Blue Waters Petascale Semester Curriculum v1.0

Unit 7: CUDA

Lesson 9: Heat Diffusion in 1–2 D

Developed by David A. Joiner

for the Shodor Education Foundation, Inc.

Browse and search the full curriculum at
http://shodor.org/petascale/materials/semester-curriculum

We welcome your improvements! You can submit your proposed changes to this material and the rest of the curriculum in our GitHub repository at https://github.com/shodor-education/petascale-semester-curriculum

We want to hear from you! Please let us know your experiences using this material by sending email to petascale@shodor.org

The Heat Equation in CUDA

The Heat Equation

- The temperature within a solid will flow from hot to cold, diffusing out over time. This can be expressed mathematically as
$$\frac{dT}{dt} = \alpha \nabla^2 T$$
- In equilibrium, the temperature is not changing
$$\nabla^2 T = 0$$
- It is expected that solids with fixed boundary conditions will approach equilibrium over time.

The Relaxation Method

- As solids are expected to evolve to the steady state, the steady state can be solved for by picking any initial temperature distribution and letting the system iterate towards equilibrium.

$$T_{next} = T_{old} + \alpha \nabla^2 T_{old}$$

- The Laplacian can be approximated in 1D as

$$\nabla^2 T = \frac{T(x+h) - 2T(x) + T(x-h)}{h^2}$$

- In steady state

$$\frac{T(x+h) - 2T(x) + T(x-h)}{h^2} = 0$$

- Or

$$T(x) = \frac{1}{2}\big(T(x+h) + T(x-h)\big)$$

The Relaxation Method, simplified

- In short, in steady state the temperature at any point is the average of its surroundings
- The relaxation method works by replacing each value of T in the solution with the average of its surroundings

The Heat Equation in PSEUDOCODE

```
FOR MANY ITERATIONS
    FOR MANY VALUES OF X
        TNEXT(i) = 0.5*(T(i-1)+T(i+1))
    FOR MANY VALUES OF X
        T(i) = 0.5*(TNEXT(i-1)+TNEXT(i+1))
```

Parallelizing Loops in CUDA

- Have 2 versions of your arrays
  - Device
  - Host
- Initialize memory on host, copy to device
- For each iteration
  - Call kernel on Host with many threads
  - Have kernel designed to change one piece of memory (or very few)
- Copy memory back to host when done

```
__global__ void compute_average_device(int n, double * average, double * T) {
    int i = 1+threadIdx.x+blockDim.x*blockIdx.x;

    while(i<n-1) {
        average[i] = (1.0/3.0)*(T[i-1]+T[i]+T[i+1]);

        i += blockDim.x*gridDim.x;
    }
}
```

```
cudaEventRecord(write_start);
cudaMemcpy(T_device,T_host,n*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(average_device,average_host,n*sizeof(double),cudaMemcpyHostToDevice);
for(iter=0;iter<itmax;iter+=2) {
    compute_average_device<<<n_blocks,n_threads_per_block>>>(n,average_device,T_device);
    compute_average_device<<<n_blocks,n_threads_per_block>>>(n,T_device,average_device);
}
cudaMemcpy(T_host,T_device,n*sizeof(double),cudaMemcpyDeviceToHost);
```

Going to 2D

- You can force everything to still work with just threadIdx.x and blockIdx.x, or you can make use of multiple dimensions of blocks and grids

```
__global__ void compute_average_device(int nrows, int ncols, double * average, double * T) {
    int i = 1+threadIdx.y+blockDim.y*blockIdx.y;

    while(i<nrows-1) {
        int j = 1+threadIdx.x+blockDim.x*blockIdx.x;
        while(j<ncols-1) {
            average[i*ncols+j] = …
            j += blockDim.x*gridDim.x;
        }
        i += blockDim.y*gridDim.y;
    }
}
```

```
dim3 grid(n_blocks_x,n_blocks_y);
dim3 block(n_threads_per_block_x,n_threads_per_block_y);

cudaMemcpy(T_device,T_host,nrows*ncols*sizeof(double),cudaMemcpyHostToDevice);
cudaMemcpy(average_device,average_host,nrows*ncols*sizeof(double),cudaMemcpyHostToDevice);
for(iter=0;iter<itmax;iter+=2) {
        compute_average_device<<<grid,block>>>(nrows,ncols,average_device,T_device);
        compute_average_device<<<grid,block>>>(nrows,ncols,T_device,average_device);
}
cudaMemcpy(T_host,T_device,nrows*ncols*sizeof(double),cudaMemcpyDeviceToHost);
```