CUDA programming basics

Michal Kreps

WARWICK
THE UNIVERSITY OF WARWICK

Content

WARWICK
THE UNIVERSITY OF WARWICK

- Parallelising your code
- Basic CUDA kernel to run on GPU
- Memory management
- NVIDIA GPU architecture and how to make code efficient
- Stride loop on GPU
- Getting information about resources used
- Exercises

- Disclaimer: Not replacement for other professional training, just my own take to get you started
- Go through these slides and then everybody can go at own pace and ask questions when they arise

Good and bad for parallelism

- Typically loops over big arrays are good for parallel execution
  - Idea is that same calculation is done on different elements of array at the same time
  - Requires that we get right data and right instructions at the right time
  - What usually works best if different “threads” do not depend on results from other threads
- Often speed will depend on actual details of the HW executing calculation
  - What might be best for CPU is not necessarily best for GPU
  - Sometimes one might want to even redesign things to get best performance
- There is also balance between amount of work which needs to be done and how quickly HW can serve data
  - Easier to get large gains on problems bound by computing rather then memory bandwidth

Executing code on GPU

WARWICK
THE UNIVERSITY OF WARWICK

- I will be talking about using NVIDIA GPU for parallel execution
- Our desktops have NVIDIA graphical card with some CUDA capabilities, so we can use them to play with ideas and develop
- Need kernel which looks like `__global__ void cudaKernel(args);`
  - Return type is fixed
  - Arguments have to be copied, no reference (more later)
- Typical Hello world kernel could look like

```
__global__ void helloWorld() {
    printf("Hello World\n");
}
```

Executing code on GPU

WARWICK
THE UNIVERSITY OF WARWICK

- Need main program, which executes given kernel

```
int main( int, char** ) {
    helloWorld<<<1, 8>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

- Here in <<<>>> brackets we have number of blocks and number of threads per block
  - We will explain these later, for now it is enough to know that we will run 1 block with 8 threads, so we should see 8 “Hello World” messages on output

Compiling code

WARWICK
THE UNIVERSITY OF WARWICK

- Save code to helloWorld.cu
  - All CUDA code uses .cu files (and .cuh for header files)
- Setup environment on desktop

```
module load CUDA/11.7.0
./cvmfs/sft.cern.ch/lcg/views/LCG_105/x86_64-el9-gcc11-opt/setup.sh
nvcc -g -O2 -arch=sm_61 -o helloWorld helloWorld.cu
```

  - arch argument specifies for which GPU code should be generated
  - arch argument can be omitted and it will pick up capabilities of GPU on given machine
- nvcc is kind of a wrapper, which adds some header files, defines couple of variables and deals with kernel invocation
- This setup is one I found working where I can use all usual SW from SFT view combined with CUDA

NVIDIA GPU understanding

WARWICK
THE UNIVERSITY OF WARWICK

- To get best out of the GPU, one has to have some understanding how they work
- Each GPU has some defined number of CUDA cores along with fixed number of streaming multiprocessors (SM)
  - One in my desktop has 4 SMs
- Each SM has 4 warp schedulers
- Warp is smallest unit, typically 32, for which all threads in the warp execute same instruction
  - Best performance is gained when working with full warps as there are no idling bits
- Typically SM should have enough warps to handle that it can hide latency
  - While one warp is being executed on scheduler, data for other warps are being prepared
  - Number of warps is given by number of threads and number of blocks for given kernel
  - I think one typically needs number of SMs times number of warp schedulers per SM times maybe times 3-4 warps to be able to get most out of GPU

NVIDIA GPU understanding

WARWICK
THE UNIVERSITY OF WARWICK

- Be aware of branching
- Modern CPU do clever tricks to hide calculation of which side of the branch to take by evaluating both branches in parallel while deciding which path code takes
  - This helps to speed to up things
- GPUs are simpler, they do not use such tricks
- GPUs in addition treat whole warp as single entity
  - All threads in warp evaluate same instruction
- If there is branching, GPU will evaluate two branches sequentially
  - All threads in warp taking first path and only then all threads taking second path
  - Effectively all threads in warp have to spend time needed for both branches

Memory management

WARWICK
THE UNIVERSITY OF WARWICK

- The CPU and GPU memory are separate entities, what is in host memory cannot be accessed by GPU and vice versa
- This is reason why arguments of CUDA kernel has to copy values
- To allocate memory use (for array of integers of given size)

```
int N = 2 << 20;
size_t bytes = N*sizeof(int);
int* a;
cudaMallocManaged( &a, bytes );
```

- This memory is accessible by both CPU and GPU
  - When it is accessed, there is check whether it is available and if not, it is copied between host and device

Memory management

WARWICK
THE UNIVERSITY OF WARWICK

- Letting memory access fail and then copy can be inefficient, if one knows that large chunk will be needed, one can prefetch by `cudaMemPrefetchAsync( a, bytes, deviceId );`
  - After this call, your code will continue, but on the background memory will be copied between host and device
  - `deviceID` is ID of the given GPU or `cudaCPUDeviceId` constant for host
- At the end when memory is not needed, it has to be freed by `cudaFree( a );`

Stride loops

WARWICK
THE UNIVERSITY OF WARWICK

- Want to parallelise loop

```
for ( int=0; i<N; ++i ) {
   a[i] = value;
}
```

- Suppose we want to execute on M threads (M<N) and as each element is independent of each other, we can just split to chunks with each thread processing N/M elements
- For this in our CUDA kernel we need to know which thread and block we are in and block size

```
threadIdx.x  // variable telling which thread within block this is
blockIdx.x   // variable telling which of the blocks we are processing
blockDim.x  // number of threads in the block
gridDim.x   // number of blocks
```

Stride loops

WARWICK
THE UNIVERSITY OF WARWICK

- Want to parallelise loop

```
for ( int=0; i<N; ++i ) {
    a[i] = value;
}
```

- There is no unique way of splitting loop into chunks
- There is good way for GPU in which all threads within warp process neighbouring elements
  - Driven by the way how memory access is done
-

```
int indexInGrid = threadIdx.x +blockIdx.x *blockDim.x;
int stride = gridDim.x * blockDim.x;
for ( int i = indexInGrid; i<N; i+=stride ) {
    a[i] = value;
}
```

Execution information

WARWICK
THE UNIVERSITY OF WARWICK

- You can get some information of what was executed and how long it took by `nsys` command

`nsys profile --stats=true -f true -o test ./laplaceGPU`

```
CUDA Kernel Statistics:

 Time (%)  Total Time (ns)  Instances  Avg (ns)   Med (ns)   Min (ns)  Max (ns)  StdDev (ns)                  Name
 --------  ---------------  ---------  ---------  ---------  --------  --------  -----------  ----------------------------------------
     97.2   31,538,883,388    200,500  157,301.2  157,121.0   151,105   418,625      3,377.1  void newIteration<float>(T1 *, T1 *, int)
      2.8      917,116,552      2,005  457,414.7  455,363.0   401,697   532,673      9,022.6  void maxDev<float>(T1 *, T1 *, T1 *, int)
      0.0        1,418,660          2  709,330.0  709,330.0   675,842   742,818     47,359.2  void init<float>(T1 *, T1, int)

[7/8] Executing 'gpumemtimesum' stats report

CUDA Memory Operation Statistics (by time):

 Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)             Operation
 --------  ---------------  -----  --------  --------  --------  --------  -----------  ---------------------------------
     63.5        6,814,465  2,004   3,400.4   2,048.0     1,696    14,624      3,457.3  [CUDA Unified Memory memcpy HtoD]
     36.5        3,911,134  2,007   1,948.7   1,664.0     1,280   175,137      5,420.7  [CUDA Unified Memory memcpy DtoH]

[8/8] Executing 'gpumemsizesum' stats report

CUDA Memory Operation Statistics (by size):

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)           Operation
 ----------  -----  --------  --------  --------  --------  -----------  ---------------------------------
     12.407  2,007     0.006     0.004     0.004     2.097        0.066  [CUDA Unified Memory memcpy DtoH]
      8.208  2,004     0.004     0.004     0.004     0.004        0.000  [CUDA Unified Memory memcpy HtoD]
```

GPU streaming

WARWICK
THE UNIVERSITY OF WARWICK

- GPU allows to have several concurrent streams
- To create stream and execute kernel in it

```
cudaStream_t stream;
cudaStreamCreate(&stream); // create stream
someKernel<<<number_of_blocks, threads_per_block, 0, stream>>>();
cudaStreamDestroy(stream); // destroy stream
```

- Few rules
  - Kernels in a given streams are processed in order they are put in
  - Order of operations in non-default stream is not fixed, idea is that they can be executed in parallel if there are enough resources
  - Default stream is blocking, it waits with execution until other streams are done and it blocks other streams until it is done

Functions on GPU

WARWICK
THE UNIVERSITY OF WARWICK

- Machine code for host CPU and device GPU is different
- Functions need to be compiled for each side where it is going to be used
- To instruct compiler to generate code for GPU
  `__device__ <type> function(<args>)`
- To instruct code generation for CPU (can be omitted)
  `__host__ <type> function(<args>)`
- One can have also function which can be executed on both CPU and GPU
  `__device__ __host__ <type> function(<args>)`

Exercises

WARWICK
THE UNIVERSITY OF WARWICK

- Following are couple of exercises which you can try to do
- All relevant files can be downloaded from https://cernbox.cern.ch/s/rrVLApERLepBdL3
- I have put up also my solutions to some of the problems for reference, but I strongly suggest you first try by yourself
- If you get stuck with something, come to me and ask

Exercise 1

WARWICK
THE UNIVERSITY OF WARWICK

- Login to your desktop (or other computer with NVIDIA graphics card)
- Use `nvidia-smi` and find out what card you have
- Try to find out capabilities of your card (google can help)
- My desktop has Quadro P620 which has 512 CUDA cores, 4 SMs and 2 GB of memory with bandwidth of 80.13 GB/s

Exercise 2

WARWICK
THE UNIVERSITY OF WARWICK

- Collect code for Hello world exercise and compile it
- Once compiled, execute it and check that you see 8 Hello World messages
- Adapt code such that it also prints block and thread ids
- You can try to see what happens if you change number of threads per block and number of blocks

Exercise 3

WARWICK
THE UNIVERSITY OF WARWICK

- It is useful to algorithmically decide on number of threads and number of blocks for your kernel
- Need to find out what are available device capabilities
- Start from code

```
int main() {
  cudaDeviceProp properties;
  int deviceId;
  int numberOfSMs;
  cudaGetDevice(&deviceId);
  cudaDeviceGetAttribute(&numberOfSMs, cudaDevAttrMultiProcessorCount,
deviceId);
  cudaGetDeviceProperties( &properties, deviceId);
}
```

- Print number of SMs, warp size and maximum number of threads
- Check what other information is available

Exercise 4

WARWICK
THE UNIVERSITY OF WARWICK

- Write a code, which
  - creates two large arrays (suggest at least 1M elements)
  - Initialises all elements in each array to some value (all elements to be same, but can be different for the two arrays)
  - Adds two arrays together (c[i]=a[i]+b[i]) on GPU
- In conjunction with next exercise you can try to play with
  - block size and number of blocks
  - initialise arrays on CPU and do sum on GPU with and without prefetching
  - initialise arrays on GPU and you can try to execute them in parallel in different streams
    - You will probably not see any real difference on GPU we have in desktops but you should be able to see what is going on in exercise 6

Exercise 5

WARWICK
THE UNIVERSITY OF WARWICK

- Take code from previous exercise and run
  `nsys profile --stats=true -f true -o test <your executable>`
- Inspect output and find in it information about
  - Kernels execution
  - Data transfers between host and device
- In previous exercise I suggested some variations so inspect those with nsys and see what helps performance and what hurts it

Exercise 6

WARWICK
THE UNIVERSITY OF WARWICK

- There is also possibility to get some visual representation on what is executed when
- On command line start `nsight-sys`
- In “Options Preset” window select “GPU Rows on Top” and say OK
- Use File → Open and choose `test.nsys-rep` file
- Under CUDA HW try to find your kernels, see in which order they are executed and identify data transfers
  - You can try initialisation of two arrays in the same stream and in different non-default streams and check whether you can see difference

n-body problem

WARWICK
THE UNIVERSITY OF WARWICK

- Good test problem is n-body simulation like moving of many bodies due to gravitational force
- In our exercise we will make life simple by having all of the bodies same mass, so we can ignore mass completely
- Rather than having to write everything from scratch, download `nbodyCPU.cc` along with `initialized*dat` files
- You can compile this by `nvcc` and execute to try CPU version
  - By default it runs with 4096 bodies, adding argument 15 will run with 65536 bodies
- Adapt code to run on GPU and see what speedup you can get
  - You will need to rethink how you store data
  - Do not try to do everything at the start, concentrate on the most time consuming part
  - When you are done, you can check against `solution*dat` files whether you get same result

Laplace equation

WARWICK
THE UNIVERSITY OF WARWICK

- Solve Laplace's equation $\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$
- Simple way is to use relaxation method for which

$$\phi(x,y) = \frac{1}{4}\left[\phi(x+a,y) + \phi(x-a,y) + \phi(x,y+a) + \phi(x,y-a)\right]$$

- We solve it with boundary condition that potential is V=1V on one side of the square and 0 on remaining three sides (and setting a=1)
- You can start from python or C++ implementation I have, depending how much you want to write by yourself
- I have `plotLaplace.py` script which takes filename as a single argument, which plots result

Laplace equation

WARWICK
THE UNIVERSITY OF WARWICK

- You should get something which looks like plot here
- This one can get tricky in a sense that it does very little calculation so one can make it easily slow
  - Think about structure of loop in kernel
  - Finding out whether to terminate is not trivial, think about sensible algorithm and whether you need to do it after each iteration
  - To find out largest deviation you are searching for reduction algorithm to be run on GPU
    - If you do not find one on web, I have put possible implementation to `maxGPU.cuh` to find maximum in a given block
- I managed 1m5s on CPU with grid size 512x512 and 33s on GPU with grid size 1024x1024