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

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 memory that they all use
  - Share results with each other
  - 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
Warps

- Blocks are divided into 32 thread wide units called warps
  - Size of warps is implementation specific and can change in the future
- The SM creates, manages, schedules and executes threads at warp granularity
  - Each warp consists of 32 threads of contiguous threadIds
- All threads in a warp execute the same instruction
  - If threads of a warp diverge the warp serially executes each branch path taken
- When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible

Each block (up to 512 thread) is dispatched to an SM (shader core) as a unit of work: all of its warps run in the core’s pipeline until they are all done.

Each warp (32 threads) execute in locksteps (one single program counter). SIMT = single instruction multiple threads -- same as SIMD
Hierarchy of Concurrent Threads

- Threads are grouped into **thread blocks**
- **Kernel** = grid of thread blocks

```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

**Thread Block 0**
```
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID - 1];
```

**Thread Block 1**
```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

**Thread Block N - 1**
```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

By definition, threads in the same block may synchronize with **barriers**

**Heterogeneous Memory Model**

- **Device 0 memory**
- **Device 1 memory**
- **Host memory**
- `cudaMemcpy()`
Kernel Memory Access

- **Per-thread**
  - Thread
  - Registers → On-chip
  - Local Memory → Off-chip, uncached

- **Per-block**
  - Block
  - Shared Memory
  - • On-chip, small
  - • Fast

- **Per-device**
  - Global Memory
  - • Off-chip, large
  - • Uncached
  - • Persistent across kernel launches
  - • Kernel I/O

Physical Memory Layout

- “Local” memory resides in device DRAM
  - Use registers and shared memory to minimize local memory use

- Host can read and write global memory but not shared memory
Any source file containing language extensions, like "<<< >>>", must be compiled with **nvcc**

**nvcc** is a **compiler driver**
- Invokes all the necessary tools and compilers like cudacc, g++, cl, ...

**nvcc** can output either:
- C code (CPU code)
  - That must then be compiled with the rest of the application using another tool
- PTX or object code directly

An executable requires linking to:
- Runtime library (**cudart**)
- Core library (**cuda**)
Compiling

CPU/GPU Source

NVCC

PTX Code

PTX to Target Compiler

G80 ...

GPU

Target code

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)`

```c
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

```c
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);
    cudaMemcpy((void **) &b_d, nBytes);
    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++) a_h[i] = 100.f + i;
    cudaMemcpy(a_h, a_d, nBytes, cudaMemcpyDeviceToHost);
    cudaMemcpy(b_d, b_h, nBytes, cudaMemcpyDeviceToDevice);
    assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```
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);
    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++) 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;
}
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;
}

© NVIDIA Corporation 2009
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);
    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++) a_h[i] = 100.f + i;
    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;
}

© NVIDIA Corporation 2009

Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Cannot access host memory
  - Must have void return type
  - No variable number of arguments (“varargs”)
  - Not recursive
  - No static variables

- Function arguments automatically copied from host to device

© NVIDIA Corporation 2009
Function Qualifiers

- Kernels designated by function qualifier:
  - __global__
    - Function called from host and executed on device
    - Must return void

- Other CUDA function qualifiers
  - __device__
    - Function called from device and run on device
    - Cannot be called from host code
  - __host__
    - Function called from host and executed on host (default)
    - __host__ and __device__ qualifiers can be combined to generate both CPU and GPU code

Launching Kernels

- Modified C function call syntax:
  
  \[ \text{kernel}<<<\text{dim3} \ dG, \ \text{dim3} \ dB>>>(...) \]

- Execution Configuration (“<<< >>>”)
  - \( \text{dG} \) - dimension and size of grid in blocks
    - Two-dimensional: \( x \) and \( y \)
    - Blocks launched in the grid: \( \text{dG} \cdot x \cdot \text{dG} \cdot y \)
  
  - \( \text{dB} \) - dimension and size of blocks in threads:
    - Three-dimensional: \( x, y, \) and \( z \)
    - Threads per block: \( \text{dB} \cdot x \cdot \text{dB} \cdot y \cdot \text{dB} \cdot z \)

- Unspecified \( \text{dim3} \) fields initialize to 1
More on Thread and Block 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
  - Image processing
  - Solving PDEs on volumes

Execution Configuration Examples

```plaintext
// Equivalent assignment using constructor functions

dim3 grid, block;
grid.x = 2; grid.y = 4;
block.x = 8; block.y = 16;
kernel<<<grid, block>>>(...);

dim3 grid(2, 4), block(8,16);
kernell<<<grid, block>>>(...);

kernel<<<8,1024>>>(...);
```
CUDA Built-in Device Variables

- All \texttt{__global__} and \texttt{__device__} functions have access to these automatically defined variables:
  - \texttt{dim3 gridDim;}
    - Dimensions of the grid in blocks (at most 2D)
  - \texttt{dim3 blockDim;}
    - Dimensions of the block in threads
  - \texttt{dim3 blockIdx;}
    - Block index within the grid
  - \texttt{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 (\texttt{threadIdx}) to a global ID which can be used as array indices

<table>
<thead>
<tr>
<th>blockIdx.x</th>
<th>blockDim.x = 5</th>
<th>threadIdx.x</th>
<th>blockIdx.x*blockDim.x + threadIdx.x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>5 6 7 8 9</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>2</td>
<td>10 11 12 13 14</td>
</tr>
</tbody>
</table>

© NVIDIA Corporation 2009
Minimal Kernels

```c
__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

```c
__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 1 1 2 2 2 2 2

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```
Output: 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4

Increment Array Example

**CPU program**

```c
void inc_cpu(int *a, int N)
{
    int idx;
    for (idx = 0; idx<N; idx++)
        a[idx] = a[idx] + 1;
}
void main()
{
    ...
    inc_cpu(a, N);
    ...
}
```

**CUDA program**

```c
__global__ void inc_gpu(int *a_d, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a_d[idx] = a_d[idx] + 1;
}
void main()
{
    ...
    dim3 dimBlock (blocksize);
    dim3 dimGrid(ceil(N/(float)blocksize));
    inc_gpu<<<dimGrid, dimBlock>>>(a_d, N);
    ...
}
```