CUDA Programming Model Overview

Peter Messmer

3 Ways to Program GPUs

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

Outline

- CUDA model
- CUDA programming basics
- Development Resources
- GPU architecture for computing

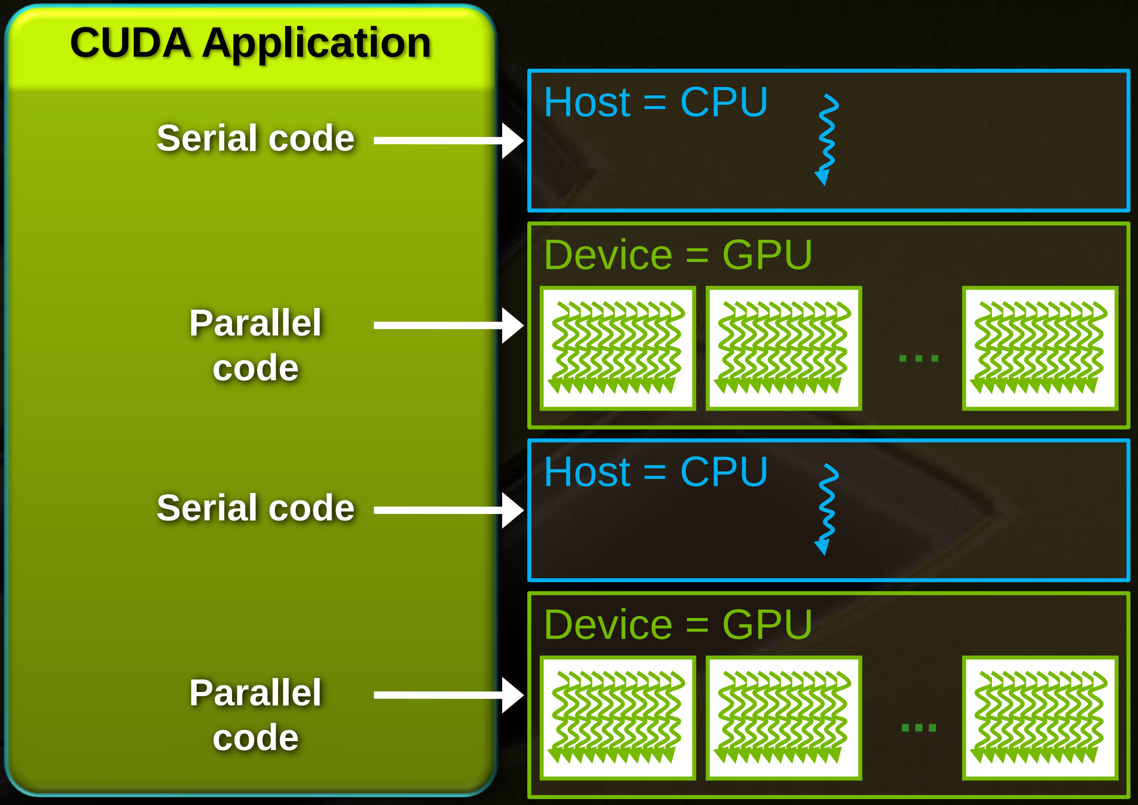
CUDA Model

What is CUDA?

- C++ with extensions
  - Fortran support via e.g. PGI's CUDA Fortran
- CUDA goals:
  - Scale to 100's of cores, 1000's of parallel threads
  - Let programmers focus on parallel algorithms
  - Enable heterogeneous systems (i.e., CPU+GPU)
- CUDA defines:
  - Programming model
  - Memory model

Anatomy of a CUDA Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements



CUDA Kernels

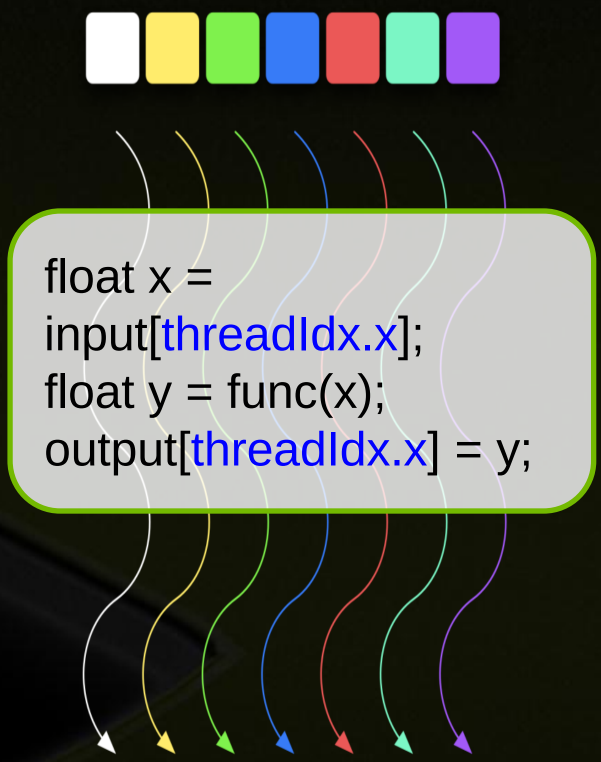
- **Parallel portion of application: execute as a kernel**
  - **Entire GPU executes kernel, many threads**

- **CUDA threads:**
  - **Lightweight**
  - **Fast switching**
  - **1000s execute simultaneously**

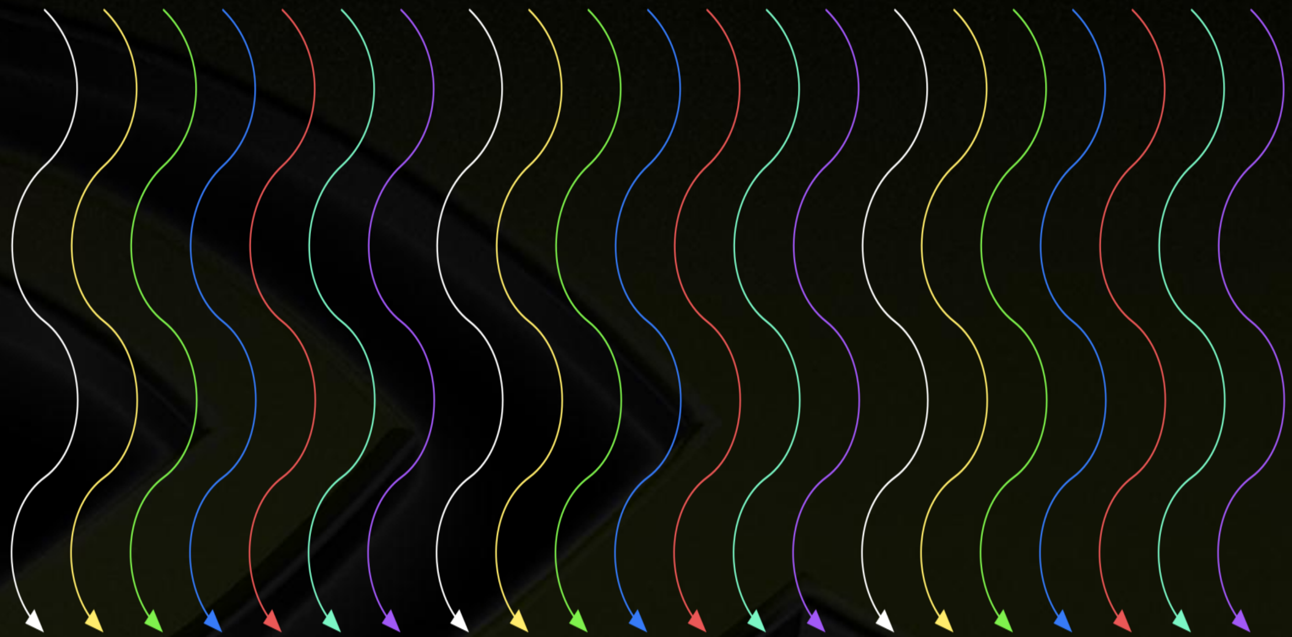
CPU	Host	Executes functions
GPU	Device	Executes kernels

CUDA Kernels: Parallel Threads

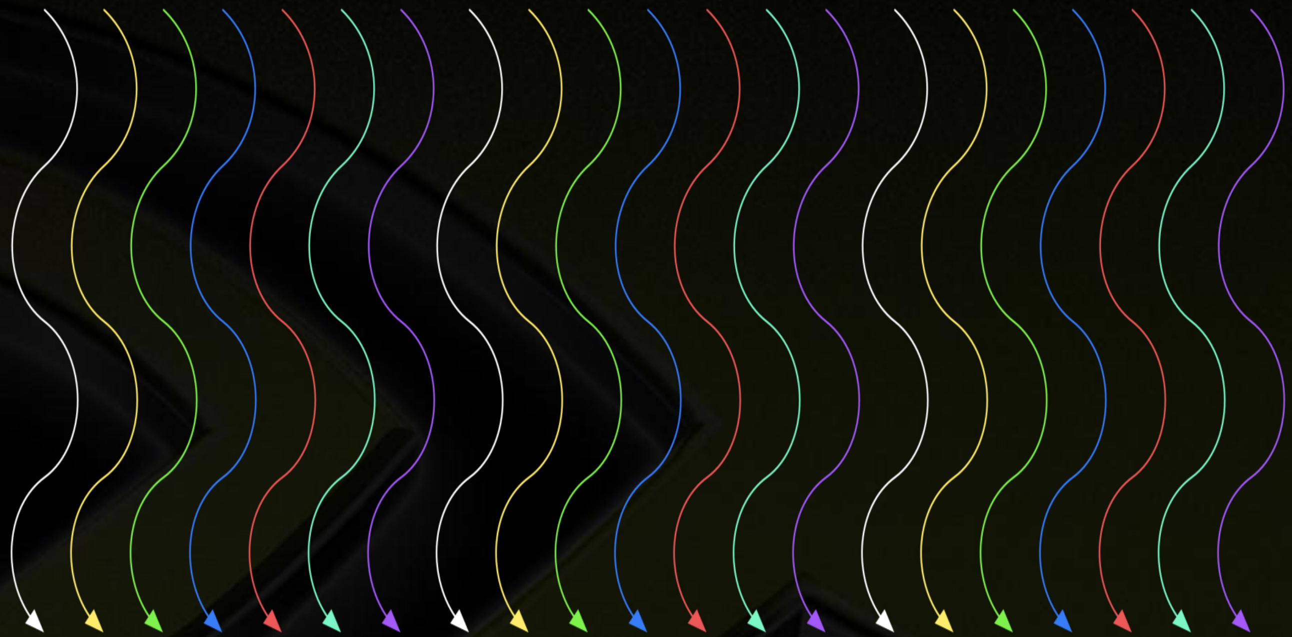
- A **kernel** is a function executed on the GPU as an array of threads in parallel
- All threads execute the same code, can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions



CUDA Kernels: Subdivide into Blocks

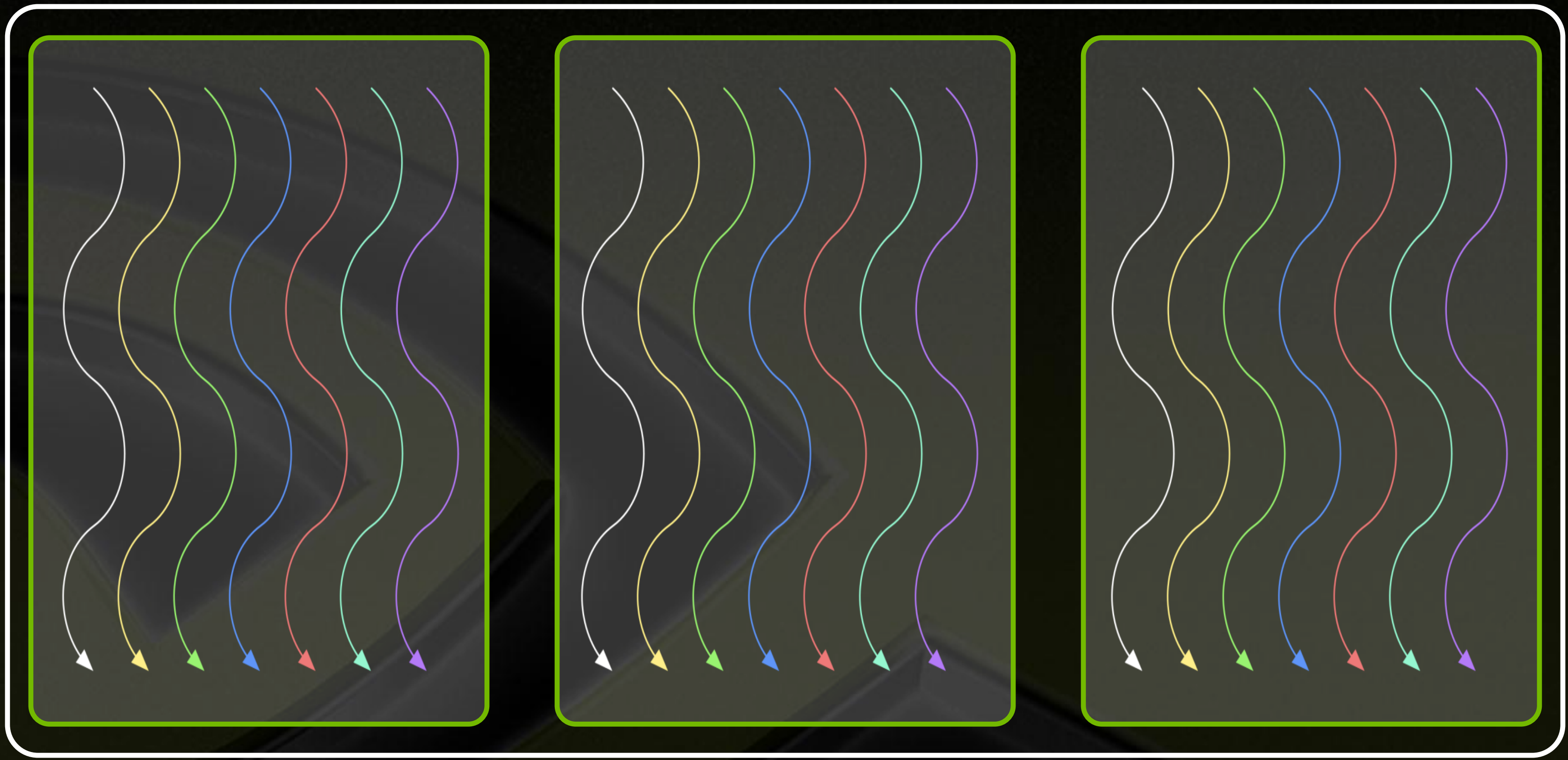


CUDA Kernels: Subdivide into Blocks



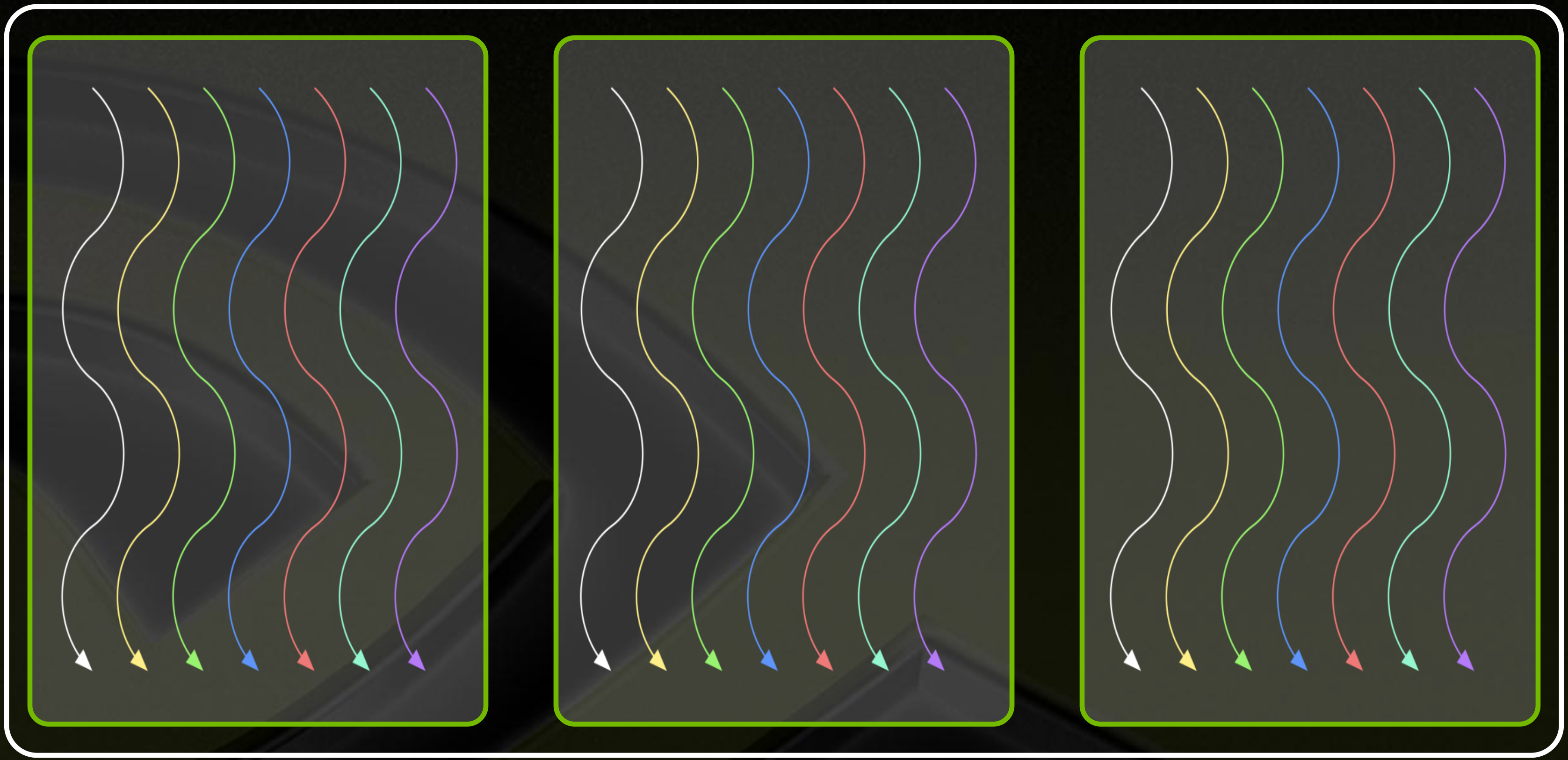
- Threads are grouped into **blocks**

CUDA Kernels: Subdivide into Blocks



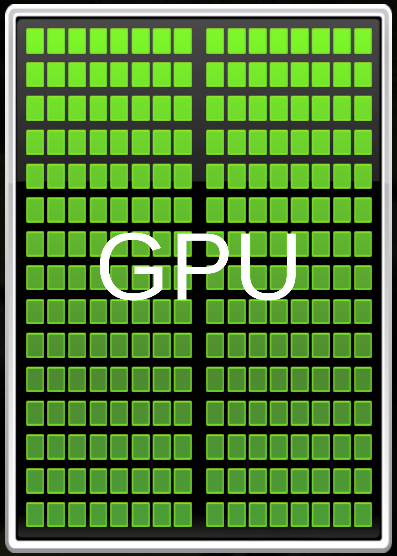
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

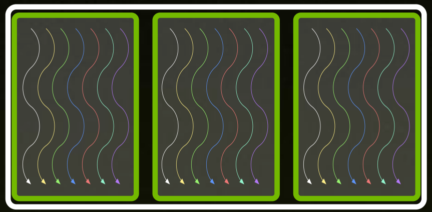
CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

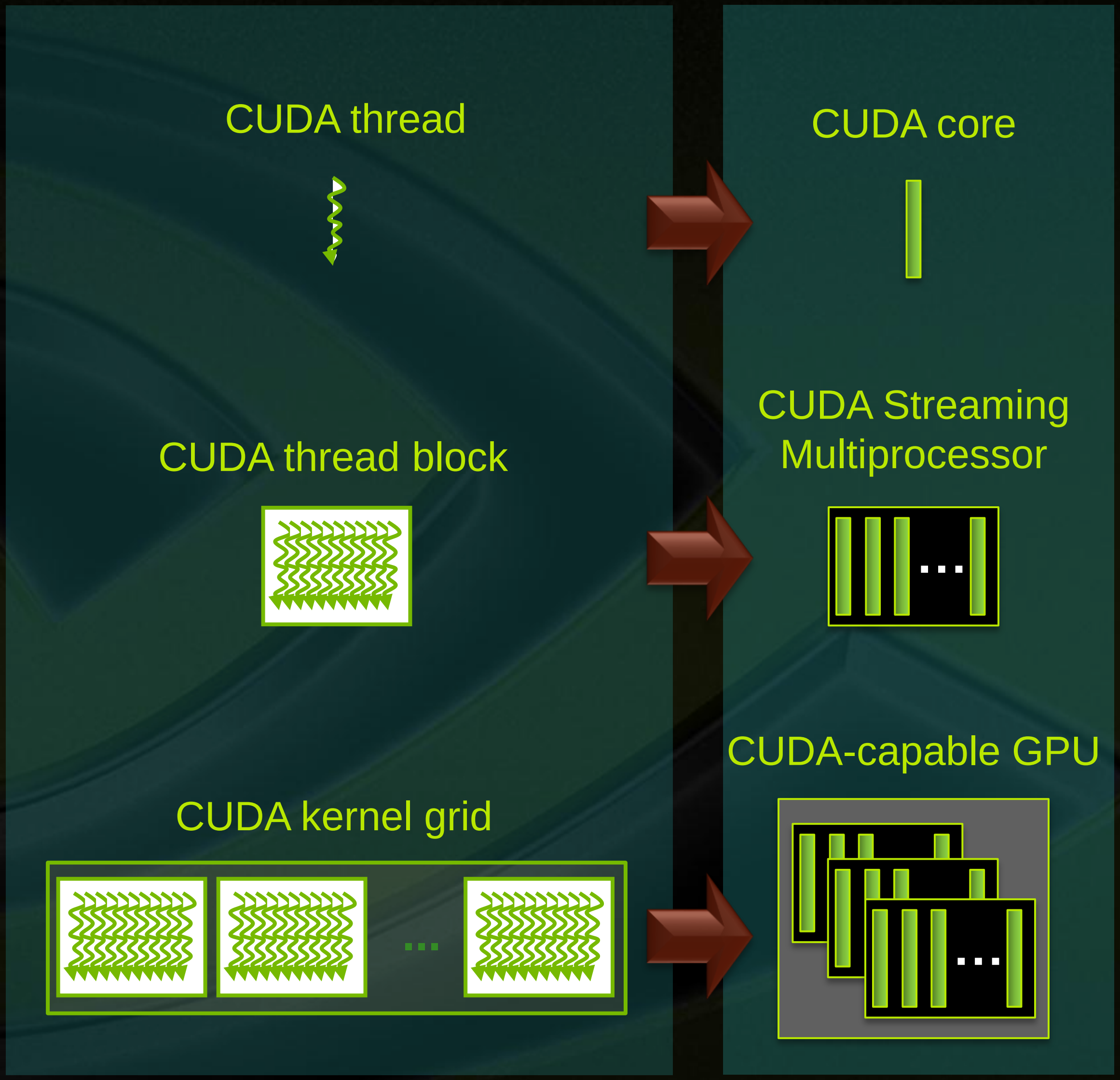
CUDA Kernels: Subdivide into Blocks





- Threads are grouped into **blocks**
  - Note: Adjacent threads execute in lock-step scheduling groupings called **warps**; a block comprises one or more warps
- **Blocks** are grouped into a **grid**
- A **kernel** is executed as a **grid** of **blocks** of **threads**

Kernel Execution



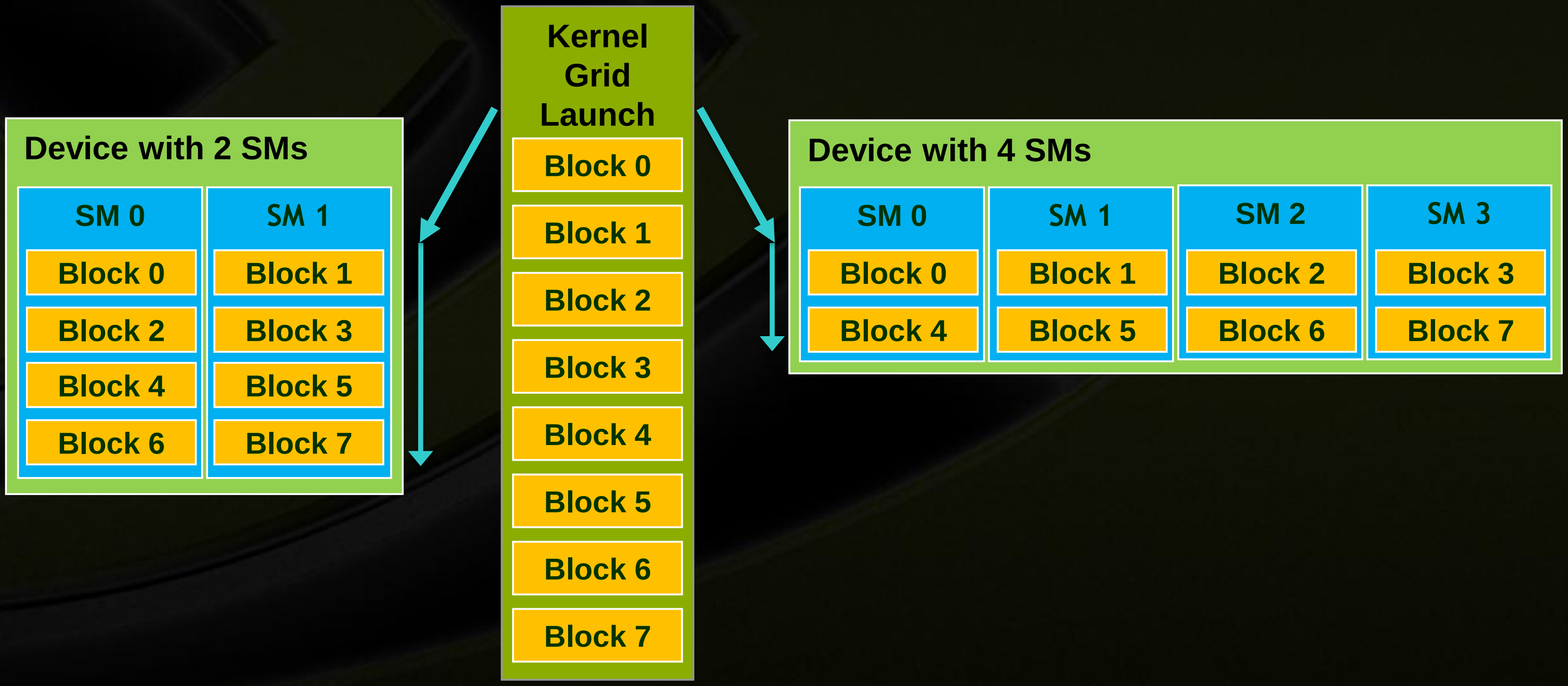
- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory that they all use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other

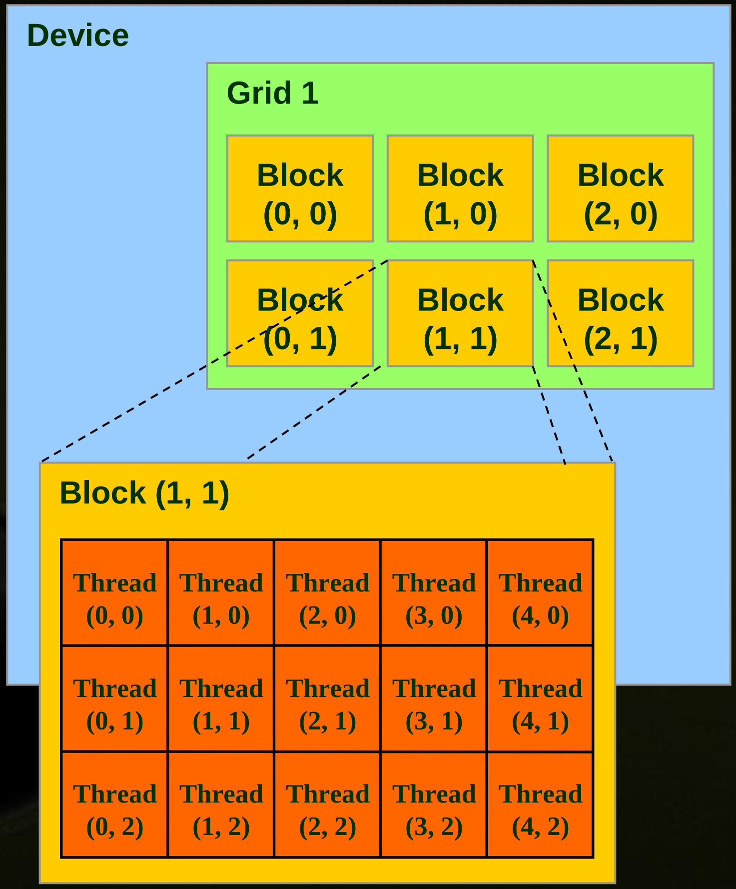
Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs



IDs and Dimensions

- **Threads:**
  - 3D IDs, unique within a block
- **Blocks:**
  - 2D IDs, unique within a grid
- **Dimensions set at launch time**
  - Can be unique for each section
- **Built-in variables:**
  - threadIdx, blockIdx
  - blockDim, gridDim



Launching kernels

- **Modified C function call syntax:**

```
kernel<<<dim3 grid, dim3 block>>>(…)
```

- **Execution Configuration (“<<< >>>”):**
  - **grid dimensions: x and y**
  - **thread-block dimensions: x, y, and z**

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

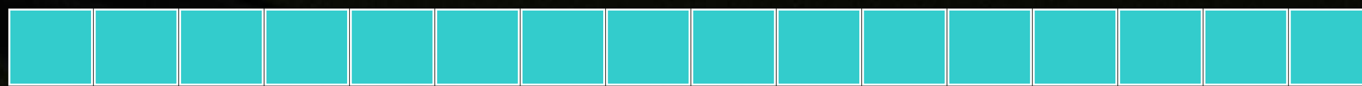
Minimal Kernels

```
__global__ void minimal( int* d_a)
{
   *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   d_a[idx] = value;
}
```

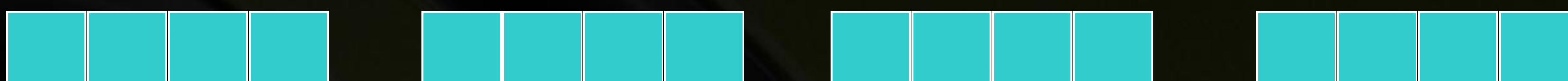
Common Pattern!

Example: Increment Array Elements

Increment N-element vector a by scalar b



Let’s assume N=16, blockDim=4 -> 4 blocks



blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

will map from local index threadIdx to global index

NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}

void main()
{
   .....
      increment_cpu(a, b, N);
}
```

CUDA program

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

void main()
{
    …..
       dim3 dimBlock (blocksize);
       dim3 dimGrid( ceil( N / (float)blocksize)  );
       increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Minimal Kernel for 2D data

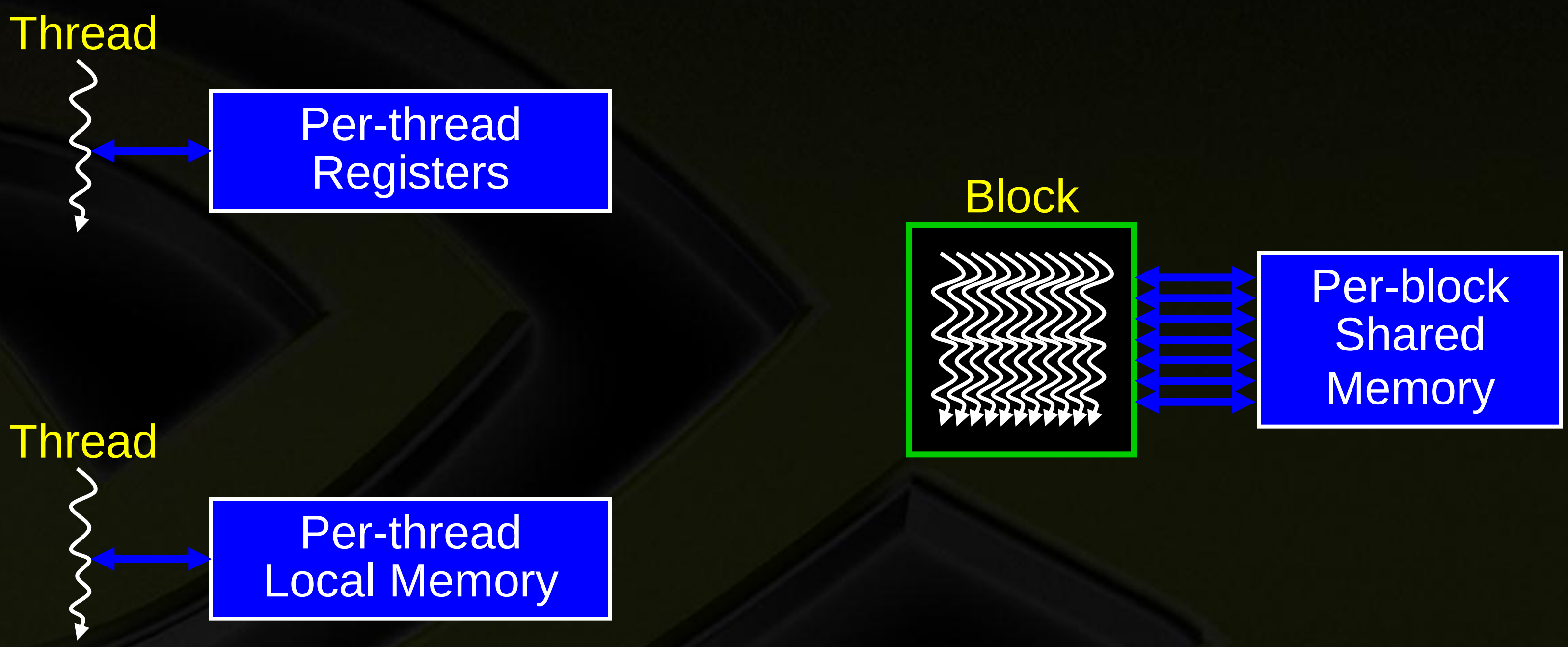
```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
   int iy = blockDim.y * blockIdx.y + threadIdx.y;
   int ix = blockDim.x * blockIdx.x + threadIdx.x;
   int idx = iy * w + ix;

   d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

CUDA Memory Model

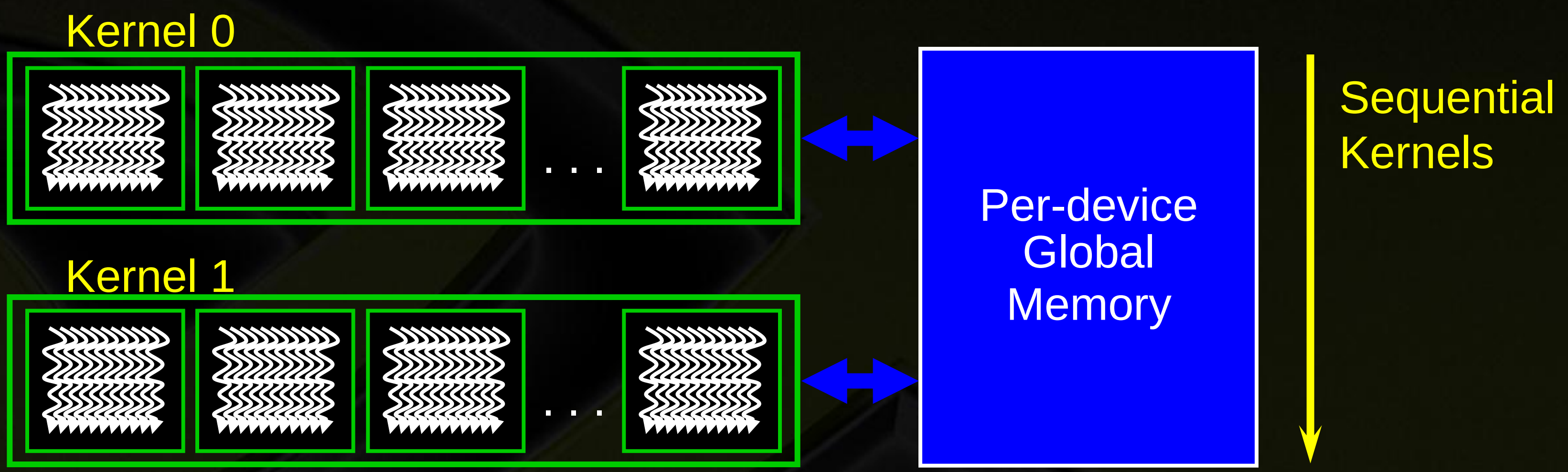
Memory Model

- Registers
  - Per thread
  - Data lifetime = thread lifetime
- Local memory
  - Per thread off-chip memory (physically in device DRAM)
  - Data lifetime = thread lifetime
- Shared memory
  - Per thread block on-chip memory
  - Data lifetime = block lifetime
- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation
- Host (CPU) memory
  - Not directly accessible by CUDA threads

Memory Model



Memory Model



Memory Model

Introduction to CUDA Programming

Outline of CUDA Basics

- Basics to setup and execute CUDA code:
  - GPU memory management
  - Extensions to C++ for kernel code
  - GPU kernel launches
- Some additional basic features:
  - Checking CUDA errors
  - CUDA event API
- See the Programming Guide for the full API

  http://docs.nvidia.com/cuda/cuda-c-programming-guide

GPU Memory Allocation / Release

- Host (CPU) manages GPU memory:
  - 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 * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes,
  enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - blocks CPU thread
  - doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- Non-blocking memcopies are provided

Programming Exercise 1

- Send a string to the GPU and back

- Allocate GPU memory for *n* chars
- Copy from CPU to GPU and back
- Print the values

Programming Exercise 1

- **Compiling a CUDA application**

```
nvcc --arch=sm_35 -o sample1  sample1.cu
```

- **Execution**

```
salloc --res=course --nodes=1  --walltime=00:10:00
aprun –n 1 ./sample1
```

- **Some CUDA API**

```
cudaMalloc (void ** pointer, size_t nbytes)
cudaFree (void* pointer)
cudaMemcpy( void *dst,   void *src,   size_t nbytes,
                    enum cudaMemcpyKind direction);
```

Programming Exercise 1

```
#include <stdio.h>
#define N 100

int main(int argc, char** argv){
    char msg_in[] = "hello accelerator";
    char msg_out[N];
    char* d_msg;                // will be message on the device

    cudaMalloc(&d_msg, N*sizeof(char));
    cudaMemcpy(d_msg, msg_in, N*sizeof(char), cudaMemcpyHostToDevice);

    cudaMemcpy(msg_out, d_msg, N*sizeof(char), cudaMemcpyDeviceToHost);

    printf("msg_out= %s\n", msg_out);
    cudaFree(d_msg);

    return 0;
}
```

Code executed on GPU

- C++ function with some restrictions:
  - Can only access GPU memory (with some exceptions)
  - No variable number of arguments
  - No static variables

- Must be declared with a qualifier:
  - __global__ : launched by CPU,
    cannot be called from GPU
    must return void
  - __device__ : called from other GPU functions,
    cannot be launched by the CPU
  - __host__ : can be executed by CPU
  - __host__ and __device__ qualifiers can be combined

- Built-in variables:
  - gridDim, blockDim, blockIdx, threadIdx

Variable Qualifiers (GPU code)

- __device__
  - stored in global memory (not cached, high latency)
  - accessible by all threads
  - lifetime: application
- __constant__
  - stored in global memory (cached)
  - read-only for threads, written by host
  - Lifetime: application
- __shared__
  - stored in shared memory (latency comparable to registers)
  - accessible by all threads in the same threadblock
  - lifetime: block lifetime
- Unqualified variables:
  - Stored in local memory:
    - scalars and built-in vector types are stored in registers
    - arrays are stored in device memory

Launching kernels on GPU

- Launch parameters:
  - grid dimensions (up to 2D)
  - thread-block dimensions (up to 3D)
  - shared memory: number of bytes per block
    - for extern smem variables declared without size
    - Optional, 0 by default
  - stream ID
    - Optional, 0 by default

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

Programming Exercise 2

- Build on Exercise 1
- Write a kernel to convert a simple string to upper case (Fact: ‘a’ = 96, ‘A’ = 64)

Programming Exercise 2

```
#include <string.h>
#include <stdio.h>
#define N 100

__global__ void kernel(char* msg, int n){
        int tid = threadIdx.x;
        if(threadIdx.x < n) {
          char m = msg[tid];
          msg[tid] = m > 96 ? m  - 32 : m ;
        }
}

int main(int argc, char** argv){
…
        cudaMemcpy(d_msg, msg_in, N*sizeof(char), cudaMemcpyHostToDevice);
        kernel<<<1, 100>>>(d_msg, strlen(msg_in));
        cudaMemcpy(msg_out, d_msg, N*sizeof(char), cudaMemcpyDeviceToHost);
..
}
```

Thread Cooperation via Smem

- Shared memory accessible by all threads in block

```
__shared__ double a[32];
```

- "software managed cache"

- Shared memory partitioned among all blocks running on a SM

Thread 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 memory
- Should be used in conditional code only if the conditional is uniform across the entire thread block

Programming Exercise 3

- Build on Exercise 2
- Write a kernel to convert the first letter in a word to upper case

Programming Exercise 3

```
__global__ void kernel(char* msg, int n){

    __shared__ char s_msg[N];
    int tid = threadIdx.x;

    if(tid < n)  s_msg[tid] = msg[tid];

    __syncthreads();

    if(tid < n & tid > 0) {
        char m = s_msg[tid - 1];
          if(m == ' ') msg[tid] = s_msg[tid] - 32;
    }
}
```

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed
- Memcopies are synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete
- Asynchronous CUDA calls provide:
  - non-blocking memcopies
  - ability to overlap memcopies and kernel execution

Device Management

- CPU can query and select GPU devices
  - cudaGetDeviceCount(int* count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *current_device)
  - cudaGetDeviceProperties(cudaDeviceProp* prop,
    int device)
  - cudaChooseDevice(int *device, cudaDeviceProp* prop)
- Multi-GPU setup:
  - device 0 is used by default

CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - except for kernel launches
  - cudaError_t type
- cudaError_t cudaGetLastError(void)
  - returns the code for the last error (no error has a code)
- char* cudaGetErrorString(cudaError_t code)
  - returns a null-terminted character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

CUDA Event API

- **Events are inserted (recorded) into CUDA call streams**
- **Usage scenarios:**
  - **measure elapsed time for CUDA calls (clock cycle precision)**
  - **query the status of an asynchronous CUDA call**
  - **block CPU until CUDA calls prior to the event are completed**
  - **asyncAPI sample in CUDA SDK**

```
cudaEvent_t start, stop;
cudaEventCreate(&start);              cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);  cudaEventDestroy(stop);
```

What about Fortran?

- ## Mixed language approach

```
call myfun()
```

```
extern "C" void myfunc() {
  mykernel<<< grid, block >>>()
}
```

- ## PGI's CUDA Fortran

```
real*8, dimension(:) :: u
real*8, device, dimension(:) :: a, b
a = u
call mykernel<<< grid, block  >>> ( a, b)

attributes(global) subroutine mykernel(a, b)
..
end subroutine mykernel
```

What did we skip? What’s next?

- Concurrent kernels and data transfers
  - Streams
- Optimization
  - Global memory access, “coalescence”
- Mapped memory
- Multi-GPU programming
  - Device-device transfers