CUDA Programming Introduction

Andreas Moshovos

Winter 2009

Some slides/material from:

UIUC course by Wen-Mei Hwu and David Kirk

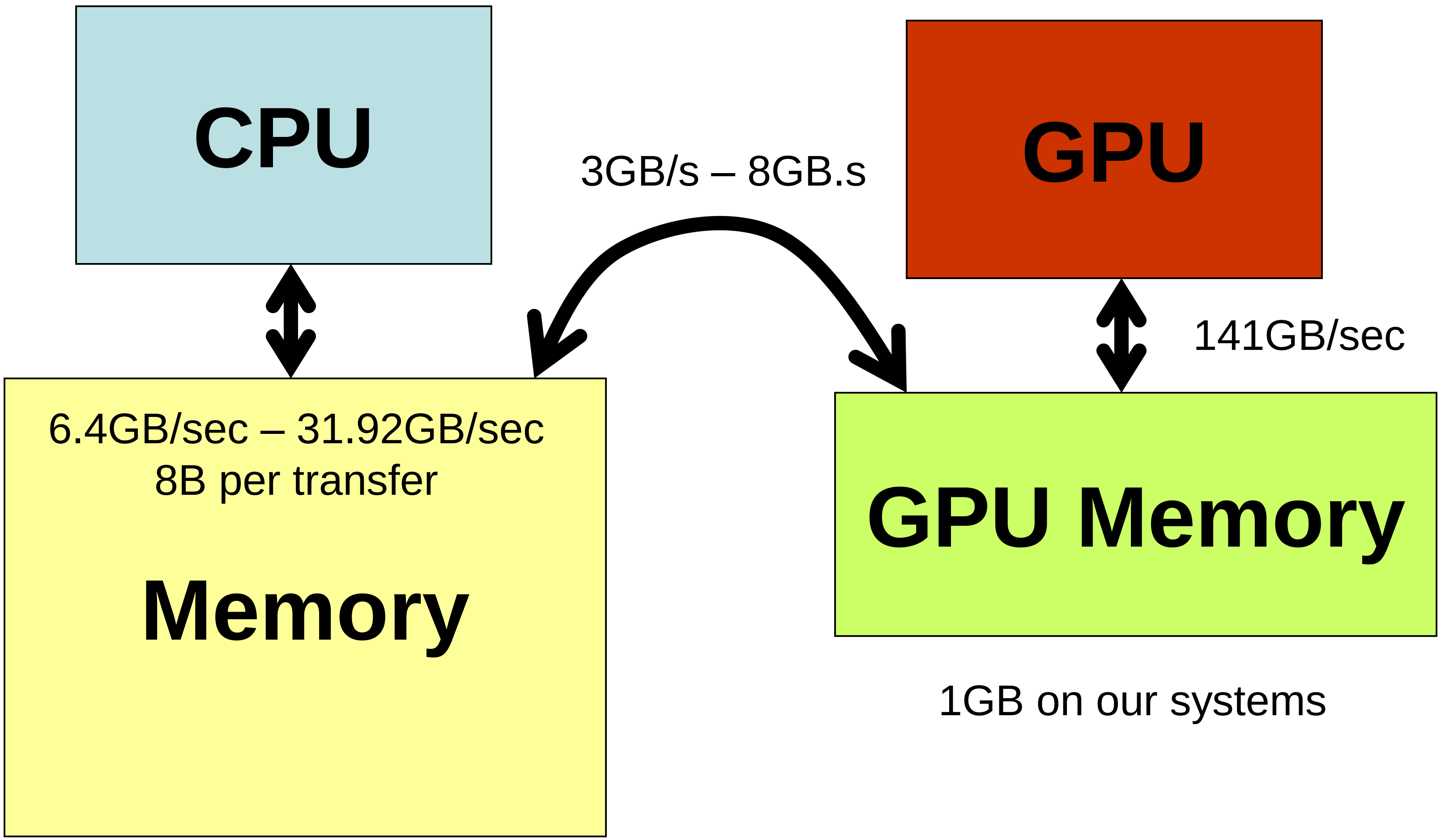
UCSB course by Andrea Di Blas

Universitat Jena by Waqar Saleem

NVIDIA by Simon Green

Programmer's view

- GPU as a co-processor



Target Applications

```
int a[N]; // N is large
for all elements of a compute
   a[i] = a[i] * fade
```

- Lots of **independent** computations
  - CUDA thread need not be independent

Programmer's View of the GPU

- GPU: a compute device that:
  - Is a coprocessor to the CPU or **host**
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as **kernels** which run in parallel on many threads

Why are threads useful?

- Concurrency:
  - Do multiple things in parallel

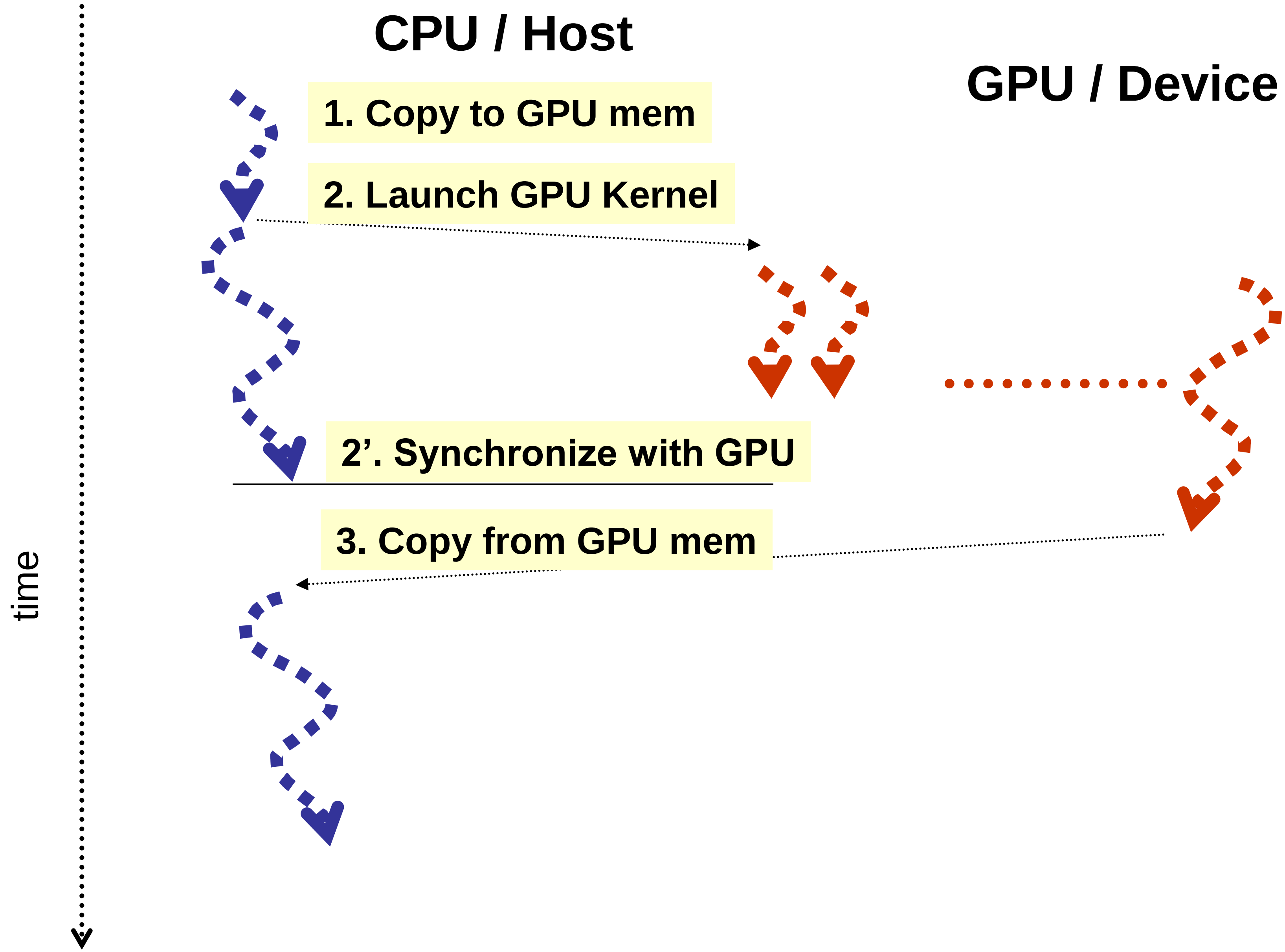
Needs more functional units

  - Put more hardware → Get higher performance

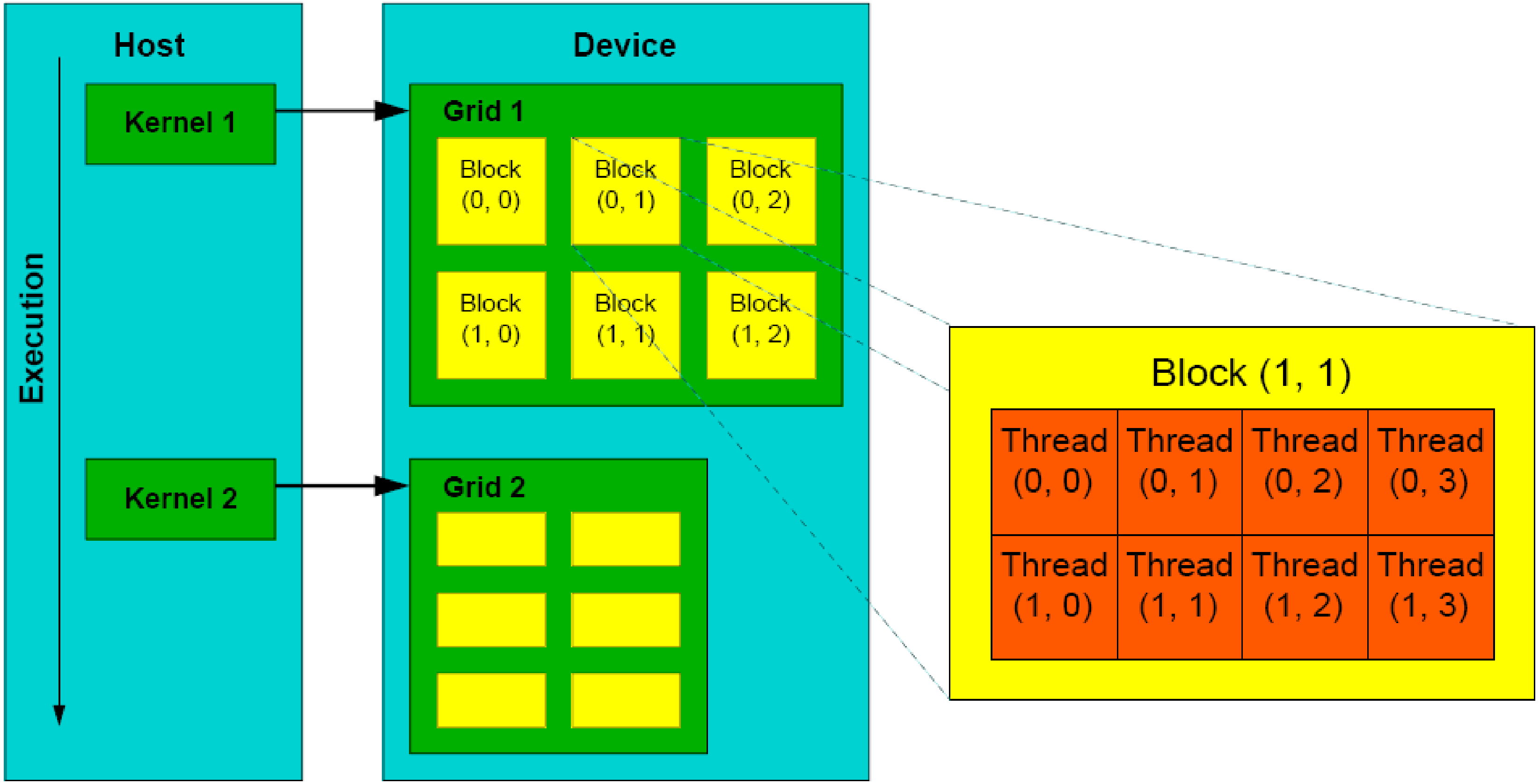
GPU vs. CPU Threads

- GPU threads are extremely lightweight
  - Very little creation overhead
  - In the order of microseconds
- GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few

Execution Timeline



GBT: Grids of Blocks of Threads



Why? Realities of integrated circuits: need to cluster computation and storage to achieve high speeds

Grids of Blocks of Threads: Dimension Limits

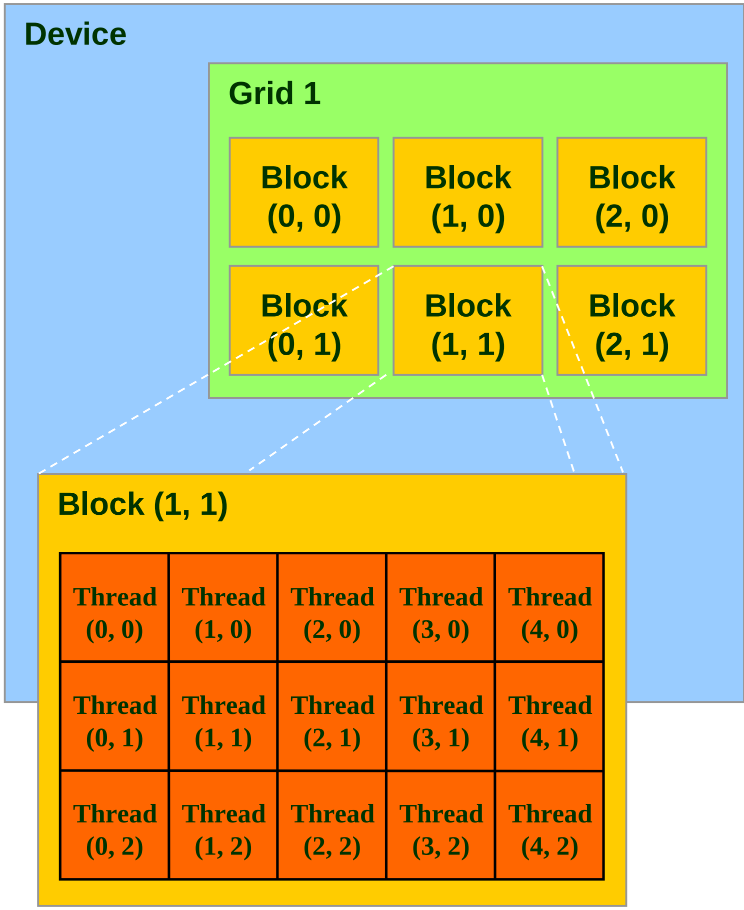
- Grid of Blocks 1D or 2D
  - Max x: 65535
  - Max y: 65535

- Block of Threads: 1D, 2D, or 3D
  - Max number of threads: 512
  - Max x: 512
  - Max y: 512
  - Max z: 64

- Limits apply to Compute Capability 1.0, 1.1, 1.2, and 1.3
  - GTX280 = 1.3

Block and Thread IDs

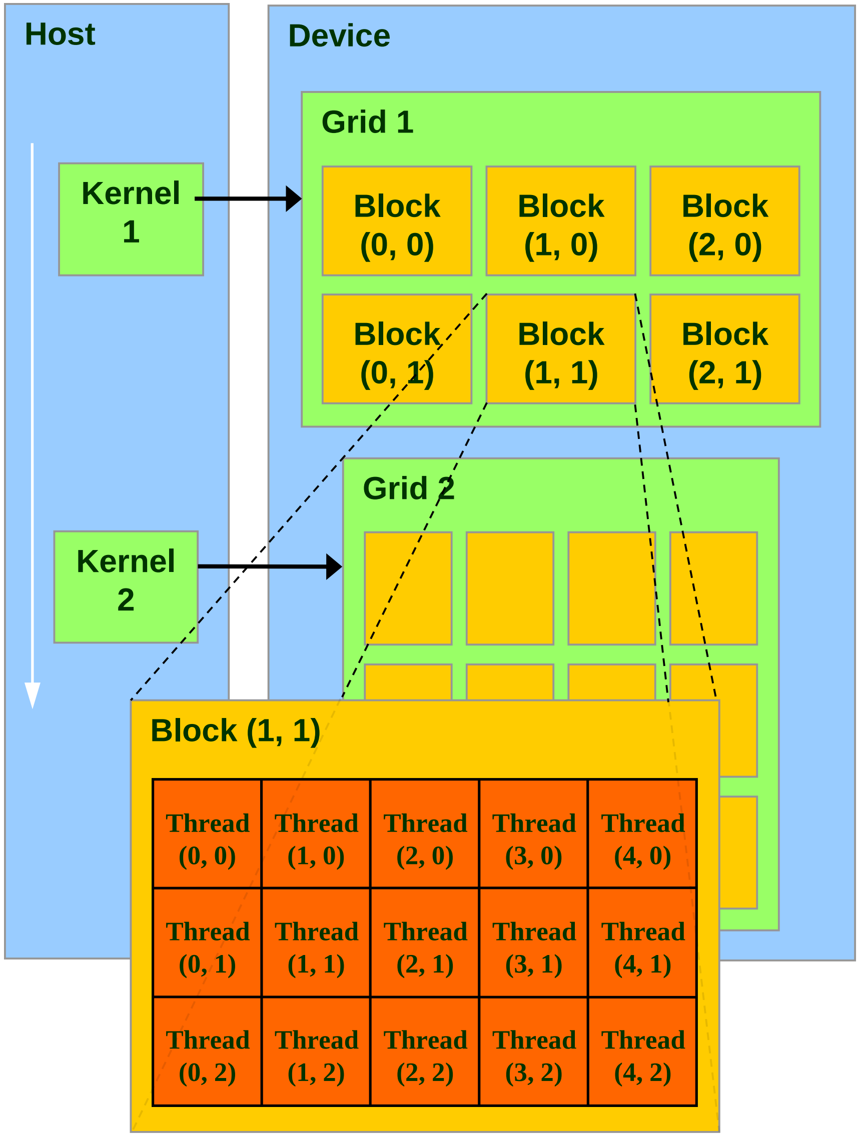
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data



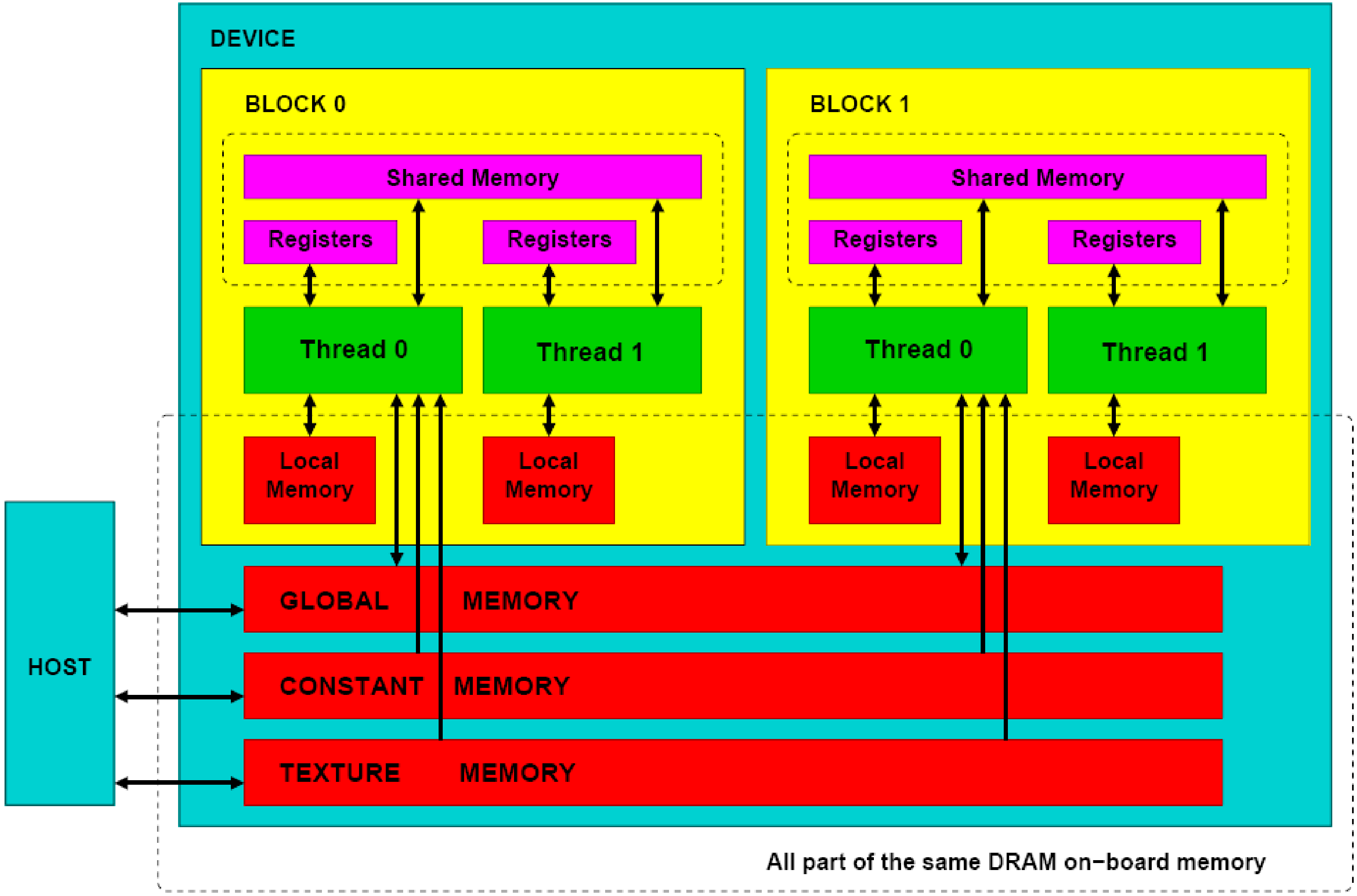
- IDs and dimensions are easily accessible through predefined “variables”, e.g., blockDim.x and threadIdx.x

Thread Batching

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



Programmer's view: Memory Model



Programmer's View: Memory Detail – Thread and Host

- Each thread 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
  - Read only per-grid texture memory

- The host can R/W:
- global, constant, and texture memories

Memory Model: Global, Constant, and Texture Memories

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads

Memory Model Summary

Memory	Location	Cached	Access	Scope
Local	off-chip	No	R/W	thread
Shared	on-chip	N/A	R/W	all threads in a block
Global	off-chip	No	R/W	all threads + host
Constant	off-chip	Yes	RO	all threads + host
Texture	off-chip	Yes	RO	all threads + host

Execution Model: Ordering

- **Execution order is undefined**

- Do not assume and use:
    - block 0 executes before block 1
    - Thread 10 executes before thread 20
    - And **any** other ordering even if you can observe it

Execution Model Summary

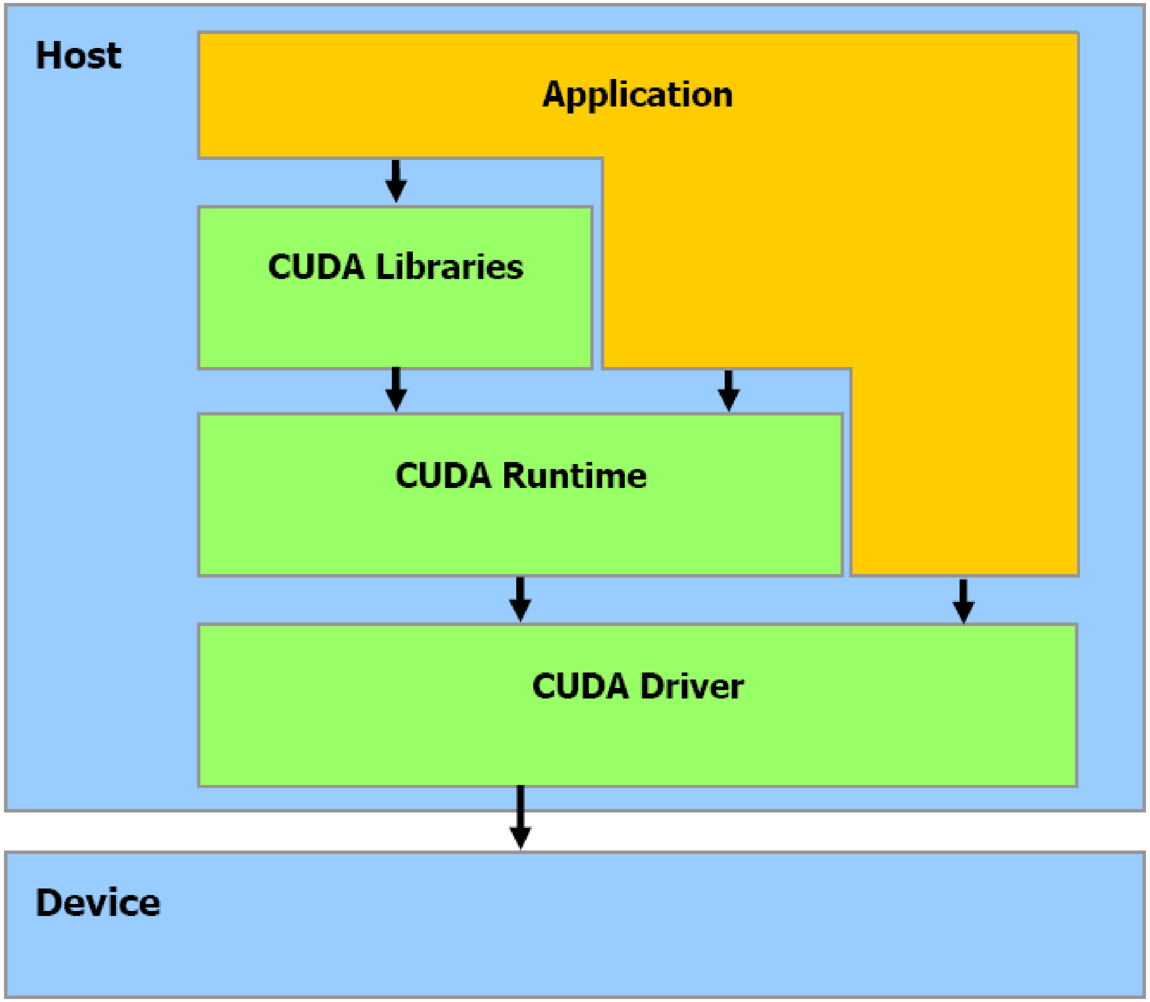
- **Grid of blocks of threads**
  - 1D/2D grid of blocks
  - 1D/2D/3D blocks of threads
- **All blocks are identical:**
  - same structure and # of threads
- **Block execution order is undefined**
- Same block threads:
  - can synchronize and share data fast (shared memory)
- Threads from different blocks:
  - Cannot cooperate
  - Communication through global memory
- **Threads and Blocks have IDs**
  - Simplifies data indexing
  - Can be 1D, 2D, or 3D (threads)
- Blocks do not migrate: execute on the same processor
- Several blocks may run over the same processor

CUDA Software Architecture

e.g., fft()

cuda…()

cu…()



Reasoning about CUDA call ordering

- GPU communication via cuda…() calls and kernel invocations
  - cudaMalloc, cudaMemCpy,
- Asynchronous from the CPU’s perspective
  - CPU places a request in a “CUDA” queue
  - requests are handled in-order
- Streams allow for multiple queues
  - More on this much later one

CUDA API: Example

```
int a[N];
   for (i =0; i < N; i++)
     a[i] = a[i] + x;
```

1. Allocate CPU Data Structure
2. Initialize Data on CPU
3. Allocate GPU Data Structure
4. Copy Data from CPU to GPU
5. Define *Execution Configuration*
6. Run Kernel
7. CPU synchronizes with GPU
8. Copy Data from GPU to CPU
9. De-allocate GPU and CPU memory

1. Allocate CPU Data

```
float *ha;

main (int argc, char *argv[])
{
  int N = atoi (argv[1]);
  ha = (float *) malloc (sizeof (float) * N);

  ...
}
```

- Pinned memory allocation results in faster CPU to/from GPU copies
- More on this later
- `cudaMallocHost (…)`

2. Initialize CPU Data

```
float *ha;

int i;

for (i = 0; i < N; i++)
  ha[i] = i;
```

3. Allocate GPU Data

```
float *da;
```

```
cudaMalloc ((void **) &da, sizeof (float) * N);
```

- Notice: no assignment side
  - NOT: da = cudaMalloc (…)
- Assignment is done internally:
  - That why we pass &da
- Allocated space in **Global Memory**

GPU Memory Allocation

- The host manages GPU memory allocation:
  - **`cudaMalloc (void **ptr, size_t nbytes)`**
  - Must explicitly cast to (`void **`)
    - `cudaMalloc ((void **) &da, sizeof (float) * N);`

  - **`cudaFree (void *ptr);`**
    - `cudaFree (da);`

  - `cudaMemset (void *ptr, int value, size_t nbytes);`
  - `cudaMemset (da, 0, N * sizeof (int));`

- Check the **CUDA Reference Manual**

4. Copy Initialized CPU data to GPU

```
float *da;
float *ha;

cudaMemCpy ((void *) da,                    // DESTINATION
            (void *) ha,                    // SOURCE
            sizeof (float) * N,             // #bytes
            cudaMemcpyHostToDevice); // DIRECTION
```

Host/Device Data Transfers

- The host initiates all transfers:
- **`cudaMemcpy`**`(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction)`

```
cudaMemcpy(void *dst, void *src,
           size_t nbytes,
           enum cudaMemcpyKind direction)
```

- Asynchronous from the CPU's perspective
  - CPU thread continues
- In-order processing with other CUDA requests

- `enum cudaMemcpyKind`
  - cudaMemcpy**Host**To**Device**
  - cudaMemcpy**Device**To**Host**
  - cudaMemcpy**Device**To**Device**

5. Define Execution Configuration

- How many blocks and threads/block

```
int threads_block = 64;
int blocks = N / threads_block;
if (blocks % N != 0) blocks += 1;
```

- Alternatively:

```
blocks = (N + threads_block - 1) /
            threads_block;
```

6. Launch Kernel & 7. CPU/GPU Synchronization

- Instructs the GPU to launch `blocks x thread_blocks` threads:

```
darradd <<<blocks, threads_block>> (da, 10f, N);
cudaThreadSynchronize ();
```

- darradd: kernel name
- <<<…>>> execution configuration
  - More on this soon
- (da, x, N): arguments
  - 256 – 8 byte limit / No variable arguments

CPU/GPU Synchronization

- CPU does not block on cuda…() calls
  - Kernel/requests are queued and processed in-order
  - Control returns to CPU immediately

- Good if there is other work to be done
  - e.g., preparing for the next kernel invocation

- Eventually, CPU must know when GPU is done
- Then it can safely copy the GPU results

- `cudaThreadSynchronize ()`
  - Block CPU until **all** preceding cuda…() and kernel requests have completed

8. Copy data from GPU to CPU & 9. DeAllocate Memory

```
float *da;
float *ha;

cudaMemCpy ((void *) ha,                    // DESTINATION
            (void *) da,                    // SOURCE
            sizeof (float) * N,             // #bytes
            cudaMemcpyDeviceToHost); // DIRECTION

cudaFree (da);
// display or process results here
free (ha);
```

The GPU Kernel

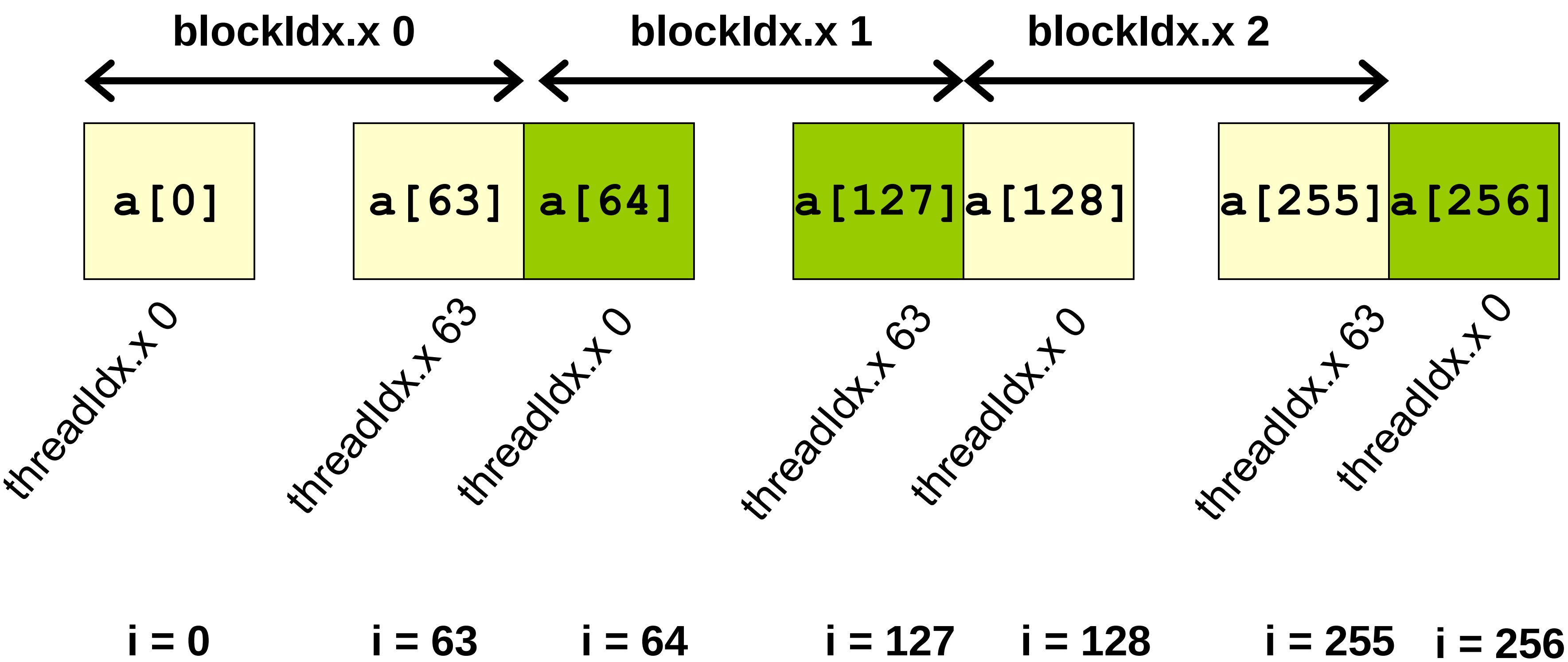
```
__global__ darradd (float *da, float x, int N)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;

   if (i < N) da[i] = da[i] + x;
}
```

- **BlockIdx:** Unique Block ID.
  - Numerically asceding: 0, 1, …
- **ThreadIdx:** Unique per Block Index
  - 0, 1, …
  - Per Block
- **BlockDim:** Dimensions of Block
  - BlockDim.x, BlockDim.y, BlockDim.z
  - Unused dimensions default to 0

Array Index Calculation Example

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



Assuming blockDim.x = 64