CME213/ME339
Lecture 3

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

A simple example

```
// Host code
int main()
{
    size_t size = ...;
    float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
    // Allocate input vectors h_A and h_B in host memory
    h_A = (float*)malloc(size);
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
    ...
```

```
    ...
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int nthreads = 256;
    int nblocks = (N + nthreads - 1) / nthreads;
    VecAdd<<<nblocks, nthreads>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    ...
}
```

```
// Device code
__global__ void VecAdd(const float* A,
                       const float* B,
                       float* C,
                       const int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

A few keywords will get you going

- Kernel: function that executes on device (GPU) and can be called from host (CPU)
  - Can only access GPU memory
  - Not recursive
  - Must have `void` return type
  - No static variables
  - No variable number of arguments
- Functions must be declared with a qualifier:

`__global__`:	GPU kernel function launched by CPU
`__device__`:	GPU kernel called from GPU
`__host__`:	CPU function called from CPU (default) `__host__` and `__device__` qualifiers can be combined

- These qualifiers determine how functions are compiled.

Memory allocation

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

Example:

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

Copying data from anywhere to anywhere

```
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 only after the copy is complete
- Doesn't start copying until previous CUDA calls complete
- `enum cudaMemcpyKind`:
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

Three key abstractions

1. hierarchy of thread groups
2. shared memories
3. barrier synchronization

CUDA threads

Kernel: function executed on the GPU as an array of parallel threads

```
// Device code
__global__ void VecAdd(...) { ... }
```

All threads execute the same kernel code, but can take different paths.

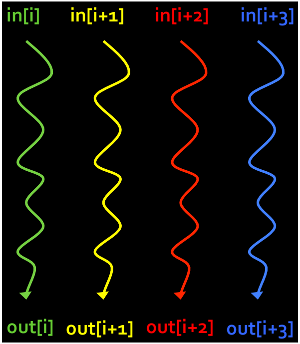
Each thread has an ID. It can be used to:

- Select input/output data
- Control decisions

CUDA threads

ID is used to:

- Select input/output data
- Control decisions

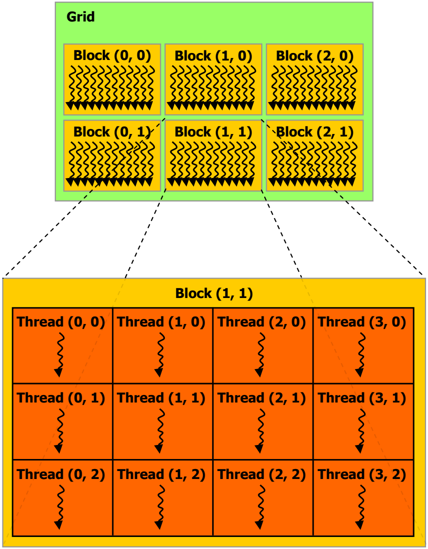



```
// Device code
__global__ void VecAdd(const float* A, const float* B,
                       float* C, const int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}
```

Many many threads

- **CPU threads:** heavyweight entities.
- The operating system must swap threads on and off CPU execution channels to provide multithreading capability.
- Context switches (when two threads are swapped) are therefore slow and expensive.
- **GPUs threads:** numerous and lightweight.
- Critical to achieve peak performance: having many threads help hide memory (and instruction) latency.

Blocks of threads and grid of blocks



Thread index

`threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a 1D, 2D or 3D thread block.

```
__global__ void MatAdd(const float* A, const float* B,
                       float* C, const int N) {
    int i = threadIdx.x; int j = threadIdx.y;
    if (i < N && j < N) C[i*N+j] = A[i*N+j] + B[i*N+j];
}
```

Grid

Blocks are organized into a 1D, 2D or 3 grid of thread blocks.

`blockIdx`: 1D, 2D or 3D index to identify a thread block.

```
__global__ void MatAdd(const float* A, const float* B,
                       float* C, const int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) C[i*N+j] = A[i*N+j] + B[i*N+j];
}
```

`blockDim`: dimension of the thread block
`gridDim`: dimension of the grid

```
int main() {
...
// Kernel invocation
dim3 nthreads(16, 16);
dim3 nblocks(N/nthreads.x, N/nthreads.y);
MatAdd<<<nblocks, nthreads>>>(A, B, C);
...
}
```

The number of threads per block and the number of blocks per grid are specified in the `<<<...>>>` syntax.

It can be of type `int` or `dim3`.

Note:

```
int nblocks = N / nthreads;
int nblocks = (N + nthreads - 1) / nthreads;
```

Example of indexing

```
__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

```
    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

```
    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

Limits

There are many limits on various quantities due to hardware limitations. We will provide a more complete list later on.

Max x-, y-, or z-dimension of a grid of thread blocks	65535
Max x- or y-dimension of a block	1024
Max z-dimension of a block	64
Max # threads per block	1024