CUDA Tutorial 02

Your first CUDA program

Martin A. Kruse - Februrary 2014

The full series of tutorials can be found online at
http://www.ieap.uni-kiel.de/et/people/kruse/,
though access to this site later than 2026 might
be impossible due to thermonuclear war.

Suggestions and corrections are very welcome at kruse@physik.uni-kiel.de.

Before diving into the specifics of how to use your graphics card efficiently, this session is meant to show you just how simple getting started with CUDA really is. With the advent of CUDA 6.0, even more barriers standing in the way of beginners fall. If you are familiar with any programming language like C, getting CUDA devices to work is ridiculously easy.

In this tutorial, we will program a function that creates an array of numbers and than increments each number in that array. This might not be a very useful utilization of computing power, but it depicts how CUDA devices work and what purposes they are able to fulfill.

1 How to build programs

As in all tutorial session, the source code presented here can be found as an archive in the header of this document. You may download and build it using the Makefile or you copy (and adjust to your liking) the source code from this document and follow the build instructions.

2 The really simple version

Please consider the following source code:

```
#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 incrementArrayViaCUDAdevice(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 and increment it
    if (idx<N)
        numberArray[idx] = numberArray[idx] + 1;
}

// this is the "normal" function to be run on the CPU
// it does the exact same thing as the CUDA function above
void incrementArray(int *numberArray, int N){

    // go through every number in the array consecutively
    // and increment it
    for(int i=0; i<N; ++i)
    {
        numberArray[i] = numberArray[i] + 1;
    }
}

int main(int argc, const char* argv[] )
{
    // some arbitrary array length
    int numberOfNumbers = 100;
```

```
    // declare some arrays for storing numbers
    int *numbers1, *numbers2;

    // reserve (allocate) some working space for the numbers in device memory
    cudaMallocManaged(&numbers1, sizeof(int)*numberOfNumbers);
    cudaMallocManaged(&numbers2, sizeof(int)*numberOfNumbers);

    // fill the input array with some numbers
    for(int i=0;i<numberOfNumbers;i++)
    {
        numbers1[i] = i;    // this will be manipulated by the CUDA device (GPU)
        numbers2[i] = i;    // this will be manipulated by the CPU (as any standard C program would
            do)
    }

    // tell the device (GPU) to do its magic
    incrementArrayViaCUDAdevice<<<1, numberOfNumbers>>>(numbers1, numberOfNumbers);

    // wait for the device to finish working
    cudaDeviceSynchronize();

    // compute the same function "normally" on the CPU
    incrementArray(numbers2, numberOfNumbers);

    // check if the GPU did the same as the CPU
    bool workedCorrectly = true;
    for(int i=0;i<numberOfNumbers;i++)
    {
        if (numbers1[i] != numbers2[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
    cudaFree(numbers1);
    cudaFree(numbers2);

    return 0;
}
```

As in all C programs, execution starts at the main function. After defining an array length (numberOfNumbers = 100), we tell the CUDA framework that we would like to have two arrays (numbers1 and numbers2) in *unified memory*. Unified memory is a construct introduced in CUDA 6.0, which simply states that the *graphics processing unit* (GPU - your graphics card) as well as the *central processing unit* (CPU) shall have access to the entries stored in it. Do not bother about this for now, the next tutorial will tell you more about it. For now, just know that this command (cudaMallocManaged) gives you some space in memory to work with.

Now we have to initialize our arrays with some number so we can start working on them. The loop just fills the arrays with incrementing numbers.

The task to fulfill is to increment each and every number in the arrays, so we tell the GPU to do that with the array numbers1 and the CPU do do the same with array numbers2.

Note that the function calls are very similar. The CUDA function (also called a *kernel*) needs some more information, which is passed to it inside the «<»>-brackets, namely the number of blocks we wish to invoke (here: 1) and the number of threads per block (here: numberOfNumbers = 100). Again, do not bother right now, just use one block with as many threads as you like.

If you take a look at the top of the file you will find the two functions we just called. The CUDA version (the kernel) is marked with the keyword ___global___. Note that both these functions are very similar. The CPU version contains a for-loop, which iterates through all the elements **consecutively**.

You will notice that the CUDA kernel only manipulates **one** element of the array, namely that at position idx. This is because the kernel is invoked numberOfNumber(100) times, once for each element. This means that for each element, one *thread* is utilized to do the work. The first line in the kernel tells the thread which element in the array it has to work with. The identifiers blockIdx.x, blockDim.x and threadIdx are given by the CUDA framework an tell the thread, in which block it is positioned, how wide the block is and which position inside the block it occupies, respectively.

After performing operations on both our arrays, the main function checks if both arrays are identical and tells you, if that is indeed true.

So let's go on an see how this program performs.

3 Compile and run the program

Save the source code from the previous section in a file called "main_really_simple.cu", open a terminal and navigate to the folder where you stored it. As we have two different processing units involved, we need two compilers in order to build our programs. The "normal" C/C++-compiler on Unix-like systems is called gcc (Gnu Compiler Collection). As for CUDA devices, NVIDIA created a compiler called nvcc (NVIDIA CUDA compiler). The following two lines create our program from source code:

```
nvcc -c main_really_simple.cu -arch=sm_20
gcc main_really_simple.o -o main_really_simple -L/usr/local/cuda-6.0/lib64 -lcudart -lcuda
```

Or, if you downloaded the archive and extracted it, you might use make (by typing exactly that in the terminal).

The only thing left to do is to run that program:

```
./main_really_simple
```

And that's it! You just passed beyond the most complicated aspect of developing CUDA programs: You convinced yourself that it might be worth looking at.

Now give yourself a pat on the shoulder and go on to the next lesson. Or try changing the value of numberOfNumbers and run the program again. For which numbers does it work? When does it fail? Why is that? Hint: I told you in the very first lesson...