CUDA Tutorial 03

The basics of getting CUDA devices to do what you want them to

Martin A. Kruse - June 2013 / Februrary 2014

The procedure to communicate and give instructions to the CUDA device is very simple. Just think of the device as a second, slightly different computer inside your host computer. In the last lesson, we used a new feature of CUDA 6.0, which is called *unified memory*. While this is a very convenient concept, it hides the fact that GPU and CPU are using quite different memories. The memory for the GPU is stored on the graphics card whereas the memory for the CPU is stored on the mainboard of your computer. They are connected via the PCI-E-Bus, so you can push data from one to the other, but be aware that this has to be done also when using unified memory, even if you as programmer do not see that.

Memory transfer might very well be what is limiting the processing speed. This is in fact what limits most of my own programs, so learning and understanding the concept of two seperate memories should be worth your time. That is even more true so as it is not very complicated. Just try to minimize communication between host (CPU) and device (GPU).

If you just want to try out some idea you might have, go on and use unified memory. It most probably speeds up things significantly and frees up time for you to find your solution instead of you waiting for the program to finish (which, I know, many of you do from time to time).

If you need an even more efficient program you might want to go the (not much) harder way of addressing device and host memory seperately. This is the procedure followed during the remainder of the lessons, because I prefer working that way .

In this lesson you will be doing the same as in the previous one, just with seperate addressing of device and host memory. As I wrote this lesson before the last one, some things you will already know.

The basic idea of any CUDA program is very straighforward:

1. Prepare some data in your host (your computer) / prepare unified memory
2. Copy that data to the CUDA device (your graphics card)
3. Tell the device to manipulate that data
4. Copy the results back to the host
5. Do some more stuff to the results

With unified memory, you simply skip steps 2 and 4.

This tutorial will show you how to accomplish all of the steps mentioned above. With this knowledge, you have everything at hand to create your own CUDA accelerated programs. Of course there is much more to know about programming efficiently with CUDA, though compared to understanding and mastering the five steps above, everything else is of minor importance. Also it helps alot to know about how the hardware in the CUDA devices looks like and how this hardware processes your program, though even that is not absoluteley necessary.

During this tutorial, we will create an array of numbers (step 1 in the listing above), copy these from the RAM in the host computer to the RAM on the CUDA device (step 2), increase each of these numbers by one (the much underpowered step 3), copy these manipulated numbers back to the host RAM (step 4) and check, if the device did everything correctly (step 5).

This program serves no other purpose than to show you how to communicate with the CUDA device. It will be even slower than if you did the manipulation with the CPU as you would do in any “normal” computer program, though acceleration of processing time is not the goal of this session. In any valuable program, you will use step 3 to let the device handle the parts of your program which will take too much time using just your CPU. The identification of what the CUDA device is good at and what should better be done by the CPU is the real art behind programming with CUDA, though I hope you will gain some idea of that by working with these tutorials.

1 A (somewhat) useful program

Create a folder and in it a file named “main_simple.cu”. Paste the following program in it:

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>

// this is the program that is to be run on the device for a
// large number of threads, in our example 100
// each thread takes care of one entry in the number array,
// so in order for the thread to know which number to manipulate,
// a scheme has to be utilized in order to assign each thread a
// unique number

__global__ void processKernel(int *numberArray, int N)
{
  // this is the assignment of a unique identifier.
  // blockIdx.x is the unique number of the block, in which the
  // thread is positioned, blockDim.x holds the number of threads
  // for each block and threadIdx.x is the number of the thread in
  // this block.
  int idx = blockIdx.x*blockDim.x + threadIdx.x;

  // this tells the thread to manipulate the assigned number in
  // the array stored in device memory
  if (idx<N)
    numberArray[idx] = numberArray[idx] + 1;
}

int main(int argc, const char* argv[] )
{
  int numberOfNumbers = 100;

  // declare some arrays for storing numbers
  int *numbers1_h, *numbers2_h, *numbers_d;

  // define the dimensions for the device program to work upon
  // we need only one block with N threads per block (so N threads total)
  // to process our data
  int numberOfBlocks = 1;
  int threadsPerBlock = numberOfNumbers;
  int maxNumberOfThreads = numberOfNumbers;

  // reserve (allocate) some space for numbers in (h)ost memory
  numbers1_h = (int*) malloc(sizeof(int)*numberOfNumbers);

  // reserve some space for resulting values in (h)ost memory
  numbers2_h = (int*) malloc(sizeof(int)*numberOfNumbers);

  // reserve some working space for the numbers in device memory
  cudaMalloc((void **) &numbers_d, sizeof(int)*numberOfNumbers);

  // fill the input array (in host memory) with some numbers (step 1)
  for(int i=0;i<numberOfNumbers;i++)
    {
      numbers1_h[i] = i;
    }

  // copy these input numbers to the device (step 2)
  cudaMemcpy(numbers_d, numbers1_h, sizeof(int)*numberOfNumbers, cudaMemcpyHostToDevice);

  // tell the device to do its magic (step 3)
  processKernel<<<numberOfBlocks, threadsPerBlock>>>(numbers_d, maxNumberOfThreads);
  // wait for the device to finish working
  cudaDeviceSynchronize();

  // copy results back to host RAM (step 4)
  cudaMemcpy(numbers2_h, numbers_d, sizeof(int)*numberOfNumbers, cudaMemcpyDeviceToHost);

  // check if the device did what it was told to do (step 5)
  int workedCorrectly = 1;
  for(int i=0;i<numberOfNumbers;i++)
    {
      if (numbers1_h[i] + 1 != numbers2_h[i])
    workedCorrectly = 0;
    }

  if (workedCorrectly == 1)
    printf("The␣device␣performed␣well!\n");
  else
    printf("Something␣went␣wrong.␣The␣output␣numbers␣are␣not␣what␣was␣to␣be␣expected...\n");

  // free the space that has been used by our arrays so that
  // other programs might use it
  free(numbers1_h);
  free(numbers2_h);
  cudaFree(numbers_d);

  return 0;
}
```

So what is happening here? At first, let us take a look at the function “main”, where the program will start when executed: We reserve (or allocate) some memory for later use by the commands malloc. We need two such arrays in host memory. The first to hold some numbers, the second to receive the results from the CUDA device after it has processed the input. There has to be space for the numbers in device memory as well, which is assured by the somewhat ugly looking instruction cudaMalloc(...). We want 100 numbers to be processed.

Then we need to define the structure in which the processing threads are to be organised. You do not need to worry about this now, we just need 100 threads and for simplicity they are grouped into one block.

After filling the array with some arbitrary numbers, the transfer to device memory can commence. The Memcpy-instruction needs the destination, source, number of bytes to be transfered and the direction of the transfer (Host to Device) in that order.

When the transfer is complete, the host tells the device to begin manipulating the data by invoking the kernel, which has to be declared using the ___global___ keyword. The numbers in the pointed brackets tell the device, how many blocks of threads and how many threads per block to utilize. Until now, there has been one sequential thread working on the host to execute the instructions given by your program. Now there are 100 threads working on the CUDA device in parallel, each executing the instructions given by the function “processKernel”. While these threads work on the device, the CPU could do some other stuff, but in our example it just waits for the device to finish manipulating the numbers. After the manipulation of the data in device memory has concluded, the results are copied back to the host (direction Device to Host) and finally compared with what we would expect.

So far so good, but does it work? Save the file, open a terminal, navigate to your project folder and type

```
  nvcc -c main_simple.cu -arch=sm_10
  gcc main_simple.o -o main_simple
```

to generate an executable of your program. Now run it by typing

```
./main_simple
```

The output will tell you, that the device performed well, as was to be expected. Isn't that great? You just pushed 100 numbers from your RAM to your graphics card, told it to add one to each and every number and then grapped the results and found out, that the graphics card did exactly what you told it to do. Just like that. You are now capable of doing all kinds of (independent) calculations by simply following the five steps: Data preparation, data copy to device, manipulation by device, data copy from device, evaluation / further manipulation by the CPU. Go ahead, try something funny. And when you are finished, keep on reading the following suggestions.

2 The structure of your program

So far, all instructions have been accumulated in one file, independent of CUDA or host instructions. There is nothing wrong with that, though in order to keep larger projects readable and maintainable, CUDA instructions and host instructions should be seperated. In all of the (hopefully soon) following tutorials, it will be necessary to manually create three files for each program. One file is called "main.cpp" and contains your "normal" host program, which at some time calls one or more functions from the second file called "cuda_wrapper.cu". The purpose of this second file is to manage communication between the CUDA device and the host computer.

In addition, the main-part of the program does not have to "know" anything about how the CUDA device is to be adressed. It simply tells the cuda_wrapper to do that kind of "dirty" work. The file that takes care of the communication with the device has to be compiled by NVIDIAs own compiler with the designation nvcc (as has been done above), whereas the main.cpp file is to be compiled using the standard C/C++-compiler (here gcc).

For the simple tutorials presented here, this two-file-approach is propably overdoing things, though the paradigm will be helpful in larger and more interesting projects.

The third file neglected so far is the Makefile, which takes care of compiling and linking your files. It is invoked by the command

```
make
```

which you propably already used during the last tutorial sessions. If you are not familiar with Makefiles: No need to worry. Your interaction with these will be very limited. Though I recommend getting familiar with them, as they are very powerfull and can save you a lot of tedious typing.

3 The program

Create a folder (or use the one from the section above) and create the three files "main.cpp", "cuda_wrapper.cu" and "Makefile". The main program will be compiled from the file "main.cpp", so thats where we start. Paste the following program in your "main.cpp":

```
#include <stdio.h>
#include <stdlib.h>

extern void cuda_doStuff(int *array_in, int *array_out, int N);

int main( int argc, const char* argv[] )
```

```
{

  int numberOfNumbers = 100;

  int *numbers1_h, *numbers2_h;

  numbers1_h = (int*) malloc(sizeof(int)*numberOfNumbers);
  numbers2_h = (int*) malloc(sizeof(int)*numberOfNumbers);

  for(int i=0;i<numberOfNumbers;i++)
    {
      numbers1_h[i] = i;
    }

  // let the wrapper take care of communication with the device
  cuda_doStuff(numbers1_h, numbers2_h, numberOfNumbers);
  // now the data is manipulated without having to take care of
  // all the CUDA stuff here in this file. Nice, isn't it?

  int workedCorrectly = 1;
  for(int i=0;i<numberOfNumbers;i++)
    {
      if (numbers1_h[i] + 1 != numbers2_h[i])
    workedCorrectly = 0;
    }

  if (workedCorrectly == 1)
    printf("The␣device␣performed␣well!\n");
  else
    printf("Something␣went␣wrong.␣The␣output␣numbers␣are␣not␣what␣was␣to␣be␣expected...\n");

  free(numbers1_h);
  free(numbers2_h);

  return 0;
}
```

and the following into "cuda_wrapper.cu":

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>

__global__ void processKernel(int *numberArray, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;

  if (idx<N)
    numberArray[idx] = numberArray[idx] + 1;
}

extern void cuda_doStuff(int *array_in, int *array_out, int N)
{
  int *numbers_d;

  int numberOfBlocks = 1;
  int threadsPerBlock = N;
  int maxNumberOfThreads = N;

  cudaMalloc((void **) &numbers_d, sizeof(int)*N);

  cudaMemcpy(numbers_d, array_in, sizeof(int)*N, cudaMemcpyHostToDevice);
  processKernel<<<numberOfBlocks, threadsPerBlock>>>(numbers_d, maxNumberOfThreads);
  cudaDeviceSynchronize();
  cudaMemcpy(array_out, numbers_d, sizeof(int)*N, cudaMemcpyDeviceToHost);

  cudaFree(numbers_d);

  return;
}
```

The “Makefile” should look something like this:

```
CC=g++
LINKER_DIRS=-L/usr/local/cuda-6.0/lib64
LINKER_FLAGS=-lcudart -lcuda
NVCC=nvcc
CUDA_ARCHITECTURE=10
OCELOT=

all: main main_simple

main: main.o cuda_wrapper.o
    $(CC) main.o cuda_wrapper.o -o main $(LINKER_DIRS) $(LINKER_FLAGS) $(OCELOT)

main.o: main.cpp
    $(CC) main.cpp -c -I .

cuda_wrapper.o: cuda_wrapper.cu
    $(NVCC) -c cuda_wrapper.cu -arch=sm_$(CUDA_ARCHITECTURE)

main_simple:
    $(NVCC) -c main_simple.cu -arch=sm_$(CUDA_ARCHITECTURE)
    $(CC) main_simple.o -o main_simple $(LINKER_DIRS) $(LINKER_FLAGS) $(OCELOT)

clean:
    rm -f *.o main main_simple
```

The function cuda_doStuff, serves as our interface to the CUDA world. You simply call it from within your main program, give it all the parameter it needs, and all the communication with the CUDA device is done from within it. Your main program stays clear of the CUDA calls. Of course there are some restrictions, e.g. in this case the parameter N has to be smaller than 1024 (see Table 1 of Tutorial00). If you need manipulation of larger chunks of data, you would have to reorganise your kernel, but that is another topic. Your only real hard constraint is the amount of memory on the device (6GB on the GTX Titan in Prometheus), though even that can be circumvented by successively pushing data to and from the device.

Now you have all the neccessary information to get going with your own projects. There are, however, more aspects to CUDA programming than what I have shown you so far. More involved CUDA tutorials can be found easily on the web, and if you have come this far, you should have no problem coping with what you find out there. If you got questions on a specific topic, please come by and ask. If you desire a tutorial on some topic, I will try and construct something for you. For now, if you wish to go on learning CUDA stuff, I recommend reading about registers and shared memory. So long.