The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

Introduction to GPU programming

Practice 4. Numerical integration of heat equation

Nizhni Novgorod

2014

Author: S.I. Bastrakov

OBJECTIVES

The objective of this practice is to develop a simple implementation of numerical integration of 2D heat equation using explicit scheme.

ABSTRACT

This practice is devoted to development a simple implementation of numerical integration of 2D heat equation using explicit scheme. We illustrate using 2D indexes in kernels and device functions.

BRIEF OVERVIEW

We consider a Dirichlet problem for 2D heat equation:

$$\frac{\partial u(x,y,t)}{\partial t} = \Delta u(x,y,t) + f(x,y,t)$$

$$a \le x \le b, \quad c \le y \le d, \quad 0 \le t \le T$$

$$u(x,y,0) = u_0(x,y)$$

$$u(a,y,t) = \varphi_1(y,t), u(b,y,t) = \varphi_2(y,t)$$

$$u(x,c,t) = \varphi_3(x,t), u(x,d,t) = \varphi_4(x,t)$$

Simulation area is covered by uniform grid with steps Δx, Δy, and Δt. Denote ${v^{(k)}}_{ij}$ – grid value in point (x_i, y_j) at k-th time moment. We will use explicit scheme for numerical integration:

$${v^{(k+1)}}_{ij} = {v^{(k)}}_{ij} + \Delta t * \left(f(x_i, y_j) + \frac{{v^{(k)}}_{i+1j} - 2{v^{(k)}}_{ij} + {v^{(k)}}_{i-1j}}{\Delta x^2} + \frac{{v^{(k)}}_{ij+1} - 2{v^{(k)}}_{ij} + {v^{(k)}}_{ij-1}}{\Delta y^2} \right)$$

Computation is performed iteration by iteration in time, computations within one time iteration are independent. For simplicity we assume boundary conditions are stationary (do not depend on time), in this case on each time step we can only update internal values.

Implementation on CPU is as follows:

```
float f(float x, float y) {
    return 0.05f * x * x + 0.001f * y * y * y;
}
void step(int nx, int ny, float dx, float dy, float dt, float a, float c,
        const float * v, float * newV) {
   for (int i = 1; i < nx - 1; ++i)
        for (int j = 1; j < ny - 1; ++j) {
            float x = a + i * dx;
            float y = c + j * dy;
            newV[i * ny + j] = v[i * ny + j] + dt * (f(x, y) +
                (v[(i + 1) * ny + j] - 2.0f * v[i * ny + j] + v[(i - 1) * ny
+ j]) / (dx * dx) +
```

```
                (v[i * ny + (j + 1)] - 2.0f * v[i * ny + j] + v[i * ny + (j -
1)]) / (dy * dy));
        }
}
```

Two loops with independent iterations are easily converted to a kernel with 2D indexes. Note that we can not call right-hand side function f from a kernel, as it is not declared with __device__ qualifier. A solution is to either declare it with __host__ and __device__ qualifiers, or to create a copy with __device__ qualifier. We choose the latter:

```
__device__ float f_gpu(float x, float y) {
    return 0.05f * x * x + 0.001f * y * y * y;
}
__global__ void kernel(int nx, int ny, float dx, float dy, float dt, float a,
        float c, const float * v, float * newV) {
    int i = 1 + blockIdx.x * blockDim.x + threadIdx.x;
    int j = 1 + blockIdx.y * blockDim.y + threadIdx.y;
    if ((i < nx - 1) && (j < ny - 1)) {
        float x = a + i * dx;
        float y = c + j * dy;
        newV[i * ny + j] = v[i * ny + j] + dt * (f_gpu(x, y) +
            (v[(i + 1) * ny + j] - 2.0f * v[i * ny + j] + v[(i - 1) * ny +
j]) / (dx * dx) + (v[i * ny + (j + 1)] - 2.0f * v[i * ny + j] + v[i * ny + (j
- 1)]) / (dy * dy));
    }
}
```

Note that .x index is used for rows and .y is used for columns. Conditions in the kernel correspond to conditions in the loop in the implementation for CPUs. A function to invoke the kernel is as follows:

```
void step_gpu(int nx, int ny, float dx, float dy, float dt, float a, float c,
const float * v_gpu, float * newV_gpu) {
const int threadsX = 16;
    const int threadsY = 16;
    dim3 blocks(threadsX, threadsY);
    dim3 grid(((nx - 2) + (threadsX - 1)) / threadsX,
              ((ny - 2) + (threadsY - 1)) / threadsY);
    kernel<<<grid, blocks>>>(nx, ny, dx, dy, dt, a, c, v_gpu, newV_gpu);
}
```

We fix block size for both dimensions and compute number of blocks for each dimension using the same scheme as for vector addition.

FOR STUDENTS

CUDA C language is described in [1, 2].

REFERENCES

1. Sanders J., Kandrot E. CUDA by Example: An Introduction to General-Purpose GPU Programming. – Addison-Wesley Professional, 2010. – 312 p.
2. NVIDIA CUDA C Programming Guide. [http://docs.nvidia.com/cuda/cuda-c-programming-guide/].

INDIVIDUAL WORK

1. Create implementation of the kernel using 1D indexes.
2. Implement a version with non-stationary boundary condition.