An Overview of General Purpose Graphics Processing Units

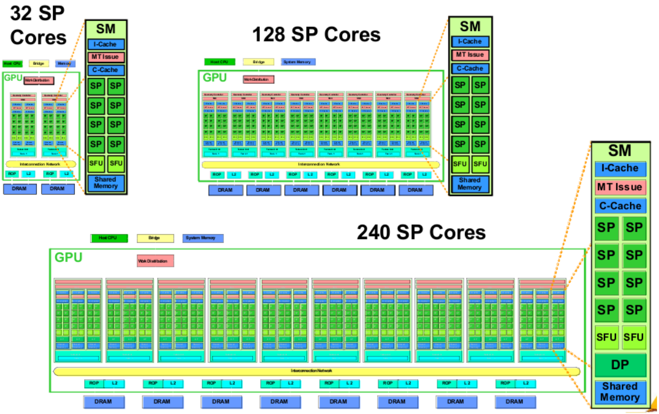
Marc Moreno Maza

University of Western Ontario, Canada

UWO
April 1st, 2014

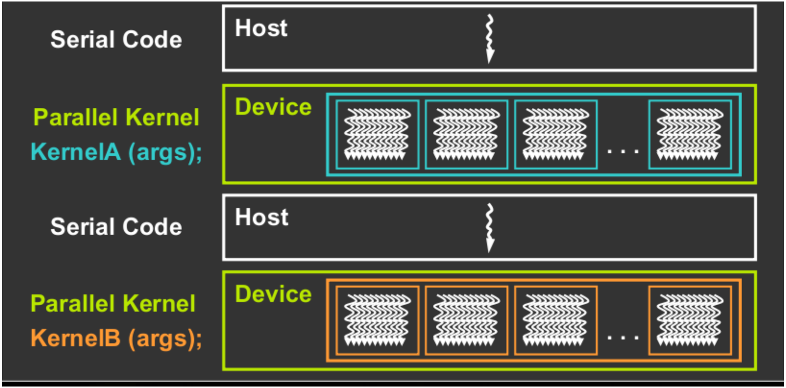
CUDA design goals

- Enable heterogeneous systems (i.e., CPU+GPU)
- Scale to 100's of cores, 1000's of parallel threads
- Use C/C++ with minimal extensions
- Let programmers focus on parallel algorithms



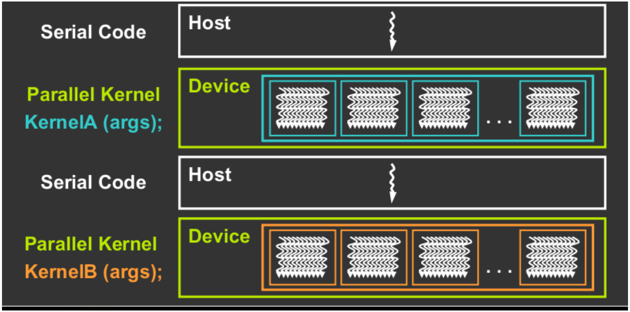
Heterogeneous programming (1/3)

- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a host (= CPU) thread
- The parallel kernel C code executes in many device threads across multiple GPU processing elements, called streaming processors (SP).



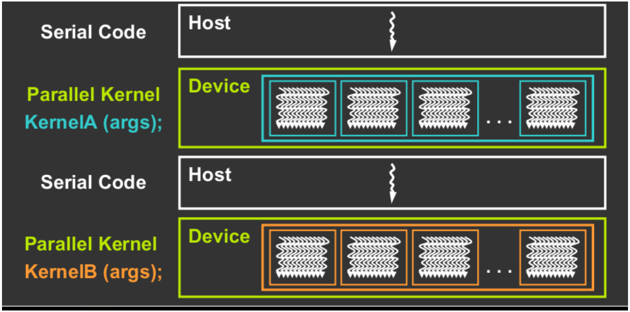
Heterogeneous programming (2/3)

- Thus, the parallel code (kernel) is launched and executed on a device by many threads.
- Threads are grouped into thread blocks.
- One kernel is executed at a time on the device.
- Many threads execute each kernel.

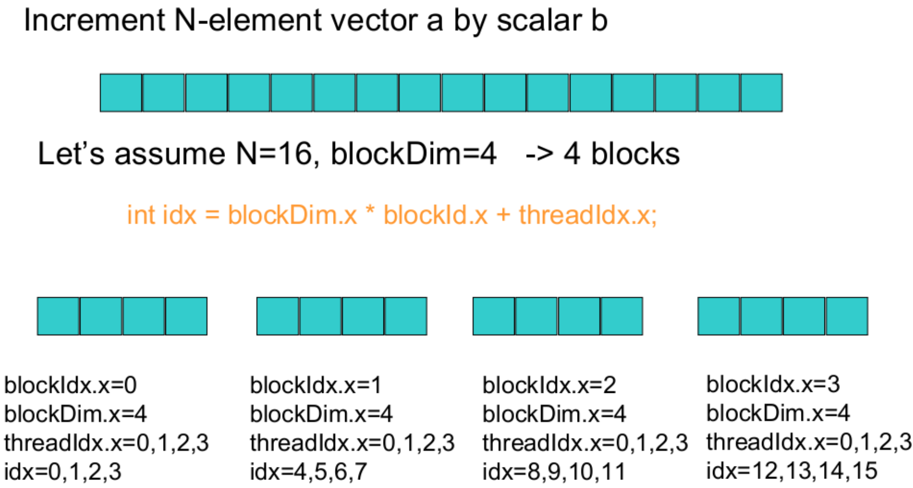


Heterogeneous programming (3/3)

- The parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in **thread and block ID variables** are used to map each thread to a specific data tile (see next slide).
- Thus, each thread executes the same code on different data based on its thread and block ID.



Example: increment array elements (1/2)



See our example number 4 in `/usr/local/cs4402/examples/4`

Example: increment array elements (2/2)

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);
}
```

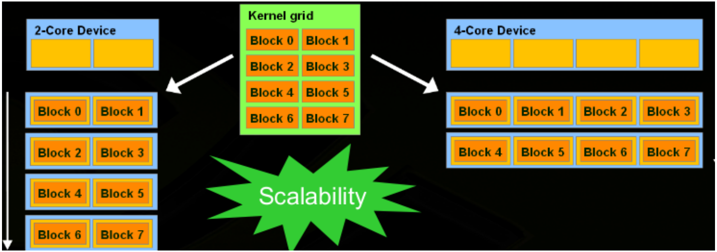
CUDA program

```
__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);
}
```

Thread blocks (1/2)

- A **Thread block** is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory
- Within a grid, **thread blocks can run in any order**:
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices



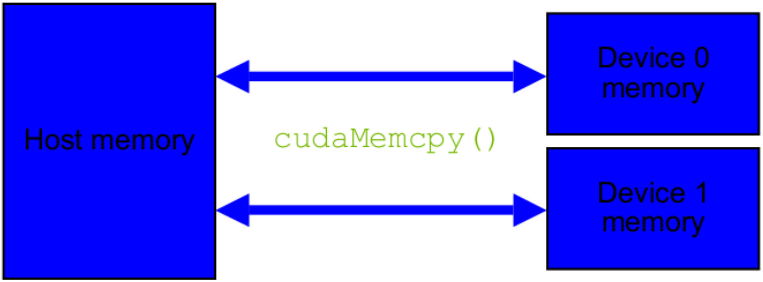
Thread blocks (2/2)

- Thus, within a grid, any possible interleaving of blocks must be valid.
- Thread blocks **may coordinate but not synchronize**
  - they may share pointers
  - they should not share locks (this can easily deadlock).
- The fact that thread blocks cannot synchronize gives **scalability**:
  - A kernel scales across any number of parallel cores
- However, within a thread block, threads may synchronize with barriers.
- That is, threads wait at the barrier until **all** threads in the **same block** reach the barrier.

Memory hierarchy (1/3)

Host (CPU) memory:

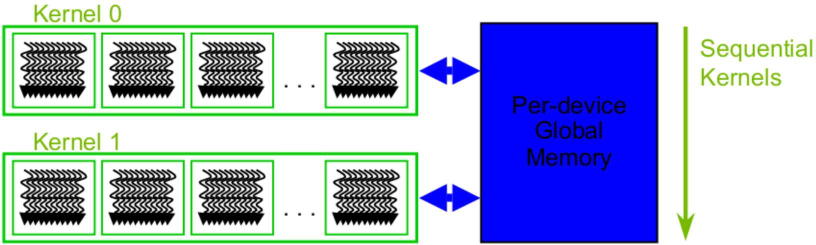
- Not directly accessible by CUDA threads



Memory hierarchy (2/3)

Global (on the device) memory:

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



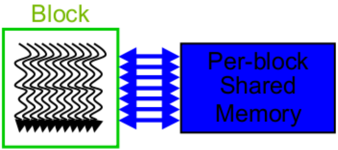
Memory hierarchy (3/3)

Shared memory:

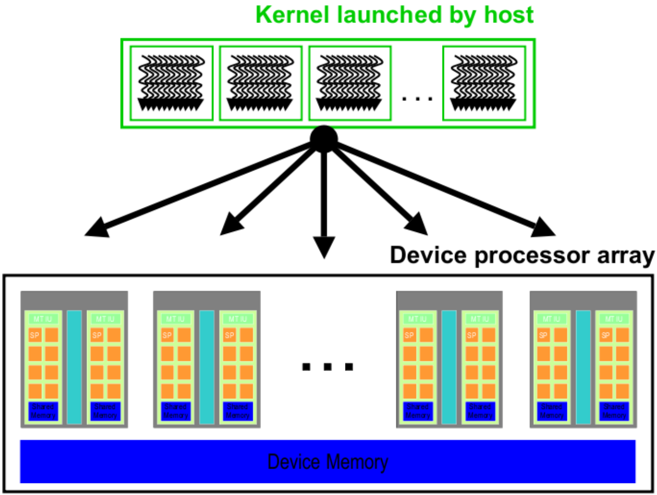
- Each thread block has its own shared memory, which is accessible only by the threads within that block
- Data lifetime = block lifetime

Local storage:

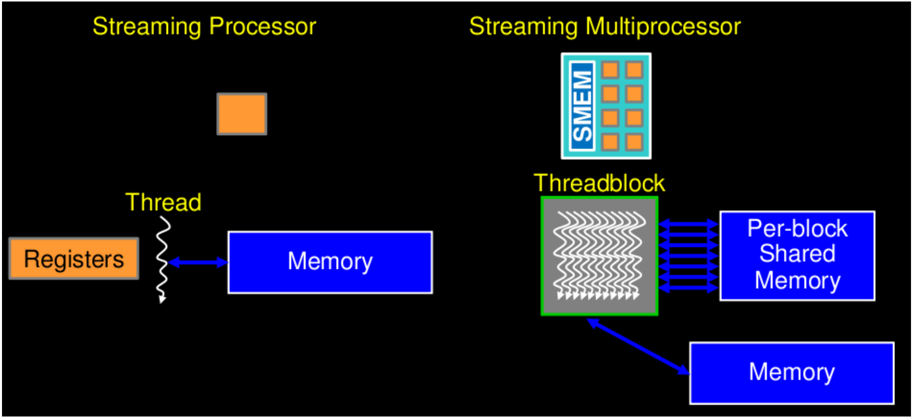
- Each thread has its own local storage
- Data lifetime = thread lifetime



Blocks run on multiprocessors

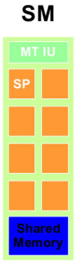


Streaming processors and multiprocessors



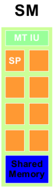
Hardware multithreading

- **Hardware allocates resources to blocks**:
  - blocks need: thread slots, registers, shared memory
  - blocks don't run until resources are available
- **Hardware schedules threads**:
  - threads have their own registers
  - any thread not waiting for something can run
  - context switching is free every cycle
- **Hardware relies on threads to hide latency**:
  - thus high parallelism is necessary for performance.



SIMT thread execution

- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a **warp**
  - The number of threads in a warp is the **warp size** (32 on G80)
  - A half-warp is the first or second half of a warp.
- Within a warp, threads
  - share instruction fetch/dispatch
  - some become inactive when code path diverges
  - hardware automatically handles divergence
- **Warps are the primitive unit of scheduling**:
  - each active block is split into warps in a well-defined way
  - threads within a warp are executed physically in parallel while warps and blocks are executed logically in parallel.



Returning to the example

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);
}
```

Example host code for increment array elements

```
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```