

**NVIDIA®**

**CUDA Basics**

# CUDA

A Parallel Computing Architecture for NVIDIA GPUs

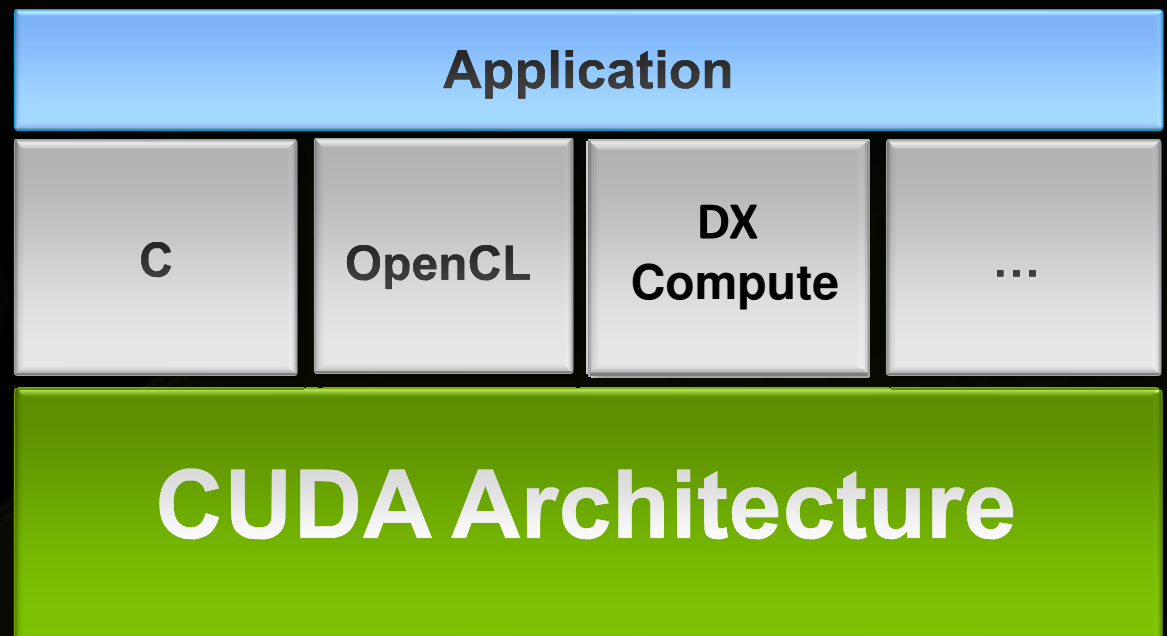


**Supports standard languages and APIs**

- C
- OpenCL
- DX Compute

**Supported on common operating systems:**

- Windows
- Mac OS
- Linux





# Outline of CUDA Basics

- **Basics Memory Management**
- **Basic Kernels and Execution on GPU**
- **Coordinating CPU and GPU Execution**
- **Development Resources**
- **See the Programming Guide for the full API**



**NVIDIA®**

## **Basic Memory Management**

# 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

# 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);
```

# 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 memcpyes 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;
}
```



**NVIDIA®**

**Basic Kernels and Execution on GPU**

# CUDA Programming Model



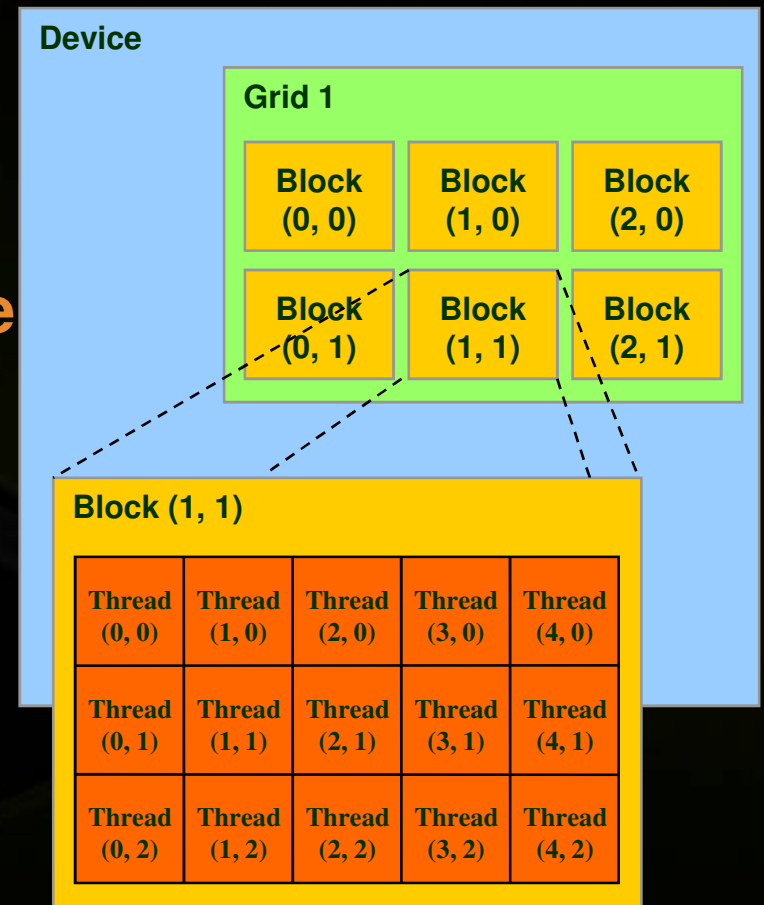
- **Parallel code (kernel) is launched and executed on a device by many threads**
- **Threads are grouped into thread blocks**
- **Parallel code is written for a thread**
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

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







# Code executed on GPU

- **C function with some restrictions:**
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - 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\_\_** : can be executed by CPU
  - **\_\_host\_\_** and **\_\_device\_\_** qualifiers can be combined
    - sample use: overloading operators



# 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 kernel( 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

```
__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 7 7

```
__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

```
__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 dimy )  
{  
    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;  
}
```





```
__global__ void kernel( int *a, int dimx, int dimy )
{
    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, dimy );

    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 );

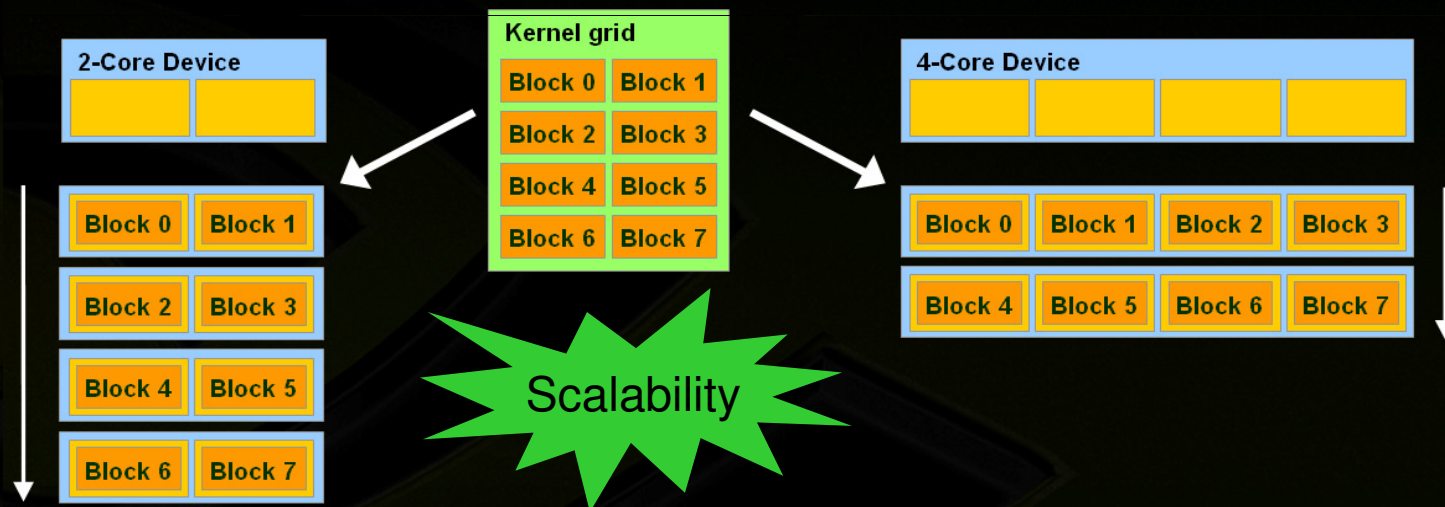
    return 0;
}
```

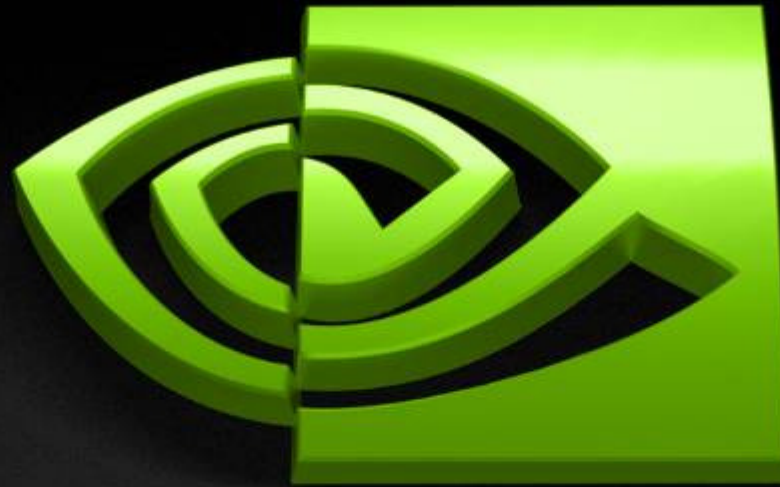
# Blocks must be independent

- **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





**NVIDIA®**

**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
- **cudaThreadSynchronize()**
  - 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





**NVIDIA®**

**Shared Memory**

# Shared Memory

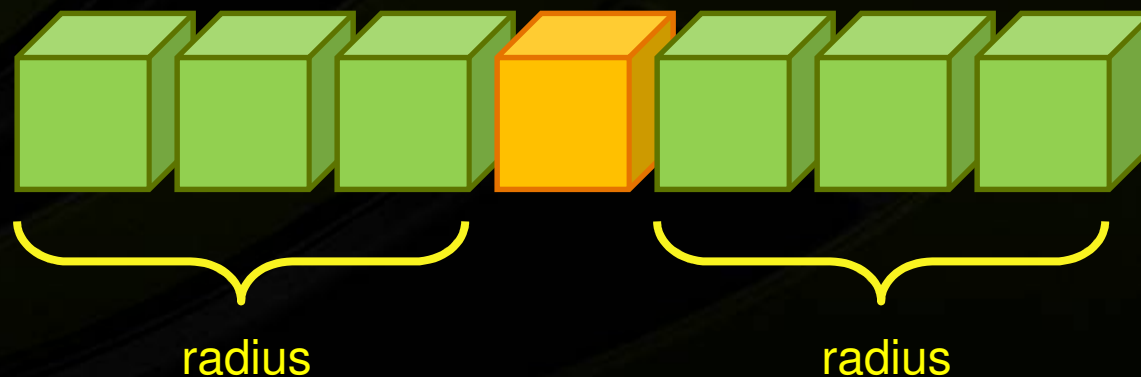


- **On-chip memory**
  - 2 orders of magnitude lower latency than global memory
  - Order of magnitude higher bandwidth than gmem
  - **16KB** per multiprocessor
    - NVIDIA GPUs contain up to **30** multiprocessors
- **Allocated per threadblock**
- **Accessible by any thread in the threadblock**
  - 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



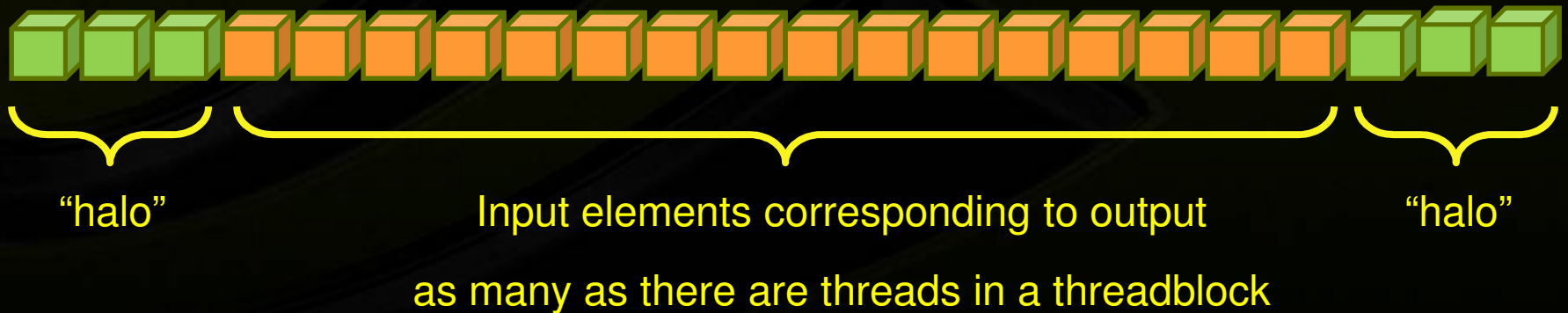
- **Applying a 1D stencil:**
  - 1D data
  - For each output element, sum all elements within a radius
- **For example, radius = 3**
  - Add 7 input elements



# 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, int dimy )
{
    __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] = input[global_ix + BLOCK_DIMX];
    }
    __syncthreads();

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

    output[global_ix] = value;
}
```



# 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 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

# Memory Model Review

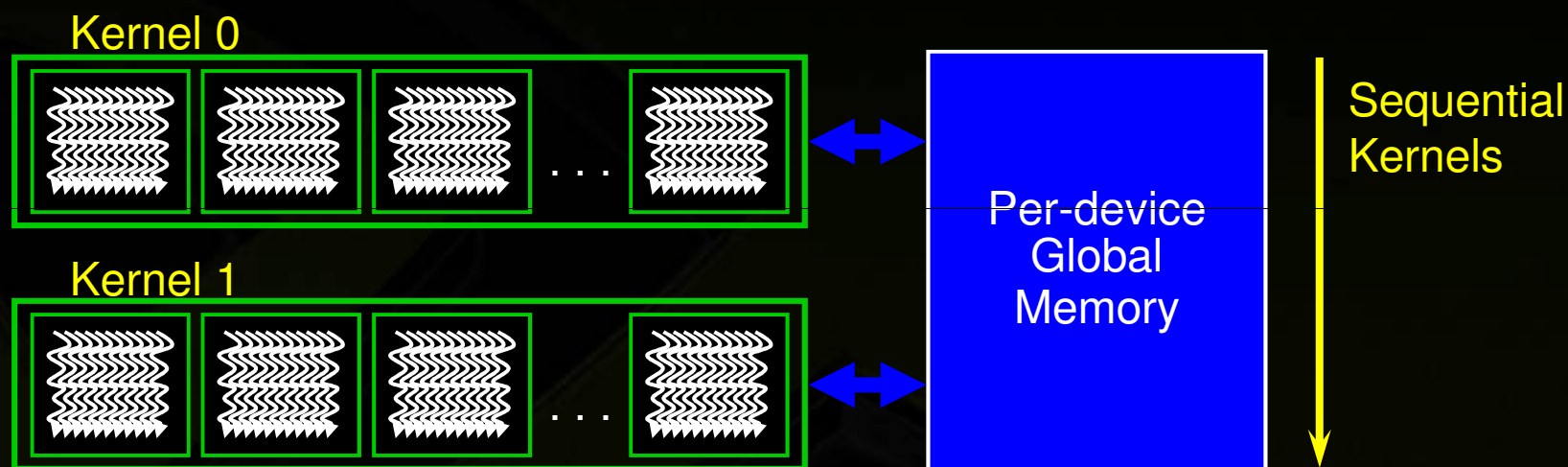
- **Local storage**
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)
  - Data lifetime = thread lifetime
- **Shared memory**
  - Each thread block has own shared memory
    - Accessible only by threads within that block
  - Data lifetime = block lifetime
- **Global (device) memory**
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation

# Memory Model Review

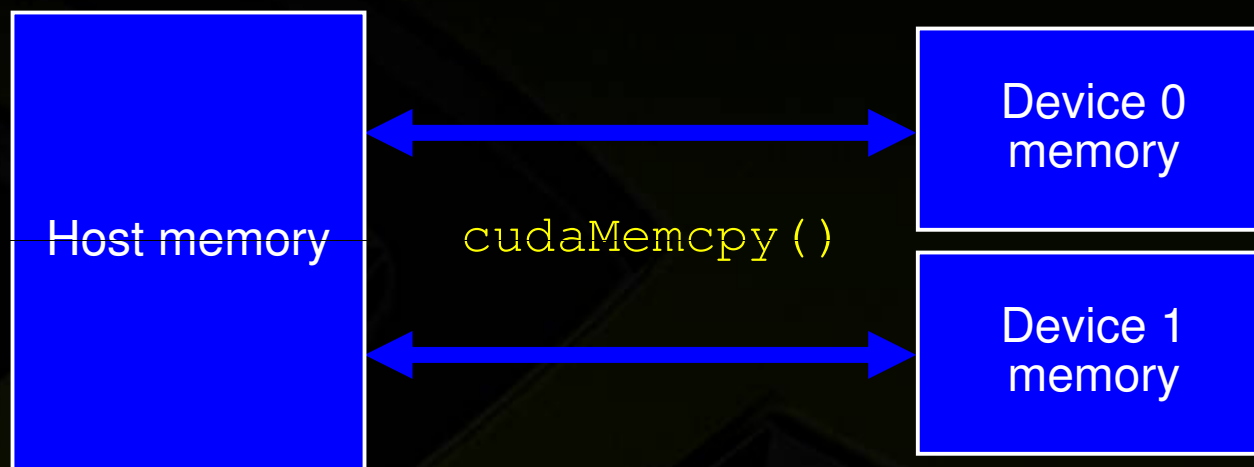


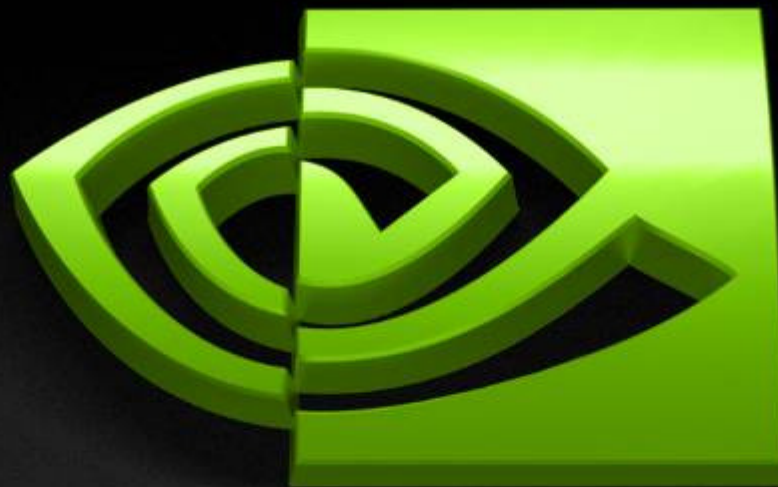


# Memory Model Review



# Memory Model Review





**NVIDIA®**

**CUDA Development Resources**

# CUDA Programming Resources



- **CUDA toolkit**
  - Compiler, libraries, and documentation
  - free download for Windows, Linux, and MacOS
- **CUDA SDK**
  - code samples
  - whitepapers
- **Instructional materials on CUDA Zone**
  - slides and audio
  - parallel programming course at University of Illinois UC
  - tutorials
  - forums

# GPU Tools



- **Profiler**
  - Available now for all supported OSs
  - Command-line or GUI
  - Sampling signals on GPU for:
    - Memory access parameters
    - Execution (serialization, divergence)
- **Debugger**
  - Currently linux only (gdb)
  - Runs on the GPU
- **Emulation mode**



# Special CUDA Developer Offer on Tesla GPUs

- 50% off MSRP on Tesla C1060 GPUs
- Up to four per developer
- Act now, limited time offer
- Visit [http://www.nvidia.com/object/webinar\\_promo](http://www.nvidia.com/object/webinar_promo)
  - *If you are outside of US or Canada, please contact an NVIDIA Tesla Preferred Provider in your country*