King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming
Lecture 12: GPU Compute APIs, Pt. 2

Markus Hadwiger, KAUST

Reading Assignment #6 (until Oct 13)

Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 3** (*Multidimensional grids and data*)

Read (optional):

- Programming Massively Parallel Processors book (4th edition), **Chapter 20** (*An introduction to CUDA streams*)
- Programming Massively Parallel Processors book (4th edition), **Chapter 21** (*CUDA dynamic parallelism*)

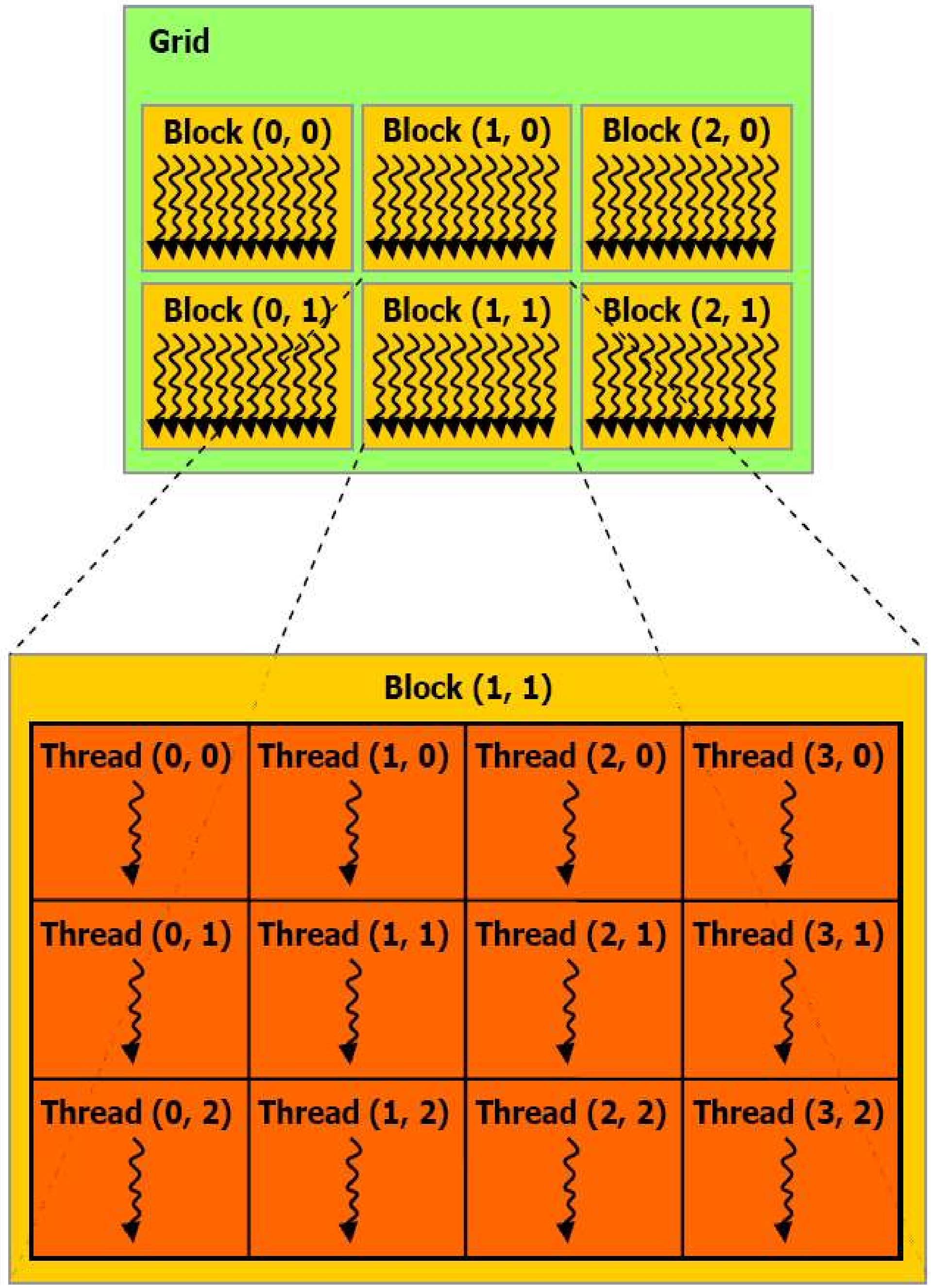
GPU Compute APIs

CUDA Multi-Threading

CUDA model groups threads into **thread blocks**; blocks into **grid**

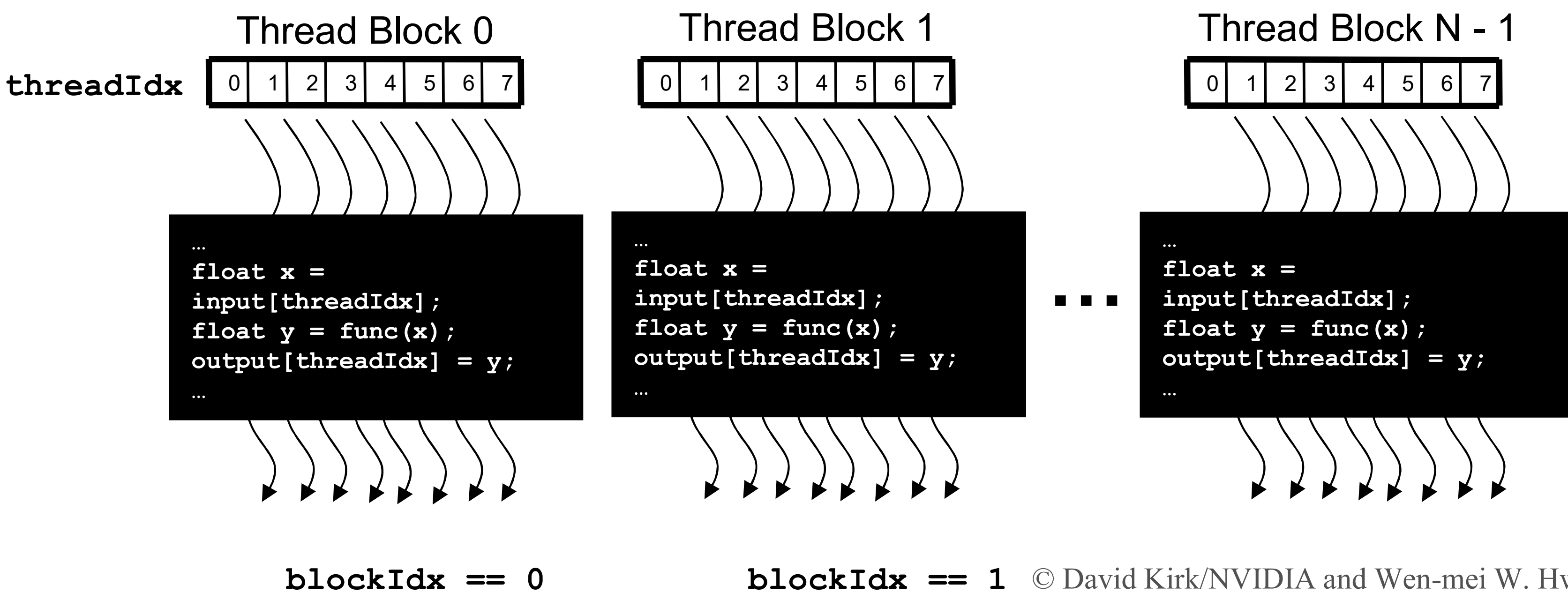
Execution on actual hardware:

- Thread blocks assigned to SM (up to 8, 16, or 32 blocks per SM; depending on compute capability)
- 32 threads grouped into a **warp** (on all compute capabilities)



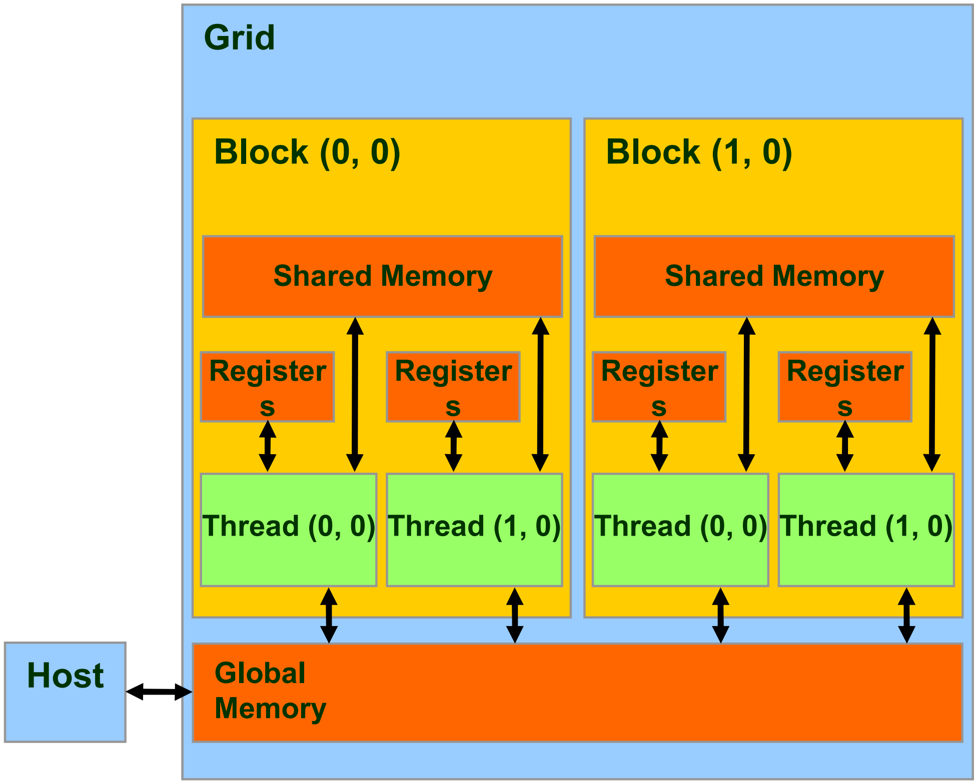
Threads in Block, Blocks in Grid

- Identify work of thread via
  - `threadIdx`
  - `blockIdx`



CUDA Memory Model and Usage

- `cudaMalloc()`, `cudaFree()`
- `cudaMallocArray()`, `cudaMalloc2DArray()`, `cudaMalloc3DArray()`
- `cudaMemcpy()`
- `cudaMemcpyArray()`
- Host ↔ host
  Host ↔ device
  Device ↔ device
- Asynchronous transfers possible (DMA)



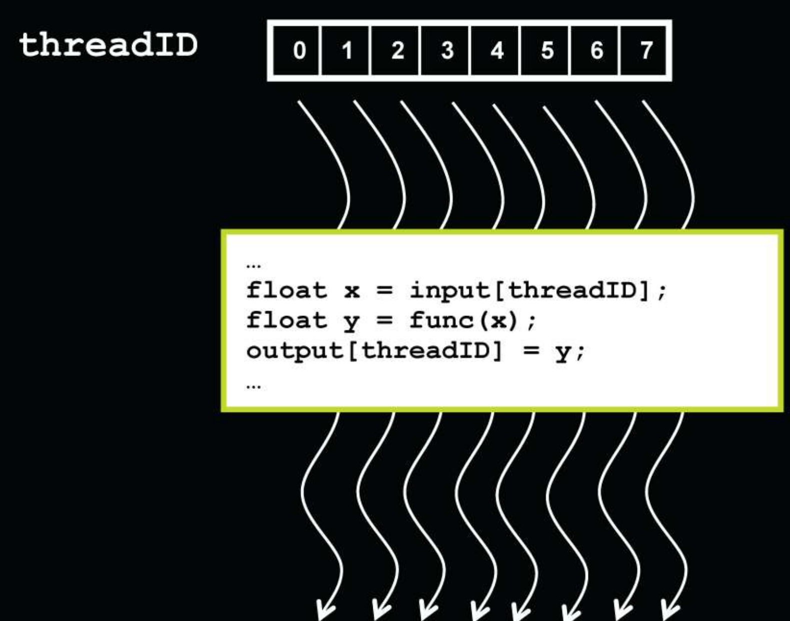
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as **kernels**
  - One **kernel** is executed at a time
  - Many threads execute each **kernel**
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions
Device = GPU
Host = CPU
Kernel = function that runs on the device

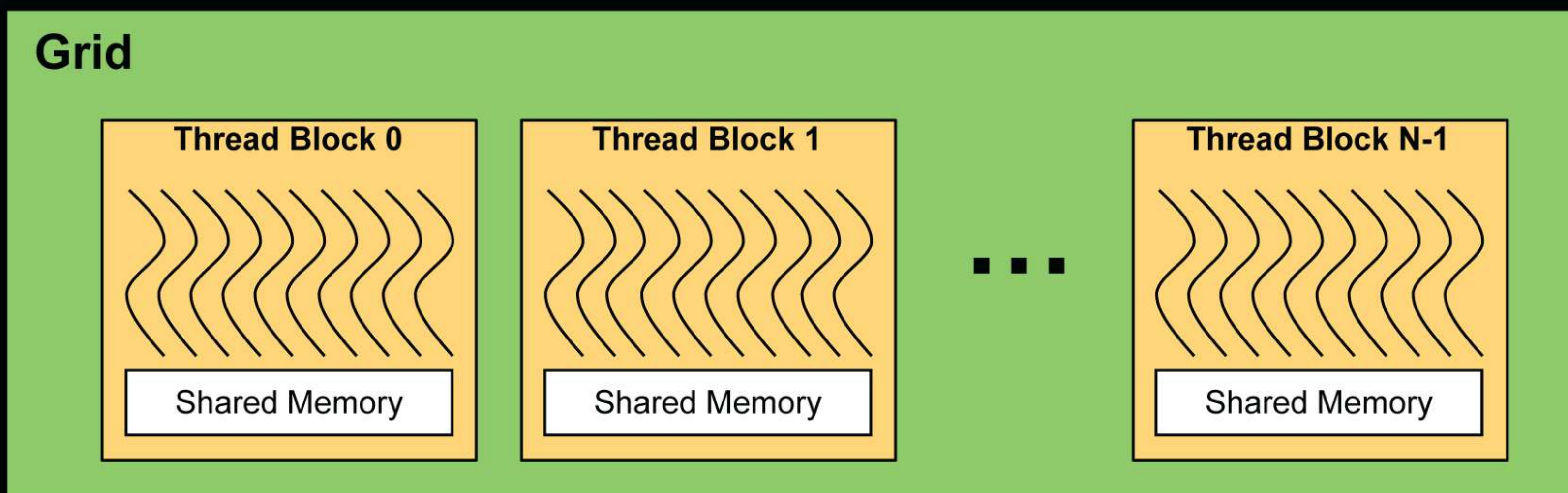
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



Thread Batching

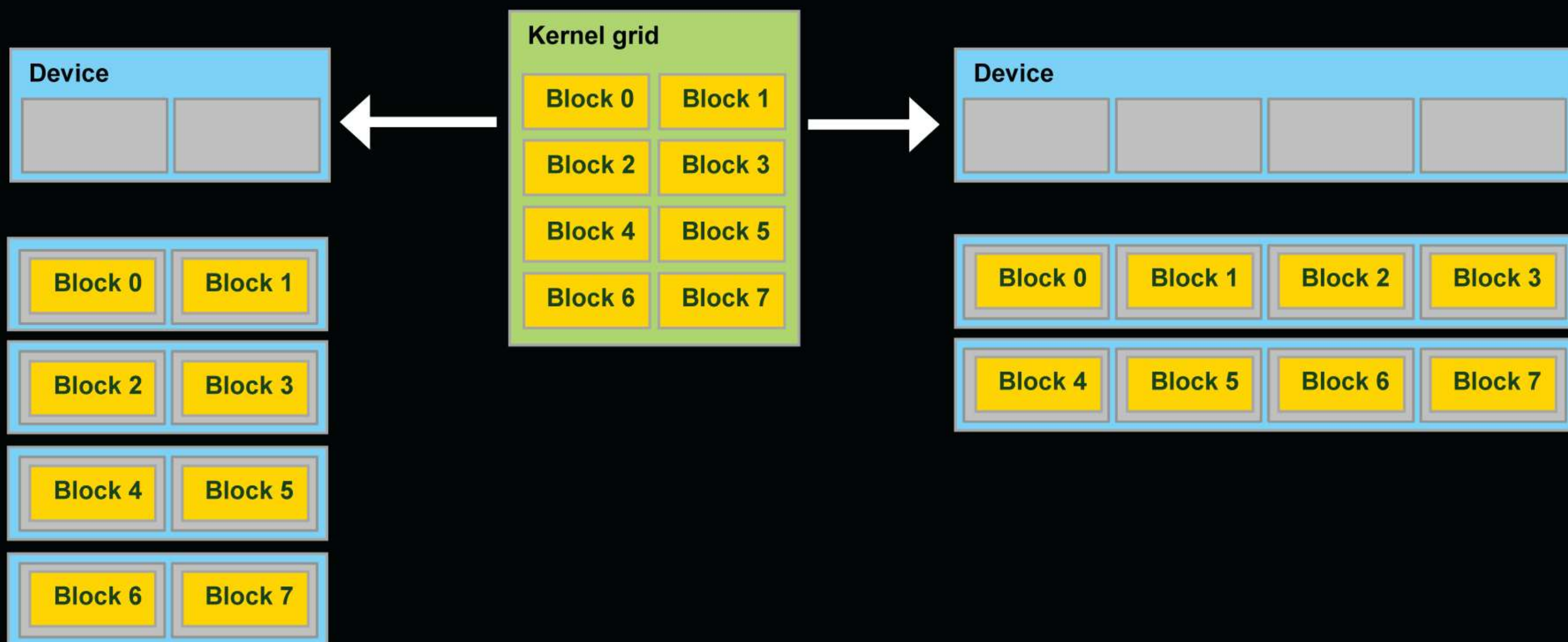
- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate*
- Allows programs to *transparently scale* to different GPUs



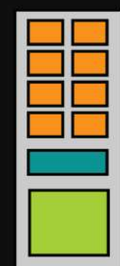
* brand new on Hopper: thread block clusters

Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - A kernel scales across parallel multiprocessors

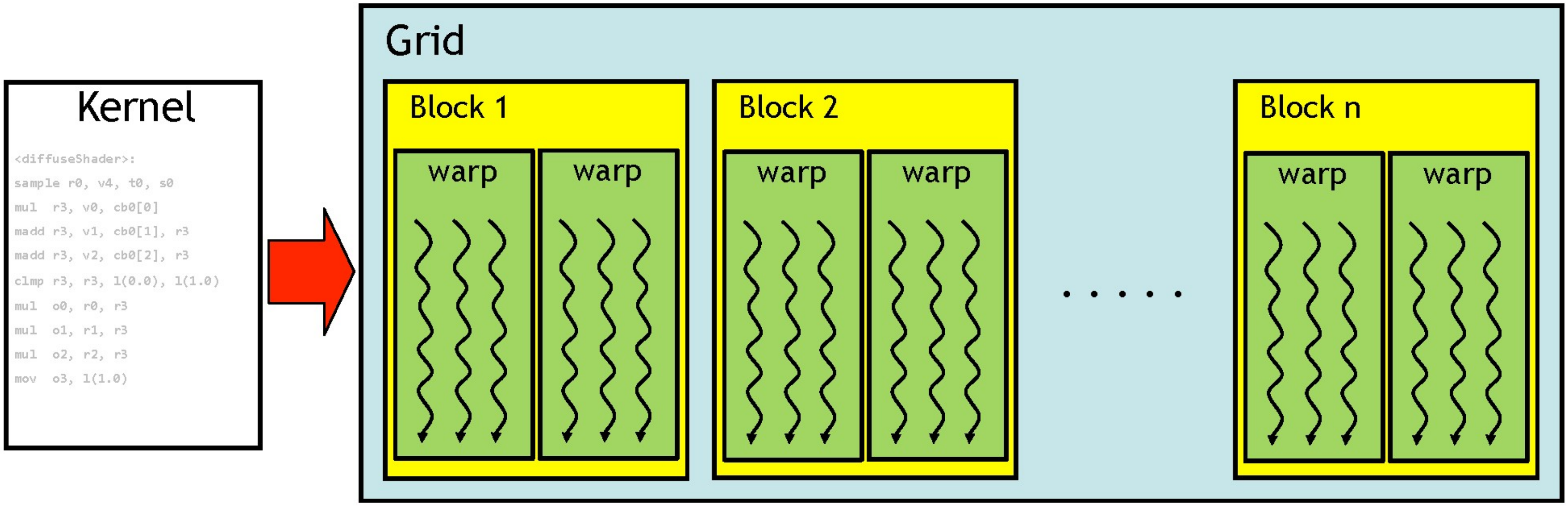


Execution Model

Software	Hardware	
 Thread	 Thread Processor	Threads are executed by thread processors
 Thread Block	 Multiprocessor	Thread blocks are executed on multiprocessors Thread blocks do not migrate Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
 Grid	 Device	A kernel is launched as a grid of thread blocks Only one kernel can execute on a device at one time

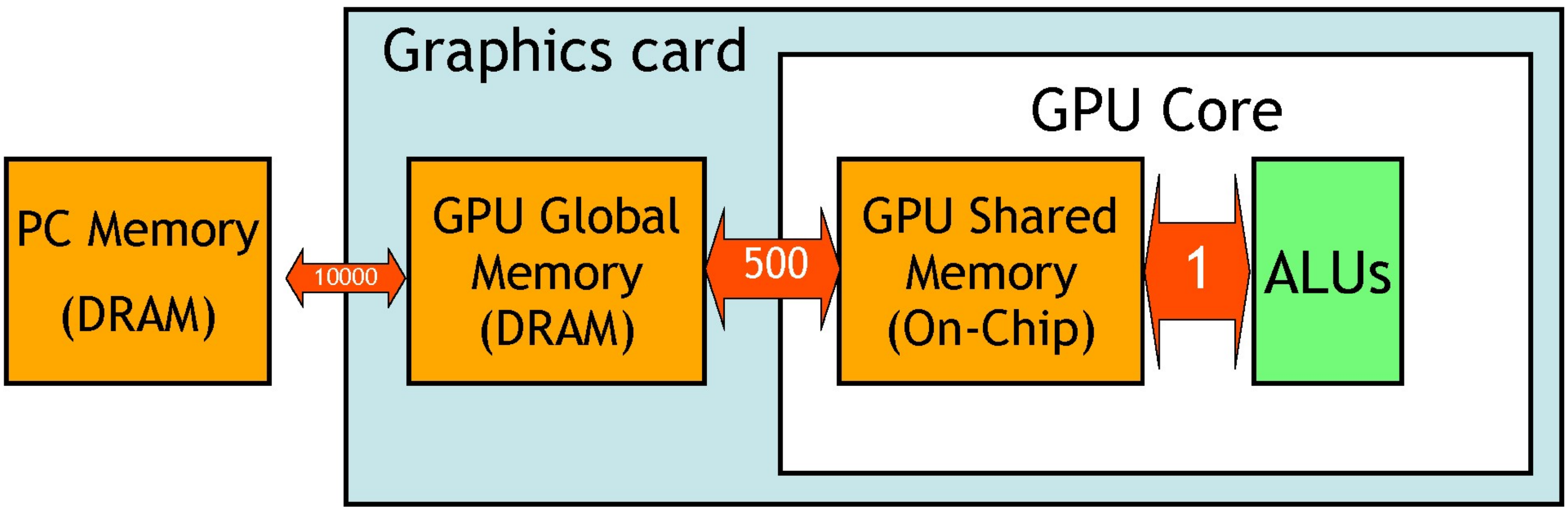
CUDA Programming Model

- Kernel
  - GPU program that runs on a thread grid
- Thread hierarchy
  - Grid : a set of blocks
  - Block : a set of warps
  - Warp : a SIMD group of 32 threads
  - Grid size * block size = total # of threads



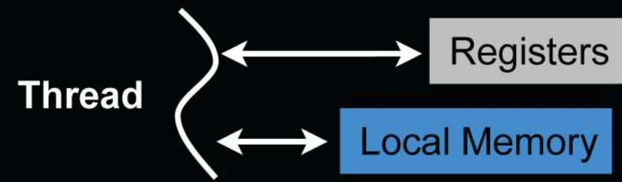
CUDA Memory Structure

- Memory hierarchy
  - PC memory : off-card
  - GPU global : off-chip / on-card
  - GPU shared/register/cache : on-chip
- The host can read/write global memory
- Each thread communicates using shared memory



Kernel Memory Access

- **Per-thread**

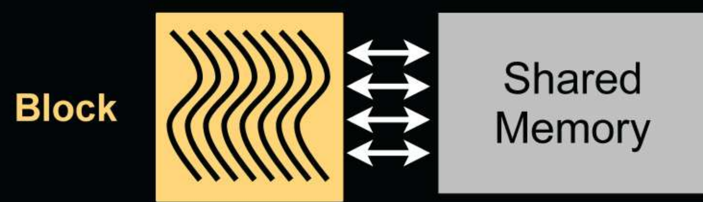


On-chip

Off-chip, ~~uncached~~

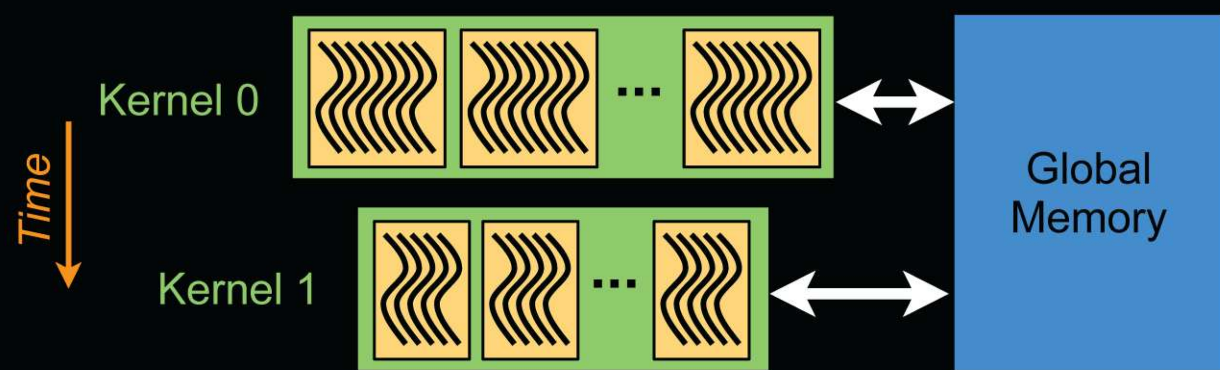
now cached in L1!
(Fermi or newer)

- **Per-block**



- On-chip, small
- Fast

- **Per-device**



- Off-chip, large
- ~~Uncached~~
- Persistent across kernel launches
- Kernel I/O

now cached in L2!
(Fermi or newer)

Memory Architecture

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	~~No~~* YES	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	~~No~~* YES	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

* cached on Fermi or newer!

PTX (Memory) State Spaces

PTX ISA 9.0 (Chapter 5)

Name	Addressable	Initializable	Access	Sharing
`.reg`	No	No	R/W	per-thread
`.sreg`	No	No	RO	per-CTA
`.const`	Yes	Yes[1]	RO	per-grid
`.global`	Yes	Yes[1]	R/W	Context
`.local`	Yes	No	R/W	per-thread
`.param` (as input to kernel)	Yes[2]	No	RO	per-grid
`.param` (used in functions)	Restricted[3]	No	R/W	per-thread
`.shared`	Yes	No	R/W	per-cluster[5]
`.tex`	No[4]	Yes, via driver	RO	Context

Notes:

[1] Variables in `.const` and `.global` state spaces are initialized to zero by default.

[2] Accessible only via the `ld.param` instruction. Address may be taken via `mov` instruction.

[3] Accessible via `ld.param` and `st.param` instructions. Device function input and return parameters may have their address taken via `mov`; the parameter is then located on the stack frame and its address is in the `.local` state space.

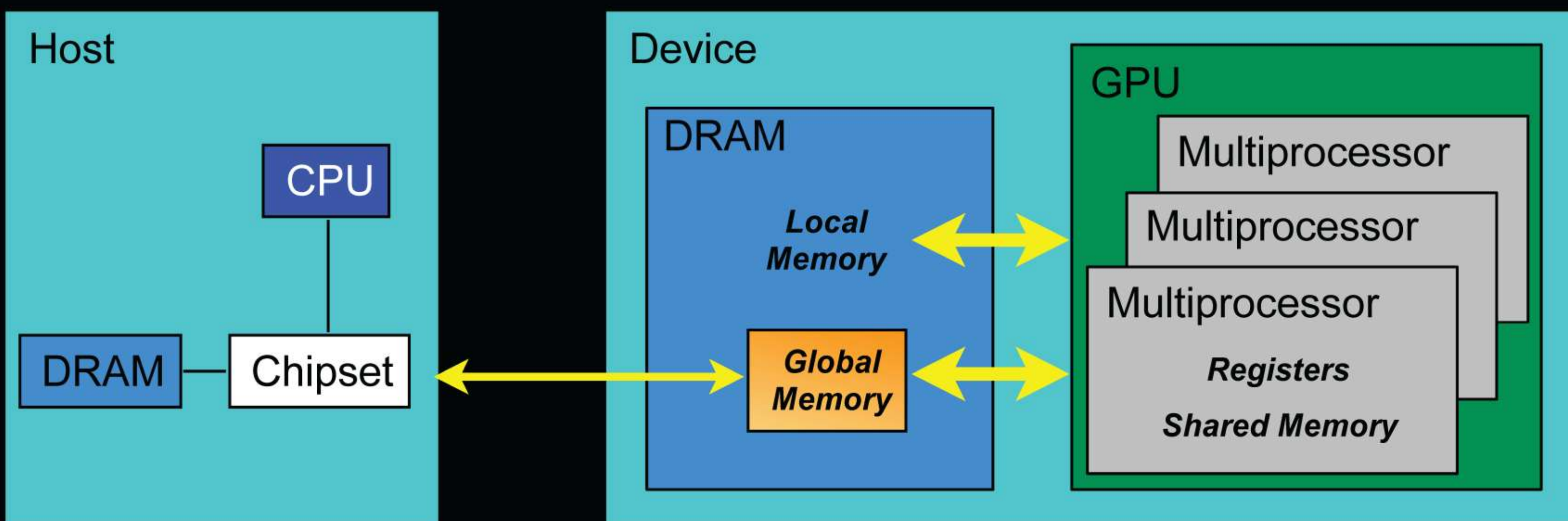
[4] Accessible only via the `tex` instruction.

[5] Visible to the owning CTA and other active CTAs in the cluster.

Managing Memory

Unified memory space can be enabled on Fermi / CUDA 4.x and newer

- **CPU and GPU have separate memory spaces**
- **Host (CPU) code manages device (GPU) memory:**
  - **Allocate / free**
  - **Copy data to and from device**
  - **Applies to *global* device memory (DRAM)**



GPU Memory Allocation / Release

- cudaMalloc(void ** pointer, size_t nbytes)
- cudaMemset(void * pointer, int value, size_t count)
- cudaFree(void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d,  nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

Data Copies

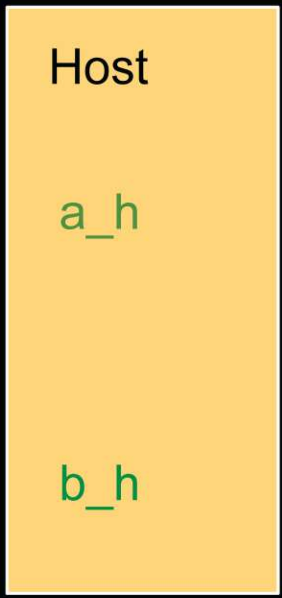
- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - `direction` specifies locations (host or device) of `src` and `dst`
  - Blocks CPU thread: returns after the copy is complete
  - Doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```

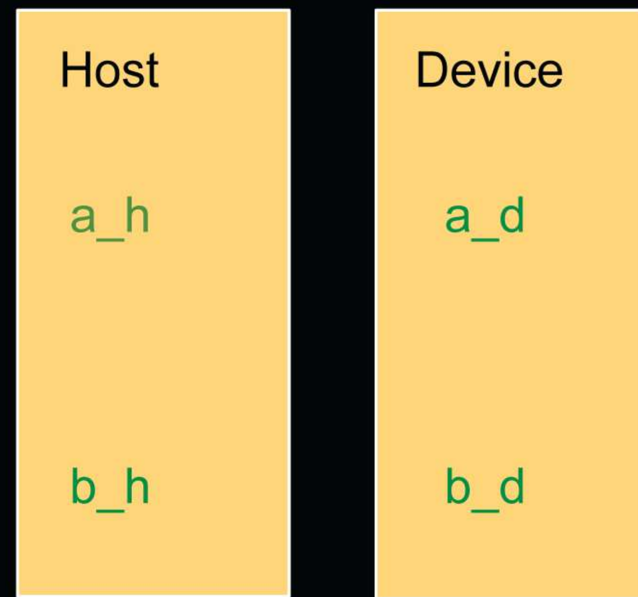
Data Movement Example

```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```



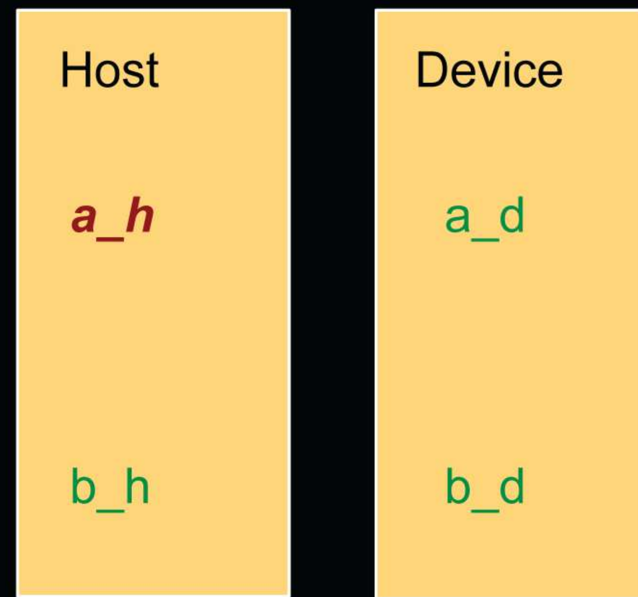
Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



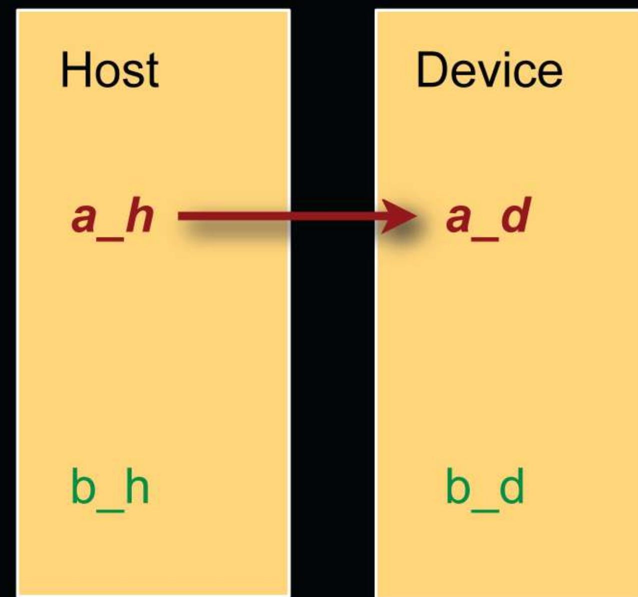
Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



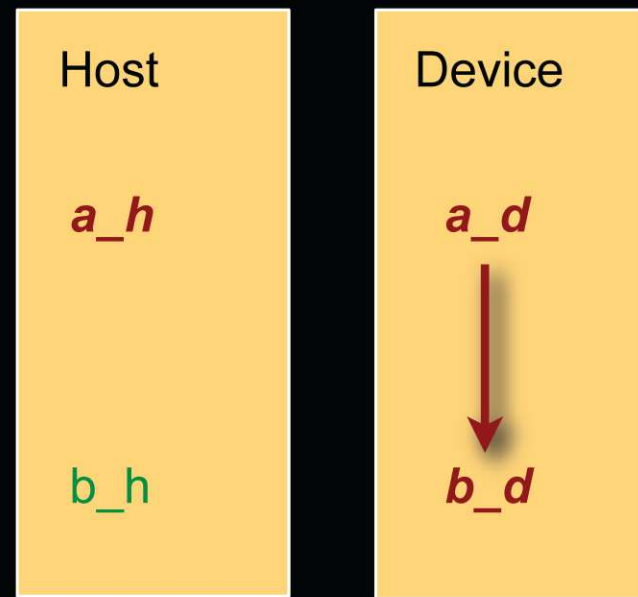
Data Movement Example

```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```



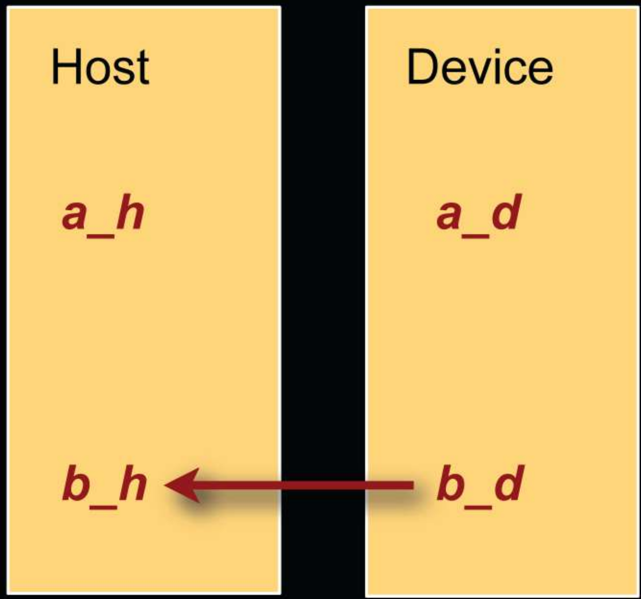
Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```

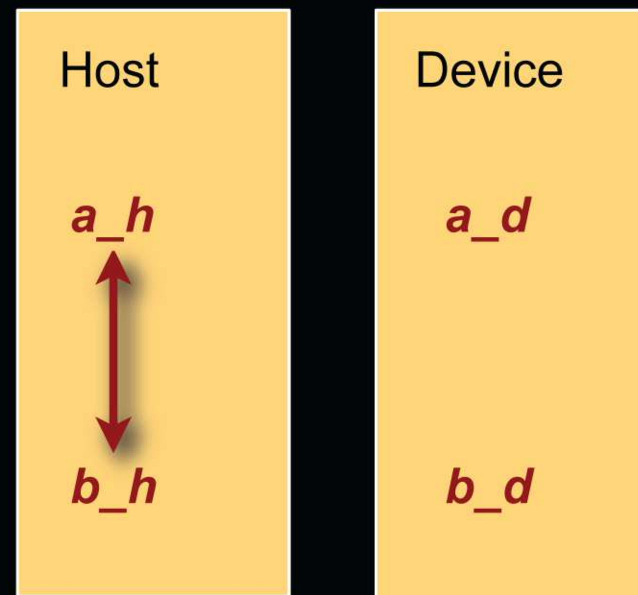


NVISION 08
THE WORLD OF VISUAL COMPUTING

NVIDIA

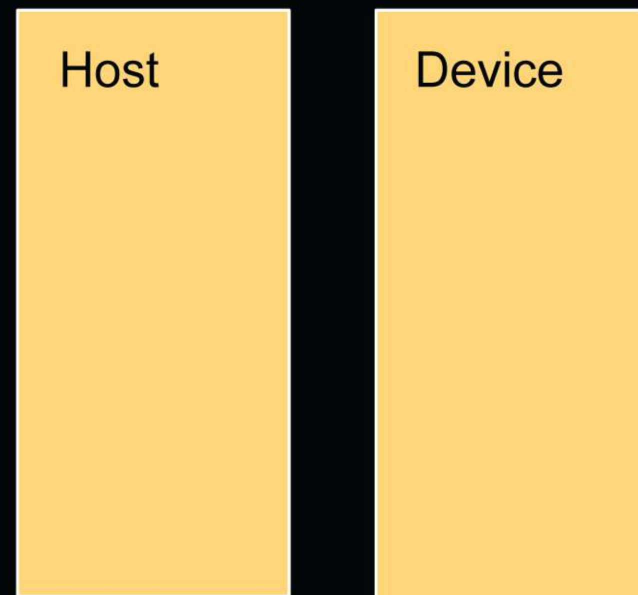
Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Cannot access host memory — except: (*) and (**)
  - Must have void return type
  - No variable number of arguments ("varargs")
  - (Not recursive) — recursion supported on __device__ functions from cc. 2.x (i.e., basically on *all* current GPUs)
  - No static variables
- Function arguments automatically copied from host to device

(*) "unified memory programming" introduced with CUDA 6 (cc. 3.x +): allocate memory with cudaMallocManaged(); uses automatic migration

(**) also: mapped pinned (page-locked) memory ("zero-copy memory") : allocate memory with cudaMallocHost(); beware of low performance!!

Note: UVA ("unified virtual addressing"; cc. 2.x +) is something different!! just pertains to unified pointers (see cudaPointerGetAttributes(), …)

Launching Kernels

- **Modified C function call syntax:**

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

- **Execution Configuration ("<<< >>>")**
  - **dG - dimension and size of grid in blocks**
    - **Two-dimensional: x and y**
    - **Blocks launched in the grid: dG.x * dG.y**
  - **dB - dimension and size of blocks in threads:**
    - **Three-dimensional: x, y, and z**
    - **Threads per block: dB.x * dB.y * dB.z**
  - **Unspecified dim3 fields initialize to 1**

Shared Memory Allocation

- 2 modes
- Static size within kernel

```
__shared__ float vec[256];
```

- Dynamic size when calling the kernel

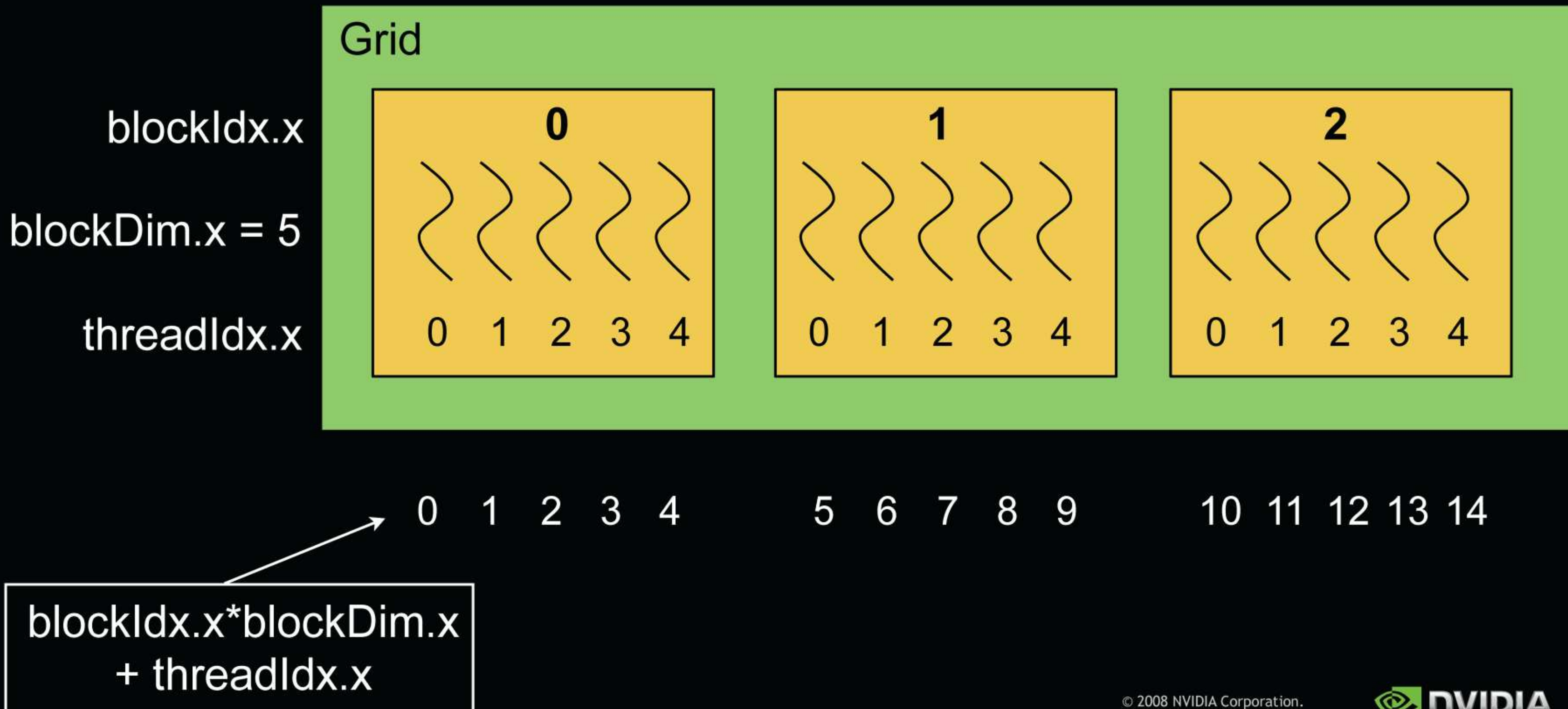
```
// in main
int VecSize = MAX_THREADS * sizeof(float4);
vecMat<<< blockGrid, threadBlock, VecSize >>>( p1, p2, …);

// declare as extern within kernel
extern __shared__ float vec[];
```

CUDA Built-in Device Variables

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

Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (threadIdx) to a global ID which can be used as array indices



Increment Array Example

CPU program

```
void inc_cpu(int *a, int N)
{
  int idx;

  for (idx = 0; idx<N; idx++)
     a[idx] = a[idx] + 1;
}

int main()
{
   ...
   inc_cpu(a, N);
}
```

CUDA program

```
__global__ void inc_gpu(int *a, int N)
{
 int idx = blockIdx.x * blockDim.x
              + threadIdx.x;
 if (idx < N)
    a[idx] = a[idx] + 1;
}

int main()
{
 ...
 dim3 dimBlock (blocksize);
 dim3 dimGrid( ceil( N / (float)blocksize)  );
 inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```

Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared
- Allowed in conditional code only if the conditional is uniform across the entire thread block

Synchronization

- Threads in the same block can communicate using shared memory

- __syncthreads()

now also _syncwarp()

  - Barrier for threads only within the current block
- __threadfence()
  - Flushes global memory writes to make them visible to all threads

Plus newer sync functions, e.g., from compute capability 2.x on:
__syncthreads_count(), __syncthreads_and/or(),
__threadfence_block(), __threadfence_system(), …

Now: *Must* use versions with _sync suffix, because of
Independent Thread Scheduling (compute capability 7.x and newer)!

COOPERATIVE GROUPS

Kyrylo Perelygin, Yuan Lin

GTC 2017

NVIDIA

LEVELS OF COOPERATION: CUDA 9.0 or newer

For current coalesced set of threads:

```
  auto g = coalesced_threads();
```

For warp-sized group of threads:

```
 auto block = this_thread_block();
 auto g = tiled_partition<32>(block)
```

For CUDA thread blocks:

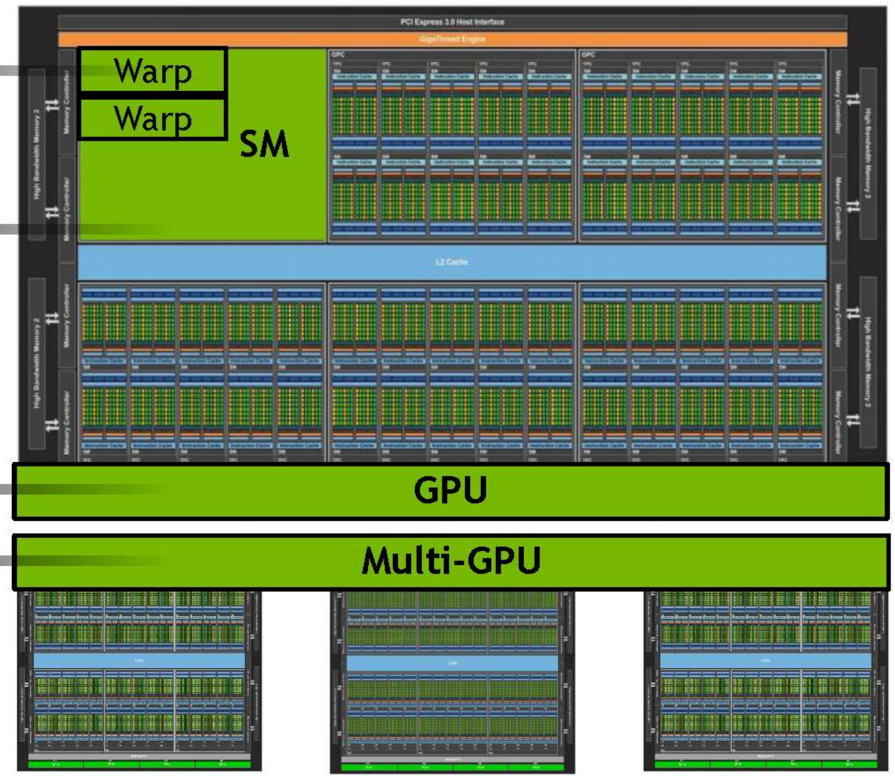
```
  auto g = this_thread_block();
```

For device-spanning grid:

```
  auto g = this_grid();
```

For multiple grids spanning GPUs:

```
  auto g = this_multi_grid();
```



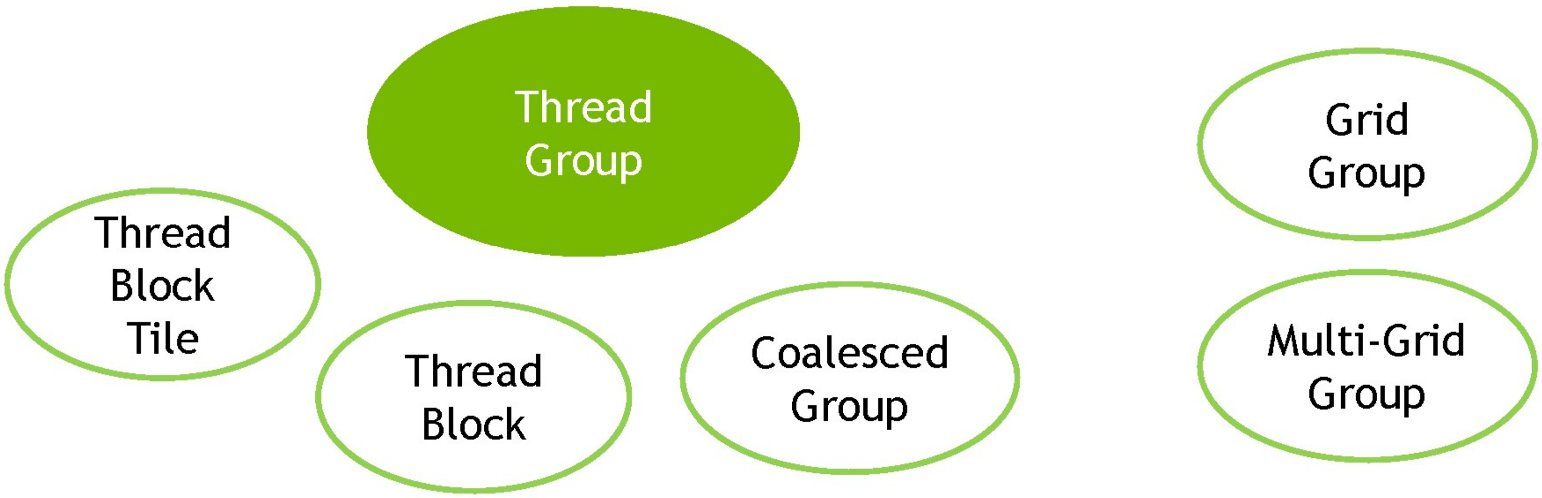
All Cooperative Groups functionality is within a **cooperative_groups::** namespace

THREAD GROUP

Base type, the implementation depends on its construction.

Unifies the various group types into one general, collective, thread group.

We need to extend the CUDA programming model with handles that can represent the groups of threads that can communicate/synchronize



THREAD BLOCK

Implicit group of all the threads in the launched thread block

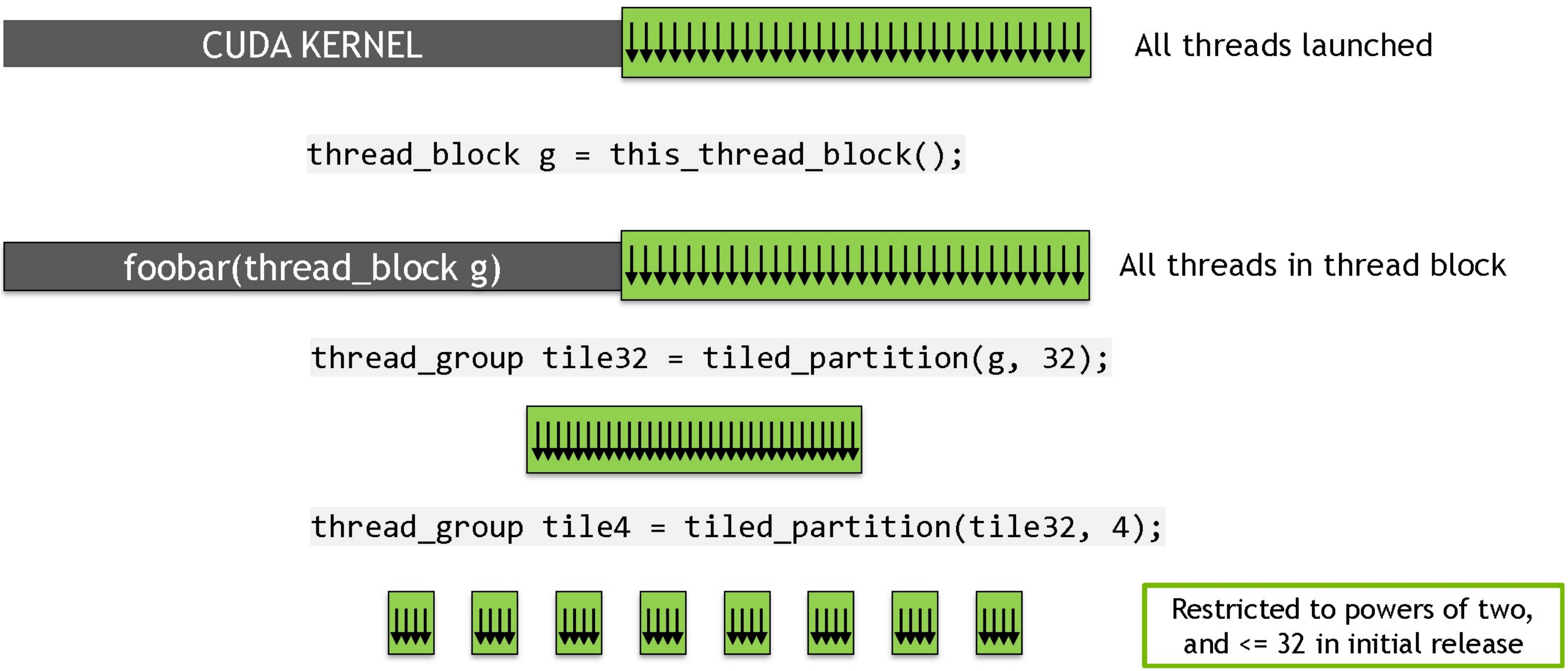
Implements the same interface as thread_group:

```
void sync();                // Synchronize the threads in the group
unsigned size();            // Total number of threads in the group
unsigned thread_rank();     // Rank of the calling thread within [0, size]
bool is_valid();            // Whether the group violated any API constraints
```

And additional thread_block specific functions:

```
dim3 group_index();         // 3-dimensional block index within the grid
dim3 thread_index();        // 3-dimensional thread index within the block
```

PROGRAM DEFINED DECOMPOSITION



GENERIC PARALLEL ALGORITHMS

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition(this_thread_block(), 32);
reduce(g, ptr, myVal);
```

```
__device__ int reduce(thread_group g, int *x, int val) {
  int lane = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    x[lane] = val;       g.sync();
    val += x[lane + i];  g.sync();
  }
  return val;
}
```

COOPERATIVE GROUPS VS BUILT-IN FUNCTIONS

Example: warp aggregated atomic

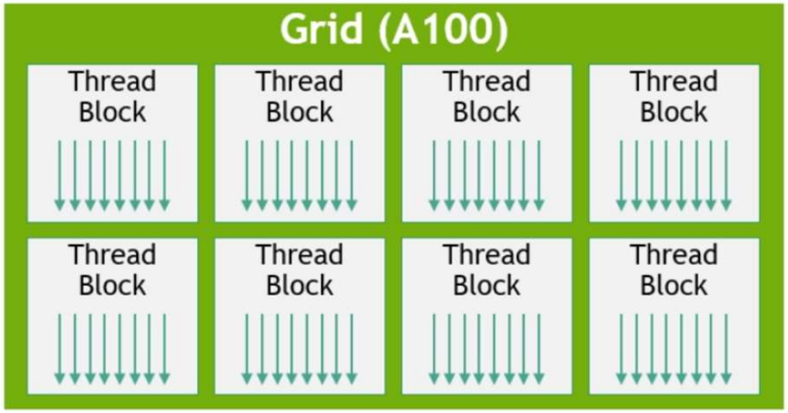
```
// increment the value at ptr by 1 and return the old value
__device__ int atomicAggInc(int *p);
```

```
coalesced_group g = coalesced_threads();



int res;
if (g.thread_rank() == 0)
   res = atomicAdd(p, g.size());
res = g.shfl(res, 0);
return g.thread_rank() + res;
```

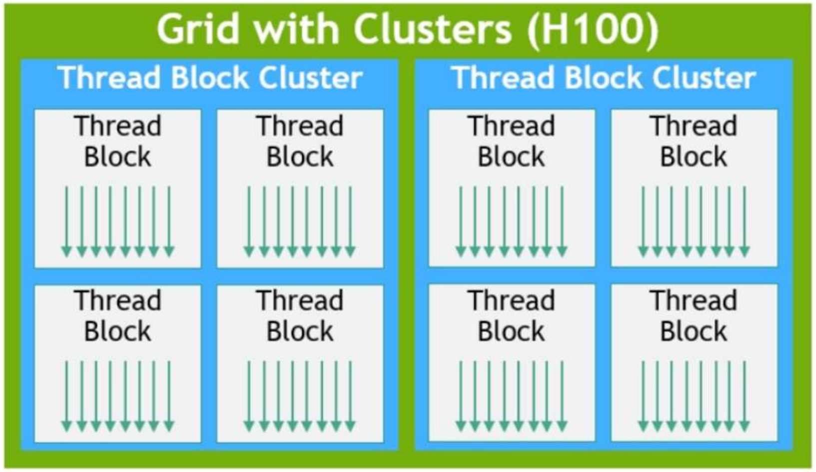
```
int mask = __activemask();
int rank = __popc(mask & __lanemask_lt());
int leader_lane = __ffs(mask) - 1;
int res;
if (rank == 0)
   res = atomicAdd(p, __popc(mask));
res = __shfl_sync(mask, res, leader_lane);
return rank + res;
```

New in CC 9.0: Thread Block Clusters

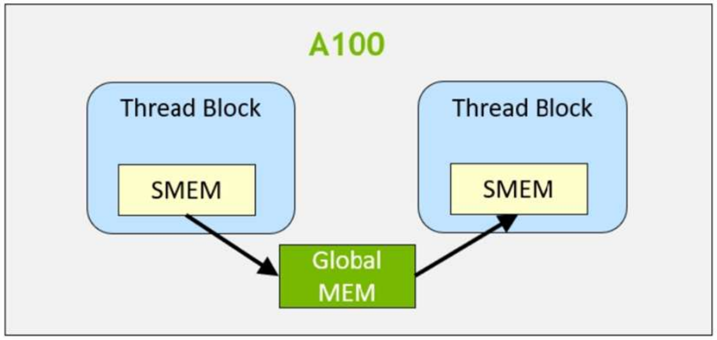
New thread hierarchy level!

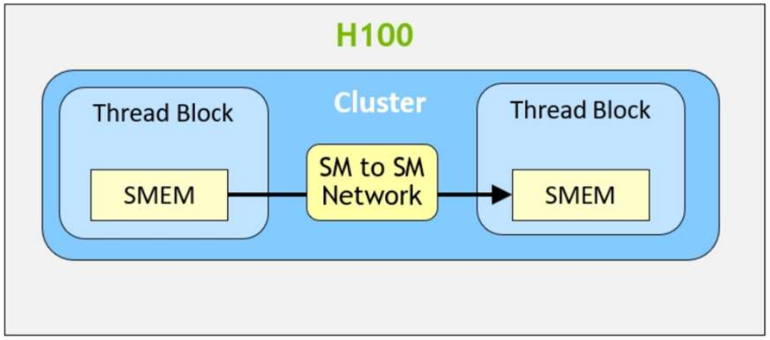


*all threads of a block are on the same **SM** !*



*all blocks of a cluster are on the same **GPC** !*

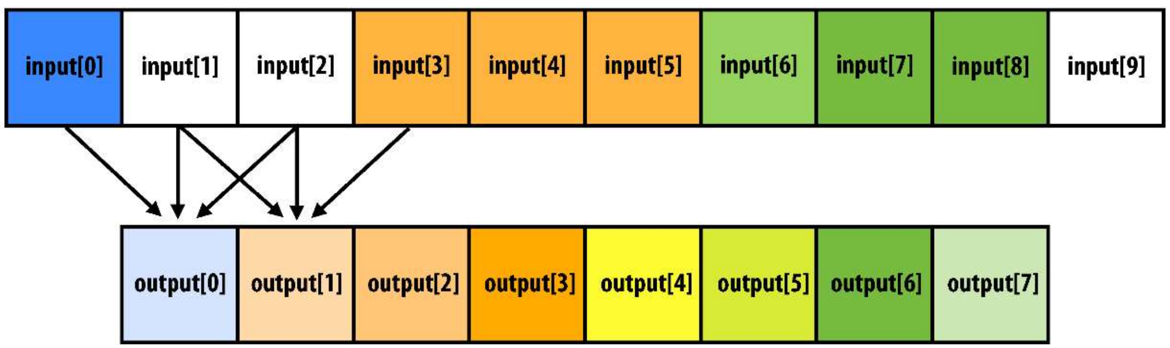




Code Example #1: 1D Convolution

Example #1: 1D Convolution

1D Convolution with 3-tap averaging kernel
(every thread is averaging three inputs)

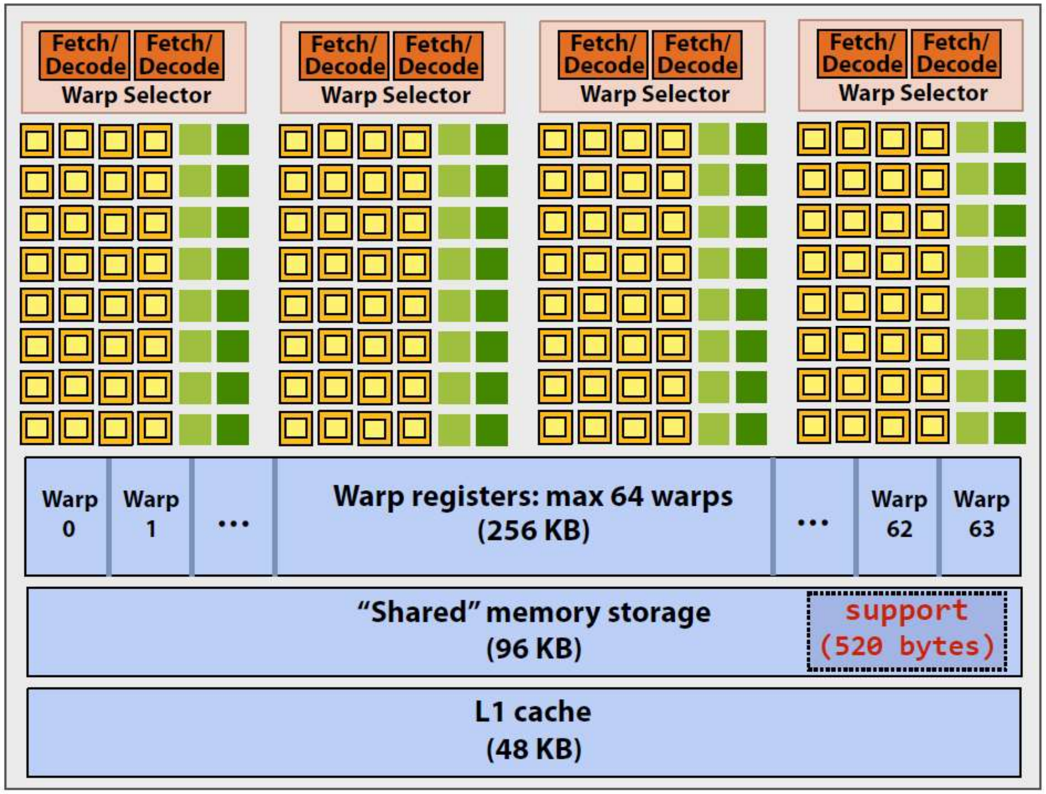



```
output[i] = (input[i] + input[i+1] + input[i+2]) / 3.f;
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```


courtesy Kayvon Fatahalian

Running on a GP104 (Pascal) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

Recall, CUDA kernels execute as SPMD programs

On NVIDIA GPUs groups of 32 CUDA threads share an instruction stream. These groups called "warps".

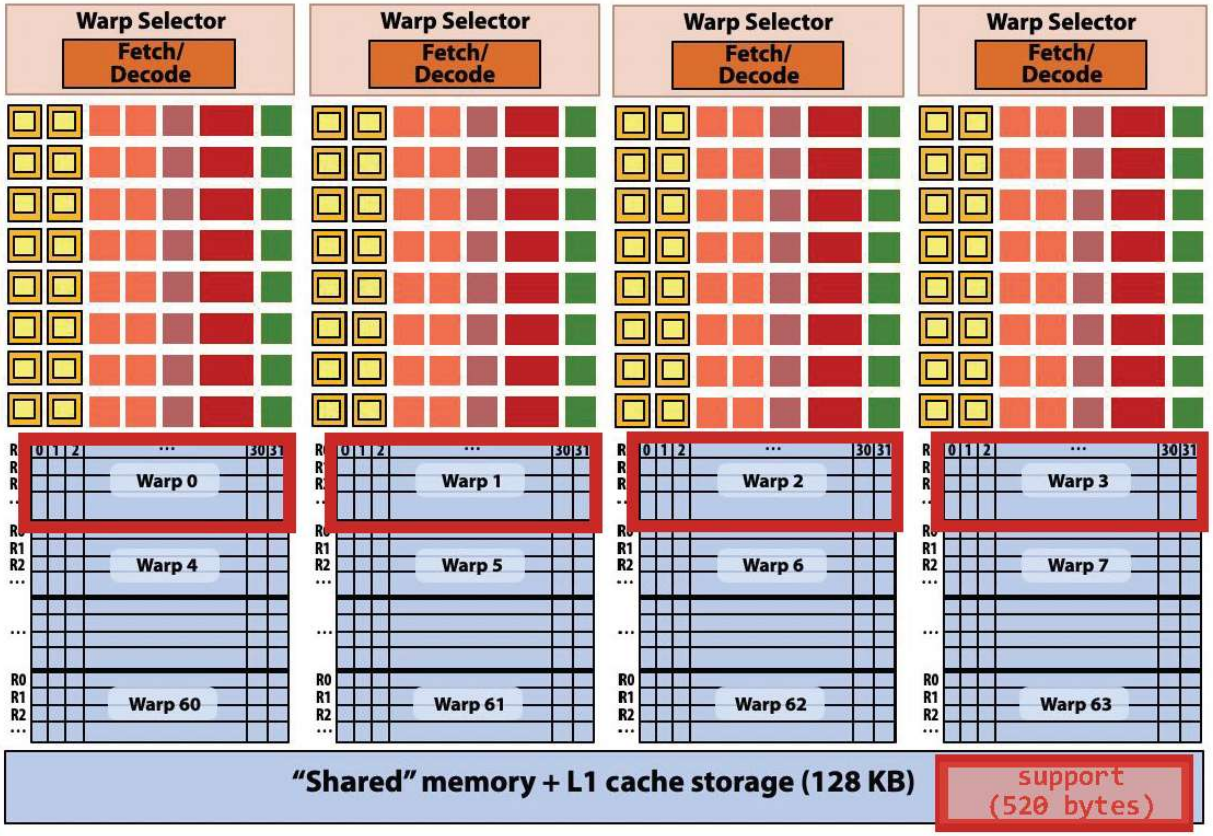
A `convolve` thread block is executed by 4 warps (4 warps x 32 threads/warp = 128 CUDA threads per block)

(Warps are an important GPU implementation detail, but not a CUDA abstraction!)

SM core operation each clock:

- Select up to four runnable warps from 64 resident on SM core (thread-level parallelism)
- Select up to two runnable instructions per warp (instruction-level parallelism) * *(but no ALU dual-issue!)*

Running on a V100 (Volta) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

A `convolve` thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

(sub-core == SM partition)

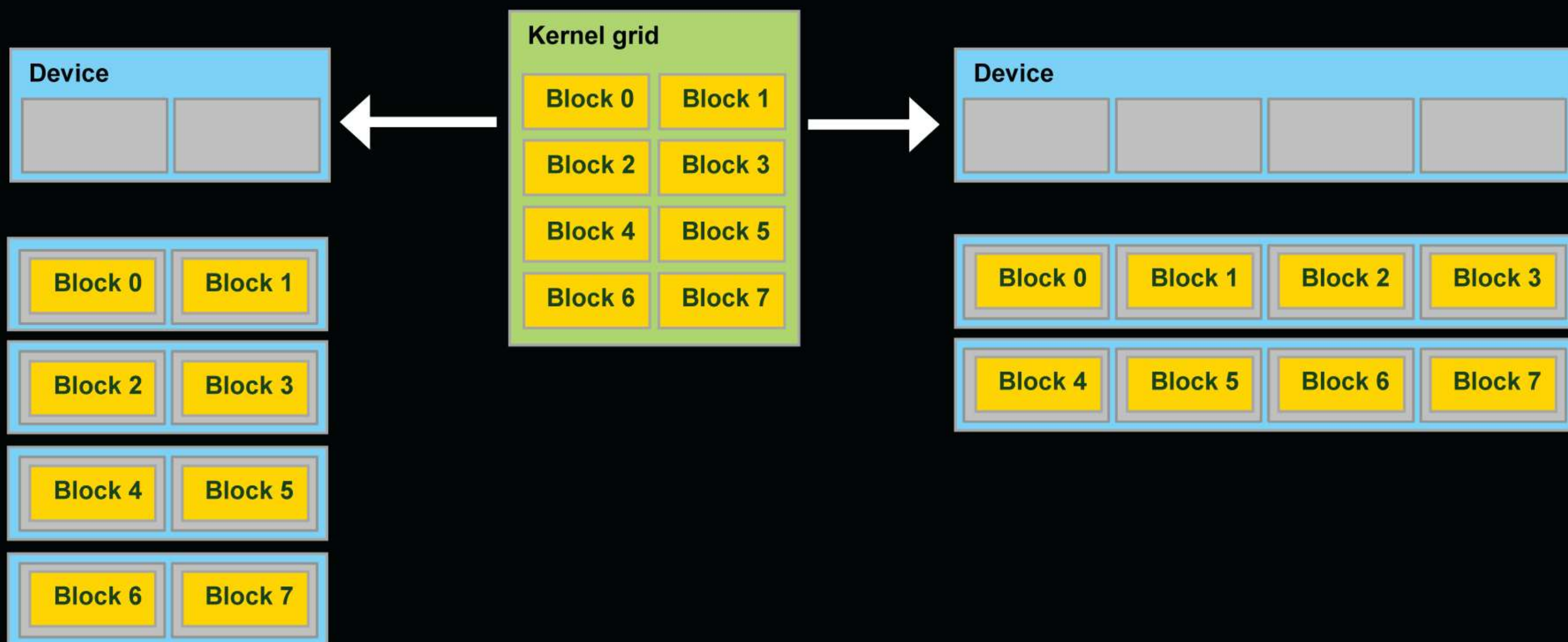
SM core operation each clock:

- **Each sub-core selects one runnable warp (from the 16 warps in its partition)**
- **Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)**

courtesy Kayvon Fatahalian

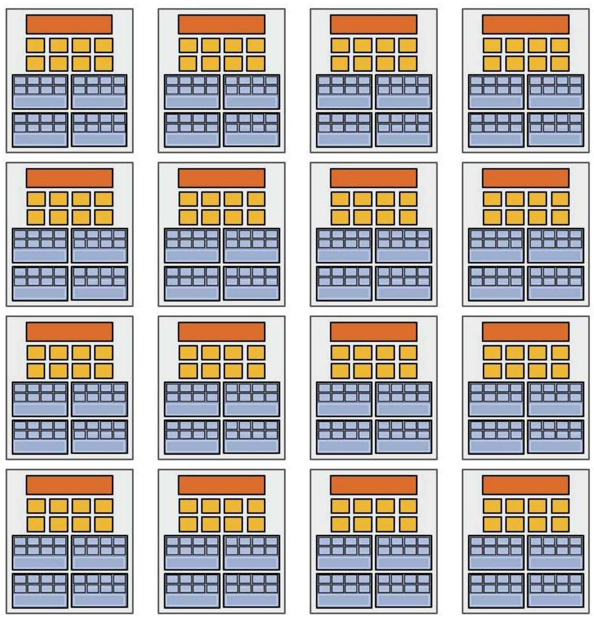
Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - A kernel scales across parallel multiprocessors

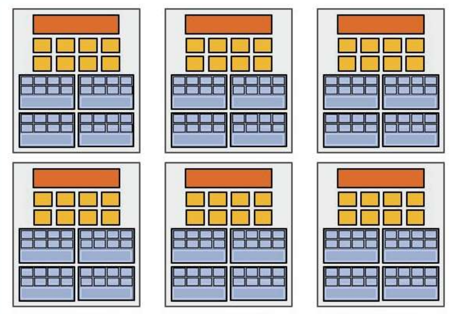


Code on Same SM Arch. But Different #SMs

Assigning work



High-end GPU
(16 cores)



Mid-range GPU
(6 cores)

Desirable for CUDA program to run on all of these GPUs without modification

Note: there is no concept of `num_cores` in the CUDA programs I have shown you. (CUDA thread launch is similar in spirit to a forall loop in data parallel model examples)

(could now be up to 192 SMs, etc., …)

Thank you.