HPC Project: Heat Diffusion Equation - Third Delivery

Authors: Raul Hidalgo & Marc Gaspà

Github: https://github.com/deinok/PARALLEL

Strategy

As our problem is a SMPD (Single Program Multiple Data) problem and we also have it already implemented in OpenMP, what we will do is transform the OpenMP to CUDA.

Allocation & Free

We changed all the `malloc` and family to `cudaMallocManaged` so we can use it in HOST and in DEVICE. Also, it's `free` have been changed to `cudaFree`.

initialize_grid

We created a kernel for `initialize_grid`, before the change, that was two for loops nested.

```
__global__ void initialize_grid(double *grid, int nx, int ny)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < nx && j < ny)
    {
        int inyj = i * ny + j;
        if (i == j)
        {
            grid[inyj] = 1500.0;
        }
        else if (i == nx - 1 - j)
        {
            grid[inyj] = 1500.0;
        }
        else
        {
            grid[inyj] = 0.0;
        }
    }
}
```

We invoke the code like this, ensuring that any CUDA error is shown, and the DEVICE is syncronized.

```
dim3 threadsPerBlock(16, 16);
dim3 numBlocks((ny + threadsPerBlock.x - 1) / threadsPerBlock.x, (nx +
threadsPerBlock.y - 1) / threadsPerBlock.y);
```

```
// Initialize the grid
initialize_grid<<<numBlocks, threadsPerBlock>>>(grid, nx, ny);
cudaError_t err1 = cudaGetLastError();
if (err1 != cudaSuccess) {
    printf("CUDA kernel launch error: %s\n", cudaGetErrorString(err1));
}
cudaDeviceSynchronize();
```

solve_heat_equation

We moved the function to main, as it's only used once, and we split it in two parts.

- solve_heat_equation
- apply_boundary_conditions

The `solve_heat_equation` it's just two nested loops, so like the `initialize_grid` we apply the same tactic.

```
__global__ void solve_heat_equation(double *grid, double *new_grid, double r, int
nx, int ny)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (i >= 1 && i < nx - 1 && j >= 1 && j < ny - 1)
    {
        int inyj = i * ny + j;
        new_grid[inyj] = grid[inyj] + r * (grid[(i + 1) * ny + j] + grid[(i - 1) *
ny + j] - 2 * grid[inyj]) + r * (grid[inyj + 1] + grid[inyj - 1] - 2 *
grid[inyj]);
    }
}
```

Regarding the `apply_boundary_conditions` it's just one loop, so it ends like this:

```
__global__ void apply_boundary_conditions(double *grid, int nx, int ny)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < nx)
    {
        grid[0 * ny + idx] = 0.0;
        grid[(nx - 1) * ny + idx] = 0.0;
    }
    if (idx < ny)
    {
        grid[idx * ny + 0] = 0.0;
        grid[idx * ny + (ny - 1)] = 0.0;
    }
}
```

Regarding how to call this kernels, we implemented the following code:

```
for (int step = 0; step < steps; step++)
{
    solve_heat_equation<<<numBlocks, threadsPerBlock>>>(grid, new_grid, r, nx, 
ny);
    cudaError_t err2 = cudaGetLastError();
    if (err2 != cudaSuccess) {
        printf("CUDA kernel launch error: %s\n", cudaGetErrorString(err1));
    }
    cudaDeviceSynchronize();

    // Apply BCs
    apply_boundary_conditions<<<(nx > ny ? nx : ny + 255) / 256, 256>>>(new_grid, 
nx, ny);
    cudaError_t err3 = cudaGetLastError();
    if (err3 != cudaSuccess) {
        printf("CUDA kernel launch error: %s\n", cudaGetErrorString(err1));
    }
    cudaDeviceSynchronize();

    // Swap pointers
    double *temp = grid;
    grid = new_grid;
    new_grid = temp;
}
```

It ensures that we always syncronize the DEVICE and to check the CUDA errors.

Metrics

In order to build the project we used `cmake`, using the following chain of calls: `cmake -S . -B build && cmake --build ./build/ && ./build/main <size> <steps> <filename>` Of course, using a better profile in `cmake` would enable better optimizations, but that is out of scope.

Serial

	100	1000	10000	100000
100x100	0.011220s	0.090778s	0.866678s	8.601631s
1000x1000	1.022262s	10.702762s	95.275654s	932.119260s
2000x2000	4.102707s	42.515603s	423.742169s	3770.095796s

In this table, we can clearly see that for small grids and steps it's so much faster. And that the increment of steps is linear.

Parallel

	100	1000	10000	100000
100x100	0.595487s	4.663546s	45.774295s	439.763672s
1000x1000	0.631579s	4.821834s	46.902149s	464.976946s
2000x2000	0.775071s	5.154828s	48.968285s	487.932836s

In this table, and taking in comparasion the serial one we can see that for small grids, the serial version wins in performace and for bigger grids, the parallel version starts gaining performace.

Speedup

Speedup is Tserial / Tparallel

	100	1000	10000	100000
100x100	0.018	0.019	0.018	0.019
1000x1000	1.618	2.219	2.031	2.004
2000x2000	5.293	8.247	8.653	7.726

In this table, we can clearly see by te speedup that in the configurations that allows more parallelization, the speedup is astonishing.

Conclusions

After developing program, we found that it's pretty easy to convert a C program to a CUDA one, atleast for simple `for` loops. Also, we have found that switching fast from HOST to DEVICE calls, add a small overhead that slows down the algoritms. We can see this case in the small grid case, as it requires two calls to a kernel every step. Probably with a better implementation we could avoid one of the two kernel calls and loop inside the kernel, avoiding most of the price of the context switching.

It would be also interesting to see the comparasions agaist Serial, OpenMP and CUDA using SIMD (AVX512) as the context switching does not penalize.