CSE 746 - Parallel and High Performance Computing Lecture 6 - Dynamic parallelism with CUDA

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
ppomorsk@sharcnet.ca
http://ppomorsk.sharcnet.ca/

Dynamic parallelism

- new feature available since late 2012, in GPUs with CC (Compute Capability) 3.5 or higher
- code with dynamic parallelism will not compile for devices with lower than 3.5 Compute Capability
- only some of recent NVIDIA GPU cards support CC 3.5, it is not universal

Dynamic parallelism

- permits kernel launches from inside kernels
- kernels no longer have to be launched exclusively from host
- relaxing this restriction permits more independence of GPU code from the CPU host, less device/host synchronization required
- opens possibilities for clearer, less complicated code
- true kernel recursion is now possible
- allows much more efficient parallelization for some problems
- without this GPUs lacked a feature which has been available on CPUs for decades, making GPUs harder to program compared to CPUs

Calling kernel inside kernel has same syntax as calling on host

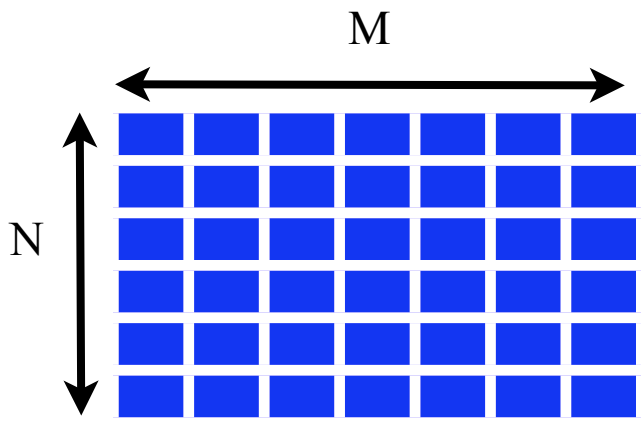
```
__global void Kernel1(){
    ...  //kernel code
    }

__global__ void Kernel2(){
     ...  //kernel code
     Kernel1<<<N,M>>>() // call kernel inside kernel
     ...  //more kernel code
    }

int main(){
    ... //host code
    Kernel2<<K,J>>() // call kernel from host
    ... //more host code
    }
```

- in this example, Kernel2 launches K*J threads when called in the main function
- each thread in Kernel2 that calls Kernel1 will launch N*M new threads
- up to K*J*N*M threads in total can be launched in this case

Program easy to write as an efficient GPU kernel

```
for (int i=0; i<N){
   for (int j=0; j<M;j++){
          some_convolution_function(i,j);
   }
}
```

- perfect target for 2D thread decomposition
- N*M threads will be launched
- all threads call the same function



Program harder to write as an efficient GPU kernel

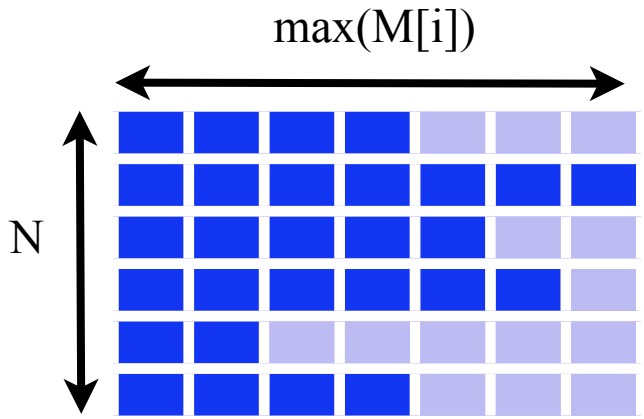
```
for (int i=0; i<N){
   for (int j=0; j<M[i];j++){
          some_convolution_function(i,j);
   }
}
```

- the i,j values no longer fit a regular grid
- cannot decompose straightforwardly into 2D array of threads
- can consider a series of bad options

More threads?

```
__global oversubscribed_kernel(){
  i=threadIdx.x + blockDim.x * blockIdx.x;
  j=threadIdx.y + blockDim.y * blockIdx.y;
     if(j<M[i]){
            some_convolution_function(i,j);
     }
  }
```

- could try oversubscription, have N*max(M[i]) threads, some do no work
- this is wasteful, threads which do nothing have to be created



Fewer threads?

```
__global void fewer_threads_kernel(){
     i=threadIdx.x + blockDim.x * blockIdx.x;
        for (int j=0; j<M[i];j++){
              some_convolution_function(i,j);
          }
  }
```

- could have just N threads
- in this case all threads would do work, but N might be too small to create enough threads for efficient use of GPU
- different threads would likely have very different amounts of work to do

Multiple kernels from host

```
__global void multi_kernel(i){

  j=threadIdx.x + blockDim.x * blockIdx.x;
            some_convolution_function(i,j);
  }

//on host
...
for(i=0;i<N;i++){
   multi_kernel<<<(enough threads to handle M[i] evaluations)>>>(i)
}
```

- could call kernel from host N times
- there is overhead to calling a kernel, if N is large, overhead will be significant
- default kernels execute in series, previous must finish before next one starts, even if independent
- could use streams to avoid this, but number of streams working asynchronously is limited

New option with CC 3.5: more kernels from GPU

```
__global void convolution_kernel(int i,int j){
    some_convolution_function(i,j);
    }
__global void dynamic_parallelism_kernel(int *M){

    for(j=0;j<M[blockIdx.x];j++){
            convolution_kernel<<1,1>>>(blockIdx.x,j);
        }
    }
//on host

    ...
    dynamic_parallelism_kernel<<<N,1>>>(M)
```

- first kernel call creates N blocks with 1 thread each, so N threads
- each thread i of the N threads launches M[i] kernels, each containing one thread
- another possibility: have each thread i launch a kernel with M[i] threads, but work done by each kernel not equal

New option with CC 3.5: more kernels from GPU

```
__global void convolution_kernel(int i,int j){
  some_convolution_function(i,j);
  }

__global void dynamic_parallelism_kernel(int *M){
  for(j=0;j<M[blockIdx.x];j++){
            convolution_kernel<<1,1>>>(blockIdx.x,j);
     }
  }
//on host
...
dynamic_parallelism_kernel<<<N,1>>>(M)
```

- overhead of launching kernels on GPU much lower than on host
- these kernels will execute asynchronously, as they are launched by different threads

Synchronization is an issue, just like on host

```
__global__ void Kernel2(){
...  //kernel code
    Kernel1<<<N,M>>>(); // thread will launch kernel and keep going
    cudaDeviceSynchronize(); // make thread wait for Kernel1 to complete
...  //code that needs data generated by Kernel1
}
```

- threads will launch kernels and keep on going, just like the host
- if you need for the thread to wait until Kernel1 is finished before continuing, use cudaDeviceSynchronize just like on host
- in this case cudaDeviceSynchronize ensures synchronization only within that 1 GPU thread (!!)

Synchronization is an issue, just like on host

```
__global__ void Kernel2(){
  ...  //kernel code
    if(threadIdx.x==0){
       Kernel1<<<N,M>>>(); // thread will launch kernel and keep going
       cudaDeviceSynchronize(); // make thread wait for Kernel1 to complete
       }
    __syncthreads();  // if all threads in block need Kernel1 to complete
   ...  //code that needs data generated by Kernel1
}
```

- cudaDeviceSynchronize will not synchronize threads in a block, it only works within thread when inside kernel
- if you want all threads in a block to wait for kernel to finish, used cudaDeviceSynchronize() followed by __syncthreads

CC 3.5 GPU card now available at SHARCNET

available on machine: cat.sharcnet.ca

CUDA module not loaded by default, so execute:

module load cuda

Two cards are mounted, it's the second one (card #1) that has the CC 3.5 capability

To make only that card visible to CUDA, execute:

export CUDA_VISIBLE_DEVICES=1

Compile with:

nvcc -arch=sm_35 --relocatable-device-code true -lcudadevrt -O2 program.cu

Exercise - enhance binary reductio code from Lecture 4 using dynamic paralleism

Start code can be found in:

/home/ppomorsk/CSE746_lec6/Reduction

Goal is to implement 3 level binary reduction using just one kernel function, and have that function called on host only once (with following kernel calls inside the kernel)

Proceed in stages, following instructions in comments at the start of each stage of the process