Universidade de Aveiro

Mestrado Integrado em Engenharia de Computadores e Telemática

Arquitectura de Computadores Avançada

Lesson 8: GPU – Programming with CUDA

Academic year 2015/2016 Nuno Lau / José Luís Azevedo

The computational power of GPUs is, currently, much higher than that of general purpose CPUs. In order to take advantage of that computational power, and using the latest programming capabilities of GPUs, several tools have been developed that allow the use of GPUs to solve general purpose tasks. One of those tools is CUDA (Compute Unified Device Architecture) which is a parallel computing platform and programming model developed by NVIDIA to be used with their GPUs.

In this lesson we start by editing and executing small programs that demonstrate the basics about the CUDA programming model. A Makefile is provided (which may have to be changed) and hence just enter "make" to compile your programs. The resulting binary code can then be executed normally. Executing the program in this way can, however, be dangerous to the system integrity (leading to unrecoverable system crashes) whenever the code has memory access errors, e.g., as a result of badly initialized pointers. Therefore, during the development phase, it is highly recommended to run the program using the "**`cuda-memcheck`**"[1] command which provides information about possible memory access errors and, more important, prevents the system from crashing.

1. Extract the archive file **`licaoCUDA.tgz`** and work on the tasks that follow.

   1.1. Open the file arrad1.cu (using vim) and examine the CUDA program which adds 2 arrays of float. Identify which code will run on the host and which code will be executed in the GPU. Compile the program and execute it (**`cuda-memcheck arrad1`** or **`./arrad1`**). Analyze the output results and check if they are correct.

   1.2. Change the program **`arradd2.cu`** in order to execute 2 threads per block. Compile and run the program and check the correctness of the produced results. Change again the program in order to execute 4 threads per block. Check the results.

   1.3. Change the program **`arradd3.cu`** in order to use two-dimensional blocks of 2 rows and 2 columns and a one-dimensional grid. Compile and run the program and check that the results are as expected. Change again the program to use two-dimensional blocks of 4 rows and 2 columns (2 x 4, i.e., "x" dimension is 2, and "y" dimension is 4). Check the results.

[1] "Accurately identifying the source and cause of memory access errors can be frustrating and time-consuming. CUDA-MEMCHECK detects these errors in your GPU code and allows you to locate them quickly. CUDA-MEMCHECK also reports runtime execution errors, identifying situations that could otherwise result in an “unspecified launch failure” error when your application is running." Citation from https://developer.nvidia.com/cuda-memcheck, last visited in November/2015.

1.4. Change the program **`arradd4.cu`** in order to use two-dimensional blocks of 2 rows and 2 columns and a two-dimensional grid (assume that the variables A, B and D represent matrices with **`NROWS`** rows and **`NCOLS`** columns). Compile and run the program again and check the correctness of the produced results.

2. Increase the dimension of the matrix to 4096 rows and 1024 columns (i.e. 1024 in "x" and 4096 in "y"), keeping the block dimension as 2x2. Measure the execution time in the GPU and in the host, considering that:

   2.1. Threads with consecutive "ids" represent adjacent values in the rows of the matrix.

   2.2. Threads with consecutive "ids" represent adjacent values in the columns of the matrix.

3. Repeat the previous exercise with the following block dimensions: 2x4, 8x4, 16x32, 32x32, 2x256, 2x512 and 1x512. For all cases verify the correctness of the produced results and draw conclusions.

4. Implement a new program which identifies the prime elements of a matrix with 8192x8192 integer values. The result should be an integer matrix of 8192x8192 in which the positions of the detected primes (in the first matrix) are set to the value 1 and the other positions set to the value 0. Compare the execution times in the host and in the GPU.

5. Implement a program in CUDA to multiply two matrices, A and B. The result should be a matrix C, set with the appropriate dimension. Tip: consider that the number of threads to execute is equal to the dimension of the result matrix.