensive non-local memory references

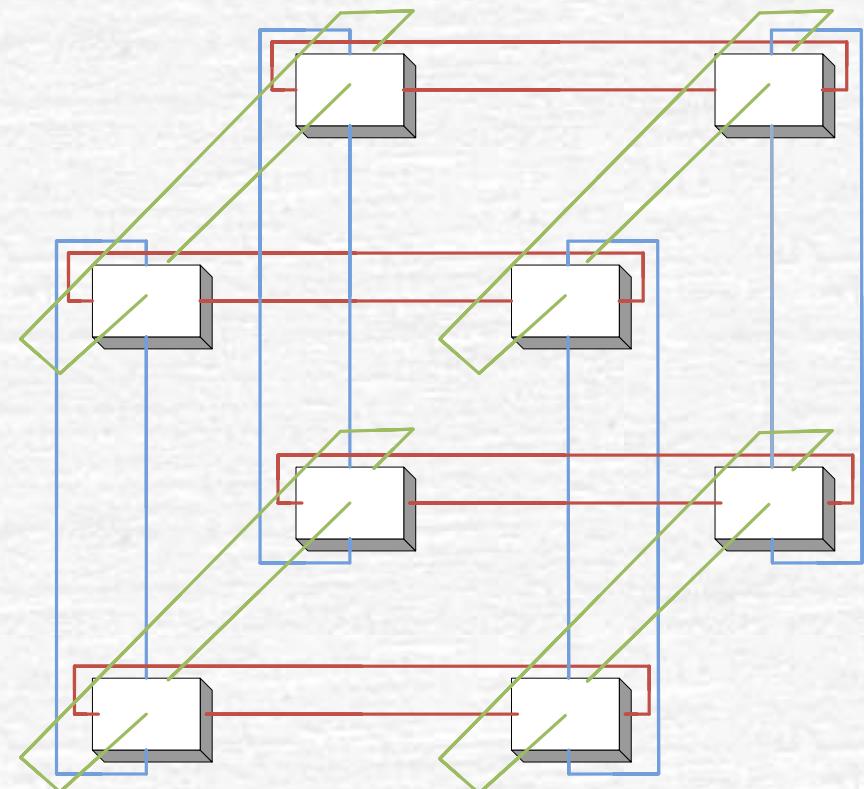


# Communication Topologies

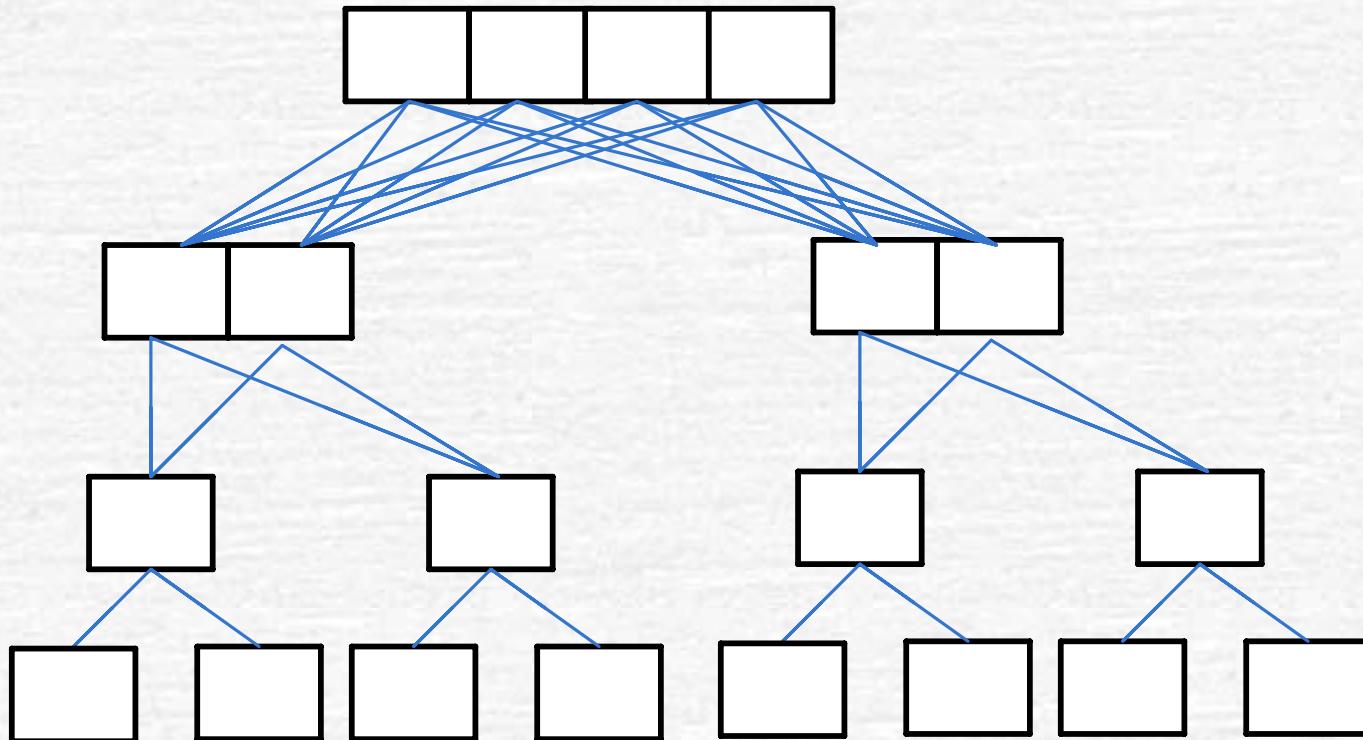
## 2-D Torus



## Binary 3-Cube



# Fat Tree



**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 1 – Fall 2021**

## Question 1

1. Give the definition of Parallel computing and Parallel Programming

Parallel computing: is a form of computation in which many calculations are carried out simultaneously.

Parallel Programming: Decomposing a programming problem into tasks and Deploy the tasks on multiple processors and run them simultaneously

2. Enumerate and give a brief description of the main opportunities of parallelism.

3. Enumerate and give a brief description of the main aspects of parallel computing.

4. What are the main differences between Distributed and Parallel Computing.

Q2:

1-Instruction Level Parallelism:

Hidden Parallelism in computer programs

2-Single computer level:

Multi core computers: Chip multi processors

Dual core, Quad core, GP GPU

Multi processor computers: Symmetric multi processors

Supercomputers

3-Multiple computers level:

Clusters(Known Configs at compile time), Servers, Grid computing(Dynamic can change at run time)

Q3

1-Parallel Computers Architecture

2-Algorithms and applications

Reasoning about performance

Designing parallel algorithms.

3-Parallel Programming

Paradigms

Programming Models

Programming languages

Frameworks

Dedicated environments

Q4

1- in distributed computing the processors are geographic distant but in parallel computing they are in the same machine

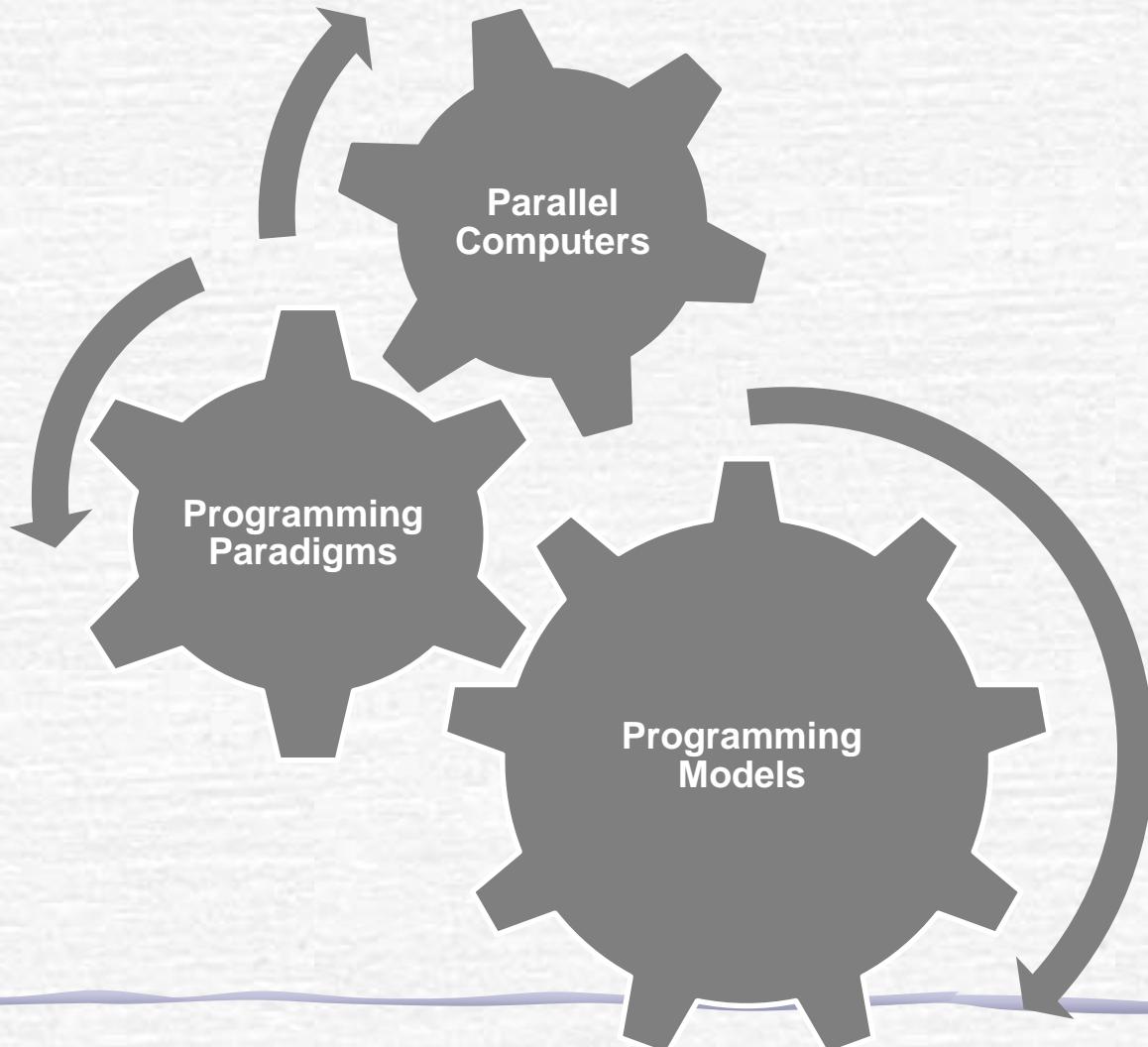
2- in distributed computing the aim is to make services available and reliable but in parallel computing the aim is to increase the performance

3- in parallel computing interaction is Fine grained with low overhead but in distributed computing interaction is Coarse grained and heavier weight

# Understanding Parallel Computing Paradigms and Programming Models

Sofien GANNOUNI  
Computer Science  
E-mail: [gnnosf@ksu.edu.sa](mailto:gnnosf@ksu.edu.sa) ; gansof@yahoo.com

# Introduction

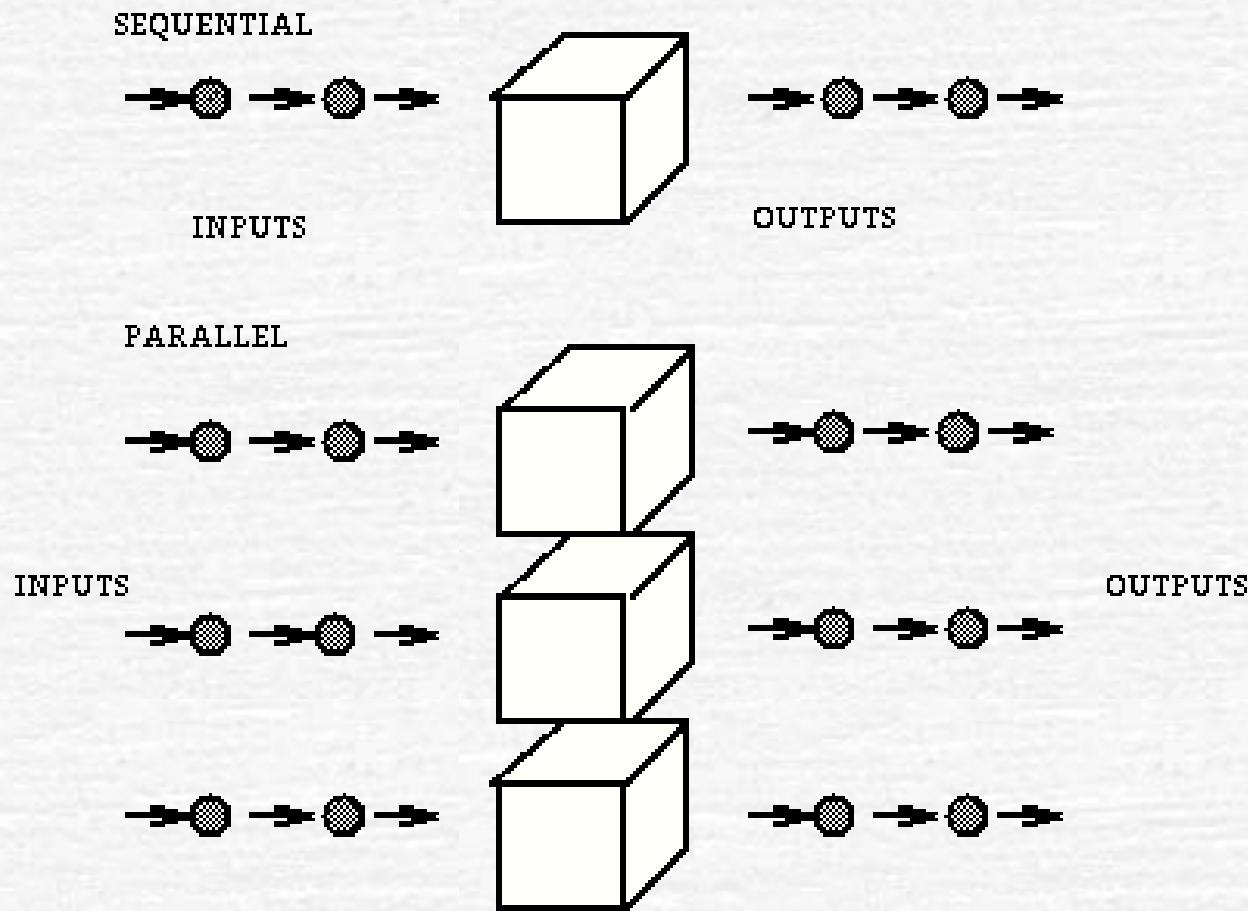


# Classification of Architectures – Flynn's classification

- Single Instruction Single Data (**SISD**): Serial Computers
- Single Instruction Multiple Data (**SIMD**)
  - Processor arrays
- Multiple Instruction Single Data (**MISD**): Not popular
- Multiple Instruction Multiple Data (**MIMD**)
  - Most popular
  - most supercomputers, clusters, computational Grids etc

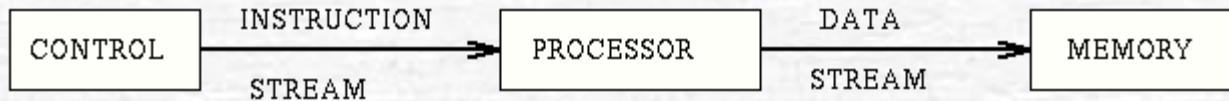
Single Instruction	Single Data	Multiple Data
<b>SISD</b>		<b>SIMD</b>
<b>MISD</b>		<b>MIMD</b>

# Sequential vs. Parallel Processing



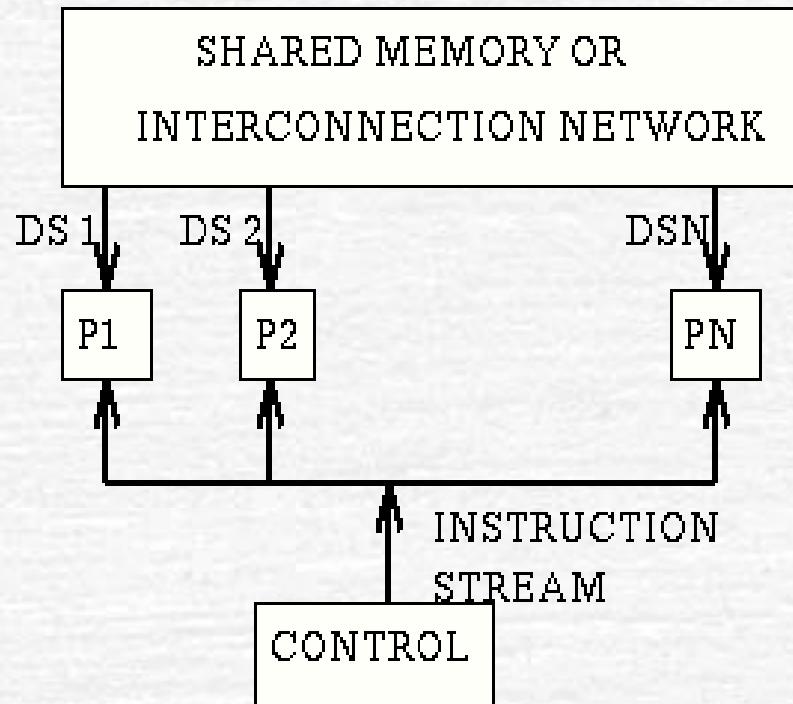
# SISD Computers

- ☞ a stream of instructions (the algorithm) tells the computer what to do.
- ☞ a stream of data (the input) is affected by these instructions.



# SIMD Computers

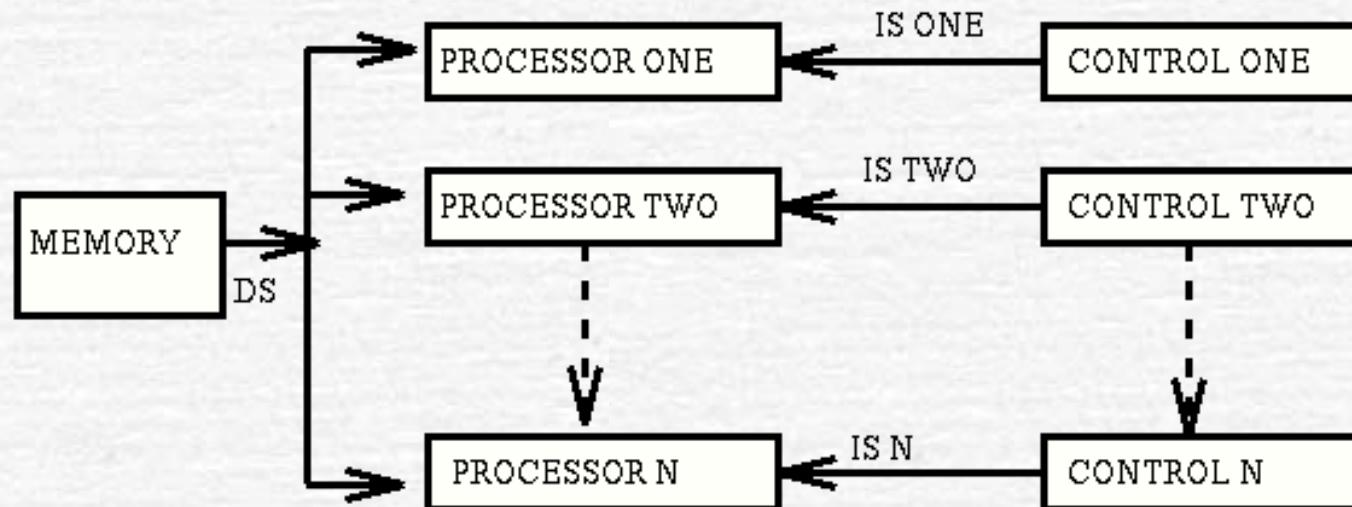
- A single instruction stream is broadcasted to multiple processors, each having its own data stream



P = PROCESSOR

DS = DATA STREAM

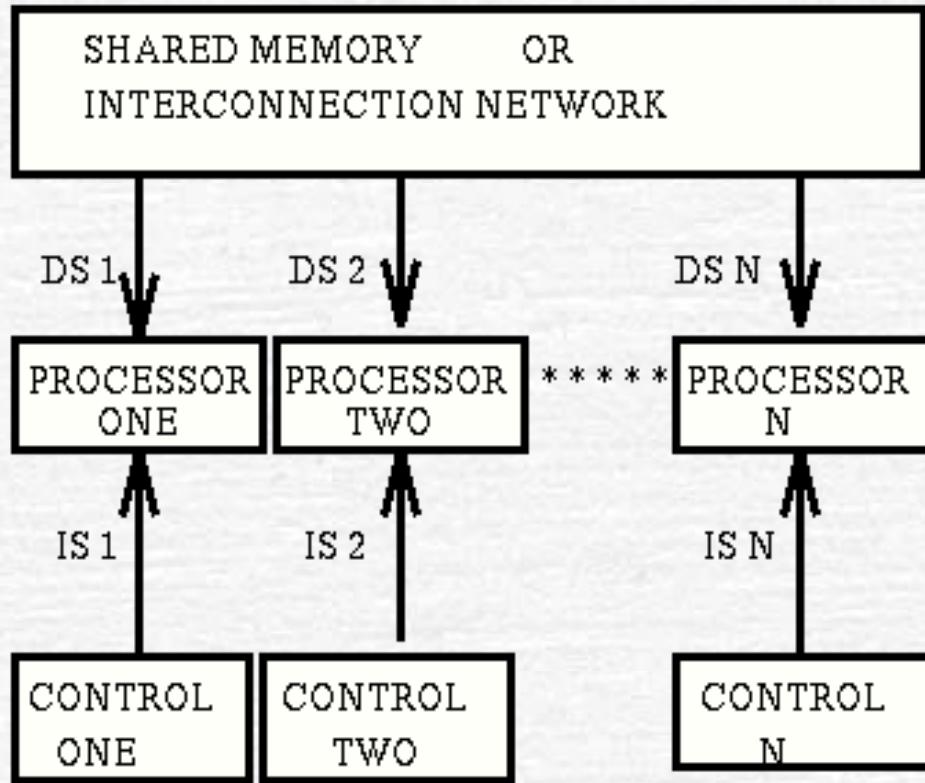
# MISD Computers



IS = INSTRUCTION STREAM

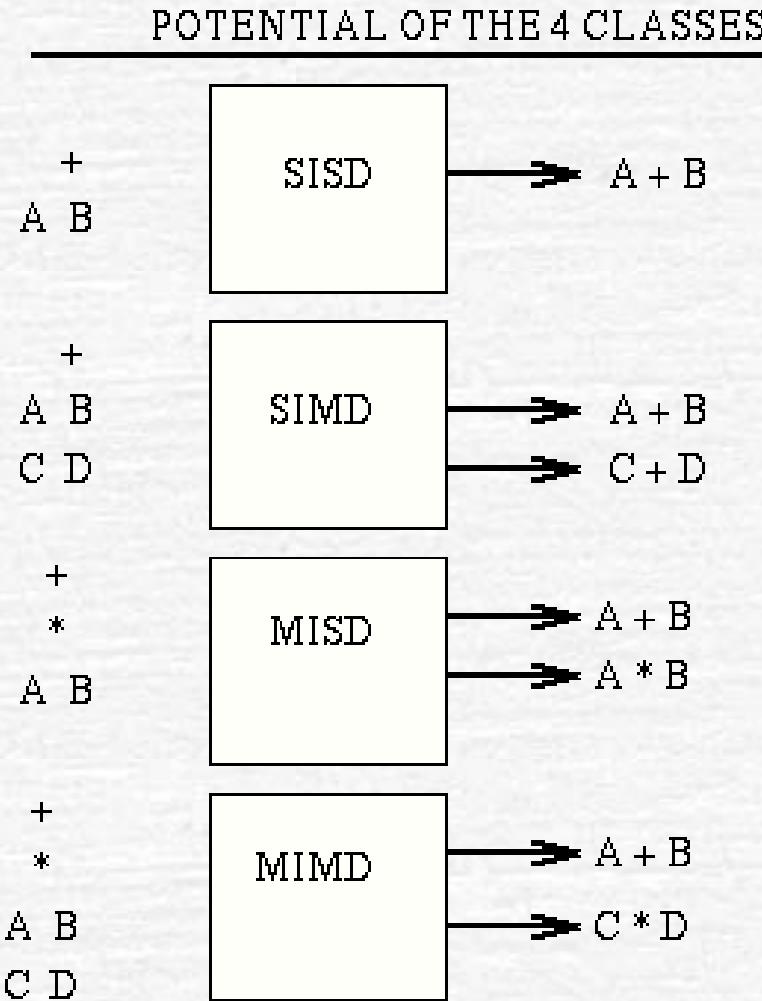
DS = DATA STREAM

# MIMD (multiprocessors / multiccomputers)



DS = DATA STREAM    IS = INSTRUCTION STREAM

# Potential of the 4 Classes



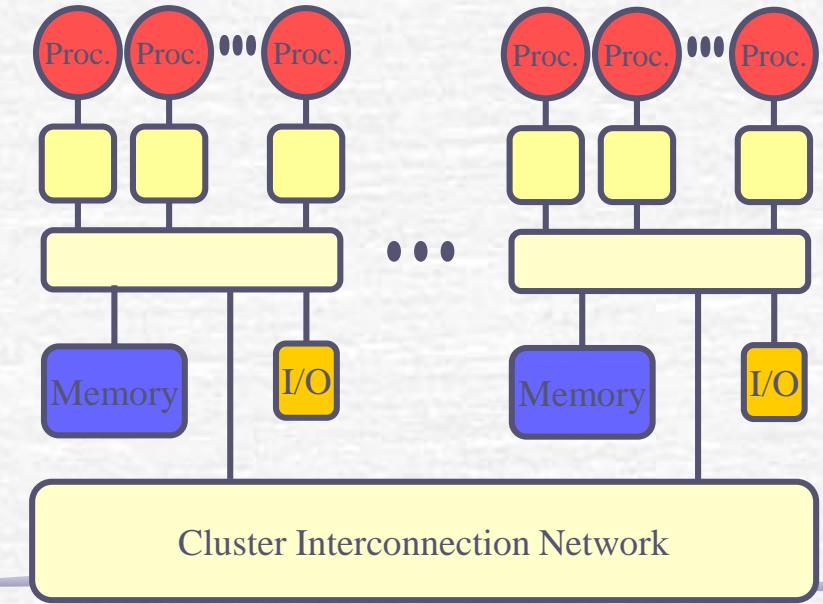
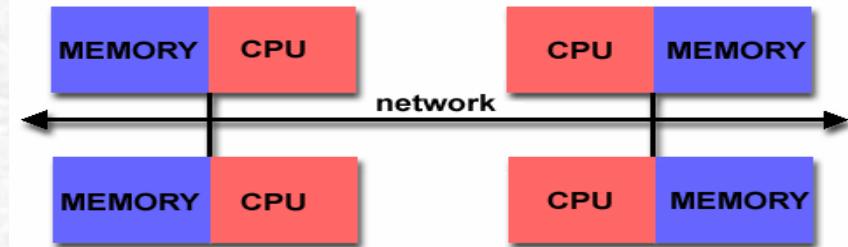
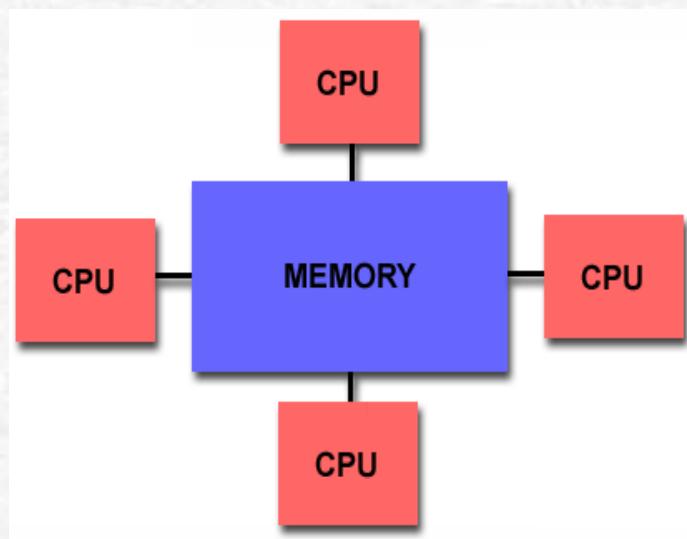


# Parallel Computing Paradigms

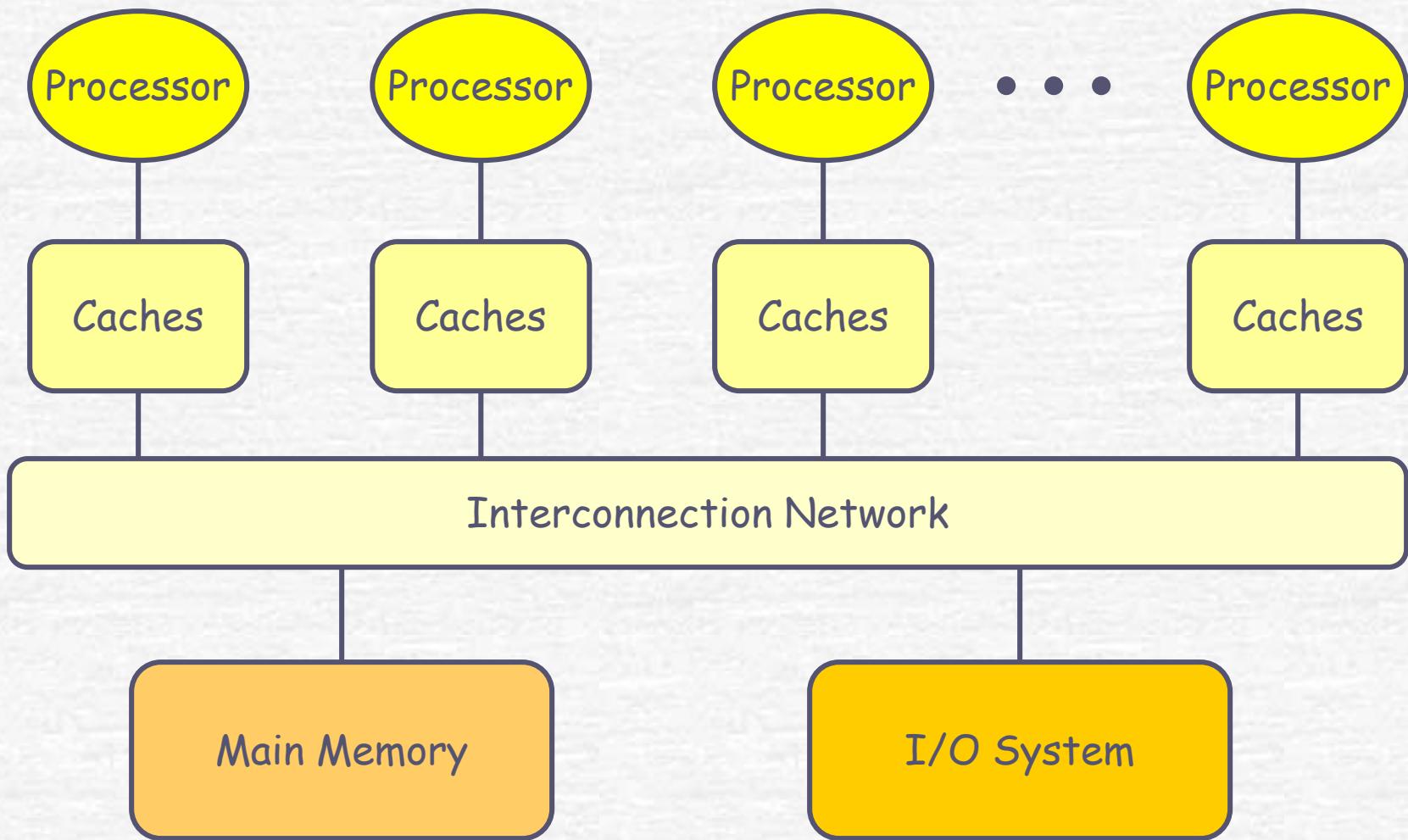
- **Shared Memory Paradigm**
  - **Message Passing**
  - **Multi-threading**
- 

# Shared Memory Paradigm

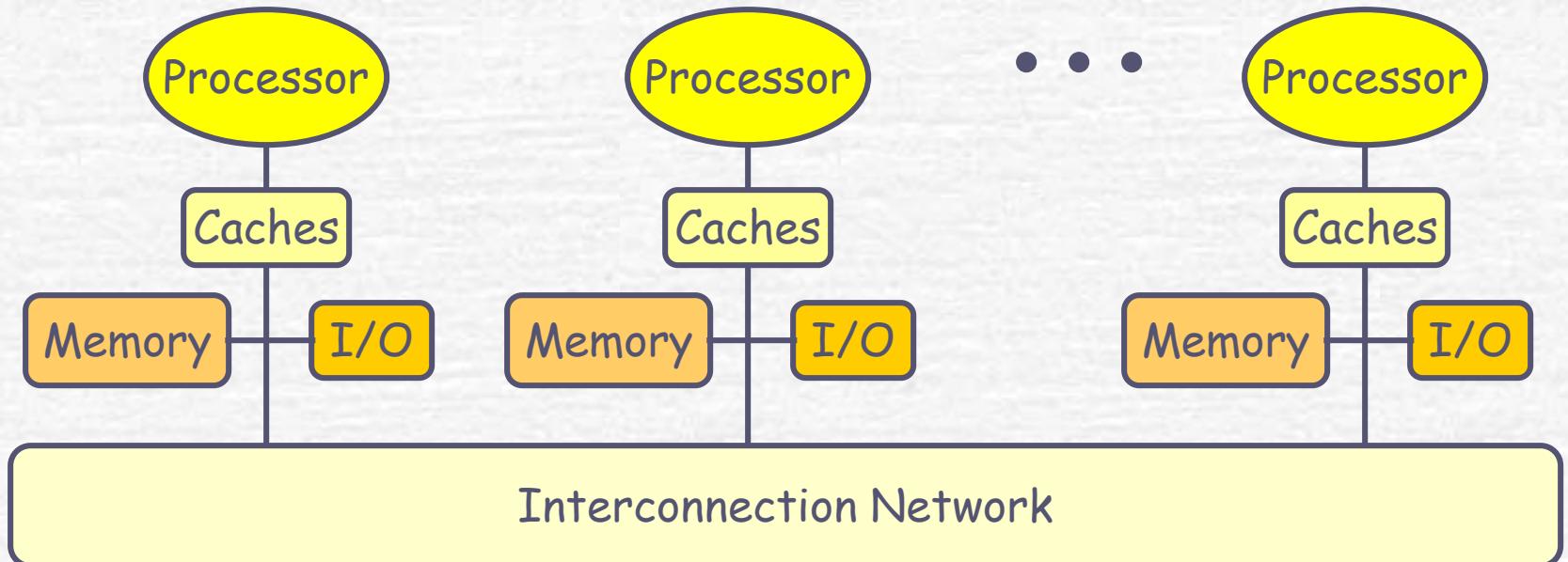
- ✓ Shared memory
- ✓ Distributed memory
- ✓ Hybrid systems



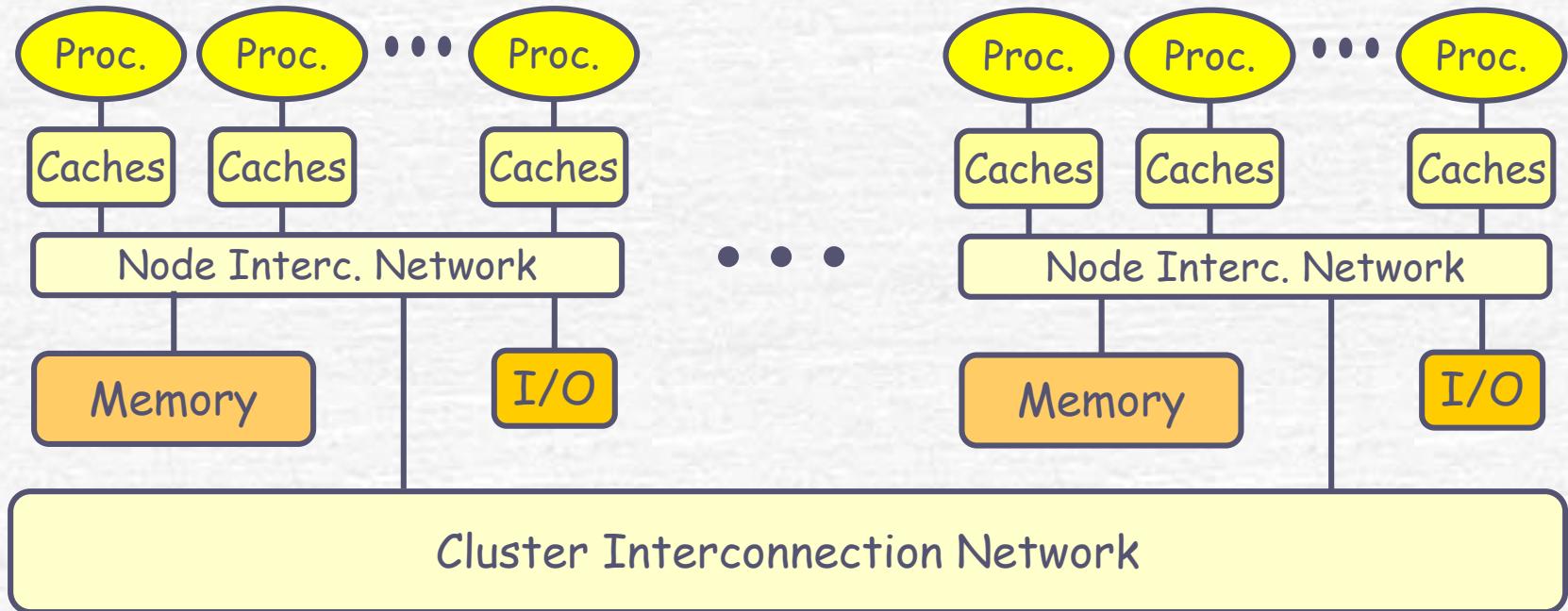
# Centralized Shared Memory



# Distributed Memory Architecture



# Hybrid Shared Memory



# Message Passing Paradigm

- Allows for communication between a set of processors.
- Each processor is required to have a local memory, no global memory is required.
- The whole address space of the system consists of multiple private address spaces.
- Communication occurs between processors by sending and receiving messages.

# Basic Operations

## Blocking non-buffered send/ receive

- the sender issues a *send* operation and cannot proceed until a matching receive at the receiver's side is encountered and the operation is complete.

## Blocking buffered send/ receive

- when the sender process reaches a *send* operation it copies the data into the buffer on its side and can proceed without waiting.
- At the receiver side it is not necessary that the received data will be stored directly at the designated location.
- When the receiver process encounters a receive operations it checks the buffer for data.

# Basic Operations

## Non-Blocking non-buffered send/ receive

- the process needs not to be idle but instead can do useful computations while waiting for the send / receive operation to complete.

## Non-Blocking buffered send/ receive

- the sender issues a direct memory access operation (DMA) to the buffer. DMA operations can be carried out without processor being involved.
- The sender can proceed with its computations.
- At the receiver side, when a receive operation is encountered the data is transferred from the buffer to memory location.

## **Broadcast**

- This function allows one process (called the root) to send the same data to all communicator members

## **Scatter**

- Allows one process to give out the content of its send buffer to all processes in a communicator.

## **Gather**

- Each process gives out the data in its send buffer to the root process which stores them according to their ranks.

# Multi-threading Paradigm

## ↗ In a single-core (superscalar) system,

- we can define multithreading as the ability of the processor's hardware to run two or more threads in an overlapping fashion by allowing them to share the functional units of that processor.

## ↗ in a multi-core system,

- we can define multithreading as the ability of two or more processors to run two or more threads simultaneously (in parallel) where each thread run on a separate processor

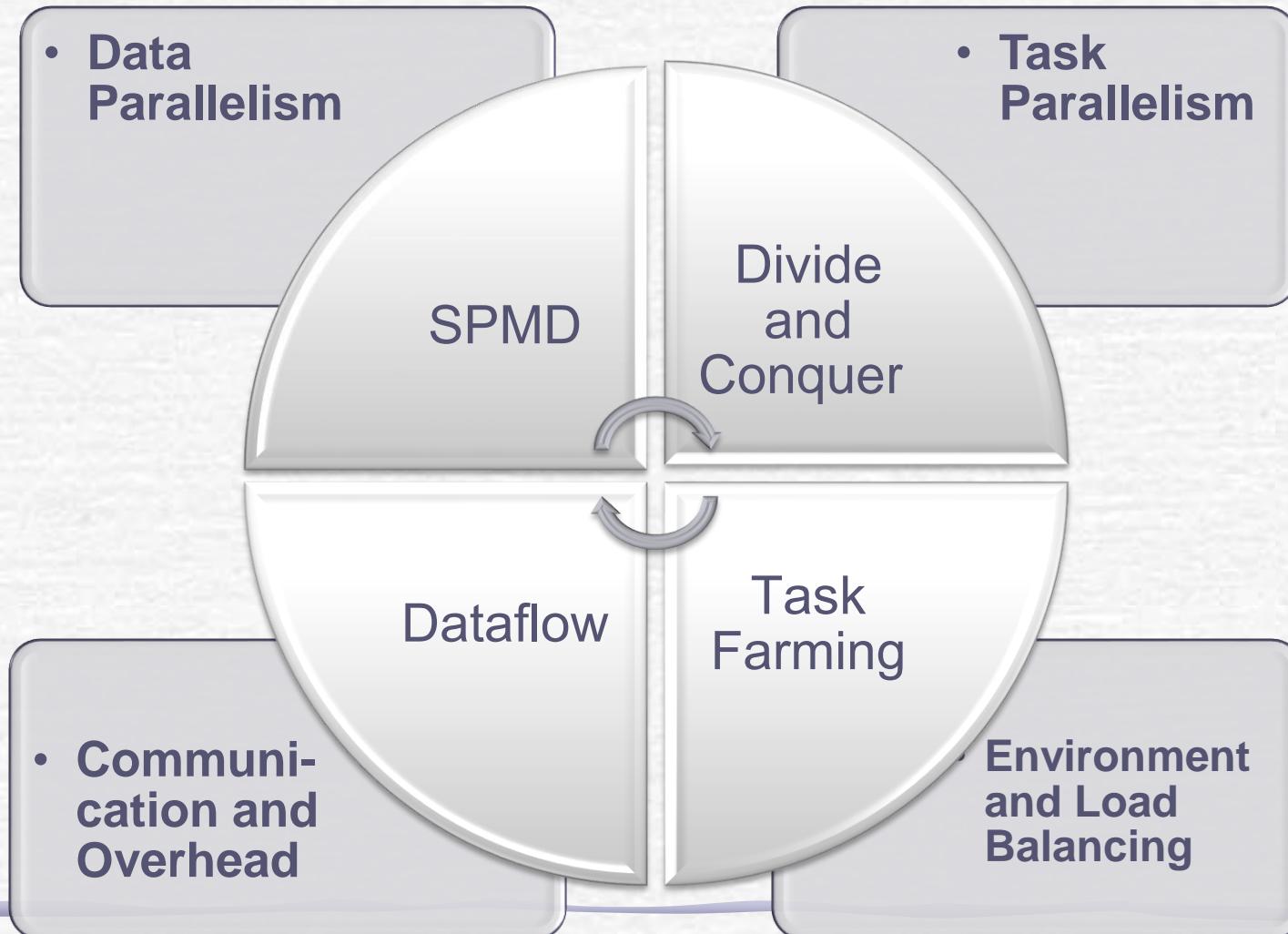
## ↗ Modern systems combine both multithreading approaches.



# Parallel Programming Models

- **SPMD**
  - **Task Farming**
  - **Divide and Conquer**
  - **Dataflow**
- 

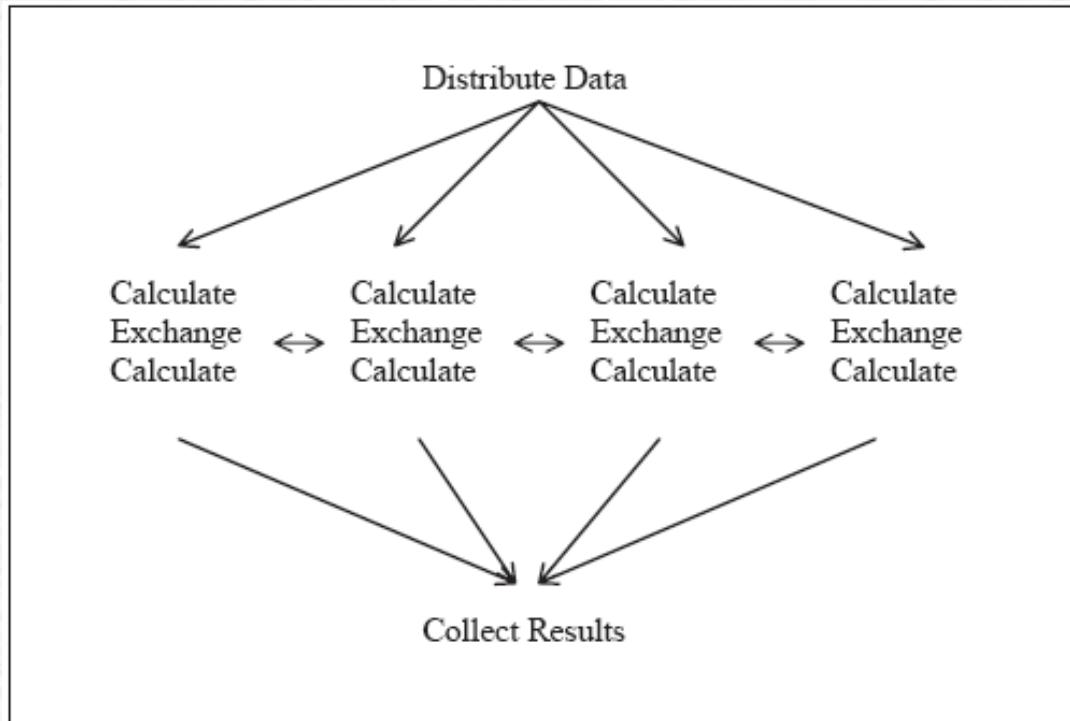
# Models and Design Considerations



# Single Program Multiple Data (SPMD)

- SPMD model is the dominant pattern to structure parallel programs.
- A single program is loaded into each node of a parallel system.
- Nodes are executing the code independently but act on multiple data sets including "private" and "shared" data.
- Nodes cooperating in the execution of the program are assigned unique IDs, allowed to self-schedule themselves, and dynamically get assigned to the required tasks under this cooperative execution.

# SPMD: The Concept

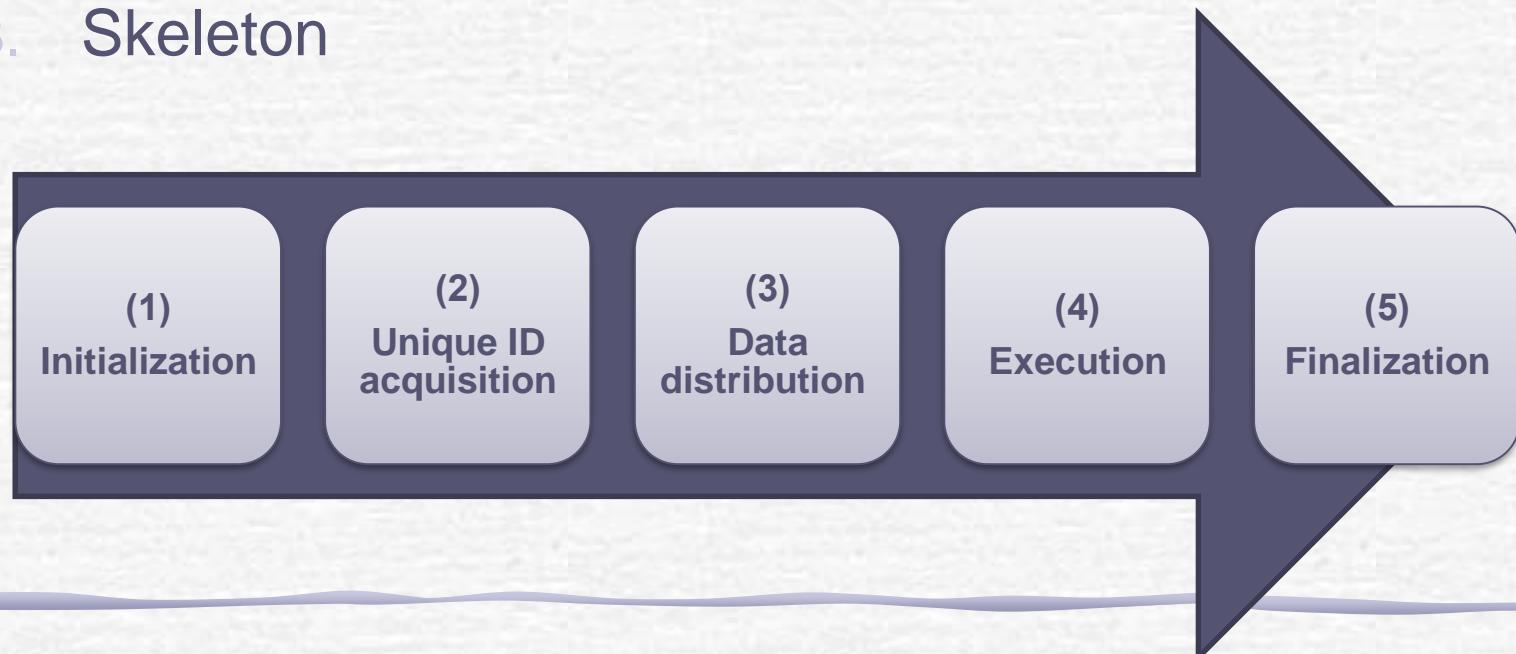


Basic structure of SPMD program

# SPMD-Style Program Components

Program components:

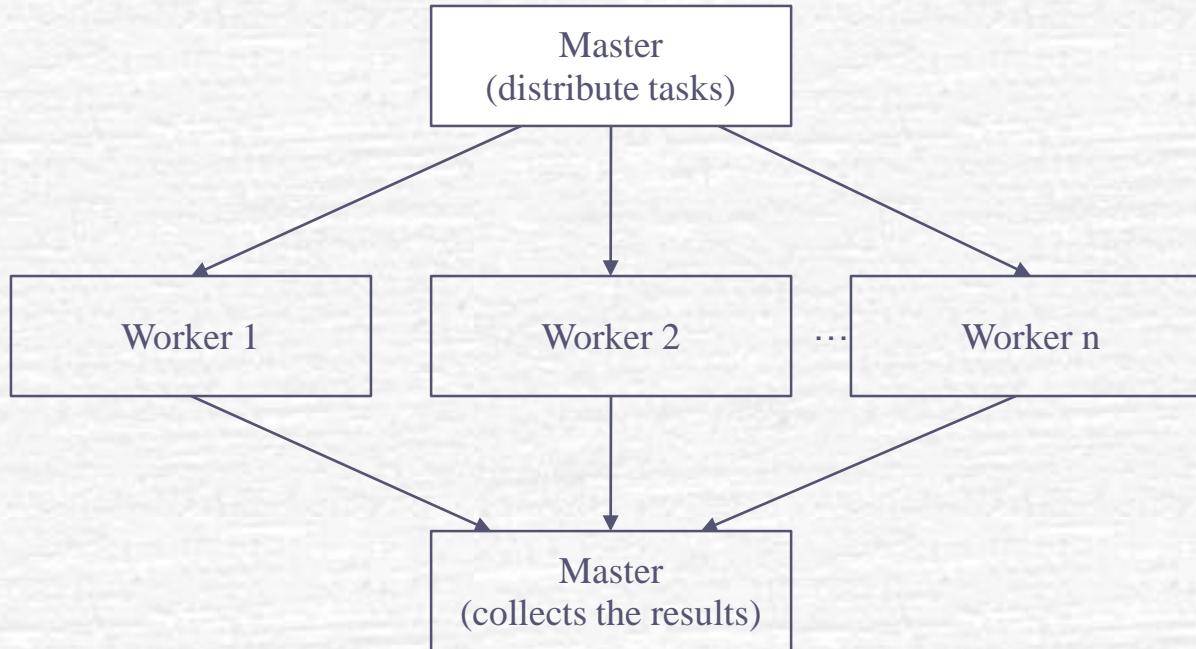
1. Source code
2. Load Balancing Strategy
3. Skeleton



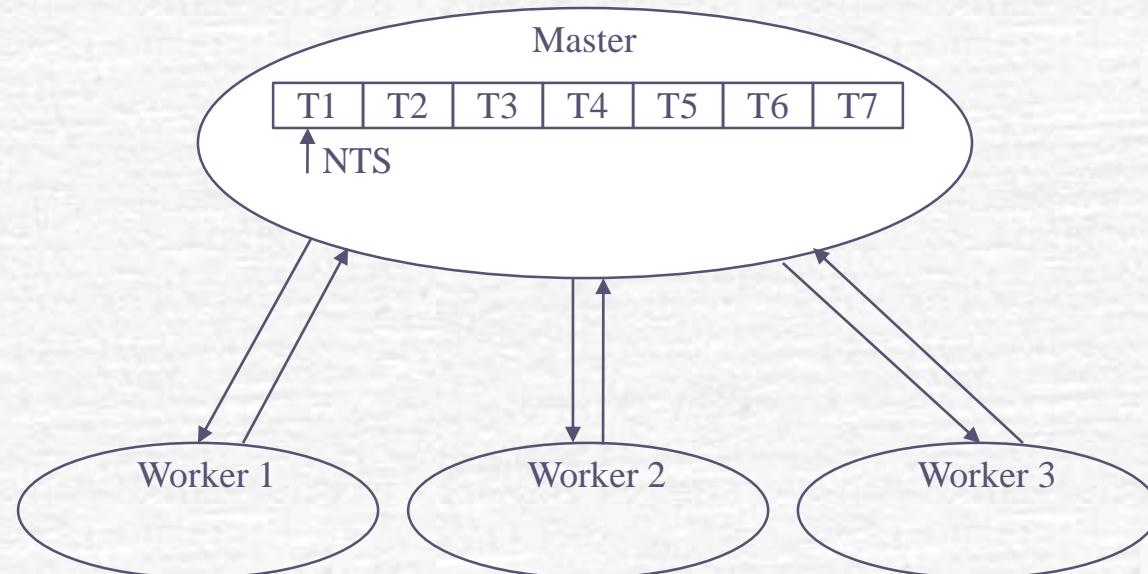
# Task Farming Model

- Task farming (or Master-Worker or Master-Slave or Manager-Worker) consists of two entities:
  - the master
  - and many workers.
- The master:
  - decomposes the problem into small tasks,
  - then distributes/farms these tasks among a farm of workers
  - and finally collects the partial results to produce the final result of the computation.
- The worker:
  - receives a task, process it and send the result back to the master.

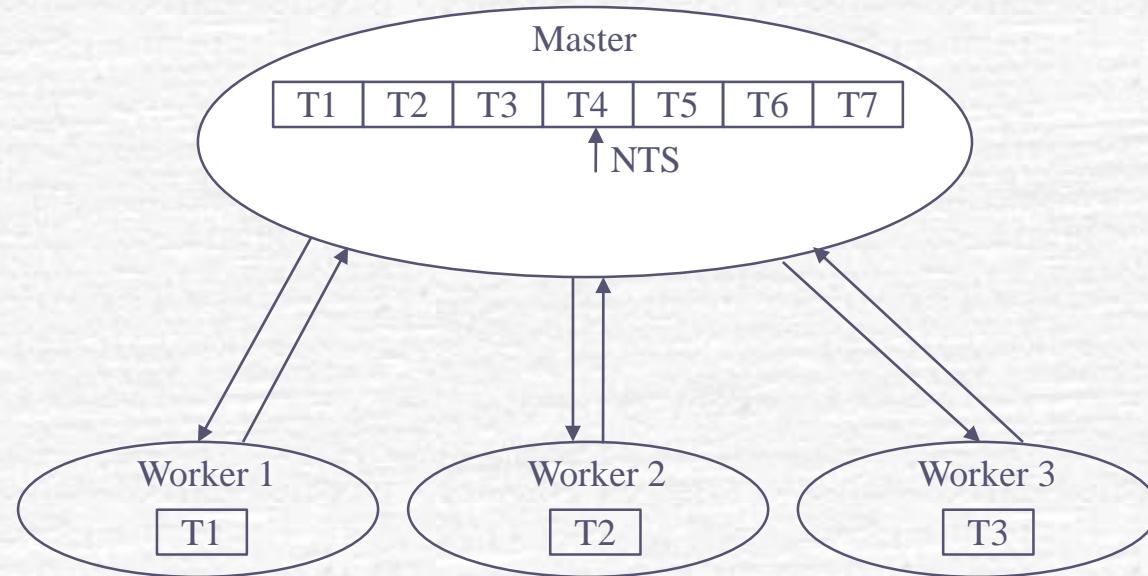
# Basic Task Farming Structure



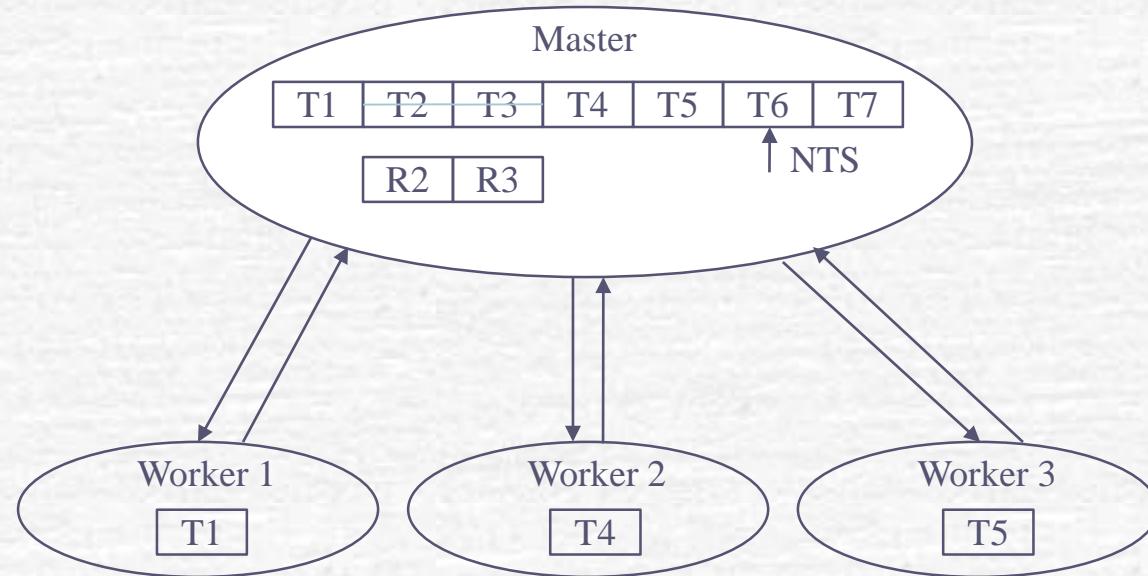
# Simple Task Farming Example (a)



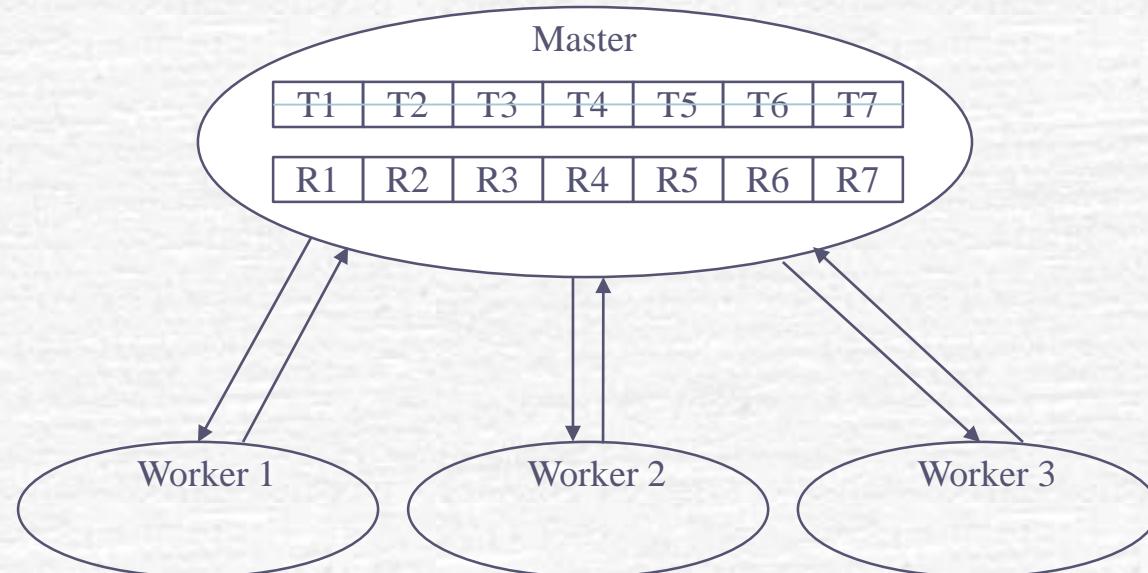
# Simple Task Farming Example (b)



# Simple Task Farming Example (c)



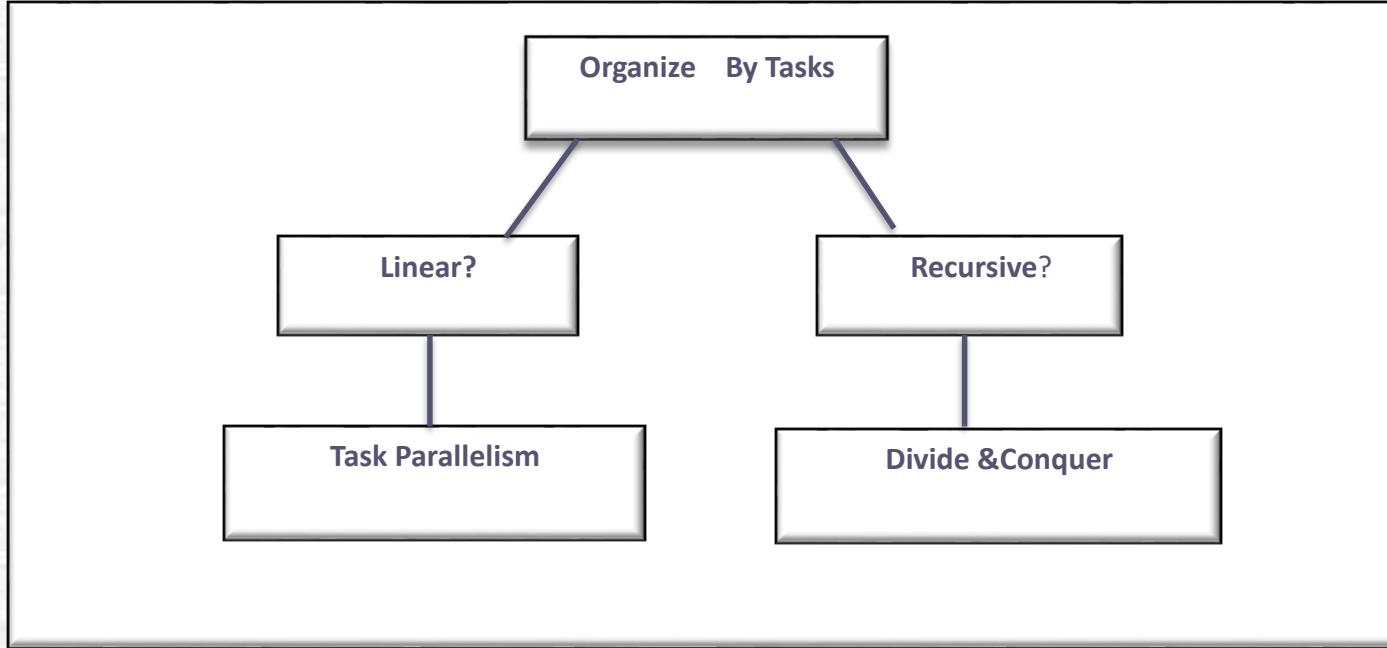
# Simple Task Farming Example (d)



# Divide and Conquer Model

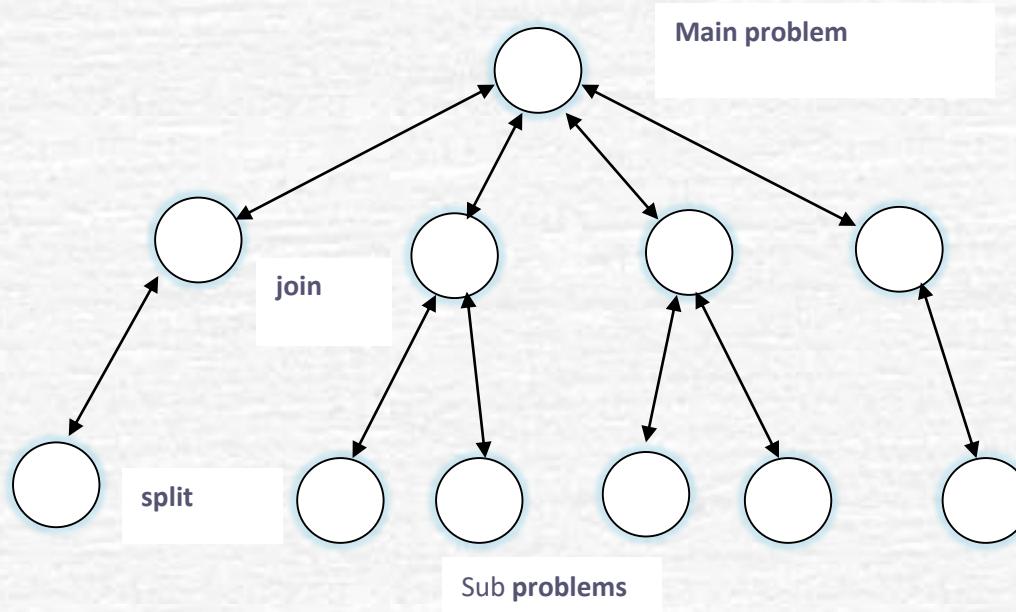
- This model solves a complex problem by splitting it into smaller easier sub-problems.
- The sub-problems have the same nature as the original one and could be solved by recursively applying the same algorithm.
- The recursion stops at the base case in which the sub-problem is solved directly.
- The results of all sub-problems are then joined to produce the final overall solution.
- The sub-problems are usually solved independently and concurrently

# Task Organization



☞ The key point in Divide and Conquer is that the same task is recursively performed on different data.

# Task Decomposition

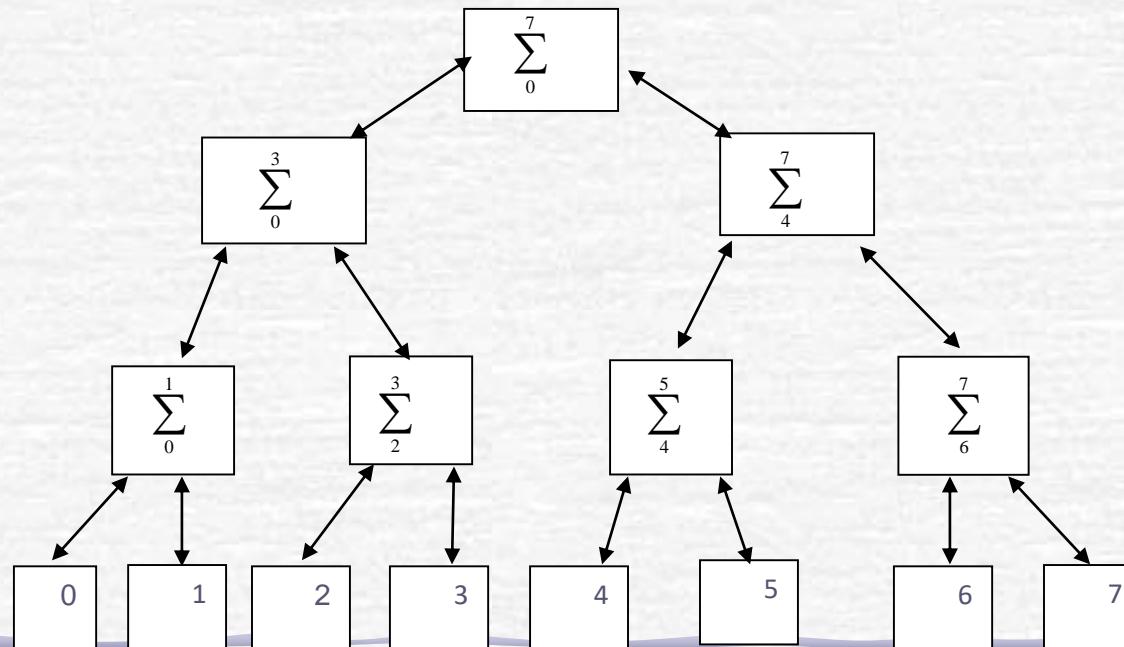


**Decomposition of Tasks is dynamic because not all tasks are known in advance.**

# Parallel Divide and Conquer Example

- Problem : Summing N numbers .
- Solution : Divide into two subproblems each of size N/2

Using the feature  $\sum_{i=0}^{2^n-1} = \sum_{i=0}^{2^{n-1}-1} + \sum_{i=2^{n-1}}^{2^n-1}$



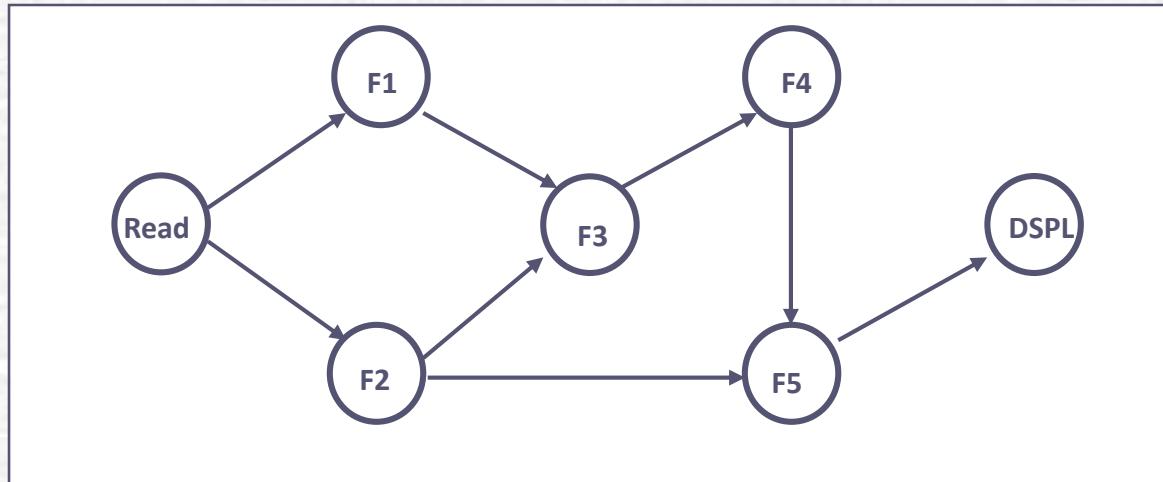
# Dataflow

- ☞ The main concept of dataflow programming is to divide the computation problem into multiple disjoint functional blocks.
- ☞ Each block solves part of the problem.
- ☞ Blocks are connected to each other to :
  - show the dependency between them.
  - express *the logical execution flow*, and they can be used to easily express parallelism.
- ☞ Data-pipelining is a specialized case of dataflow model.

# Principles of Dataflow Model

- In dataflow model, computation is modeled as a directed graph.

- Nodes: functional blocks,  
Arcs: Dependency,  
Tokens: Data.



*There may be many nodes that are ready to fire (execute) at a given time.*

**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 2 – Spring 2021**

## Question

1. Give the flynn's classification of computers.  
SISD , SIMD , MISD , MIMD
2. Use an example to explain the differences between the **SIMD** and **MIMD** computers.
3. Explain the main differences between the **Blocking non-buffered** and the **Non-Blocking non-buffered** send/receive operations of the message passing paradigm.
4. Let's consider that a root process has N child processes. Let's consider that the root process has an array called **Data** of size N. Explain the following operations using the array Data.
  - a. The root executes the operation **broadcast** of the message passing paradigm.
  - b. The root executes the operation **scatter** of the message passing paradigm.
  - c. The root executes the operation **gather** of the message passing paradigm.
5. Describe the **Task Farming** and the **Divide-and-Conquer** programming models and explain the main differences between them.

Q2:

SIMD: Every proccsor has same instruction stream but each proccsor has it's own data stream

MIMD: Every proccsor has it's own instruction stream and data stream

Q3:

Blocking non-buffered: The sender when he send a send operation he will be blocked until the reciver match a recive operation

Non-Blockiing non-bufferd: The sender sends a send operation and then he will continue proccsing until the reciver send an interuption signal

Q4:

- A- The Same Data will be copied to each proccses
- B- The Data will be split between the procces
- C- The Data will be gathered and joined in the root

Q5:

In divide and conquer the sub-tasks have the same nature and will be recursivly decomposed

In master-slave the sub-tasks may have diffrent nature

# CUDA Programming

# Outline

❑ GPU

❑ CUDA Introduction

    ❑ What is CUDA

    ❑ CUDA Programming Model

    ❑ Advantages & Limitations

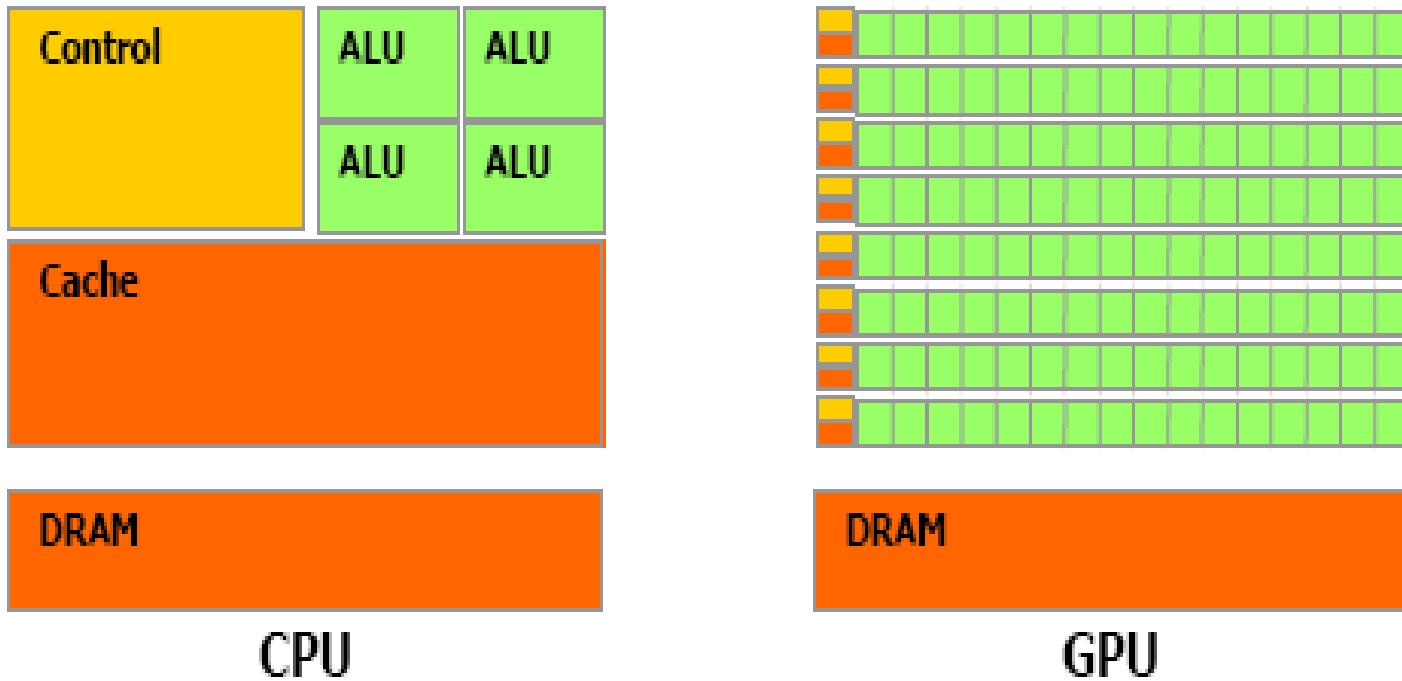
❑ CUDA Programming

❑ Future Work

# GPU

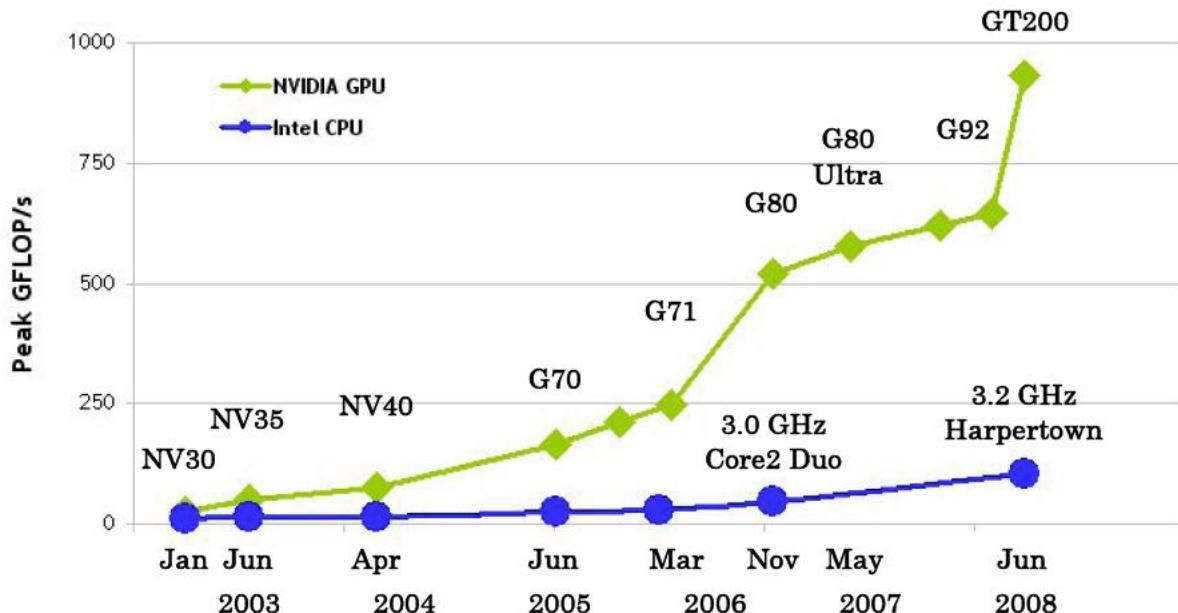
- ❑ GPUs are massively multithreaded many core chips
  - ❑ handle computation only for computer graphics
  - ❑ Hundreds of processors
  - ❑ Tens of thousands of concurrent threads
  - ❑ TFLOPs peak performance
  - ❑ Fine-grained data-parallel computation
- ❑ Users across science & engineering disciplines are achieving tenfold and higher speedups on GPU

# CPU v/s GPU



© NVIDIA Corporation 2009

# CPU v/s GPU



GT200 = GeForce GTX 280

G71 = GeForce 7900 GTX

NV35 = GeForce FX 5950 Ultra

G92 = GeForce 9800 GTX

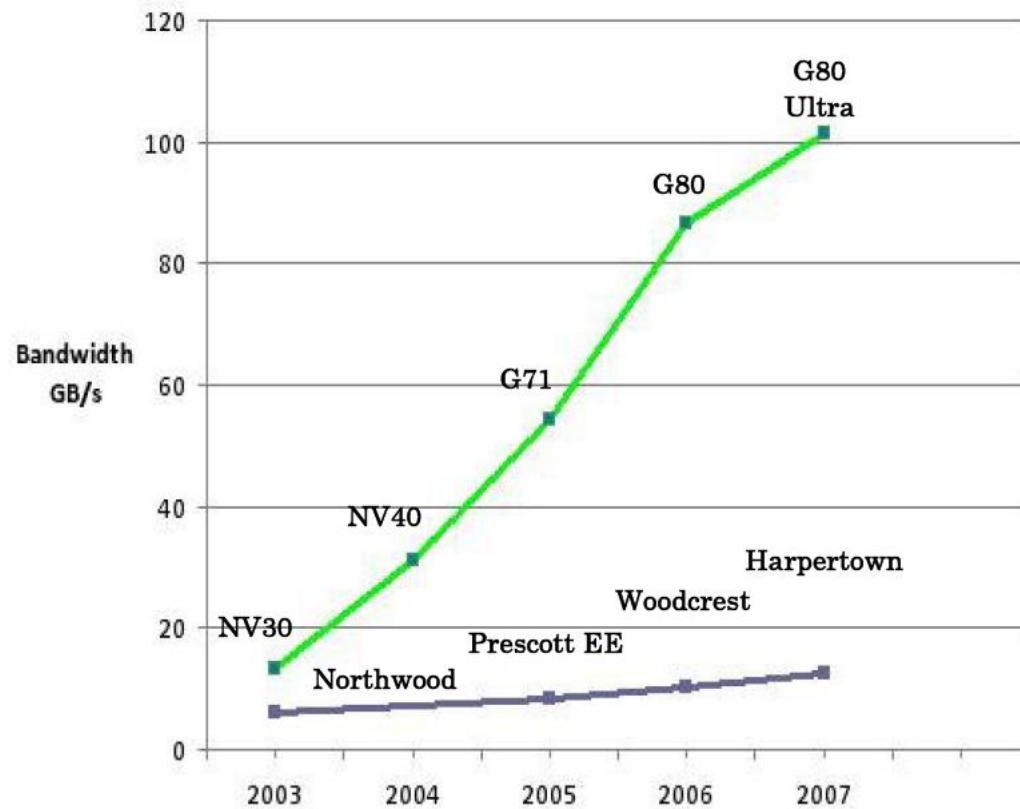
G70 = GeForce 7800 GTX

NV30 = GeForce FX 5800

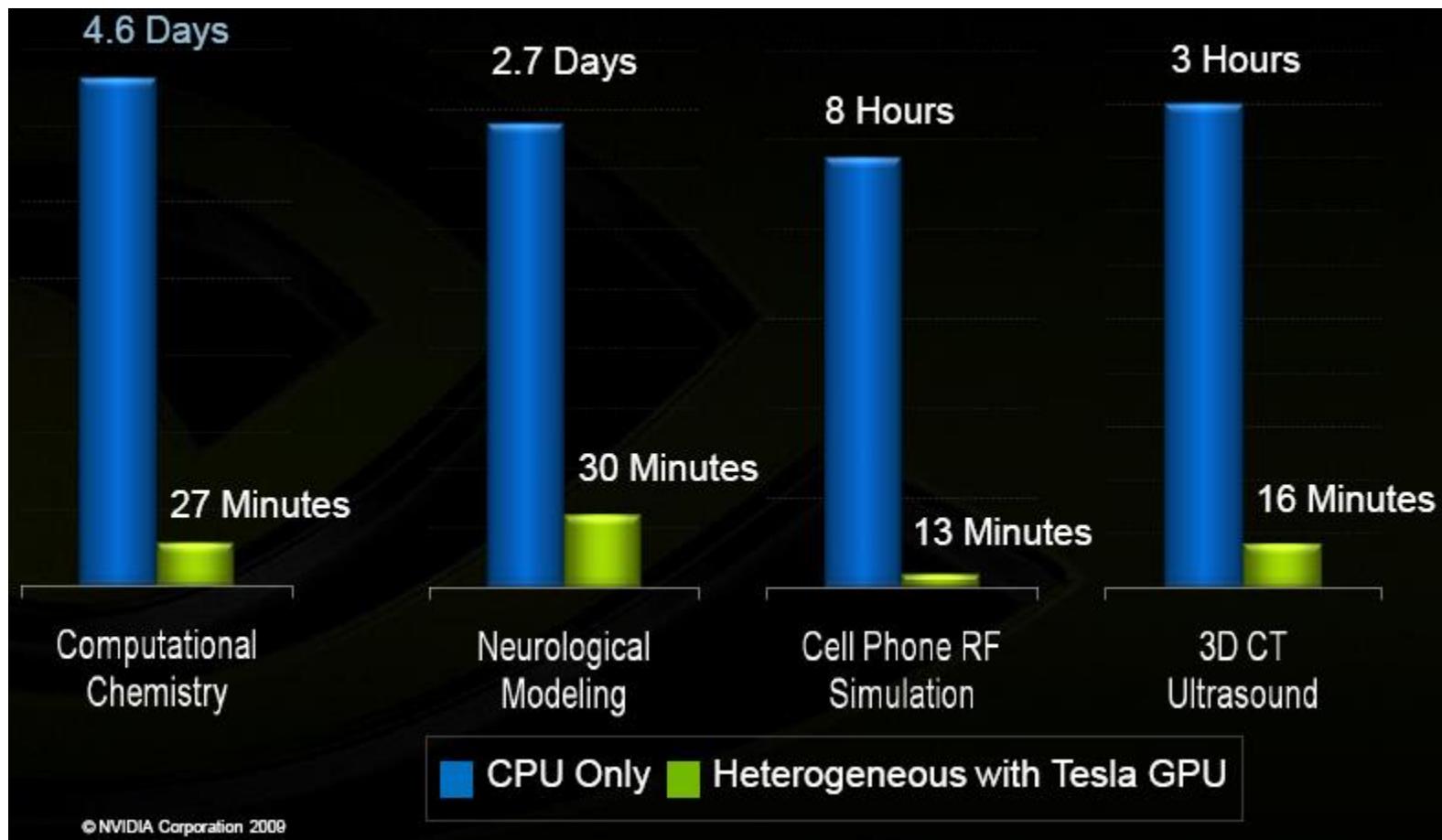
G80 = GeForce 8800 GTX

NV40 = GeForce 6800 Ultra

# CPU v/s GPU



# CPU v/s GPU



# GPGPU

- What is GPGPU?
  - General purpose computing on GPUs
  - **GPGPU** is the use of a **GPU**, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).
- Why GPGPU?
  - Massively parallel computing power
  - Inexpensive

# GPGPU

- How?
  - CUDA
  - OpenCL
  - DirectCompute

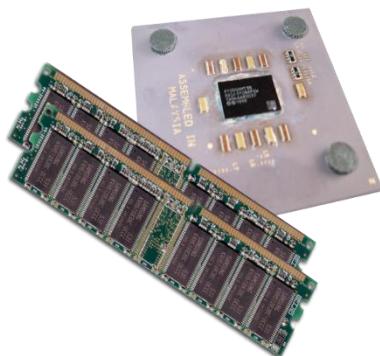
# What is CUDA?

- ❑ CUDA is the acronym for Compute Unified Device Architecture.
  - ❑ A parallel computing architecture developed by NVIDIA.
  - ❑ Heterogeneous serial-parallel computing
  - ❑ The computing engine in GPU.
  - ❑ CUDA can be accessible to software developers through industry standard programming languages.
- ❑ CUDA gives developers access to the instruction set and memory of the parallel computation elements in GPUs.

# Heterogeneous Computing

- Terminology:

- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)



Host



Device

# Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N      1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockDim.x * blockIdx.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;           // host copies of a, b, c
    int *d_in, *d_out;       // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

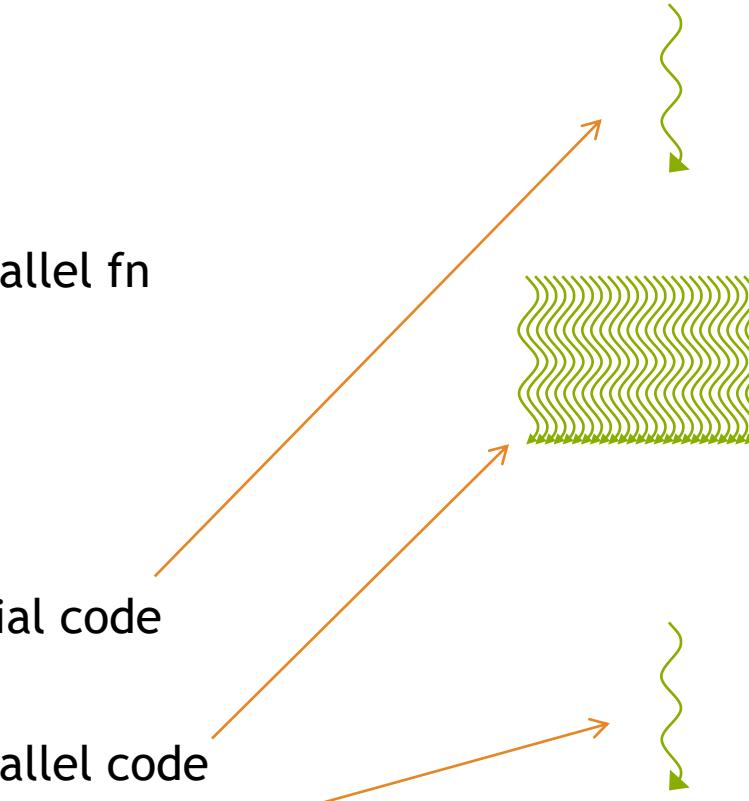
    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

serial code

parallel code  
serial code



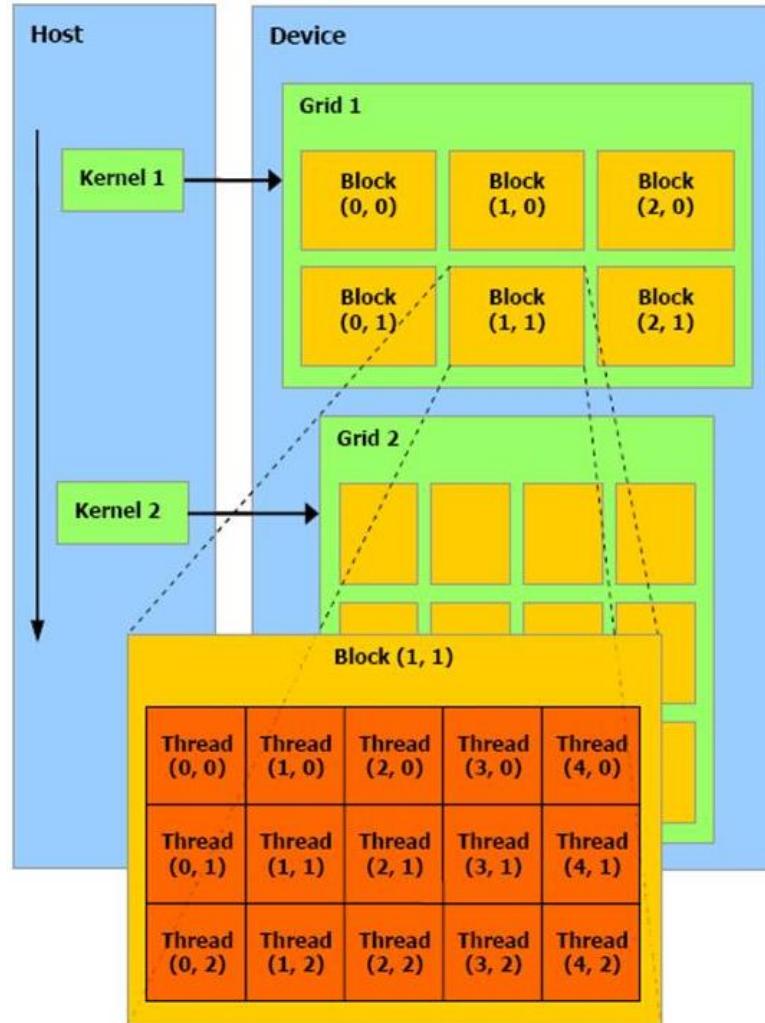
# CUDA Kernels and Threads

- ❑ Parallel portions of an application are executed on the device as kernels
- ❑ A Kernel is a Function that runs on a device
  - ❑ One kernel is executed at a time
  - ❑ Many threads execute each kernel
- ❑ Differences between CUDA and CPU threads
  - ❑ CUDA threads are extremely lightweight
    - ❑ Very little creation overhead
    - ❑ Instant switching

# CUDA Programming Model

- ❑ A kernel is executed by a grid of thread blocks
- ❑ A thread block is a batch of threads that can cooperate with each other by:
  - ❑ Sharing data through shared memory
  - ❑ Synchronizing their execution
- ❑ Threads from different blocks cannot cooperate

# CUDA Programming Model



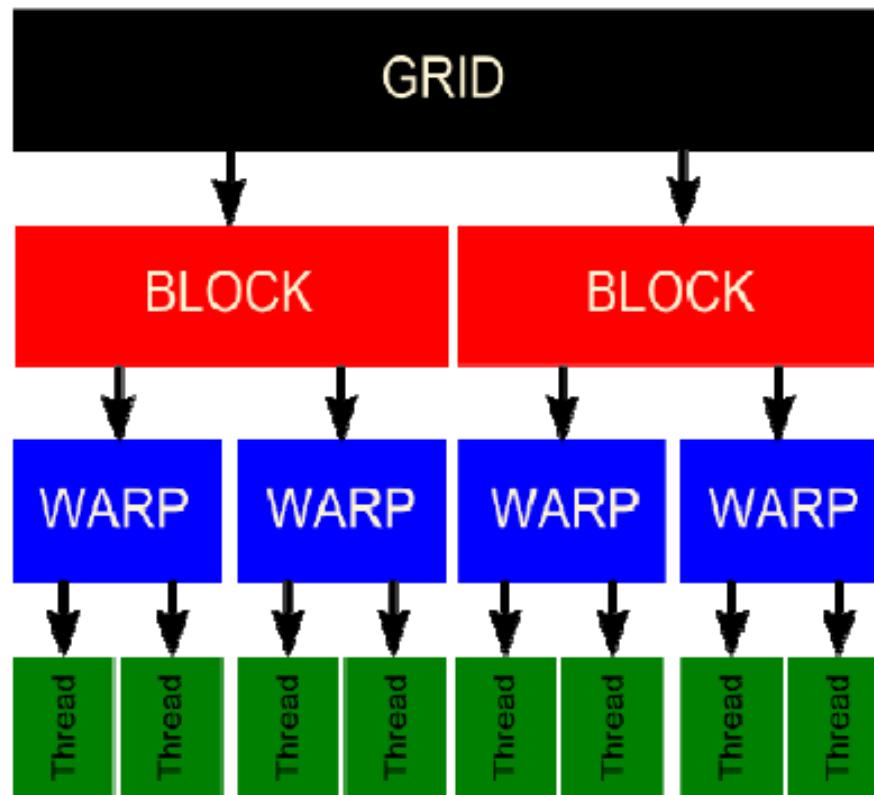
# CUDA Programming Model

- All threads within a block can
  - Share data through ‘Shared Memory’
  - Synchronize using ‘`_syncthreads()`’
- Threads and Blocks have unique IDs
  - Available through special variables

# CUDA Programming Model

- SIMT (Single Instruction Multiple Threads) Execution
- Threads run in groups of 32 called warps
- Every thread in a warp executes the same instruction at a time

# CUDA Programming Model

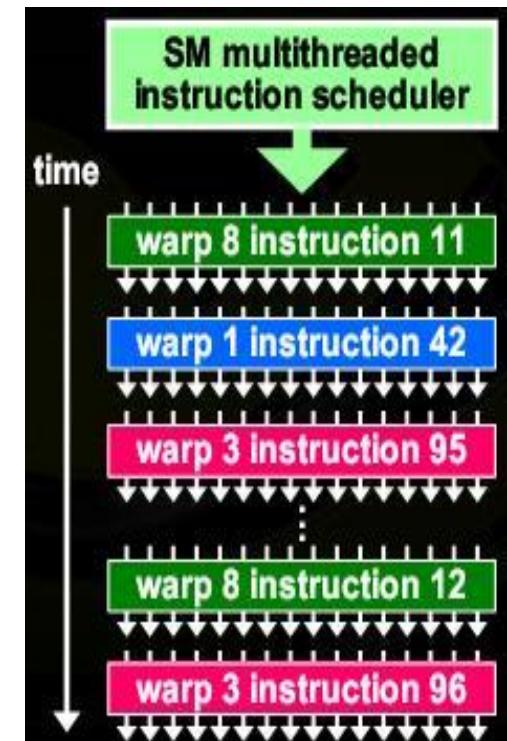


© NVIDIA Corporation

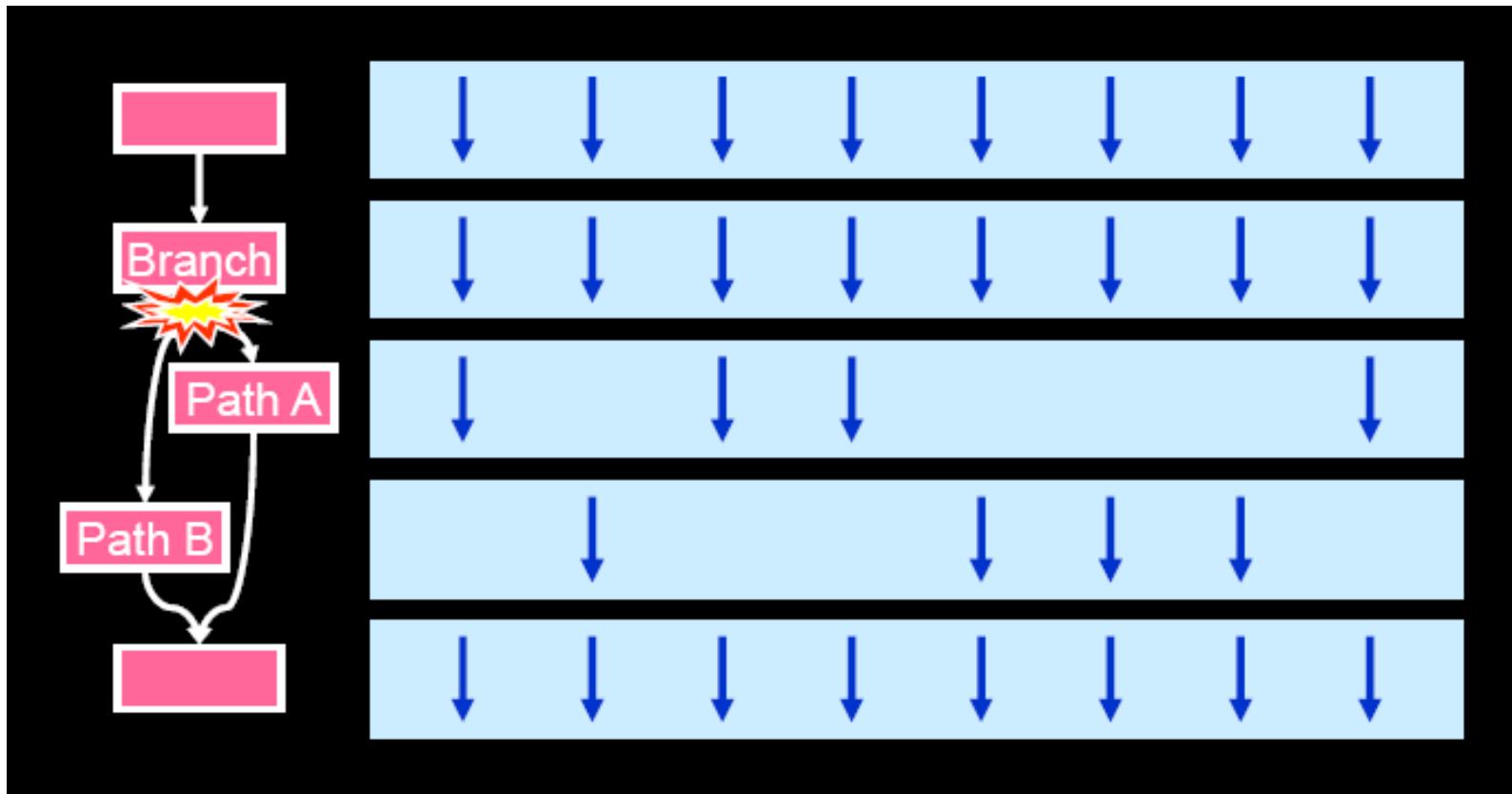
# CUDA Programming Model

## Single Instruction Multiple Thread (SIMT) Execution:

- Groups of 32 threads formed into **warps**
  - always executing same instruction
  - share instruction fetch/dispatch
  - some become **inactive** when code path diverges
  - hardware **automatically handles divergence**
- **Warps** are primitive unit of scheduling
  - all warps from all active blocks are time-sliced

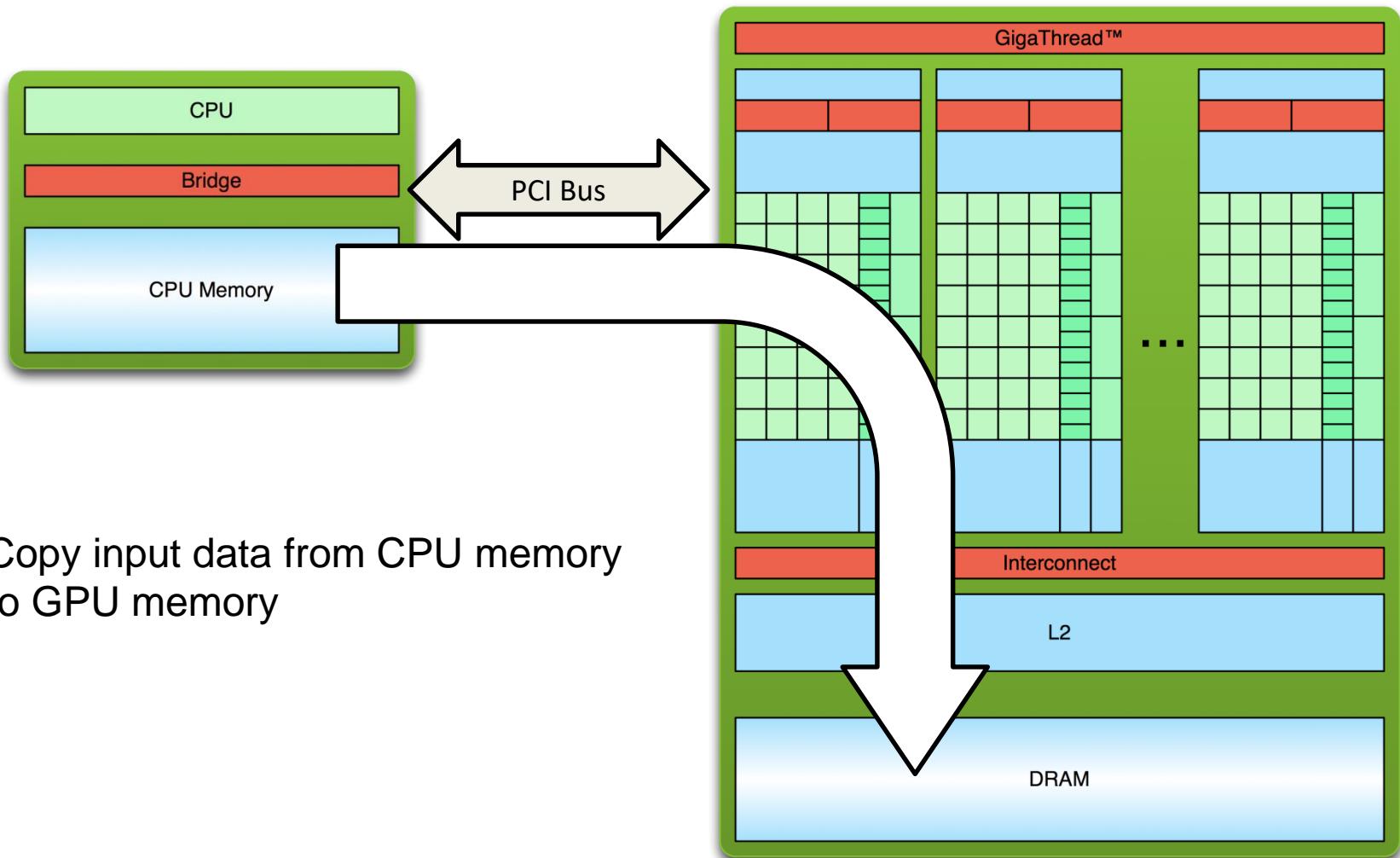


# Control Flow Divergence

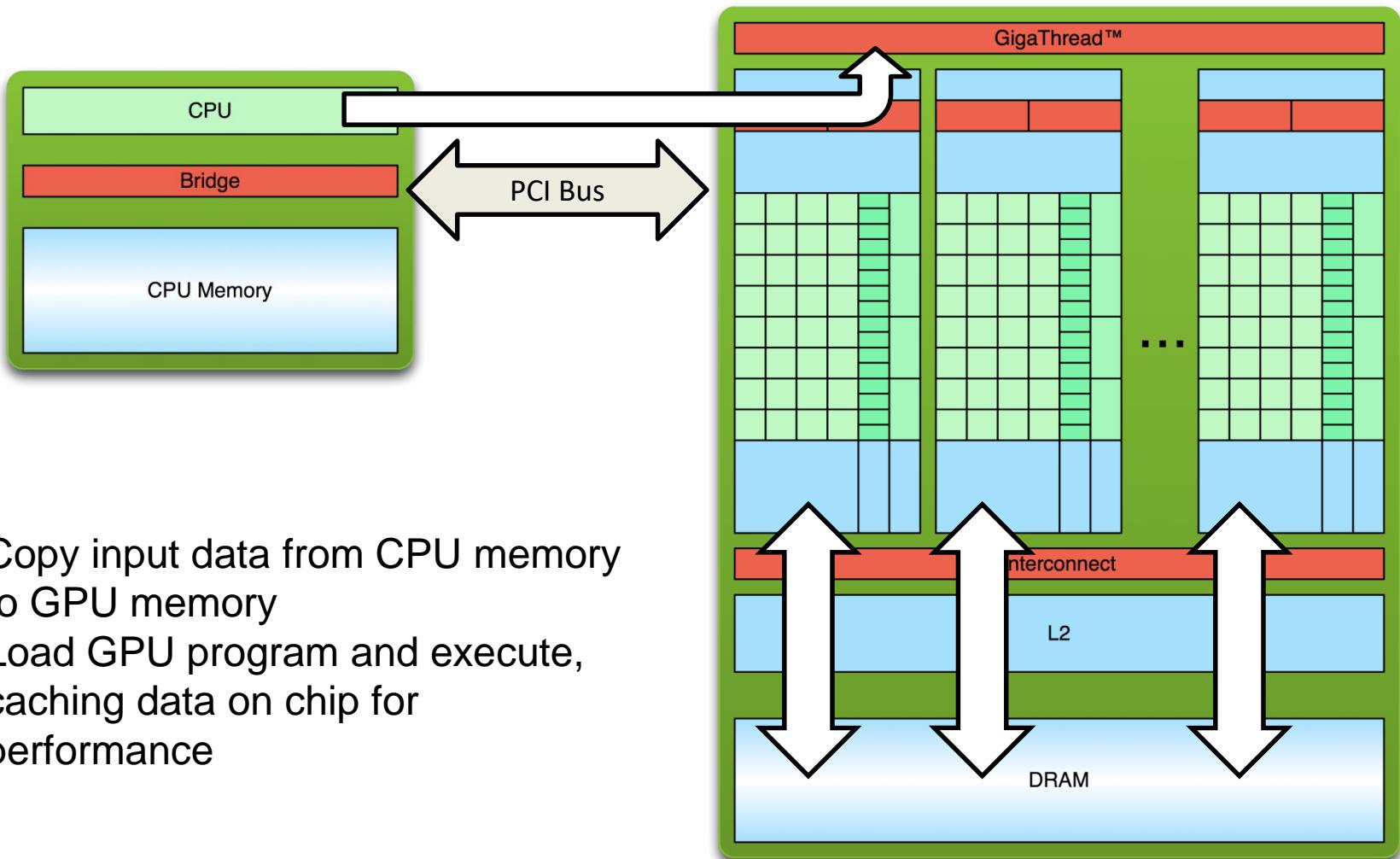


Courtesy Fung et al. MICRO '07

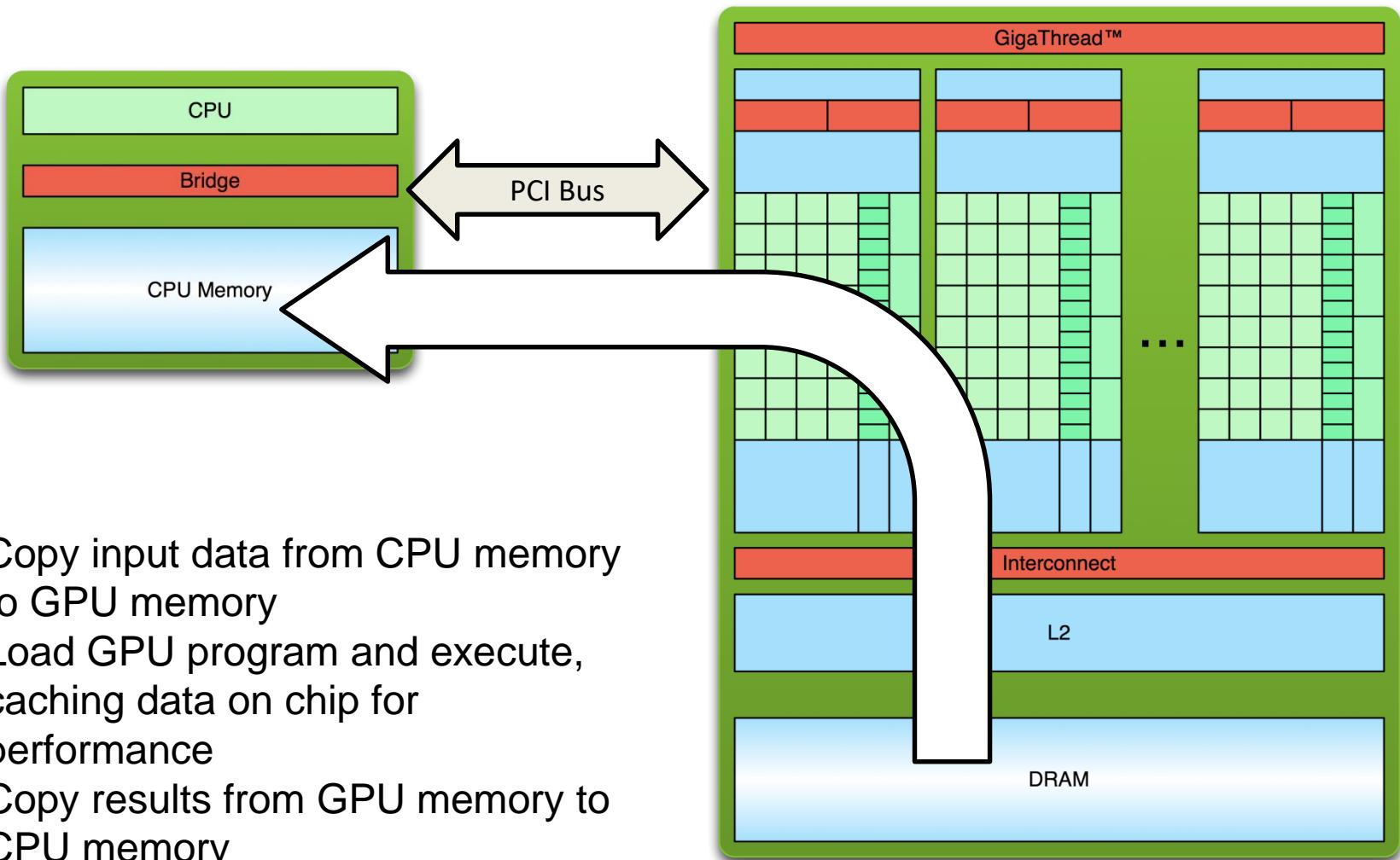
# Simple Processing Flow



# Simple Processing Flow



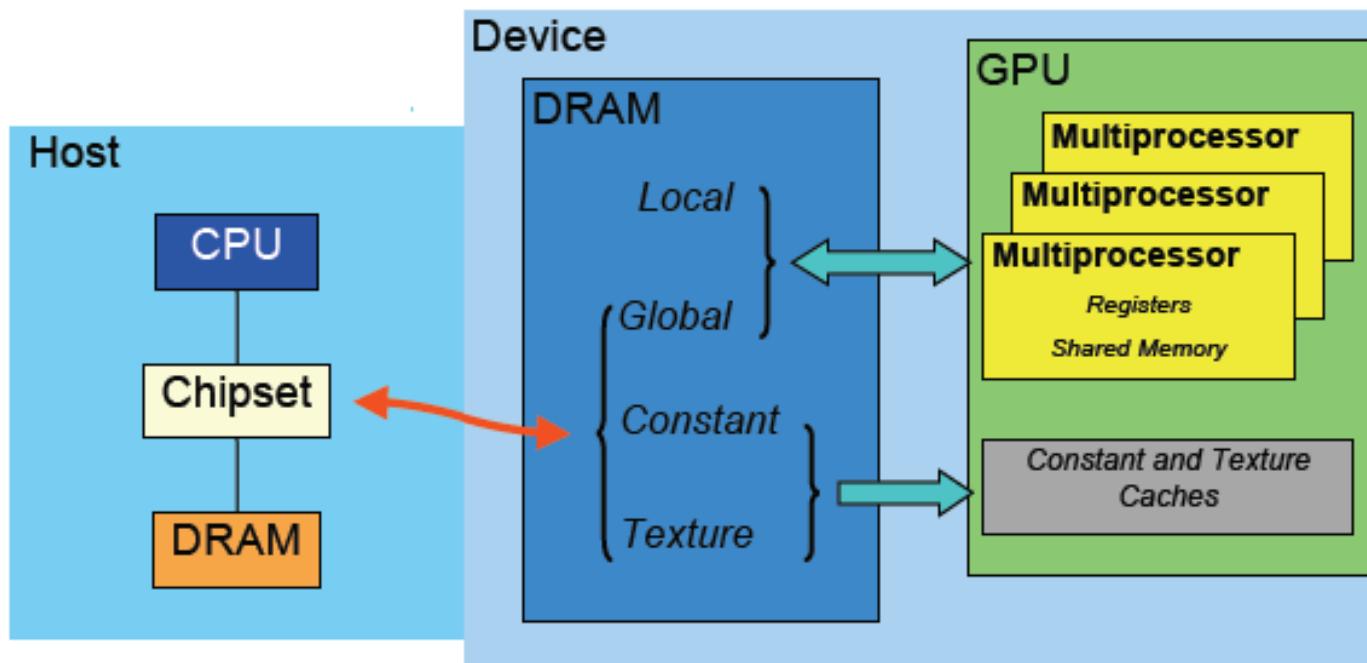
# Simple Processing Flow



# Memory Model

- Types of device memory
  - Registers – read/write per-thread
  - Local Memory – read/write per-thread
  - Shared Memory – read/write per-block
  - Global Memory – read/write across grids
  - Constant Memory – read across grids
  - Texture Memory – read across grids

# Memory Model

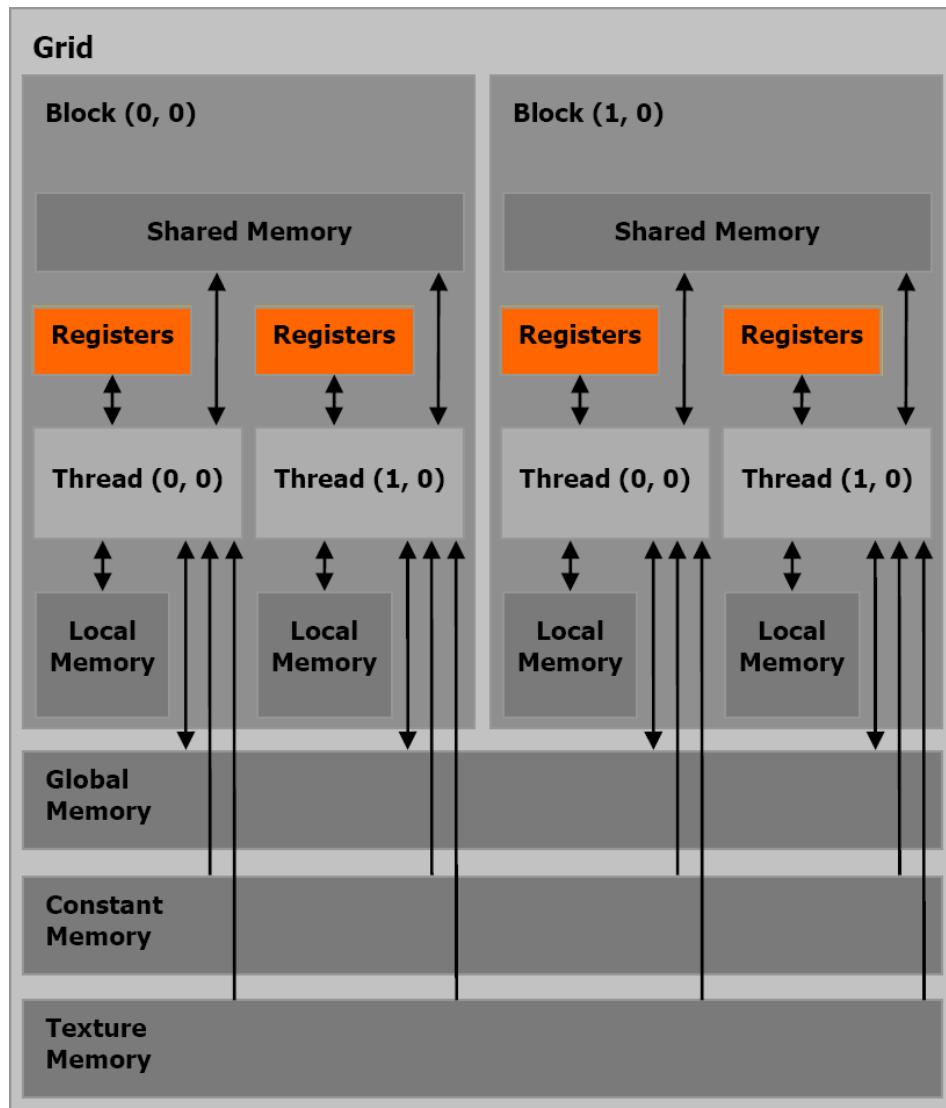


© NVIDIA Corporation

# Memory Model

There are 6 Memory Types :

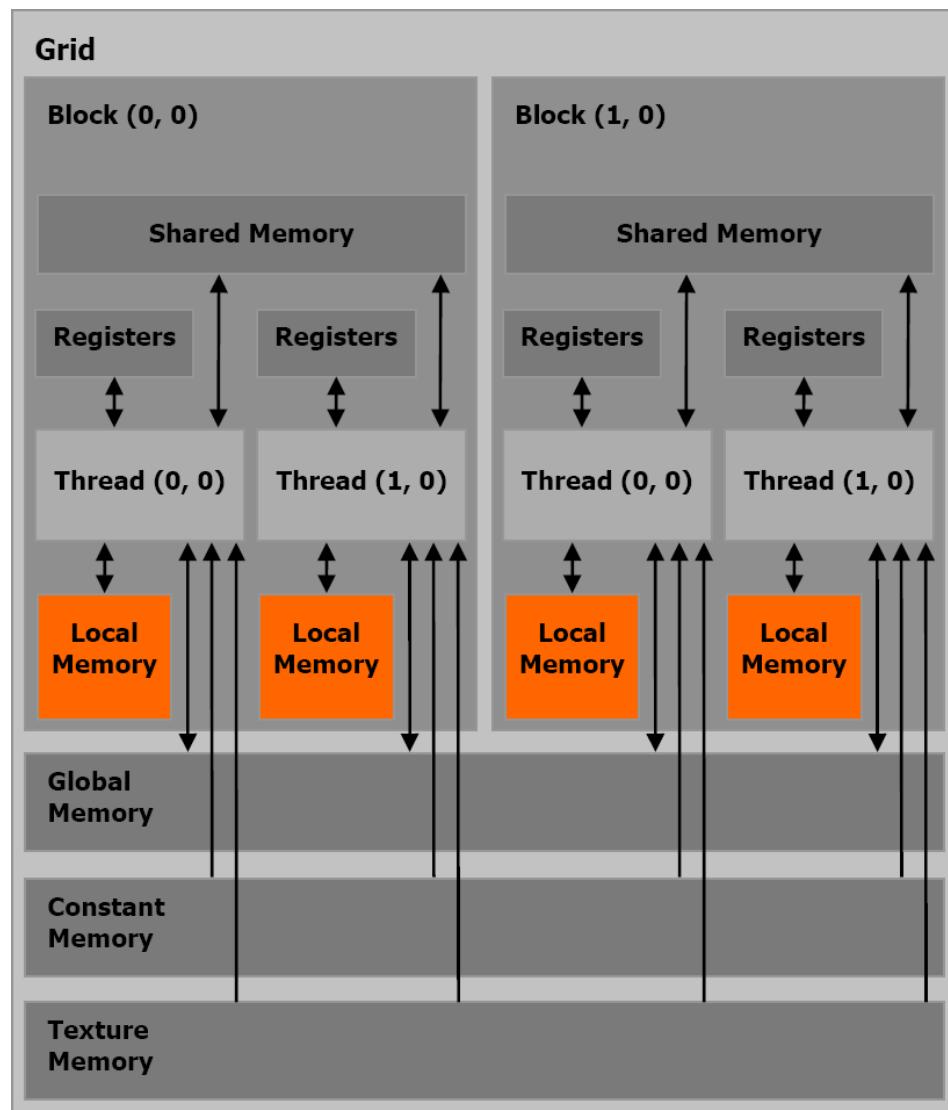
- **Registers**
  - on chip
  - fast access
  - per thread
  - limited amount
  - 32 bit



# Memory Model

There are 6 Memory Types :

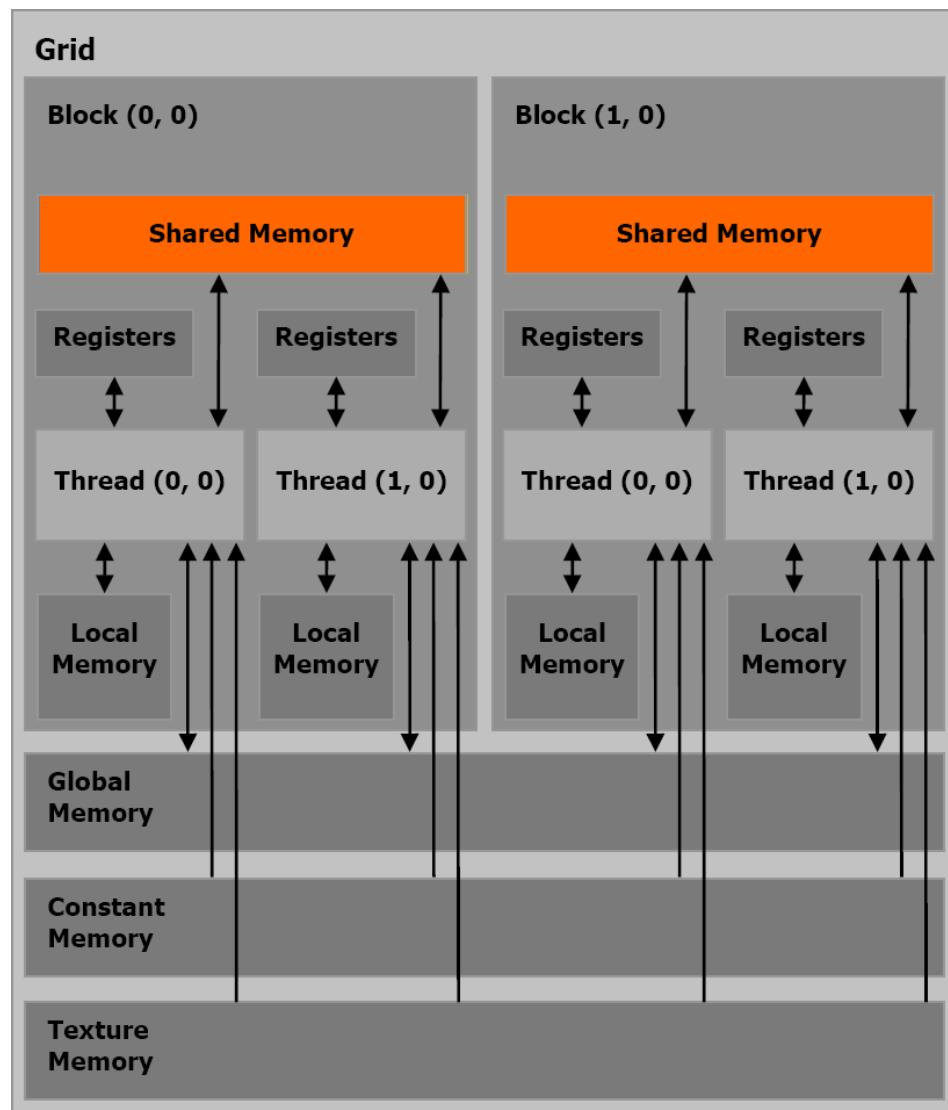
- Registers
- **Local Memory**
  - in DRAM
  - slow
  - non-cached
  - per thread
  - relative large



# Memory Model

There are 6 Memory Types :

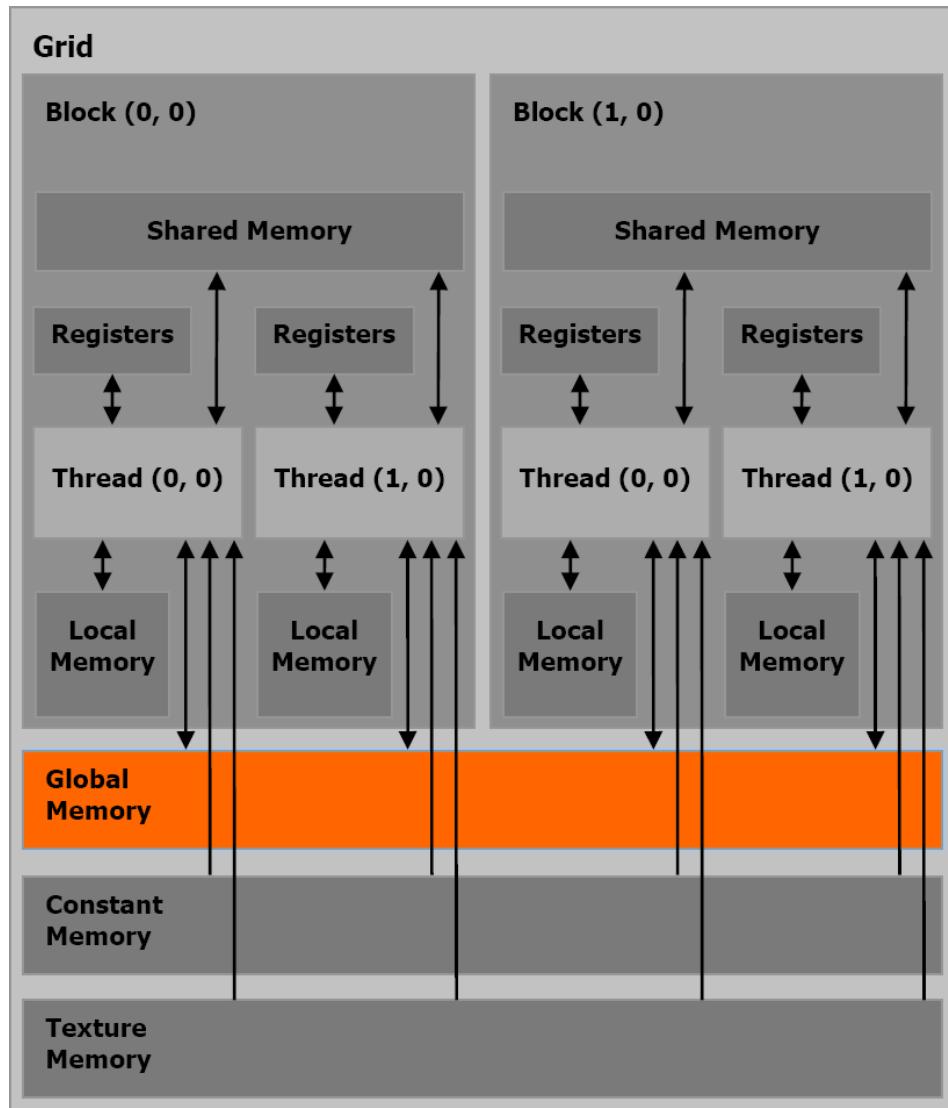
- Registers
- Local Memory
- **Shared Memory**
  - on chip
  - fast access
  - per block
  - 16 KByte
  - synchronize between threads



# Memory Model

There are 6 Memory Types :

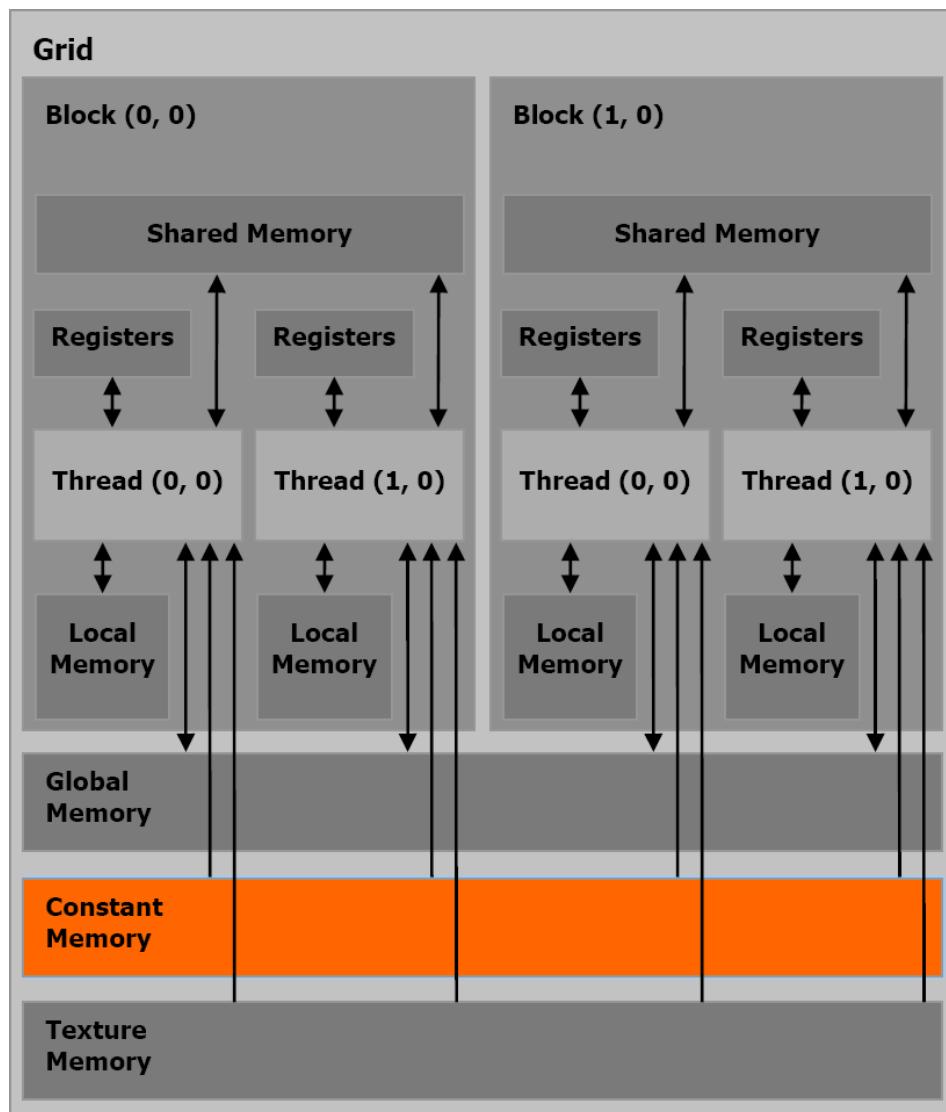
- Registers
- Local Memory
- Shared Memory
- **Global Memory**
  - in DRAM
  - slow
  - non-cached
  - per grid
  - communicate between grids



# Memory Model

There are 6 Memory Types :

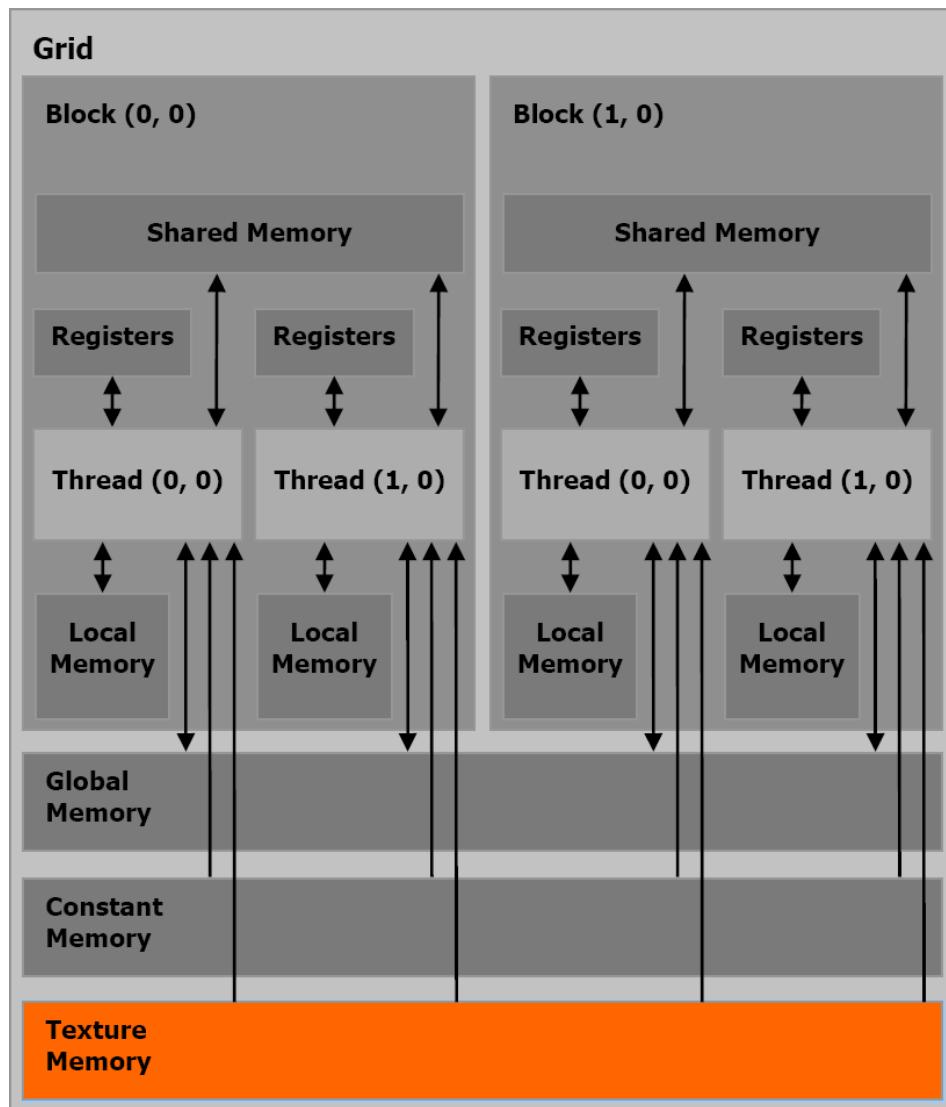
- Registers
- Local Memory
- Shared Memory
- Global Memory
- **Constant Memory**
  - in DRAM
  - cached
  - per grid
  - **read-only**



# Memory Model

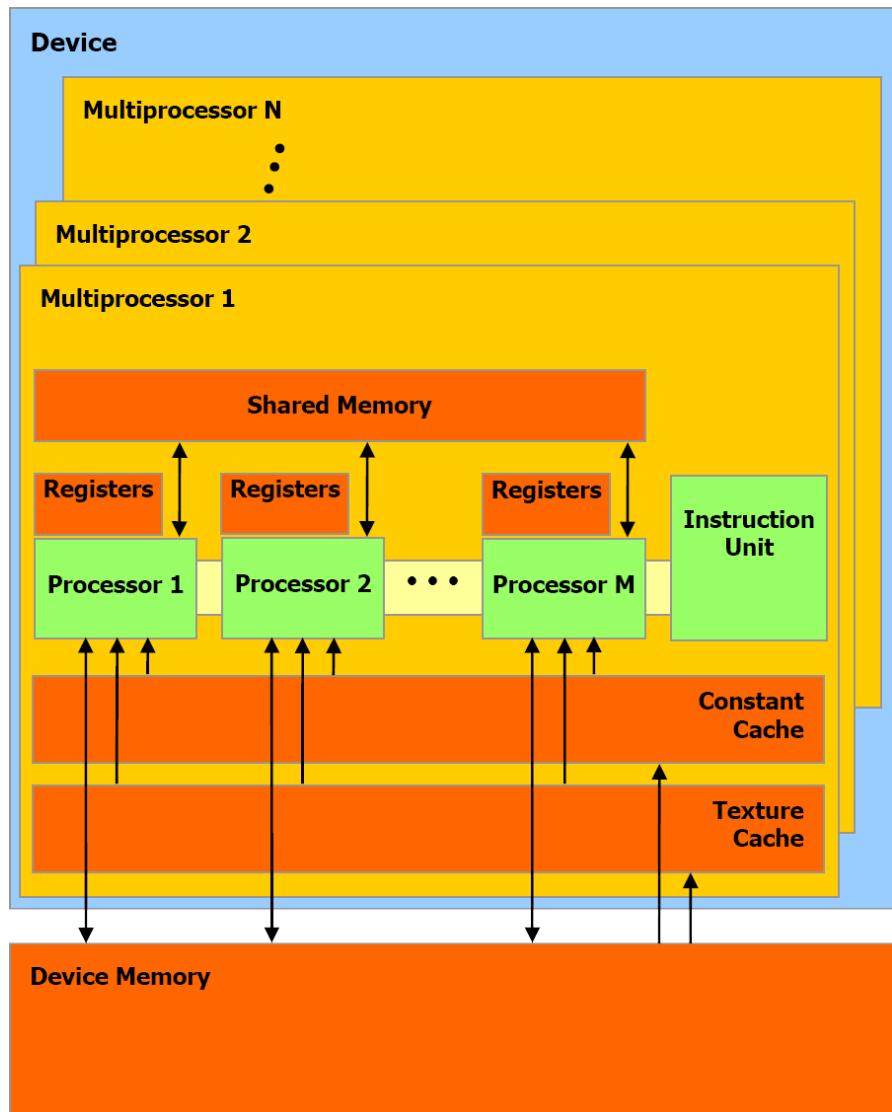
There are 6 Memory Types :

- Registers
- Local Memory
- Shared Memory
- Global Memory
- Constant Memory
- **Texture Memory**
  - in DRAM
  - cached
  - per grid
  - **read-only**



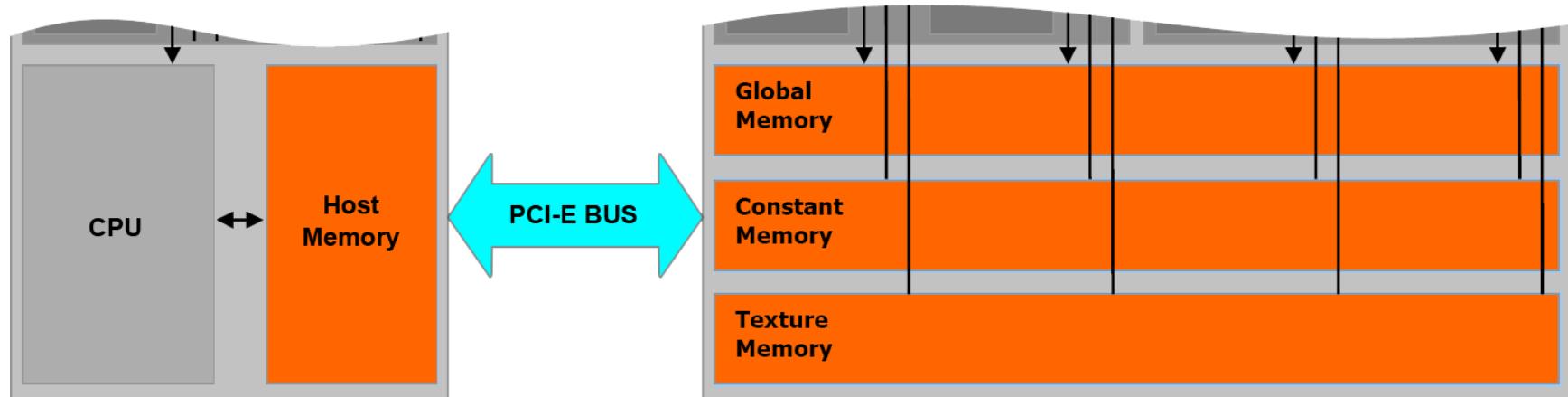
# Memory Model

- Registers
- Shared Memory
  - on chip
- Local Memory
- Global Memory
- Constant Memory
- Texture Memory
  - in Device Memory



# Memory Model

- Global Memory
- Constant Memory
- Texture Memory
  - managed by host code
  - persistent across kernels



# Advantages of CUDA

- ❑ CUDA has several advantages over traditional general purpose computation on GPUs:
  - ❑ Scattered reads – code can read from arbitrary addresses in memory.
  - ❑ Shared memory - CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads.

# Limitations of CUDA

- ❑ CUDA has several limitations over traditional general purpose computation on GPUs:
  - ❑ A single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.
  - ❑ The bus bandwidth and latency between the CPU and the GPU may be a bottleneck.
  - ❑ CUDA-enabled GPUs are only available from NVIDIA.

**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 3 – Fall 2021**

## Question

1. What GPGPU stands for and what does it mean.

General Purpose Graphical Procces Unit, It means that the gpu does the procceses that are usally done by cpu

2. Why CUDA is said Heterogeneous computing.

Becasue there are some portaion of the code that is done in serial by the cpu and other portaion of the code is done in parallel by gpu

3. Give the definition of the following terms:

- a. Device: GPU and it's memory
- b. Kernel. the portaion of the code that will be preformed on the device
- c. Grid of thread blocks. it's a computing model of cuda it consist of orginazing threads in diffrent blocks where each of them is composed of set of threads
- d. Warp. groups of 32 thread that always excute same instruction

4. Explain the parallel programming model of CUDA.

Grids composed of set of blocks evry block composed of set threads, threads are grouped of warps that excute the same instruction warps are time sliced

5. Enumerate and explain the different types of memory adopted by CUDA.

- 1- register ( on chip, fast, per thread, store data up to 32bit )
- 2- local memory ( slow, DRAM ,not cached, per thread )
- 3- shared memory ( on chip , fast , per block, not cached )
- 4- global memory ( DRAM , not cached , per grid )
- 5- constant ( DRAM , read-only , cached , per grid )
- 6- texture memory ( DRAM , cached , read-only )

# CUDA Programming

Hello Program

# Outline

- ❑ CUDA Programming
  - ❑ Functions Qualifiers
  - ❑ Built-in Device Variables
  - ❑ Variable Qualifiers
- ❑ Addition on the device
  - ❑ Moving to parallel using blocks
  - ❑ Moving to parallel using threads
  - ❑ Combining blocks and threads

# Cuda Programming

- **Kernels** are C functions with some restrictions
  - Can only access GPU memory
  - Must have void return type
  - No variable number of arguments (“varargs”)
  - Not recursive
  - No static variables
- Function arguments automatically copied from CPU to GPU memory

# Function Qualifiers

- **\_\_global\_\_** : invoked from within host (CPU) code,
  - cannot be called from device (GPU) code
  - must return void
- **\_\_device\_\_** : called from other GPU functions,
  - cannot be called from host (CPU) code
- **\_\_host\_\_** : can only be executed by CPU, called from host
- **\_\_host\_\_ and \_\_device\_\_ qualifiers can be combined**
  - Sample use: overloading operators
  - Compiler will generate both CPU and GPU code

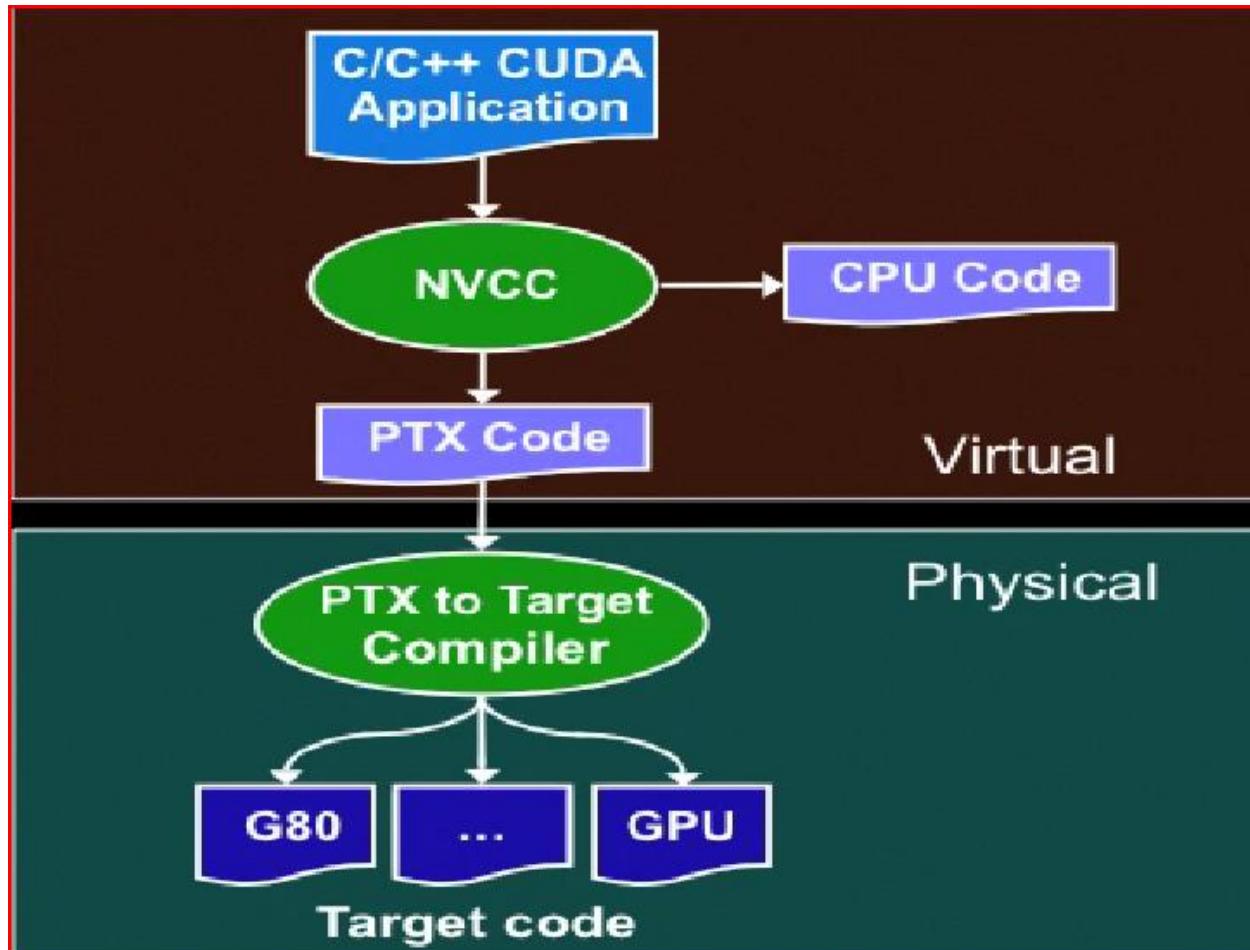
# Variable Qualifiers (GPU code)

- **\_\_device\_\_**
  - Stored in device memory (large, high latency, no cache)
  - Allocated with cudaMalloc (**\_\_device\_\_** qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- **\_\_shared\_\_**
  - Stored in on-chip shared memory (very low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: kernel execution
- **Unqualified variables:**
  - Scalars and built-in vector types are stored in registers
  - Arrays of more than 4 elements stored in device memory

# CUDA Built-in Device Variables

- All `__global__` and `__device__` functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

# CUDA Compile



# CUDA Compile

`nvcc <filename>.cu [-o <executable>]`

- Builds release mode

`nvcc -g <filename>.cu`

- Builds debug mode
- Can debug host code but not device code

`nvcc -deviceemu <filename>.cu`

- Builds device emulation mode
- All code runs on CPU, no debug symbols

`nvcc -deviceemu -g <filename>.cu`

- Builds debug device emulation mode
- All code runs on CPU, with debug symbols

# Hello World!

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc  
hello_world.  
cu  
$ a.out  
Hello World!  
$
```

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...

# Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

**Output:**

```
$ nvcc  
hello.cu  
$ a.out  
Hello World!  
$
```

- **mykernel () does nothing**

# Hello World! with Device Code

```
__global__ void mykernel(void) {
    printf("Hello World!\n");
}
```

**Output:**

```
int main(void) {
    mykernel<<<1,1>>>();
    return 0;
}
```

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

# Hello World! with Device Code

```
__global__ void mykernel(void) {
    printf("Hello World!\n");
}
```

Output:

```
int main(void) {
    mykernel<<<2,2>>>();
    return 0;
}
```

```
$ nvcc
hello.cu
$ a.out
Hello World!
Hello World!
Hello World!
Hello World!
$
```

# CUDA Programming

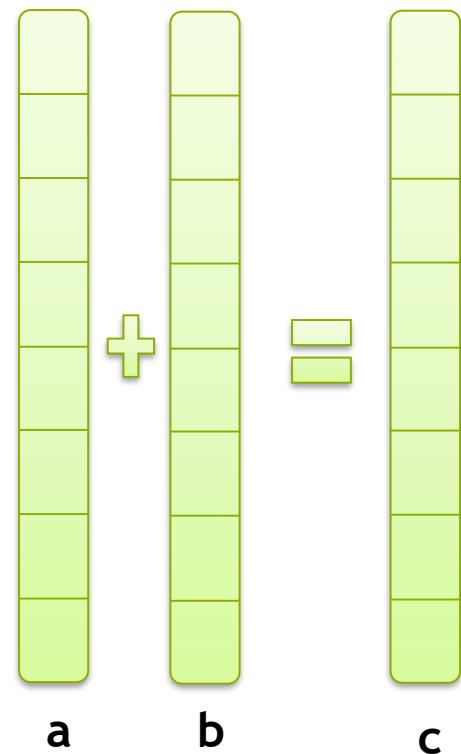
Sum of 2 Arrays

# Outline

- ❑ Addition on the device
  - ❑ Moving to parallel using blocks
  - ❑ Moving to parallel using threads
  - ❑ Combining blocks and threads

# Parallel Programming in CUDA C/C++

- GPU computing is about massive parallelism!
- We'll start by adding two integers and build up to vector addition



# Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

# Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

# Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



# Addition on the Device: add()

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Let's take a look at `main()`...

# Addition on the Device: main()

```
int main(void) {
    int a, b, c;                      // host copies of a, b, c
    int *d_a, *d_b, *d_c;              // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```

# Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Moving to Parallel

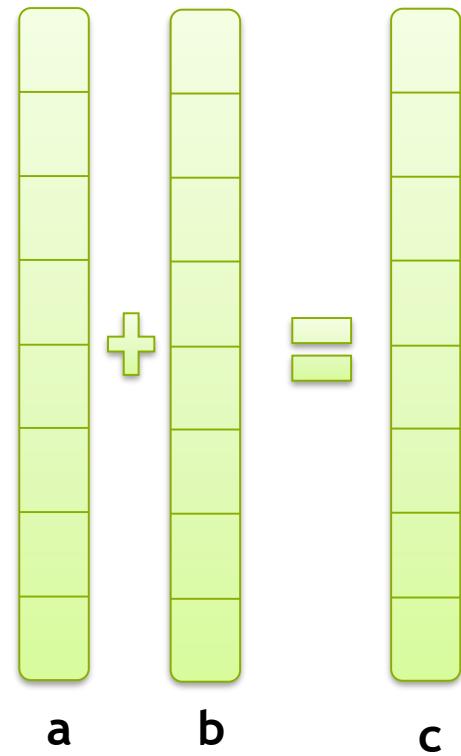
- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```



```
add<<< N, 1 >>>();
```

- Instead of executing add () once, execute N times in parallel



# Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
  - The set of blocks is referred to as a **grid**
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

# Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

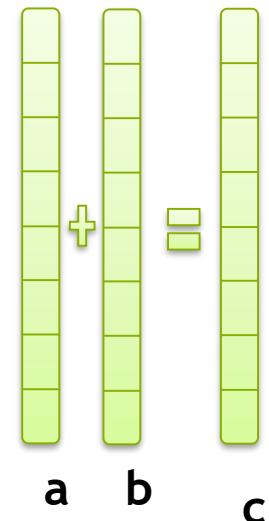
```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```



# Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let's take a look at `main()`...

# Vector Addition on the Device: main()

```
#define N 512

int main(void) {
    int *a    *b    *c          // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# CUDA Threads

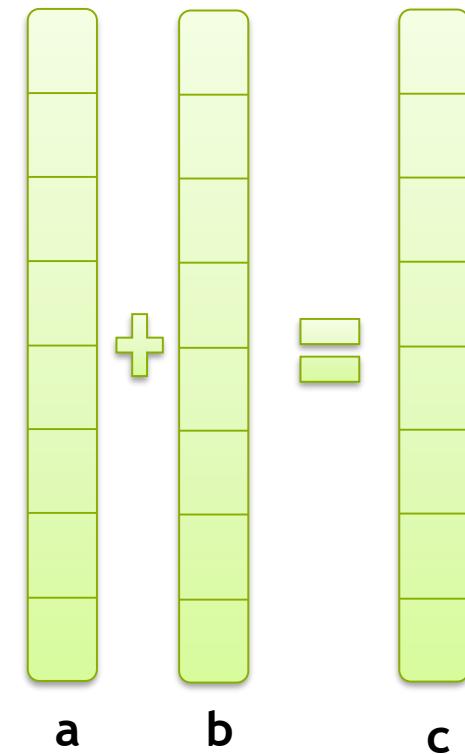
- Terminology: a block can be split into parallel **threads**

Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c)  
{  
}
```

- Need to make one change in `main()`...

```
    add<<< 1, 1 >>>();  
    add<<< 1, N >>>();
```



# Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **thread**
  - Each invocation can refer to its thread index using `threadIdx.x`

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- By using `threadIdx.x` to index into the array, each thread handles a different index

# Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- On the device, each thread can execute in parallel:

Thread 0

```
c[0] = a[0] + b[0];
```

Thread 1

```
c[1] = a[1] + b[1];
```

Thread 2

```
c[2] = a[2] + b[2];
```

Thread 3

```
c[3] = a[3] + b[3];
```

# Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- Let's take a look at `main()`...

# Vector Addition on the Device: main()

```
#define N 512

int main(void) {
    int *a    *b    *c          // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<1, N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

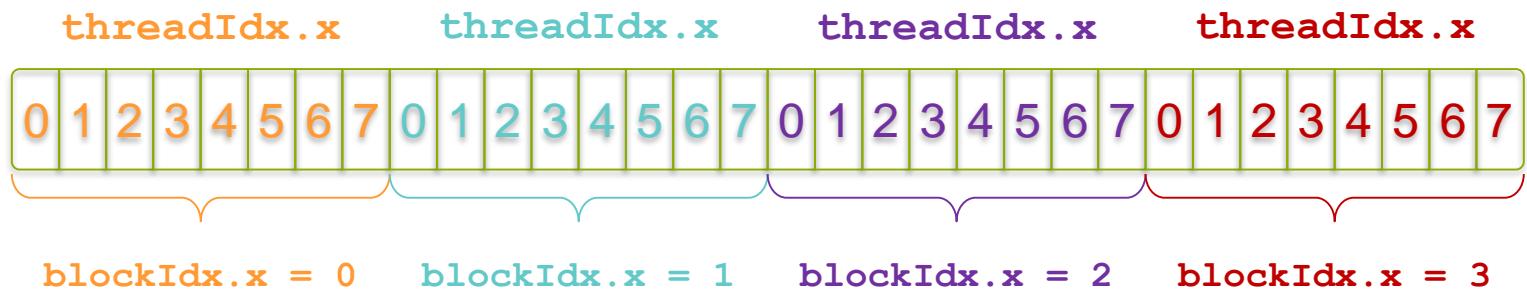
// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that...
- First let's discuss data indexing...

# Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

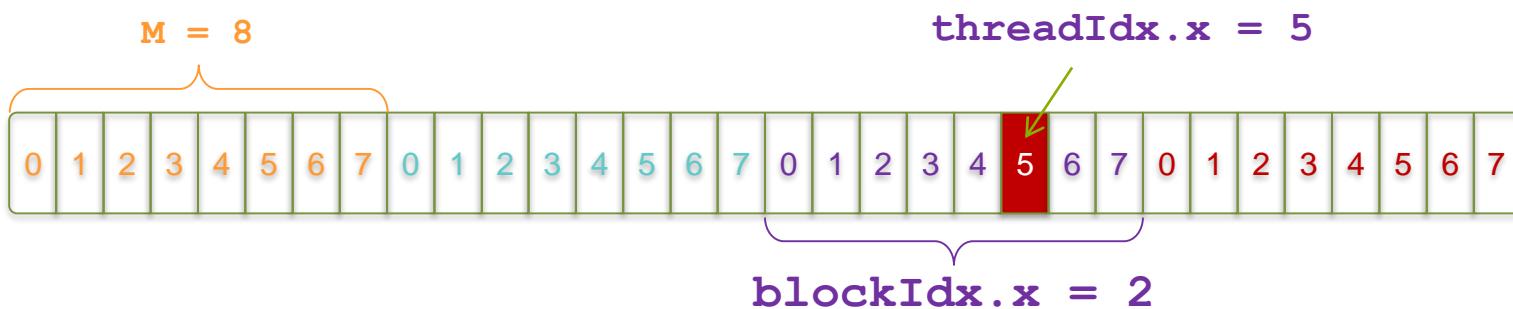


- With  $M$  threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

# Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
= 5 + 2 * 8;  
= 21;
```

# Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}
```

- What changes need to be made in `main()`?

# Addition with Blocks and Threads: main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                          // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Addition with Blocks and Threads: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Handling Arbitrary Vector Sizes

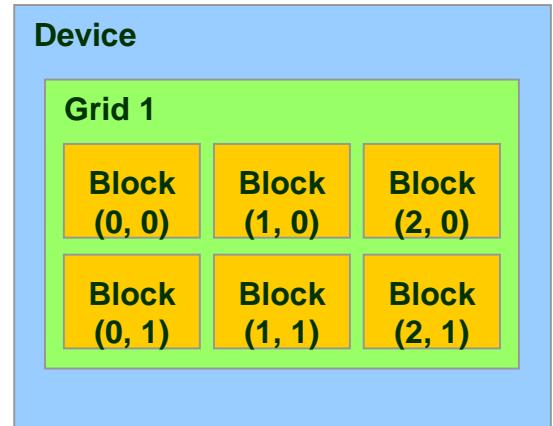
- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- Update the kernel launch:  
`add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);`

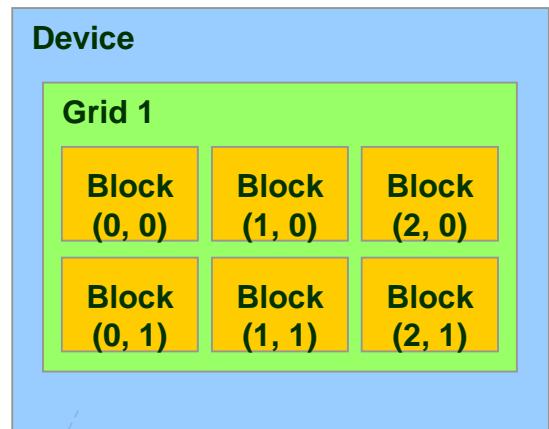
# Formatting the grid as a Matrix

- `dim3 grid(3,2);`
- `kernel<<grid, 1>>(...);`



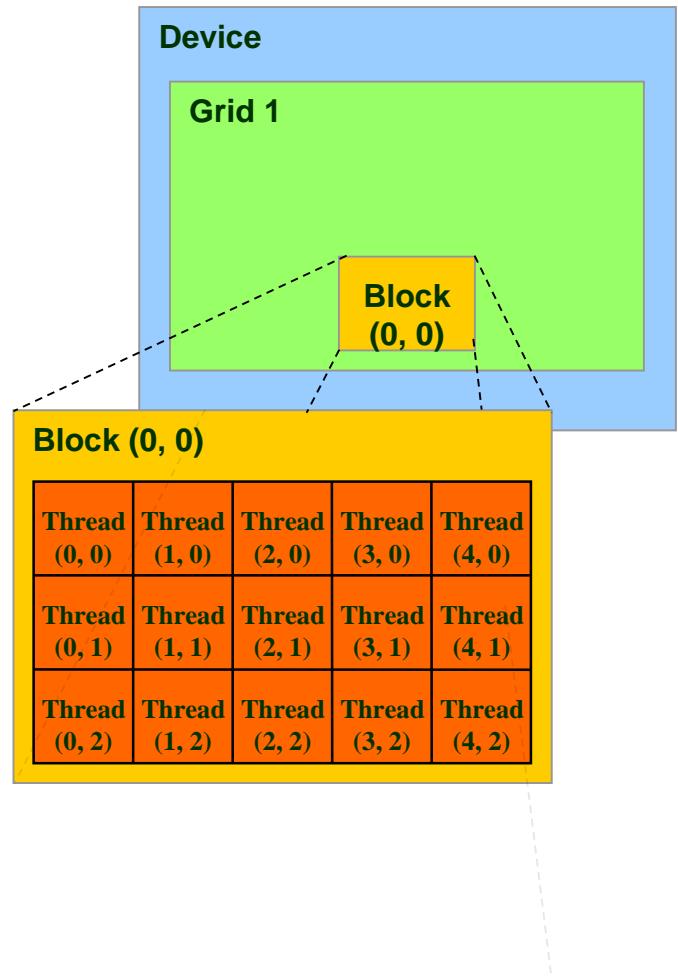
# Formatting the grid as a Matrix

- dim3 `grid` (3,2);
- `kernel<<<grid, 1>>>(...);`
- `int index = blockIdx.x + blockIdx.y * gridDim.x;`



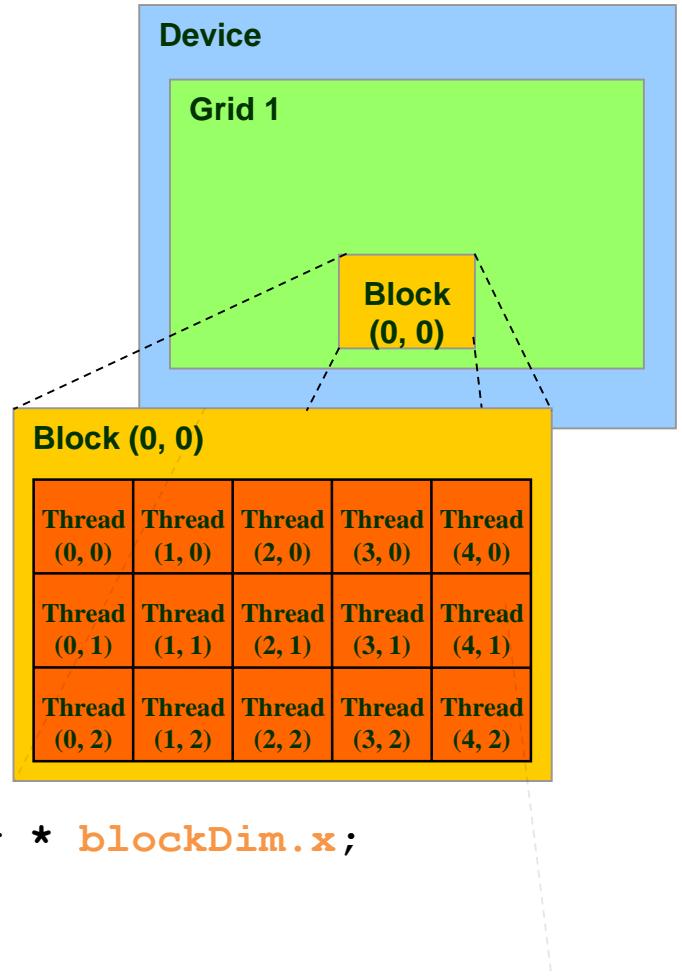
# Formatting the grid as a Matrix

- dim3 threads(5,3);
- kernel<<<1, threads>>>(...);



# Formatting the grid as a Matrix

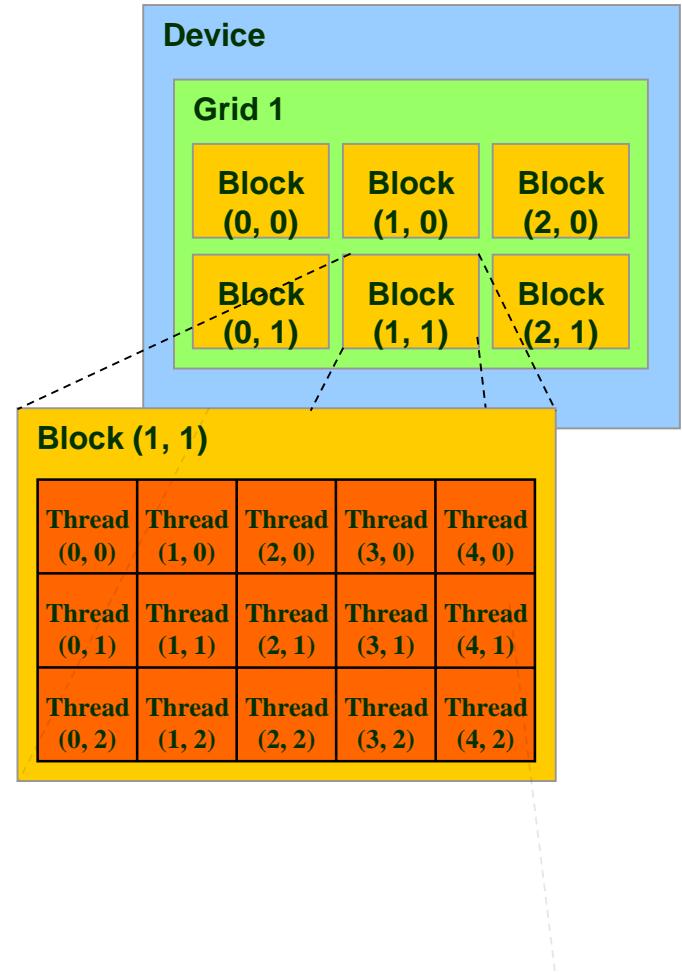
- `dim3 threads(5,3);`
- `kernel<<1, threads>>(...);`



- `int index = threadIdx.x + threadIdx.y * blockDim.x;`

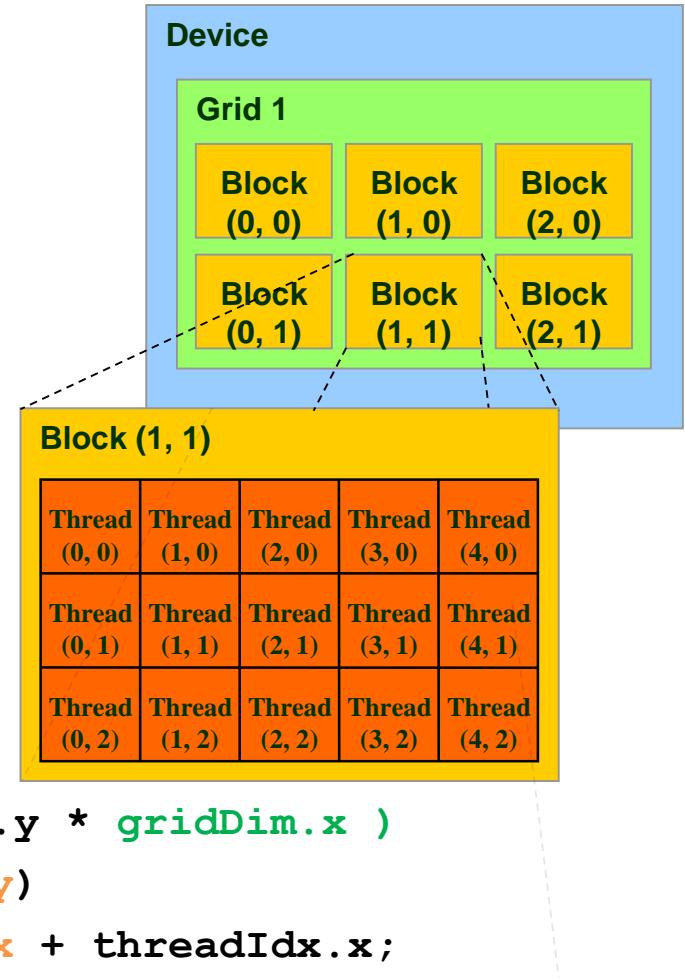
# Formatting the grid as a Matrix

- dim3 **grid** (3,2);
- dim3 **block**(5,3);
- kernel<<<**grid**, **block**>>>(...);



# Formatting the grid as a Matrix

- dim3 **grid**(3,2);
- dim3 **block**(5,3);
- kernel<<<**grid**, **block**>>>(...);



- `int index = ( blockIdx.x + blockIdx.y * gridDim.x )  
* (blockDim.x * blockDim.y)  
+ threadIdx.y * blockDim.x + threadIdx.x;`

**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 4 – Spring 2021**

## Question 1

1. Let's consider 2 integer Arrays A and B of dimension N. Let's consider that we would like to write a C program that runs in parallel and that computes the sum of the 2 arrays:

$$C[i] = A[i] + B[i]$$

- a. Write the kernel (called *kernel\_1*) that will run on 1 Block of N threads.

```
__global__ void add( int *a, int *b, int *c){  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- b. Write another kernel (called *kernel\_2*) that will run on N blocks with 1 thread each.

```
__global__ void add( int *a, int *b, int *c){  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- c. Write the main program that will call both kernels.

Page 14 pdf

**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 4 – Spring 2021**

## Question 2

Let's consider 2 integer Arrays A and B of dimension N. Let's consider that we would like to write a C program that runs in parallel and that computes the sum of the 2 arrays:

$$C[\text{index}] = A[\text{index}] + B[\text{index}];$$

For every configuration of the grid of thread blocks described below, give the statement that computes the index for each each thread:

1. The grid is composed of 1 block and threads should have ids as in the following figure:

Block (0, 0)				
Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
Thread 5	Thread 6	Thread 7	Thread 8	Thread 9
Thread 10	Thread 11	Thread 12	Thread 13	Thread 14

```
int index = threadIdx.x + ( blockDim.x * threadIdx.y );
```

2. The grid is composed of 1 block and threads should have ids as in the following figure:

Block (0, 0)				
Thread 0	Thread 3	Thread 6	Thread 9	Thread 12
Thread 1	Thread 4	Thread 7	Thread 10	Thread 13
Thread 2	Thread 5	Thread 8	Thread 11	Thread 14

2

```
int index = threadIdx.y + ( blockDim.y * threadIdx.x );
```

**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 4 – Spring 2021**

# CUDA Programming

Addition of 2 Matrices

# Matrix Addition on the Device: add()

- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
    int rowIndex, colIndex;

    rowIndex = threadIdx.y;
    colIndex = threadIdx.x;

    c[rowIndex][colIndex] = a[rowIndex][colIndex] +
                           b[rowIndex][colIndex];
}
```

Running the kernel with 1 Block of  $N \times N$  threads.

# Matrix Addition on the Device: main()

```
#define N 1024

int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;          // device copies of a, b, c
    int nb = N * N;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

# Matrix Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
dim3 block(N, N);

// Launch add() kernel on GPU with N blocks
add<<<1, block>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Matrix Addition on the Device: add()

- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
    int rowIndex, colIndex;

    rowIndex = blockIdx.y;
    colIndex = blockIdx.x;

    c[rowIndex][colIndex] = a[rowIndex][colIndex] +
                           b[rowIndex][colIndex];
}
```

Running the kernel with  $N \times N$  Blocks with 1 thread each.

# Matrix Addition on the Device: main()

```
#define N 1024

int main(void) {
    int *a    *b    *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int nb = N * N;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

# Matrix Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
dim3 grid(N, N);

// Launch add() kernel on GPU with N blocks
add<<<grid, 1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Matrix Addition on the Device: add()

- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
    int rowIndex, colIndex;

    rowIndex = blockIdx.y * blockDim.y + threadIdx.y;
    colIndex = blockIdx.x * blockDim.x + threadIdx.x;

    c[rowIndex][colIndex] = a[rowIndex][colIndex] +
                           b[rowIndex][colIndex];

}
```

Running the kernel with  $(N \times N)$  grid Blocks with  $(N \times N)$  threads each.

# Matrix Addition on the Device: main()

```
#define N 1024

int main(void) {
    int *a  *b  *c                      // host copies of a, b, c
    int *d_a, *d_b, *d_c;   // device copies of a, b, c
    int nb = N * N * N * N;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

# Matrix Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
dim3 grid(N, N);
dim3 block(N, N);

// Launch add() kernel on GPU with N blocks
add<<<grid, block>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Matrix Addition on the Device: add()

- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c, int width) {
    int rowIndex, colIndex, width;

    rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;
    colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;

    for (int i=0; i < width; i++)
        for (int j=0; j < width; j++)
            c[rowIndex + i][colIndex + j] =
                a[rowIndex + i][colIndex + j] +
                b[rowIndex + i][colIndex + j];
}
```

# Matrix Addition on the Device: main()

```
#define N 16

int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c, *w;      // device copies of a, b, c
    int width = 16;
    int nb = N * N * N * N * width * width;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

# Matrix Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
dim3 grid(N, N);
dim3 block(N, N);

// Launch add() kernel on GPU with N blocks
add<<<grid,block>>>(d_a, d_b, d_c, width);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 5 – Fall 2021**

## Question 1

We would like to run a kernel on grid configured as  $M * N$  matrix of thread blocks. Every thread handles only one cell. Give the statement that calculates the *cell\_id* for each thread as shown in the following figure:

Block (0, 0)				
Cell 0	Cell 1	Cell 2	Cell 3	Cell 4
Cell 5	Cell 6	Cell 7	Cell 8	Cell 9
Cell 10	Cell 11	Cell 12	Cell 13	Cell 14

Block (1, 0)				
Cell 30	Cell 31			Cell 34
Cell 35				Cell 39
Cell 40				Cell 44

Block (0, 1)				
Cell 15	Cell 16			Cell 19
Cell 20				Cell 24
Cell 25				Cell 29

Block (1, 1)				
Cell 45	Cell 46	Cell 47	Cell 48	Cell 49
Cell 50	Cell 51	Cell 52	Cell 53	Cell 54
Cell 55	Cell 56	Cell 57	Cell 58	Cell 59

# CUDA Programming

Product of Matrices

## Matrix multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 0 & 7 \end{bmatrix} = \begin{bmatrix} 1*5 + 2*0 & 1*6 + 2*7 \\ 3*5 + 4*0 & 3*6 + 4*7 \end{bmatrix} = \begin{bmatrix} 5 & 20 \\ 15 & 46 \end{bmatrix}$$

$$C[i][j] = \sum_{k=0}^n A[i][k] * B[k][j]$$

# Matrix Product on the Device: add()

- Parallelized `product()` kernel

```
__global__ void product(int *a, int *b, int *c, int n) {
    int rowIndex, colIndex, width;

    rowIndex = blockIdx.y * blockDim.y + threadIdx.y;
    colIndex = blockIdx.x * blockDim.x + threadIdx.x;

    for (int k=0; k < n; k++)
        c[rowIndex][colIndex] += a[rowIndex][k] * b[k][colIndex];
}
```

# Matrix Addition on the Device: main()

```
#define N 16

int main(void) {
    int *a    *b    *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c, *w;        // device copies of a, b, c
    int width = 16;
    int nb = N * N * N * N;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

# Matrix Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
dim3 grid(N, N);
dim3 block(N, N);

// Launch add() kernel on GPU with N blocks
product<<<grid,block>>>(d_a, d_b, d_c, N * N);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# Matrix Product on the Device: add()

- Parallelized `product()` kernel

```
__global__ void product(int *a, int *b, int *c, int width, int n) {
    int rowIndex, colIndex, width;

    rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;
    colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;

    for (int i=rowIndex; i < rowIndex + width; i++)
        for (int j=colIndex; j < colIndex + width; j++)
            for (int k=0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];

}
```

# Matrix Addition on the Device: main()

```
#define N 16

int main(void) {
    int *a    *b    *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c, *w;        // device copies of a, b, c
    int width = 16;
    int nb = N * N * width * N * N * width;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

# Matrix Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
dim3 grid(N, N);
dim3 block(N, N);

// Launch add() kernel on GPU with N blocks
product<<<grid,block>>>(d_a, d_b, d_c, width, N * N * width );

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# CUDA Programming

# Outline

## □Querying Device

# Querying Device

```
int main(void) {  
    int count;  
    cudaGetDeviceCount( &count);  
    return 0;  
}
```

# Querying Device

Device Property	Description
<code>char name[256]</code>	An ASCII string identifying the device (e.g., "GeForce GTX 280")
<code>size_t totalGlobalMem</code>	The amount of global memory on the device in bytes
<code>size_t sharedMemPerBlock</code>	The maximum amount of shared memory a single block may use in bytes
<code>int regsPerBlock</code>	The number of 32-bit registers available per block
<code>int warpSize</code>	The number of threads in a warp
<code>size_t totalConstMem</code>	The amount of available constant memory

# Querying Device

Device Property	Description
<code>int maxThreadsPerBlock</code>	The maximum number of threads that a block may contain
<code>int maxThreadsDim[3]</code>	The maximum number of threads allowed along each dimension of a block
<code>int maxGridSize[3]</code>	The number of blocks allowed along each dimension of a grid
<code>int multiProcessorCount</code>	The number of multiprocessors on the device
<code>int concurrentKernels</code>	A boolean value representing whether the device supports executing multiple kernels within the same context simultaneously

# Querying Device

```
int main(void) {  
    int count;  
    cudaDeviceProp prop;  
  
    cudaGetDeviceCount( &count);  
    for (int i=0; i< count; i++) {  
        cudaGetDeviceProperties( &prop, i );  
        //Do something with our device's properties  
    }  
    return 0;  
}
```

# Querying Device

```
int main(void) {  
    int count;  
    cudaDeviceProp prop;  
  
    cudaGetDeviceCount( &count);  
    for (int i=0; i< count; i++) {  
        cudaGetDeviceProperties( &prop, i );  
        //Do something with our device's properties  
    }  
}
```

# Querying Device

```
printf( "Multiproc: %d\n", prop.multiProcessorCount );
printf( "Shared mem: %d\n", prop.sharedMemPerBlock );
printf( "Registers per mp: %d\n", prop.regsPerBlock );
printf( "Threads in warp: %d\n", prop.warpSize );
printf( "Threads: %d\n", prop.maxThreadsPerBlock );
printf( "Max Thread dimensions: (%d,%d,%d)\n",
        prop.maxThreadsDim[0],
        prop.maxThreadsDim[1],
        prop.maxThreadsDim[2] );
printf( "Max grid dimensions: (%d, %d, %d)\n",
        prop.maxGridSize[0], prop.maxGridSize[1],
        prop.maxGridSize[2] );
}

return 0;
```



# Querying Device

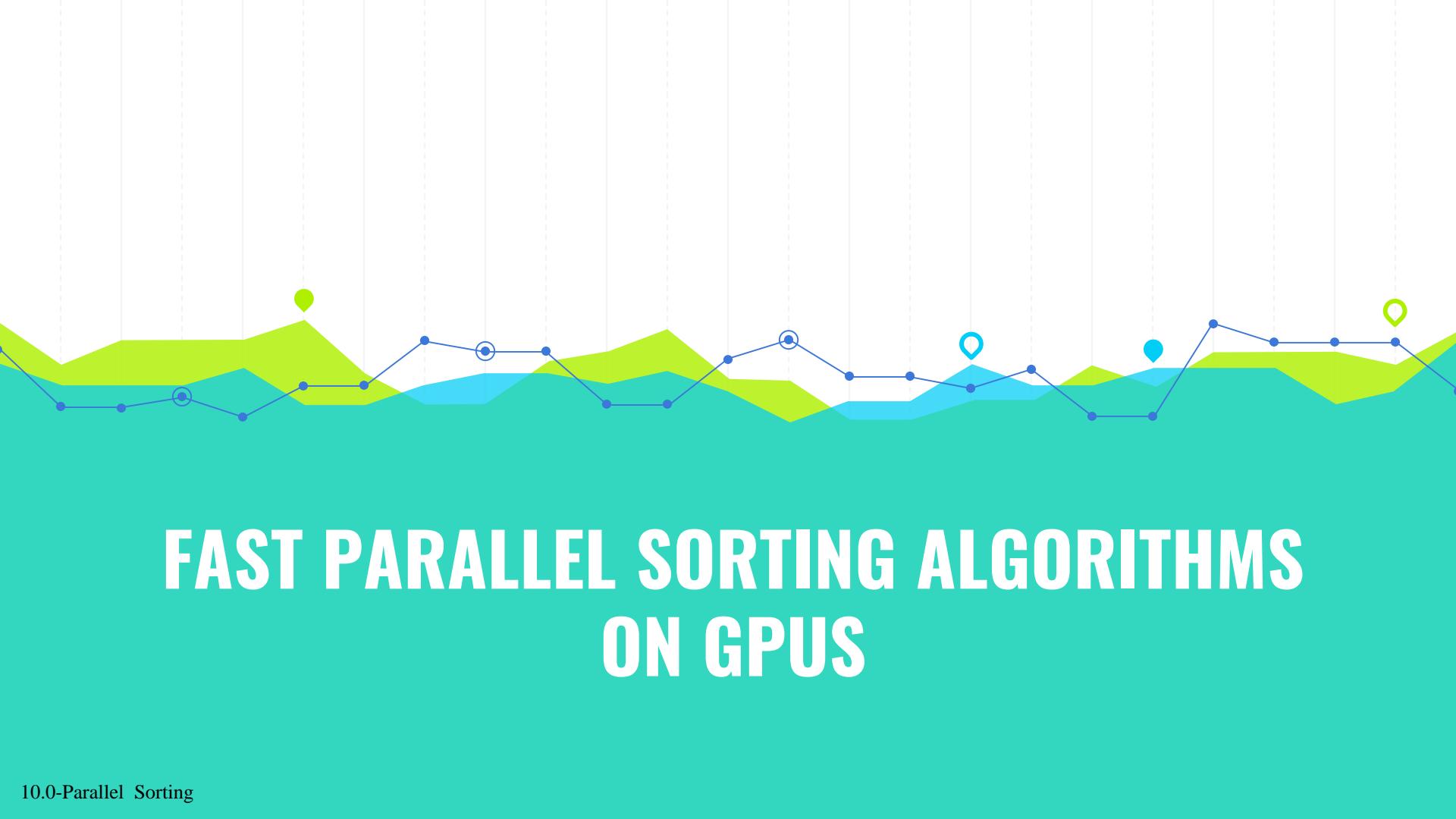
```
int main(void) {  
    cudaDeviceProp prop;  
    int dev;  
  
    cudaGetDevice( &dev );  
    printf( "ID of current CUDA device: %d\n", dev );  
    memset( &prop, 0, sizeof( cudaDeviceProp ) );  
    prop.major = 1;  
    prop.minor = 3;  
    cudaChooseDevice( &dev, &prop );  
    printf( "ID of device closest to revision 1.3: %d\n", dev );  
    cudaSetDevice( dev );  
    return 0;  
}
```

**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 6 – Fall 2021**

## **Question 1**

Write a C program which do the following:

- a. Displays the number of devices available on your computer.
- b. The number of multiprocessors on every device.



# FAST PARALLEL SORTING ALGORITHMS ON GPUS

# Introduction

*Odd- Even sort*

*Rank sort*

*Bitonic sort*

1

# Odd-Even Sort



# Odd-Even Sort

- The odd-even sort is a parallel sorting algorithm and is based on bubble-sort technique.
- Adjacent pairs of items in an array are exchanged if they are found to be out of order.
- Total running time for this technique is  $O(\log 2N)$ .



# Odd-Even Sort Algorithm

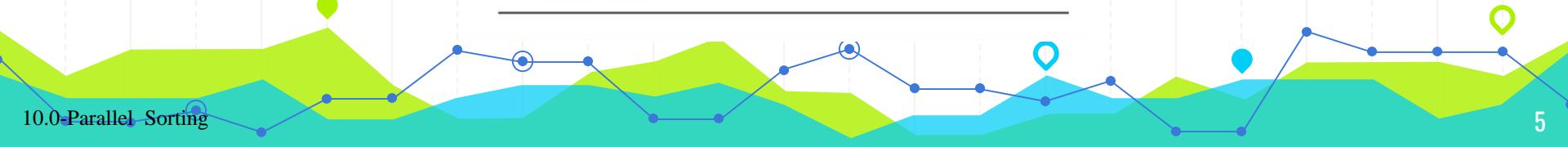
---

## Algorithm 1 Odd Even Sort

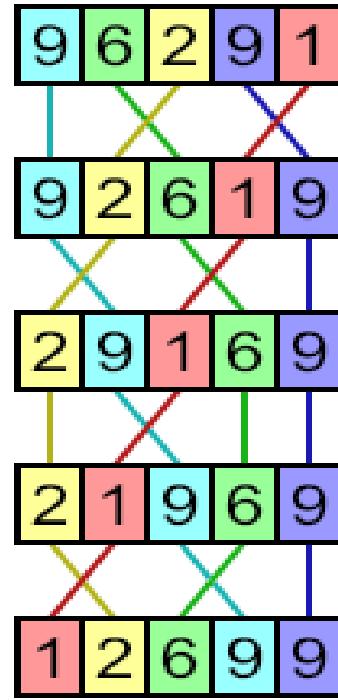
---

```
for  $k = 1 \rightarrow N/2$  do
    do parallel
        if  $i > i + 1 \wedge i \% 2 != 0$  then
            swap  $i, i + 1$ 
        end if
    end parallel
    do parallel
        if  $i > i + 1 \wedge i \% 2 == 0$  then
            swap  $i, i + 1$ 
        end if
    end parallel
end for
```

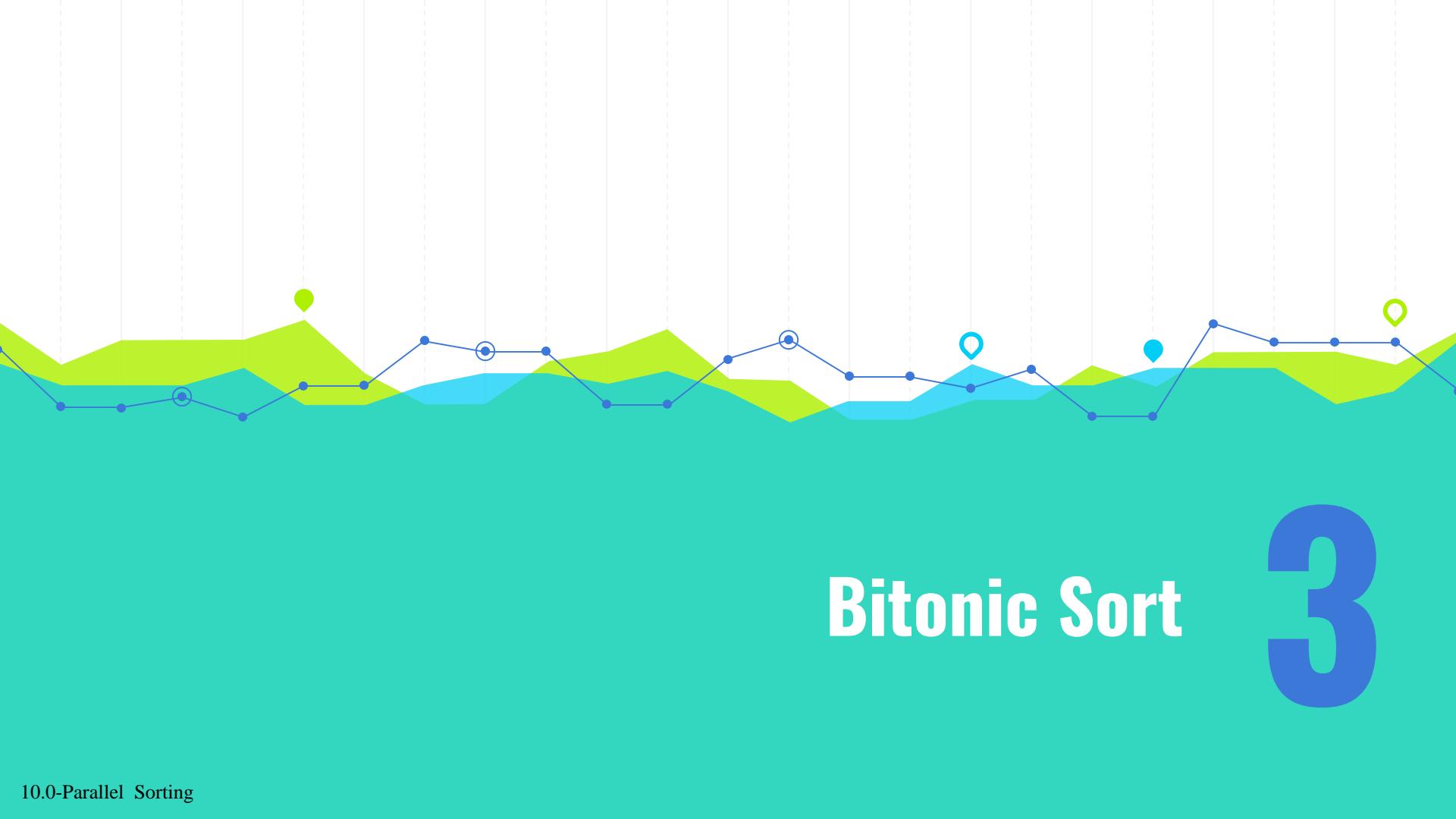
---



# Odd-Even Sort Algorithm



# Bitonic Sort 3



## Sorting Networks: Bitonic Sort

- A bitonic sorting network sorts  $n$  elements in  $\Theta(\log^2 n)$  time.
- A bitonic sequence has two tones - increasing and decreasing, or vice versa. Any cyclic rotation of such networks is also considered bitonic.
- $\langle 1,2,4,7,6,0 \rangle$  is a bitonic sequence, because it first increases and then decreases.  $\langle 8,9,2,1,0,4 \rangle$  is another bitonic sequence.
- The kernel of the network is the rearrangement of a bitonic sequence into a sorted sequence.

## Sorting Networks: Bitonic Sort

- Let  $s = \langle a_0, a_1, \dots, a_{n-1} \rangle$  be a bitonic sequence such that  $a_0 \leq a_1 \leq \dots \leq a_{n/2-1}$  and  $a_{n/2} \geq a_{n/2+1} \geq \dots \geq a_{n-1}$ .
- Consider the following subsequences of  $s$ :

$$s_1 = \langle \min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \dots, \min\{a_{n/2-1}, a_{n-1}\} \rangle$$

$$s_2 = \langle \max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \dots, \max\{a_{n/2-1}, a_{n-1}\} \rangle$$

(1)

Note that  $s_1$  and  $s_2$  are both bitonic and each element of  $s_1$  is less than every element in  $s_2$ .

- We can apply the procedure recursively on  $s_1$  and  $s_2$  to get the sorted sequence.

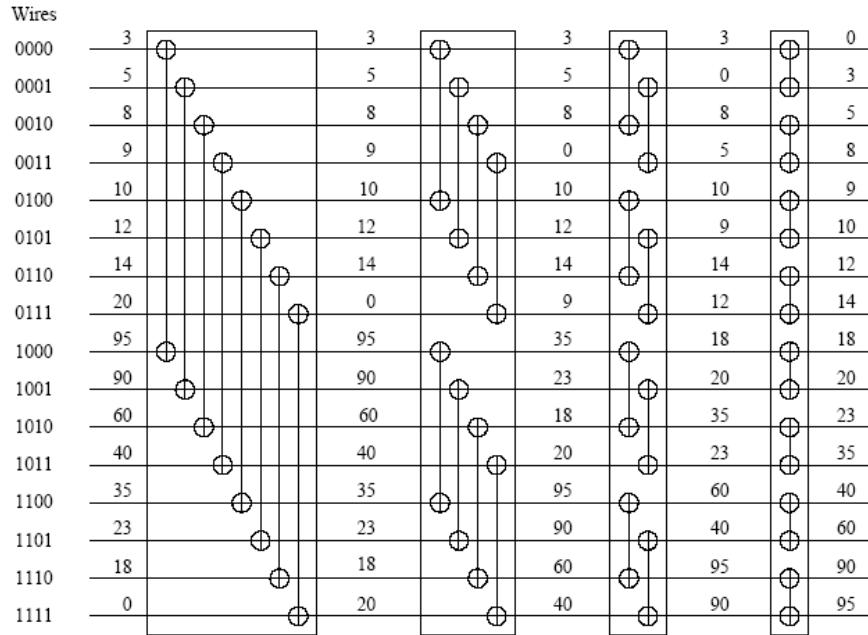
Original sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

Merging a 16-element bitonic sequence through a series of  $\log 16$  bitonic splits.

## Sorting Networks: Bitonic Sort

- We can easily build a sorting network to implement this bitonic merge algorithm.
- Such a network is called a *bitonic merging network*.
- The network contains  $\log n$  columns. Each column contains  $n/2$  comparators and performs one step of the bitonic merge.
- We denote a bitonic merging network with  $n$  inputs by  $\oplus\text{BM}[n]$ .
- Replacing the  $\oplus$  comparators by  $\ominus$  comparators results in a decreasing output sequence; such a network is denoted by  $\ominus\text{BM}[n]$ .

# Sorting Networks: Bitonic Sort



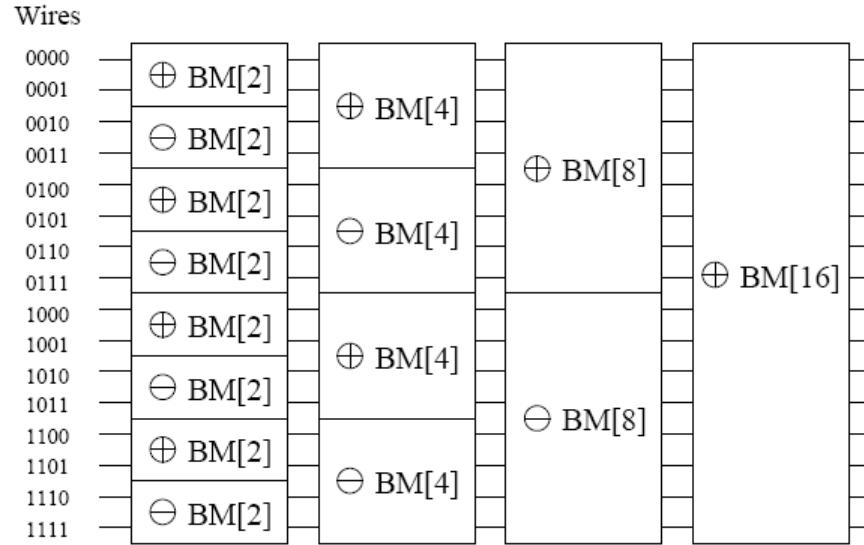
A bitonic merging network for  $n = 16$ . The entire figure represents a  $\oplus\text{BM}[16]$  bitonic merging network. The network takes a bitonic sequence and outputs it in sorted order.

## Sorting Networks: Bitonic Sort

How do we sort an unsorted sequence using a bitonic merge?

We must first build a single bitonic sequence from the given sequence.

- A sequence of length 2 is a bitonic sequence.
- A bitonic sequence of length 4 can be built by sorting the first two elements using  $\oplus\text{BM}[2]$  and next two, using  $\ominus\text{BM}[2]$ .
- This process can be repeated to generate larger bitonic sequences.



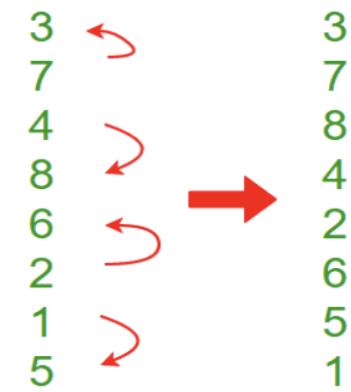
A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example,  $\oplus \text{BM}[k]$  and  $\ominus \text{BM}[k]$  denote bitonic merging networks of input size  $k$  that use  $\oplus$  and  $\ominus$  comparators, respectively. The last merging network ( $\oplus \text{BM}[16]$ ) sorts the input. In this example,  $n = 16$ .

# Bitonic Sequence

● Step #1

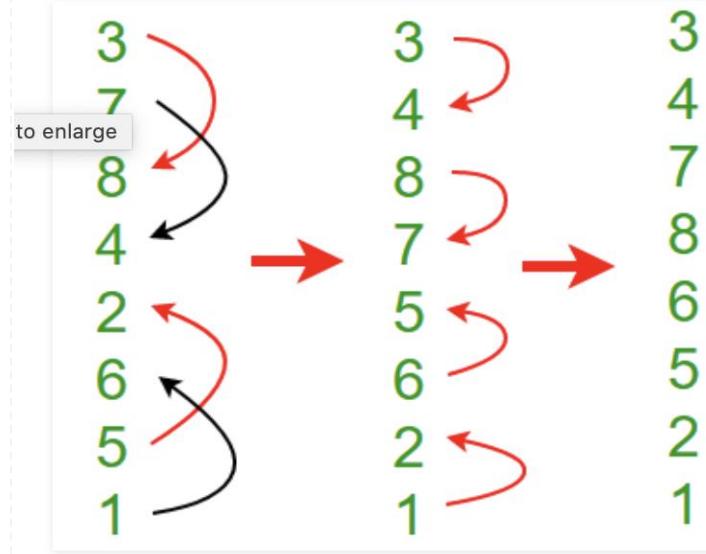
$$\begin{array}{c} a \curvearrowleft \\ b \end{array} = \frac{\text{Min}(a,b)}{\text{Max}(a,b)}$$

$$\begin{array}{c} a \curvearrowleft \\ b \end{array} = \frac{\text{Max}(a,b)}{\text{Min}(a,b)}$$

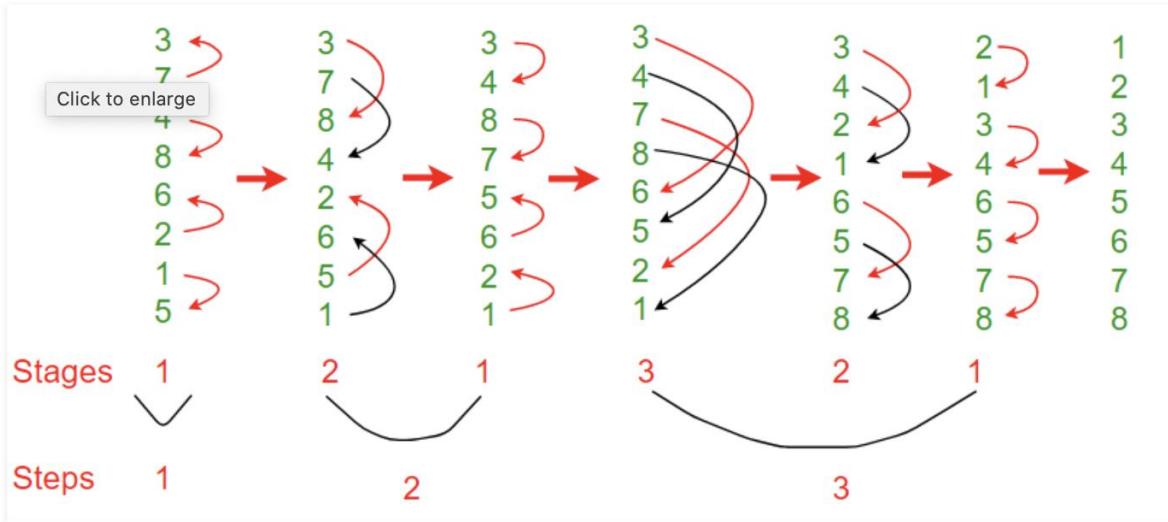


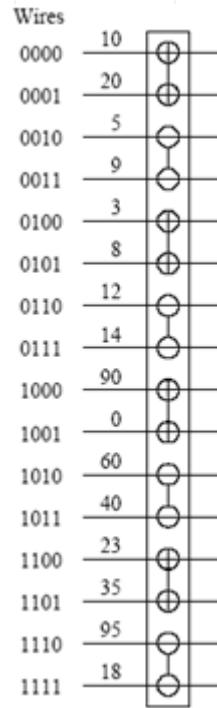
# Bitonic Sequence

● Step #2

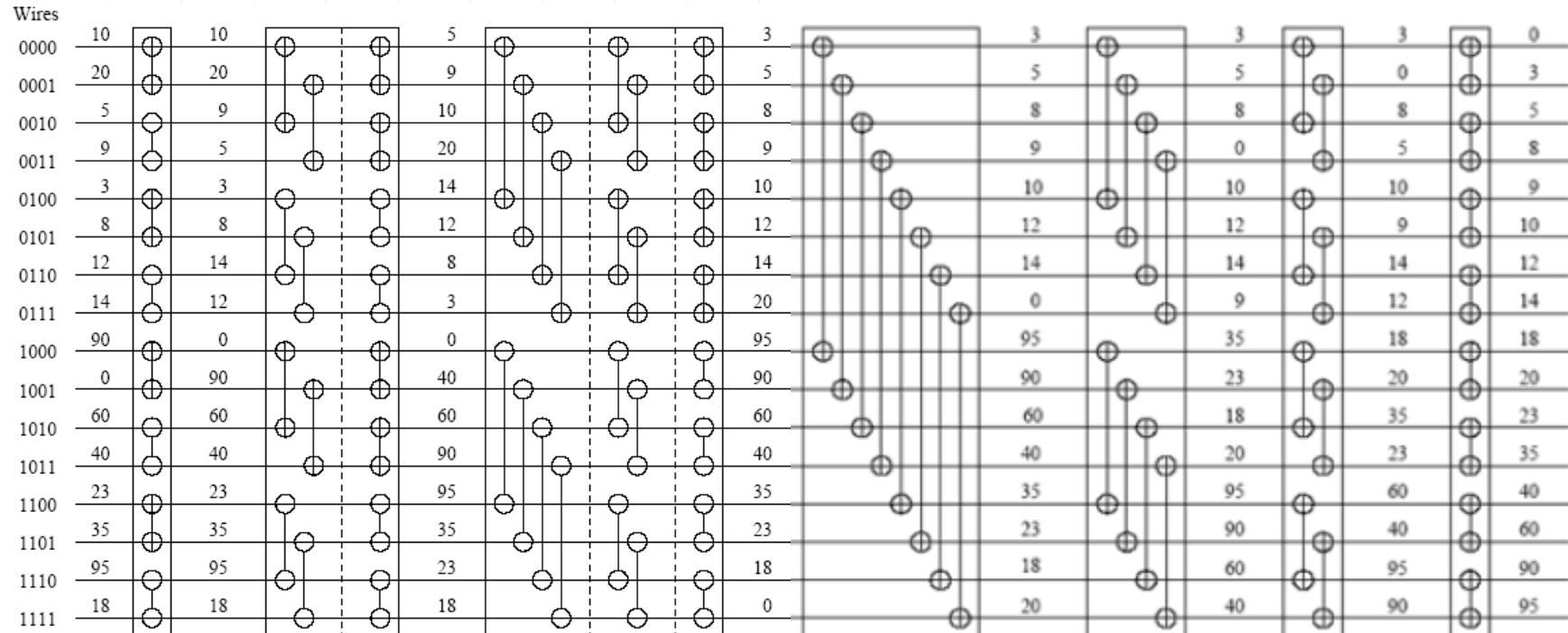


# Bitonic Sequence





# Sorting Networks: Bitonic Sort



# Bitonic Sort

- The sequence of comparisons is data-independent makes it one of the fastest and suitable parallel sorting algorithms.
- In the first step it makes the arbitrary sequence in to bitonic sequence.
- In the second step the bitonic sequence is sorted.



# Conclusion

- Sorting is one of the most fundamental problems in computer science, as it is used in most software applications.
- This presentation shows different parallel sorting algorithms on GPUs.
- The performance is affected mainly by : nature of algorithm and hardware platforms.



**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Tutorial No 8 – Fall 2021**

## Question 1

- [1] Give the number of steps that are required to sort the elements of the array.
- [2] Give the size of bitonic sequences in every step.
- [3] Give the bitonic sequences (just give an interval [b, e] where b is the index of the first cell of the sequence and the e is the index of the last cell of the sequence) that are processed in every step.
- [4] Specify, in every step, for every bitonic sequence whether the sequence is sorted in an ascending (+BM) or a descending (-BM) way.

## Question 2

Let's consider an array of integers of size 8. We would like to sort the elements of the array in an ascending way using the Bitonic sort algorithm. We focus on step No 1. Give the following information related to step 1.

- [1] Give the IDs of threads involved in step 1.
- [2] Give for every thread the bitonic sequence (just give an interval [b, e] where b is the index of the first cell of the sequence and the e is the index of the last cell of the sequence) that the thread is processing.
- [3] Specify for every thread whether the corresponding sequence is sorted in an ascending (+BM) or a descending (-BM) way.

## Question 3

Let's consider an array of integers of size 8. We would like to sort the elements of the array in an ascending way using the Bitonic sort algorithm. We focus on step No 2. Give the following information related to every stage **i** of step 2:

- [1] Give the IDs of threads involved in stage **i**.
- [2] Infer the condition that is satisfied by the involved thread in stage **i**.
- [3] Give for every thread, involved in stage **i**, the bitonic sequence (just give an interval [b, e] where b is the index of the first cell of the sequence and the e is the index of the last cell of the sequence) that the thread is processing.
- [4] Specify for every thread whether the corresponding sequence is sorted in an ascending (+BM) or a descending (-BM) way.

$$\begin{array}{l}
 \text{Step } 1 \quad \left\{ \begin{array}{c} 2.1 \quad 2.2 \\ 9 - 5 \quad 5 - 5 \\ \hline 4 \end{array} \right\} \quad \left\{ \begin{array}{c} 3.1 \quad 3.2 \\ 5 - 3 \quad 5 - 5 \\ \hline 2 \end{array} \right\} \\
 \text{Step } 2 \quad \left\{ \begin{array}{c} 5 - 9 \quad 8 - 8 \\ 20 - 10 \quad 8 - 8 \\ \hline 10 \end{array} \right\} \\
 \text{Step } 3 \quad \left\{ \begin{array}{c} 9 - 10 \quad 3 - 9 - 9 \\ 5 - 20 - 20 \quad 3 - 9 - 9 \\ \hline 14 - 10 - 10 \end{array} \right\} \\
 \text{Step } 4 \quad \left\{ \begin{array}{c} 3 - 14 \quad 14 - 14 - 10 - 10 \\ 8 - 12 \quad 12 - 12 - 12 \\ \hline 12 - 3 - 8 - 10 - 14 - 14 \end{array} \right\} \\
 \text{Step } 5 \quad \left\{ \begin{array}{c} 12 - 8 - 3 - 20 - 20 - 20 \\ 0 - 0 - 0 - 95 - 95 - 95 \end{array} \right\} \\
 \text{Step } 6 \quad \left\{ \begin{array}{c} 40 - 40 - 40 - 90 - 90 - 90 \\ 60 - 60 - 60 - 60 - 60 - 60 \end{array} \right\} \\
 \text{Step } 7 \quad \left\{ \begin{array}{c} 60 - 60 - 60 - 60 - 60 - 60 \\ 90 - 90 - 90 - 90 - 90 - 90 \end{array} \right\} \\
 \text{Step } 8 \quad \left\{ \begin{array}{c} 90 - 40 - 40 - 40 \\ 23 - 23 - 23 - 23 \end{array} \right\} \\
 \text{Step } 9 \quad \left\{ \begin{array}{c} 0 - 23 - 23 - 23 \\ 35 - 35 - 35 - 35 \end{array} \right\} \\
 \text{Step } 10 \quad \left\{ \begin{array}{c} 0 - 35 - 35 - 35 \\ 95 - 95 - 95 - 95 \end{array} \right\} \\
 \text{Step } 11 \quad \left\{ \begin{array}{c} 18 - 18 - 18 - 18 \\ 18 - 18 - 18 - 18 \end{array} \right\}
 \end{array}$$

Step N°	1	2	
stage	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{2}{4}$
SeqL = 2	$2^{\frac{1}{2}} = 2$	$\frac{1}{4}$	$\frac{1}{4}$
$N = \frac{\text{SeqL}}{2}$	$\frac{2}{2}$	$\frac{1}{4}$	$\frac{2}{4}$
shift = $\frac{N}{2}$	1	2	1

$(idx \% N) < shift \rightarrow \text{true} \Rightarrow \underline{idx} \text{ is working.}$

$(idx, idx + shift) \rightarrow \text{False} \Rightarrow \underline{idx} \text{ is idle.}$

$(idx / SeqL)$  is an odd number  $\Rightarrow \textcircled{-}$

is an even number  $\Rightarrow \textcircled{+}$

SeqL 2<sup>Step</sup>

N 2<sup>Step = Stage + 1</sup>

Shift 2<sup>Step - Stage</sup>

active |dead| % N < Shift

ascending ~~idx~~ / SeqL % 2 ==

```
51  
52     odd = ! odd; //switch phase of sorting  
53 }  
54  
55 __syncthreads();  
56  
57 //Store this phase's in[] array to out[] array  
58 if ( idx < size )  
59     out[idx] = in[idx];  
60  
61 }  
62  
63  
64 int main(void)  
65 {  
66     int i;
```

Press ESC or double-click to exit full screen mode

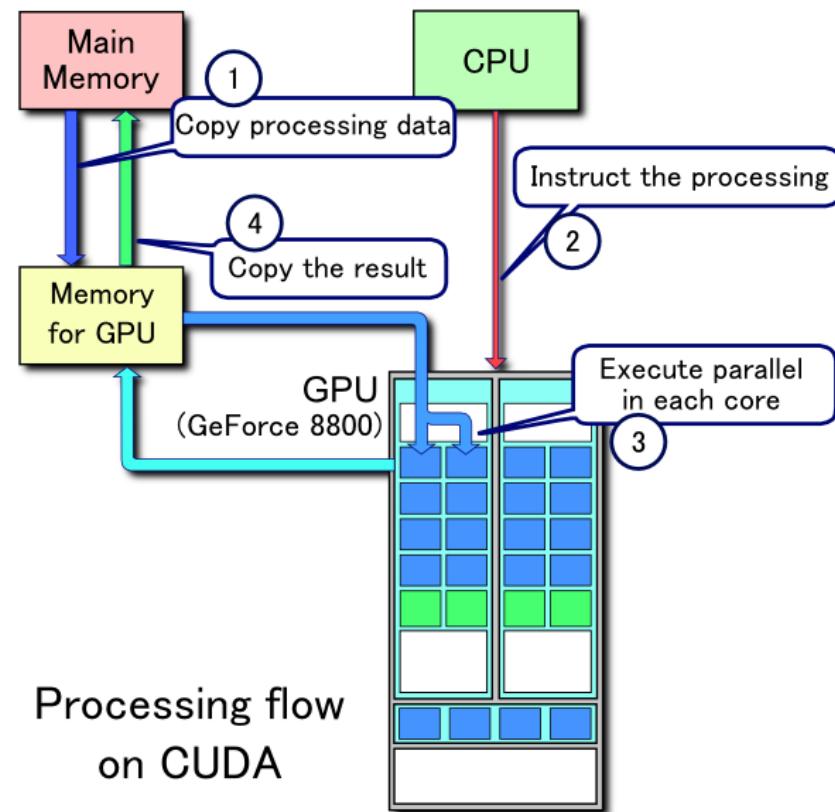
```
31     else{
32         swappedeven=false;
33
34         __syncthreads();
35     }
36
37     if( idx % 2 == 0 && idx < size-1 ){
38         if ( in[idx] > in[idx+1] ){
39             temp=in[idx];
40             in[idx]=in[idx+1];
41             in[idx+1]=temp;
42             swappedeven=true;
43         }
44     }
45
46     __syncthreads();
47
48 //if there are no swaps in odd phase as well as even phase then break
49 // (which means all sorting is done)
50 // !(false) => true
51     if( !( swappedodd || swappedeven ) )
52         break;
```

```
4 __global__ void sortKernel(int *in, int *out, int size){  
5     // shared for shared memory  
6     __shared__ bool swappedodd;  
7     __shared__ bool swappedeven;  
8  
9     bool odd = true;  
10    int temp;  
11    int idx = threadIdx.x + blockDim.x * blockIdx.x;  
12  
13    while(true){  
14        __syncthreads();  
15  
16        if(odd == true){  
17            swappeodd=false;  
18  
19            __syncthreads();  
20  
21            if( idx %2 == 1 && idx < size - 1 ){  
22                if ( in[idx] > in[idx+1] ){  
23                    //BUBBLE SORT LOGIC  
24                    temp= in[idx];  
25                    in[idx]=in[idx+1];  
26                    in[idx+1]=temp;  
27                }  
28            }  
29        }  
30        __syncthreads();  
31    }  
32}
```

# Dynamic Parallelism

# Compute Unified Device Architecture

- Hybrid CPU/GPU Code
- Low latency code is running on CPU
  - Result immediately available
- High latency, high throughput code is running on GPU
  - Result on bus
  - GPU has many more cores than CPU



# Types of Parallelism

- Different Types of Parallelism
  - Task parallelism
    - Problem is divided to tasks, which are processed independently
  - Data parallelism
    - Same operation is performed over many data items
  - Pipeline parallelism
    - Data are flowing through a sequence (or oriented graph) of stages, which operate concurrently
  - Other types of parallelism
    - Event driven, ...

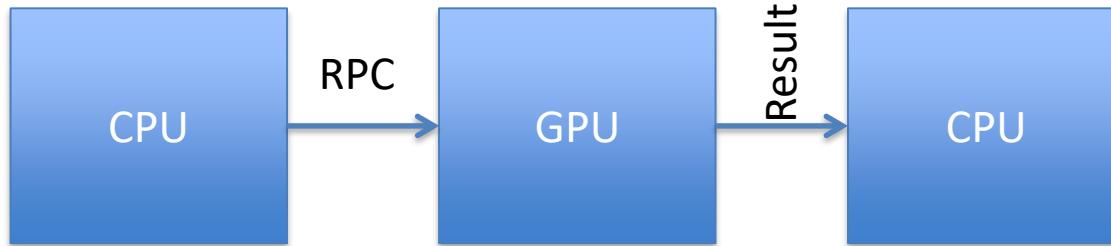
# GPU Execution Model

- Parallelism in GPU
  - Data parallelism
    - The same kernel is executed by many threads
    - Thread process one data item
  - Limited task parallelism
    - Multiple kernels executed simultaneously (since Fermi)
    - At most as many kernels as SMPs
  - But we do not have
    - Any means of kernel-wide synchronization (barrier)
    - Any guarantees that two blocks/kernels will actually run concurrently

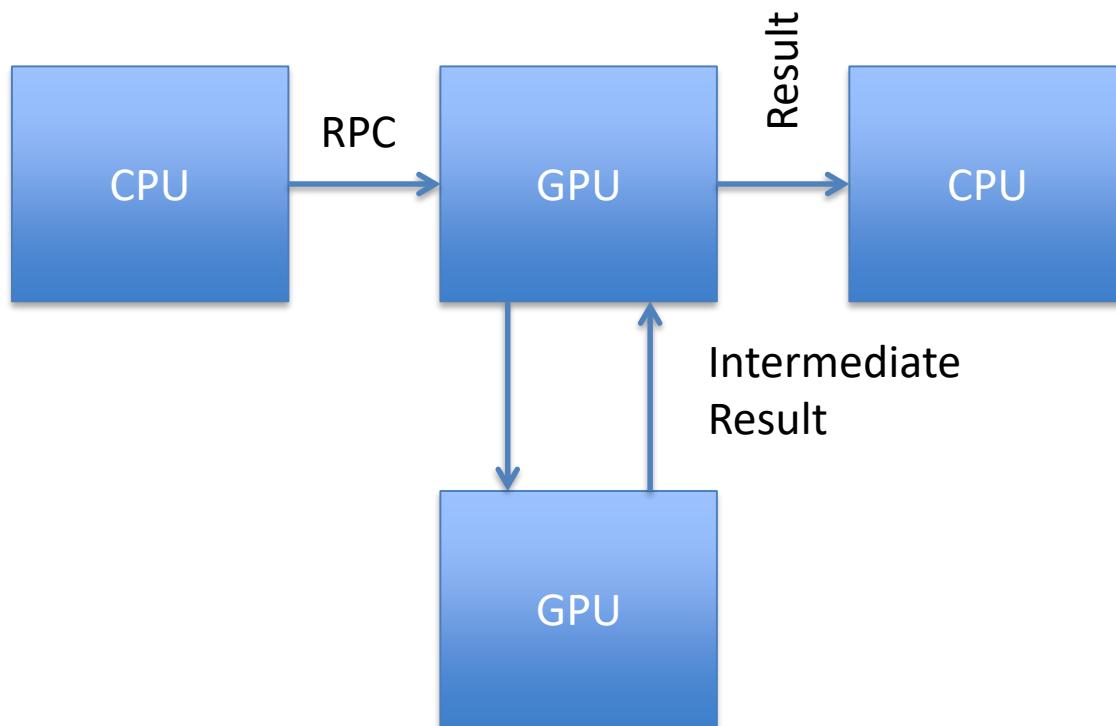
# What is Dynamic Parallelism

- The ability to launch new kernels from the GPU
  - Dynamically
    - based on run-time data
  - Simultaneously
    - from multiple threads at once
  - Independently
    - each thread can launch a different grid
- Introduced with CUDA 5.0 and compute capability 3.5 and up

# Execution Model (Overview)



*Fermi: Only CPU can generate GPU work*



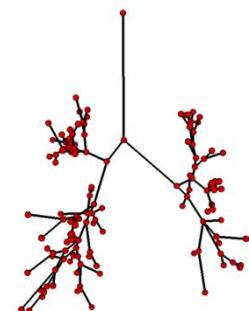
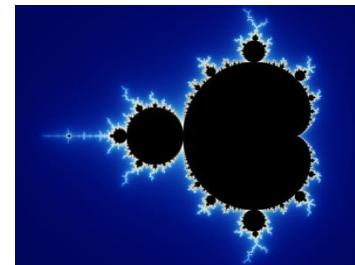
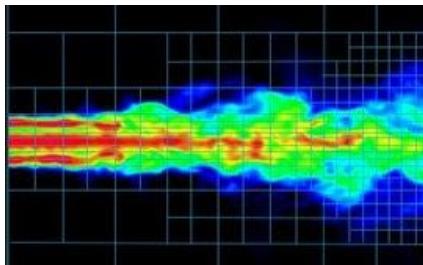
*Kepler: GPU can generate work for itself*

# Dynamic Parallelism

- Allows program flow to be controlled by GPU
- Allows recursion and subdivision of problems
- Interesting data is not uniformly distributed
- Dynamic parallelism can launch additional threads in interesting areas
- Allows higher resolution in critical areas without slowing down others

# Problematic Cases

- Unsuitable Problems for GPUs
  - Processing irregular data structures
    - Trees, graphs, ...
  - Regular structures with irregular processing workload
    - Difficult simulations, iterative approximations
  - Iterative tasks with explicit synchronization
  - Pipeline-oriented tasks with many simple stages

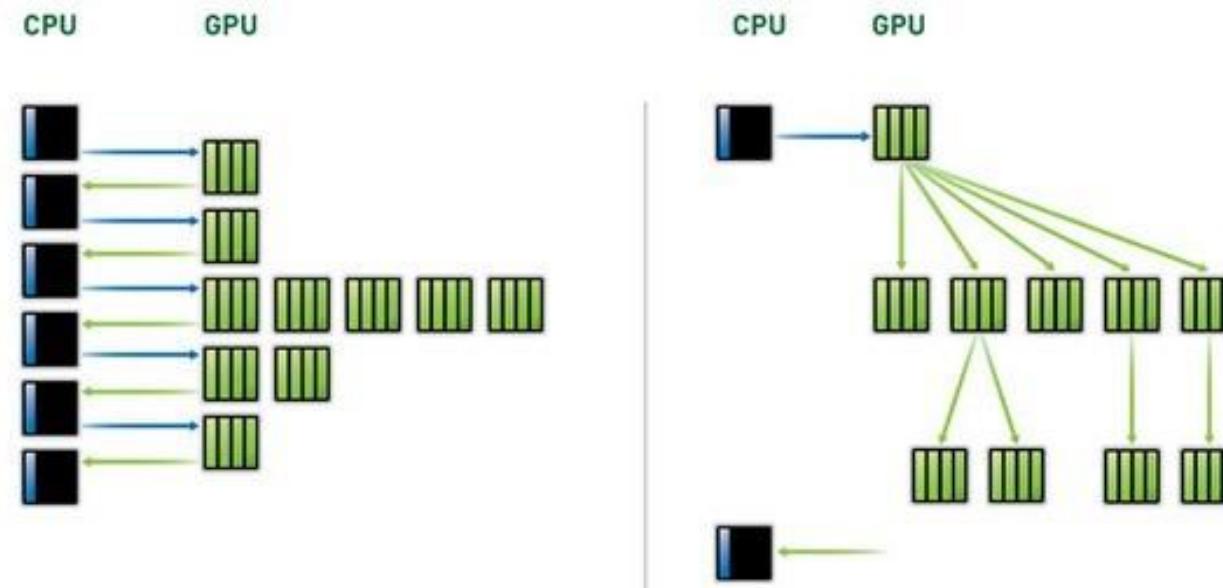


# Problematic Cases

- Solutions
  - Iterative kernel execution
    - Usually applicable only for cases when there are none or few dependencies between the subsequent kernels
    - The state (or most of it) is kept on the GPU
  - Mapping irregular structures to regular grids
    - May be too fine/coarse grained
    - Not always possible
  - 2-phase Algorithms
    - First phase determines the amount of work (items, ...)
    - Second phase process tasks mapped by first phase

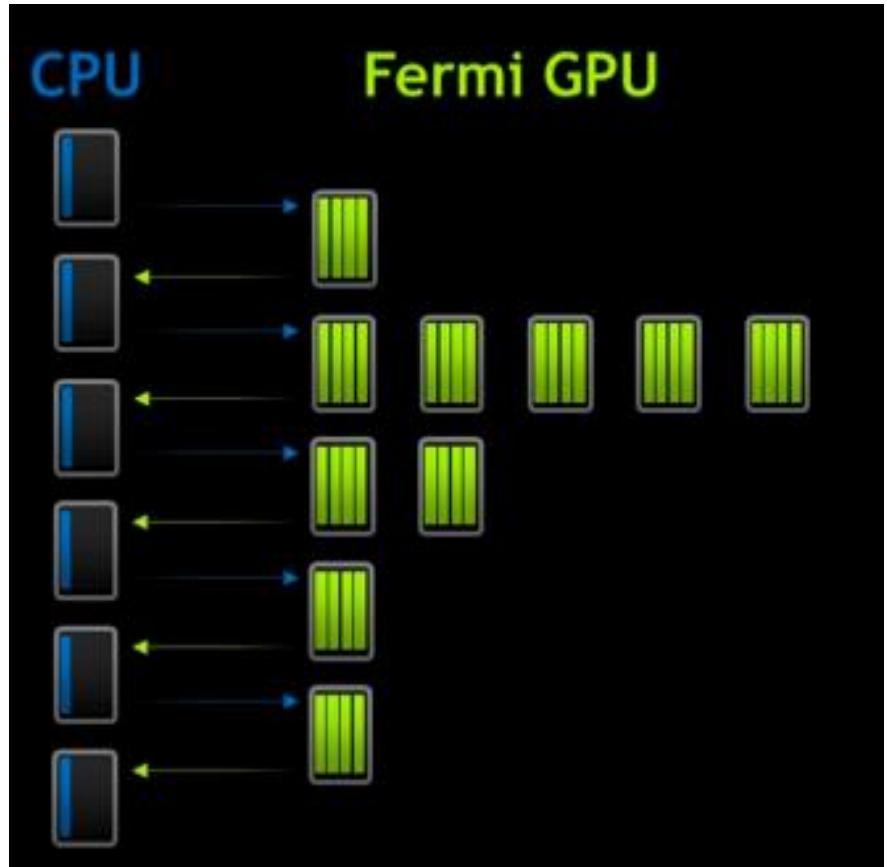
# Dynamic Parallelism

- Dynamic Parallelism Purpose
  - The device does not need to synchronize with host to issue new work to the device
  - Irregular parallelism may be expressed more easily

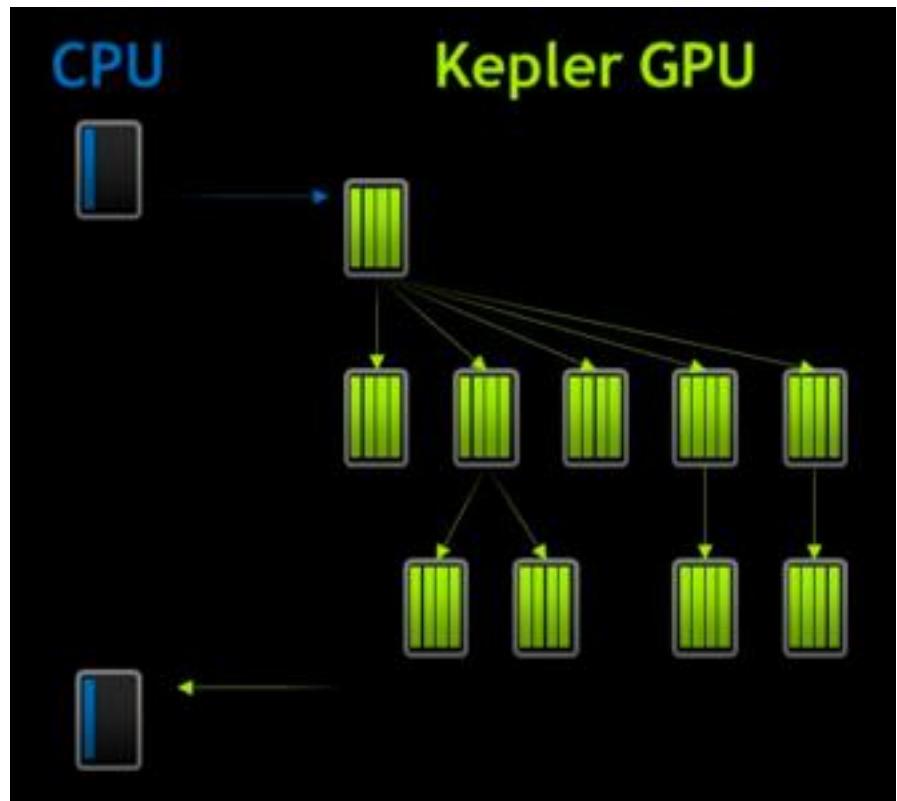


# Dynamic Parallelism

- Without Dynamic Parallelism
  - Data travels back and forth between the CPU and GPU many times.
  - This is because of the inability of the GPU to create more work on itself depending on the data.



- With Dynamic Parallelism:
  - GPU can generate work on itself based on intermediate results, without involvement of CPU.
  - Permits Dynamic Run Time decisions.
  - Leaves the CPU free to do other work, conserves power.

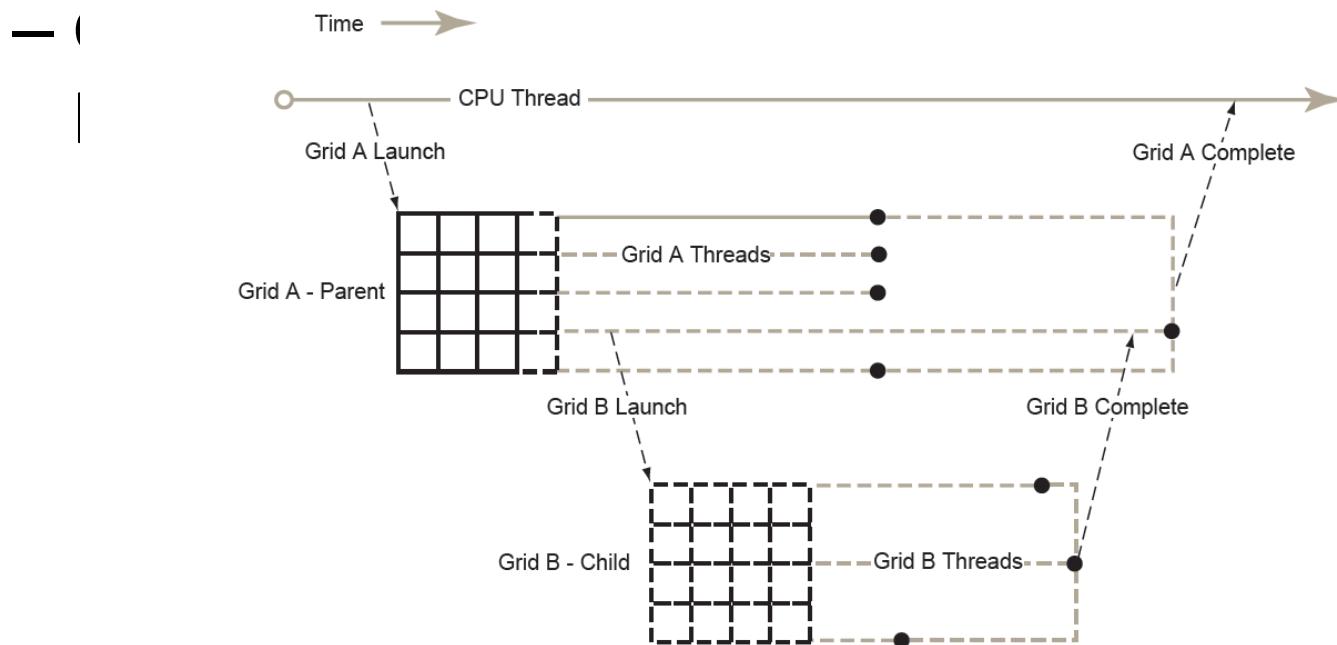


# Dynamic Parallelism

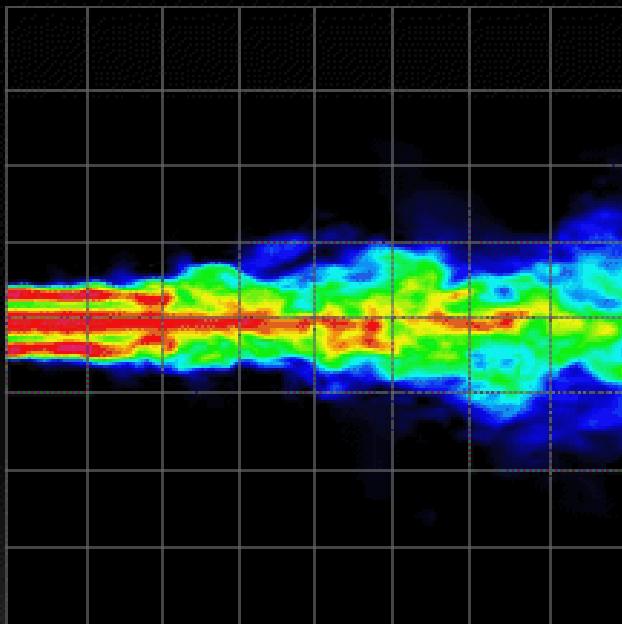
- How It Works
  - Portions of CUDA runtime are ported to device side
    - Kernel execution
    - Device synchronization
    - Streams, events, and async memory operations
  - Kernel launches are asynchronous
    - No guarantee the child kernel starts immediately
    - Synchronization points may cause context switch
      - Entire blocks are switched on a SMP
  - Block-wise locality of resources
    - Streams and events are shared in thread block

# Dynamic Parallelism

- CUDA Dynamic Parallelism
  - New feature presented in CC 3.5 (Kepler)

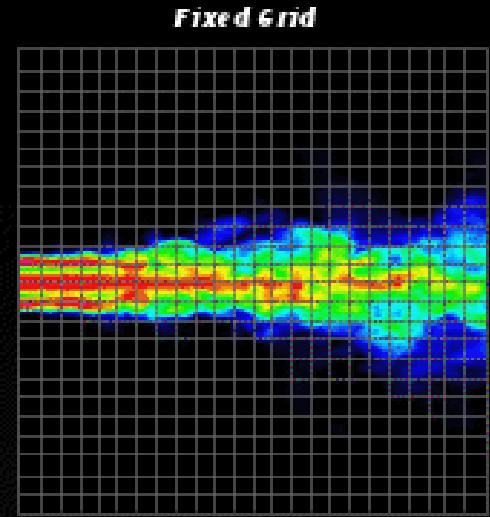


# Dynamic Work Generation



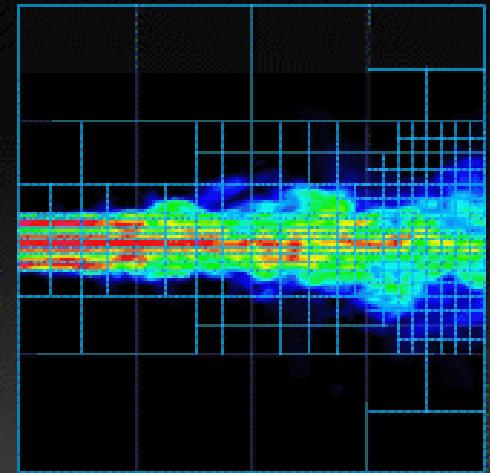
*Initial Grid*

*Statically assign conservative  
worst-case grid*



*Fixed Grid*

*Dynamically assign performance  
where accuracy is required*



*Dynamic Grid*

# Dynamic Parallelism

- Example

```
__global__ void child_launch(int *data) {
    data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        child_launch<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

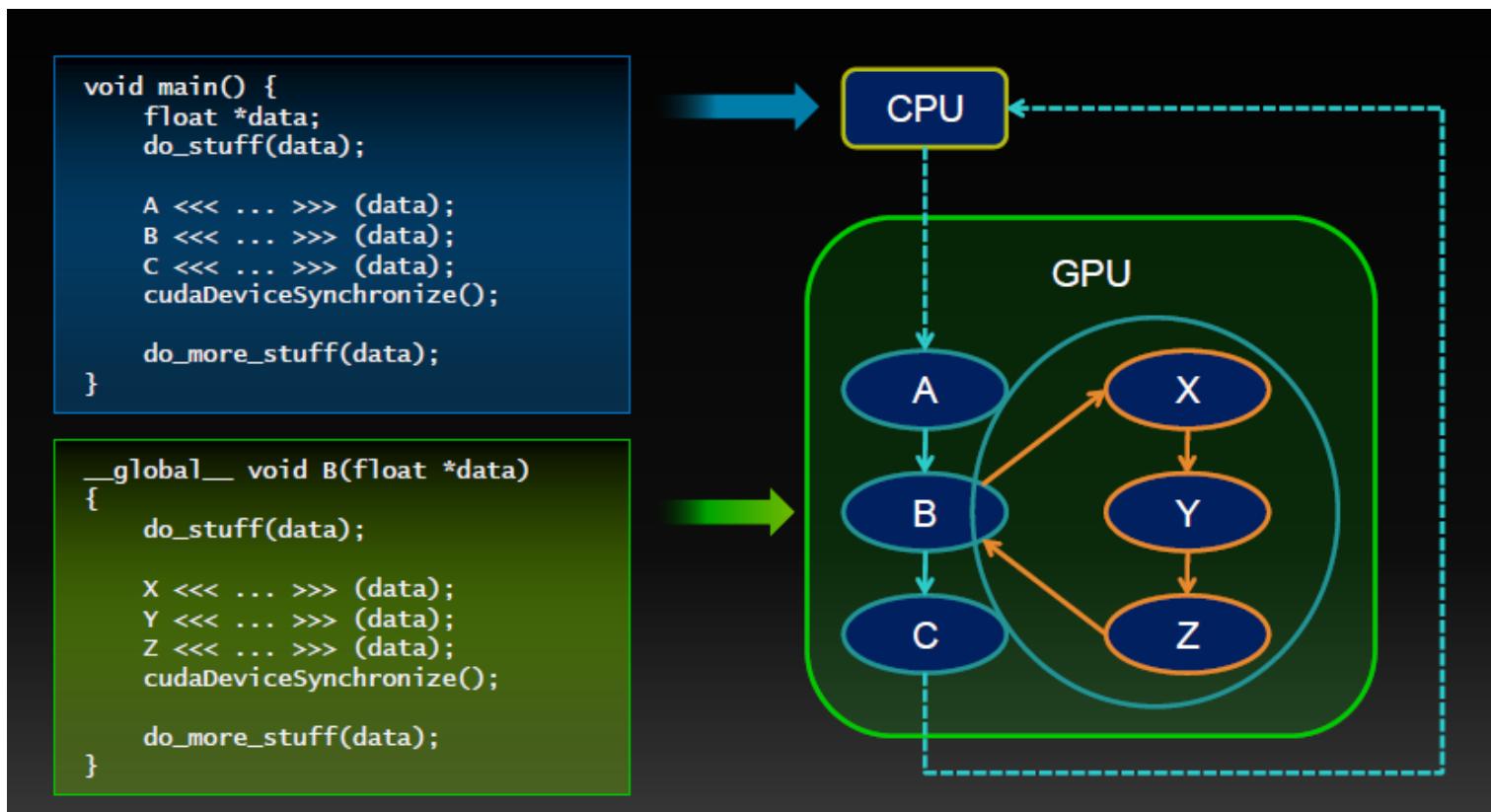
Thread 0 invokes a grid of child threads

Synchronization does not have to be invoked by all threads

Device synchronization does not synchronize threads in the block

# Dynamic Parallelism

- Nested Dependencies



Source: NVIDIA

# Dynamic Parallelism

- Scheduling can be controlled by streams
- Launched kernels may execute out-of-order within a stream
- Named streams can guarantee concurrency

# Dynamic Parallelism

- Nested Dependencies -  
`cudaDeviceSynchronize ()`
- Can be used inside a kernel
- Synchronizes all launches by any kernel in block
- Does NOT imply `__syncthreads()`!

# Dynamic Parallelism

- Kernel launch implies memory sync operation
- Child sees state at time of launch
- Parent sees child writes after sync
- Local and shared memory are private, cannot be shared with children

## Question

Let's consider the following parallel code:

```
__global__ void Kernel_A(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        Kernel_C<<< 1, 256 >>>(data);
        Kernel_D<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}
__global__ void Kernel_C(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        Kernel_E<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}
void host_launch(int *data) {
    kernel_A<<< 1, 256 >>>(data);
    kernel_B<<< 1, 256 >>>(data);
    cudaDeviceSynchronize();
}
```

1. Give and explain the order of execution of the given parallel nested kernels.
2. Explain the role of the `__syncthreads()` statements.
3. Explain the role of the `cudaDeviceSynchronize()` statements.



NVIDIA®

# Optimizing Parallel Reduction in CUDA

Mark Harris

NVIDIA Developer Technology

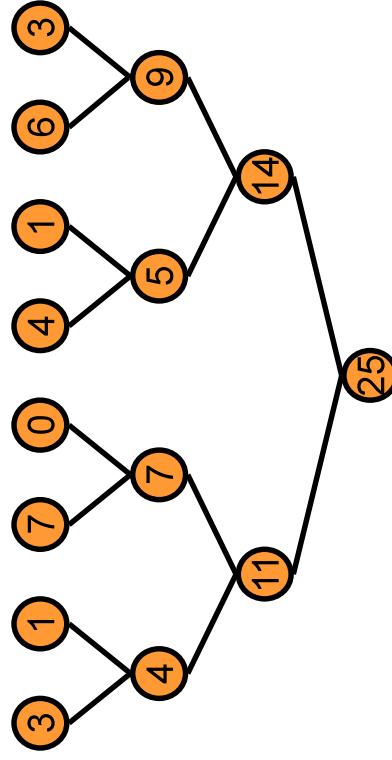
# Parallel Reduction



- Common and important data parallel primitive
- Easy to implement in CUDA
  - Harder to get it right
- Serves as a great optimization example
  - We'll walk step by step through 7 different versions
  - Demonstrates several important optimization strategies

# Parallel Reduction



- Tree-based approach used within each thread block
  - A diagram illustrating a parallel reduction using a tree-based approach. It shows a binary tree structure where each node contains a value. The tree has 7 levels:
    - Level 0: 3
    - Level 1: 1, 7
    - Level 2: 0, 4, 7
    - Level 3: 1, 4, 5
    - Level 4: 6, 9, 11
    - Level 5: 14
    - Level 6: 25Each node is connected to its parent and children via black lines.
- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array
- But how do we communicate partial results between thread blocks?

# Problem: Global Synchronization

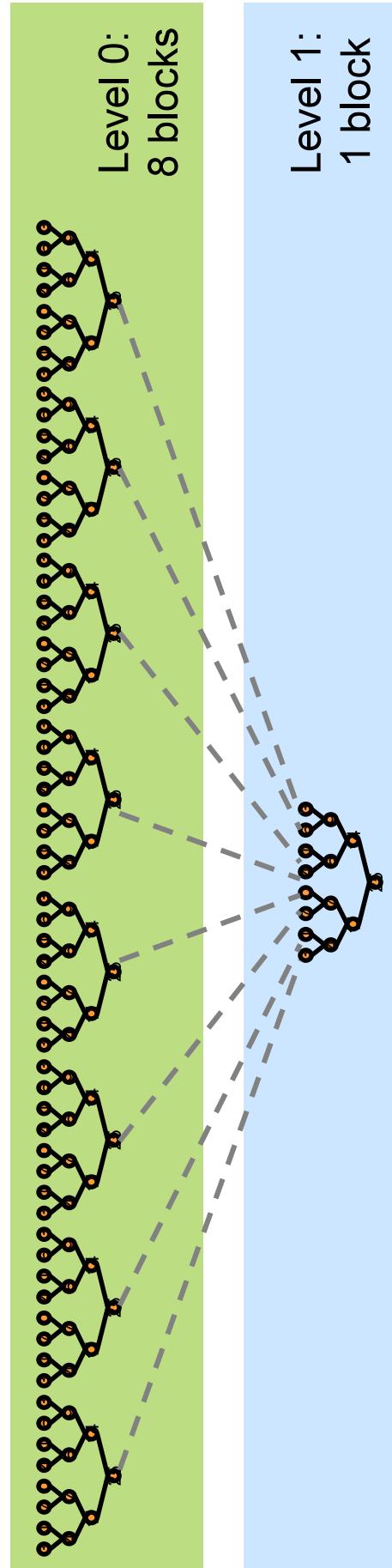


- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors \* # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

# Solution: Kernel Decomposition



- Avoid global sync by decomposing computation into multiple kernel invocations



- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation



# Reduction #1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];
```

// each thread loads one element from global to shared mem

```
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
```

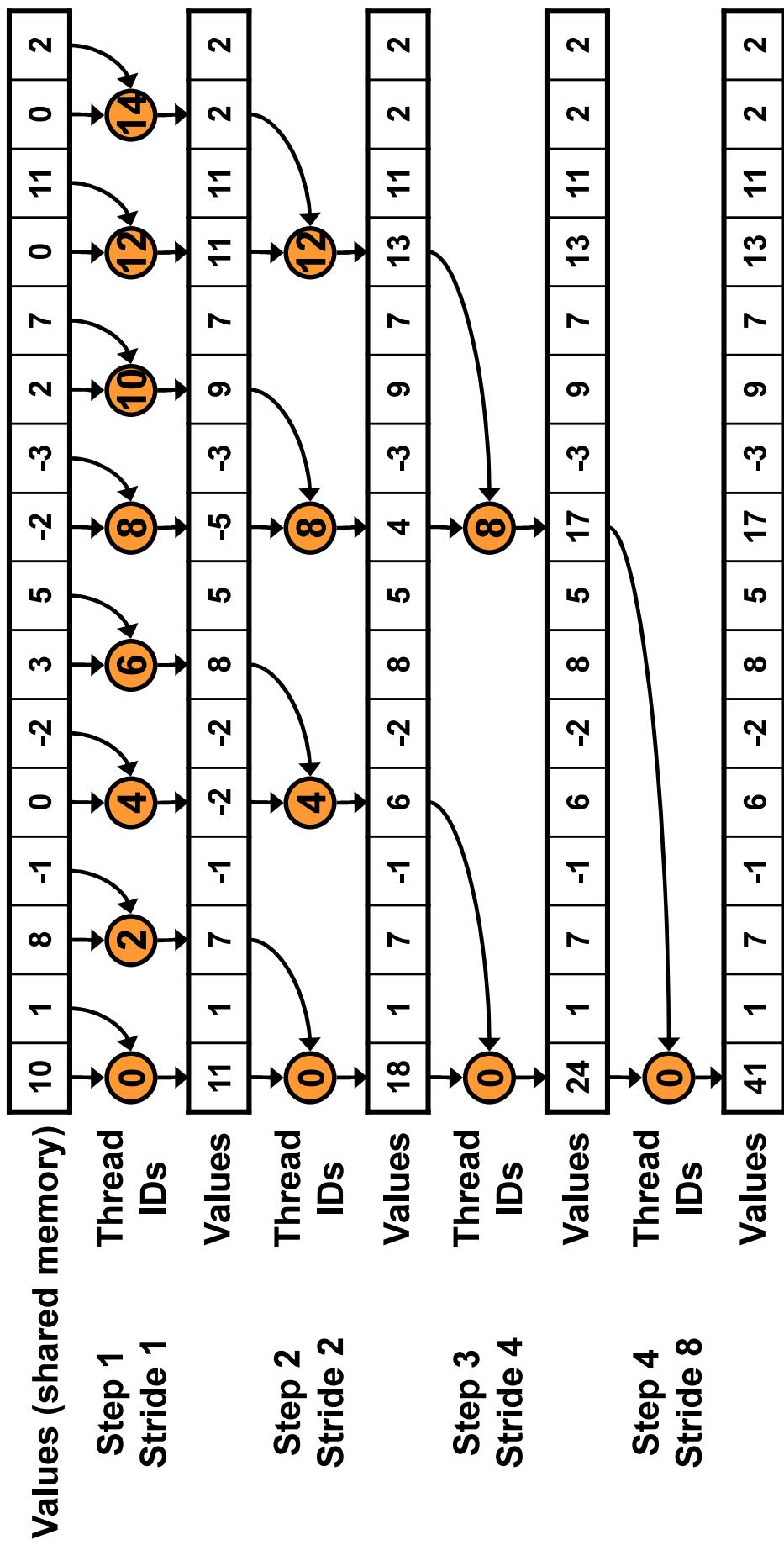
// do reduction in shared mem

```
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
```

// write result for this block to global mem

```
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Parallel Reduction: Interleaved Addressing



# Reduction #1: Interleaved Addressing

```
global __ void reduce1(int *g_idata, int *g_odata) {
extern __ shared __ int sdata[];
```

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

```
// do reduction in shared mem
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

**Problem:** highly divergent  
warps are very inefficient, and  
% operator is very slow

```
// write result for this block to global mem
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Performance for 4M element reduction



	Time ( $2^{22}$ ints)	Bandwidth
<b>Kernel 1:</b> interleaved addressing with divergent branching	8.054 ms	2.083 GB/s

Note: Block Size = 128 threads for all tests

## Reduction #2: Interleaved Addressing

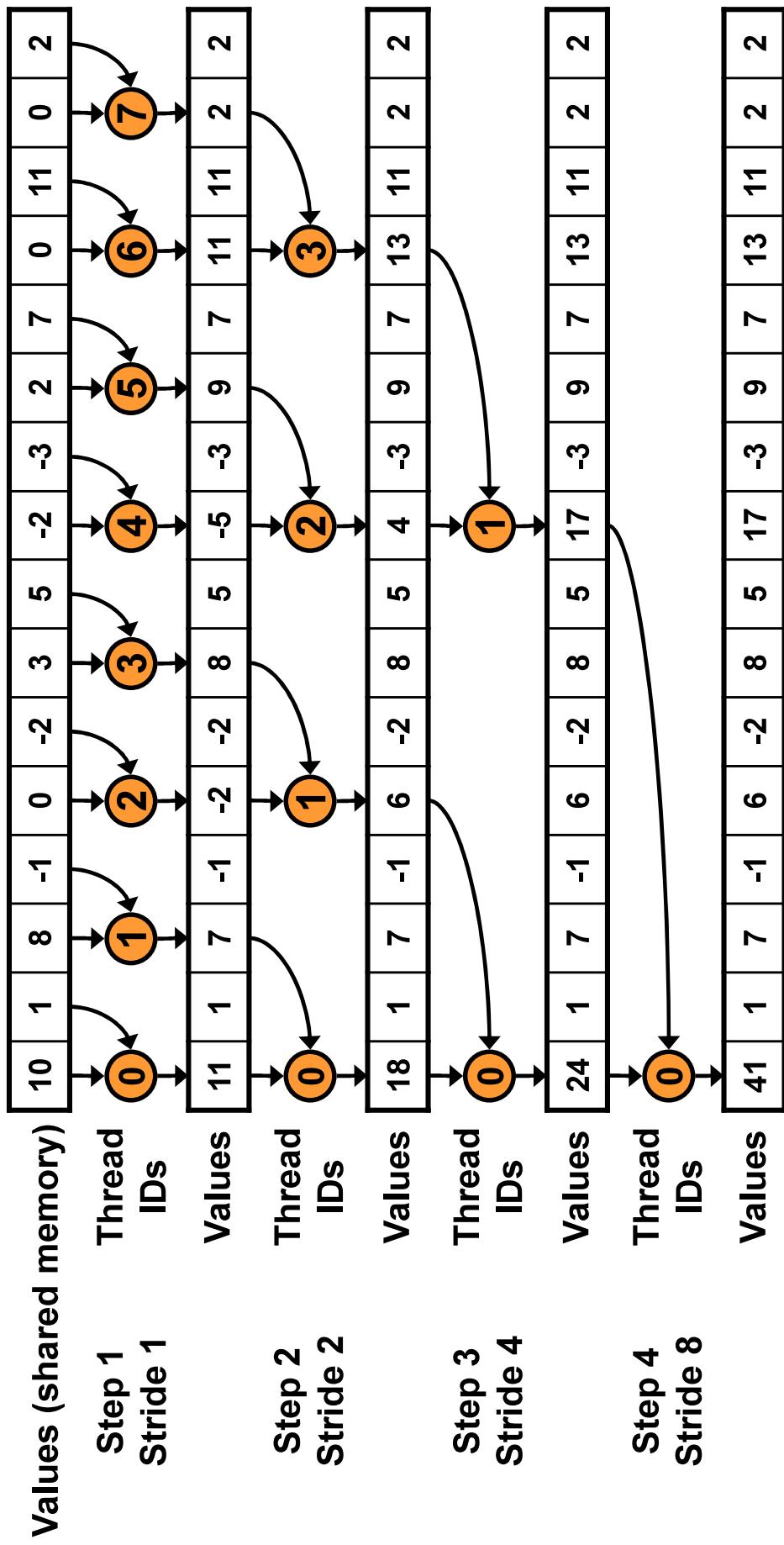
Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    if (tid % (2*s) == 0) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

With **strided index** and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```

# Parallel Reduction: Interleaved Addressing



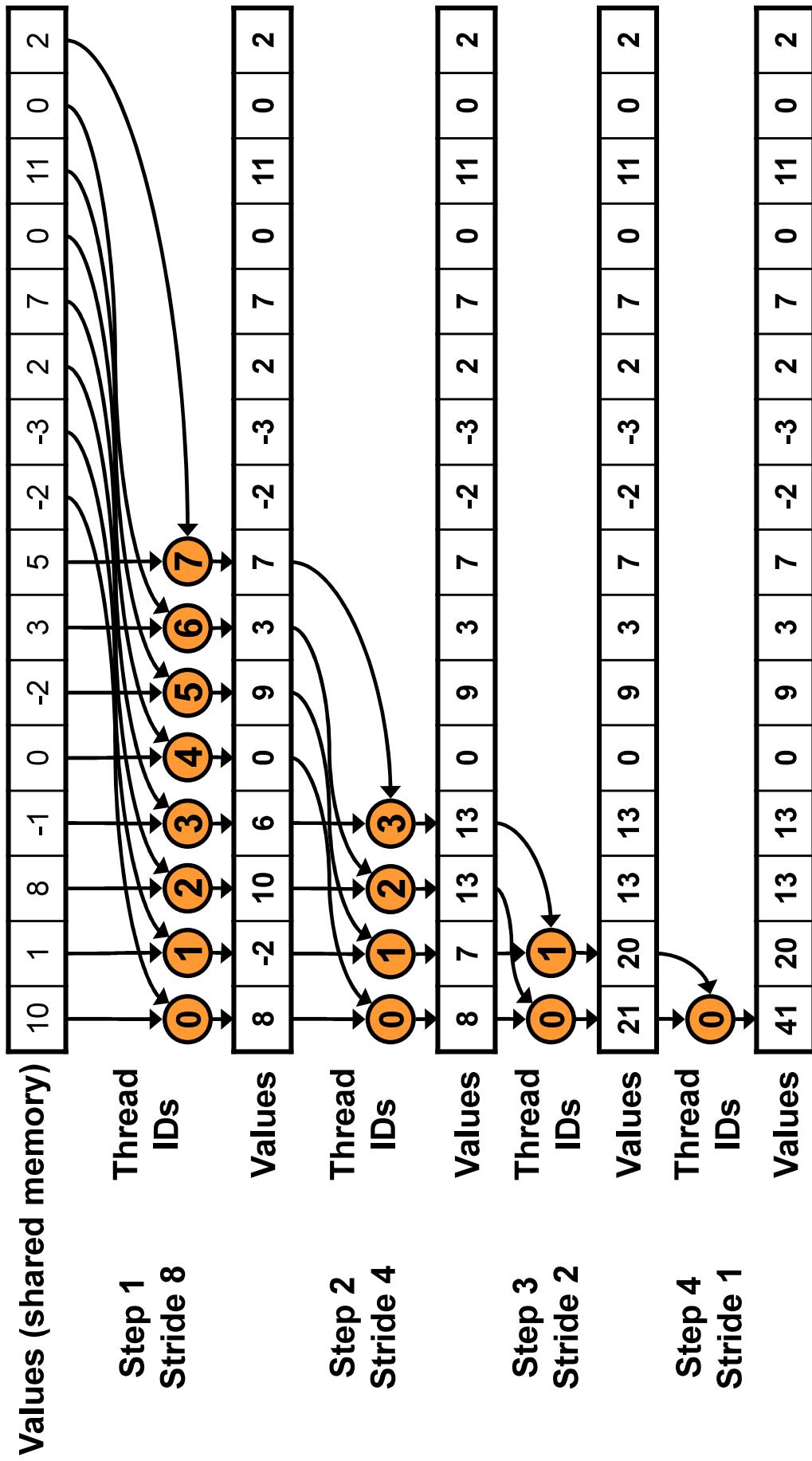
New Problem: Shared Memory Bank Conflicts

# Performance for 4M element reduction



Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>	
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>

# Parallel Reduction: Sequential Addressing



**Sequential addressing is conflict free**

## Reduction #3: Sequential Addressing

Just replace **strided indexing** in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```

With **reversed loop** and **threadID-based indexing**:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {  
    if (tid < s) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

# Performance for 4M element reduction



	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
<b>Kernel 3:</b> sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x

# Idle Threads

## Problem:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {  
    if (tid < s) {  
        sdata[tid] += sdata[tid + s];  
    }  
    __syncthreads();  
}
```

Half of the threads are idle on first loop iteration!

This is wasteful...



# Reduction #4: First Add During Load

Half the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem  
unsigned int tid = threadIdx.x;  
unsigned int i = blockDim.x*blockDim.x + threadIdx.x;  
sdata[tid] = g_idata[i];  
__syncthreads();
```

With two loads and first add of the reduction:

```
// perform first level of reduction,  
// reading from global memory, writing to shared memory  
unsigned int tid = threadIdx.x;  
unsigned int i = blockDim.x*(blockDim.x*2) + threadIdx.x;  
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];  
__syncthreads();
```

# Performance for 4M element reduction



	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>

# Instruction Bottleneck



- At 17 GB/s, we're far from bandwidth bound
  - And we know reduction has low arithmetic intensity
- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - In other words: address arithmetic and loop overhead
- Strategy: unroll loops

# Unrolling the Last Warp



- As reduction proceeds, # “active” threads decreases
  - When  $s \leq 32$ , we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when  $s \leq 32$ :
  - We don't need to syncthreads()
  - We don't need “if ( $tid < s$ )” because it doesn't save any work
- Let's unroll the last 6 iterations of the inner loop

# Reduction #5: Unroll the Last Warp

```
device void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}
```

**IMPORTANT:**  
For this to be correct,  
we must use the  
“volatile” keyword!

```
// later...
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}
if (tid < 32) warpReduce(sdata, tid);
```

**Note: This saves useless work in all warps, not just the last one!**

Without unrolling, all warps execute every iteration of the for loop and if statement

# Performance for 4M element reduction



	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>

# Complete Unrolling



- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Luckily, the block size is limited by the GPU to 512 threads
  - Also, we are sticking to power-of-2 block sizes
- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don't know at compile time?
- Templates to the rescue!
  - CUDA supports C++ template parameters on device and host functions

# Unrolling with Templates



- Specify block size as a function template parameter:

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```



## Reduction #6: Completely Unrolled

```
Template <unsigned int blockSize>
__device __ void warpReduce(volatile int* sdata, int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

```
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
    if (blockSize >= 256) {
        if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
        if (blockSize >= 128) {
            if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
        }
    }
}

if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

**Note:** all code in RED will be evaluated at compile time.

Results in a very efficient inner loop!

# Invoking Template Kernels



- Don't we still need block size at compile time?
- Nope, just a switch statement for 10 possible block sizes:

```
switch (threads)
{
    case 512:
        reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 256:
        reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 128:
        reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 64:
        reduce5< 64><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 32:
        reduce5< 32><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 16:
        reduce5< 16><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 8:
        reduce5< 8><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 4:
        reduce5< 4><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 2:
        reduce5< 2><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
    case 1:
        reduce5< 1><<< dimGrid, dimBlock, smemSize >>>(d__idata, d__odata); break;
}
```

# Performance for 4M element reduction



	Time (2 <sup>22</sup> ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>		
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>	<b>4.68x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>	<b>8.34x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>	<b>15.01x</b>
<b>Kernel 6:</b> completely unrolled	<b>0.381 ms</b>	<b>43.996 GB/s</b>	<b>1.41x</b>	<b>21.16x</b>

# What About Cost?



- **Cost of a parallel algorithm is processors time complexity**
  - Allocate threads instead of processors:  $O(N)$  threads
  - Time complexity is  $O(\log N)$ , so cost is  $O(N \log N)$  : **not cost efficient!**
- **Brent's theorem suggests  $O(N/\log N)$  threads**
  - Each thread does  $O(\log N)$  sequential work
  - Then all  $O(N/\log N)$  threads cooperate for  $O(\log N)$  steps
  - $\text{Cost} = O((N/\log N) * \log N) = O(N) \rightarrow \text{cost efficient}$
- **Sometimes called algorithm cascading**
  - Can lead to significant speedups in practice

# Algorithm Cascading



- **Combine sequential and parallel reduction**
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory
- **Brent's theorem says each thread should sum  $O(\log n)$  elements**
  - i.e. 1024 or 2048 elements per block vs. 256
- **In my experience, beneficial to push it even further**
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
    - High kernel launch overhead in last levels with few blocks
- **On G80, best perf with 64-256 blocks of 128 threads**
  - 1024-4096 elements per thread

# Reduction #7: Multiple Adds / Thread

**Replace load and add of two elements:**

```
unsigned int tid = threadIdx.x;
unsigned int i = blockDim.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

**With a while loop to add as many as necessary:**

```
unsigned int tid = threadIdx.x;
unsigned int i = blockDim.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

# Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```
unsigned int tid = threadIdx.x;
unsigned int i = blockDim.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

Note: gridSize loop stride  
to maintain coalescing!

```
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.y*gridSize + threadIdx.x;
unsigned int gridSize;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+gridSize];
    i += gridSize;
}
__syncthreads();
```

# Performance for 4M element reduction



Time ( $2^{22}$ ints)	Bandwidth	Step Speedup	Cumulative Speedup
<b>Kernel 1:</b> interleaved addressing with divergent branching	<b>8.054 ms</b>	<b>2.083 GB/s</b>	
<b>Kernel 2:</b> interleaved addressing with bank conflicts	<b>3.456 ms</b>	<b>4.854 GB/s</b>	<b>2.33x</b>
<b>Kernel 3:</b> sequential addressing	<b>1.722 ms</b>	<b>9.741 GB/s</b>	<b>2.01x</b>
<b>Kernel 4:</b> first add during global load	<b>0.965 ms</b>	<b>17.377 GB/s</b>	<b>1.78x</b>
<b>Kernel 5:</b> unroll last warp	<b>0.536 ms</b>	<b>31.289 GB/s</b>	<b>1.8x</b>
<b>Kernel 6:</b> completely unrolled	<b>0.381 ms</b>	<b>43.996 GB/s</b>	<b>1.41x</b>
<b>Kernel 7:</b> multiple elements per thread	<b>0.268 ms</b>	<b>62.671 GB/s</b>	<b>1.42x</b>
<b>Kernel 7 on 32M elements: 73 GB/s!</b>			



```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

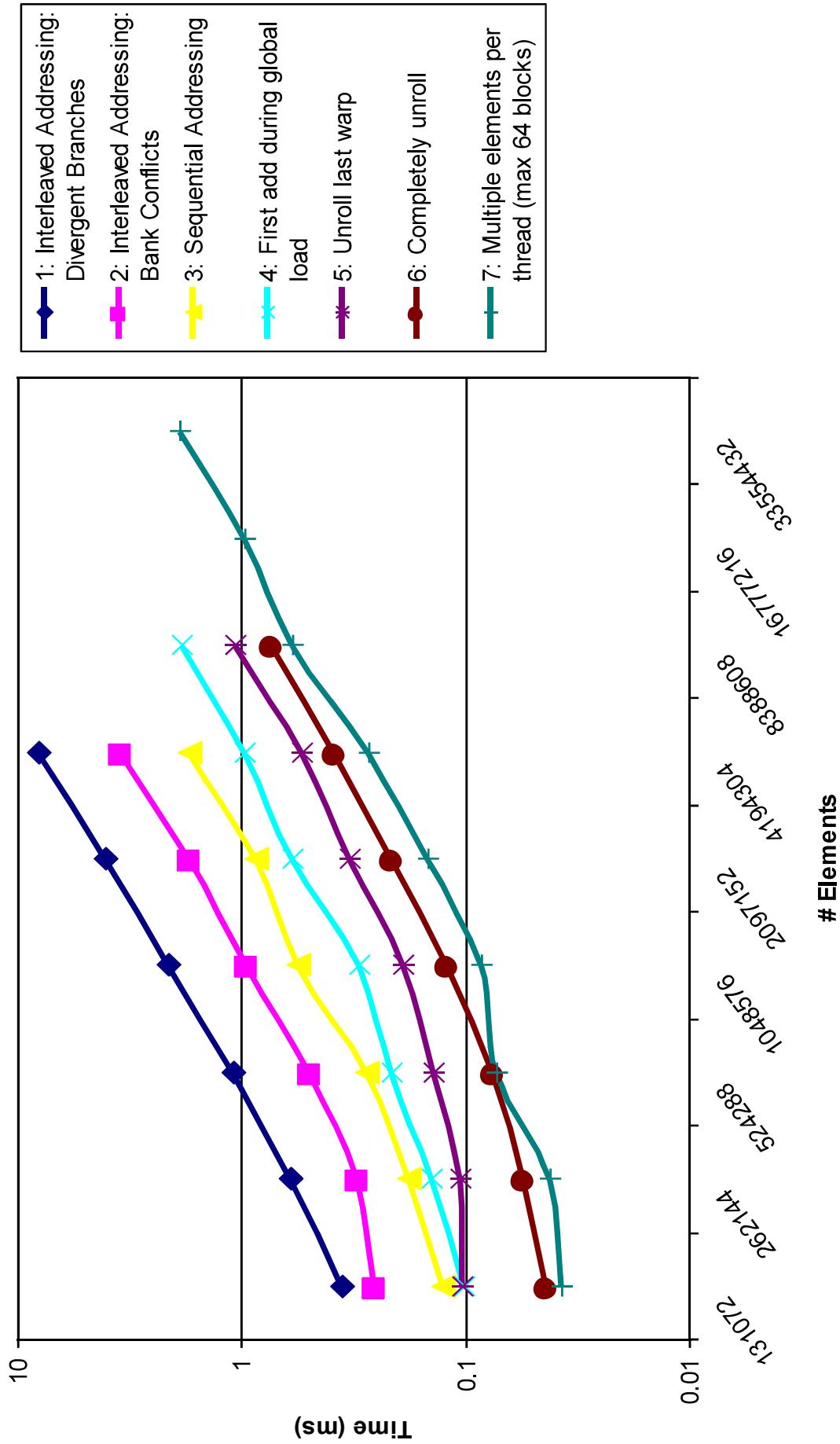
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;
}

while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
__syncthreads();

if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }

if (tid < 32) warpReduce(sdata, tid);
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Performance Comparison



# Types of optimization



- Interesting observation:
- Algorithmic optimizations
  - Changes to addressing, algorithm cascading
  - 11.84x speedup, combined!
- Code optimizations
  - Loop unrolling
  - 2.54x speedup, combined

# Conclusion



- Understand CUDA performance characteristics
    - Memory coalescing
    - Divergent branching
    - Bank conflicts
    - Latency hiding
  - Use peak performance metrics to guide optimization
  - Understand parallel algorithm complexity theory
  - Know how to identify type of bottleneck
    - e.g. memory, core computation, or instruction overhead
  - Optimize your algorithm, *then* unroll loops
  - Use template parameters to generate optimal code
- 
- Questions: [mharris@nvidia.com](mailto:mharris@nvidia.com)

# Reasoning about Performance

Sofien GANNOUNI  
Computer Science  
E-mail: [gnnosf@ksu.edu.sa](mailto:gnnosf@ksu.edu.sa) ; [gansof@yahoo.com](mailto:gansof@yahoo.com)



# Parallelism vs Performance

## ➤ Ideally, a problem:

- that takes  $T$  time on a single processor
- Can execute in  $T/P$  time on  $P$  processors

## ➤ The challenge meeting $T/P$ goal becomes more difficult as $P$ increases

## ➤ There are certain cases where the $P$ processors produce shorter execution time than $T/P$

## ➤ Parallelism and Performance are related

# Serial 3s Count Program

```
public class Count3S {  
    int array[];  
    int count;  
  
    public int count3s() {  
        count = 0;  
        for (int i = 0; i < array.length; i++) {  
            if (array[i] == 3)  
                count++;  
        }  
        return count;  
    }  
}
```

# Parallel 3s Count Program

```
public class Count3sParallel implements Runnable {  
    int array[];  
    int count, nbThread;  
    Thread t;  
    LinkedList<Integer> threadIds = new LinkedList<Integer>();  
  
    public void count3s() {  
        count = 0;  
        for (int i=0; i < nbThread; i++) {  
            t = new Thread(this);  
            threadIds.add(new Integer(i));  
            t.start();  
        }  
    }  
    public void run() {  
        int depth = (array.length / nbThread);  
        int start = threadIds.poll().intValue() * depth;  
        int end = start + depth;  
  
        for (int i = start; i < end; i++ ) {  
            if (array[i] == 3)  
                count++;  
        }  
    }  
}
```

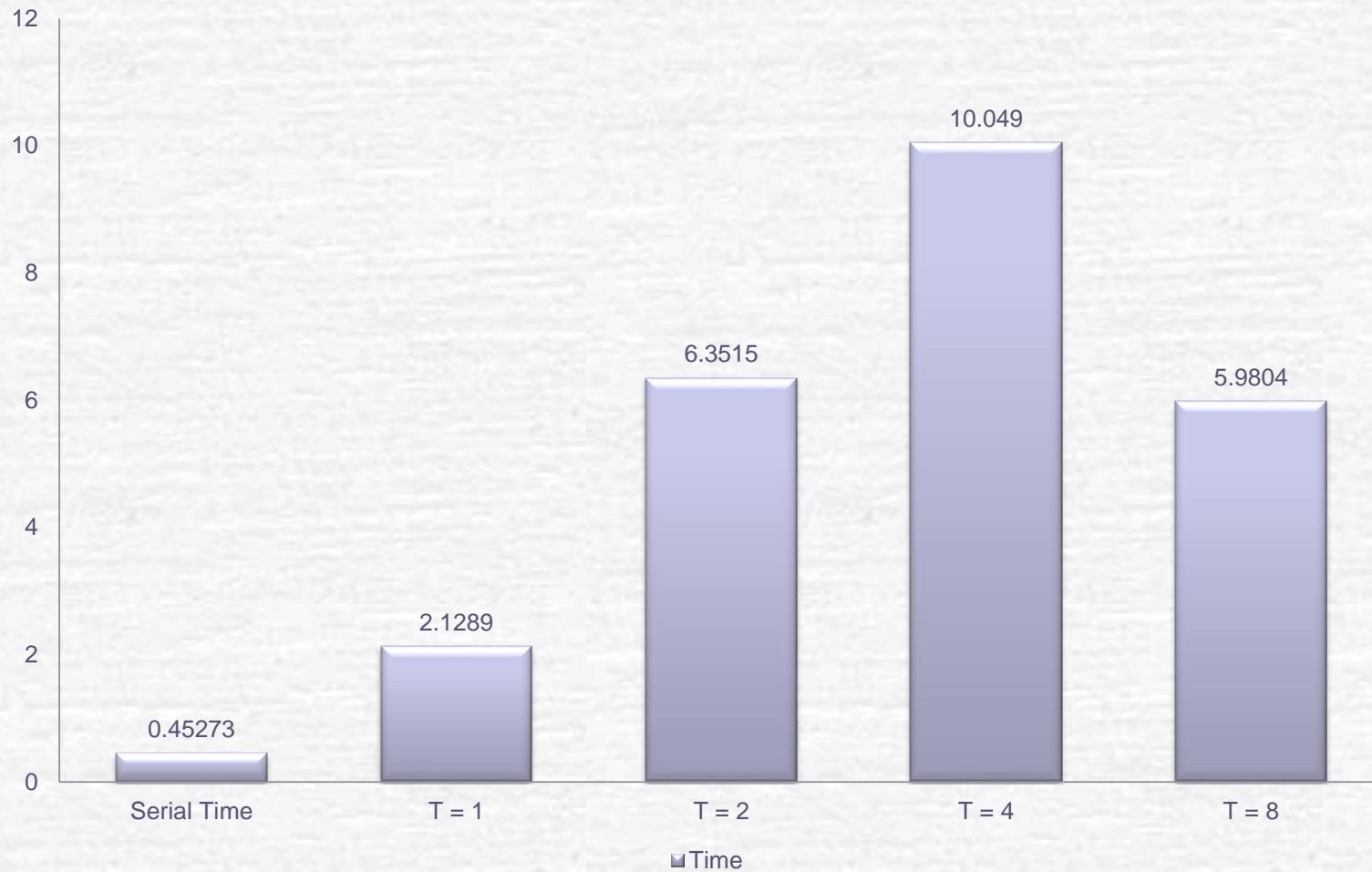
```
public class Count3SParallel2 implements Runnable {
    int array[];
    int count, nbThread;
    Thread t;
    LinkedList<Integer> threadIds = new LinkedList<Integer>();
    Mutex m = new Mutex();

    public void count3s() {
        count = 0;
        for (int i=0; i < nbThread; i++) {
            t = new Thread(this);
            threadIds.add(new Integer(i));
            t.start();
        }
    }

    public void run() {
        int depth = (array.length / nbThread);
        int start = threadIds.poll().intValue() * depth;
        int end = start + depth;

        for (int i = start; i < end; i++ ) {
            if (array[i] == 3) {
                try {
                    m.acquire();
                    count++;
                    m.release();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

# Measures of the Second Solution



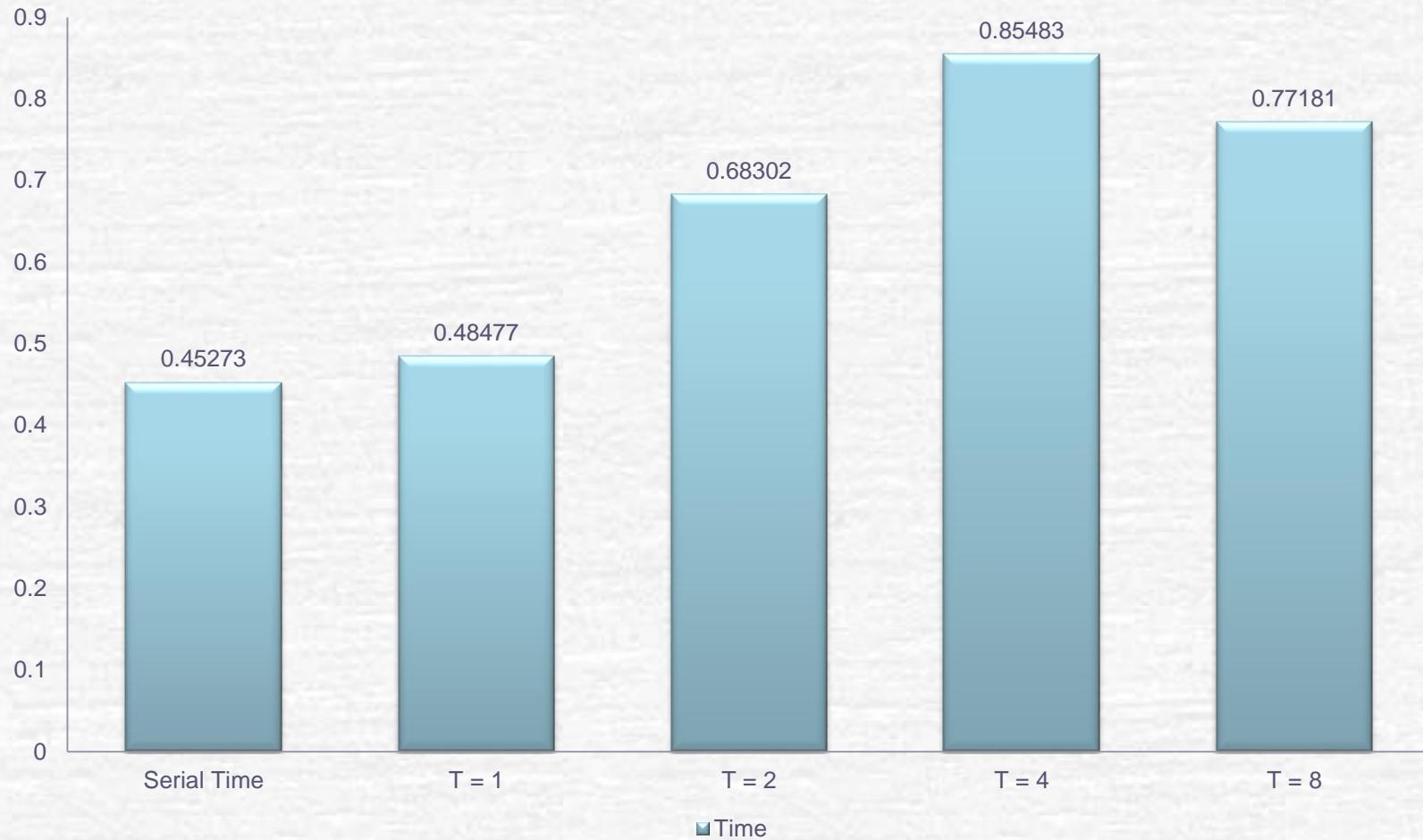
```
public class Count3sParallel3 implements Runnable {
    int array[];
    int count, nbThread;
    Thread t;
    LinkedList<Integer> threadIds = new LinkedList<Integer>();
    Mutex m = new Mutex();

    public void count3s() {
        count = 0;
        for (int i=0; i < nbThread; i++) {
            t = new Thread(this);
            threadIds.add(new Integer(i));
            t.start();
        }
    }

    public void run() {
        int depth = (array.length / nbThread);
        int start = threadIds.poll().intValue() * depth;
        int end = start + depth;
        int localCount = 0;

        for (int i = start; i < end; i++ ) {
            if (array[i] == 3)
                localCount++;
        }
        try {
            m.acquire();
            count += localCount;
            m.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

# Measures of the Third Solution



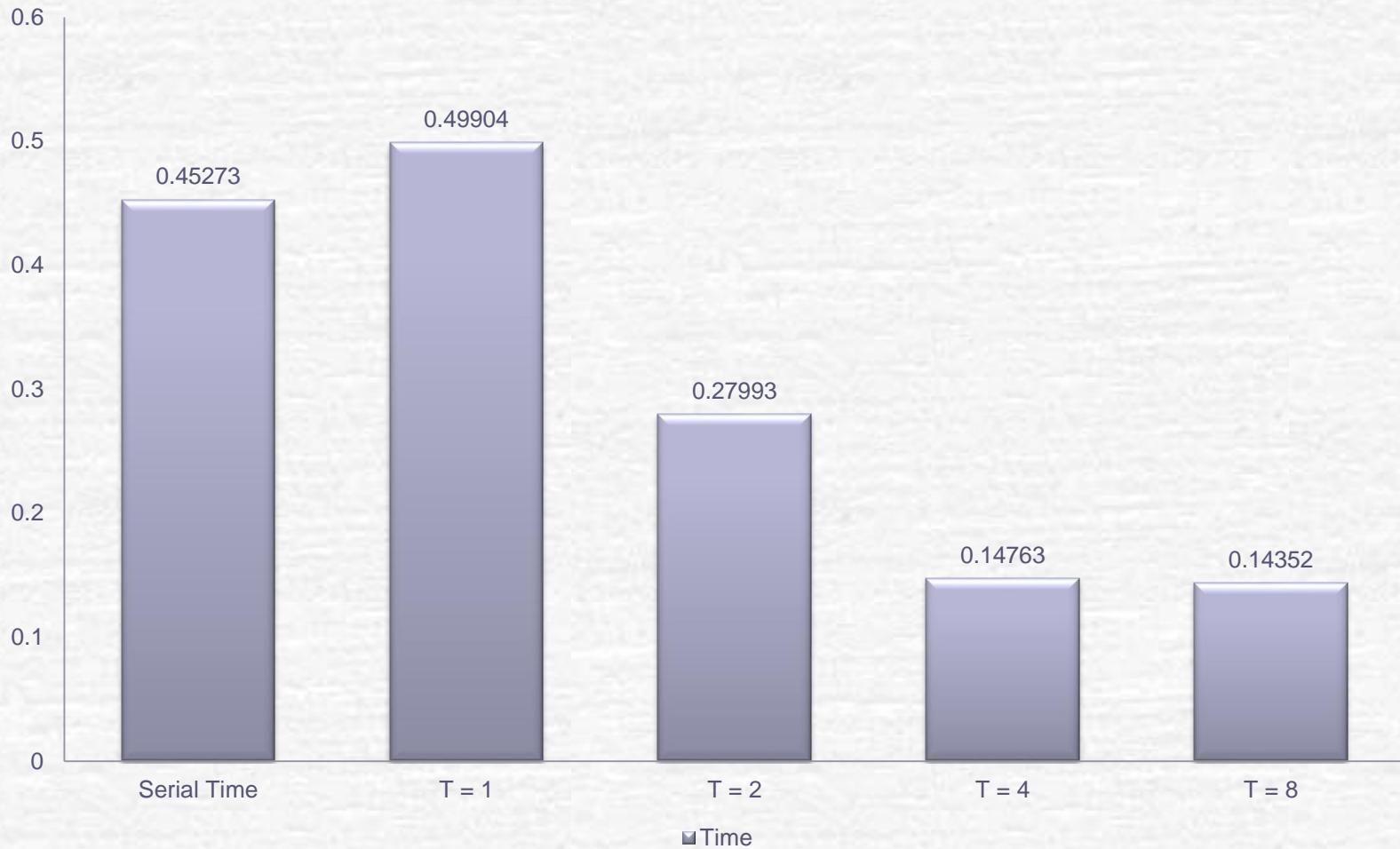
```
public class Count3sParallel4 implements Runnable {
    int array[];
    int count, nbThread;
    Thread t;
    LinkedList<Integer> threadIds = new LinkedList<Integer>();
    Mutex m = new Mutex();
    int localCounts[ ];

    public void count3s() {
        count = 0;
        for (int i=0; i < nbThread; i++) {
            t = new Thread(this);
            threadIds.add(new Integer(i));
            t.start();
        }
    }

    public void run() {
        int id = threadIds.poll().intValue();
        int depth = (array.length / nbThread);
        int start = id * depth;
        int end = start + depth;

        for (int i = start; i < end; i++ ) {
            if (array[i] == 3)
                localCounts[id]++;
        }
        try {
            m.acquire();
            count += localCounts[id];
            m.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

# Measures of the Fourth Solution



# Threads

- **A thread consists mainly of:**

- A program code
- A call stack
- And some modest amount of thread specific data

- **Threads share access to memory:**

- Communicate by reading from or writing to memory that is visible to all
- Shared-memory parallel programming

# Processes

- ➊ A process is a thread that also has its own private address space
- ➋ Processes communicate by passing messages
  - Message passing parallel programming
- ➌ Processes have much more associated state than threads
  - Larger cost to create and destroy a process
  - Process tend to be long-lived

# Latency and Throughput

- Latency refers to the amount of time it takes to complete a given unit of work
- Throughput refers to the amount of work that can be completed per unit of time
- Parallelism can be exploited to improve throughput.
- Parallelism can be exploited to hide latency

# Source of Performance Loss

Ideally,  $P$  processors could speed up a computation by a factor of  $P$ .

Reasons for not reaching this purpose

- Overhead
- Non-parallelizable computation
- Idle processors
- Contention of resource

# Overhead

- Any cost that is incurred in the parallel solution but not in the serial solution is considered **overhead**
- There is overhead in:
  - Setting up the threads and processes
  - Executing them concurrently
  - Tearing them down
- Because memory allocation and its initialization are expensive, processes incur greater setup overhead than threads

# Sources of Overhead

## Communication

## Synchronization

## Computation

- Parallel computations always perform extra computations that are not needed in the sequential solution

## Memory

# Non-Parallelizable Code

## Amdahl's Law

- If  $1/S$  of a computation is inherently sequential
- The maximum performance improvement is a factor of  $S$
- $T_p = (1/S * T_s) + ( (1 - 1/S) * T_S / P )$

# Idle Time

- Ideally, all processors are working all the time
- A process or thread might not be able to proceed due to a lack of work:
  - Waiting for some external event such as arrival of data
- Idle time is consequence of synchronization and communication

# **Contention**

- ➊ **Contention is the degradation of system performance caused by competition**
- ➋ **Contention can lead to slow down performance**
- ➌ **Lock contention can reduce performance**

# Parallel Structure

There are various factors that can degrade parallel performance

- Overhead
- Non-parallelizable computation
- Idle processors
- Contention of resource

There are three related concepts that can help avoid such problems

- Dependences
- Granularity
- Locality

# Dependences

- A dependence is an ordering relationship between two computation
- Different forms of dependences
  - A process waiting a message to arrive from another process
  - Read and Write operations: Threaded computation correspond to memory loads and stores
- Dependences provide a way to reason about potential sources of performance loss

# Data Dependence

➤ A Data dependence is an ordering on a pair of memory operations that must be preserved to maintain correctness

## ➤ Kinds of data dependences

- Flow dependence: read after write
- Anti dependence: write after read
- Output dependence: write after write
- Input dependence: read after read

# Data Dependences

## Flow dependence

- is called true dependence
- Represent the fundamental ordering of memory operations

## Anti and output dependences

- are called false dependences
- Arise from the reuse of memory



## Consider the following code fragment:

- 1: sum = a + 1;
- 2: first\_term = sum \* scale1;
- 3: sum = b +1;
- 4: second\_term = sum \* scale2;

### Flow dependence: lines 1&2 and 3&4

### Anti dependence: line 2&3

- Prevents from executing the two lines concurrently
- Solution rename the variables: firstSum , secondSum

# Dependences Limit Parallelism

- sum = 0;
- for (int i=0 ; i < n ; i++)
  - Sum += x[ i ];
- (x[0] + x[1]) +...+ (x[i] + x[i+1])

# Granularity

## Granularity

- is a key concept for managing the constraints imposed by dependences
  - Coarse and fine grain
  - Large and small grain

## Granularity is determined by the frequency of interactions among threads or processes

- Frequency is measured in number of instructions between interactions

# Coarse and Fine Grain

## Coarse grain

- refers to threads or processes that only infrequently depend on data or events in other threads or processes

## Fine grain

- Refers to threads or processes that interact frequently

## Each interaction introduces communication and synchronization overheads

# Locality

- Locality is closely related to granularity
- Computations can exhibit :
  - Temporal locality
    - Memory references that are clustered in time
  - Spatial locality
    - Memory references that are clustered by address
- Non-local references imply some form of communication

# Performance Trade-Offs

## Communication vs. computation

- Overlapping communication and computation
- Redundant computation

## Memory vs. Parallelism

- Privatization
- Padding

## Overhead vs. parallelism

- Parallelize overhead
- Load balance vs. overhead
- Granularity Trade-offs

# Measuring Performance

- Execution time
- Speedup
- Superlinear Speedup
- Efficiency

## QUESTION 1

Let's consider an array of integers of size 8. We would like to sort the elements of the array in an ascending way using the Bitonic sort algorithm. We focus on step No 2. Give the following information related to every stage *i* of step 2:

1. Give the IDs of threads involved in stage *i*
2. Infer the condition that is satisfied by the involved thread in stage *i*.
3. Give for every thread, involved in stage *i*, the bitonic sequence (just give an interval  $[b, e]$  where **b** is the index of the first cell of the sequence and the **e** is the index of the last cell of the sequence) that the thread is processing
4. Specify for every thread whether the corresponding sequence is sorted in an ascending (+BM) or a descending (-BM) way.

## QUESTION 2

Let's consider an array of integers of size 8. We would like to sort the elements of the array in an ascending way using the Bitonic sort algorithm. We focus on step No 1. Give the following information related to step 1.

1. Give the IDs of threads involved in step 1.
2. Give for every thread the bitonic sequence (just give an interval  $[b, e]$  where **b** is the index of the first cell of the sequence and the **e** is the index of the last cell of the sequence) that the thread is processing
3. Specify for every thread whether the corresponding sequence is sorted in an ascending (+BM) or a descending (-BM) way.

## QUESTION 3

Let's consider an array of integers of size 32. We would like to sort the elements of the array in an ascending way using the Bitonic sort algorithm:

1. Give the number of steps that are required to sort the elements of the array.
2. Give the size of bitonic sequences in every step.
3. Give the bitonic sequences (just give an interval  $[b, e]$  where **b** is the index of the first cell of the sequence and the **e** is the index of the last cell of the sequence) that are processed in every step.
4. Specify, in every step, for every bitonic sequence whether the sequence is sorted in an ascending (+BM) or a descending (-BM) way.

## QUESTION 4

Let's consider 2 arrays of integers called A and B respectively. Let's consider that we would like to write a C program that runs in parallel and that computes the sum of the 2 arrays:

$$C[i] = A[i] + B[i]$$

1. Write the kernel (called **add\_kernel**) that will run on N Blocks of M threads where every thread processes W elements.
2. Write the main that will call the kernel **add\_kernel** to compute the sum of two arrays in parallel and displays the result.

Question 5: Find the cell\_index

Block (0, 0)				
Cell 0	Cell 3	Cell 6	Cell 9	Cell 12
Cell 1	Cell 4	Cell 7	Cell 10	Cell 13
Cell 2	Cell 5	Cell 8	Cell 11	Cell 14

Block (1, 0)				
Cell 15	Cell 18			Cell 27
Cell 16				Cell 28
Cell 17				Cell 29

Block (0, 1)				
Cell 30	Cell 33			Cell 42
Cell 31				Cell 43
Cell 32				Cell 44

Block (1, 1)				
Cell 45	Cell 48	Cell 51	Cell 54	Cell 57
Cell 46	Cell 49	Cell 52	Cell 55	Cell 58
Cell 47	Cell 50	Cell 53	Cell 56	Cell 59

## Question 6

```
__global__ void Kernel_A(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        Kernel_C<<< 1, 256 >>>(data);
        Kernel_D<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}

__global__ void Kernel_C(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        Kernel_E<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}

void host_launch(int *data) {
    kernel_A<<< 1, 256 >>>(data);
    kernel_B<<< 1, 256 >>>(data);
    cudaDeviceSynchronize();
}
```

1. Give and explain the order of execution of the given parallel nested kernels.
2. Explain the role of the `__syncthreads()` statements.
3. Explain the role of the `cudaDeviceSynchronize()` statements

## QUESTION 7

Let's consider that a kernel A launches a Kernel B. Explain the memory synchronization (what the child kernel can see and when the parent kernel can read the child writes) between the parent and the child kernels.

*Q. 5*  
Write a C program that displays the following:

- The number of CUDA devices available on your computer.
- The number and names of CUDA devices having more than 256 multiprocessors.

a) int count;

```
cudaGetDeviceCount(&count);  
printf("%d", count);
```

b) int i; Cuda Device properties prop;

```
CudaGetDeviceCount(&count);
```

```
int i; int num = 0;
```

```
for (i = 0; i < count; i++) {
```

```
    CudaGetDeviceProp(&prop, i);
```

```
    if (prop.multiProcessorCount > 256) {
```

```
        num += 1;
```

```
        printf("%s", prop.name);
```

```
printf("%d", num);
```

King Saud University  
College of Computer and Information Sciences  
Department of Computer Science  
**CSC453 – Parallel Processing – Midterm 2 Exam – Fall 2021**

## Question 2

Let's consider 2  $N \times N$  square matrices of integers A and B. Let's consider that we would like to write a C program that runs in parallel and that computes and returns the sum of the 2 matrices:

$$C[i][j] = A[i][j] + B[i][j]$$

1. We would like to run this kernel within a grid composed of a single 2-D thread-block where every thread processes a single cell of the matrix.

- Give the code of the following kernel.

```
_global_ void add(int *a, int *b, int *c, int N) {
```

```
int rowIndex = ThreadIdx.y ;  
int colIndex = ThreadIdx.x ;  
C[rowIndex][colIndex] = A[rowIndex][colIndex] +  
B[rowIndex][colIndex];
```

3

King Saud University  
 College of Computer and Information Sciences  
 Department of Computer Science  
 CSC453 – Parallel Processing – Midterm 2 Exam – Fall 2021

2. We would like to run this kernel within a grid composed of many 2-D thread-blocks where every thread processes a single cell of the matrix.

- Give the code of the kernel.

```
global_ void add(int *a, int *b, int *c, int N) {
```

```
int rowIndex = ThreadIdx.y + blockIdx.x * blockDim.y;
```

```
int colIndex = ThreadIdx.x + blockIdx.y * blockDim.x;
```

```
C[rowIndex][colIndex] = A[rowIndex][colIndex] + B[rowIndex][colIndex];
```

3

3. We would like to run this kernel within a grid composed of many 2-D thread-blocks where every thread processes a sub-square matrix of size  $W$  by  $W$ .

- Give the code of the kernel.

```
global_ void add(int *a, int *b, int *c, int W, int N) {
```

```
int rowIdx = (ThreadIdx.y + blockIdx.y * blockDim.y) * W;
```

```
int colIdx = (ThreadIdx.x + blockIdx.x * blockDim.x) * W;
```

```
int i; int j;
```

```
for (i = 0; i < W; i++) {
```

```
    for (j = 0; j < W; j++) {
```

```
* C[rowIdx+i][colIdx+j] = A[rowIdx+i][colIdx+j]
+ B[rowIdx+i][colIdx+j]; } }
```

5

3

### Question 3

Let's consider that we would like to sort ascendingly an array of integers of size  $N$  using the bitonic merge-sort algorithm.

- Give the number of steps that are required to sort the elements of the array.

1  $\text{Steps} = \log_2(N)$

- Give the number of stages that are required in every step  $i$ .

~~Number of stages in every step  $i$~~  Stages =  $i$

1 Same number as the step number

- Give the size of bitonic sequences in every step.

1  $\xrightarrow{\text{Step}} 2$

- Give the size of bitonic sequences in every stage of a step  $i$ .

1  $\xrightarrow{\text{stage } i} 2^{i-1}$

- Give the condition that should satisfy a thread to participate in the processing of bitonic sequences of a stage  $j$  of a step  $i$ .

ThreadIndex  $\% \frac{2^j}{2^{j-1}} < \frac{2^j}{2}$

- Give the condition that should satisfy a thread that participates in the processing of sequences of a stage  $j$  of a step  $i$  to sort its corresponding bitonic-sequence ascendingly.

ThreadIndex  $\% 2 = = 0$

```
cudaProp:  
int n;  
cudaGetDeviceCount(&n);  
printf("%d", n);  
int x=0;  
for (int i=0; i<n; i++) {  
    cudaGetDeviceProperties(&prop, i);  
    if (prop.multiProcessorCount >= 256) {  
        x++;  
        printf("%d", prop.name);  
    }  
}  
printf("%d", x);
```

1:10  
1:05

$$1) \begin{aligned} - r\bar{\Sigma} &= \text{thread}[\bar{dx}, \bar{y}] \\ - c\bar{\Sigma} &= \text{thread}[\bar{dx}, \bar{x}] \\ - C[\bar{r}\bar{\Sigma}][\bar{c}\bar{\Sigma}] &= A[\bar{r}\bar{\Sigma}][\bar{c}\bar{\Sigma}] + B[\bar{r}\bar{\Sigma}][\bar{c}\bar{\Sigma}] \end{aligned}$$

$$2) \begin{aligned} r\bar{\Sigma} &= b\bar{\Sigma}[\bar{dx}, \bar{y}] * b\bar{\Sigma}[\bar{dim}, \bar{y}] + T\bar{\Sigma}[\bar{dx}, \bar{y}] \\ C\bar{\Sigma} &= b\bar{\Sigma}[\bar{dx}, \bar{n}] * b\bar{\Sigma}[\bar{dim}, \bar{n}] + T\bar{\Sigma}[\bar{dx}, \bar{n}] \end{aligned}$$

$$C[\bar{r}\bar{\Sigma}][\bar{c}\bar{\Sigma}] = A[\bar{r}\bar{\Sigma}][\bar{c}\bar{\Sigma}] + B[\bar{r}\bar{\Sigma}][\bar{c}\bar{\Sigma}]$$

$$3) \begin{aligned} r\bar{\Sigma} &= (b\bar{\Sigma}[\bar{dx}, \bar{y}] * b\bar{\Sigma}[\bar{dim}, \bar{y}] + T\bar{\Sigma}[\bar{dx}, \bar{y}]) * \omega \\ C\bar{\Sigma} &= (b\bar{\Sigma}[\bar{dx}, \bar{n}] * b\bar{\Sigma}[\bar{dim}, \bar{n}] + T\bar{\Sigma}[\bar{dx}, \bar{n}]) * \omega \\ &\quad \text{for (int } i=r\bar{\Sigma}; i < r\bar{\Sigma}+\omega; i++) \\ &\quad \text{for (int } j=c\bar{\Sigma}; j < c\bar{\Sigma}+\omega; j++) \\ - C[\bar{i}][\bar{j}] &= A[\bar{i}][\bar{j}] + B[\bar{i}][\bar{j}] \end{aligned}$$

3. Enumerate and give a brief description of the main opportunities of parallelism.

1 - instruction level parallelism: parallelism which are hidden in computer programs

2 - single computer parallelism: parallelism using multi-core or multi-processor

3 - multi-computer parallelism: parallelism using multiple computers such as grids, servers, and clusters 3

2. Give the main differences between parallel processing and Distributed computing.

1 - in distributed computing processors are distant geographically but in parallel processing processors are in the same machine

2 - in distributed computing the goal is to provide availability and reliability but in parallel processing the goal is to increase performance

3 - IN The back of the page →

3. Enumerate and give a brief description of the main aspects of parallel computing.

1 - parallel computing architecture: PRAM, CTA

2 - parallel algorithms and application: Reasoning and design

3 - parallel programming:

paradigms

models

parallel programming language

frameworks

environments

3

1

Q1:

②

3-in distributed computing the interaction are infrequent and coarse grained but in parallel processing the interaction are frequent and fine grained

## Question 2

1. Explain the main differences between the Blocking non-buffered and the Non-Blocking non-buffered send/receive operations of the message passing paradigm.

\* in blocking non-buffered when the sender issues a send operation he has to wait until the receiver match a receive operation, but in non-blocking non-buffered when the sender issues a send operation he can continue processing until the receiver send a interrupt

2

2. Describe the **Task Farming** and the **Divide-and-Conquer** programming models and explain the main differences between them.

- Task farming has two entities the master and the workers, the master split the problem into tasks and send it to the workers and gather the result from them, the worker take a task and process it then send it to the master

- divide-and-conquer decompose the problem into sub-tasks ~~non~~ recursively until it reach base case which can be solved directly

2

the main differences:

1- in divide-and-conquer we decompose the task recursively but in task farming we split the tasks

2- in divide-and-conquer ~~non~~ sub-tasks have the same nature as the main problem but in task farming tasks may have different nature

2

## Question 3

1. Enumerate and explain the different types of memory adopted by CUDA.

- 1 - register: per thread, not cached, on chip, fast, read \ write
- 2 - Local-memory: per thread, not cached, on DRAM, read \ write
- 3 - Shared-memory: per block, not cached, on chip, synchronize block threads, read \ write
- 4 - Global-memory: per Grid, not cached, on DRAM, synchronize between grid, read \ write
- 5 - constant-memory: per Grid, cached, on DRAM, read only
- 6 - texture-memory: per Grid, cached, on DRAM, read only

2. Explain why the Global memory is not cached in CUDA, while the Constant memory is cached.

Global memory is not cached because it is read and written so the values may change and it will be different if we cache it, but in constant memory it is only read so we can cache the value and it will not change.

1  
2

10 /

**King Saud University**  
**College of Computer and Information Sciences**  
**Department of Computer Science**  
**CSC453 – Parallel Processing – Midterm Exam – Fall 2021**

## Question 4

Let's consider 2 arrays of integers A and B of size  $S$ . Let's consider that we would like to write a C program that runs in parallel and that computes the sum of 2 arrays as following:

$$C[i] = A[i] + B[i]$$

Let's consider the following kernel:

```
_global_ void add(int *a, int *b, int *c, int size) {
    int cell_id = .....;
    if (cell_id < size)
        c[cell_id] = a[cell_id] + b[cell_id];
}
```

- We would like to run this kernel on grid composed of **1 block** where every thread evaluates a single cell as shown in the following figure:

Block (0, 0)				
Cell 0	Cell 3	Cell 6	Cell 9	Cell 12
Cell 1	Cell 4	Cell 7	Cell 10	Cell 13
Cell 2	Cell 5	Cell 8	Cell 11	Cell 14

thread\_idx.y +  
 $(\text{thread\_id}_x \cdot x * \text{blockdim}_y)$

$$2(2 \times 3) = 6$$

- Give the formula that allows every thread to compute the cell\_id of the cell he is going to process.

$$\text{int cell\_id} = \text{ThreadIdx}_x \cdot y + (\text{ThreadIdx}_x \cdot x * \underline{\text{blockDim}}_y)$$

2  
—

thread evaluates a single cell as shown in the following figure:

Block (0, 0)					Block (1, 0)				
Cell 0	Cell 6			Cell 24	Cell 30	Cell 36			Cell 54
Cell 1	Cell 7			Cell 25	Cell 31	Cell 37			Cell 55
Cell 2	Cell 8			Cell 26	Cell 32	Cell 38			Cell 56
Block (0, 1)					Block (1, 1)				
Cell 3	Cell 9			Cell 27	Cell 33	Cell 39			Cell 57
Cell 4	Cell 10			Cell 28	Cell 34	Cell 40			Cell 58
Cell 5	Cell 11			Cell 29	Cell 35	Cell 41			Cell 59

- Give the formula that allows every thread to compute the cell\_id of the cell he is going to process.

int cell\_id = Thread.y + (blockIdx.y \* blockDim.y) +  
~~(ThreadIdx.x \* blockDim.y \* GridDim.y) +~~  
~~(blockIdx.x \* blockDim.x \* blockDim.y \* GridDim.y);~~

30  
~~20~~  
~~50~~  
~~11 3~~  
~~3 2~~  
~~thredIdx.y \* (blockIdx.y \* blockDim.y) + (thredIdx.x \* blockDim.y \* GridDim.y) +~~  
~~blockIdx.x \* blockDim.x \* blockDim.y \* GridDim.y;~~  
~~5 3 9 2~~

## Midterm Exam – Fall 2021

Let's consider 2 arrays of integers A and B of size  $S$ . Let's consider that we would like to write a kernel in C that computes the sum of 2 arrays:  
 $C[i] = A[i] + B[i]$  for  $i:0$  to  $S-1$ .

We would like to run this kernel on a grid composed of 1 block where every thread evaluates  $N$  cells as shown in the following figure (where  $N=4$  as a sample):

Block (0, 0)				
Cells 0-3	Cells 4-7	Cells 8-11	Cells 12-15	Cells 16-19
Cells 20-23	Cells 24-27	Cells 28-31	Cells 32-35	Cells 36-39
Cells 40-43	Cells 43-47	Cells 48-51	Cells 52-55	Cells 56-60

- Write the kernel

```
_global_ void add(int *a, int *b, int *c, int size, int N) {
```

~~int index = (ThreadIdx.x \* N + ThreadIdx.y \* blockDim.x) \* N~~

4  
  ↓

```

int index = (ThreadIdx.x * N + ThreadIdx.y * blockDim.x) * N
for(int i=0; i< N; i++) {
    c[index+i] = a[index+i] + b[index+i]
}
}
```

We would like to run a kernel on grid configured as M \* N matrix of thread blocks. Every thread handles only one cell. Give the statement that calculates the `cell_id` for each thread as shown in the following figure:

Block (0, 0)					Block (1, 0)				
Cell 0	Cell 3	Cell 6	Cell 9	Cell 12	Cell 15	Cell 18			Cell 27
Cell 1	Cell 4	Cell 7	Cell 10	Cell 13	Cell 16				Cell 28
Cell 2	Cell 5	Cell 8	Cell 11	Cell 14	Cell 17				Cell 29

Block (0, 1)					Block (1, 1)				
Cell 30	Cell 33			Cell 42	Cell 45	Cell 48	Cell 51	Cell 54	Cell 57
Cell 31				Cell 43	Cell 46	Cell 49	Cell 52	Cell 55	Cell 58
Cell 32				Cell 44	Cell 47	Cell 50	Cell 53	Cell 56	Cell 59

`Cell_id=(threadIdx.y+threadIdx.x*blockDim.y)+((blockIdx+blockIdx.x*gridDim.y)*(blockDim.x*blockDim.y));`

**Give the CUDA statements that allow to get the number of available GPU devises.**

```
int count;
cudaGetDeviceCount(&count);
printf(" the number of available GPU devises is %d ",count)
```

**Give the statements that displays the name and the number of multi-processors of the current GPU device .**

```
Int dev;  
  
cudaDeviceProp prop;  
  
cudaGetDevice(&dev);  
  
cudaGetDeviceProperties(%prop,dev);  
  
printf("name of the current GPU device is:%s /n",prop.name);  
  
printf("number of multi-processors is: %d/n  
",prop.MultiProcessorCount);
```

**Let's consider a (N by N) Matrix of integers . We assume that the space memory on the GPU device for the Matrix is already allocated.**

**Complete the following program to upload the elements of the Matrix from the host to the GPU device.**

```
#define N 16  
int main(void) {  
    int *a; // The host copie of the Matrix a  
    int *d_a; // The device copie of the Matrix a  
    int size = N * N * sizeof(int);  
  
    // Allocate space for the host copie of a and setup  
    // input values  
    a = (int *)malloc(size); random_ints(a, N * N);  
  
    cudaMemset(d_a, 0, size);  
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);  
}
```

We would like to run a kernel on grid configured as M \* N matrix of thread blocks. Every thread handles only one cell. Give the statement that calculates the *cell\_id* for each thread as shown in the following figure:

Block (0, 0)					Block (1, 0)				
Cell 0	Cell 6	Cell 12	Cell 18	Cell 24	Cell 30	Cell 36			Cell 54
Cell 1	Cell 7	Cell 13	Cell 19	Cell 25	Cell 31				Cell 55
Cell 2	Cell 8	Cell 14	Cell 20	Cell 26	Cell 32				Cell 56
Block (0, 1)					Block (1, 1)				
Cell 3	Cell 9			Cell 27	Cell 33	Cell 39	Cell 45	Cell 51	Cell 57
Cell 4				Cell 28	Cell 34	Cell 40	Cell 46	Cell 52	Cell 58
Cell 5				Cell 29	Cell 35	Cell 41	Cell 47	Cell 53	Cell 59

`Cell_id=((threadIdx.x*gridDim.y*blockDim.y)+(threadIdx.y+blockIdx.y*blockDim.y)+(blockIdx.x*gridDim.y)*(blockDim.x*blockDim.y));`

Give the statements that allow to allocate a space memory in the GPU device for a N by N Matrix of integers.

```
#define N (2048*2048)

Int size = N* sizeof(int);

Int *d_a

cudaMalloc((void**) &d_a, size);
```



**Block (0, 0)**

Cell 0	Cell 6	Cell 12	Cell 18	Cell 24
Cell 1	Cell 7	Cell 13	Cell 19	Cell 25
Cell 2	Cell 8	Cell 14	Cell 20	Cell 26

**Block (1, 0)**

Cell 30	Cell 36			Cell 54
Cell 31				Cell 55
Cell 32				Cell 56

**Block (0, 1)**

Cell 3	Cell 9			Cell 27
Cell 4				Cell 28
Cell 5				Cell 29

**Block (1, 1)**

Cell 33	Cell 39	Cell 45	Cell 51	Cell 57
Cell 34	Cell 40	Cell 46	Cell 52	Cell 58
Cell 35	Cell 41	Cell 47	Cell 53	Cell 59

```
cell_ID = threadIdx.y + blockIdx.y * blockDim.y  
+ (threadIdx.x * (gridDim.y * blockDim.y)  
+ (blockIdx.x * gridDim.y) * (blockDim.x * blockDim.y);
```

We would like to run a kernel on grid configured as M \* N matrix of thread blocks. Every thread handles only one cell. Give the statement that calculates the **cell\_id** for each thread as shown in the following figure:

**Block (0, 0)**

Cell 0	Cell 3	Cell 6	Cell 9	Cell 12
Cell 1	Cell 4	Cell 7	Cell 10	Cell 13
Cell 2	Cell 5	Cell 8	Cell 11	Cell 14

**Block (1, 0)**

Cell 15	Cell 18			Cell 27
Cell 16				Cell 28
Cell 17				Cell 29

**Block (0, 1)**

Cell 30	Cell 33			Cell 42
Cell 31				Cell 43
Cell 32				Cell 44

**Block (1, 1)**

Cell 45	Cell 48	Cell 51	Cell 54	Cell 57
Cell 46	Cell 49	Cell 52	Cell 55	Cell 58
Cell 47	Cell 50	Cell 53	Cell 56	Cell 59

```
cell_ID = threadIdx.y + threadIdx.x * blockDim.y +  
(blockIdx.y * gridDim.x + blockIdx.x) * (blockDim.x * blockDim.y)
```

Let's consider a (N by N) Matrix of integers . We assume that the space memory on the GPU device for the Matrix is already allocated.

Complete the following program to upload the elements of the Matrix from the host to the GPU device.

```
#define N 16
int main(void) {
    int *a;          // The host copie of the Matrix a
    int *d_a;        // The device copie of the Matrix a
    int size = N * N * sizeof(int);

    // Allocate space for the host copie of a and setup input values
    a = (int *)malloc(size); random_ints(a, N * N);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

Give the statements that allow to allocate a space memory in the GPU device for a N by N Matrix of integers.

```
int size = N * N * sizeof(int);  
int *d_a;  
cudaMalloc( (void **) &d_a, size);
```

Give the statements that displays the name et the number of multi-processors of the current GPU device .

```
cudaDeviceProp prop;  
int dev ;  
cudaGetDevice( &dev );  
cudaGetDeviceProperties( &prop, dev );  
printf( "Name: %s\n", prop.name );  
printf( "Multiproc: %d\n", prop.multiProcessorCount );
```

Give the CUDA statements that allow to get the number of available GPU devices.

```
int count;  
cudaGetDeviceCount( &count);
```