Chapter 4: GPU programming

Getting started with CUDA

All normal programs that we usually write in C++ or other commonly used programming languages are executed by the CPU. To use the GPU, we will need to explicitly communicate with the GPU, upload our program to the GPU, upload our input data to the GPU, instruct the GPU to run the program, wait for it to finish, and fetch the results back from the GPU to the main memory.

There are many programming environments that we can use to achieve this. We will use here CUDA, which is the native programming interface of NVIDIA GPUs, but we will also show later how to do the same thing using OpenCL, which is a cross-platform programming interface that is supported by multiple vendors.

Different APIs may look at first very different from each other, but the differences are fortunately superficial. The underlying key concepts are near identical, and if we do not use too fancy features of the APIs, we can often port code between CUDA and OpenCL in a rather mechanical manner.

CUDA toolkit

To use CUDA, we need a computer with an NVIDIA GPU and CUDA Toolkit. Then we can write programs in the "CUDA language", which is in essence C++ with some additional features in the core language, plus lots of new library functions related to GPU programming. We can compile such a program with `nvcc` and run it.

Note that the `main` function of your program is still executed by the CPU! Merely compiling something with `nvcc` does not make any difference as such, unless you write code that explicitly asks the GPU to do some part of the work. There is no speedup that you get for free.

Programming model

The key primitive in a CUDA program is called "launching a kernel". Here a "kernel" is simply some C++ function that we would like to run on the GPU very many times.

Computation is organized hierarchically in "blocks" of "threads": we will have to specify the number of blocks and the number of threads per block.

Here is a simple example of a complete CUDA program that does nothing — but in a massively parallel manner. We ask the GPU to launch 100 blocks of threads, each with 128 threads:

```
__device__ void foo(int i, int j) {}

__global__ void mykernel() {
    int i = blockIdx.x;
    int j = threadIdx.x;
    foo(i, j);
}

int main() {
    mykernel<<<100, 128>>>();
    cudaDeviceSynchronize();
}
```

If we save this code in a file called `example.cu` and compile it with `nvcc example.cu -o example`, it should produce a program that we can run with `./example`. It does absolutely nothing visible, but if we run it under NVIDIA Profiler with `nvprof ./example`, we can see some GPU activity.

What will happen is that the GPU will run the following 12800 function calls in some order, most likely in a parallel fashion:

```
f(0,  0), f(0,  1), f(0,  2), ..., f(0,  127),
f(1,  0), f(1,  1), f(1,  2), ..., f(1,  127),
                    ...,
f(99, 0), f(99, 1), f(99, 2), ..., f(99, 127).
```

Let us now dissect the program:

- In the `main` function, we ask the GPU to launch the kernel called `mykernel`. The strange angle bracket notation indicates that this is a kernel launch. We specify that we want 100 blocks of threads, and each block should have 128 threads. In total, we will run the kernel 12800 times. The second line with `cudaDeviceSynchronize()` just waits for the GPU to finish.
- The `mykernel` function looks a lot like a normal C++ function. However, the attribute `__global__` indicates that this is indeed a kernel. Kernels are special; they can access structures `blockIdx` and `threadIdx` to find what is the index of the current block (here 0–99), and what is the index of the current thread within the block (here 0–127).
- Finally, we have declared a function called `foo` that does the actual work, which in this case is nothing. Here the `__device__` attribute indicates that it is something that is supposed to be executed on the GPU. When we compile this code with `nvcc`, it will produce machine code suitable for the GPU here.

In summary, the GPU will run `mykernel` 12800 times (without any parameters, but with different values in `blockIdx` and `threadIdx`), and hence we will run `foo` 12800 times (with different parameters). This is very much like a parallelized version of the following loop:

```
for (int i = 0; i < 100; ++i) {
    for (int j = 0; j < 128; ++j) {
        foo(i, j);
    }
}
```

Internally, the GPU will divide each block in smaller units of work, called "warps". Each warp consists of 32 threads. This is one of the reasons why it makes sense to pick a nice round number such as 64, 128, or 256 threads per block. Very small values make our code inefficient. Very large values do not work at all: the GPU will simply refuse to launch the kernel if we try to ask for more than 1024 threads per block.

Multidimensional grids and blocks

Often it is convenient to index blocks and threads with 2-dimensional or 3-dimensional indexes. Here is an example in which we create 20 × 5 blocks, each with 16 × 8 threads:

```
__device__ void foo(int iy, int ix, int jy, int jx) {}

__global__ void mykernel() {
    int ix = blockIdx.x;
    int iy = blockIdx.y;
    int jx = threadIdx.x;
    int jy = threadIdx.y;
    foo(iy, ix, jy, jx);
}

int main() {
    dim3 dimGrid(20, 5);
    dim3 dimBlock(16, 8);
    mykernel<<<dimGrid, dimBlock>>>();
    cudaDeviceSynchronize();
}
```

The end result is similar to the parallelized version of the following loop:

```
for (int iy = 0; iy < 5; ++iy) {
    for (int ix = 0; ix < 20; ++ix) {
        for (int jy = 0; jy < 8; ++jy) {
            for (int jx = 0; jx < 16; ++jx) {
                foo(iy, ix, jy, jx);
            }
        }
    }
}
```

The term "grid" is often used to refer to the multidimensional arrangement of blocks.

Multidimensional indexes are primarily a matter of convenience. One could equally well use one-dimensional indexes and a little bit of arithmetic to turn them into multidimensional indexes.