CUDA Basics

Dr. Alptekin Temizel

atemizel@metu.edu.tr

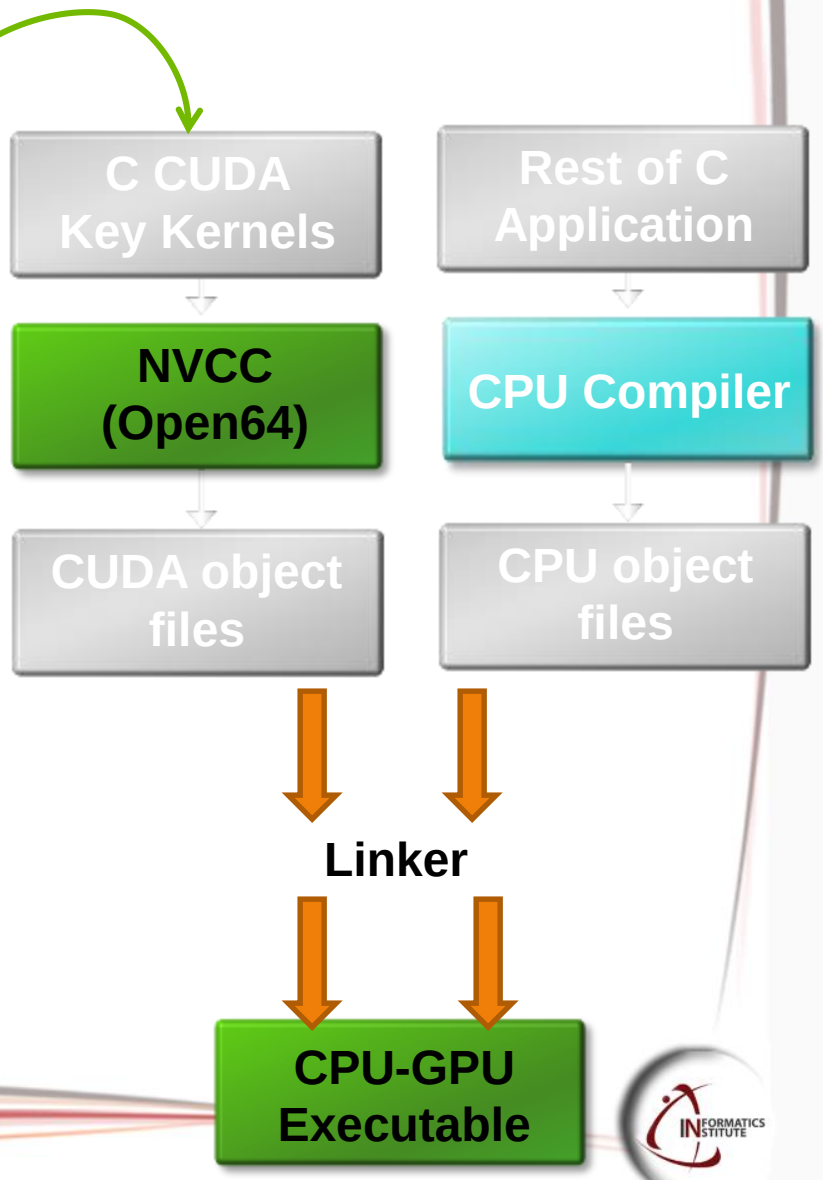
Compiling C for CUDA Applications

```
void serial_function(… ) {
  ...
}
void other_function(int ... ) {
  ...
}
```

```
void saxpy_serial(float ... ) {
   for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

```
void main( ) {
  float x;
  saxpy_serial(..);
  ...
}
```

Modify into Parallel CUDA code



CUDA Compiler (1)

- Any source file containing CUDA language extensions must be compiled with NVCC

- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...

- NVCC outputs:
  - C code (host CPU Code): Must then be compiled with the rest of the application using another tool
  - PTX

CUDA Compiler (2)

- **Parallel Thread Execution (PTX)** is a pseudo-assembly language used CUDA programming environment.
- The nvcc compiler translates code written in CUDA, into PTX.
- The graphics driver contains a compiler which translates the PTX into a binary code which can be run on the processing cores.

CUDA Compiler (3)

- nvcc can be used to generate both architecture-specific cubin files and forward-compatible PTX versions of each kernel.

- Each cubin file targets a specific compute-capability version and is forward-compatible only with GPU architectures of the same major version number.

CUDA Compiler (4)

- For example, cubin files that target compute capability 3.0 are supported on all compute-capability 3.x (Kepler) devices but are not supported on compute-capability 5.x (Maxwell) or 6.x (Pascal) devices.

- For this reason, to ensure forward compatibility with GPU architectures introduced after the application has been released, it is recommended that all applications include PTX versions of their kernels.

- CUDA Runtime applications containing both cubin and PTX code for a given architecture will automatically use the cubin by default, keeping the PTX path strictly for forward-compatibility purposes.

CUDA Compiler (5)

- When a CUDA application launches a kernel, the CUDA Runtime determines the compute capability of each GPU in the system and uses this information to automatically find the best matching cubin or PTX version of the kernel that is available.

- If a cubin file supporting the architecture of the target GPU is available, it is used; otherwise, the CUDA Runtime will load the PTX and JIT-compile that PTX to the GPU's native cubin format before launching it. If neither is available, then the kernel launch will fail.

CUDA Compiler (6)

- The main advantages of providing native cubins:
  - It saves the end user the time it takes to JIT-compile kernels that are available only as PTX. All kernels compiled into the application must have native binaries at load time or else they will be built just-in-time from PTX, including kernels from all libraries linked to the application, even if those kernels are never launched by the application. Especially when using large libraries, this JIT compilation can take a significant amount of time.
  - The CUDA driver will cache the cubins generated as a result of the PTX JIT, so this is mostly a one-time cost for a given user, but it is time best avoided whenever possible.

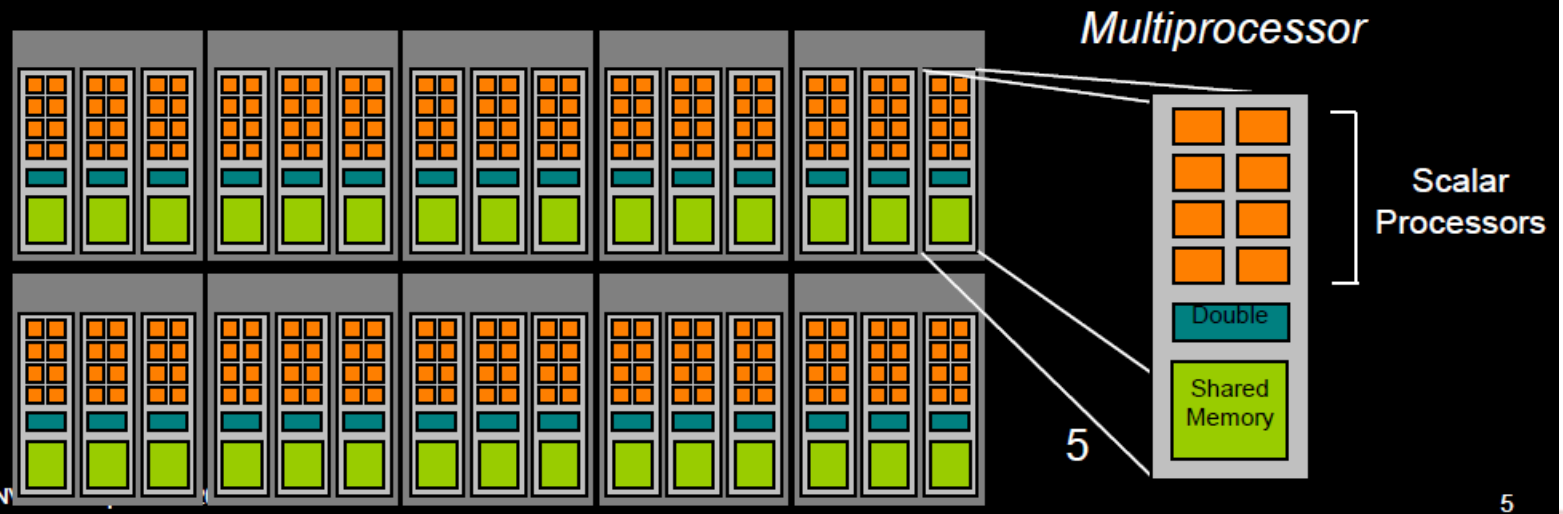
- PTX JIT-compiled kernels often cannot take advantage of architectural features of newer GPUs, meaning that native-compiled code may be faster or of greater accuracy.

Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

GT200 Architecture

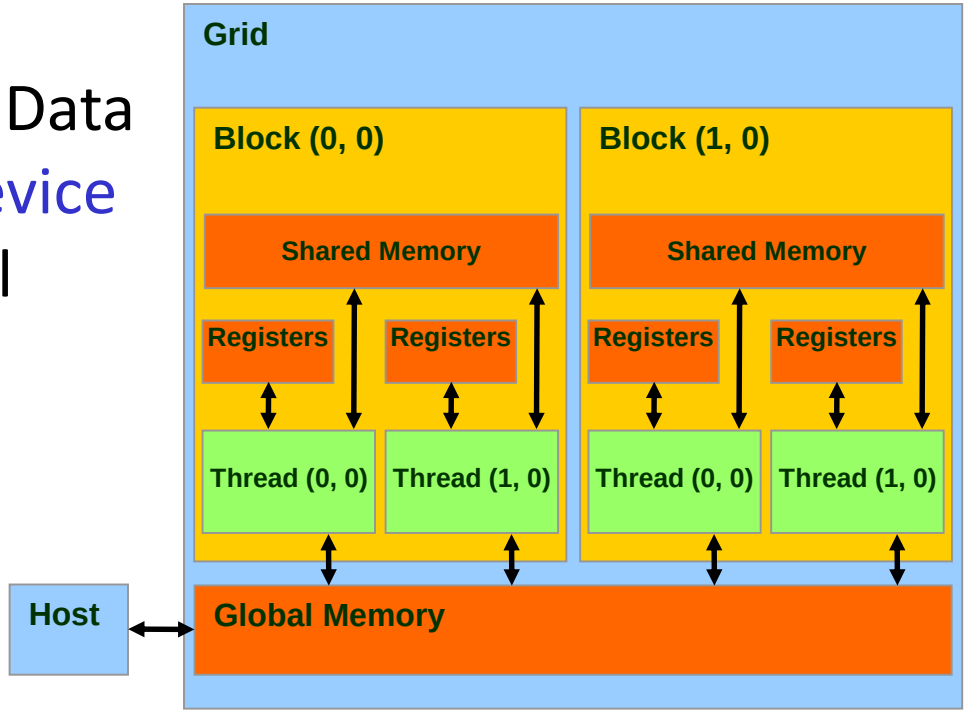
- Device contains 30 Streaming Multiprocessors (SMs)
- Each SM contains
  - 8 scalar processors
  - 1 double precision unit
  - 2 special function units
  - shared memory (16 K)
  - registers (16,384 32-bit=64 K)



CUDA Memory Model Overview

Global memory

- Main means of communicating R/W Data between host and device
- Contents visible to all threads
- Long latency access



Outline of CUDA Basics

- Basic Memory Management
- Basic Kernels and Execution on GPU
- Coordinating CPU and GPU Execution
- Development Resources

- See the Programming Guide for the full API – available at:

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

Basic Memory Management

CPU Memory Allocation / Release

- void* malloc (size_t size);
- void * memset (void * ptr, int value, size_t num);
- void free (void* ptr);

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * h_a = 0;
h_a = malloc(nbytes );
memset( h_a, 0, nbytes);
free(h_a);
```

GPU Memory Allocation / Release

- Host (CPU) manages device (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);
```

Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions
- Pointers are just addresses
  - Can't tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

Data Copies

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

Code Walkthrough 1

- Allocate CPU memory for *n* integers
- Allocate GPU memory for *n* integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Basic Kernels and Execution on GPU

CUDA Programming Model

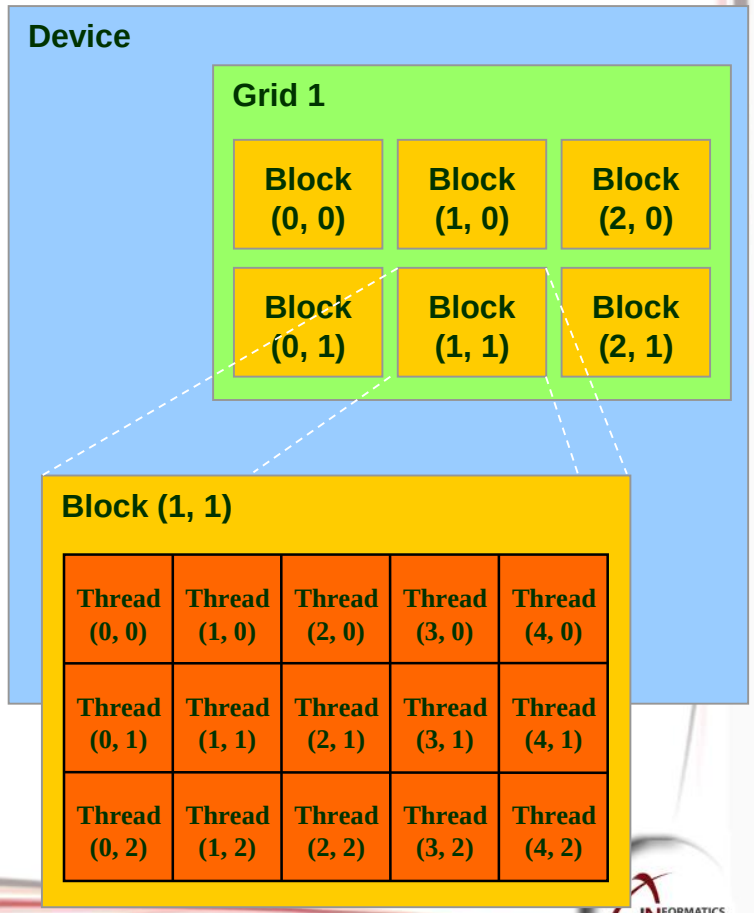
- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

IDs and Dimensions

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



* Devices with capability <2.0 support up to 2D arrays of blocks

IDs and Dimensions

- Number of Threads:
  - Total size of a block is limited to 1024 threads*
  - These could be distributed into the three dimensions as long as total of 1024 is not exceeded:
    - (512,1,1), (8,16,4) and (32,16,2) are all valid
    - (32,32,2) is invalid – exceed the maximum
- Number of Blocks:
  - Each dimension of grid could be up to 65536.
    - gridDim.x, gridDim.y, gridDim.z range from 1 to 65536

* Devices with capability <2.0 support up to 512 threads in a block

IDs and Dimensions

- Choice of 1D, 2D or 3D thread organizations is usually based on the nature of the data.
- For example, to 2D data such as images or matrices, it is more convenient to use a 2D thread organization.

Code executed on GPU

- C function with some restrictions:
  - Can only access GPU memory
  - No static variables
  - ~~No variable number of arguments~~
  - ~~No recursion~~

- 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__ : executed by CPU – this is the default if no qualifier is specified

Note: __host__ and __device__ can be combined -> this results in two versions of the same function, one for the device and one for the host

Code Walkthrough 2

- Build on Walkthrough 1
- Write a kernel to initialize integers
- Copy the result back to CPU
- Print the values

Kernel Code (executed on GPU)

```
__global__ void mykernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Launching kernels on GPU

- Launch parameters:
  - grid dimensions (up to 2D), dim3 type
  - thread-block dimensions (up to 3D), dim3 type
  - 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>>>(...);
```

```
#include <stdio.h>

__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

     dim3 grid, block;
     block.x = 4;
     grid.x  = dimx / block.x;

     kernel<<<grid, block>>>( d_a );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Kernel Variations and Output (1)

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

Kernel Variations and Output (2)

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

Kernel Variations and Output (3)

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Code Walkthrough 3

- Build on Walkthrough 2
- Write a kernel to increment $n \times m$ integers
- Copy the result back to CPU
- Print the values

Kernel with 2D Indexing

```
__global__ void kernel( int *a, int dimx )
{
    int Col   = blockIdx.x*blockDim.x + threadIdx.x;
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = Row*dimx + Col;

    a[idx]  = a[idx]+1;
}
```

Kernel with 2D Indexing

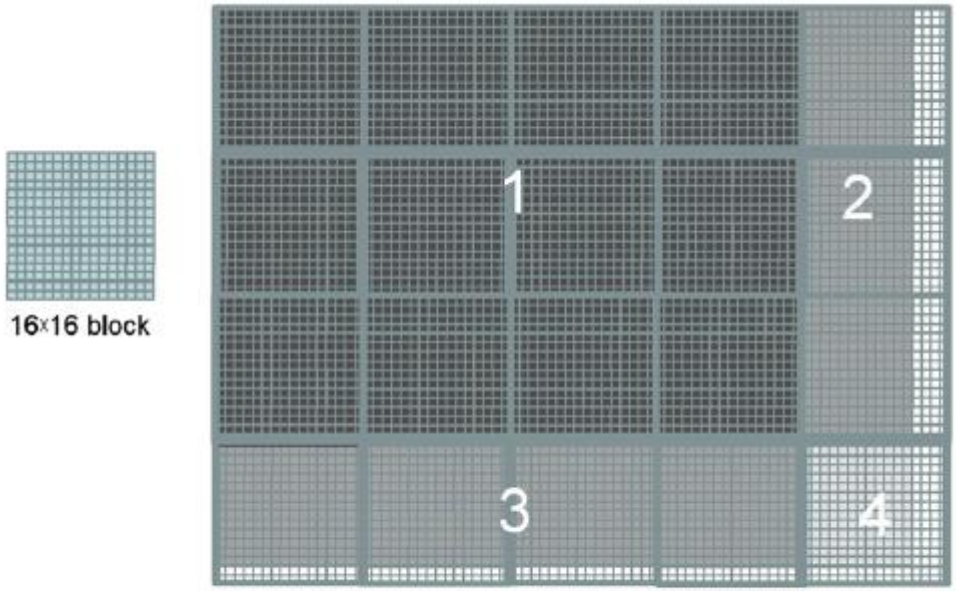


FIGURE 4.5

Covering a 76 × 62 picture with 16 × 16 blocks.

```
__global__ void kernel( int *a, int dimx )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

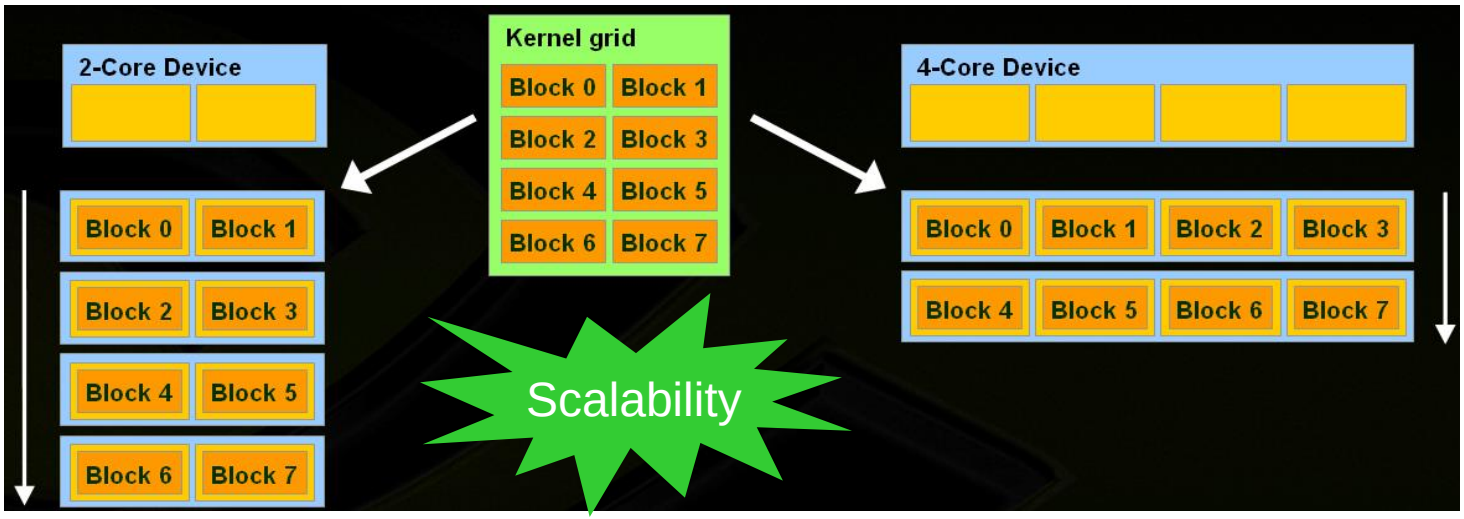
```
int main()
{
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int row=0; row<dimy; row++)
    {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col] );
        printf("\n");
    }

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement gives scalability

Blocks must be independent

- Thread blocks can run in any order
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices



Coordinating CPU and GPU Execution

Synchronizing GPU and CPU

- 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
- cudaDeviceSynchronize()
  - blocks until all previous CUDA calls complete
- Asynchronous CUDA calls provide:
  - non-blocking memcopies
  - ability to overlap memcopies and kernel execution

CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - except 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-terminated 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);
```

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
  - one CPU thread can control one GPU
    - multiple CPU threads can control the same GPU
      - calls are serialized by the driver

Thread Synchronization Function

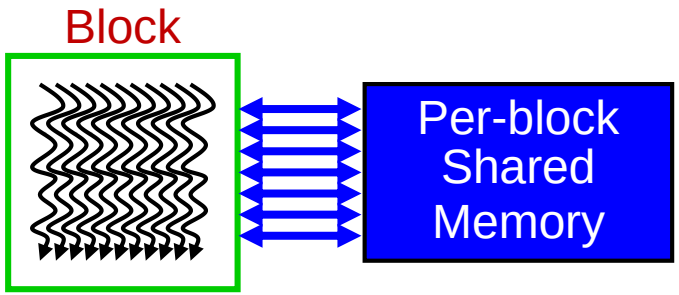
- `void __syncthreads();`
- Synchronizes all threads in a *thread-block*
  - Since threads are scheduled at run-time
  - Once all threads have reached this point, execution resumes normally
  - Used for synchronizing when accessing shared memory
- Should be used in conditional code only if the conditional is uniform across the entire thread block

Memory Model Review

- Local storage
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)
  - Data lifetime = thread lifetime

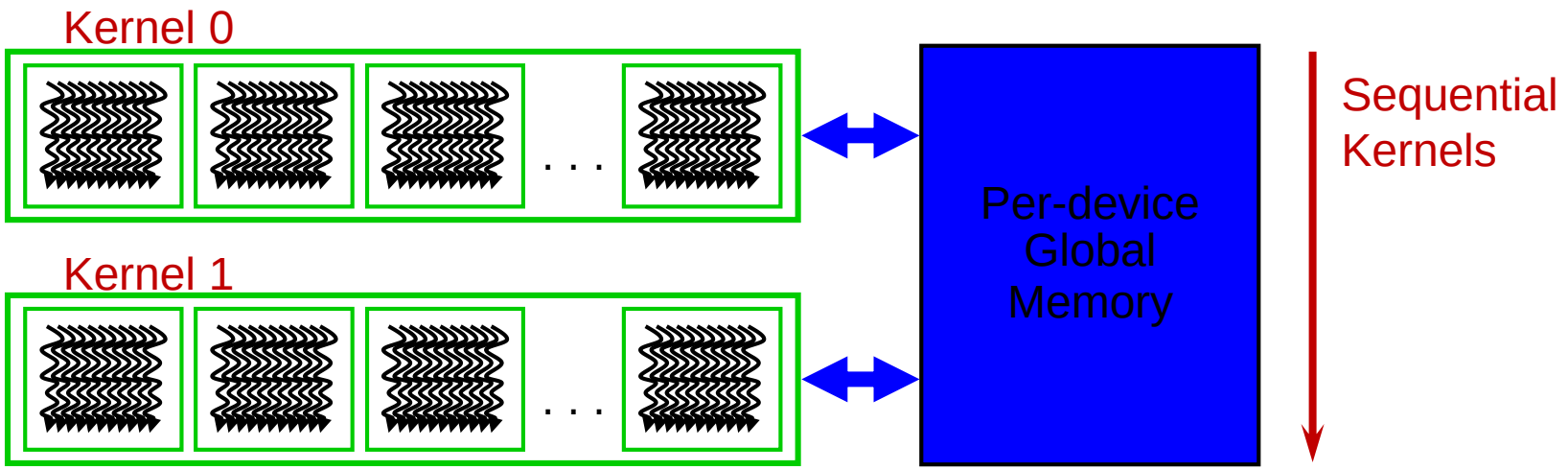
Memory Model Review

- Shared memory
  - Each thread block has own shared memory
    - Accessible only by threads within that block
  - Data lifetime = block lifetime

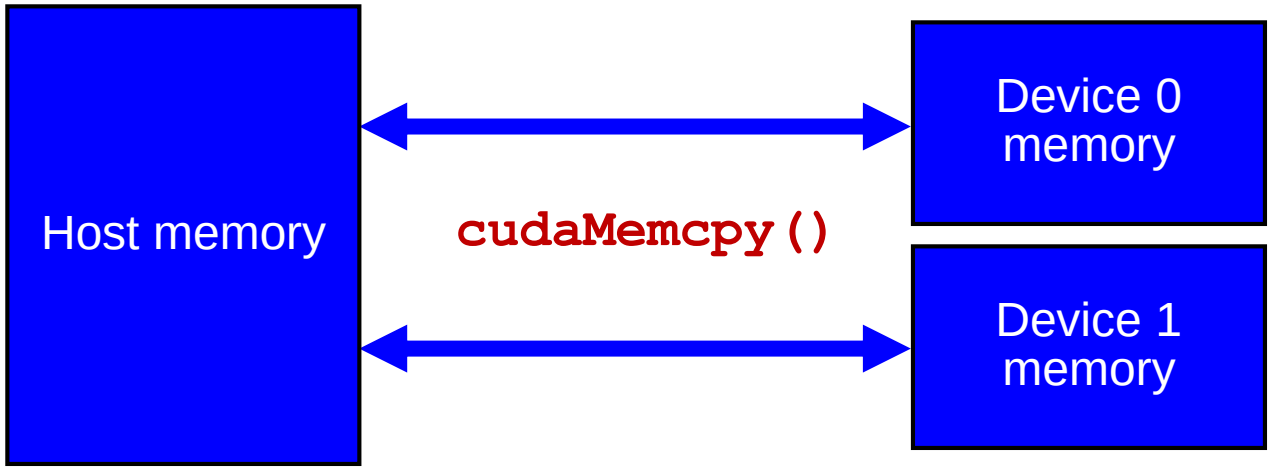


Memory Model Review

- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation



Memory Model Review



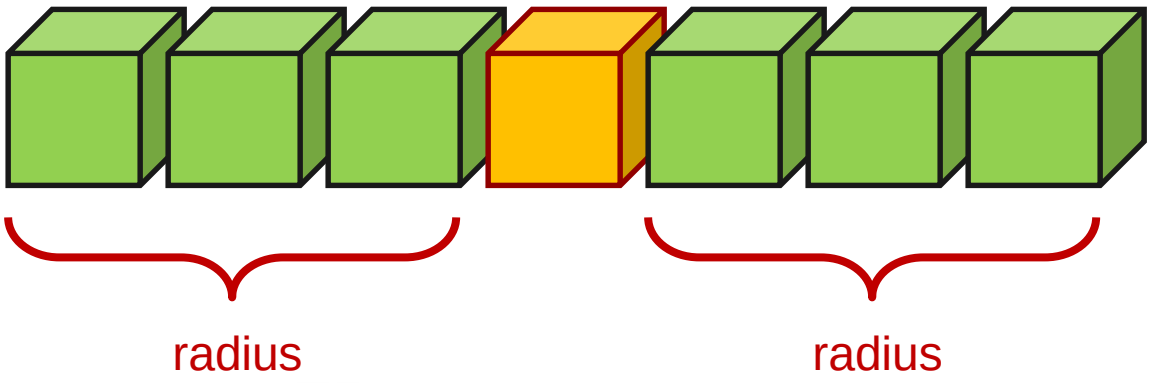
Shared Memory

Shared Memory

- On-chip memory
  - 2 orders of magnitude lower latency than global memory
  - Order of magnitude higher bandwidth than gmem
  - 16KB per multiprocessor (Fermi: 48KB, Maxwell: 64 or 96KB, Volta: 96KB)
- Allocated per threadblock
- Accessible by any thread in the threadblock
  - Not accessible to other threadblocks
- Several uses:
  - Sharing data among threads in a threadblock
  - User-managed cache (reducing gmem accesses)

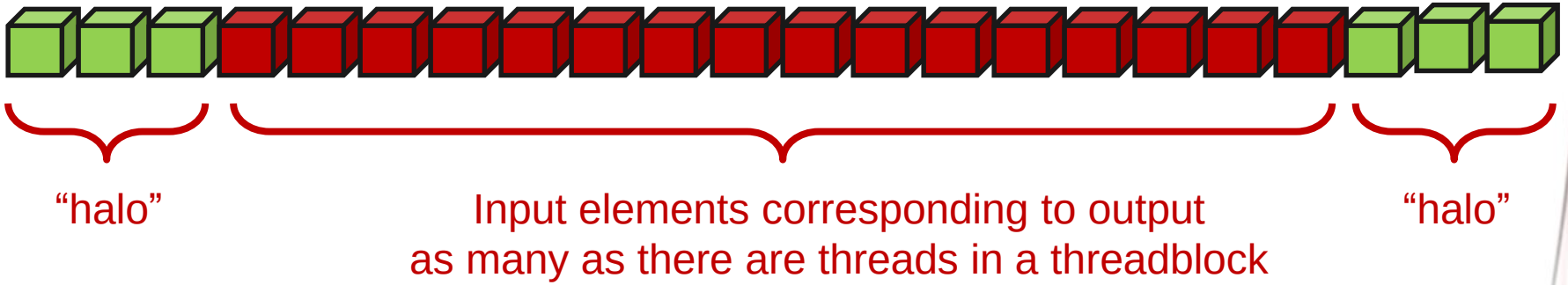
Example of Using Shared Memory

- Applying a 1D stencil:
  - 1D data
  - For each output element, sum all elements within a radius
- For example, radius = 3
  - Add 7 input elements

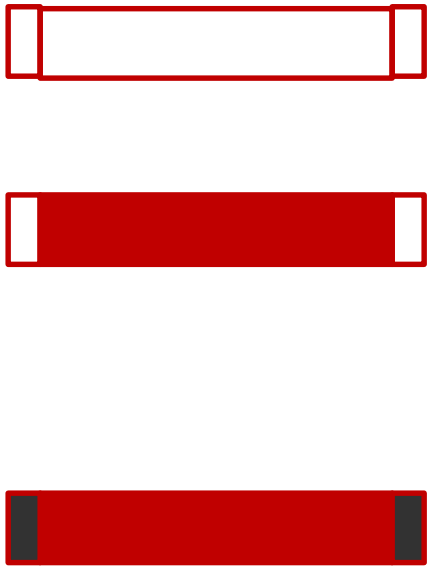


Implementation with Shared Memory

- 1D threadblocks (partition the output)
- Each threadblock outputs *BLOCK_DIMX* elements
  - Read input from gmem to smem
    - Needs *BLOCK_DIMX + 2*RADIUS* input elements
  - Compute
  - Write output to gmem



Kernel code



```
__global__ void stencil( int *output, int *input, int dimx )
{
    __shared__ int s_a[BLOCK_DIMX+2*RADIUS];

    int global_ix = blockIdx.x*blockDim.x + threadIdx.x;
    int local_ix   = threadIdx.x + RADIUS;

    s_a[local_ix] = input[global_ix];

    if ( threadIdx.x < RADIUS )
    {
        s_a[local_ix – RADIUS]                    = input[global_ix – RADIUS];
        s_a[local_ix + BLOCK_DIMX + RADIUS] = input[global_ix + RADIUS];
    }
    __syncthreads();

    int value = 0;
    for( offset = -RADIUS; offset<=RADIUS; offset++ )
        value += s_a[ local_ix + offset ];

     output[global_ix] = value;
}
```

Dynamically Determining the Size of the Shared Memory

- Notice that in the previous example we defined shared memory size fixed as compile time constant:

```
__shared__ int s_a[BLOCK_DIMX+2*RADIUS];
```

- We can have it decided at runtime without recompilation:

```
extern __shared__ s_a[];
```

- Use `CudaGetDeviceProperties()` function and check the `dev_prop.sharedMemPerBlock` to see the size of shared memory
- Then define the shared memory size as launch parameter (3rd parameter) – type of size_t:

```
kernel<<<grid, block, smem_size, 0>>>(...);
```

Dynamically Determining the Size of the Shared Memory - CAUTION

- If you have multiple extern declaration of shared:

```
extern __shared__ float As[];
extern __shared__ float Bs[];
```

this will lead to As pointing to the same address as Bs!

- You will need to keep As and Bs inside the 1D-array and access by setting an offset.