Programming Massively Parallel Processors Exercise 1

TECHNISCHE UNIVERSITÄT DARMSTADT

TU Darmstadt, WS 2014
André Schulz <andre.schulz@gris.tu-darmstadt.de>

Introduction

Your task is to implement a simple matrix multiplication as shown in the lecture. A naive implementation is sufficient. The purpose of this task is to get you acquainted with CUDA. A basic code skeleton is provided in this zip-file which serves as a guideline. You're free to make modifications to the code skeleton as you see fit. Please use CMake, which is installed on the lab machines to compile your CUDA code.

Prerequisites

If you want to work with your own computer, you will need a CUDA-capable graphics card and the following software:

- C++ compiler (GCC or Visual C++)
- CMake 2.8 or higher (`http://www.cmake.org`)
- CUDA Toolkit 5.0 or higher (`https://developer.nvidia.com/cuda-downloads`)

Task 1

First, you need to create a new CMake project which will help you compile the exercise. Fill the `CMakeLists.txt` file with the following:

```
project(matrixMul)
cmake_minimum_required(VERSION 2.8)

find_package(CUDA REQUIRED)

cuda_add_executable(matrixMul
        matrix.cpp
        matrixMul_cpu.cpp
        matrixMul_gpu.cu
        matrixMul.cpp)
```

This tells CMake that the project we're going to work on is called `matrixMul`, that we need CUDA, and which files are to be compiled into a CUDA executable called "matrixMul". `cuda_add_executable` enables the NVIDIA compiler to compile all ".cu" files containing GPU code. To compile and run the code, you now execute the following commands in the source directory:

```
cmake .
make
./matrixMul
```

Task 2

Determine which CUDA-capable devices are installed in your computer by implementing a function which enumerates the available CUDA devices. The function should print the following information to the console for each device:

- Compute capability
- Multiprocessor count
- GPU clock rate

- Total global memory
- L2 cache size

You can find the appropriate functions in the ”Device Management” section of the CUDA runtime API documentation. Further information about the compute capability is available in appendix A of the ”NVIDIA CUDA Programming Guide” or the online CUDA GPU list at `https://developer.nvidia.com/cuda-gpus`.

Task 3

Write a CPU implementation of the matrix multiplication in `matrixMul_cpu.cpp`. Then use it and the functions from `matrix.cpp` to compute the result matrix on the CPU in `matrixMul.cpp`.

Task 4

Write a CUDA implementation of the matrix multiplication in `matrix.cpp` and `matrixMul_gpu.cu`. Then use the functions to compute the result matrix on the GPU in `matrixMul.cpp`. Your program should be able to process matrices of arbitrary dimensions (i.e. not only power of 2) up to grid dimensions. Your program should do the following:

1. Set the CUDA device
2. Allocate necessary CPU and GPU memory (use `cudaMallocPitch()` for GPU memory)
3. Fill the input matrices with the `rand()` function
4. Transfer input matrices to the GPU
5. Multiply matrices on the GPU
6. Transfer the result matrix back to the CPU
7. Compare the result matrix against the CPU version. Where do the differences come from?

A macro which you can use for checking the return values of CUDA API calls is given in `common.h`. In the same file there is a function called `divUp()` which comes in handy when computing the number of thread blocks in a grid for the execution configuration. You can find the appropriate functions in the ”Memory Management” section of the CUDA runtime API documentation.
Hint: As a first test you could multiply two identity matrices to see whether your algorithm performs as expected or not.
Hint 2: Most of the boilerplate code is given in the lecture slides.

Task 5

To see whether your CUDA or CPU implementation runs faster, you need to add a few lines to your code. For timing the CPU code, you can use the `SysGetTime_ms()` function in `timer.h`. To measure the GPU time, you can use CUDA’s event management functions by wrapping your kernel invocation with the following code. Compare your results.

```
cudaEvent_t evStart, evStop;
cudaEventCreate(&evStart);
cudaEventCreate(&evStop);

cudaEventRecord(evStart, 0);
// kernel execution here
cudaEventRecord(evStop, 0);
cudaEventSynchronize(evStop);

float elapsedTime_ms;
cudaEventElapsedTime(&elapsedTime_ms, evStart, evStop);
printf("CUDA processing took: %f ms\n", elapsedTime_ms);

cudaEventDestroy(evStart);
cudaEventDestroy(evStop);
```

Modalities

For the exercises, group work is not allowed. The program should compile and run on any workstation in the Linux lab (Windows is not allowed). Please submit your solution (an archive of your project directory) using Moodle. The archive should have the following naming convention: `PMPP14_ex1_name.zip` (substitute ”name” with your name). The deadline is Oct 28th, 11:39am.