Chapter 4: GPU programming

Version 0: OpenCL

We will now look at how to implement the same solution using OpenCL instead of CUDA.

OpenCL has a somewhat different toolchain. There is no separate "OpenCL compiler" — we will write normal C++ code, compile with a normal C++ compiler, and just use some functions that are provided in the OpenCL libraries.

This, unfortunately, also means that we will not be able to compile the GPU code while we compile the rest of the program. The kernel code is merely a string of characters, and only at run time it is passed to an OpenCL library function that compiles it to the machine code of whatever GPU we happen to be using.

Every time we run our program, we will call an OpenCL library function that runs a C++ compiler. Compilation errors related to the kernel code will be noticed only at run time.

GPU side

Recall that our CUDA kernel looked like this:

```
__global__ void mykernel(float* r, const float* d, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if (i >= n || j >= n)
        return;
    float v = HUGE_VALF;
    for (int k = 0; k < n; ++k) {
        float x = d[n*i + k];
        float y = d[n*k + j];
        float z = x + y;
        v = min(v, z);
    }
    r[n*i + j] = v;
}
```

In OpenCL we simply define a string that contains the same code, with minor modifications:

```
const char* kernel_source =
R"(
__kernel void mykernel(__global float* r, __global const float* d, int n) {
    int i = get_global_id(0);
    int j = get_global_id(1);
    if (i >= n || j >= n)
        return;
    float v = HUGE_VALF;
    for (int k = 0; k < n; ++k) {
        float x = d[n*i + k];
        float y = d[n*k + j];
        float z = x + y;
        v = min(v, z);
    }
    r[n*i + j] = v;
}
)";
```

Note that e.g. `threadIdx.x + blockIdx.x * blockDim.x` is replaced with `get_global_id(0)`, `__global__` is now known as `__kernel`, and the pointers to the global GPU memory have a `__global` attribute, but a mechanical search-and-replace is enough to do the translation.

CPU side

Now comes the ugly part. We will naturally go through the same steps as what we did with CUDA: allocate some GPU memory, copy the input to the GPU memory, launch the kernel, and copy the results back. Unfortunately, we will also need to call a bunch of OpenCL functions just to get started: find the right GPU device, compile the kernel, etc.:

```
void step(float* r, const float* d, int n) {
    // Setup device
    cl_int err;
    cl_platform_id platform;
    CHECK(clGetPlatformIDs(1, &platform, NULL));
    cl_device_id device;
    CHECK(clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL));
    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
    check(err, "clCreateContext");
#ifdef CL_VERSION_2_0
    cl_command_queue queue
        = clCreateCommandQueueWithProperties(context, device, NULL, &err);
#else
    cl_command_queue queue = clCreateCommandQueue(context, device, 0, &err);
#endif
    check(err, "clCreateCommandQueue");

    // Compile kernel
    cl_program program
        = clCreateProgramWithSource(context, 1, &kernel_source, NULL, &err);
    check(err, "clCreateProgramWithSource");
    check_build(program, device, clBuildProgram(program, 1, &device, NULL, NULL, NULL));
    cl_kernel kernel = clCreateKernel(program, "mykernel", &err);
    check(err, "clCreateKernel");

    // Allocate memory & copy data to GPU
    cl_mem dGPU = clCreateBuffer(context, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
                                 n * n * sizeof(float), (void*)d, &err);
    check(err, "clCreateBuffer");
    cl_mem rGPU = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                 n * n * sizeof(float), NULL, &err);
    check(err, "clCreateBuffer");

    // Run kernel
    size_t wlsize[2] = {16, 16};
    size_t wgsize[2] = {size_t(roundup(n, wlsize[0])), size_t(roundup(n, wlsize[1]))};
    CHECK(clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&rGPU));
    CHECK(clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&dGPU));
    CHECK(clSetKernelArg(kernel, 2, sizeof(int), (void*)&n));
    CHECK(clEnqueueNDRangeKernel(queue, kernel, 2, NULL, wgsize, wlsize, 0, NULL, NULL));
    CHECK(clFinish(queue));

    // Copy data back to CPU & release memory
    CHECK(clEnqueueReadBuffer(queue, rGPU, true, 0,
                              n * n * sizeof(float), r, 0, NULL, NULL));
    CHECK(clReleaseMemObject(rGPU));
    CHECK(clReleaseMemObject(dGPU));

    // Release everything else
    CHECK(clReleaseKernel(kernel));
    CHECK(clReleaseProgram(program));
    CHECK(clReleaseCommandQueue(queue));
    CHECK(clReleaseContext(context));
}
```

Note that here we need to specify the total number of threads (`wgsize`), while in CUDA we specified the number of blocks (`dimGrid`).

Error checking is also more complicated, as we will need to deal with compilation errors. This also shows the typical file names of the relevant header files:

```
#include <cstdlib>
#include <iostream>
#ifdef __APPLE__
    #include "OpenCL/opencl.h"
#else
    #include "CL/cl.h"
#endif

static inline void check(cl_int err, const char* context) {
    if (err != CL_SUCCESS) {
        std::cerr << "OpenCL error: " << context << ": " << err << std::endl;
        std::exit(EXIT_FAILURE);
    }
}

static inline void check_build(cl_program program, cl_device_id device, cl_int err) {
    if (err == CL_BUILD_PROGRAM_FAILURE) {
        std::cerr << "OpenCL build failed:" << std::endl;
        size_t len;
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL, &len);
        char* log = new char[len];
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, len, log, NULL);
        std::cout << log << std::endl;
        delete[] log;
        std::exit(EXIT_FAILURE);
    } else if (err != CL_SUCCESS) {
        std::cerr << "OpenCL build failed: " << err << std::endl;
        std::exit(EXIT_FAILURE);
    }
}

#define CHECK(x) check(x, #x)
```

Compilation and linking

The C++ compiler should not require any additional arguments; we will just need to link our program with the OpenCL libraries. On Linux, linking with `-lOpenCL` should be sufficient, provided that suitable libraries are installed. On macOS, linking with `-framework OpenCL` should do the trick.