CUDA

Politecnico di Milano
Seminar Room A. Alario
18 November, 2015

Antonio R. Miele
Marco D. Santambrogio
Politecnico di Milano
Introduction

• CUDA (Compute Unified Device Architecture) has been introduced in 2007 with NVIDIA Tesla architecture

• C-like language to write programs that are able to use GPU as a processing resource to accelerate functions

• CUDA’s abstraction level closely match the characteristics and organization of NVIDIA GPUs
Structure of a program

- A CUDA program is organized in
  - A serial C code executed on the CPU
  - A set of independent kernel functions parallelized on the GPU
Parallelization of a kernel

- The set of input data is decomposed into a stream of elements
- A single computational function (kernel) operates on each element
  - “thread” defined as execution of kernel on one data element
- Multiple cores can process multiple threads in parallel
- Suitable for data-parallel problems
CUDA hierarchy of threads

- A kernel executes in parallel across a set of parallel threads
  - Each thread executes an instance of the kernel on a single data chuck

- Threads are grouped in thread blocks

- Thread blocks are grouped in a grid

- Blocks and grids are organized as an N-dimensional (up to 3) array
  - IDs are used to identify elements in the array
CUDA hierarchy of threads

- CUDA hierarchy of threads perfectly match with NVIDIA 2-level architecture
  - A GPU executes one or more kernel grids
  - A streaming multiprocessor (SM) executes concurrently one or more interleaved thread blocks
  - CUDA cores in the SM execute threads
  - The SM executes threads of the same block in groups of 32 threads called a warp
CUDA device memory model

- There are mainly three distinct memories visible in the device-side code:
  - per-Thread Private Local Memory
  - per-Block Shared Memory
  - per-Application Context Global Memory
CUDA device memory model

- Matching between the memory model and the NVIDIA GPU architecture
Execution flow

1. Copy input data from CPU memory to GPU memory
Execution flow

2. Load GPU program and execute
Execution flow

3. Transfer results from GPU memory to CPU memory
Example of CUDA kernel

- Compute the sum of two vectors
- C program:

```c
#define N 256

void vsum(int* a, int* b, int* c){
    int i;
    for (i=0; i<N; i++){
        c[i]=a[i]+b[i];
    }
}

void main(){
    int va[N], vb[N], vc[N];
    ...
    vsum(va, vb, vc);
    ...
}
```
Example of CUDA kernel

- CUDA C program:

```c
#define N 256

__global__ void vsumKernel(int* a, int* b, int *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

void main() {
    ...
    dim3 blocksPerGrid(N/256,1,1); //assuming 256 divides N exactly
    dim3 threadsPerBlock(256,1,1);

    vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(va,vb,vc);
    ...
}
```
Example of CUDA kernel

• CUDA C program:

```c
#define N 256

__global__ void vsumKernel(int* a, int* b, int* c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

void main()
{
    ...
    dim3 blocksPerGrid(N/256,1,1); //assuming 256 divides N exactly
    dim3 threadsPerBlock(256,1,1);

    vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(va,vb,vc);
    ...
}
```
Example of CUDA kernel

**CUDA C program:**

```c
#define N 256

__global__ void vsumKernel(int* a, int* b, int *c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

void main(){
    ...
    dim3 blocksPerGrid(N/256,1,1); //assuming 256 divides N exactly
    dim3 threadsPerBlock(256,1,1);
    vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(va,vb,vc);
    ...
}
```

- **Execute the kernel**
- **Number of blocks in the grid**
- **Number of threads in each block**
Data transfer

• GPU memory is separate from CPU one
  – GPU memory has to be allocated
  – Data have to be transferred explicitly in GPU memory

```c
void main(){
    int *va;
    int mya[N];  // to be populated
...  
cudaMalloc(&va, N*sizeof(int));
cudaMemcpy(va, mya, N*sizeof(int), cudaMemcpyHostToDevice);

...  // execute kernel

cudaMemcpy(mya, va, N*sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(va);
}
```
Host/device synchronization

- Kernel calls are non-blocking. This means that the host program continues immediately after it calls the kernel
  - Allows overlap of computation on CPU and GPU
- Use `cudaThreadSynchronize()` to wait for kernel to finish

```c
vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
//do work on host (that doesn’t depend on c)
cudaThreadSynchronise(); //wait for kernel to finish
```

- Standard `cudaMemcpy` calls are blocking
  - Non-blocking variants exist
- It is also possible to synchronize threads within the same block with the `__syncthreads()` call
Another example of CUDA kernel

- Sum of matrices: 2D decomposition of the problem

```c
__global__ void matrixAdd(float a[N][N], float b[N][N], float c[N][N])
{
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    c[i][j] = a[i][j] + b[i][j];
}

int main()
{
    dim3 blocksPerGrid(N/16,N/16,1); // (N/16)x(N/16) blocks/grid (2D)
    dim3 threadsPerBlock(16,16,1); // 16x16=256 threads/block (2D)
    matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
}
```
References

• NVIDIA website: