LAU
School of Arts and Sciences
Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Programming with Shared Memory

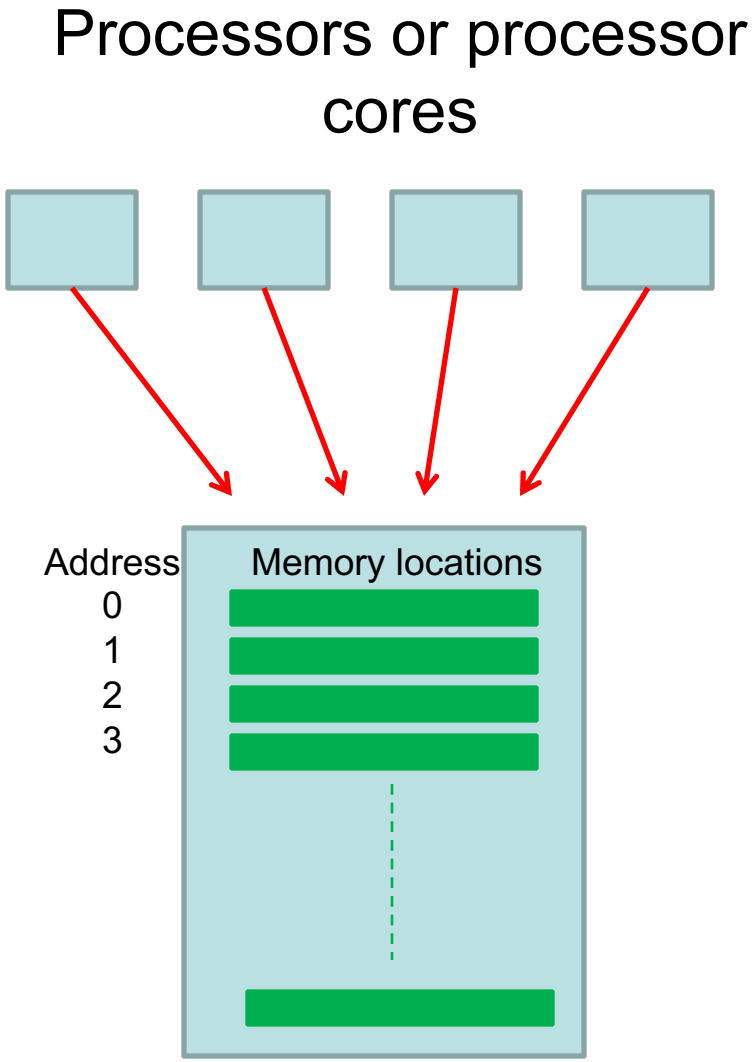
Multiprocessors and Multicore Processors

Shared memory multiprocessor/multicore processor system

Single address space exists – each memory location given unique address within single range of addresses.

Any memory location can be accessible by any of the processors.

Programming can take advantage of shared memory for holding data. However access to shared data by different processors needs to be carefully controlled, usually explicitly by programmer.



Alternatives for Programming Shared Memory Multiprocessors:

- Using heavy weight processes.
- Using threads. Example Pthreads
- Using a completely new programming language for parallel programming - not popular. Example Ada.
- Using library routines with an existing sequential programming language.
- Modifying the syntax of an existing sequential programming language to create a parallel programing language. Example UPC
- Using an existing sequential programming language supplemented with compiler directives for specifying parallelism. Example OpenMP

Concept of a process

Basically a self-contained program having its own allocation of memory, stack, registers, instruction pointer, and other resources.

Operating systems often based upon notion of a process.

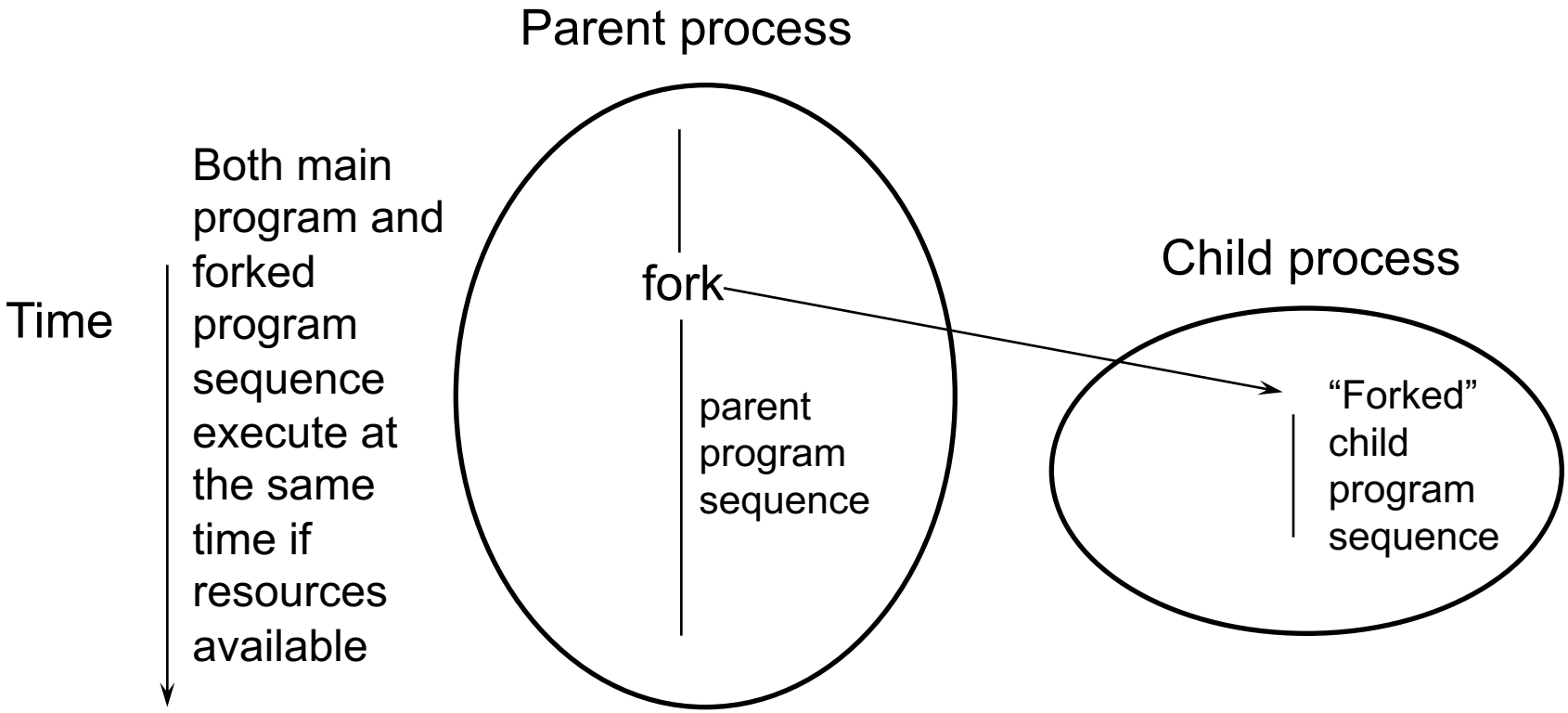
Processor time shared between processes, switching from one process to another. Might occur at regular intervals or when an active process becomes delayed.

Offers opportunity to de-schedule processes blocked from proceeding for some reason, e.g. waiting for an I/O operation to complete.

Process is the basic execution unit in message-passing MPI.

Fork pattern

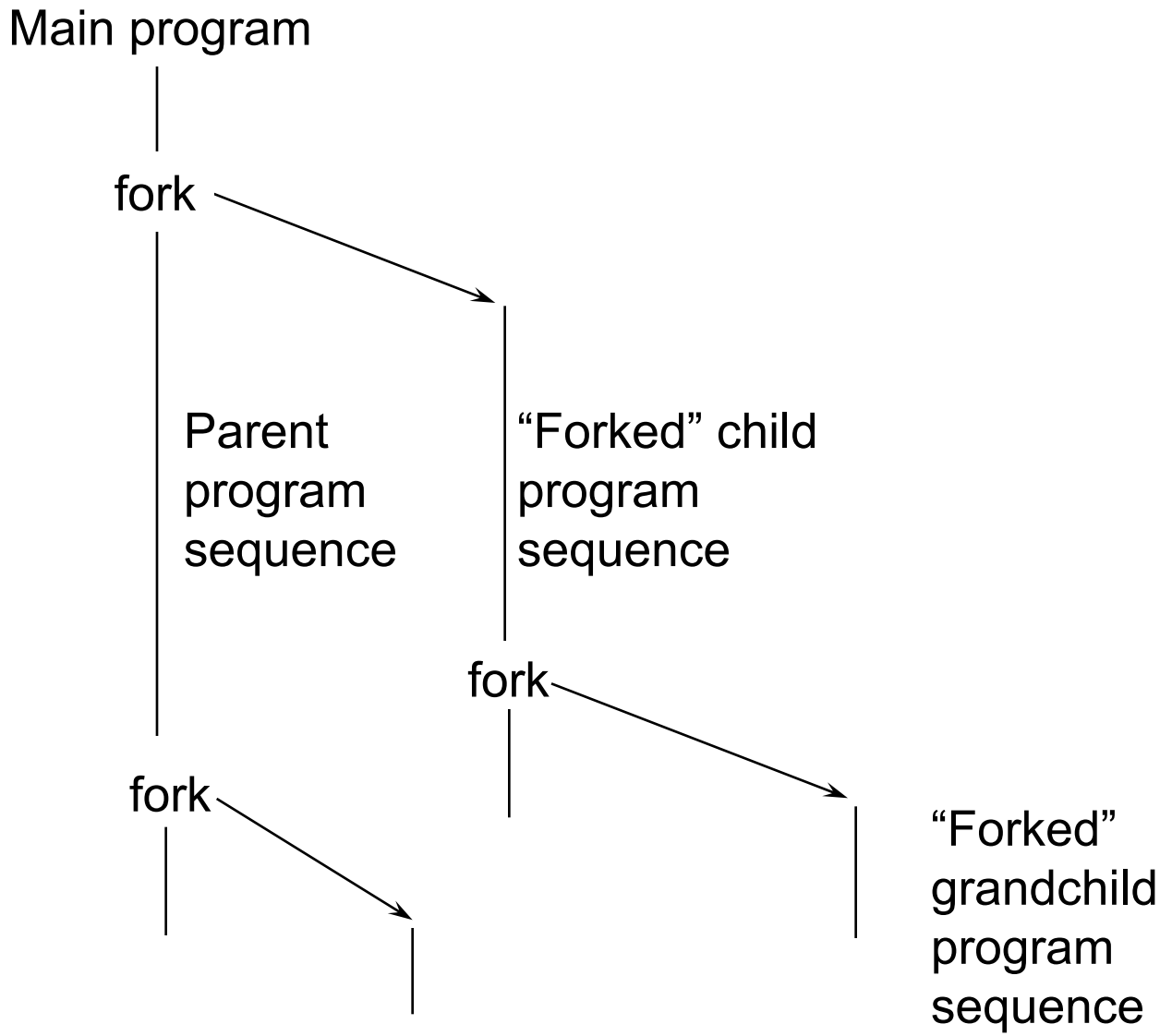
As used to dynamically create a process from a process



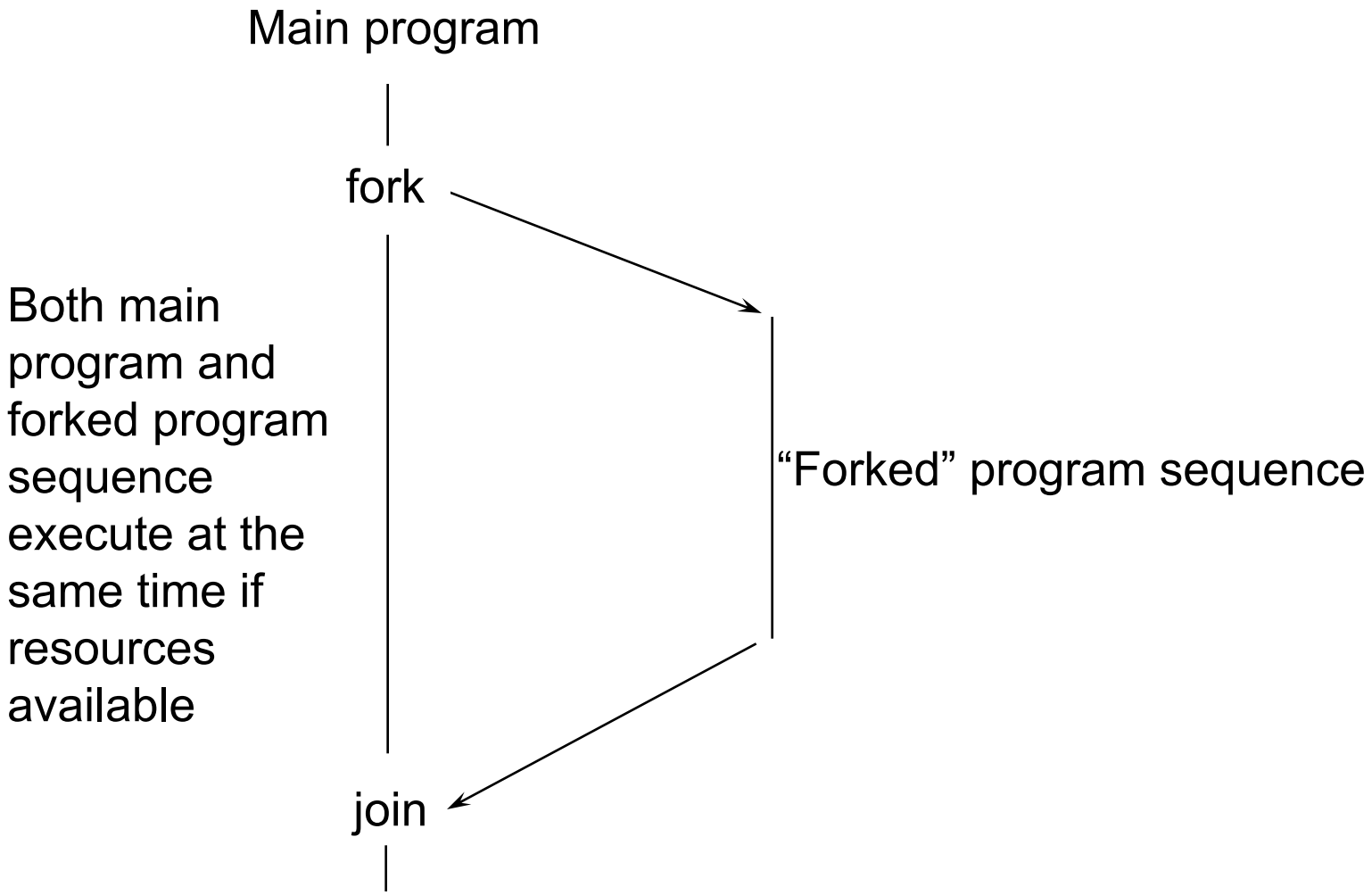
Although general concept of a fork does not require it, child process created by the Linux fork is a replica of parent program with same instructions and variable declarations even prior to fork. However, child process only starts at fork and both parent and child process execute onwards together.

Multiple and nested fork patterns

Both main program and forked program sequence execute at the same time if resources available



Fork-join pattern



Explicit “join” placed in calling parent program. Parent will not proceed past this point until child has terminated. Join acts a barrier synchronization point for both sequences. Child can terminate before join is reached, but if not, parent will wait for it terminate.

UNIX System Calls to Create Fork-Join Pattern

No join routine – use **exit(**) to exit from process and **wait()** to wait for child to complete:

```
        .
        .
pid = fork(); // returns 0 to child and positive # to parent (-1 if error)
if (pid == 0) {
        // code to be executed by child
} else {
        //code to be executed by parent
}
if (pid == 0) exit(0); else wait (0);        // join
        .
        .
```

Using processes in shared memory programming

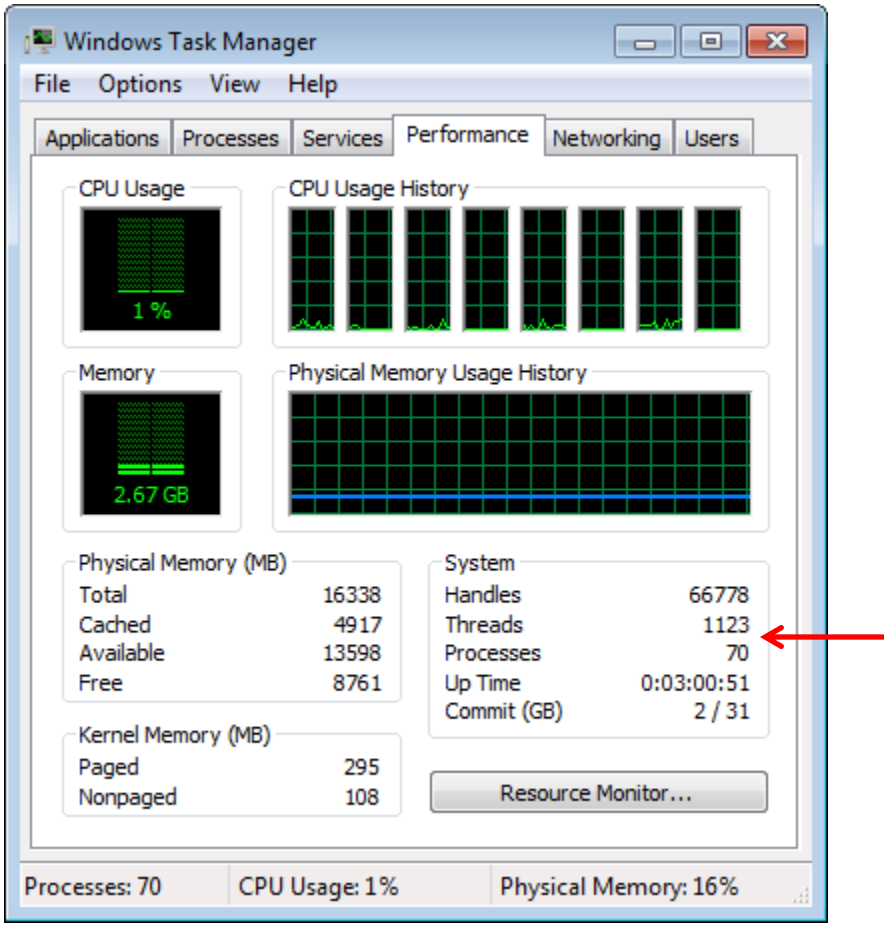
Concept could be used for shared memory parallel programming but not much used because of overhead of process creation and not being able to share data directly between processes

Threads

A separate program sequence that can be executed separately by a processor core, usually within a process.

Threads share memory space and global variables but have their own instruction pointer and stack.

An OS will manage the threads within each process.

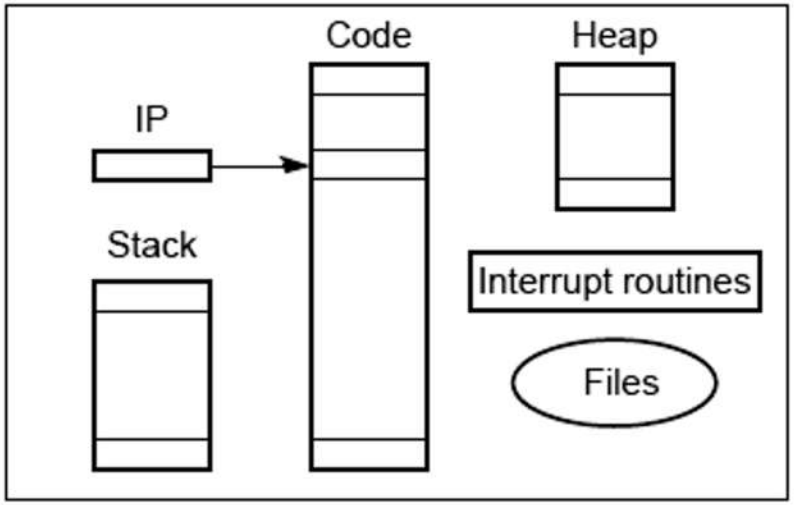


Example : destop i7-3770 quad core processor. Supports 8 threads simultaneously (hyperthreading)

Differences between a process and threads

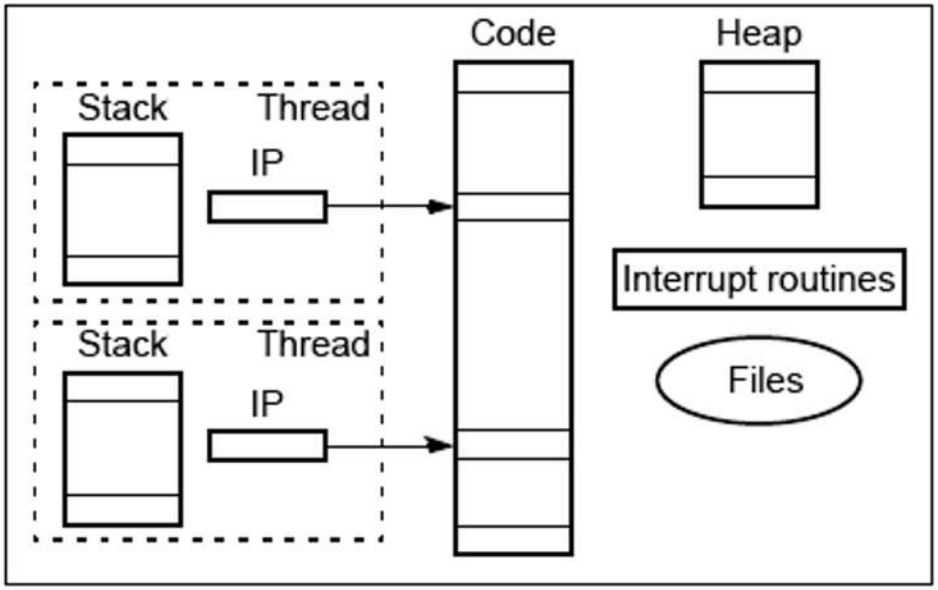
"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



Threads - shares the same memory space and global variables between routines.

(b) Threads



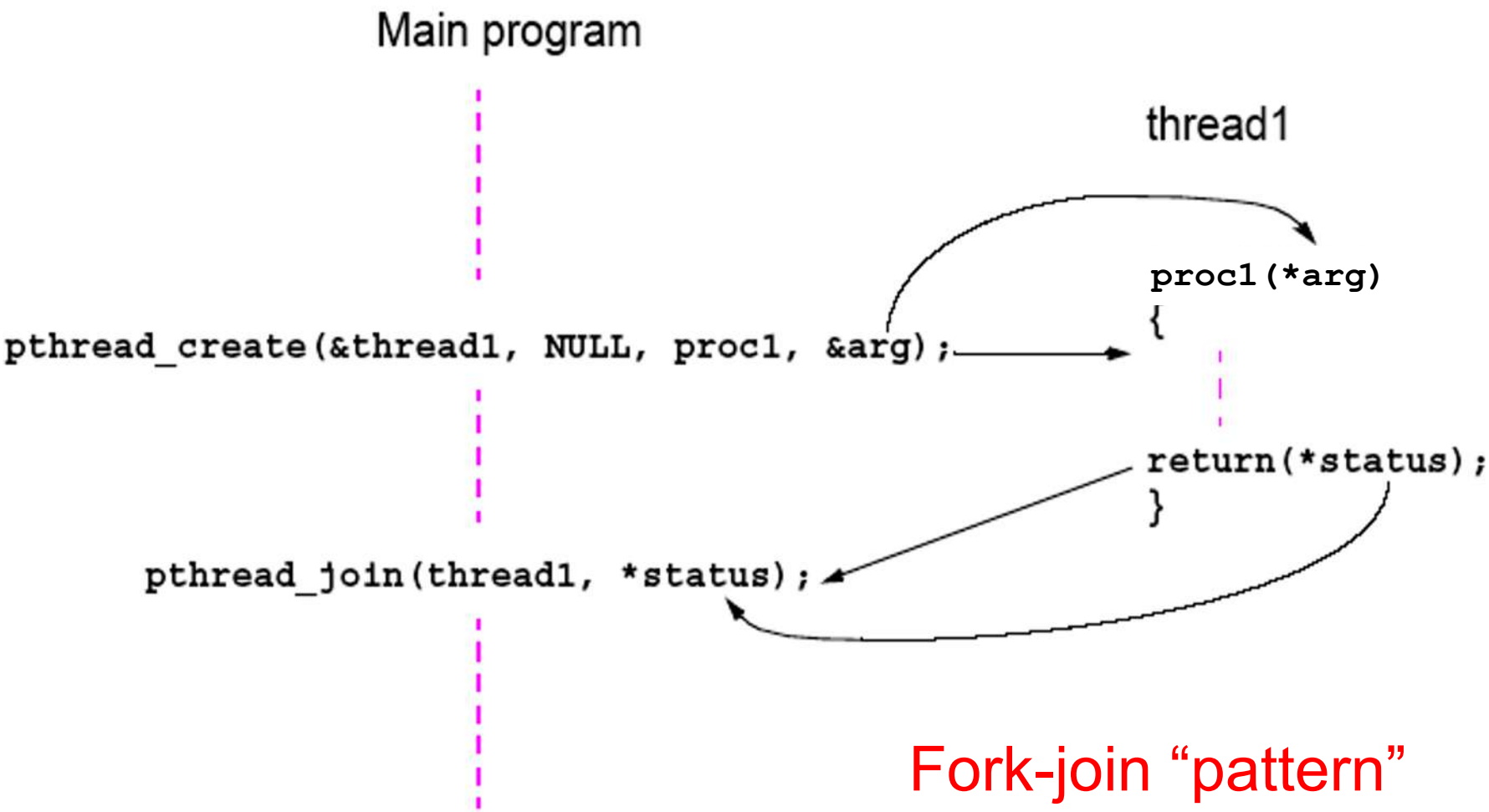
Threads in shared memory programming

A common approach, either directly creating threads (a low level approach) or indirectly.

Really low level -- Pthreads

IEEE Portable Operating System Interface, POSIX standard.

Executing a Pthread Thread



Fork-join “pattern”

int pthread_create(pthread_t *restrict *tid*, const pthread_attr_t *restrict *attr*, void*(**start_routine*)(void *), void *restrict *arg*);

Example:

```
#include <pthread.h>
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;
/* Create a thread with default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);
```

- *start_routine* is the function with which the new thread begins execution. When *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine*.

- When **pthread_create()** is successful, the ID of the created thread is stored in the location referred to as *tid*.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
//thread function definition
void *slave(void *argument) {
  int tid = *((int *) argument);
  printf("Hello World! It's me, thread %d!\n", tid);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int thread_args[NUM_THREADS];
  int i;

  for (i = 0; i < NUM_THREADS; i++) {                    // create threads
    thread_args[i] = i;
    printf("In main: creating thread %d\n", i);
    if ( pthread_create(&threads[i], NULL, slave, (void *) &thread_args[i] ) != 0)
            printf("Pthread_create fails");
  }

  for (i = 0; i < NUM_THREADS; i++) {                     // join threads
    if ( pthread_join(threads[i], NULL) != 0 )
            printf("Pthread_join fails");
    printf("In main: thread %d is complete\n", i);
  }
  printf("In main: All threads completed successfully\n");
  return 0;
}
```

Save your Program as **hello.c**
Compile: **gcc –o hello hello.c -lpthread**

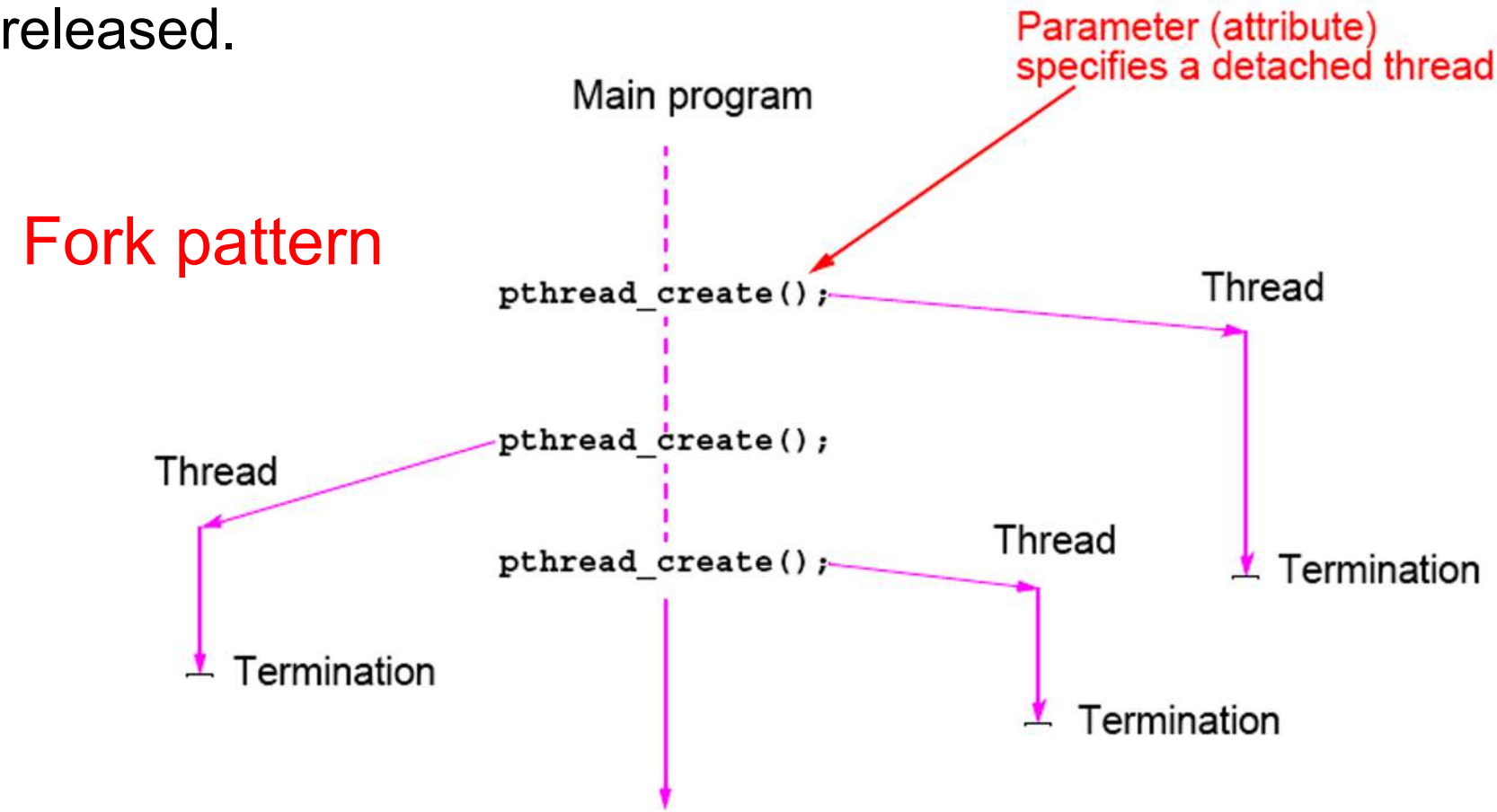
Sample Output

```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread 4!
Hello World! It's me, thread 0!
Hello World! It's me, thread 1!
Hello World! It's me, thread 2!
In main: thread 0 is complete
In main: thread 1 is complete
In main: thread 2 is complete
Hello World! It's me, thread 3!
In main: thread 3 is complete
In main: thread 4 is complete
In main: All threads completed successfully
```

Very simple to compile, Just add pthread library, but Pthreads very low level programming

Pthreads detached threads

Threads not joined are called *detached threads*. When detached threads terminate, they are destroyed and their resources released.



By default, the thread is created in joinable state unless the *attr* parameter is set to create thread in a detached state.

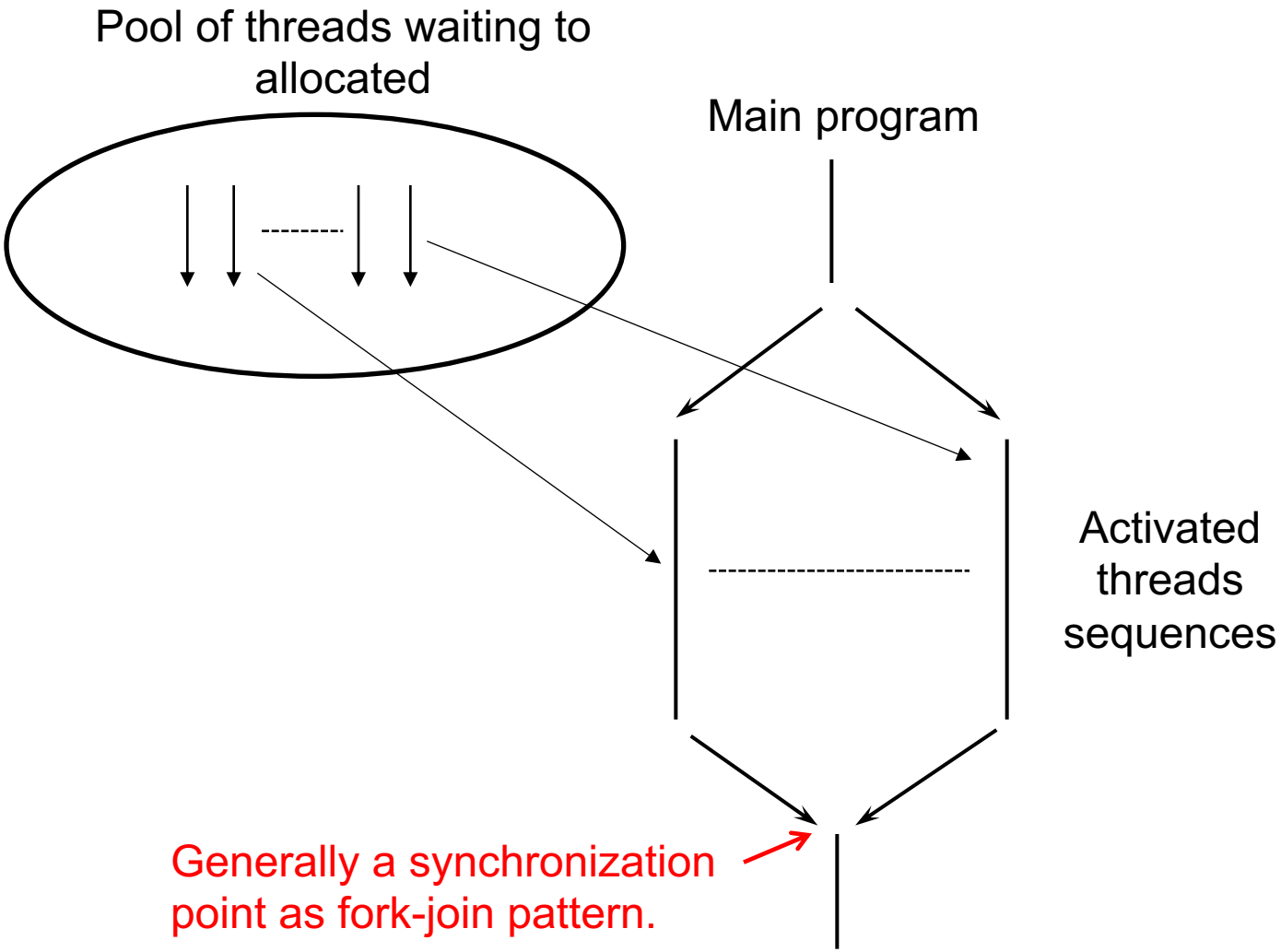
Thread pool pattern

Common to need group of threads to be used together from one execution point.

Group of threads readied to be allocated work and are brought into service.

Whether threads actually exist or are created just for then is an implementation detail.

Thread pool implies threads already created. Probably best as eliminates thread creation overhead.



Thread pool pattern or the thread team pattern is the underlying structure of OpenMP, see next.

Questions

LAU
School of Arts and Sciences
Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Programming with Shared Memory - 2

Issues with sharing data

Accessing Shared Data

Accessing shared data needs careful control.

Consider two threads each of which is to add one to a shared data item, **x**.

Location **x** is read, **x + 1** computed, and the result written back to the location **x**:

time

Instruction	Thread 1	Thread 2
x = x + 1;	**read x**	**read x**
	compute x + 1	**compute x + 1**
	write to x	**write to x**

Different possible interleavings:

Assume initially x = 0

Instruction	Thread 1	Thread 2
x = x + 1;	**read x**	
	compute x + 1	
	write to x	
		read x
		compute x + 1
		write to x

Get x = 2 finally.

Instruction	Thread 1	Thread 2
x = x + 1;	**read x**	
		read x
	compute x + 1	
		compute x + 1
	write to x	
		write to x

Get x = 1 finally.

Shared data

- A different scheduling of the execution by various processors can lead to different results (*race condition*).
- More importantly, simultaneous access can lead to incorrect or corrupted data.
- At least one of those accesses needs to be an update or write to need special handling.

Critical Section

A mechanism for ensuring that only one process accesses a particular resource at a time (write access).

critical section – a section of code for accessing resource
Arrange that only one such critical section is executed at a time.

This mechanism is known as *mutual exclusion*.

Concept also appears in operating systems.

Locks

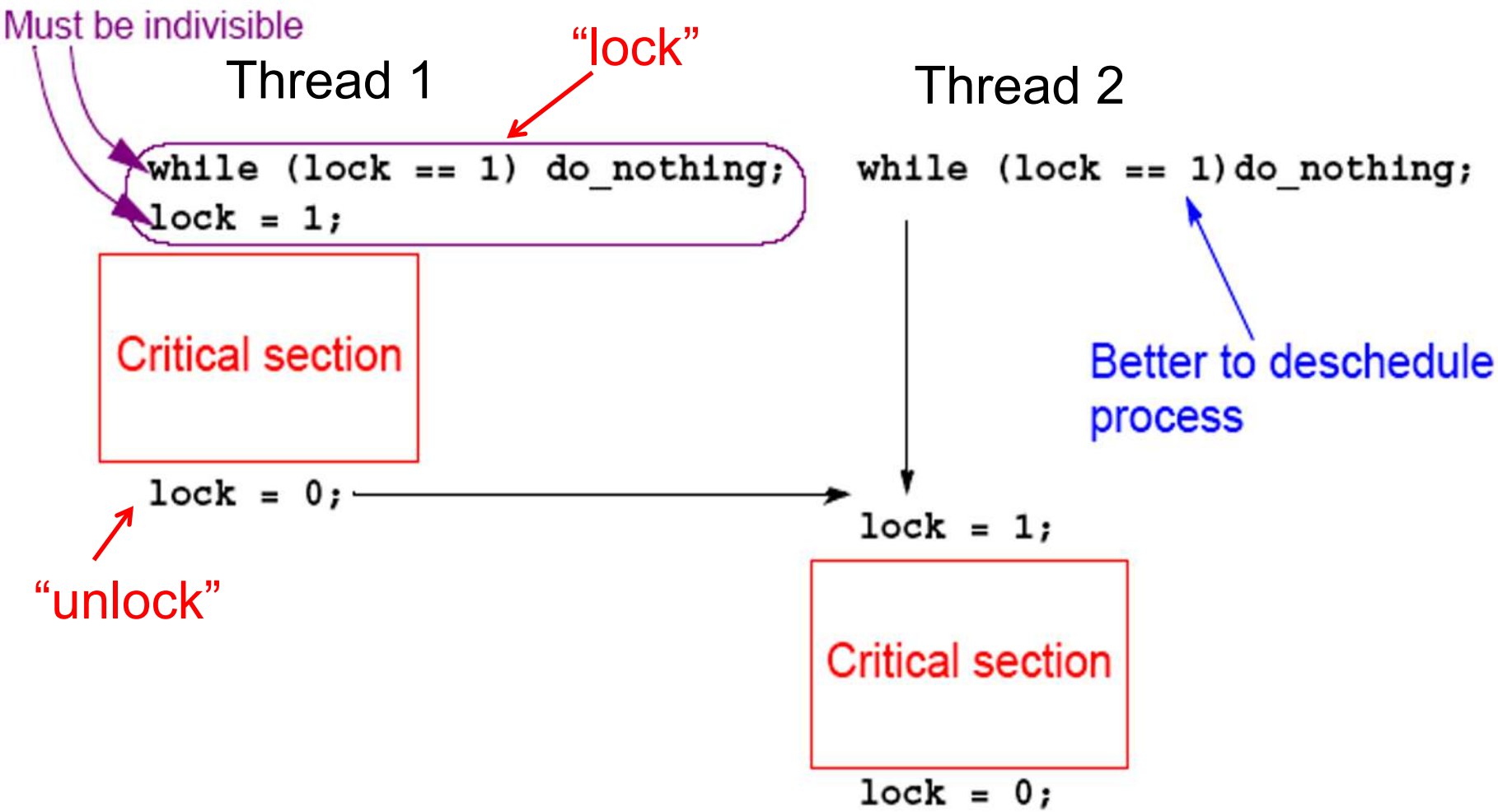
Simplest mechanism for ensuring mutual exclusion of critical sections.

A lock - a 1-bit variable that is a 1 to indicate that a process has entered the critical section and a 0 to indicate that no process is in the critical section.

Operates much like that of a door lock:

A process coming to the “door” of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.

Control of critical sections through busy waiting



Example two threads

Thread 1	Thread 2
⋮	⋮
lock(L1)	**lock(L1)**
x = x + 1;	**x = x + 1;**
unlock(L1)	**unlock(L1)**
⋮	⋮

where the lock variable **L1** is associated with accessing the shared variable **x**.

Pthreads example

Locks implemented in Pthreads with *mutually exclusive lock variables*, or "mutex" variables:

```
pthread_mutex_t   mutex1;        // declare mutex
…
pthread_mutex_lock(&mutex1);
        critical section
pthread_mutex_unlock(&mutex1);
                …
```

Same mutex variable

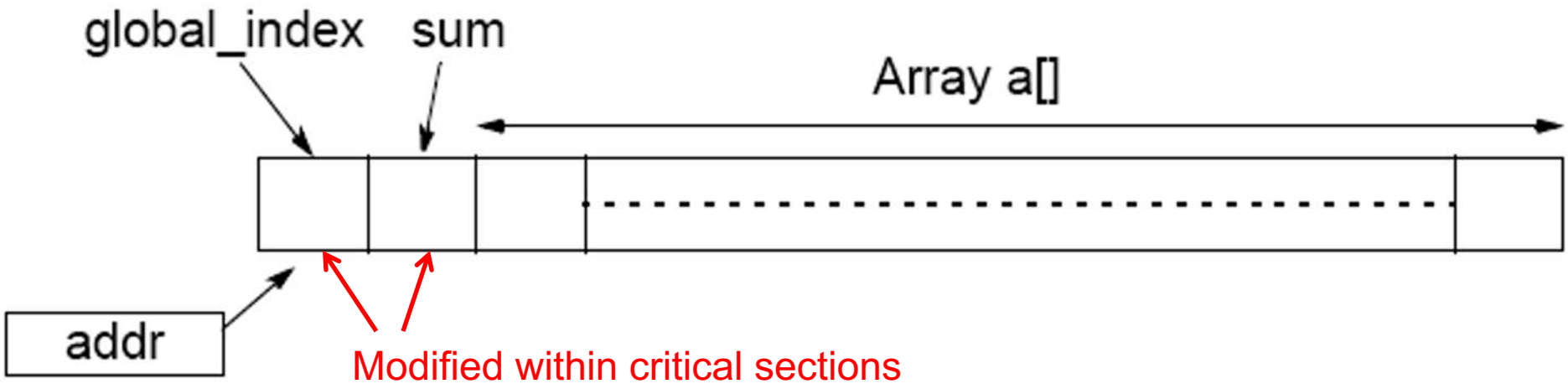
If a thread reaches mutex lock and finds it locked, it will wait for lock to open. If more than one thread waiting for lock to open, when it does open, system selects one thread to be allowed to proceed. Only thread that locks a mutex can unlock it.

Program example

To sum the elements of an array, **a[1000]**:

```
int sum, a[1000];
        sum = 0;
        for (i = 0; i < 1000; i++)
                sum = sum + a[i];
```

Pthreads program example



n threads created, each taking numbers from list to add to their local partial sums. When all numbers taken, threads can add their partial sums to a shared location sum.

Shared location global_index used by each thread to select next element of a[]. After global_index read, it is incremented in preparation for next element to be read.

```
#include <stdio.h>
#include <pthread.h>
#define N 10
#define NUM_THREADS 10
int a[N]; int global_index=0; int sum=0; // shared variables
pthread_mutex_t mutex1, mutex2; // mutexes, actually could share 1

void *slave(void *argument) {
   int tid = *((int *) argument);
   printf("Hello World! It's me, thread %d!\n", tid);
   int local_index, partial_sum = 0;
   do {
      pthread_mutex_lock(&mutex1);  // critical section
         local_index= global_index;
         global_index++;
      pthread_mutex_unlock(&mutex1);
      if (local_index < N) {
         printf("thread %d takes index %d\n",tid,local_index);
         partial_sum += a[local_index];
      }
   } while (local_index < N);
   pthread_mutex_lock(&mutex2);              // critical section
      sum+= partial_sum;
   pthread_mutex_unlock(&mutex2);
   return NULL;
}
```

```
int main(int argc, char *argv[]) {
   int i;
   pthread_t threads[NUM_THREADS];
   int thread_args[NUM_THREADS];
   pthread_mutex_init(&mutex1,NULL); pthread_mutex_init(&mutex2,NULL);

   for (i = 0; i < N; i++)                          // initialize array with some values
      a[i] = i+1;

   for (i = 0; i < NUM_THREADS; i++) {              // create threads
      thread_args[i] = i;
      if (pthread_create(&threads[i],NULL,slave,(void *) &thread_args[i]) != 0)
         perror("Pthread_create fails");
   }

   for (i = 0; i < NUM_THREADS; i++)                // join threads
      if(pthread_join(threads[i],NULL) != 0)
         perror("Pthread_join fails");
   printf("The sum of 1 to %d is %d\n",N, sum);
   return (0);
}
```

Sample Output

```
Hello World! It's me, thread 8!
thread 8 takes index 0
thread 8 takes index 1
thread 8 takes index 2
thread 8 takes index 3
thread 8 takes index 4
thread 8 takes index 5
thread 8 takes index 6
Hello World! It's me, thread 5!
thread 5 takes index 8
thread 5 takes index 9
Hello World! It's me, thread 7!
Hello World! It's me, thread 9!
Hello World! It's me, thread 6!
thread 8 takes index 7
Hello World! It's me, thread 4!
Hello World! It's me, thread 3!
Hello World! It's me, thread 2!
Hello World! It's me, thread 1!
Hello World! It's me, thread 0!
The sum of 1 to 10 is 55
```

Program on VM in directory **Pthreads** as **critical.c**

Compile:

cc –o critical critical.c -lpthread

Another run

```
Hello World! It's me, thread 5!
Hello World! It's me, thread 1!
thread 1 takes index 0
thread 1 takes index 1
thread 1 takes index 2
thread 1 takes index 3
thread 1 takes index 4
thread 1 takes index 5
thread 1 takes index 6
thread 1 takes index 7
thread 1 takes index 8
thread 1 takes index 9
Hello World! It's me, thread 6!
Hello World! It's me, thread 8!
Hello World! It's me, thread 7!
Hello World! It's me, thread 4!
Hello World! It's me, thread 3!
Hello World! It's me, thread 2!
Hello World! It's me, thread 0!
Hello World! It's me, thread 9!
The sum of 1 to 10 is 55
```

Critical Sections Serializing Code

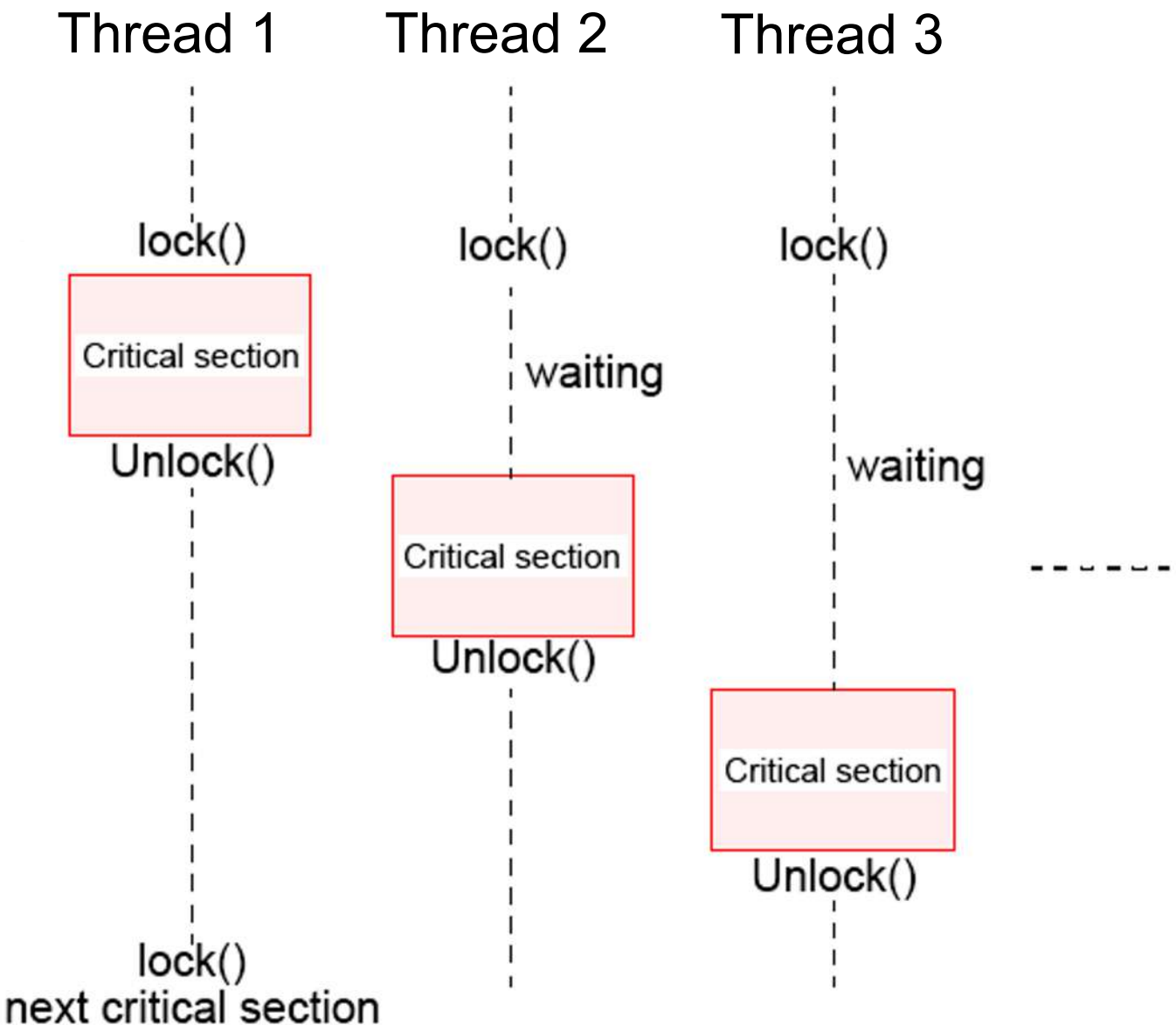
High performance programs should have as few as possible critical sections as their use can serialize the code.

Suppose, all processes happen to come to their critical section together.

They will execute their critical sections one after the other.

In that situation, the execution time becomes almost that of a single processor.

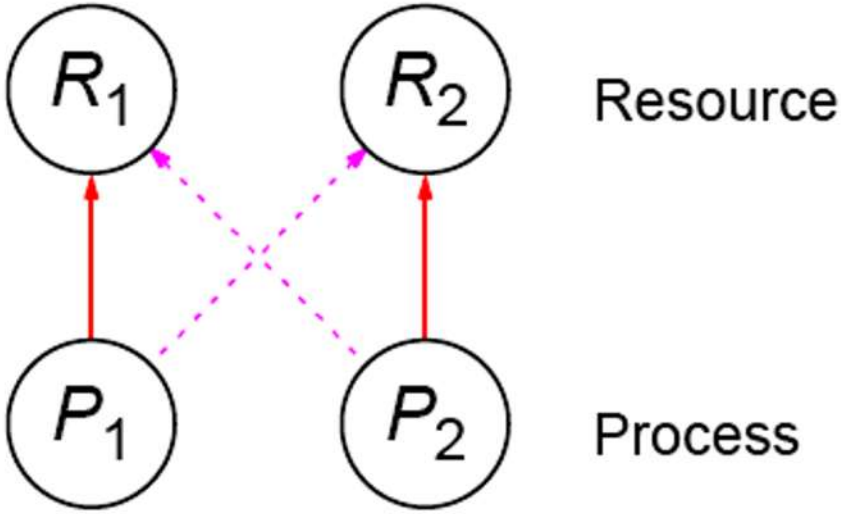
Illustration



Deadlock

Can occur with two processes/threads when one requires a resource held by the other, and this process/thread requires a resource held by the first process/thread.

Two-process deadlock



"Deadly embrace"

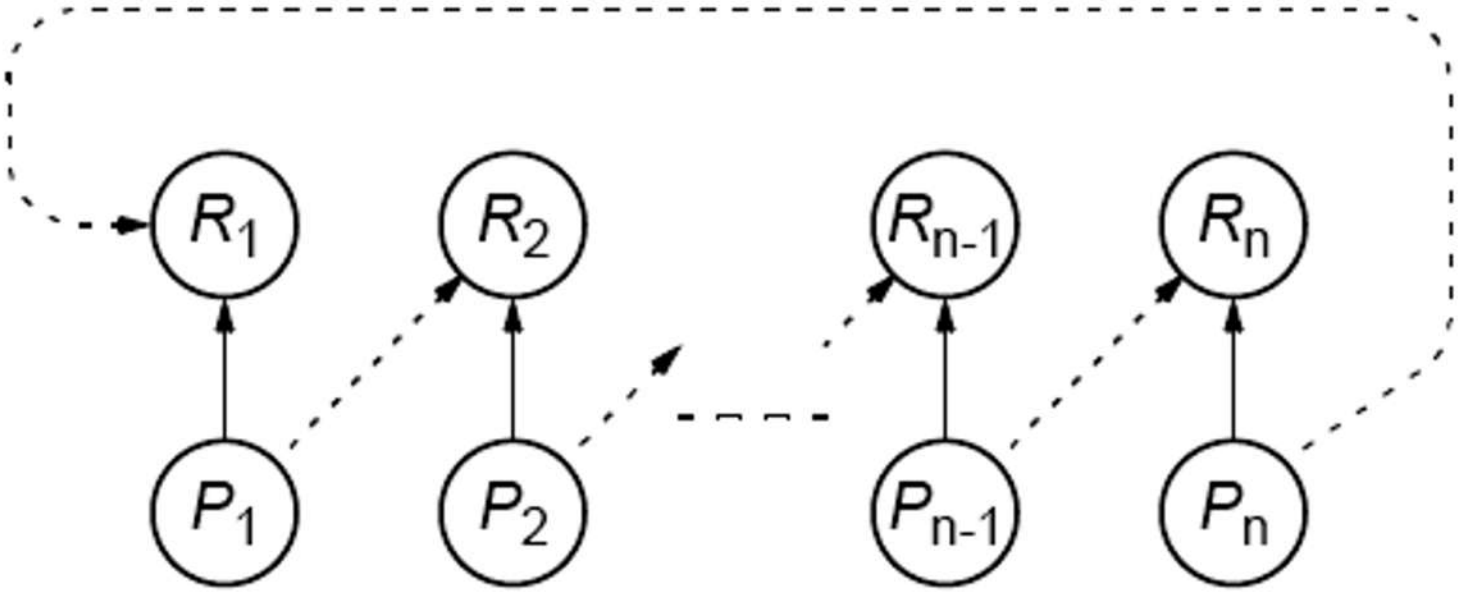
Deadlock Example

Two threads each wanting to access shared variables **x** and **y**.

Thread 1	Thread 2
⋮	⋮
lock(L1);	**lock(L2);**
lock(L2);	**lock(L1);**
temp = x + y;	**temp = x + y;**
unlock(L2);	**unlock(L1);**
unlock(L1);	**unlock(L2);**
⋮	⋮

where the lock variable **L1** is associated with accessing the shared variable **x** and the lock variable **L2** is associated with accessing the shared variable **y**.

Deadlock can also occur in a circular fashion with several processes having a resource wanted by another.



Condition Variables

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals (“polled”) within a critical section.

Very time-consuming and unproductive exercise.

Can be overcome by introducing so-called *condition variables*.

Condition variables in Pthreads.

Pthread Condition Variables

Pthreads arrangement for signal and wait:

Thread 1

```
action()
{
        .
        .
pthread_mutex_lock(&mutex1);
while (c <> 0)
  pthread_cond_wait(cond1,mutex1);
pthread_mutex_unlock(&mutex1);
take_action();
        .
        .
}
```

Thread 2

```
counter()
{
    .
    .
pthread_mutex_lock(&mutex1);
c--;
if (c == 0) pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);
  .
  .
  .
}
```

Signals *not* remembered - threads must already be waiting for a signal to receive it.

Semaphores

A slightly more powerful mechanism than a binary lock. A positive integer (including zero) operated upon by two operations:

P operation on semaphore *s* - Waits until ***s*** is greater than zero and then decrements ***s*** by one and allows the process to continue.

V operation **on semaphore *s* -** Increments ***s*** by one and releases one of the waiting processes (if any). Mechanism for activating waiting processes implicit in **P** and **V** operations. Though exact algorithm not specified, algorithm expected to be fair.

P and **V** operations are performed indivisibly. Processes delayed by **P**(s) are kept in abeyance until released by a **V**(s) on the same semaphore.

Devised by Dijkstra in 1968.
Letter **P** from Dutch word *passeren*, meaning “to pass”
Letter **V** from Dutch word *vrijgeven*, meaning “to release”

Mutual exclusion of critical sections can be achieved with one semaphore having the value 0 or 1 (a binary semaphore), which acts as a lock variable, but **P** and **V** operations include a process scheduling mechanism:

Thread 1	***Thread 2***	***Thread 3***
Noncritical section	Noncritical section	Noncritical section
…	**…**	**…**
P(s)	**P(s)**	**P(s)**
Critical section	**Critical section**	**Critical section**
V(s)	**V(s)**	**V(s)**
…	**…**	**…**
Noncritical section	Noncritical section	Noncritical section

General semaphore
(or counting semaphore)

Can take on positive values other than zero and one.

Provide, for example, a means of recording number of “resource units” available or used. Can solve producer/ consumer problems - more on that in operating system courses.

Semaphore routines exist for UNIX processes.

Pthread Semaphore

Does not exist in Pthreads as such, though they can be written.

Do exist in real-time extension to Pthreads.

Monitor

Suite of procedures that provides only way to access shared resource. ***Only one process can use a monitor procedure at any instant.***

Could be implemented using a semaphore or lock to protect entry, i.e.:

```
monitor_proc1() {
lock(x);

        monitor body

unlock(x);
return;
}
```

Questions

GPU Programming

Lecture 1.1 – Introduction

Introduction to Heterogeneous Parallel Computing

Objectives

- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To understand why winning applications increasingly use both types of devices

Latency vs. Throughput

- Latency: The time required to perform some action.
  - measured in units of time.
- Throughput: The number of actions executed in a unit of time.
  - Measured in units of what is produced per unit of time.
- E.g. An assembly line manufactures GPUs. It takes 6 hours to manufacture a GPU but the assembly line can manufacture 100 GPUs per day.

Interesting Article:
D. Patterson, “Latency lags bandwidth”, IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD, San Jose, CA, USA, 2005.

Latency vs. Throughput

– A real life example: Need to go on a trip of 400Km.

Car: 2 people, 200 km/hour
Bus: 40 people, 50 km/hour

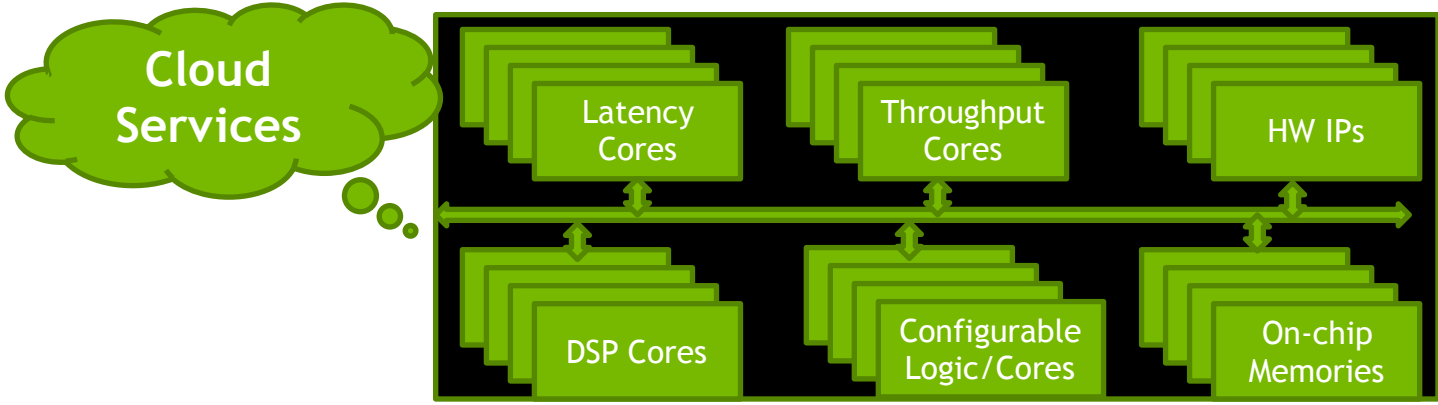
Travel Distance: 400 Km

Car:
Latency=?? Hours
Throughput=?? People/Hour

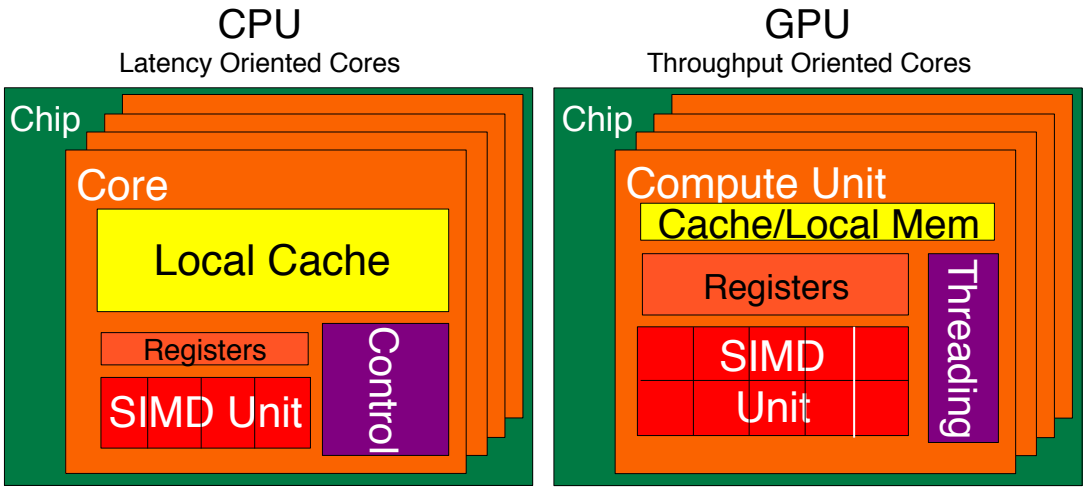
Bus:
Latency=?? Hours
Throughput=?? People/Hour

Heterogeneous Parallel Computing

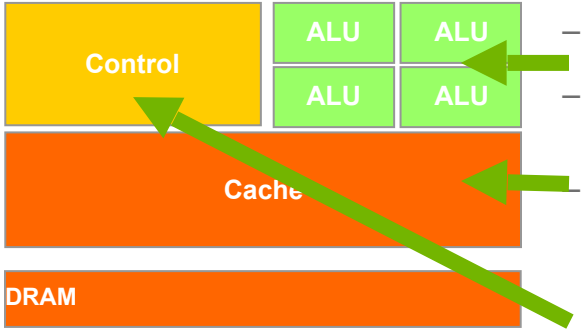
– Use the best match for the job (heterogeneity in mobile SOC)



CPU and GPU are designed very differently

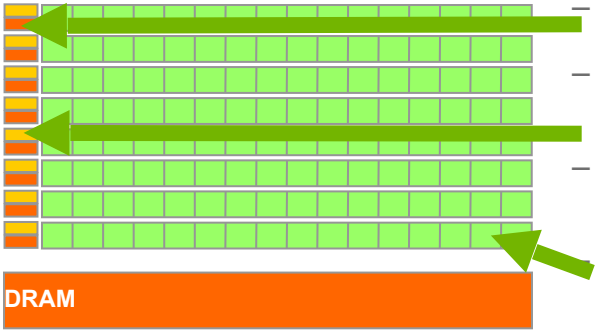


CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design



- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10X+ faster than CPUs for parallel code

Heterogeneous Parallel Computing in Many Disciplines

- Financial Analysis
- Scientific Simulation
- Engineering Simulation
- Data Intensive Analytics
- Medical Imaging
- Digital Audio Processing
- Digital Video Processing
- Computer Vision
- Biomedical Informatics
- Electronic Design Automation
- Statistical Modeling
- Numerical Methods
- Ray Tracing Rendering
- Interactive Physics

GPU Programming

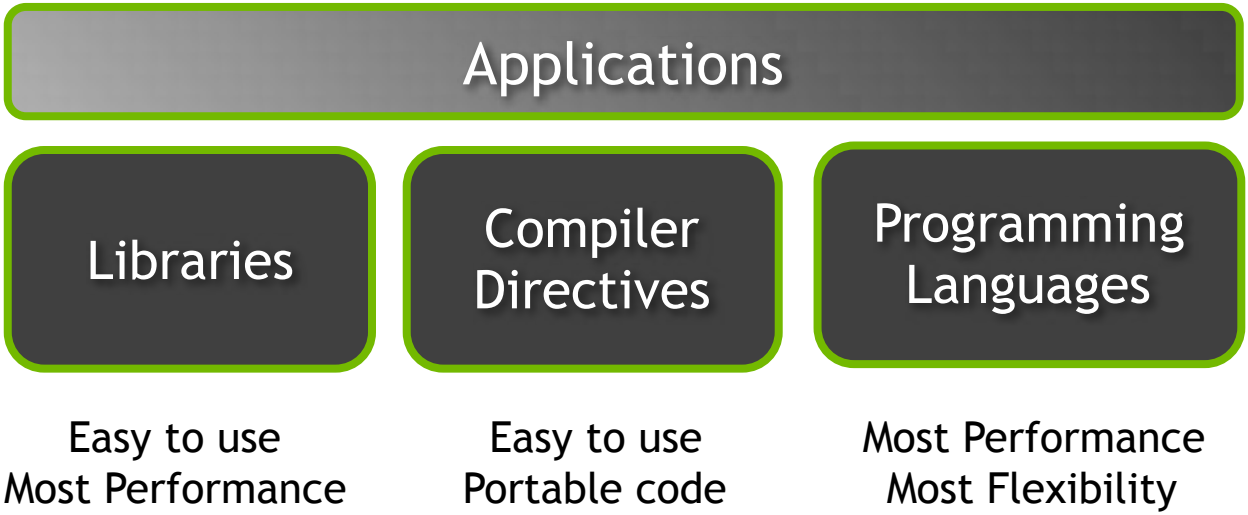
Lecture 2.1 - Introduction to CUDA C

CUDA C vs. Thrust vs. CUDA Libraries

Objective

- To learn the main venues and developer resources for GPU computing
- Where CUDA C fits in the big picture

3 Ways to Accelerate Applications

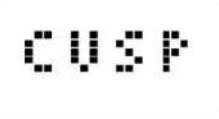


Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

GPU Accelerated Libraries

Linear Algebra
FFT, BLAS,
SPARSE, Matrix



Numerical & Math
RAND, Statistics

Data Struct. & AI
Sort, Scan, Zero Sum

Visual Processing
Image & Video

NVIDIA
Video
Encode

Vector Addition in Thrust

```
thrust::device_vector<float> deviceInput1(inputLength);
thrust::device_vector<float> deviceInput2(inputLength);
thrust::device_vector<float> deviceOutput(inputLength);

thrust::copy(hostInput1, hostInput1 + inputLength,
deviceInput1.begin());
thrust::copy(hostInput2, hostInput2 + inputLength,
deviceInput2.begin());

thrust::transform(deviceInput1.begin(), deviceInput1.end(),
deviceInput2.begin(), deviceOutput.begin(),
      thrust::plus<float>());
```

Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement
- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain:** Performance of code can vary across compiler versions

OpenACC

– Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
copyout(output[0:inputLength])
    for(i = 0; i < inputLength; ++i) {
        output[i] = input1[i] + input2[i];
    }
```

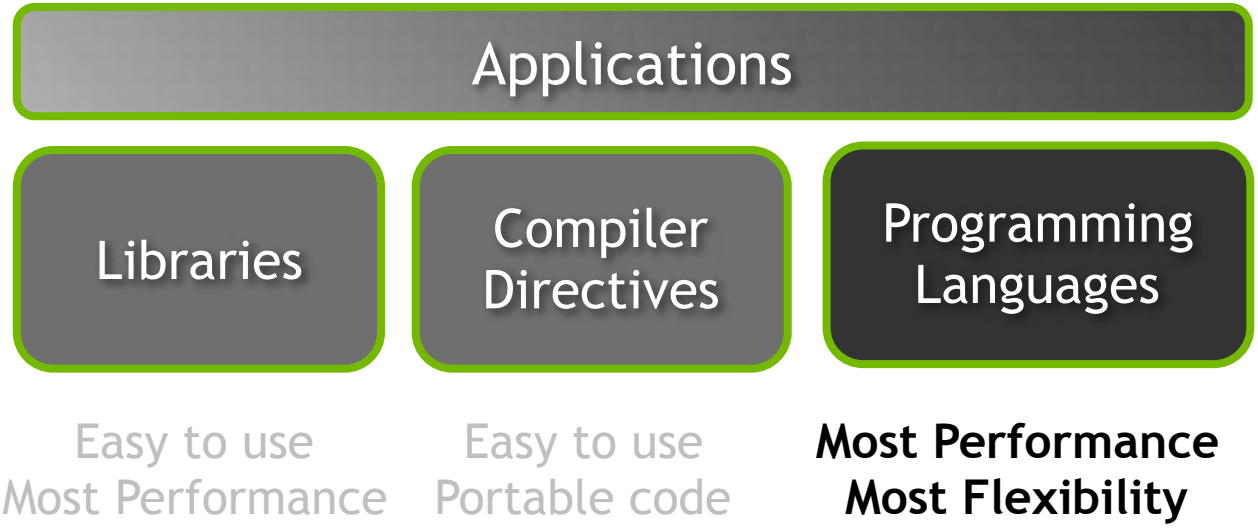
Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** Programmer has best control of parallelism and data movement
- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types
- **Verbose:** The programmer often needs to express more details

GPU Programming Languages

Numerical analytics	MATLAB Mathematica, LabVIEW
Fortran	CUDA Fortran
C	CUDA C
C++	CUDA C++
Python	PyCUDA, Copperhead, Numba
F#	Alea.cuBase

CUDA - C


Applications
Libraries
Compiler Directives
Programming Languages
Easy to use
Most Performance
Easy to use
Portable code
Most Performance
Most Flexibility

GPU Programming

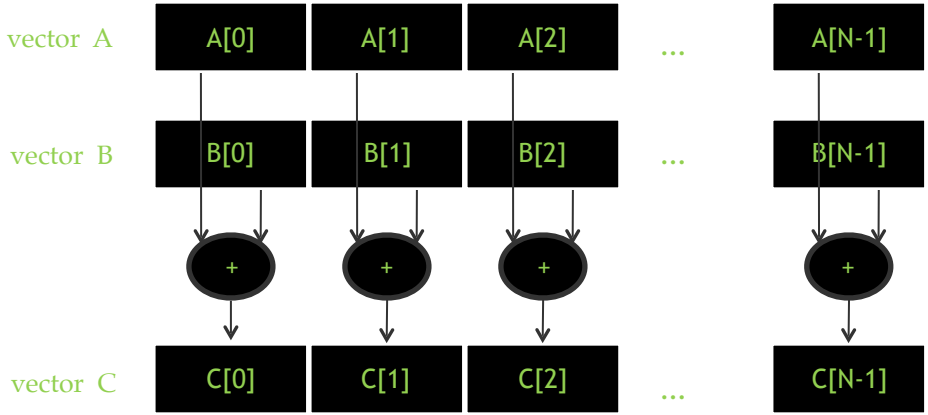
Lecture 2 - Introduction to CUDA C

Memory Allocation and Data Movement API Functions

Objective

- To learn the basic API functions in CUDA host code
- Device Memory Allocation
- Host-Device Data Transfer

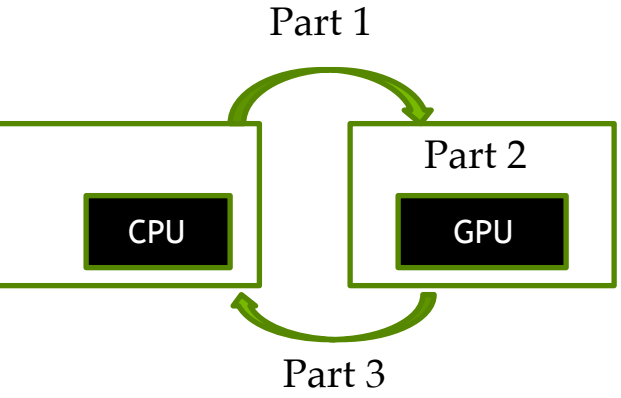
Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

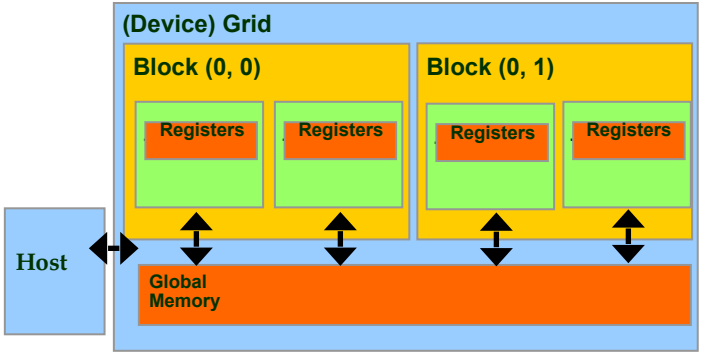
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

Heterogeneous Computing vecAdd CUDA Host Code




```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n* sizeof(float);
   float *d_A, *d_B, *d_C;
   // Part 1
   // Allocate device memory for A, B, and C
   // copy A and B to device memory

   // Part 2
   // Kernel launch code – the device performs the actual vector addition

   // Part 3
                    // copy C from the device memory
   // Free device vectors
}
```

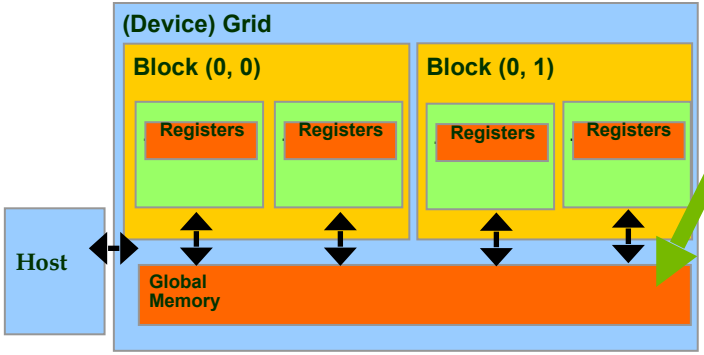
Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

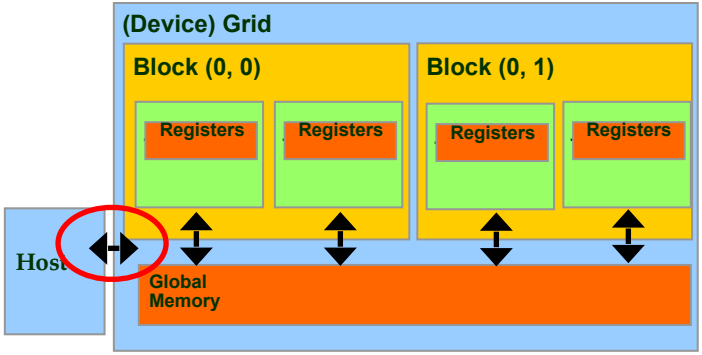
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
- Allocates an object in the device global memory
- Two parameters
  - **Address of a pointe**r to the allocated object
  - **Size of** allocated object in terms of bytes
- cudaFree()
- Frees object from device global memory
- One parameter
  - **Pointer** to freed object

Host-Device Data Transfer API functions



- cudaMemcpy()
- memory data transfer
- Requires four parameters
- Pointer to destination
- Pointer to source
- Number of bytes copied
- Type/Direction of transfer

- Transfer to device is asynchronous

Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n * sizeof(float); float *d_A, *d_B, *d_C;
   cudaMalloc((void **) &d_A, size);
   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    // Kernel invocation code – to be shown later
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

In Practice, Check for API Errors in Host Code

```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err),
__FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

GPU Programming

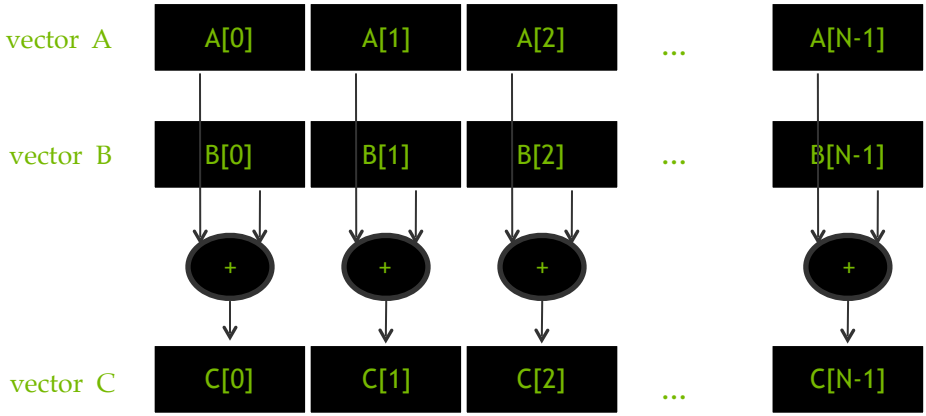
Lecture 3 – Introduction to CUDA C

Threads and Kernel Functions

Objective

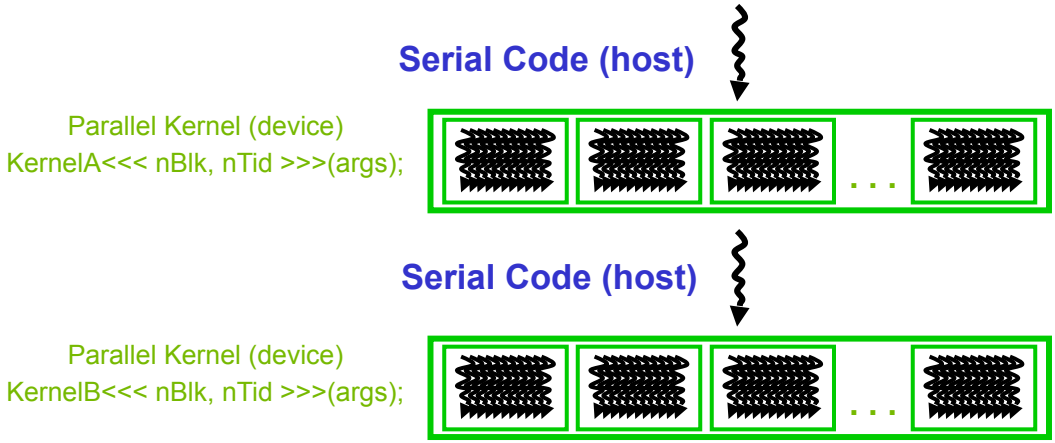
- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
- Hierarchical thread organization
- Launching parallel execution
- Thread index to data index mapping

Data Parallelism - Vector Addition Example

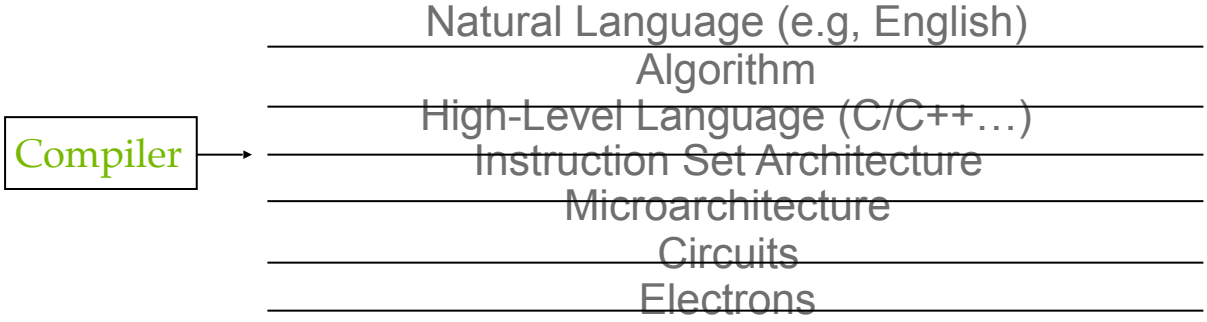


CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code



From Natural Language to Electrons



©Yale Patt and Sanjay Patel, *From bits and bytes to gates and beyond*

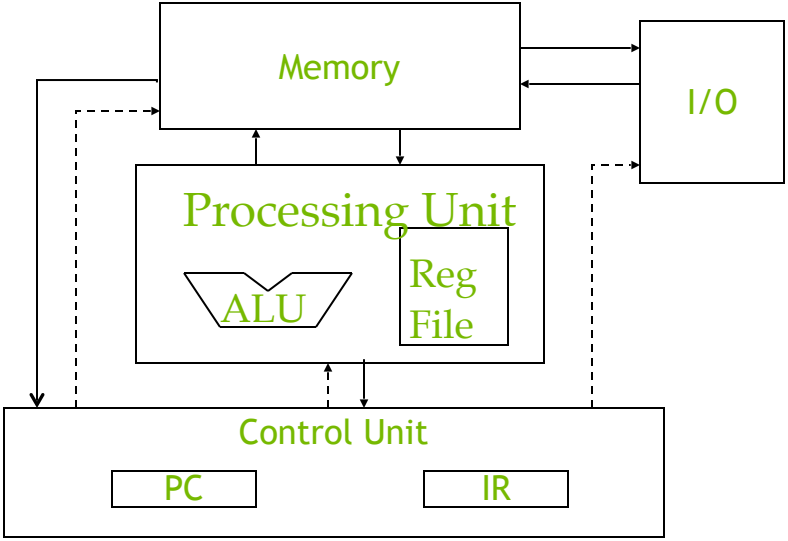
A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
- Both CPUs and GPUs are designed based on (different) instruction sets

- Program instructions operate on data stored in memory and/or registers.

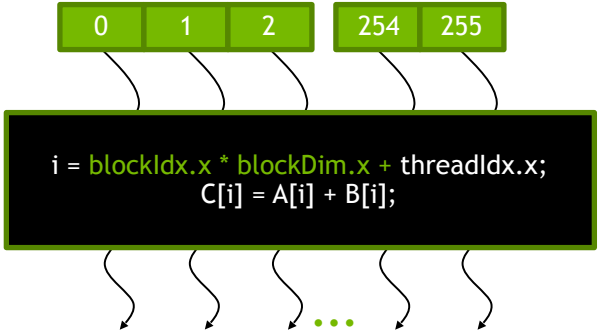
A Thread as a Von-Neumann Processor

A thread is a “virtualized” or “abstracted” Von-Neumann Processor

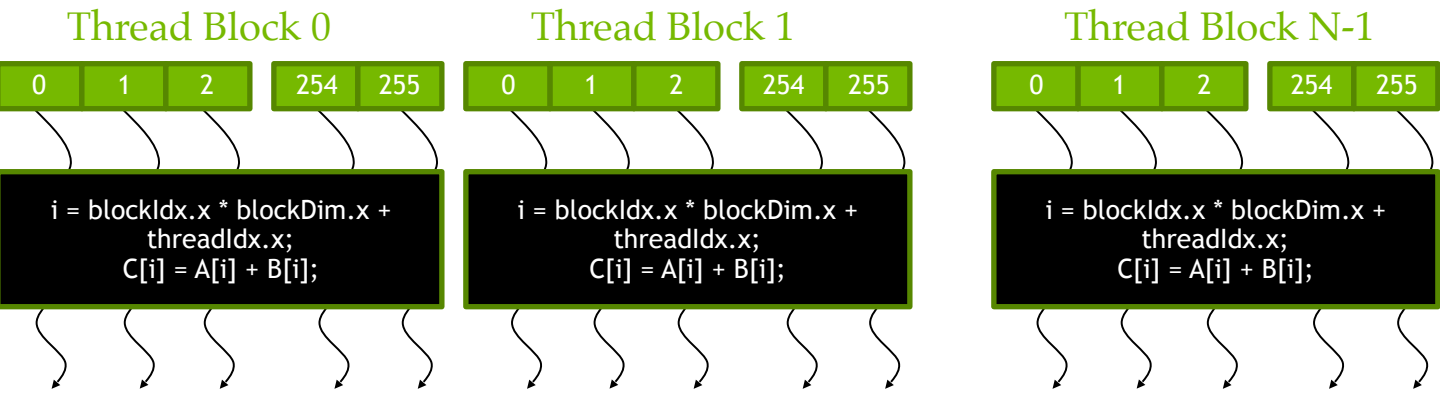


Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



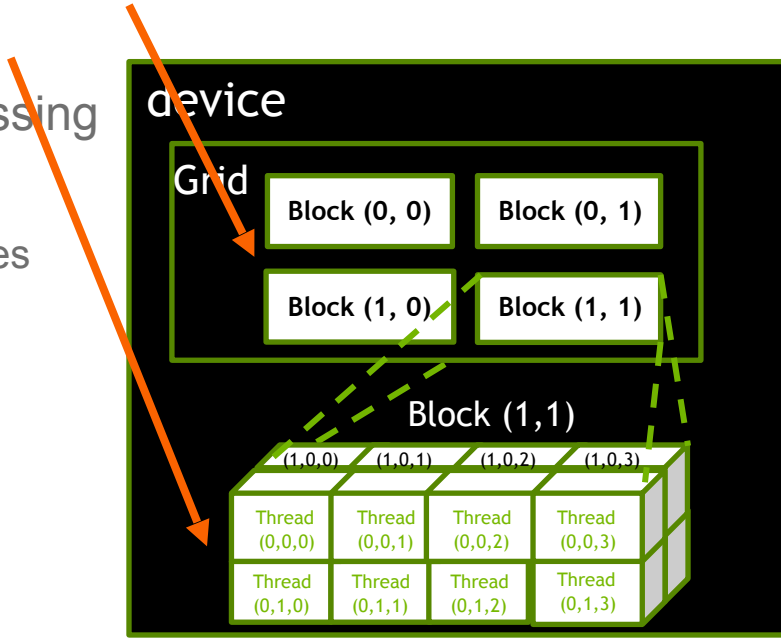
Thread Blocks: Scalable Cooperation



- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



GPU Programming

Lecture 4 - CUDA Parallelism Model

Kernel-Based SPMD Parallel Programming

Objective

- To learn the basic concepts involved in a simple CUDA kernel function
- Declaration
- Built-in variables
- Thread index to data index mapping

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int
  n)
{
  int i = threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Example: Vector Addition Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and copies omitted
 // Run ceil(n/256.0) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C,
  n);
}
```

The ceiling function makes sure that there are enough threads to cover all elements.

More on Kernel Launch (Host Code)

Host Code

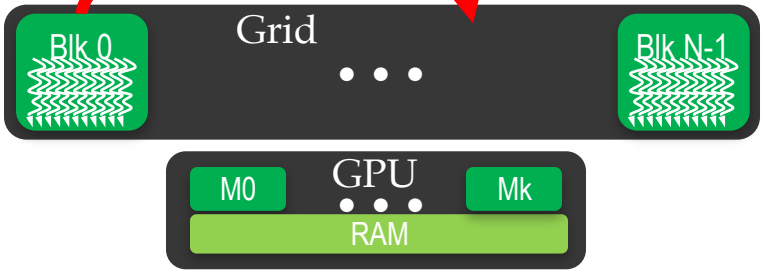
```
void vecAdd(float* h_A, float* h_B, float* h_C, int
  n)
{
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C,
  n);
}
```

This is an equivalent way to express the ceiling function.

Kernel execution in a nutshell

```
__host__
void vecAdd(…)
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);
vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B,d_
C,n);
}
```

```
__global__
void vecAddKernel(float *A,
     float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x
           + threadIdx.x;

   if( i<n ) C[i] = A[i]+B[i];
}
```



More on CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

CSC 447 Parallel Programming

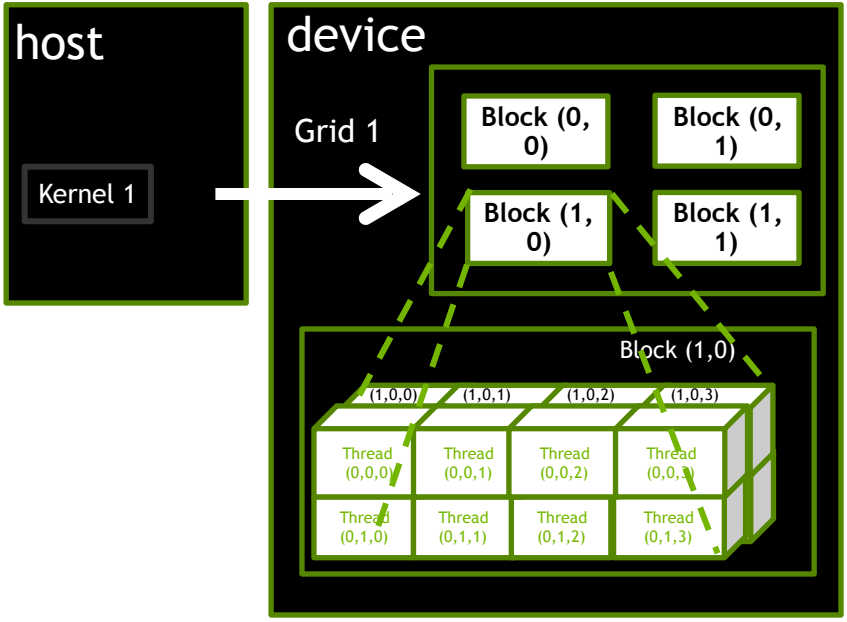
Lecture 6 – CUDA Parallelism Model

Multidimensional Kernel Configuration

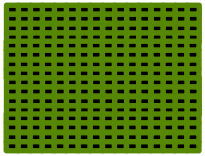
Objective

- To understand multidimensional Grids
- Multi-dimensional block and thread indices
- Mapping block/thread indices to data indices

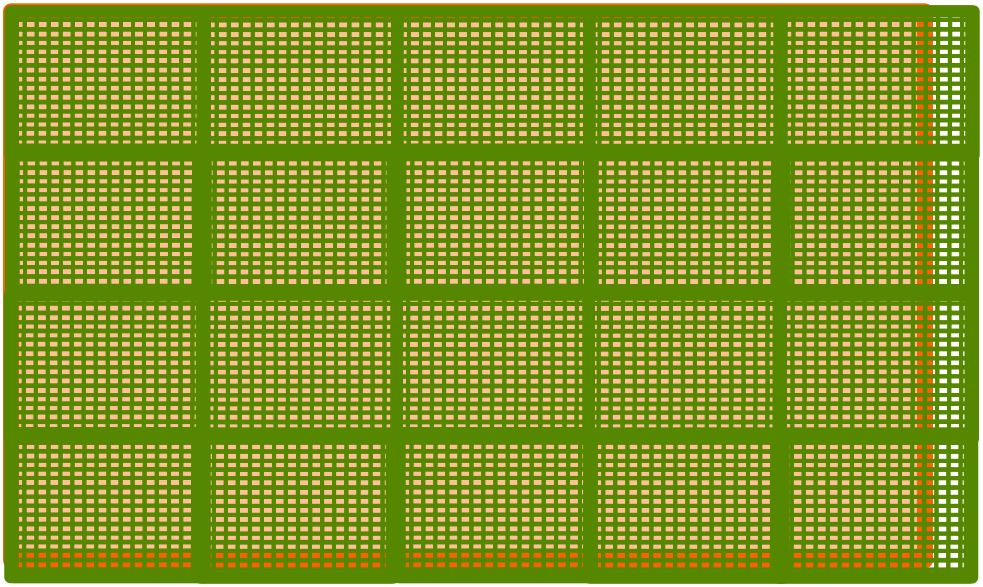
A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid

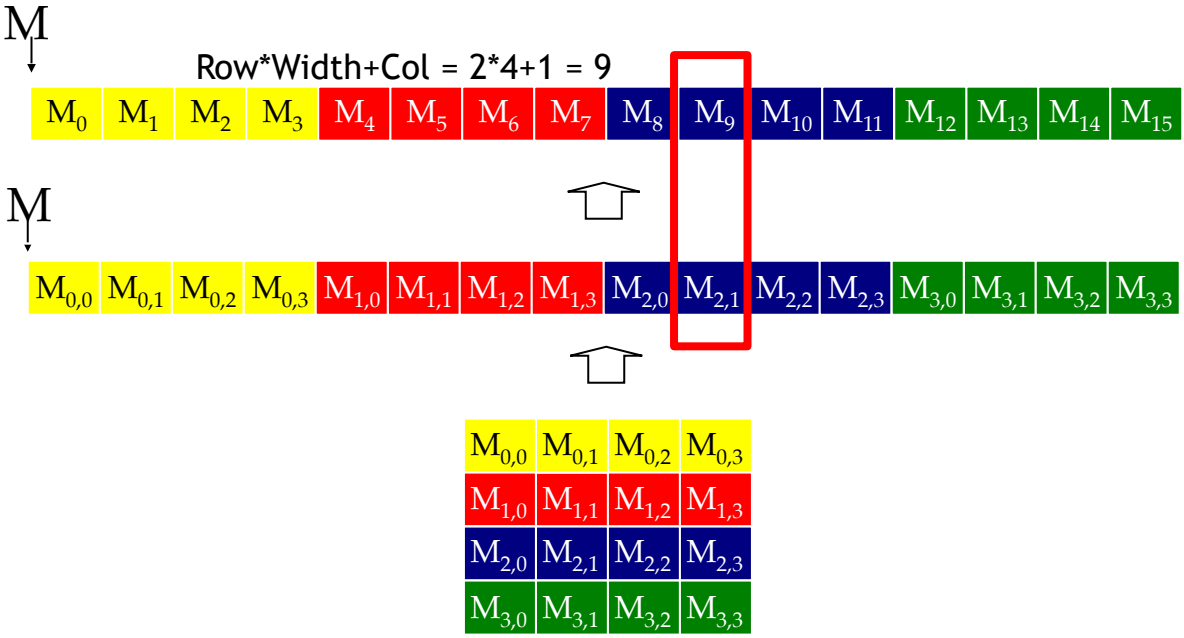


1616 blocks



6276 picture

Row-Major Layout in C/C++



Source Code of a PictureKernel

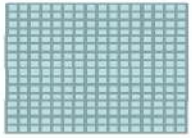
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                                    int height, int width)
{

  // Calculate the row # of the d_Pin and d_Pout element
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < height) && (Col < width)) {
    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
  }
}
```

Scale every pixel value by 2.0

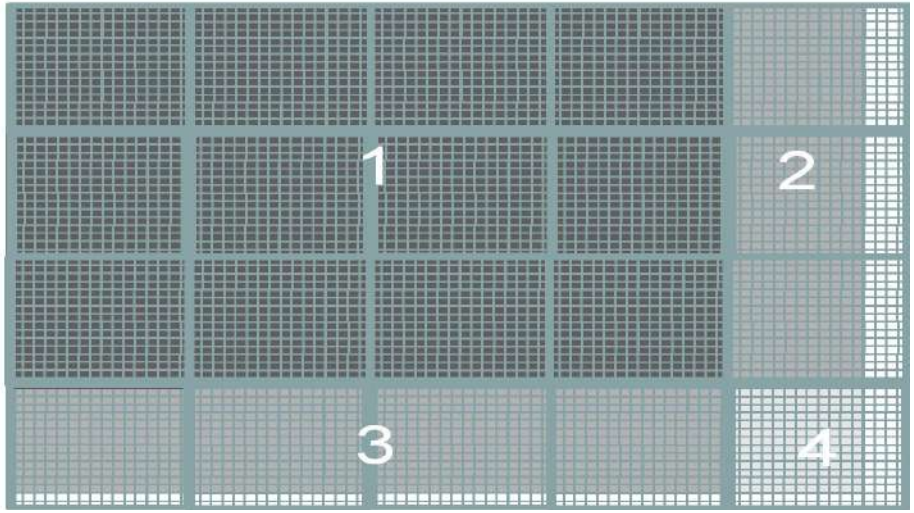
Host Code for Launching PictureKernel

```
// assume that the picture is mn,
// m pixels in y dimension and n pixels in x dimension
// input d_Pin has been allocated on and copied to device
// output d_Pout has been allocated on device
...
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
...
```

Covering a 6276 Picture with 1616 Blocks

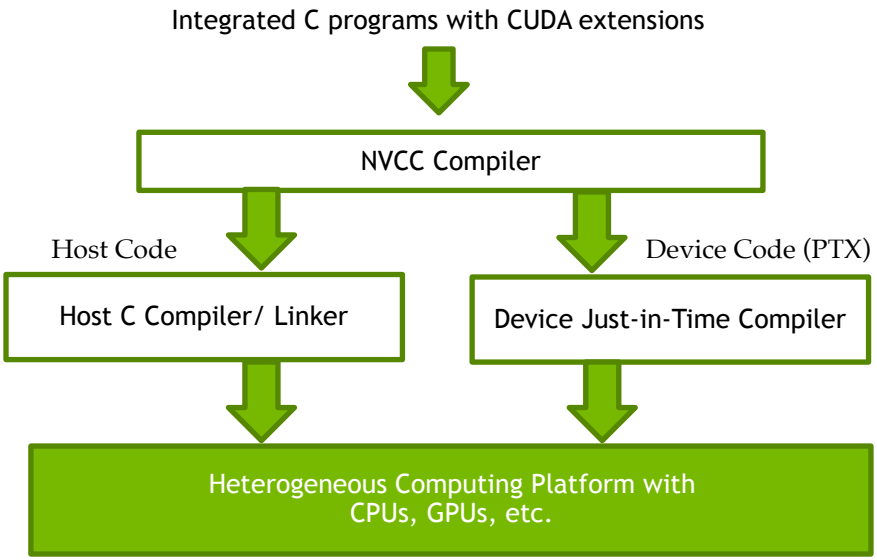


16×16 block



Not all threads in a Block will follow the same control flow path.

Compiling A CUDA Program



CSC 447: Parallel Programming

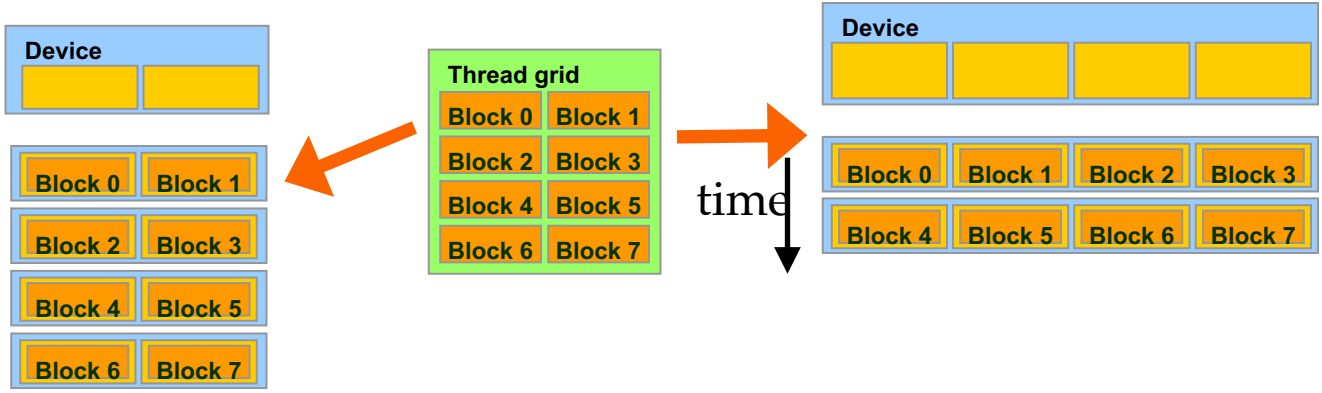
CUDA Parallelism Model

Thread Scheduling

Objective

- To learn how a CUDA kernel utilizes hardware execution resources
  - Assigning thread blocks to execution resources
  - Capacity constrains of execution resources
  - Zero-overhead thread scheduling

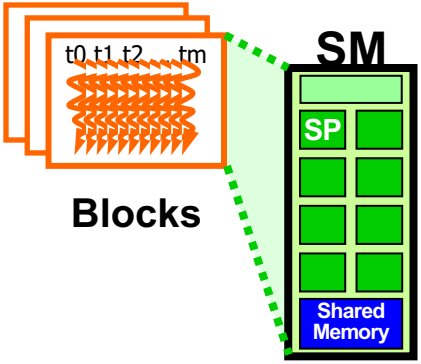
Transparent Scalability



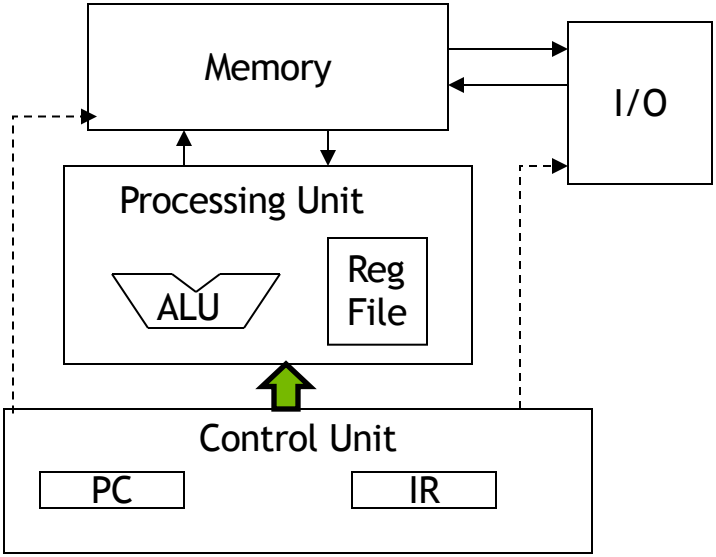
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors

Example: Executing Thread Blocks

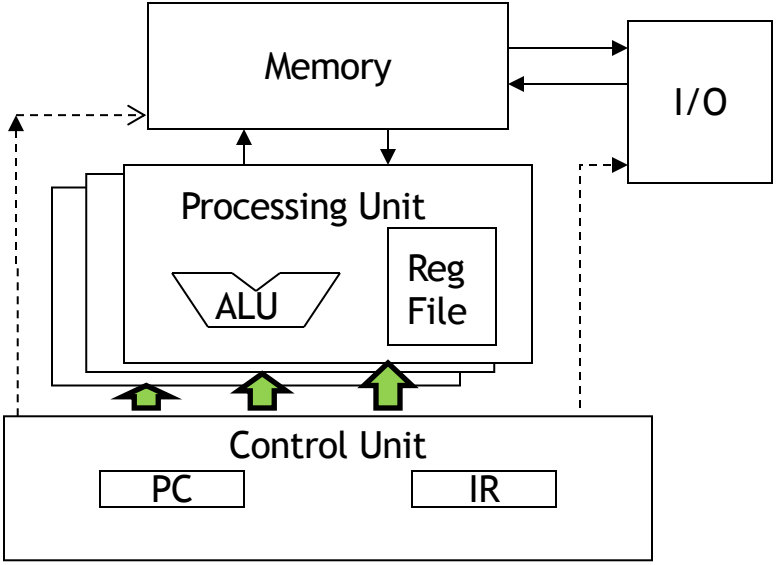
- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Up to **8** blocks to each SM as resource allows
  - Fermi SM can take up to **1536** threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



The Von-Neumann Model



The Von-Neumann Model with SIMD units



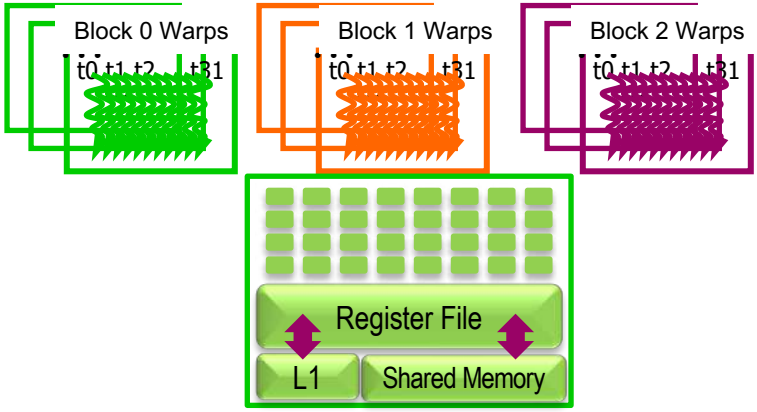
Single Instruction Multiple Data
(SIMD)

Warps as Scheduling Units

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp

Warp Example

- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps



Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution based on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks for Fermi?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, which translates to 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM.

CSC 447: Parallel Programming

Lecture 8– Memory and Data Locality

CUDA Memories

Objective

- To learn to effectively use the CUDA memory types in a parallel program
- Importance of memory access efficiency
- Registers, shared memory, global memory
- Scope and lifetime

Review: Image Blur Kernel.

```
// Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

        int curRow = Row + blurRow;
        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
            pixVal += in[curRow * w + curCol];
            pixels++; // Keep track of number of pixels in the
accumulated total
        }
    }
}

// Write our new pixel value out
out[Row * w + Col] = (unsigned char)(pixVal / pixels);
```

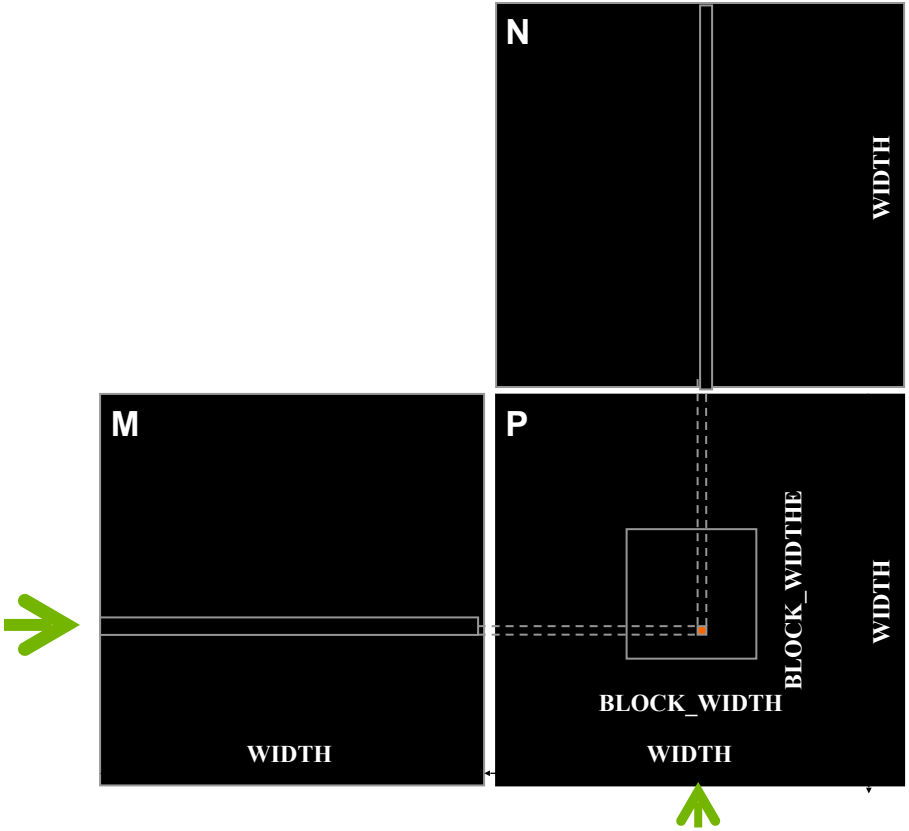
How about performance on a GPU

- All threads access global memory for their input matrix elements
- One memory accesses (4 bytes) per floating-point addition
- 4B/s of memory bandwidth/FLOPS
- Assume a GPU with
- Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth
- 4*1,500 = 6,000 GB/s required to achieve peak FLOPS rating
- The 200 GB/s memory bandwidth limits the execution at 50 GFLOPS

- This limits the execution rate to 3.3% (50/1500) of the peak floating-point execution rate of the device!

- Need to drastically cut down memory accesses to get close to the1,500 GFLOPS

Example – Matrix Multiplication



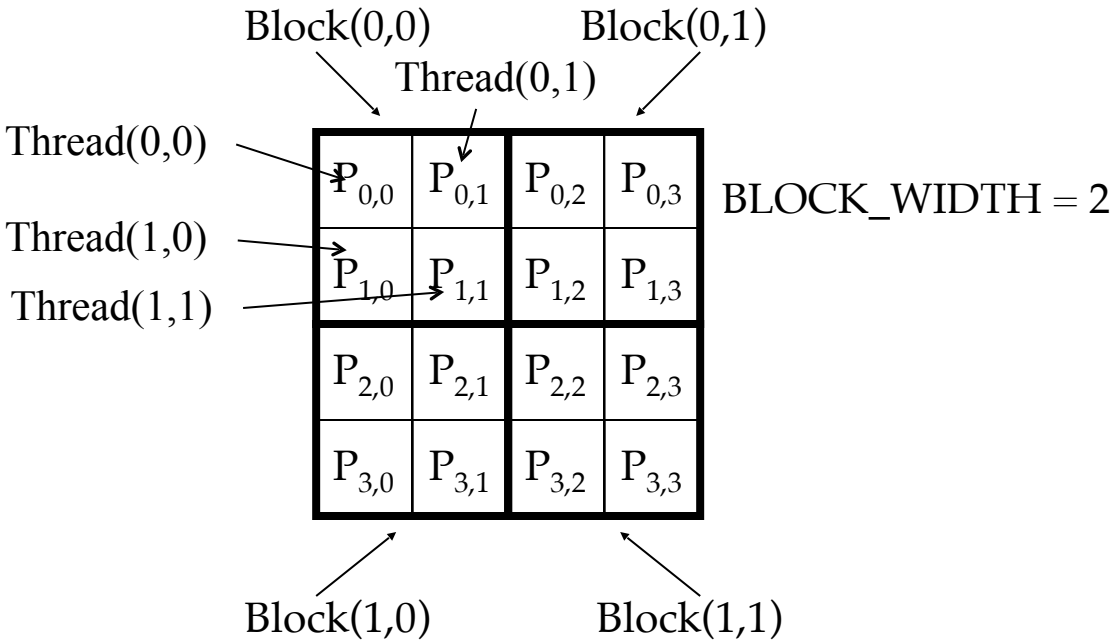
A Basic Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {

  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

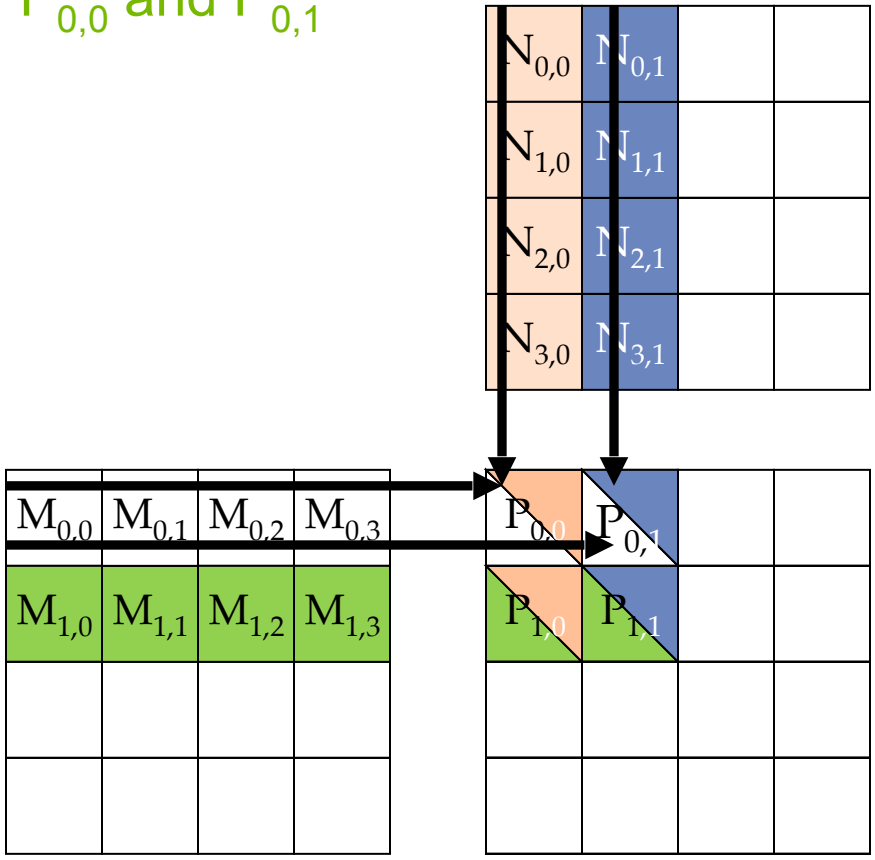
Example – Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {

  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

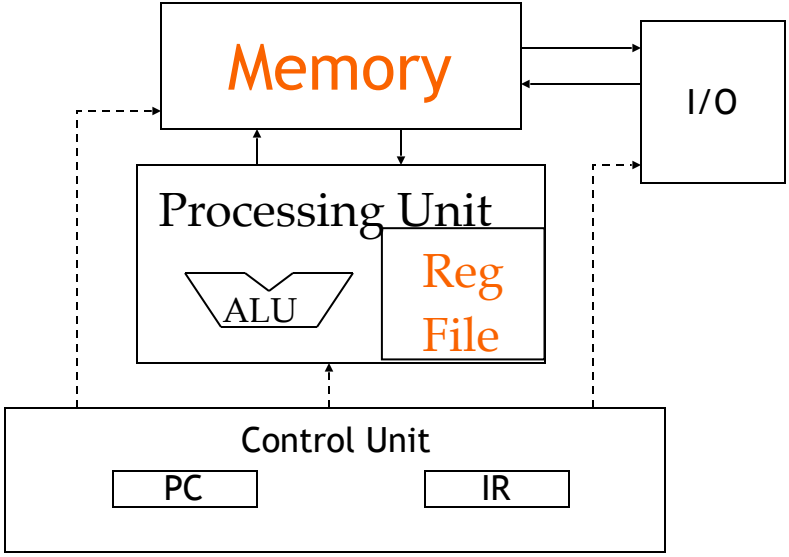
A Toy Example: Thread to P Data Mapping



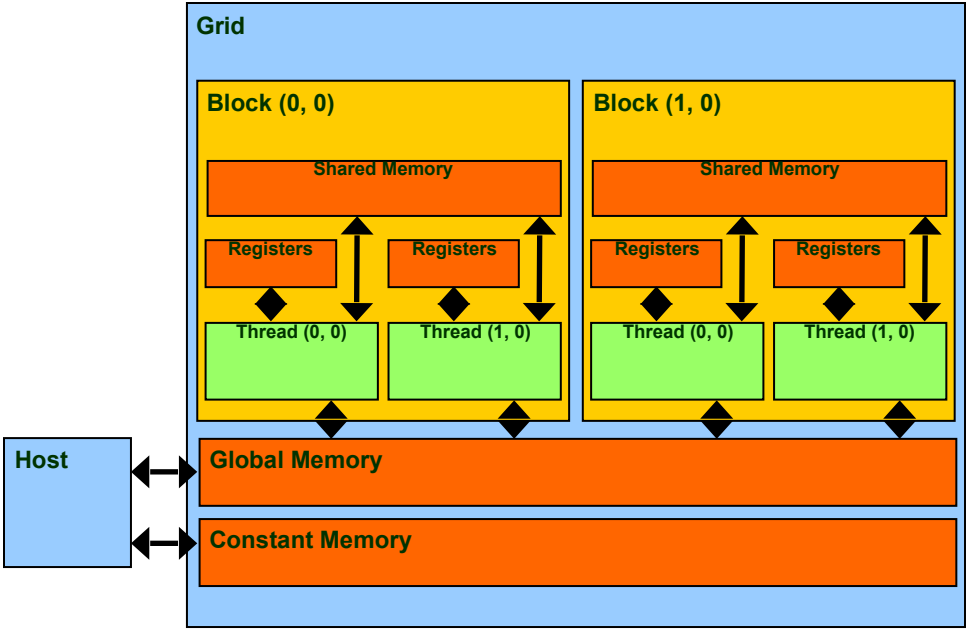
Calculation of $P_{0,0}$ and $P_{0,1}$



Memory and Registers in the Von-Neumann Model



Programmer View of CUDA Memories



Declaring CUDA Variables

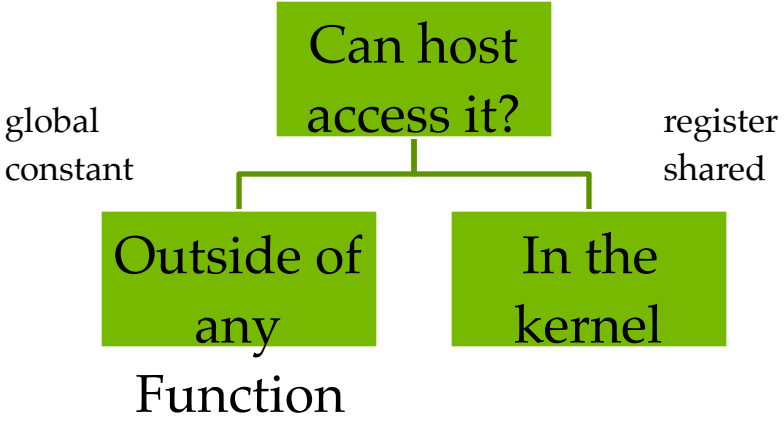
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- **`__device__`** is optional when used with **`__shared__`**, or **`__constant__`**
- Automatic variables reside in a register
- Except per-thread arrays that reside in global memory

Example:
Shared Memory Variable Declaration

```
void blurKernel(unsigned char * in, unsigned char * out,
int w, int h)
{
  __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];

 …
}
```

Where to Declare Variables?



Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
- One in each SM
- Accessed at much higher speed (in both latency and throughput) than global memory
- Scope of access and sharing - thread blocks
- Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
- Accessed by memory load/store instructions
- A form of scratchpad memory in computer architecture

CSC447 Parallel Programming

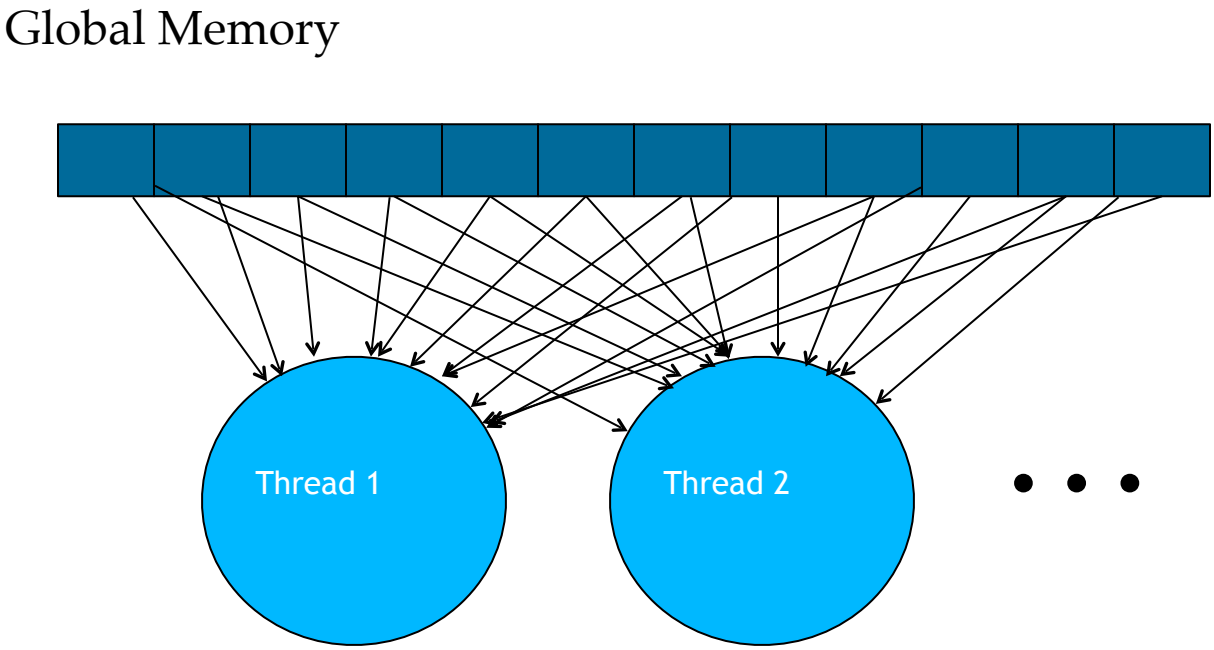
Lecture 9 – Memory and Data Locality

Tiled Parallel Algorithms

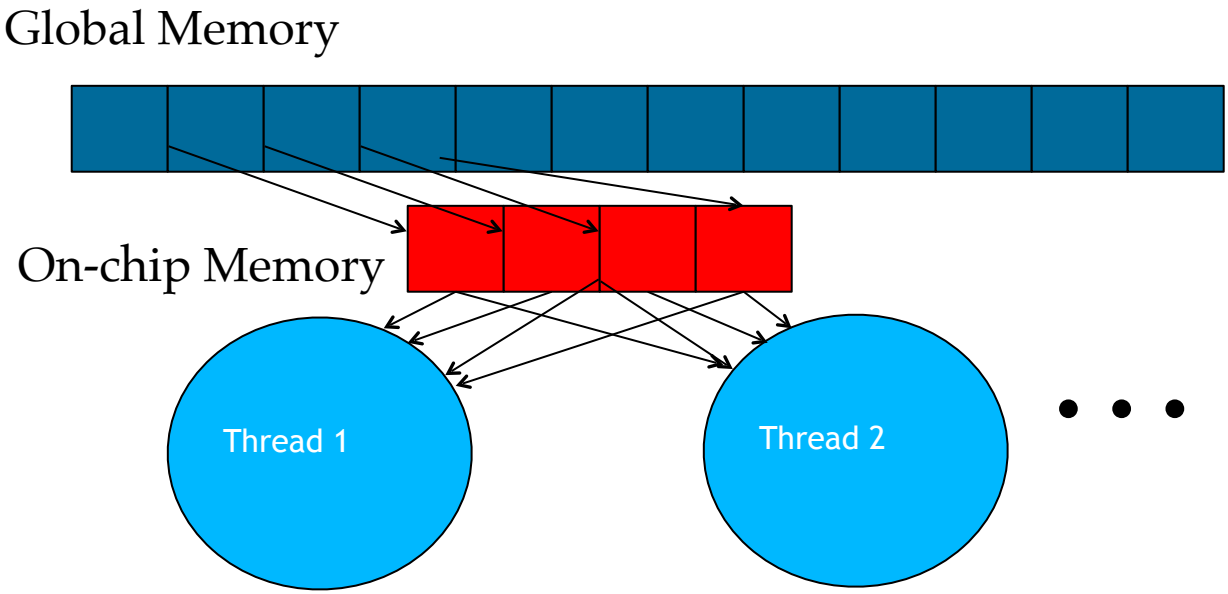
Objective

- To understand the motivation and ideas for tiled parallel algorithms
- Reducing the limiting effect of memory bandwidth on parallel kernel performance
- Tiled algorithms and barrier synchronization

Global Memory Access Pattern of the Basic Matrix Multiplication Kernel



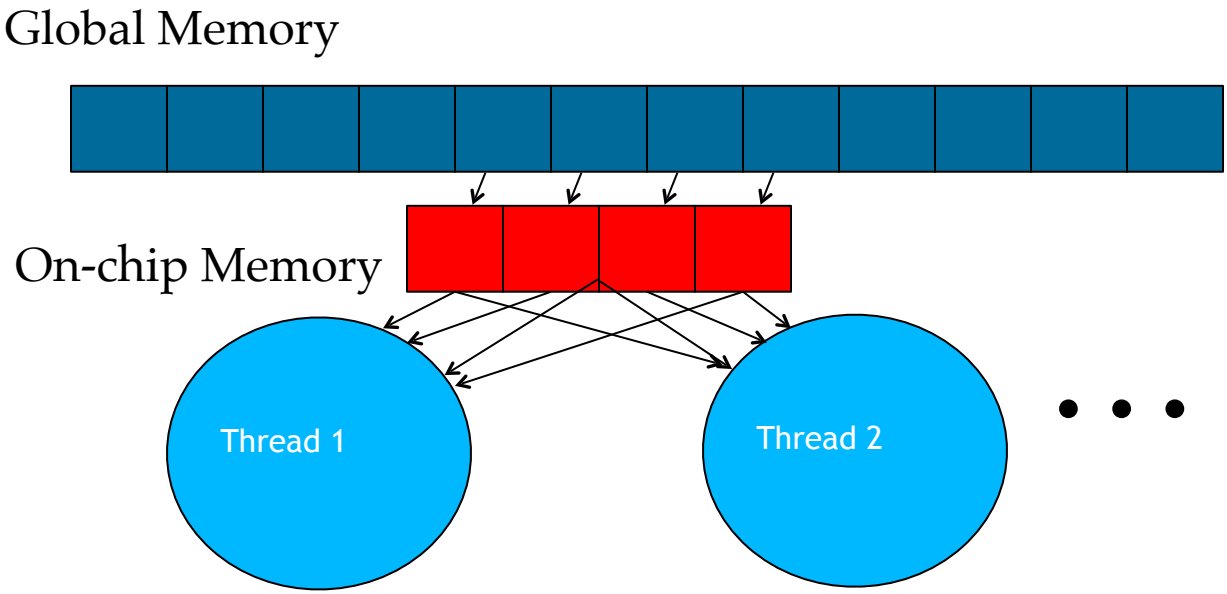
Tiling/Blocking - Basic Idea



Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

Tiling/Blocking - Basic Idea



Basic Concept of Tiling

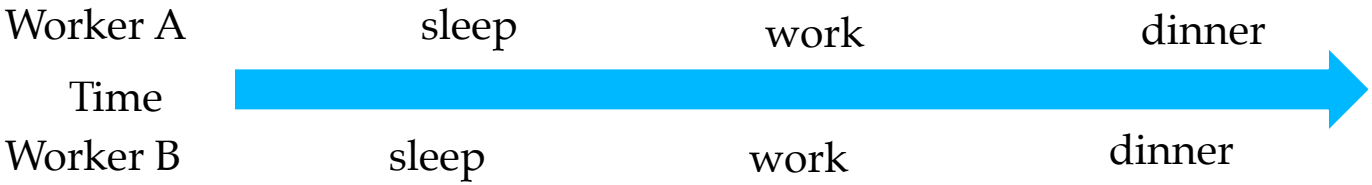
- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
- Carpooling for commuters
- Tiling for global memory accesses
- drivers = threads accessing their memory data operands
- cars = memory access requests

Some Computations are More Challenging to Tile

- Some carpools may be easier than others
- Car pool participants need to have similar work schedule
- Some vehicles may be more suitable for carpooling
- Similar challenges exist in tiling

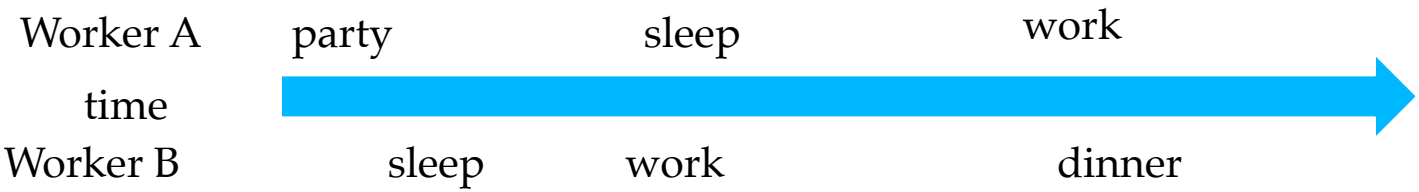
Carpools need synchronization.

– Good: when people have similar schedule



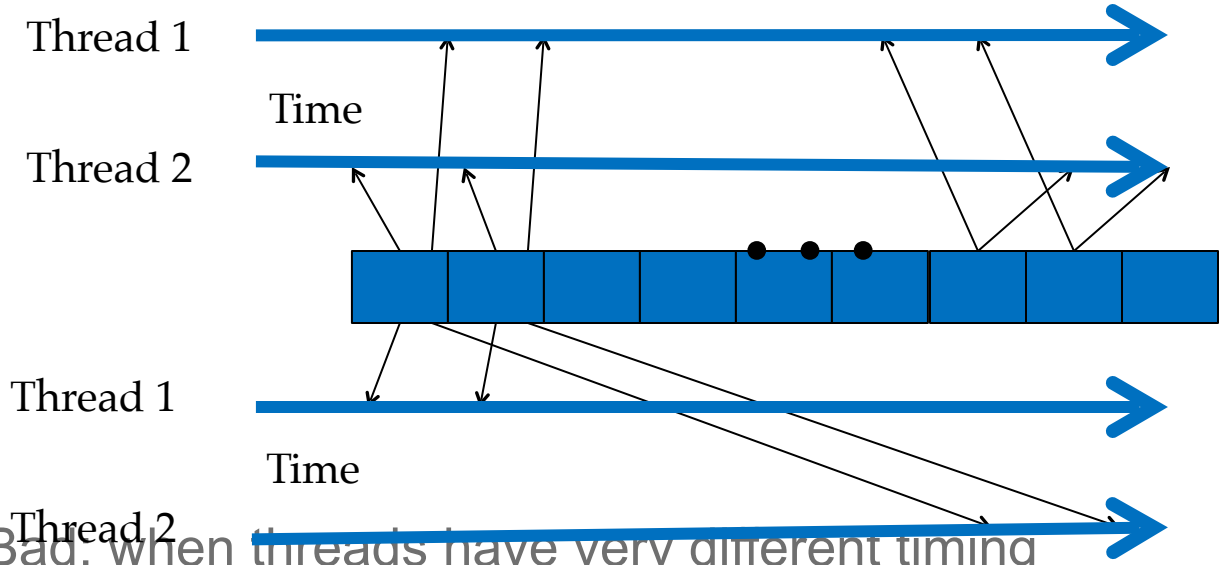
Carpools need synchronization.

– Bad: when people have very different schedule



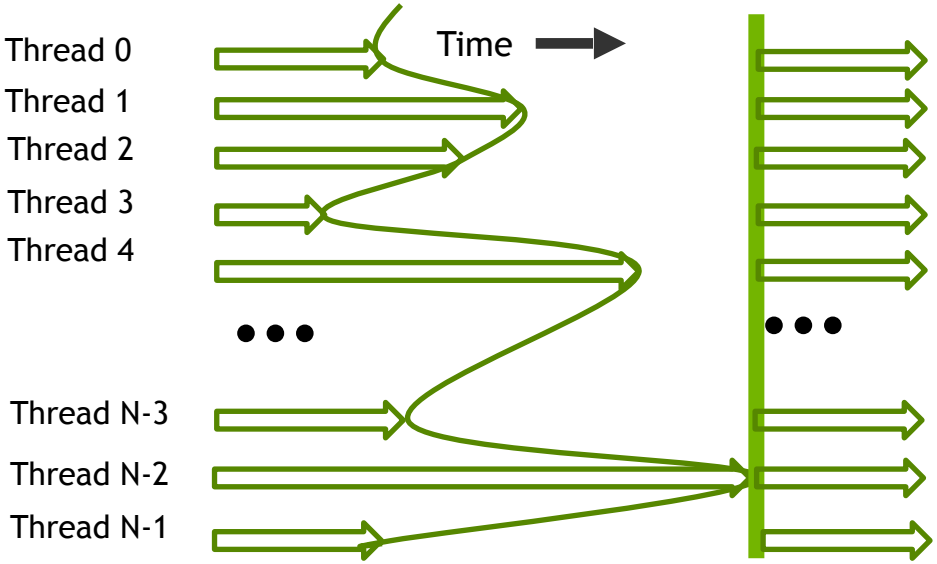
Same with Tiling

– Good: when threads have similar access timing



– Bad: when threads have very different timing

Barrier Synchronization for Tiling



Outline of Tiling Technique

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile

CSC 447 Parallel Programming

Lecture 10 - Memory Model and Locality

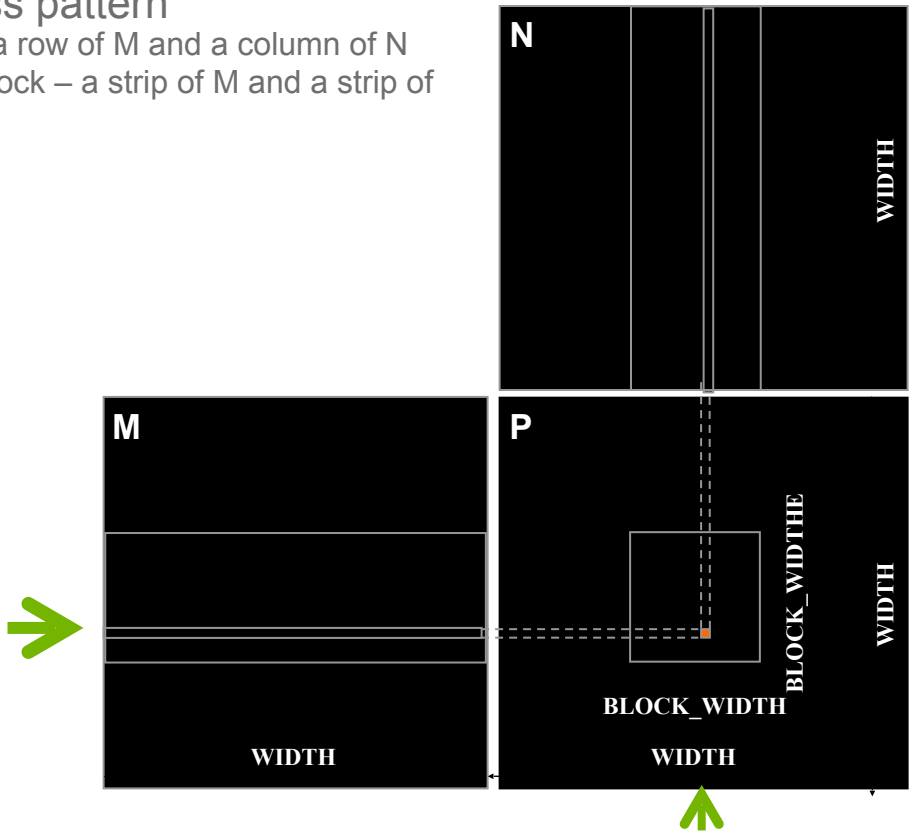
Tiled Matrix Multiplication

Objective

- To understand the design of a tiled parallel algorithm for matrix multiplication
- Loading a tile
- Phased execution
- Barrier Synchronization

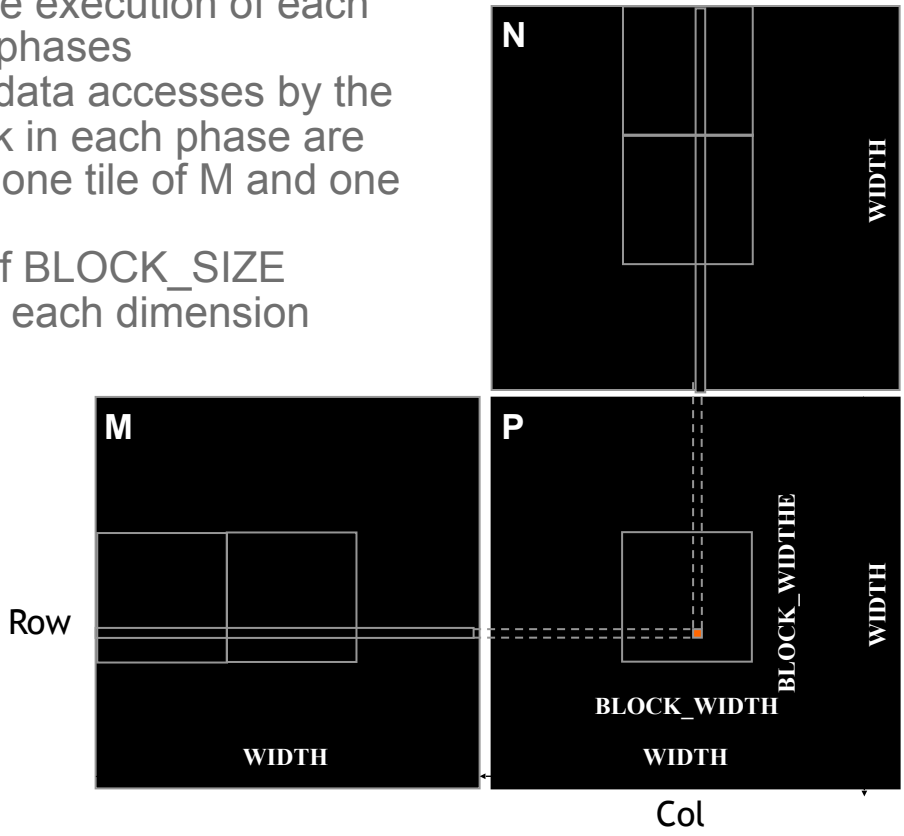
Matrix Multiplication

- Data access pattern
- Each thread - a row of M and a column of N
- Each thread block – a strip of M and a strip of N



Tiled Matrix Multiplication

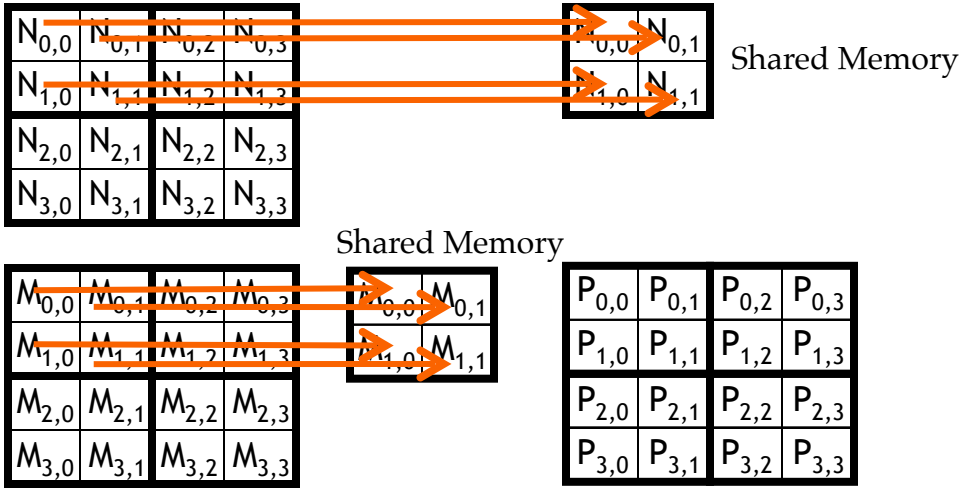
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



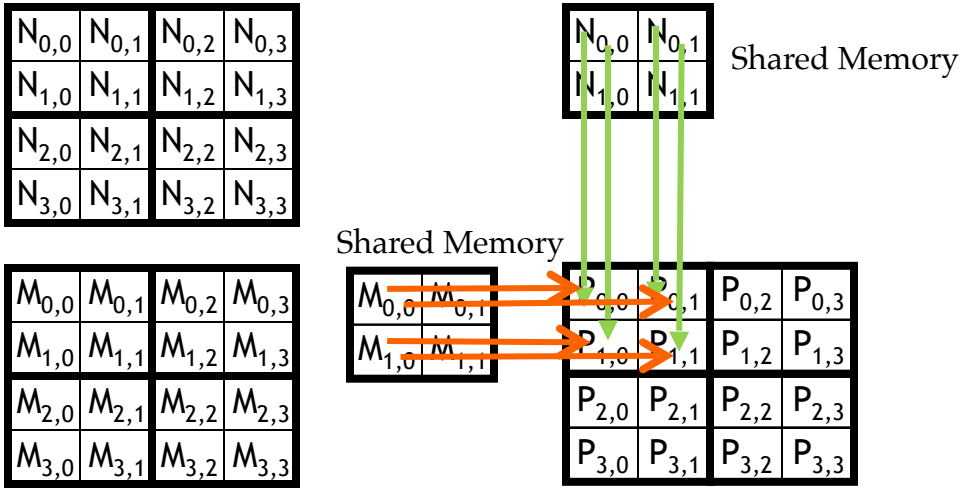
Loading a Tile

- All threads in a block participate
- Each thread loads one M element and one N element in tiled code

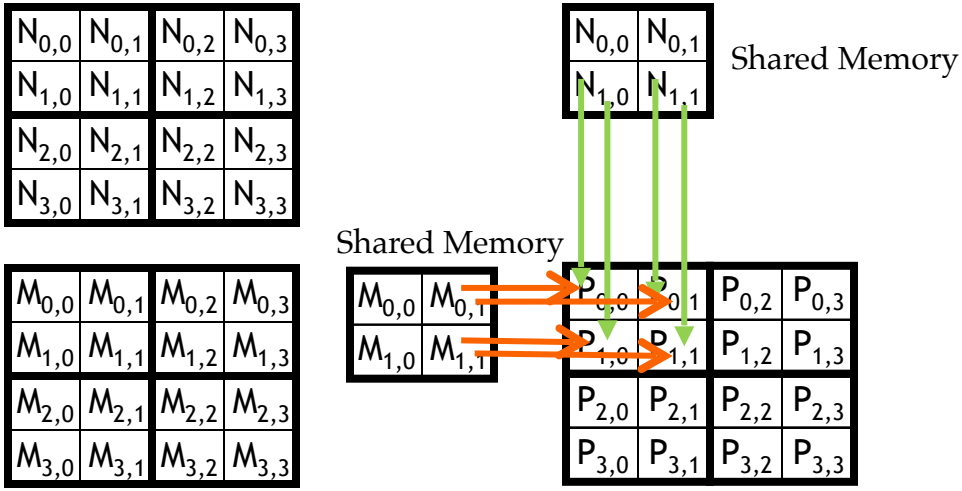
Phase 0 Load for Block (0,0)



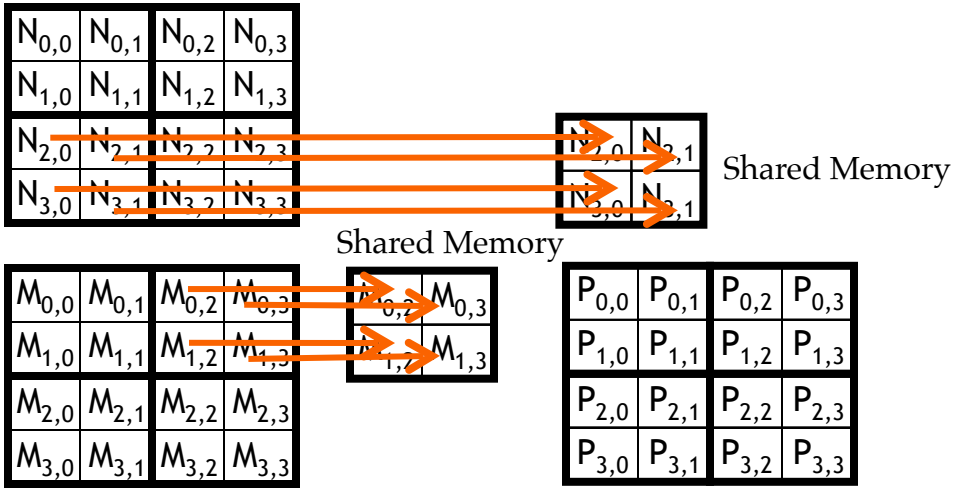
Phase 0 Use for Block (0,0) (iteration 0)



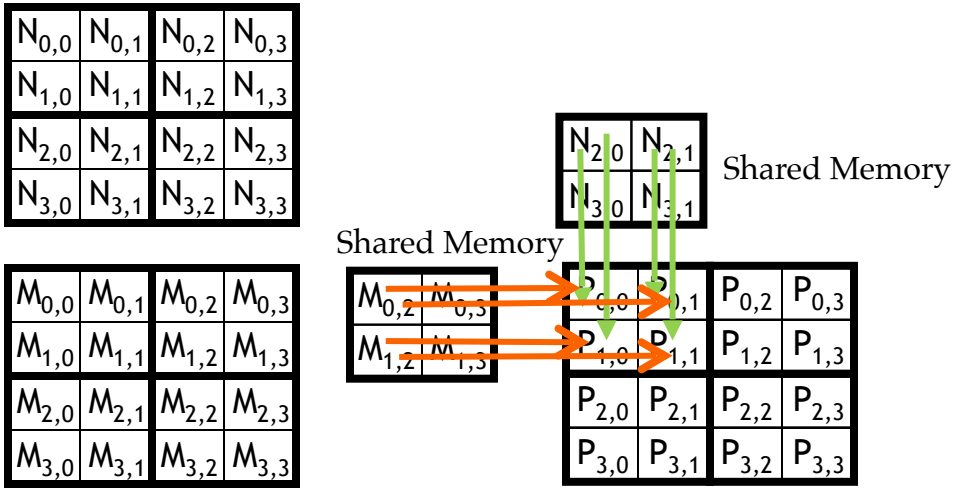
Phase 0 Use for Block (0,0) (iteration 1)



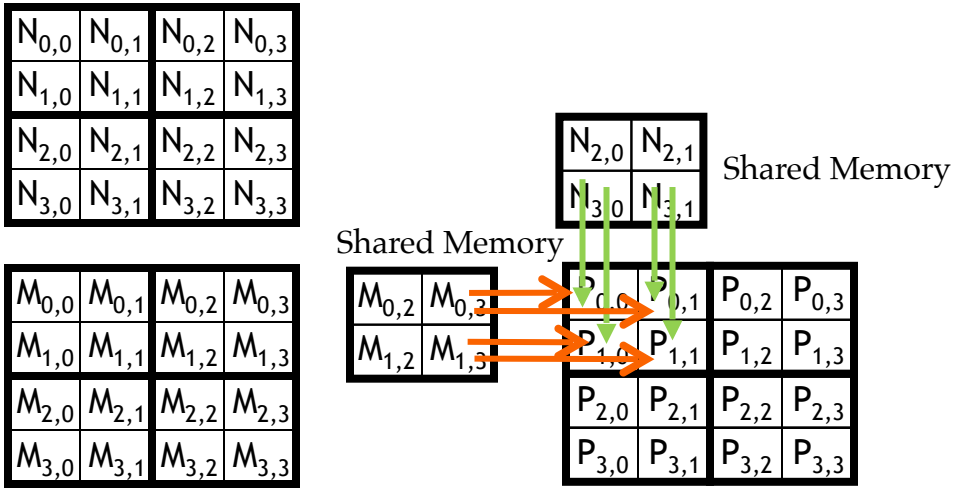
Phase 1 Load for Block (0,0)



Phase 1 Use for Block (0,0) (iteration 0)



Phase 1 Use for Block (0,0) (iteration 1)



Execution Phases of Toy Example

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time →

Execution Phases of Toy Example (cont.)

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{1,0}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time →

Shared memory allows each value to be accessed by multiple threads

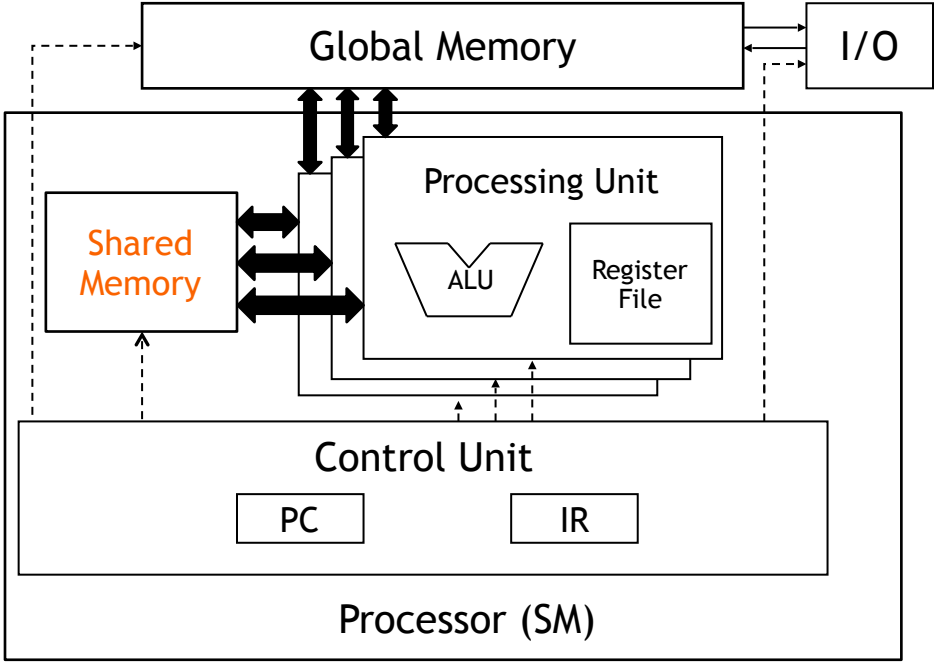
Barrier Synchronization

- Synchronize all threads in a block
- __syncthreads()

- All threads in the same block must reach the __syncthreads() before any of the them can move on

- Best used to coordinate the phased execution tiled algorithms
- To ensure that all elements of a tile are loaded at the beginning of a phase
- To ensure that all elements of a tile are consumed at the end of a phase

Hardware View of CUDA Memories



CSC 447 Parallel Programming

Lecture 11- Memory and Data Locality

Tiled Matrix Multiplication Kernel

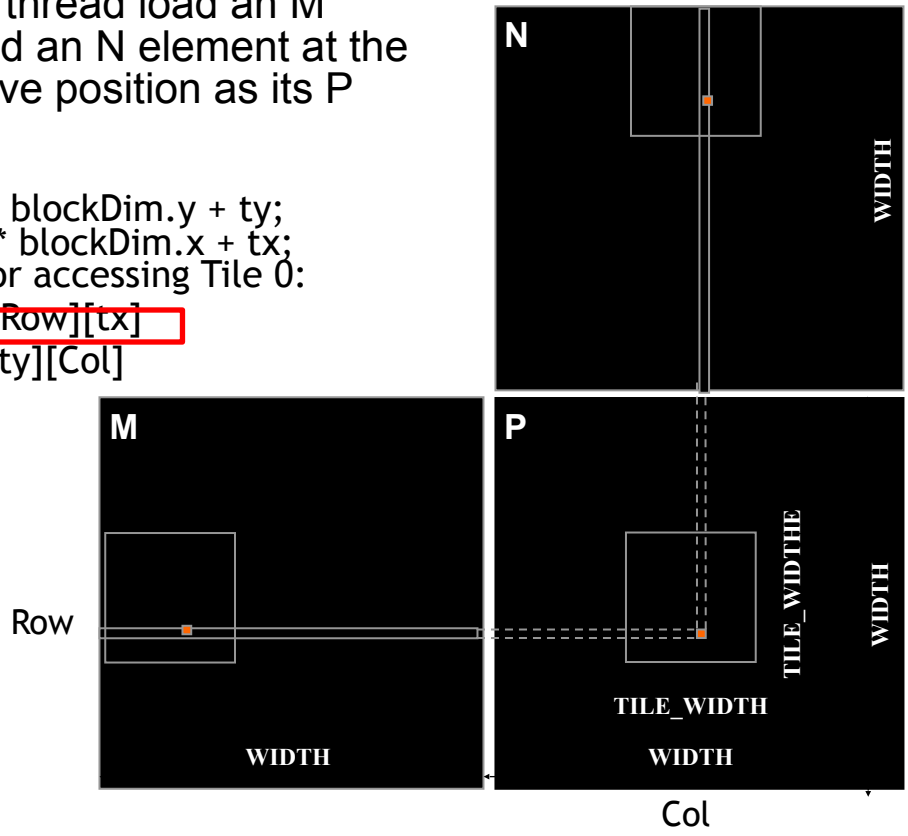
Objective

- To learn to write a tiled matrix-multiplication kernel
- Loading and using tiles for matrix multiplication
- Barrier synchronization, shared memory
- Resource Considerations
- Assume that Width is a multiple of tile size for simplicity

Loading Input Tile 0 of M (Phase 0)

– Have each thread load an M element and an N element at the same relative position as its P element.

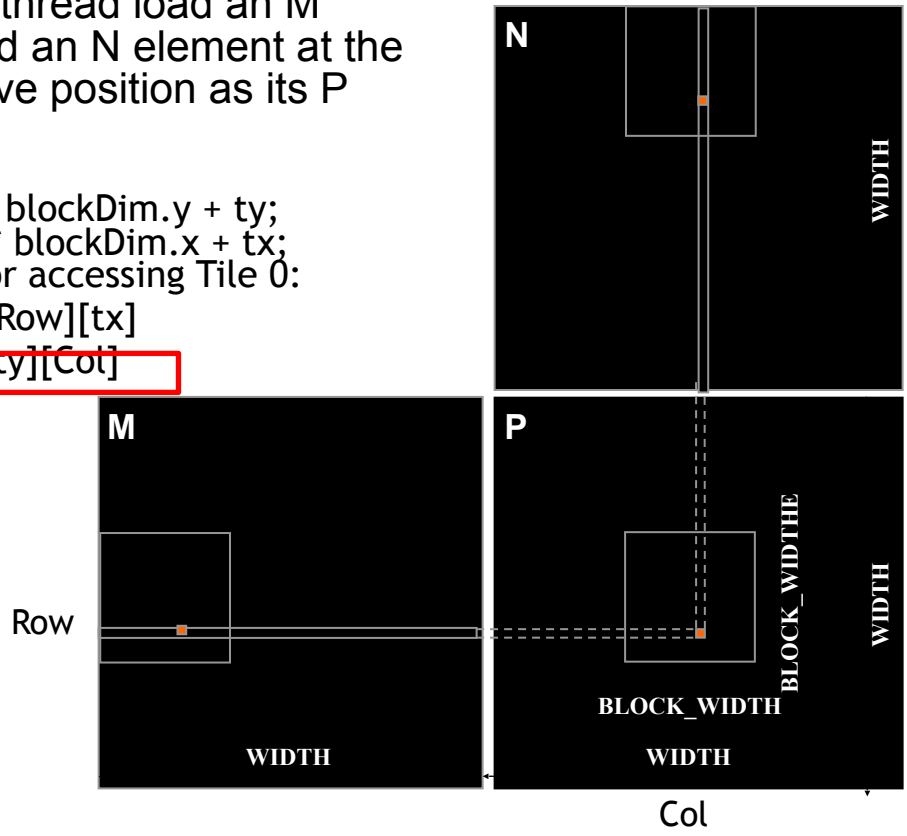
```
int Row = by * blockDim.y + ty;
int Col =   bx * blockDim.x + tx;
2D indexing for accessing Tile 0:
        M[Row][tx]
        N[ty][Col]
```



Loading Input Tile 0 of N (Phase 0)

– Have each thread load an M element and an N element at the same relative position as its P element.

```
int Row = by * blockDim.y + ty;
int Col =   bx * blockDim.x + tx;
2D indexing for accessing Tile 0:
            M[Row][tx]
            N[ty][Col]
```



Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

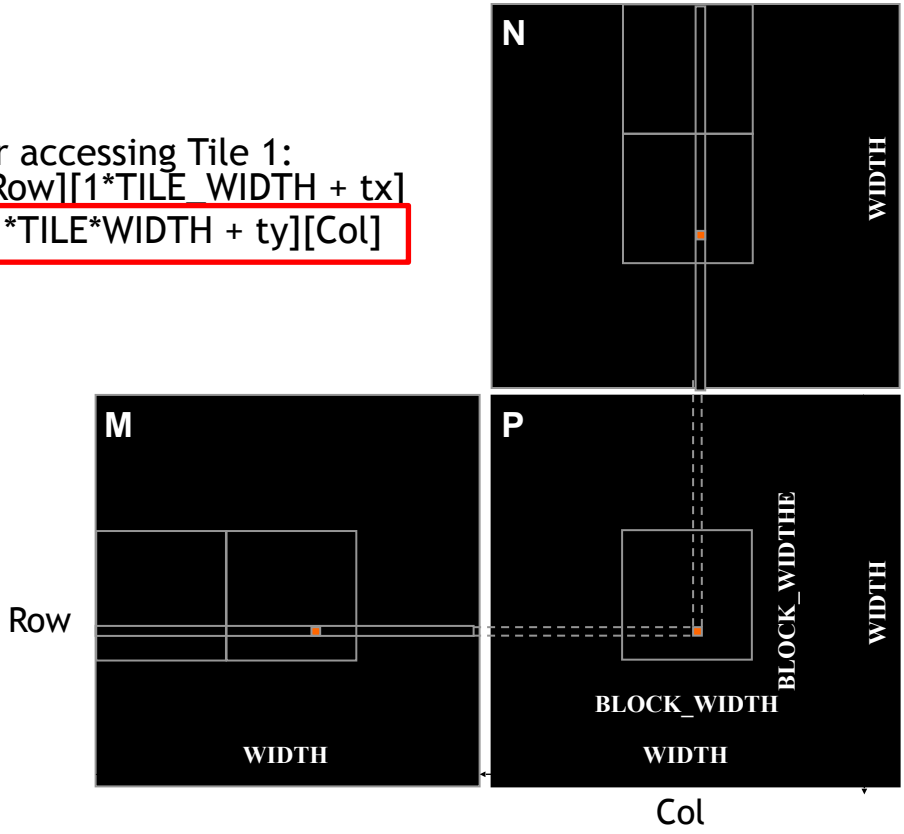
```
M[Row][1*TILE_WIDTH + tx]
N[1*TILE*WIDTH + ty][Col]
```

Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

M[Row][1*TILE_WIDTH + tx]

N[1*TILE*WIDTH + ty][Col]



M and N are dynamically allocated - use 1D indexing

```
M[Row][p*TILE_WIDTH+tx]
M[Row*Width + p*TILE_WIDTH + tx]

N[p*TILE_WIDTH+ty][Col]
N[(p*TILE_WIDTH+ty)*Width + Col]
```

where p is the sequence number of the current phase

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
    __synchthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < n/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Tile (Thread Block) Size Considerations

- Each thread block should have many threads
- TILE_WIDTH of 16 gives 16*16 = 256 threads
- TILE_WIDTH of 32 gives 32*32 = 1024 threads

- For 16, in each phase, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations. (16 floating-point operations for each memory load)

- For 32, in each phase, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations. (32 floating-point operation for each memory load)

Shared Memory and Threading

- For an SM with 16KB shared memory
- Shared memory size is implementation dependent!
- For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
- For 16KB shared memory, one can potentially have up to 8 thread blocks executing
- This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
- The next TILE_WIDTH 32 would lead to 2*32*32*4 Byte= 8K Byte shared memory usage per thread block, allowing 2 thread blocks active at the same time
- However, in a GPU where the thread count is limited to 1536 threads per SM, the number of blocks per SM is reduced to one!
- Each __syncthread() can reduce the number of active threads for a block
- More thread blocks can be advantageous

CSC 447 Parallel Programming

Lecture 12- Memory and Data Locality

Handling Arbitrary Matrix Sizes in Tiled Algorithms

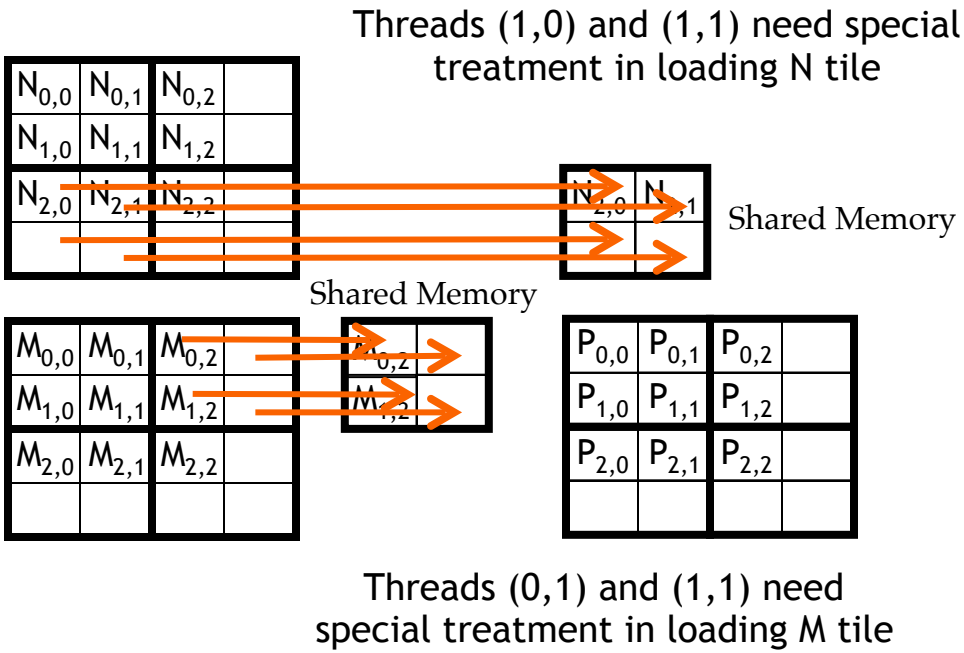
Objective

- To learn to handle arbitrary matrix sizes in tiled matrix multiplication
- Boundary condition checking
- Regularizing tile contents
- Rectangular matrices

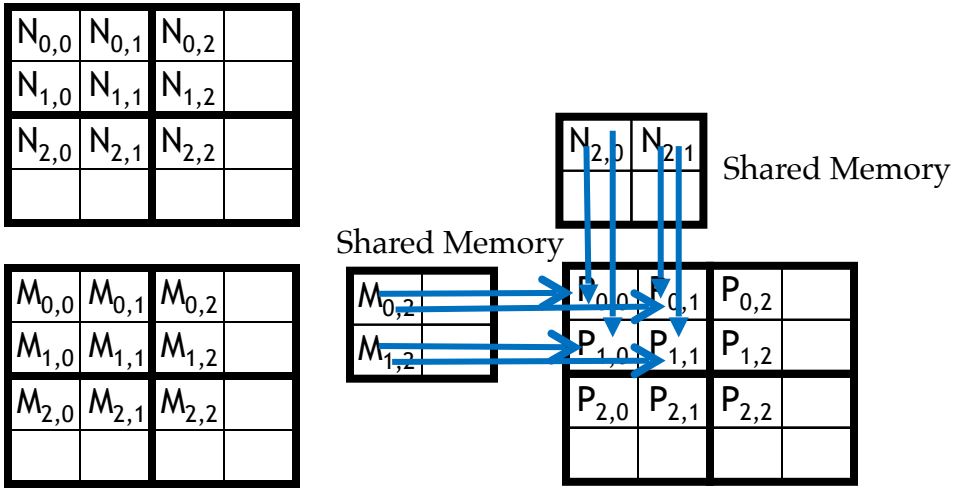
Handling Matrix of Arbitrary Size

- The tiled matrix multiplication kernel we presented so far can handle only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)
- However, real applications need to handle arbitrary sized matrices.
- One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.
- We will take a different approach.

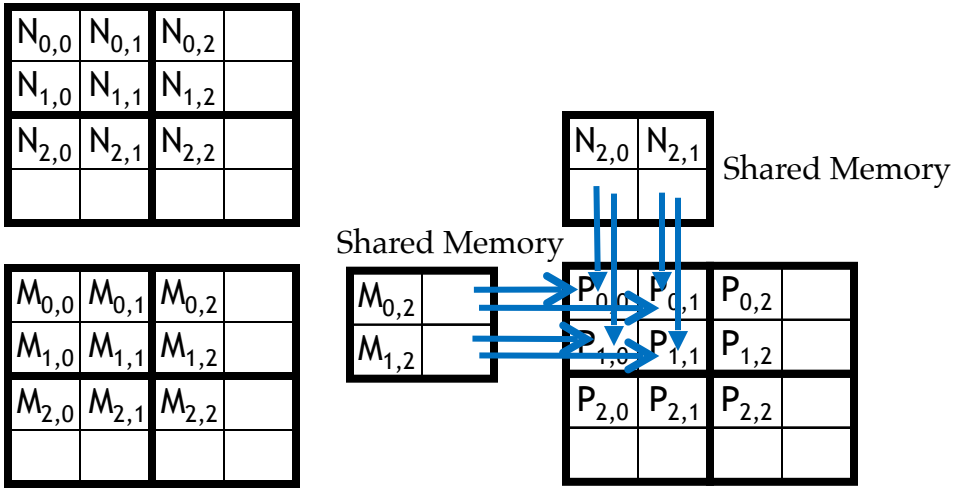
Phase 1 Loads for Block (0,0) for a 3x3 Example



Phase 1 Use for Block (0,0) (iteration 0)



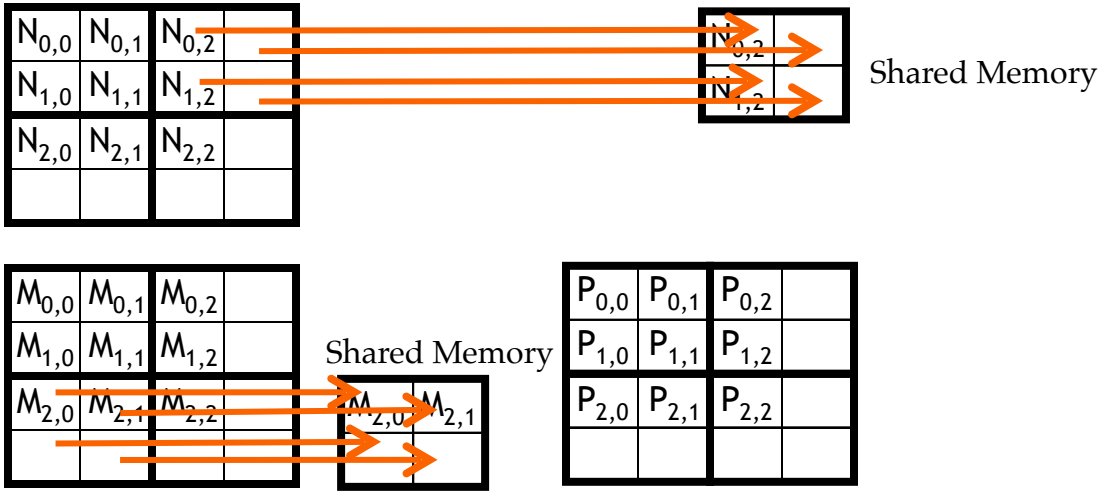
Phase 1 Use for Block (0,0) (iteration 1)



All Threads need special treatment. None of them should introduce invalidate contributions to their P elements.

Phase 0 Loads for Block (1,1) for a 3x3 Example

Threads (0,1) and (1,1) need special treatment in loading N tile



Threads (1,0) and (1,1) need special treatment in loading M tile

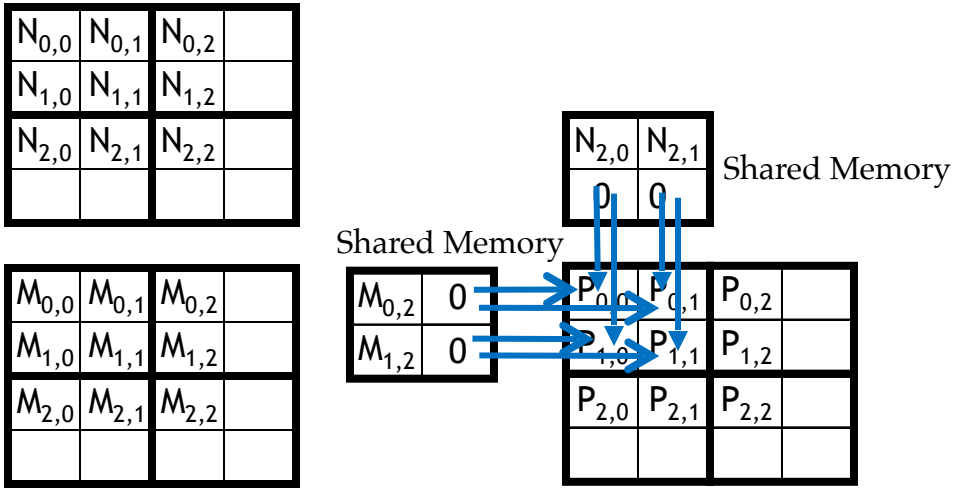
Major Cases in Toy Example

- Threads that do not calculate valid P elements but still need to participate in loading the input tiles
  - Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent P[3,2] but need to participate in loading tile element N[1,2]
- Threads that calculate valid P elements may attempt to load non-existing input elements when loading input tiles
  - Phase 0 of Block(0,0), Thread(1,0), assigned to calculate valid P[1,0] but attempts to load non-existing N[3,0]

A “Simple” Solution

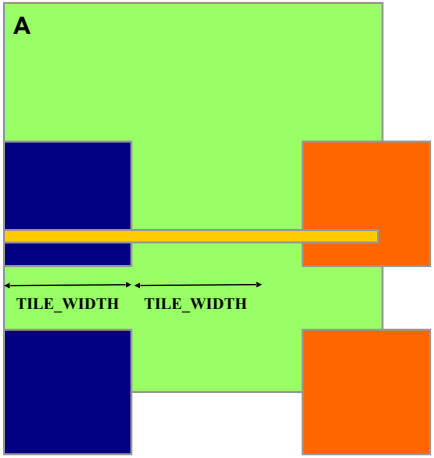
- When a thread is to load any input element, test if it is in the valid index range
  - If valid, proceed to load
  - Else, do not load, just write a 0
- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element
- The condition tested for loading input elements is different from the test for calculating output P element
  - A thread that does not calculate valid P element can still participate in loading input tile elements

Phase 1 Use for Block (0,0) (iteration 1)



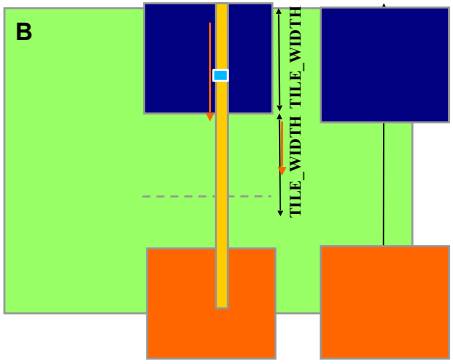
Boundary Condition for Input M Tile

- Each thread loads
- M[Row][p*TILE_WIDTH+tx]
- M[Row*Width + p*TILE_WIDTH+tx]
- Need to test
- (Row < Width) && (p*TILE_WIDTH+tx < Width)
- If true, load M element
- Else , load 0



Boundary Condition for Input N Tile

- Each thread loads
- N[p*TILE_WIDTH+ty][Col]
- N[(p*TILE_WIDTH+ty)*Width+ Col]
- Need to test
- (p*TILE_WIDTH+ty < Width) && (Col< Width)
- If true, load N element
- Else , load 0



Loading Elements – with boundary check

```
–   8   for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {
–
–   ++      if(Row < Width && t * TILE_WIDTH+tx < Width) {
–   9           ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
–   ++      } else {
–   ++          ds_M[ty][tx] = 0.0;
–   ++      }
–   ++      if (p*TILE_WIDTH+ty < Width && Col < Width) {
–   10          ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
–   ++      } else {
–   ++          ds_N[ty][tx] = 0.0;
–   ++      }
–   11     __syncthreads();
–
```

Inner Product – Before and After

```
–   ++   if(Row < Width && Col < Width) {
–   12     for (int i = 0; i < TILE_WIDTH; ++i) {
–   13             Pvalue += ds_M[ty][i] * ds_N[i][tx];
–          }
–   14     __syncthreads();
–   15   } /* end of outer for loop */
–   ++   if (Row < Width && Col < Width)
–   16         P[Row*Width + Col] = Pvalue;
–     } /* end of kernel */
```

Some Important Points

- For each thread the conditions are different for
- Loading M element
- Loading N element
- Calculating and storing output elements
- The effect of control divergence should be small for large matrices