Chapter 4: GPU programming

Version 2: OpenCL

We do not need any new OpenCL features; a mechanical translation from CUDA to OpenCL is sufficient. We can keep multiple kernels in the same string:

```
const char* kernel_source =
R"(
__kernel void myppkernel(__global const float* r, __global float* d, int n, int nn) {
    int ja = get_local_id(0);
    int i = get_group_id(1);

    __global float* t = d + nn * nn;

    for (int jb = 0; jb < nn; jb += 64) {
        int j = jb + ja;
        float v = (i < n && j < n) ? r[n*i + j] : HUGE_VALF;
        d[nn*i + j] = v;
        t[nn*j + i] = v;
    }
}

__kernel void mykernel(__global float* r, __global const float* d, int n, int nn) {
    int ia = get_local_id(0);
    int ja = get_local_id(1);
    int ic = get_group_id(0);
    int jc = get_group_id(1);

    __global const float* t = d + nn * nn;

    float v[8][8];
    for (int ib = 0; ib < 8; ++ib) {
        for (int jb = 0; jb < 8; ++jb) {
            v[ib][jb] = HUGE_VALF;
        }
    }
    for (int k = 0; k < n; ++k) {
        float x[8];
        float y[8];
        for (int ib = 0; ib < 8; ++ib) {
            int i = ic * 64 + ib * 8 + ia;
            x[ib] = t[nn*k + i];
        }
        for (int jb = 0; jb < 8; ++jb) {
            int j = jc * 64 + jb * 8 + ja;
            y[jb] = d[nn*k + j];
        }
        for (int ib = 0; ib < 8; ++ib) {
            for (int jb = 0; jb < 8; ++jb) {
                v[ib][jb] = min(v[ib][jb], x[ib] + y[jb]);
            }
        }
    }
    for (int ib = 0; ib < 8; ++ib) {
        for (int jb = 0; jb < 8; ++jb) {
            int i = ic * 64 + ib * 8 + ia;
            int j = jc * 64 + jb * 8 + ja;
            if (i < n && j < n) {
                r[n*i + j] = v[ib][jb];
            }
        }
    }
}
)"";
```

We can then compile all kernels with one `clBuildProgram` operation, and just call `clCreateKernel` separately for each kernel:

```
void step(float* r, const float* d, int n) {
    int nn = roundup(n, 64);

    // 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 ppkernel = clCreateKernel(program, "myppkernel", &err);
    check(err, "clCreateKernel");
    cl_kernel kernel = clCreateKernel(program, "mykernel", &err);
    check(err, "clCreateKernel");

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

    // Run kernel
    {
        size_t wlsize[2] = {64, 1};
        size_t wgsize[2] = {64, size_t(nn)};
        CHECK(clSetKernelArg(ppkernel, 0, sizeof(cl_mem), (void*)&rGPU));
        CHECK(clSetKernelArg(ppkernel, 1, sizeof(cl_mem), (void*)&dGPU));
        CHECK(clSetKernelArg(ppkernel, 2, sizeof(int), (void*)&n));
        CHECK(clSetKernelArg(ppkernel, 3, sizeof(int), (void*)&nn));
        CHECK(clEnqueueNDRangeKernel(
            queue, ppkernel, 2, NULL, wgsize, wlsize, 0, NULL, NULL
        ));
        CHECK(clFinish(queue));
    }

    // Run kernel
    {
        size_t wlsize[2] = {8, 8};
        size_t wgsize[2] = {size_t(nn / 8), size_t(nn / 8)};
        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(clSetKernelArg(kernel, 3, sizeof(int), (void*)&nn));
        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(ppkernel));
    CHECK(clReleaseKernel(kernel));
    CHECK(clReleaseProgram(program));
    CHECK(clReleaseCommandQueue(queue));
    CHECK(clReleaseContext(context));
}
```

Results

We tried out the OpenCL implementation using an Apple iMac with an AMD Radeon R9 M390 GPU. The GPU has 1024 parallel arithmetic units and a clock frequency of 0.958 GHz, which would suggest a theoretical maximum performance of 981 billion operations per second in our application. For n = 6300, we got a total running time of less than 0.9 seconds (including all overhead related to kernel compilation and memory management), and we were able to do approx. 580 billion operations per second. The fastest CPU solution on the same computer took 2.7 seconds, so with the help of GPUs we got a nice factor-3 speedup.

As a very different example, we used an old Apple MacBook Air laptop with an integrated Intel HD Graphics 5000 GPU. We used the instance size of n = 4000. On this laptop, the fastest CPU solution took 2.9 seconds. If we now tried to use the above OpenCL solution as such, the performance would be very poor — indeed, the kernel call simply times out and the program crashes. However, we can slightly tune the parameters and use e.g. blocks with 4 × 4 threads and let each block calculate 5 × 5 results. Such a small change already gives a reasonable running time of 1.8 seconds, which is much better than what we achieved with the CPU. This is far from the theoretical limitations of the hardware, but it demonstrates that even integrated laptop GPUs can outperform the CPU that we have on the same machine.

In summary, the ideas that we used in the CUDA solution can be directly used in other platforms as well; however, sometimes parameters need some tuning depending on the GPU that we use.