COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 38: Course Review (Second half of semester)

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Announcements

- Homework 7 due by 5pm today
  - Send email to comp322-staff if you're running into issues with accessing SUG@R nodes, or are delayed for any other reason
- Take-home final exam will be given at the end of today's lecture
  - Content will focus on second half of semester
    - Knowledge of supporting content from first half of semester will be assumed e.g., async, finish, isolated, forall, critical-path-length and work metrics
    - This week's lectures on MPI will not be included in the exam
  - Due by 5pm on Friday, April 29th

Table 1: Methods in java.util.concurrent atomic classes AtomicInteger and AtomicIntegerArray (Lecture 19)

`j.u.c.atomic` Class and Constructors	`j.u.c.atomic` Methods	Equivalent HJ `isolated` statements
AtomicInteger	int j = v.**get**();	int j; `isolated` j = v.val;
	v.**set**(newVal);	`isolated` v.val = newVal;
AtomicInteger() // init = 0	int j = v.**getAndSet**(newVal);	int j; `isolated` { j = v.val; v.val = newVal; }
	int j = v.**addAndGet**(delta);	`isolated` { v.val += delta; j = v.val; }
	int j = v.**getAndAdd**(delta);	`isolated` { j = v.val; v.val += delta; }
AtomicInteger(init)	boolean b = v.**compareAndSet** (expect,update);	boolean b; `isolated` if (v.val==expect) {v.val=update; b=true;} else b = false;
AtomicIntegerArray	int j = v.**get**(i);	int j; `isolated` j = v.arr[i];
	v.**set**(i,newVal);	`isolated` v.arr[i] = newVal;
AtomicIntegerArray (length) // init = 0	int j = v.**getAndSet**(i,newVal);	int j; `isolated` { j = v.arr[i]; v.arr[i] = newVal; }
	int j = v.**addAndGet**(i,delta);	`isolated` { v.arr[i] += delta; j = v.arr[i]; }
	int j = v.**getAndAdd**(i,delta);	`isolated` { j = v.arr[i]; v.arr[i] += delta; }
AtomicIntegerArray (arr)	boolean b = v.**compareAndSet** (i,expect,update);	boolean b; `isolated` if (v.arr[i]==expect) {v.arr[i]=update; b=true;} else b = false;

Parallel Depth-First Search Spanning Tree Example w/ isolated construct

```
class V  {
  V [] neighbors; // adjacency list for input graph
  V parent;     // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent=n;
    return parent == n;
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish root.compute();
. . .
```

Parallel Depth-First Search Spanning Tree Example w/ AtomicReference

```
class V  {
  V [] neighbors; // adjacency list for input graph
  AtomicReference parent; // output value of parent in spanning tree
  boolean tryLabeling(V n) {
    return parent.compareAndSet(null ,n);

  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
. . .
root.parent = root; // Use self-cycle to identify root
finish root.compute();
. . .
```


java.util.concurrent.concurrentHashMap (Lecture 20)

- Implements ConcurrentMap sub-interface of Map
- Allows read (traversal) and write (update) operations to overlap with each other
- Some operations are atomic with respect to each other e.g.,
  - get(), put(), putIfAbsent(), remove()
- Aggregate operations may not be viewed atomically by other operations e.g.,
  - putAll(), clear()
- Expected degree of parallelism can be specified in ConcurrentHashMap constructor
  - ConcurrentHashMap(initialCapacity, loadFactor, concurrencyLevel)
  - A larger value of concurrencyLevel results in less serialization, but a larger space overhead for storing the ConcurrentHashMap

Example usage of ConcurrentHashMap in org.mirrorfinder.model.BaseDirectory

```java
public abstract class BaseDirectory extends BaseItem implements Directory {
  Map files = new ConcurrentHashMap();
  . . .
  public Map getFiles() {
    return files;
  }
  public boolean has(File item) {
    return getFiles().containsValue(item);
  }
  public Directory add(File file) {
    String key = file.getName();
    if (key == null) throw new Error(. . .);
    getFiles().put(key, file);
    . . .
    return this;
  }
  public Directory remove(File item) throws NotFoundException {
    if (has(item)) {
      getFiles().remove(item.getName());
      . . .
    } else throw new NotFoundException("can't remove unrelated item");
  }
}
```

Listing 1: Example usage of ConcurrentHashMap in org.mirrorfinder.model.BaseDirectory [1]

java.util.concurrent.ConcurrentLinkedQueue (Lecture 20)

- `Queue` interface added to `java.util`
  - interface Queue extends Collection and includes

    boolean offer(E x); // same as add() in Collection
    E poll(); // remove head of queue if non-empty
    E remove(o) throws NoSuchElementException;
    E peek(); // examine head of queue without removing it
- Non-blocking operations
  - Return `false` when full
  - Return `null` when empty
- Fast thread-safe non-blocking implementation of Queue interface: `ConcurrentLinkedQueue`

Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

```java
class BufferPool15Impl implements BufferPool.BufferPoolAPI {
  protected int maxSize;
  protected AtomicInteger size = new AtomicInteger(0);
  protected ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
  . . .
  public XByteBuffer getBuffer(int minSize, boolean discard) {
    XByteBuffer buffer = (XByteBuffer) queue.poll();
    if ( buffer != null ) size.addAndGet(-buffer.getCapacity());
    if ( buffer == null ) buffer = new XByteBuffer(minSize,discard);
    else if ( buffer.getCapacity() <= minSize ) buffer.expand(minSize);
    . . .
    return buffer;
  }
  public void returnBuffer(XByteBuffer buffer) {
    if ( (size.get() + buffer.getCapacity()) <= maxSize ) {
      size.addAndGet(buffer.getCapacity());
      queue.offer(buffer);
    }
  }
}
```

Listing 2: Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

Informal definition of Linearizability (Lecture 21)

1. A *linearizable execution* is one in which the semantics of a set of method calls performed in parallel on a concurrent object is equivalent to that of some legal linear sequence of those method calls.

2. A *linearizable concurrent object* is one for which all possible executions are linearizable.

Table 1: Example execution of a monitor-based implementation of FIFO queue q

Is this a linearizable execution?

Time	Task *A*	Task *B*
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	
2	Work on `q.enq(x)`	
3	Return from `q.enq(x)`	
4		Invoke `q.enq(y)`
5		Work on `q.enq(y)`
6		Work on `q.enq(y)`
7		Return from `q.enq(y)`
8		Invoke `q.deq()`
9		Return `x` from `q.deq()`

Yes! Equivalent to "q.enq(x) ; q.enq(y) ; q.deq():x"

Table 3: Example of a non-linearizable execution on a concurrent FIFO queue q

Is this a linearizable execution?

Time	Task *A*	Task *B*
0	Invoke `q.enq(x)`	
1	Return from `q.enq(x)`	
2		Invoke `q.enq(y)`
3	Invoke `q.deq()`	Work on `q.enq(y)`
4	Work on `q.deq()`	Return from `q.enq(y)`
5	Return `y` from `q.deq()`	

- **No! q.enq(x) must precede q.enq(y) in all linear sequences of method calls invoked on q. It is illegal for the q.deq() operation to return y.**

Places in HJ (Lecture 22)

here = place at which current task is executing

place.MAX_PLACES = total number of places (runtime constant)

Specified by value of p in runtime option, `-places p:w`

place.factory.place(i) = place corresponding to index i

<place-expr>.toString() returns a string of the form "place(id=0)"

<place-expr>.id returns the id of the place as an int

async at(P) S

- Creates new task to execute statement S at place P
- **async S** is equivalent to **async at(here) S**

Note that **here** in a child task for an async/future computation will refer to the place P at which the child task is executing, not the place where the parent task is executing

Listing 1: Example HJ program with places

```
class T1 {
  final place affinity;
  . . .
  // T1's constructor sets affinity to place where instance was created
  T1() { affinity = here; ... }
  . . .
}
. . .
finish { // Inter-place parallelism
  System.out.println("Parent place = ", here); // Parent task s place
  for (T1 a = . . .) {
    async at (a.affinity) { // Execute async at place with affinity to a
      a.foo();
      System.out.println("Child place = ", here); // Child task's place
    } // async
  } // for
} // finish
. . .
```

Distributions (Lecture 23)

- A distribution maps points in a rectangular index space (region) to places e.g.,
  - i → place.factory.place(i % place.MAX_PLACES-1)
- Programmers are free to create any data structure they choose to store and compute these mappings
- For convenience, the HJ language provides a predefined type, hj.lang.dist, to simplify working with distributions
- Some public members available in an instance d of hj.lang.dist are as follows
  - d.rank = number of dimensions in the input region for distribution d
  - d.get(p) = place for point p mapped by distribution d. It is an error to call d.get(p) if p.rank != d.rank.
  - d.places() = set of places in the range of distribution d
  - d.restrictToRegion(pl) = region of points mapped to place pl by distribution d

Block Distribution

- dist.factory.block([lo:hi]) creates a *block distribution* over the one-dimensional region, lo:hi.
- A block distribution splits the region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible.
- Block distributions can improve the performance of parallel loops that exhibit spatial locality across contiguous iterations.
- Example in Table 1: dist.factory.block([0:15]) for 4 places

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0				1				2				3			

Cyclic Distribution

- dist.factory.cyclic([lo:hi]) creates a cyclic distribution over the one-dimensional region, lo:hi.
- A cyclic distribution "cycles" through places 0 … place.MAX PLACES – 1 when spanning the input region
- Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance
- Example in Table 3: dist.factory.cyclic([0:15]) for 4 places

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

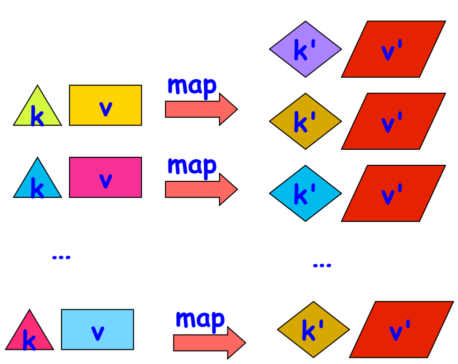
- Example in Table 4: dist.factory.cyclic([0:7,0:1]) for 4 places

Index	[0,0]	[0,1]	[1,0]	[1,1]	[2,0]	[2,1]	[3,0]	[3,1]	[4,0]	[4,1]	[5,0]	[5,1]	[6,0]	[6,1]	[7,0]	[7,1]
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

MapReduce: The Map Step (Lecture 24)

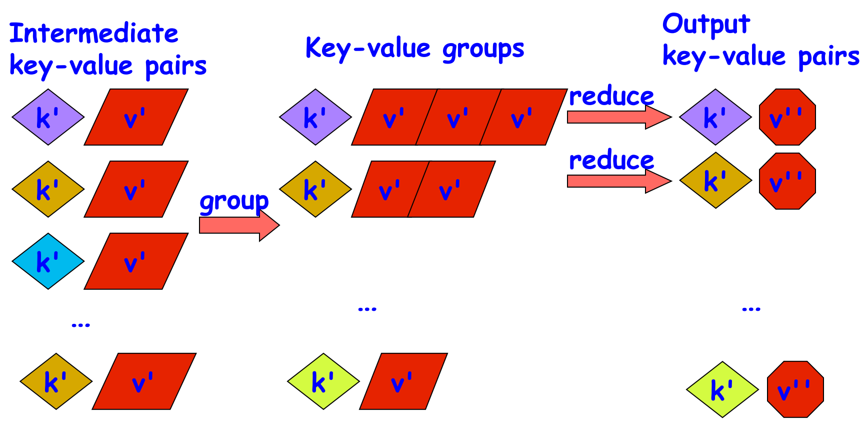
Input set of key-value pairs

Flattened intermediate set of key-value pairs



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

MapReduce: The Reduce Step



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

HJ Data-Driven Futures (Lecture 25)

ddfA = new DataDrivenFuture()

- Allocate an instance of a DDF object (container)

async await(ddfA, ddfB, …) <Stmt>

- Create a new async task to start executing **Stmt** after all of **ddfA, ddfB, …** become *available*
- Task is said to be *enabled* when **ddfA, ddfB, …** become available

ddfA.put(V)

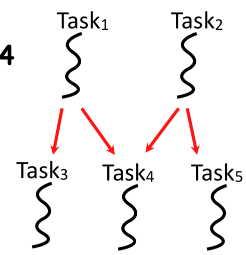
- Store object V in **ddfA**, thereby making **ddfA** available
- Single-assignment rule: at most one put is permitted on a given DDF

ddfA.get()

- Return value stored in **ddfA**
- Can only be performed by async's that contain **ddfA** in their await clause (no blocking is necessary)

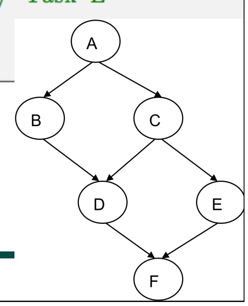
Figure 2: Example Habanero Java code fragment with Data-Driven Futures

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
    async left.put(leftBuilder()); // Task1
    async right.put(rightBuilder()); // Task2
    async await ( left ) leftReader(left); // Task3
    async await ( right ) rightReader(right); // Task5
    async await ( left, right )
              bothReader( left, right); // Task4
}
```



Listing 1: use of DDFs with empty objects

```
finish {
  DataDrivenFuture ddfA = new DataDrivenFuture();
  DataDrivenFuture ddfB = new DataDrivenFuture();
  DataDrivenFuture ddfC = new DataDrivenFuture();
  DataDrivenFuture ddfD = new DataDrivenFuture();
  DataDrivenFuture ddfE = new DataDrivenFuture();
  async { . . . ; ddfA.put(""); } // Task A
  async await(ddfA) { . . . ; ddfB.put(""); } // Task B
  async await(ddfA) { . . . ; ddfC.put(""); } // Task C
  async await(ddfB,ddfC) { . . . ; ddfD.put(""); } // Task D
  async await(ddfC) { . . . ; ddfE.put(""); } // Task E
  async await(ddfD,ddfE) { . . . } // Task F
} // finish
```



java.lang.Thread class (Lecture 27)

- Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program's main() method.
- Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

```
public class Thread extends Object implements Runnable {
  Thread() { ... } // Creates a new Thread
  Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
  void  run() { ... } // Code to be executed by thread
   // Case 1: If this thread was created using a Runnable object,
   //         then that object's run method is called
   // Case 2: If this class is subclassed, then the run() method
   //         in the subclass is called
  void start() { ... } // Causes this thread to start execution
  void join() { ... } // Wait for this thread to die
  void  join(long m) // Wait at most m milliseconds for thread to die
  static Thread currentThread() // Returns currently executing thread
  . . .
}
```

Listing 3: java.lang.Thread class

Objects and Locks in Java --- synchronized statements and methods (Lecture 28)

- Every Java object has an associated lock acquired via:
  - synchronized statements
    - synchronized(foo){
       // execute code while holding foo's lock
      }
  - synchronized methods
    - public synchronized void op1(){
       // execute op1 while holding 'this' lock
      }
- Language does not enforce any relationship between object used for locking and objects accessed in isolated code
  - If same object is used for locking and data access, then the object behaves like monitors
- Locking and unlocking are automatic
  - Locks are released when a synchronized block exits
    - By normal means: end of block reached, **return**, **break**
    - When an exception is thrown and not caught

Example: Obvious Deadlock

- This code can deadlock if `leftHand()` and `rightHand()` are called concurrently from different threads
  - Because the locks are not acquired in the same order

```
public class ObviousDeadlock {
   . . .
   public void leftHand() {
       synchronized(lock1) {
           synchronized(lock2) {
               for (int i=0; i<10000; i++)
                   sum += random.nextInt(100);
           }
       }
   }
   public void rightHand() {
       synchronized(lock2) {
           synchronized(lock1) {
               for (int i=0; i<10000; i++)
                   sum += random.nextInt(100);
           }
       }
   }
}
```


Dynamic Order Deadlocks

- There are even more subtle ways for threads to deadlock due to inconsistent lock ordering
  - Consider a method to transfer a balance from one account to another:

```
public class SubtleDeadlock {
    public void transferFunds(Account from,
                              Account to,
                              int amount) {
        synchronized (from) {
            synchronized (to) {
                from.subtractFromBalance(amount);
                to.addToBalance(amount);
            }
        }
    }
}
```

  - What if one thread tries to transfer from A to B while another tries to transfer from B to A ?

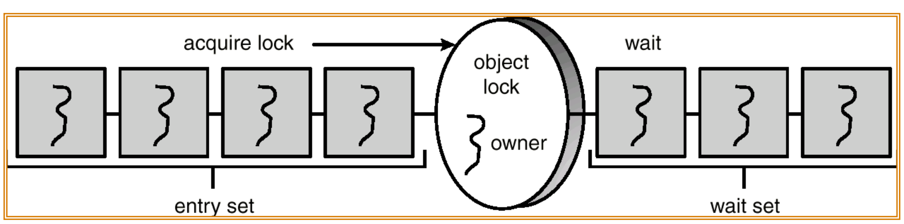
    Inconsistent lock order again – Deadlock!

The Java wait() Method (Lecture 29)

- A thread can perform a wait() method on an object that it owns:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set
- Causes thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- Since interrupts and spurious wakeups are possible, this method should always be used in a loop e.g.,

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();
    ... // Perform action appropriate to condition
}
```


Entry and Wait Sets



The notify() Method

When a thread calls notify(), the following occurs:

1. selects an arbitrary thread *T* from the wait set
2. moves *T* to the entry set
3. sets *T* to Runnable

T can now compete for the object's lock again

Multiple Notifications

- notify() selects an arbitrary thread from the wait set. *This may not be the thread that you want to be selected.
- Java does not allow you to specify the thread to be selected
- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
- notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set

java.util.concurrent.Executor interface (Lecture 31)

- Framework for asynchronous task execution
- A design pattern with a single-method interface
  - interface Executor { void execute(Runnable w); }
- Separate work from workers (what vs how)
  - ex.execute(work), not new Thread(..).start()
- Cancellation and shutdown support
- Usually created via `Executors` factory class
  - Configures flexible `ThreadPoolExecutor`
  - Customize shutdown methods, before/after hooks, saturation policies, queuing
- Normally use group of threads: ExecutorService

Executor Framework Features

- There are a number of factory methods in `Executors`
  - `newFixedThreadPool(n)`, `newCachedThreadPool()`, `newSingleThreadedExecutor()`
- Can also instantiate `ThreadPoolExecutor` directly
- Can customize the thread creation and teardown behavior
  - Core pool size, maximum pool size, timeouts, thread factory
- Can customize the work queue
  - Bounded vs unbounded
  - FIFO vs priority-ordered
- Can customize the *saturation policy* (queue full, maximum threads)
  - discard-oldest, discard-new, abort, caller-runs
- Execution hooks for subclasses
  - `beforeExecute()`, `afterExecute()`

ExecutorService interface

- ExecutorService extends Executor interface with lifecycle management methods e.g.,
  - shutdown()

    Graceful shutdown – stop accepting tasks, finish executing already queued tasks, then terminate
  - shutdownNow()

    Abrupt shutdown – stop accepting tasks, attempt to cancel running tasks, don't start any new tasks, return unstarted tasks
- An ExeuctorService is a group of thread objects, each running some variant of the following loop
  - *while (…) { get work and run it; }*
- ExecutorService's take responsibility for the threads they create
  - Service owner starts and shuts down ExecutorService
  - ExecutorService starts and shuts down threads

Volatile Variables (Lecture 32)

- Java provides a "light" form of synchronization/fence operations in the form of volatile variables
- Volatile variables guarantee *visibility*
  - An access of a volatile variable is like an access of a synchronization variable in the Weak Ordering model
  - Adds serialization edges to computation graph due to isolated read/ write operations
- Incrementing a volatile variable (++v) is <u>not thread-safe</u>
  - Increment operation looks atomic, but isn't (read and write are two separate operations)
- Volatile variables are best suited for flags that have no dependencies

```
-volatile boolean asleep
-while (! asleep)
    ++sheep;
```

•Warning: a volatile declaration on an array variable may not give you the semantics you expect

COMP 322: Fundamentals of Parallel Programming

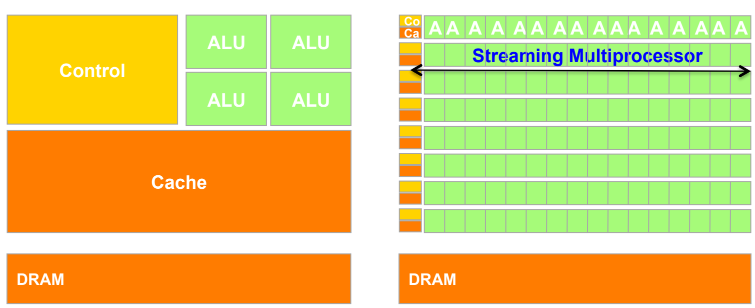
https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 33: GPGPU Programming with CUDA

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

CPUs and GPUs have fundamentally different design philosophies (Lecture 33)

Single CPU core **Multiple GPU processors**



Process Flow of a CUDA Kernel Call (Figure 2)

- Data parallel programming architecture from NVIDIA
  - Execute programmer-defined kernels on extremely parallel GPUs
  - CUDA program flow:
    1. Push data on device
    2. Launch kernel
    3. Execute kernel and memory accesses in parallel
    4. Pull data off device
- Device threads are launched in batches
  - Blocks of Threads, Grid of Blocks
- Explicit device memory management
  - cudaMalloc, cudaMemcpy, cudaFree, etc.

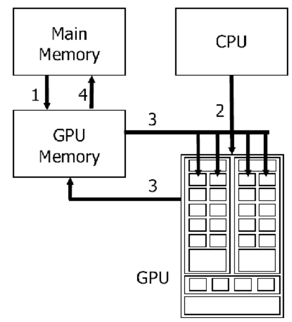


Figure source: Y. Yan et. al "JCUDA: a Programmer Friendly Interface for Accelerating Java Programs with CUDA." Euro-Par 2009.

37

Execution of a CUDA program (Figure 3)

- Integrated host+device application
  - Serial or modestly parallel parts on CPU host
  - Highly parallel kernels on GPU device

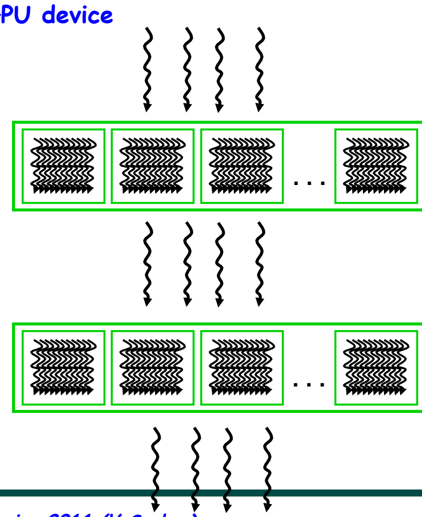
Host Code
(small number of threads)

Device Kernel
(large number of threads)

Host Code
(small number of threads)

Device Kernel
(large number of threads)

Host Code
(small number of threads)

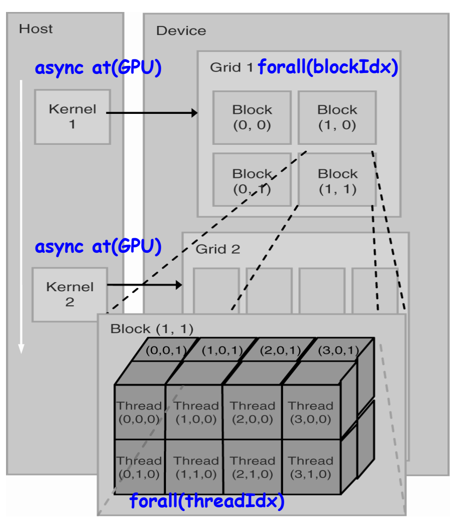


Logical Structure of a CUDA kernel invocation (Listing 1)

```
finish async at(GPU) {
  // Parallel execution of blocks in grid
  forall (point[blockIdx.x,blockIdx.y] : [0:gridDim.x-1,0:gridDim.y-1]) {
    // Parallel execution of threads in block (blockIdx.x,blockIdx.y)
    forall (point[threadIdx.x,threadIdx.y,threadIdx.z]
              : [0:blockDim.x-1,0:blockDim.y-1,0:blockDim.z-1]) {
      // Perform kernel computation as function of blockIdx.x,blockIdx.y
      // and threadIdx.x,threadIdx.y,threadIdx.z
      . . .
      next; // barrier synchronizes inner forall only (__syncthreads)
      . . .
    } // forall threadIdx.x,threadIdx.y,threadIdx.z
  } // forall blockIdx.x, blockIdx.y
} // finish async (GPU)
```

Listing 1: Logical structure of a CUDA kernel invocation

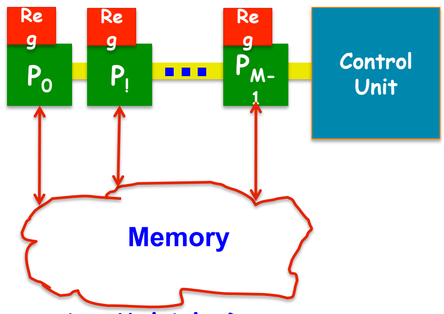
Organization of a CUDA grid (Figure 4)



Impact of Single Control Unit for a Block of Threads executing on an SM (Lecture 34)

Control flow example

```
if (threadIdx >= 2) {
    out[threadIdx] += 100;
}
else {
    out[threadIdx] += 10;
}
```



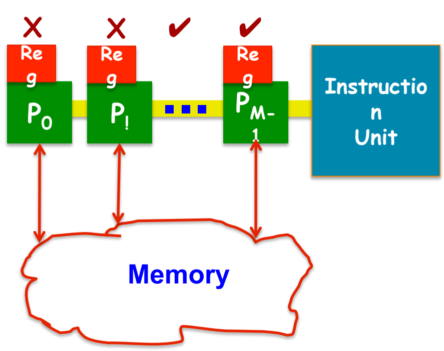
compare threadIdx,2

SIMD = Single Instruction Multiple Data

SIMD Execution of Control Flow

Control flow example

```
if (threadIdx.x >= 2) {
    out[threadIdx.x] += 100;
}
else {
    out[threadIdx.x] += 10;
}
```

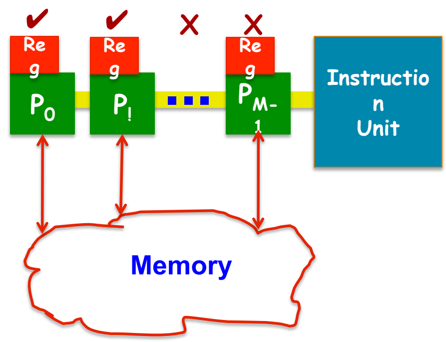


/* Condition code cc = true branch set by predicate execution */
(CC) LD R5, &(out+threadIdx.x)
(CC) ADD R5, R5, 100
(CC) ST R5, &(out+threadIdx.x)

SIMD Execution of Control Flow

Control flow example

```
if (threadIdx >= 2) {
    out[threadIdx] += 100;
}
else {
    out[threadIdx] += 10;
}
```




```
/* possibly predicated
using CC */
(not CC) LD R5,
            &(out
+threadIdx)
(not CC) ADD R5, R5,
10
(not CC) ST R5,
            &(out
+threadIdx)
```


Divergence

- Divergent paths
  - What happens if different threads within a block take different control flow paths?
  - N divergent paths
    - An N-way divergent block is serially issued over the N different paths
    - Performance decreases by about a factor of N
    - GPU is better suited for blocks of threads with low intra-block divergence
    - Multicore CPU is better equipped to handle divergence than CPU
- Implementation note
  - Current GPUs subdivide a block of threads into “warps” of a fixed size (e.g., 32 or 64)
  - Divergence can be tolerated among threads in different warps, but not among threads in the same warp
  - If you avoid divergence within a block, you will also guarantee the absence of divergence within a warp

Desirable Properties of Parallel Program Executions (Lecture 35)

- Data-race freedom (Lecture 6)
- Termination
  - But some applications are designed to be non-terminating
- Liveness = a program's ability to make progress in a timely manner
- Different levels of liveness guarantees (from weaker to stronger)
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Bounded wait

Scope of Course (Lecture 1)

- Fundamentals of parallel programming
  - Task creation and termination, computation graphs, scheduling theory, futures, forall parallel loops, barrier synchronization (phasers), isolation & mutual exclusion, task affinity, bounded buffers, data flow, threads, data races, deadlock, memory models
- Introduction to parallel algorithms
- Habanero-Java (HJ) language, developed in the Habanero Multicore Software Research project at Rice
- Abstract executable performance model for HJ programs
- Java Concurrency
- Written assignments
- Programming assignments
  - Abstract metrics
  - Real parallel systems (8-core Intel, Rice SUG@R system)
- Beyond HJ and Java: introduction to CUDA and MPI