COMP 322: Fundamentals of Parallel Programming

Lecture 38: Comparison of Programming Models

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

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

Acknowledgments

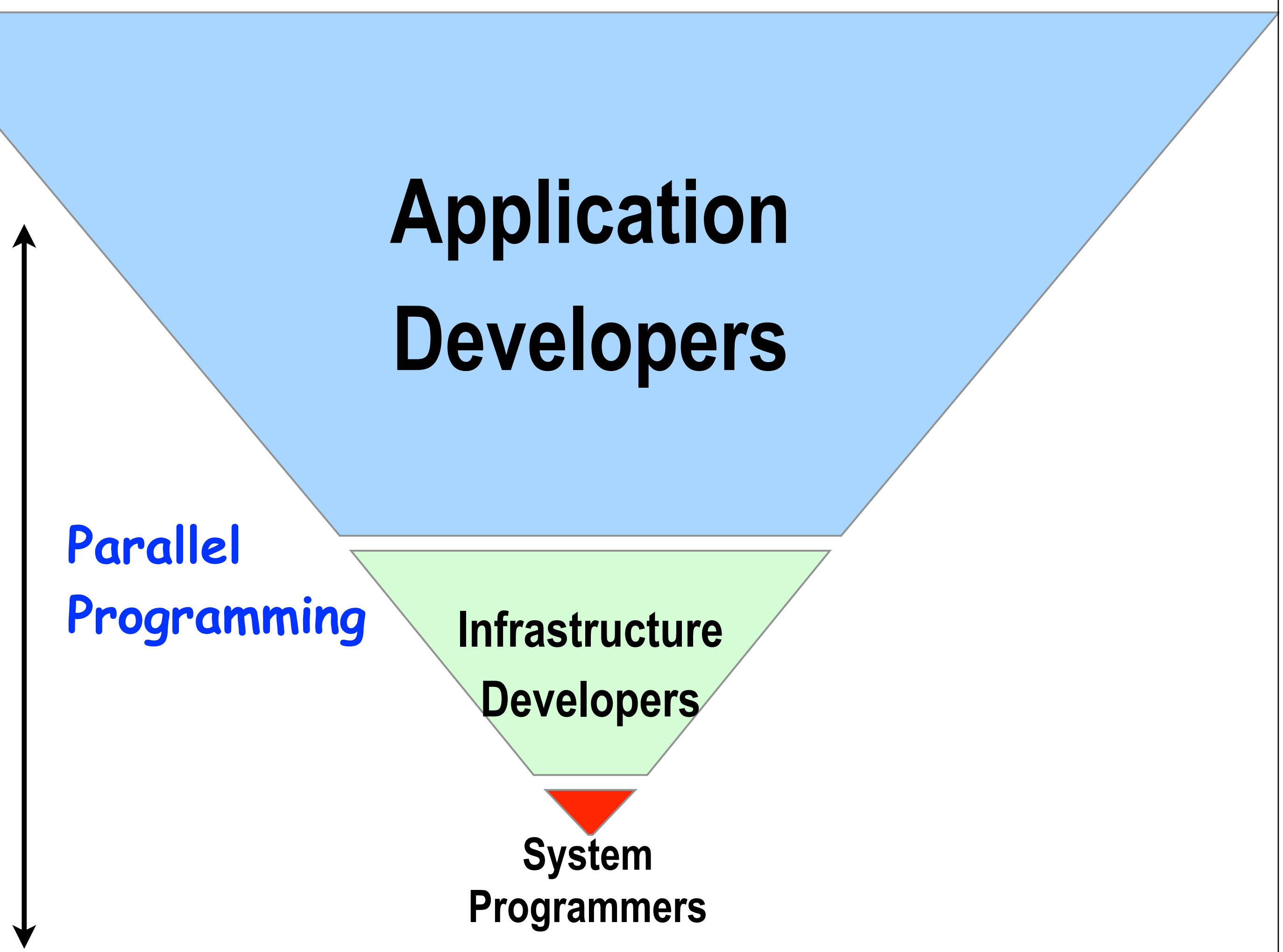
- "Introduction to Parallel Computing" by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003, and accompanying slides
  - http://www-users.cs.umn.edu/~karypis/parbook/
- Slides from COMP 422 course at Rice University
  - http://www.clear.rice.edu/comp422/
- Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell. "Pthreads Programming: A POSIX Standard for Better Multiprocessing." O'Reilly Media, 1996
- Slides from OpenMP tutorial given by Ruud van der Paas at HPCC 2007
  - http://www.tlc2.uh.edu/hpcc07/Schedule/OpenMP
- "Towards OpenMP 3.0", Larry Meadows, HPCC 2007 presentation
  - http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc07_Larry.ppt
- Pthreads: A Brief Introduction, CSCI 8530 lecture, University of Nebraska Omaha
  - http://cs.unomaha.edu/~stanw/053/csci8530/pthreads.pdf
- "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
  - Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html
- Tim Warburton, Rice University, "Introduction to GPGPU Programming"
  - 5-day course taught at Danish Technical University (DTU) in May 2011
- David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

Parallel Programming is a Cross-Cutting Concern

Developer Pyramid (not drawn to scale!)

Software Stack

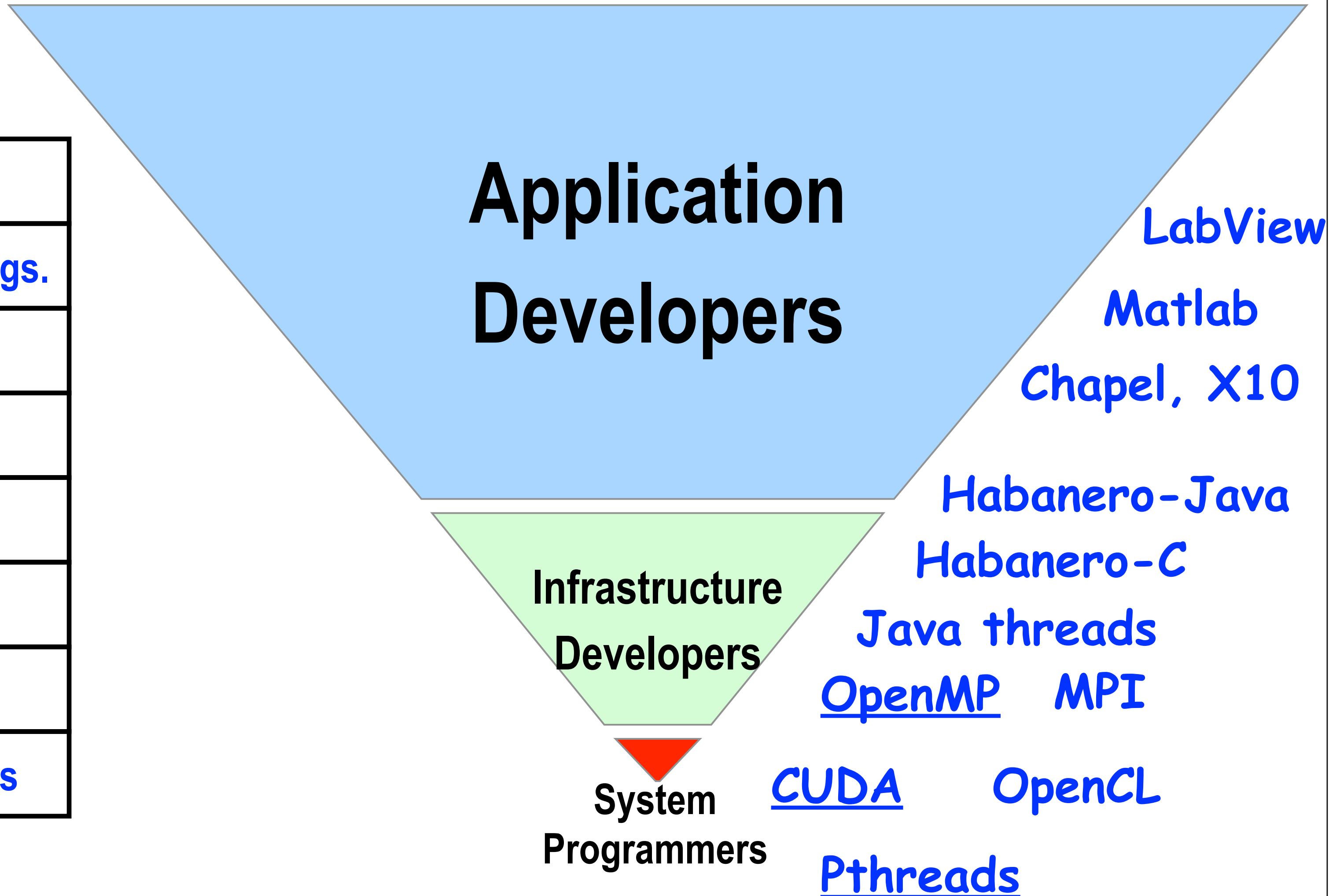
Application
Domain-Specific Langs.
Middleware
Programming
Runtime Systems
Compilers
System Libraries
OS and Hypervisors



Different Parallel Programming Models for different Levels of Developer Pyramid and Software Stack

Software Stack

Application
Domain-Specific Langs.
Middleware
Programming
Runtime Systems
Compilers
System Libraries
OS and Hypervisors



Outline

- Pthreads
- OpenMP
- CUDA

POSIX Thread API (Pthreads)

- Standard user threads API supported by most vendors
- Library interface, intended for system programmers
- Concepts behind Pthreads interface are broadly applicable
  - largely independent of the API
  - useful for programming with other thread APIs as well
    - Windows threads
    - Solaris threads
    - Java threads
    - …
- Threads are peers, unlike Linux/Unix processes
  - no parent/child relationship

PThread Creation

Asynchronously invoke `thread_function` in a new thread

```
#include <pthread.h>
int pthread_create(
   pthread_t *thread_handle, /* returns handle here */
   const pthread_attr_t *attribute,
   void * (*thread_function)(void *),
   void *arg); /* single argument; perhaps a structure */
```

`attribute` created by `pthread_attr_init`

contains details about
- whether scheduling policy is inherited or explicit
- scheduling policy, scheduling priority
- stack size, stack guard region size

Can use NULL for `pthread_attr_init` for default values

Pthread Termination

- A thread terminates by calling the function `pthread_exit()`. A single argument, a pointer to a void* object, is supplied as the argument to pthread_exit. This value is returned to any thread that has blocked while waiting for this thread to exit.

- Suspend parent thread until child thread terminates

```
#include <pthread.h>
int pthread_join (
  pthread_t thread, /* thread id */
  void **ptr); /* ptr to location for return code a terminating
                  thread passes to pthread_exit */
```

Example: Creation and Termination (`main`)

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 32
void *compute_pi (void *);
...
int main(...) {
    ...
    pthread_t p_threads[NUM_THREADS];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    for (i=0; i< NUM_THREADS; i++) {
       hits[i] = i;
       pthread_create(&p_threads[i], &attr, compute_pi,
           (void*) &hits[i]);
    }
    for (i=0; i< NUM_THREADS; i++) {
       pthread_join(p_threads[i], NULL);
       total_hits += hits[i];
    }
    ...
}
```

default attributes

thread function

thread argument

Example of Implementing a Reduction Using Mutex Locks

```
pthread_mutex_t cost_lock;
...
int main() {
   ...
   pthread_mutex_init(&cost_lock, NULL);
   ...
}
void *find_best(void *list_ptr) {
   ...
   pthread_mutex_lock(&cost_lock);   /* lock the mutex */
  if (my_cost < best_cost)
      best_cost = my_cost;
   pthread_mutex_unlock(&cost_lock); /* unlock the mutex */
}
```

use default (normal) lock type

critical section

Pthread's Condition Variable API

```
/* initialize or destroy a condition variable */
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

/* block until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *wtime);

/* signal one or all waiting threads that condition is true */
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

abort wait if time exceeded

wake one

wake all

Condition Variable Producer-Consumer (main)

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
}
```

default initializations

Producer Using Condition Variables

```
void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
      create_task();
      pthread_mutex_lock(&task_queue_cond_lock);
      while (task_available == 1)
         pthread_cond_wait(&cond_queue_empty,
             &task_queue_cond_lock);
      insert_into_queue();
      task_available = 1;
      pthread_cond_signal(&cond_queue_full);
      pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

note loop

releases mutex on wait

reacquires mutex when woken

Consumer Using Condition Variables

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_full,
                &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```

note loop

releases mutex on wait

reacquires mutex when woken

Composite Synchronization Constructs

- Pthreads provides only basic synchronization constructs
- Build higher-level constructs from basic ones e.g., *barriers*
  - Pthreads extension includes barriers as synchronization objects (available in Single UNIX Specification)
    - Enable by #define _XOPEN_SOURCE 600 at start of file
  - Initialize a barrier for count threads
    - `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrier attr_t *attr, int count);`
  - Each thread waits on a barrier by calling
    - `int pthread_barrier_wait(pthread_barrier_t *barrier);`
  - Destroy a barrier
    - `int pthread_barrier_destroy(pthread_barrier_t *barrier);`
- Java threads and HJ worker threads are also implemented as pthreads

Summary of key features in Pthreads

Pthreads construct	Related HJ/Java constructs
pthread_create()	HJ's async; Java's "new Thread" and "Thread.start()"
pthread_join()	HJ's finish & future get(); Java's "Thread.join()"
pthread_mutex_lock()	HJ's begin-isolated, actors; Java's begin-synchronized, and lock() libray calls
pthread_mutex_unlock()	HJ's end-isolated, actors; Java's begin-synchronized, and lock() libray calls
pthread_cond_signal()	Deterministic use: HJ's phasers; Nondeterministic use: j.u.c.locks.condition
pthread_cond_wait()	Deterministic use: HJ's phasers; Nondeterministic use: j.u.c.locks.condition

Outline

- Pthreads
- <u>OpenMP</u>
- CUDA

What is OpenMP?

- **Well-established standard for writing shared-memory parallel programs in C, C++ Fortran**
- **Programming model is expressed via**
  - —Pragmas/directives (not language extensions)
  - —Runtime routines
  - —Environment variables

—Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)

- —Latest specification: Version 3.0 (May 2008)
- —Previous specification: Version 2.5 (May 2005)

A first OpenMP example

For-loop with independent iterations

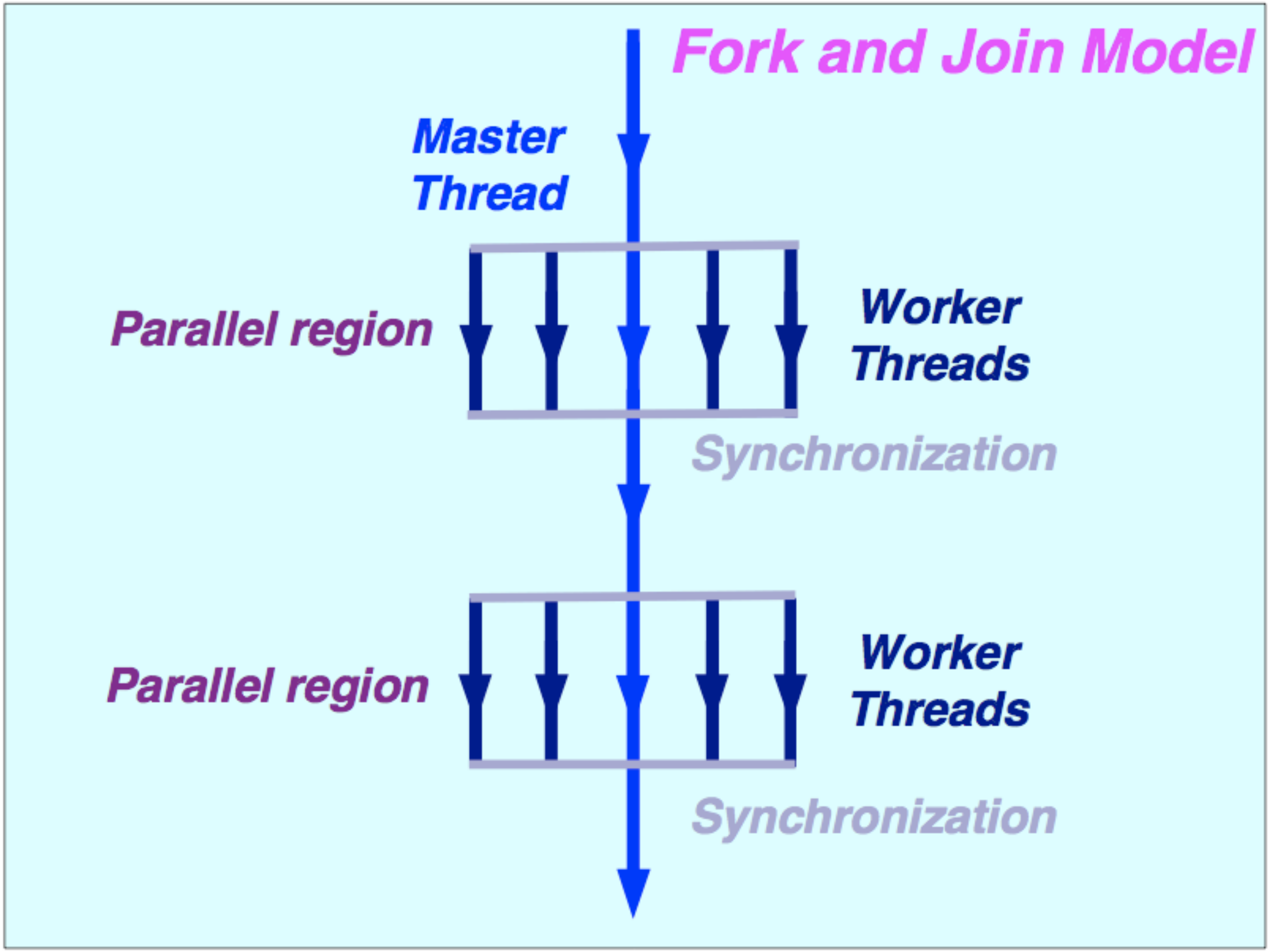
```
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for  \
        shared(n, a, b, c)\
        private(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out
```

The OpenMP Execution Model



Terminology

- ***OpenMP Team := Master + Workers***
- ***A Parallel Region is a block of code executed by all threads simultaneously***
  - *The master thread always has thread ID 0*
  - *Thread adjustment (if enabled) is only done before entering a parallel region*
  - *Parallel regions can be nested, but support for this is implementation dependent*
  - *An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially*
- ***A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work***

Parallel Region

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this is executed in parallel"

} (implied barrier)
```

A parallel region is a block of code executed by multiple threads simultaneously, and supports the following clauses:

if	(*scalar expression*)	
private	(*list*)	
shared	(*list*)	
default	(*none|shared*)	(*C/C++*)
default	(*none|shared|private*)	(*Fortran*)
reduction	(*operator*: *list*)	
copyin	(*list*)	
firstprivate	(*list*)	
num_threads	(*scalar_int_expr*)	

Work-sharing constructs in a Parallel Region

```
#pragma omp for
{
    ....
}
```

```
#pragma omp sections
{
        ....
}
```

```
#pragma omp single
{
        ....
}
```

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch any new threads

```
#pragma omp parallel
#pragma omp for
    for (...)
```

```
#pragma omp parallel for
for (....)
```

with single work-sharing construct e.g.,

Legality constraints for work-sharing constructs

- Each worksharing region must be encountered by all threads in a team or by none at all.
- The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team.

```
#pragma omp parallel
{
  do {
    // c1 and c2 may depend on the OpenMP thread-id
    boolean c1 = … ; boolean c2 = … ;
    . . .
    if (c2) {
      // Start of work-sharing region with no wait clause
      #pragma omp …
      . . . // Worksharing statement
    } // if (c2)
  } while (! c1);
}
```

==> No OpenMP implementation checks for conformance with this rule

Example of work-sharing “omp for” loop

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)

Implicit finish

Like HJ’s forasync

Like HJ’s forasync

Reduction Clause in OpenMP

- The `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The syntax of the `reduction` clause is as follows
  - `reduction (operator: variable list).`
- The variables in the list are implicitly specified as being private to threads.
- The `operator` can be one of `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local instances of sum here */
}
/*sum here contains sum of all local instances of sum */
```

"single" and "master" constructs in a parallel region

Only one thread in the team executes the code enclosed

```
#pragma omp single [clause[[,] clause] ...]
{
        <code-block>
}
```

Only the master thread executes the code block

```
#pragma omp master
{<code-block>}
```

- Single and master are useful for computations that are intended for single-processor execution e.g., I/O and initializations
- There is no implied barrier on entry or exit of a single or master construct

`task` Construct

```
#pragma omp task [clause[[,]clause] ...]
            structured-block
```

where *clause* can be one of:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none  )
```

Example – parallel pointer chasing using tasks

```
#pragma omp parallel
{
  #pragma omp single private(p)
   {
    p = listhead ;
    while (p) {
        #pragma omp task
                process (p);
         p= p->next ;
        }
     }
}
```

Spawn call to process(p)

Implicit finish at end of parallel region

Example – parallel pointer chasing on multiple lists using tasks (nested parallelism)

```
#pragma omp parallel
{
   #pragma omp for private(p)
   for ( int i =0; i <numlists ; i++) {
       p = listheads [ i ] ;
       while (p ) {
       #pragma omp task
           process (p)
       p=next (p ) ;
        }
    }
}
```

Example: postorder tree traversal

```
void postorder(node *p) {
    if (p->left)
      #pragma omp task
        postorder(p->left);
    if (p->right)
      #pragma omp task
        postorder(p->right);
  #pragma omp taskwait // wait for child tasks
  process(p->data);
}
```

- Parent task suspended until children tasks complete

Summary of key features in OpenMP

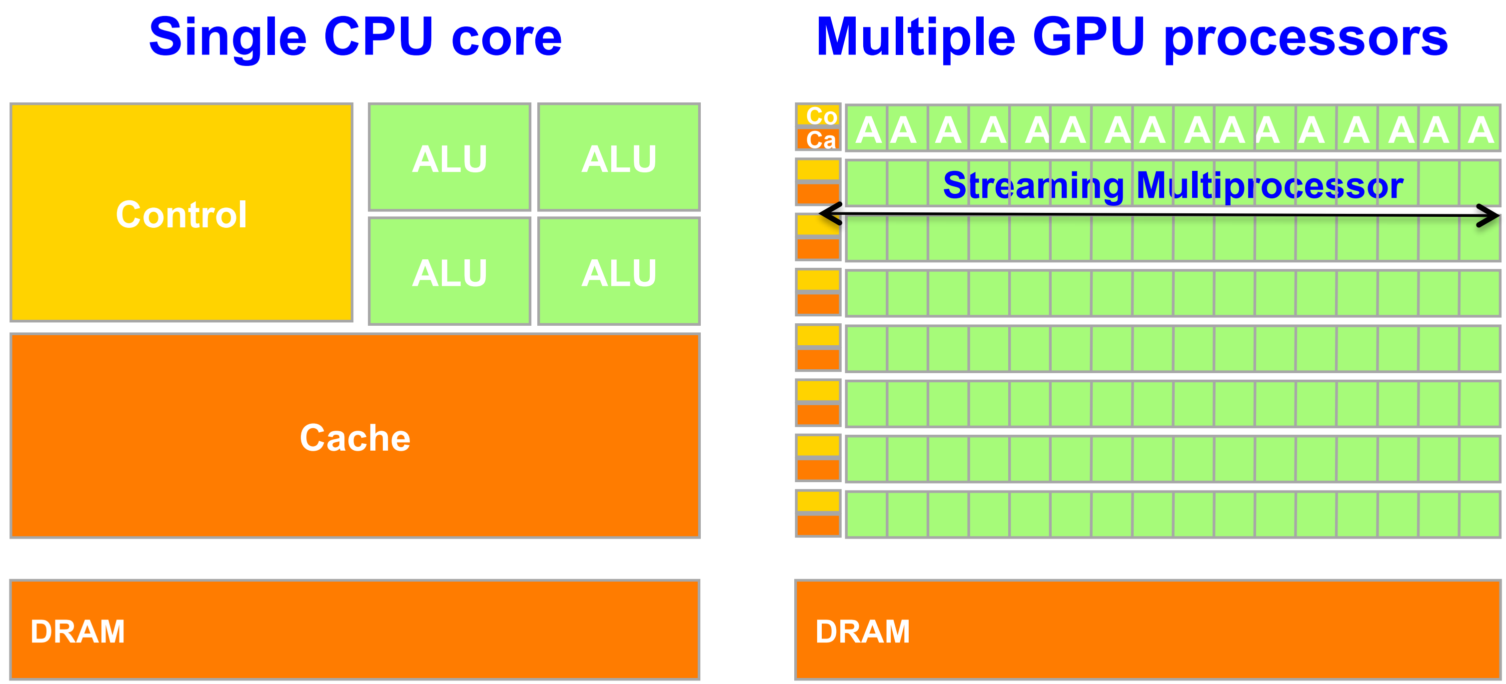
OpenMP construct	Related HJ/Java constructs
Parallel region #pragma omp parallel	HJ forall (forall iteration = OpenMP thread)
Work-sharing constructs: parallel loops, parallel sections	No direct analogy in HJ or Java
Barrier #pragma omp barrier	HJ forall-next on implicit phaser
Single #pragma omp single	HJ's forall-next-single on implicit phaser (but HJ does not support single + nowait)
Reduction clauses	HJ's finish accumulators (in forall)
Critical section #pragma omp critical	HJ's isolated statement
Task creation #pragma omp task	HJ's async statement
Task termination #pragma omp taskwait	HJ's finish statement

Outline

- Pthreads
- OpenMP
- CUDA

CPUs and GPUs have fundamentally different design philosophies

GPU = Graphics Processing Unit



GPUs are provided to accelerate graphics, but they can also be used for non-graphics applications that exhibit large amounts of data parallelism and require large amounts of "streaming" throughput

Process Flow of a CUDA Kernel Call (Compute Unified Device Architecture)

- 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.

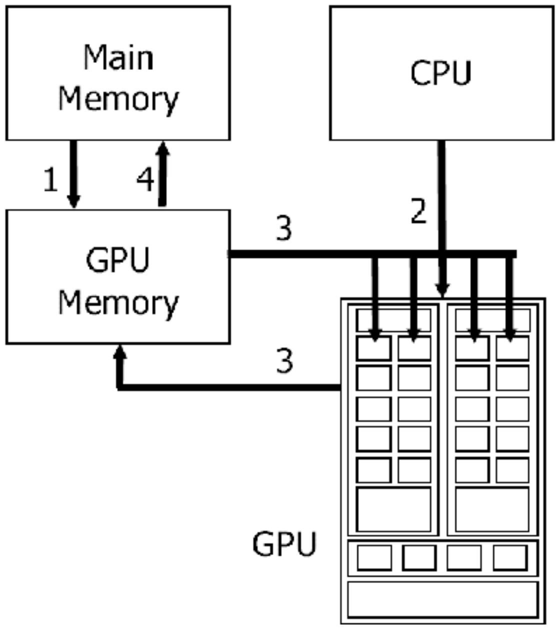


Figure source: Y. Yan et. al "JCUDA: a Programmer Friendly Interface for Accelerating Java Programs with CUDA." Euro-Par 2009.

Execution of a CUDA program

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

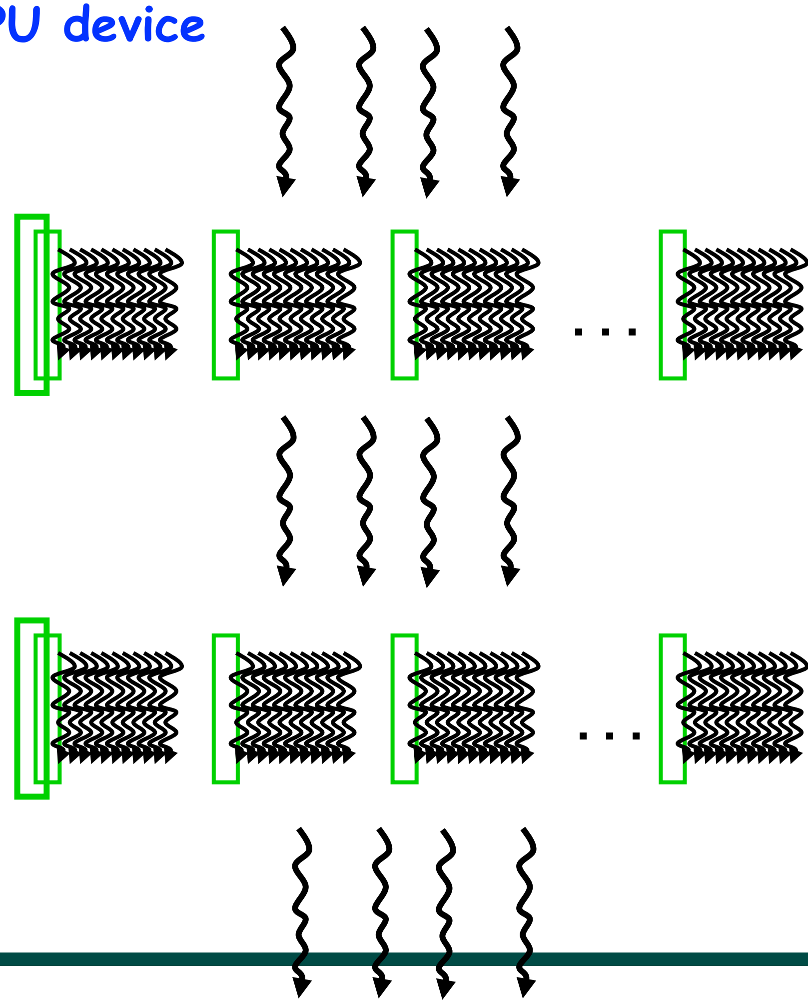
Host Code
(small number of threads)

Device Kernel
(large number of threads)

Host Code
(small number of threads)

Device Kernel
(large number of threads)

Host Code
(small number of threads)



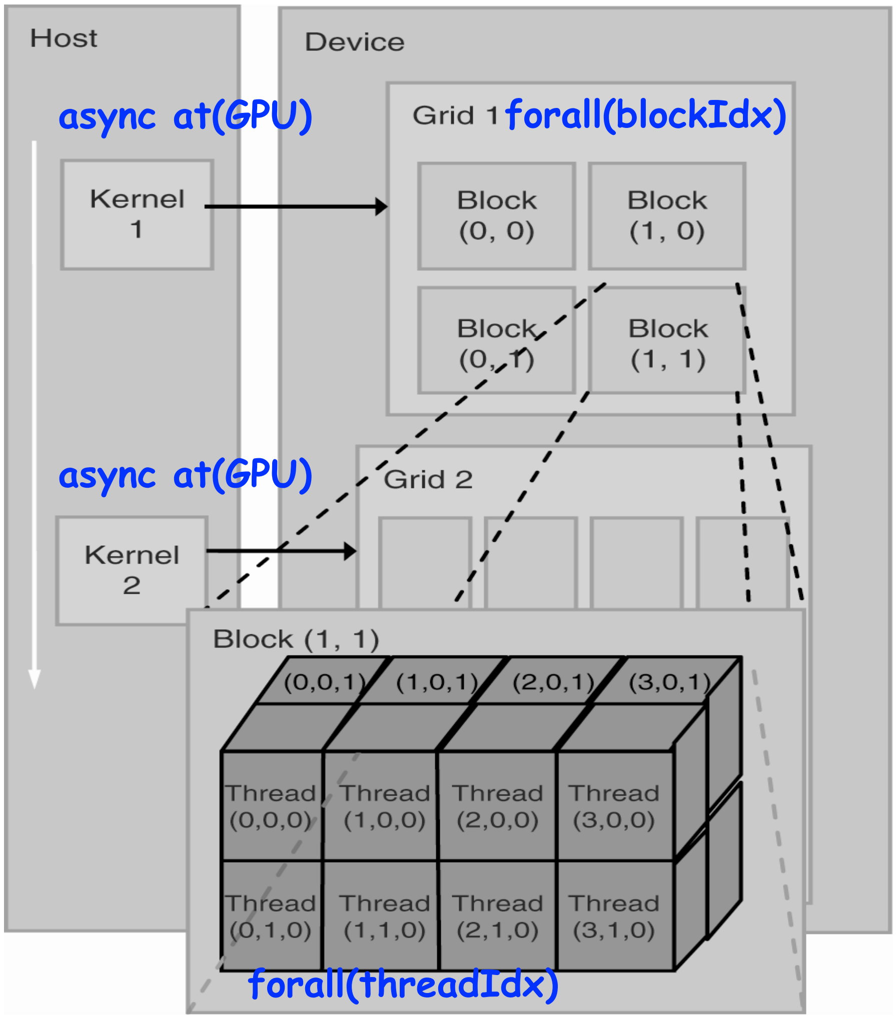
Matrix multiplication kernel code in CUDA (SPMD model with index = threadIdx)

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
     float Mdelement = Md[ty * Width + k];
     float Ndelement = Nd[k * Width + tx];
     Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```

Host Code in C for Matrix Multiplication

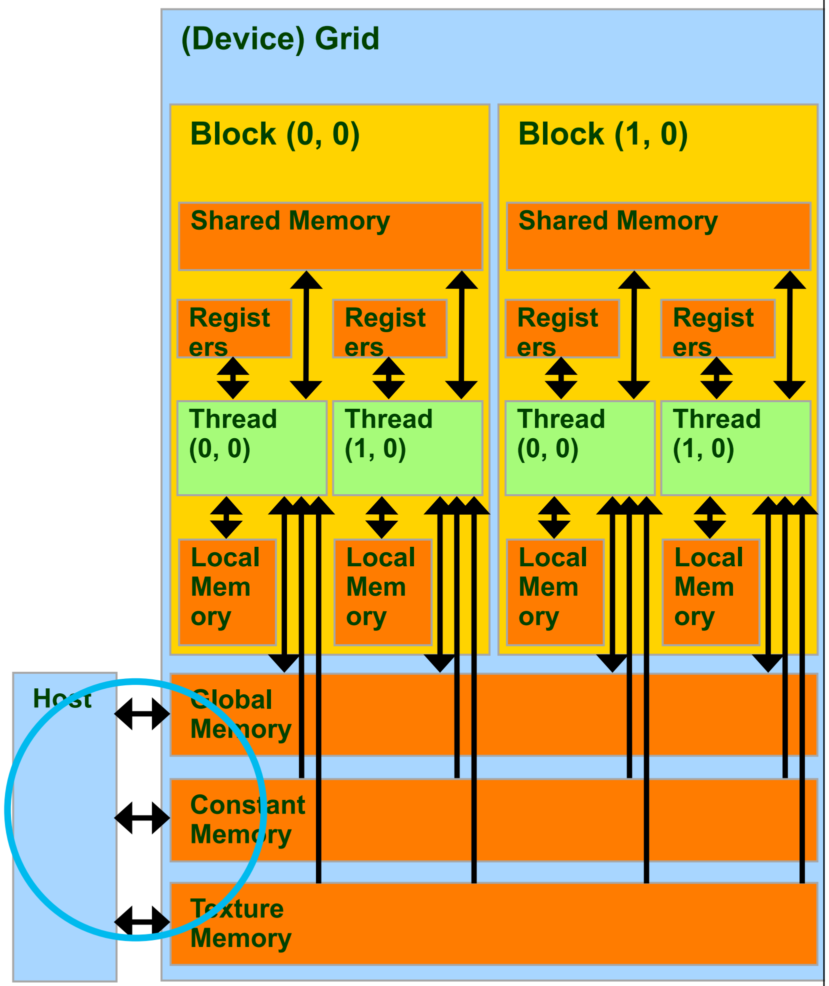
```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width*Width*sizeof(float); // matrix size
  float* Md, Nd, Pd; // pointers to device arrays
  cudaMalloc((void**)&Md, size); // allocate Md on device
  cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice); // copy M to Md
  cudaMalloc((void**)&Nd, size); // allocate Nd on device
  cudaMemcpy(Nd, M, size, cudaMemcpyHostToDevice); // copy N to Nd
  cudaMalloc((void**)&Pd, size); // allocate Pd on device
  dim3 dimBlock(Width,Width); dim3 dimGrid(1,1);
  // launch kernel (equivalent to "async at(GPU), forall, forall"
  MatrixMulKernel<<<dimGrid,dimBlock>>>(Md, Nd, Pd, Width);
  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost); // copy Pd to P
  // Free device matrices
  cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

Organization of a CUDA grid (Figure 4)



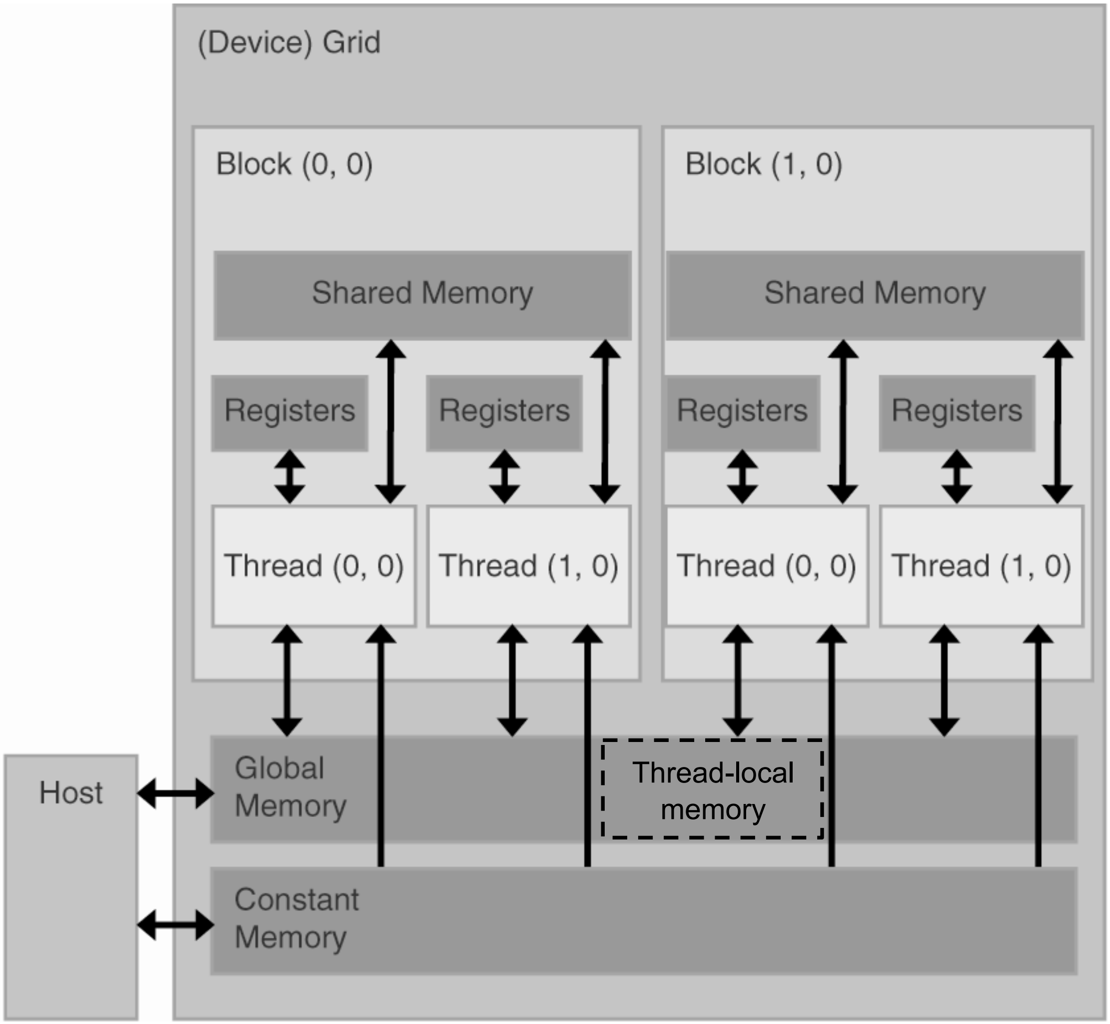
CUDA Host-Device Data Transfer

- cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
- copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of
  - —cudaMemcpyHostToHost
  - —cudaMemcpyHostToDevice
  - —cudaMemcpyDeviceToHost
  - —cudaMemcpyDeviceToDevice
- The memory areas may not overlap
- Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.



CUDA Storage Classes

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories

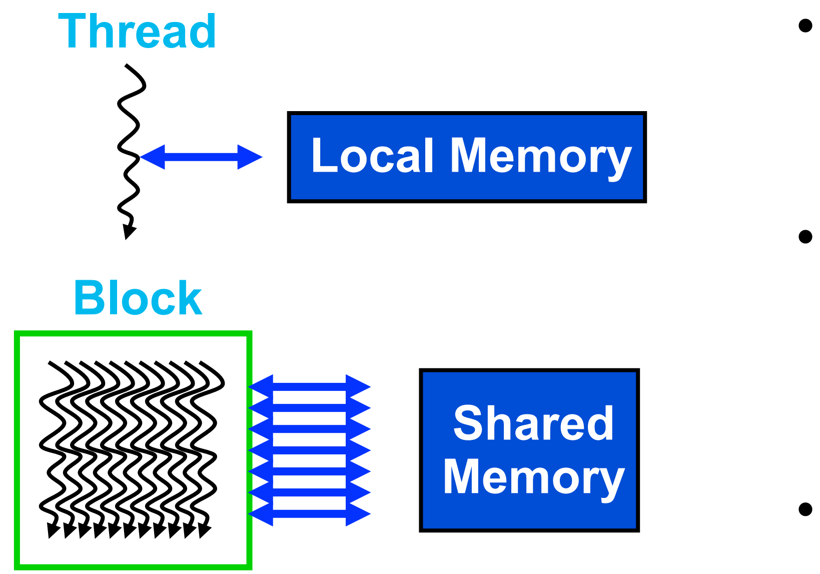


CUDA Variable Type Qualifiers

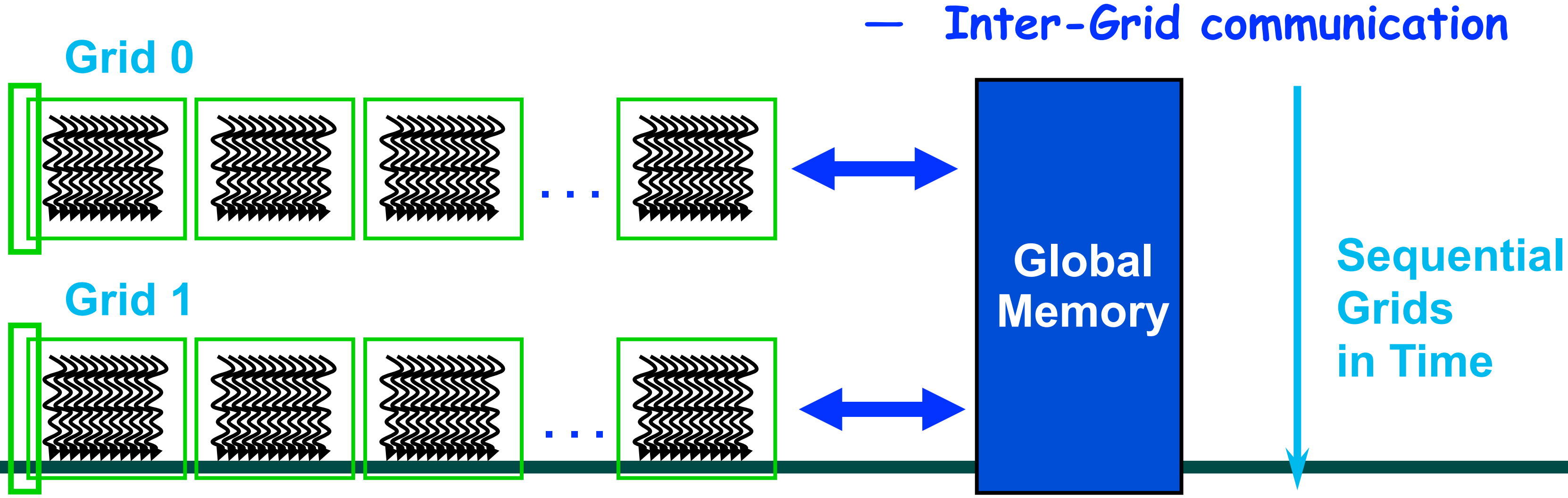
Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory
- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel: `__global__ void KernelFunc(float* ptr)`
  - Obtained as the address of a global variable: `float* ptr = &GlobalVar;`

CUDA Storage Classes



- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication



Summary of key features in CUDA

CUDA construct	Related HJ/Java constructs
Kernel invocation, <<<. . .>>>	async at(gpu-place)
1D/2D grid with 1D/2D/3D blocks of threads	Outer 1D/2D forall with inner 1D/2D/3D forall
Intra-block barrier, __syncthreads()	HJ forall-next on implicit phaser for inner forall
cudaMemcpy()	No direct equivalent in HJ/Java (can use System.arraycopy() if needed)
Storage classes: local, shared, global	No direct equivalent in HJ/Java (method-local variables are scalars)