Introduction to GPU computing using CUDA

Felice Pantaleo

felice@cern.ch

Set your environment up

Connection to the machine

Before we start, you need to connect to your CUDA-enabled machine.

You can access the nodes from `frontend.recas.ba.infn.it` using the login data provided, and

```
qsub -q bigmpi2@sauron.recas.ba.infn.it -I
```

You are now connected to the machine you will be working on.

Check that CUDA is installed properly

Once you are connected, check that your environment is correctly configured to compile CUDA code by running:

```
$ module load cuda
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2015 NVIDIA Corporation
Built on Tue_Aug_11_14:27:32_CDT_2015
Cuda compilation tools, release 7.5, V7.5.17
```

Copy and extract the archive that contains the exercises in your home directory:

```
$ wget www.cern.ch/felice.pantaleo/intro_cuda/exercises.tar.gz
$ tar -xzvf exercises.tar.gz
```

Compile and run the `deviceQuery` application:

```
$ cd utils/deviceQuery/
$ make
$ ./deviceQuery
```

You can get some useful information about the features and the limits that you will find on the device you will be running your code on. For example:

```
./deviceQuery Starting...
CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 4 CUDA Capable device(s)
Device 0: "Tesla C2070"
  CUDA Driver Version / Runtime Version          5.5 / 5.5
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 5375 MBytes (5636554752 bytes)
  (14) Multiprocessors x ( 32) CUDA Cores/MP:    448 CUDA Cores
  GPU Clock rate:                                1147 MHz (1.15 GHz)
  Memory Clock rate:                             1494 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535),
3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x
2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >
```

- Some of you are sharing the same machine and some time measurements can be influenced by other users running at the very same moment. It can be necessary to run time measurements multiple times.
- http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

Exercise 1. CUDA Memory Model

In this exercise you will learn what heterogeneous memory model means, by demonstrating the difference between host and device memory spaces.

1. Allocate device memory;
2. Copy the host array `h_a` to `d_a` on the device;
3. Copy the device array `d_a` to the device array `d_b`;
4. Copy the device array `d_b` to the host array `h_a`;
5. Free the memory allocated for `d_a` and `d_b`.
6. Compile and run the program by running:

```
$ nvcc cuda_mem_model.cu -o ex01
$ ./ex01
```

- Bonus: Can you do that using Thrust?
- Bonus: Measure the PCI Express bandwidth.

Exercise 2. Launch a kernel

By completing this exercise you will learn how to configure and launch a simple CUDA kernel.

1. Allocate device memory;
2. Configure the kernel to run using a one-dimensional grid of one-dimensional blocks (using only x dimension);
3. Each GPU thread should set one element of the array to

```
d_a[i] = blockIdx.x + threadIdx.x;
```

4. Copy the results to the host memory;
5. Check the correctness of the results

Exercise 3. Two-dimensional grid

M is a matrix of NxN integers.

1. Set N=4
2. Write a kernel that sets each element of the matrix to its linear index (e.g. M[2,3] = 2*N + 3), by making use of two-dimensional grid and blocks.
3. Copy the result to the host and check that it is correct.
4. Try with a rectangular matrix 19x67. Hint: check the kernel launch parameters.

Exercise 4. Measuring throughput and profiling with NVVP

The throughput of a kernel can be defined as the number of bytes **read and written** by a kernel in the unit of time.

The CUDA event API includes calls to create and destroy events, record events, and compute the elapsed time in milliseconds between two recorded events.

CUDA events make use of the concept of CUDA streams. A CUDA stream is simply a sequence of operations that are performed in order on the device. Operations in different streams can be interleaved and in some cases overlapped, a property that can be used to hide data transfers between the host and the device. Up to now, all operations on the GPU have occurred in the default stream, or stream `0` (also called the "Null Stream").

The peak theoretical throughput can be evaluated as well: if your device comes with a memory clock rate of 1GHz DDR (double data rate) and a 256-bit wide memory interface, the peak theoretical throughput can be computed with the following:

$$\text{Throughput } (GB/s) = \text{Memory\_rate}(Hz) * \text{memory\_interface\_width}(byte) * 2 / 10^9 = 64 GB/s$$

1. Compute the theoretical peak throughput of the device you are using:

2. Modify ex04.cu to give the measurement of actual throughput of the kernel:

3. Measure the throughput with a decreasing number of elements (in logarithmic scale).

   Before doing that write down what do you expect (you can also draw a diagram):

4. What did you find out? Can you give an explanation?

5. NVIDIA Visual Profiler can deliver vital feedback for optimizing your CUDA applications.

Run it and analyze `ex04`.

```
$ nvvp &
```

Exercise 5. Parallel Reduction

Given an array `a[N]`, the reduction sum `Sum` of `a` is the sum of all its elements: `Sum=a[0]+a[1]+...a[N-1]`.

1. Implement a block-wise parallel reduction (using the shared memory).
2. For each block, save the partial sum.
3. Sum all the partial sums together.
4. Check the result comparing with the host result.
5. Measure the throughput of your reduction kernel using CUDA Events (see exercise 4):

6. Analyze your application using `nvvp`. Do you think it can be improved? How?

- Bonus: Can you implement a one-step reduction? Measure and compare the throughput of the two versions.
- Bonus: The cumulative sum of an array `a[N]` is another array `b[N]`, which contains the sum of prefixes of `a`:

`b[i] = a[0] + a[1] + … + a[i]`. Implement a cumulative sum kernel assuming that the size of the input array is a friendly multiple of the block size.

Cheatsheet

Measuring time using CUDA Events

```
cudaEvent_t start, stop; float time;
cudaEventCreate(&start);  cudaEventCreate(&stop);
cudaEventRecord(start, 0);
myFirstKernel <<< n_blocks, block_size >>> (d_a);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
printf ("Time for the kernel: %f ms\n", time);
```

Function attributes

- `__global__` function called by the host, executes on the device
- `__device__` function called by the device, executes on the device
- `__host__` function called by the host, executes on the host
- `__host__ __device__` generates both host and device code for the function

Variables attributes

- `__device__` variable on device (Global Memory)
- `__shared__` variable in Shared Memory
- `__restrict__` restricted pointers, assert to the compiler that pointers are not aliased
- `No qualifier` automatic variable, resides in Register or in Local Memory

Built-in Variables

- `dim3 gridDim` size of the grid in number of blocks along the x, y, z axes
- `dim3 blockDim` size of the block in number of threads along the x, y, z axes
- `dim3 blockIdx` position (x,y,z) of the block in the grid
- `dim3 threadIdx` position (x,y,z) of the thread in the block

Shared memory

- `__shared__ int x[10];` statically allocated array in shared memory
- `extern __shared__ int x[];` dynamically allocated array in shared memory
  - `kernel<<<blocks, threadsperblock, dyn shared mem in bytes>>>`

Memory Management

- `cudaMalloc(&dptr, size)` allocates `size` memory on the device
- `cudaFree(dptr)` frees `size` memory from the device
- `cudaMallocHost(&hptr, size)` allocates `size` pinned memory on the host
- `cudaFreeHost(hptr)` frees `size` memory from the host
- `cudaMemcpy(trgptr, srcptr, size, direction)` copies `size` memory from the source pointer to the target pointer using the direction specified (e.g. `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`)