Heat equation in CUDA

Mihaela Gledacheva, Nadja Zivkovic

May - June 2024

1 Introduction

The objective of this project is to compute approximate solutions of the heat equation on a 2D plane, as expressed by the partial differential equation $\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}$, where U is the heat function defined at a given time t, and at coordinates x and y in the 2D plane.

2 Problem Definition

2.1 Input

We provide as input the initial non-zero heat values, that are going to be referred to as **base**, as well as the **time** and the area at which we want to compute the heat equation, the latter called **target**. Additionally, we fix a time step parameter **gamma** (Γ) and a space step parameter **delta** (Δ).

The type and role of these input parameters is going to be explained below.

2.2 Custom Classes

In order to describe the subsection of the plane relevant to the problem, we define a class `Region`, which supports both discrete and continuous areas. It can be initialized in two ways:

- by a finite vector of points
- by two functions, one that returns true only if the point belongs to the region, and another that computes the initial heat value at a point according to a given equation

Every `Point` has coordinates x and y, as well as a heat value U, set to 0 by default.

```
// example of using a set of points to define a region
Region({Point(-2, -3, 100), Point(3.5, 0, 150), Point(0.5, 2, 50)});

// example of using condition functions to define a region
Region([](const Point& p) { return p.x*p.x + p.y*p.y < 64; },
       [](const Point& p) { return 100; });
```

Both the base and the target area are going to be instances of class `Region`. The base can be of any form, while the target must be a rectangle, given by its lower left and upper right corners.

```
// example of a target area
target = Region({Point(-20, -20), Point(20, 20)});
```

2.3 Discretization

To find approximate solutions of the heat equation, we discretize the value of U at coordinates $x_i = i\Delta$ and $y_j = j\Delta$, and at times $t_n = n\Gamma$ for the given time and space steps Γ and Δ. That leads to the explicit formula $U^{n+1}_{i,j} = (1 - 4\lambda)U^n_{i,j} + \lambda(U^n_{i+1,j} + U^n_{i,j+1} + U^n_{i-1,j} + U^n_{i,j-1})$, where $\lambda = \frac{\Gamma}{\Delta^2}$.

We are going to assume that $\lambda < 0.25$.

In order to be able to apply the above formula, we first create a heat function grid, U_{grid}. The idea is to have $U_{grid}[i][j]$ corresponding to the value of $U^n_{i,j}$.

The size of this grid is computed in two parts:

- First use the space step to determine how many rows and columns does the target rectangle have
- Then add $2 * iterations$ to both the number of rows and the number columns, where *iterations* is determined by the time at which we want to compute the heat equation and the time step. This is necessary, since the formula for computing $U^{n+1}_{i,j}$ requires us to have computed $U^n_{i-1,j}$, $U^n_{i,j}$, $U^n_{i+1,j}$, $U^n_{i,j-1}$ and $U^n_{i,j+1}$ in the previous step, thus adding 2 more rows and columns per iteration.

Once we have the grid's dimensions, we proceed to fill it in with the given initial values. For every cell (i, j) at the grid, we can convert back to its original coordinates, then check whether that point belongs to the base. If so, we obtain its heat value and store it in the grid.

3 Methodology

From the initial heat function grid, we can obtain the approximate solutions recursively by performing $time/\Gamma$ iterations according to the formula described above.

3.1 Sequential Algorithm

```
# pseudo-code
function ComputeSequential(U, λ, rows, columns, iterations):
    U_next = new array of size rows*columns
    for n = 0 to iterations-1:
        for i = 0 to rows-1:
            for j = 0 to columns-1:
                compute U_next[i,j] using λ, U[i,j], U[i-1,j], U[i+1,j], U[i,j-1] and U[i,j+1]
        U = U_next
```

3.2 GPU Algorithm

```
# pseudo-code
function ComputeGPUAux1(U, U_next, rows, columns, λ, size):
    index = blockIdx.x * blockDim.x + threadIdx.x
    begin = size * index
    end = min(size * (index + 1), rows * columns)
    for k = begin to end-1:
        i = k / columns
        j = k % columns
        compute U_next[i,j] using λ, U[i,j], U[i-1,j], U[i+1,j], U[i,j-1] and U[i,j+1]

function ComputeGPU1(U, λ, rows, columns, iterations):
    BLOCKS_NUM = 48
    THREADS_PER_BLOCK = 256
    TOTAL_THREADS = BLOCKS_NUM * THREADS_PER_BLOCK
    size = (rows * columns + TOTAL_THREADS + 1) / TOTAL_THREADS

    d_U = allocate device memory of size rows * columns
    d_U_next = allocate device memory of size rows * columns

    copy U to d_U

    for n = 0 to iterations-1:
        launch ComputeGPUAux1 kernel with <<<BLOCKS_NUM, THREADS_PER_BLOCK>>>
        synchronize device
        d_U = d_U_next

    copy d_U to U

    free d_U
    free d_U_next
```

We propose several ways to parallelize the code on CUDA. In the first one, showcased above, the number of blocks and threads per block is fixed. The second version (see below) uses a separate thread for every cell in the heat function grid.

```
# pseudo-code
function ComputeGPUAux2(U, U_next, rows, columns, λ):
    i = blockIdx.y * blockDim.y + threadIdx.y
    j = blockIdx.x * blockDim.x + threadIdx.x
    if i < rows and j < columns:
        compute U_next[i,j] using λ, U[i,j], U[i-1,j], U[i+1,j], U[i,j-1] and U[i,j+1]

function ComputeGPU2(U, λ, rows, columns, iterations):
    threadsPerBlock = (16, 16)
    blocksPerGrid = ((columns + threadsPerBlock.x + 1) / threadsPerBlock.x,
                     (columns + threadsPerBlock.y + 1) / threadsPerBlock.y)

    d_U = allocate device memory of size rows * columns
    d_U_next = allocate device memory of size rows * columns

    copy U to d_U

    for n = 0 to iterations-1:
        launch ComputeGPUAux2 kernel with <<<blocksPerGrid, threadsPerBlock>>>
        synchronize device
        d_U = d_U_next

    copy d_U to U

    free d_U
    free d_U_next
```

The third and final method is similar to the previous one, but instead of a thread for every cell, we create one for every row in the grid.

```
# pseudo-code
function ComputeGPUAux3(U, U_next, rows, columns, λ)
    i = blockIdx.x * blockDim.x + threadIdx.x
    if i < rows:
        for j = 0 to columns-1:
            compute U_next[i,j] using λ, U[i,j], U[i-1,j], U[i+1,j], U[i,j-1] and U[i,j+1]

function ComputeGPU3(U, lambda, rows, columns, iterations)
    threadsPerBlock = 256
    blocksPerGrid = (rows + threadsPerBlock + 1) / threadsPerBlock

    d_U = allocate device memory of size rows * columns
    d_U_next = allocate device memory of size rows * columns

    copy U to d_U

    for n = 0 to iterations-1:
        launch ComputeGPUAux3 kernel with <<<blocksPerGrid, threadsPerBlock>>>
        synchronize device
        d_U = d_U_next

    copy d_U to U

    free d_U
    free d_U_next
```

4 Analysis

All of the above implementations possess distinct advantages and drawbacks. Their performance depends on the size and shape of the heat function grid.

The sequential method `ComputeSequential` executes in a single-threaded manner. This approach is beneficial for small grid sizes due to the absence of overhead associated with parallelism. However, as the grid size increases, the performance degrades significantly due to the O(rows * cols * iterations) complexity.

In contrast, `ComputeGPU1` leverages CUDA to parallelize the computation with a fixed number of threads. Dividing the problem into chunks and assigning each chunk to a thread, provides a substantial speedup over the sequential version for larger grids. The fixed thread count (48 blocks * 256 threads) makes it well-suited for moderate-sized grids where the chunking strategy aligns well with the grid size. However, this fixed count might not scale efficiently for very large or very small grids.

The `ComputeGPU2` implementation takes a different approach by assigning one thread per cell using a 2D grid of threads. This method maximizes parallelism and can significantly enhance the performance for very large grids, where a high degree of parallelism is beneficial. However, it is inefficient for smaller grids due to the under utilization of GPU resources.

Lastly, in `ComputeGPU3` each thread handles an entire row independently. This moderate level of parallelism can be particularly effective for grids with a large number of rows and fewer columns. As before, the method is best suited for medium to large grids.

5 Results

We proceed to confirm the described behaviour experimentally for several test cases. Verifying the correctness of the code for small discrete cases can be done by manually checking the results. For larger point sets and continuous regions that is not possible. Instead, we compare the output from the sequential and the parallel methods and confirm that they are identical. As a second step we visualize the obtained solution and check that it aligns with our expectations.

5.1 Discrete region with a single hot spot

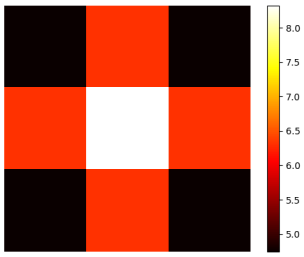
We begin with a single point, where the heat value is non-zero, and observe the area around it.

```
base = Region({Point(2, 2, 100)});
target = Region({Point(1, 1), Point(3, 3)});
time = 1;
```

Below are the obtained result images for different time and space steps; note that we are only displaying the target area and not the entirety of U_{grid}.

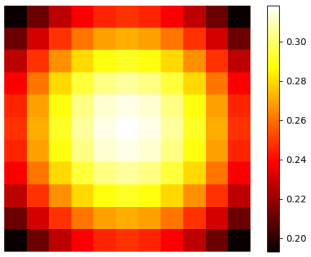
We observe that as Γ and Δ decrease, the runtime for every algorithm increases.

$\Gamma = 0.1$
$\Delta = 1$



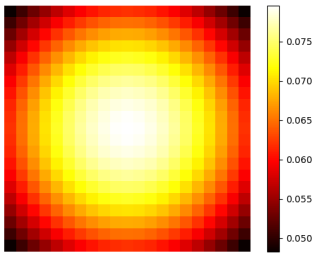
Seq: 0ms
GPU1: 80ms
GPU2: 62ms
GPU3: 61ms

$\Gamma = 0.005$
$\Delta = 0.2$



Seq: 232ms
GPU1: 81ms
GPU2: 83ms
GPU3: 115ms

$\Gamma = 0.00125$
$\Delta = 0.1$

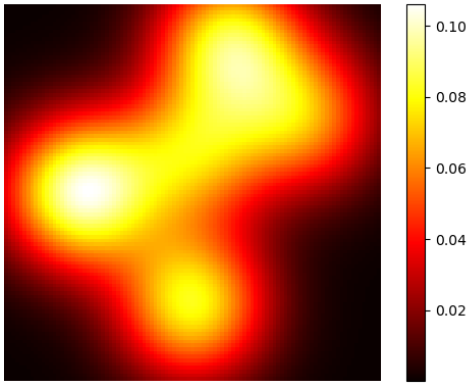


Seq: 15278ms
GPU1: 2141ms
GPU2: 316ms
GPU3: 1044ms

That is due to the increase of the grid size. The first grid is of size 23×23, the second is of size 411×411 and the last one - 1621×1621. As expected the parallel methods are not as effective on the smallest grid, however their runtime increases much more gradually as the grid gets bigger, with `ComputeGPU2` performing the best out of the three overall.
We also observe that `ComputeGPU1` gets outperformed by the other two parallel methods as the grid size increases.

5.2 Discrete region with a several hot spots

Next we are going to consider several points.
For this specific example we are working on a medium sized 1101×1101 grid.




```
base = Region({Point(-4, 1, 100),
               Point(-2, 3, 75),
               Point(0, -3, 125),
               Point(3, 0, 100),
               Point(-1, 0, 75)});
target = Region({Point(-5, -5), Point(5, 5)});
gamma = 0.002;
delta = 0.1;
time = 1;
```

Seq: 4203ms GPU1: 503ms
GPU2: 139ms GPU3: 532ms

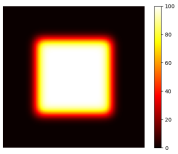
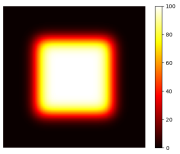
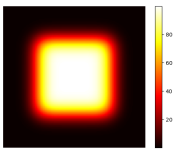
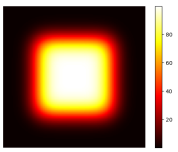
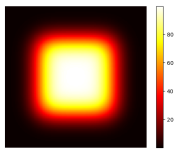
Once again the observed behaviour supports our theoretical analysis. The sequential algorithm's long execution time reflects its inefficiency for larger grids, while the varying degrees of parallelization in the GPU algorithms demonstrate substantial performance improvements. `ComputeGPU2` continues to be most effective with around 30 times speed up.
It is interesting to note that `ComputeGPU1` and `ComputeGPU3` demonstrate very similar performances, since both are equally well-matched to this problem's characteristics.

5.3 Continuous region

We are now going to showcase two continuous examples.

In the first we observe the diffusion from a square hot area over time.

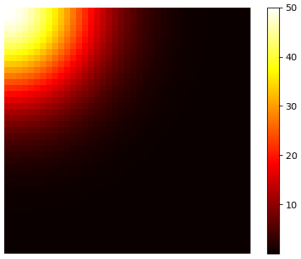
```
base = Region([](const Point& p) { return p.x*p.x < 64 && p.y*p.y < 64; },
              [](const Point& p) { return 100; });
target = Region({Point(-15, -15), Point(15, 15)});
gamma = 0.005;
delta = 0.2;
```

time = 0.5 grid 351×351	time = 1 grid 551×551	time = 1.5 grid 751×751	time = 2 grid 951×951	time = 2.5 grid 1151×1151
				
Seq: 86ms GPU1: 61ms GPU2: 71ms GPU3: 85ms	Seq: 424ms GPU1: 76ms GPU2: 67ms GPU3: 144ms	Seq: 1181ms GPU1: 163ms GPU2: 92ms GPU3: 273ms	Seq: 2526ms GPU1: 266ms GPU2: 113ms GPU3: 367ms	Seq: 4685ms GPU1: 642ms GPU2: 151ms GPU3: 513ms

For the second example we consider an area in which the initial heat values follow a Gaussian distribution.

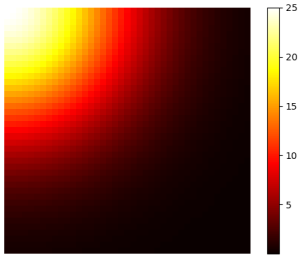
```
base = Region([](const Point& p) { return true; },
              [](const Point& p) { return 100 * std::exp(-(p.x*p.x + p.y*p.y)); });
target = Region({Point(0, 0), Point(4, 4)});
gamma = 0.001;
delta = 0.1;
```

time = 0.25
grid 541 × 541



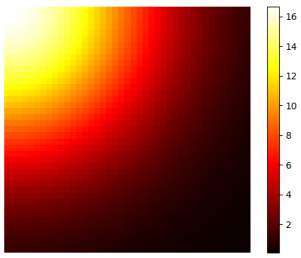
Seq: 592ms
GPU1: 82ms
GPU2: 60ms
GPU3: 191ms

time = 0.75
grid 1541 × 1541



Seq: 14016ms
GPU1: 1447ms
GPU2: 293ms
GPU3: 965ms

time = 1.25
grid 2541 × 2541



Seq: 60465ms
GPU1: 9170ms
GPU2: 955ms
GPU3: 2350ms

We can confirm that the behaviour of the algorithms is consistent with the previous results in both cases.

5.4 Overview

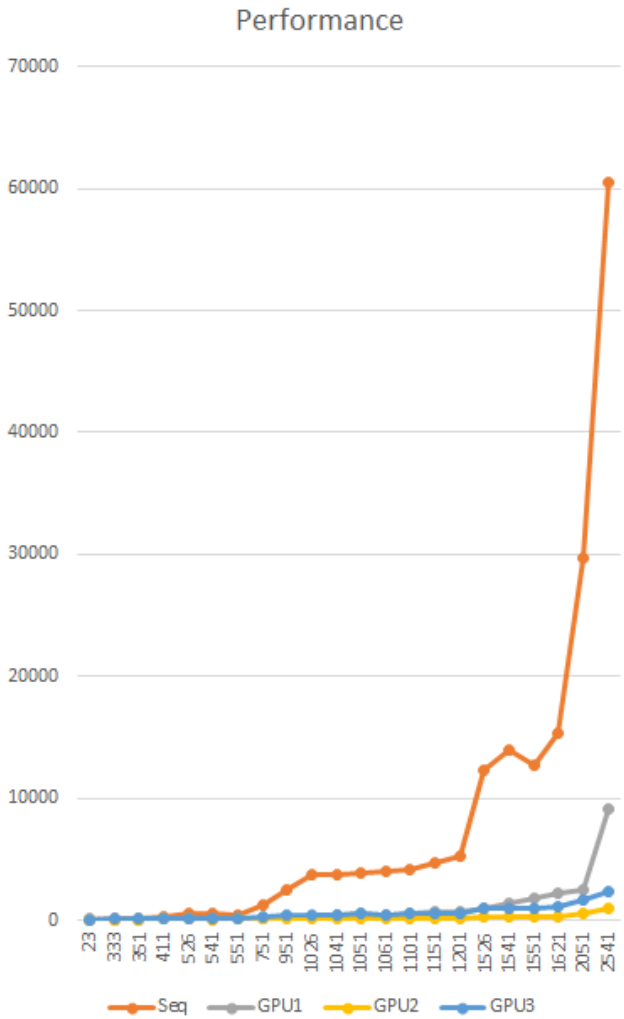
We ran a total of 30 tests. The exact performance comparisons have been recorded in `results/test_results.txt`.

A summary of the runtimes (in milliseconds) for all tested grid sizes is shown on the right. It allows us to conclude that the larger the size of the grid, the better the speed up when computing in parallel.

`ComputeGPU2` is consistently the best of the three parallel algorithms for larger grids.

`ComputeGPU1` can outperform the other two for small to medium sized grids of the right shape, however it is the least efficient as the grid size increases.

`ComputeGPU3` is not as efficient `ComputeGPU2`, but works well for large in terms of number of rows grids.



6 Conclusion

We claim to have successfully implemented a solver for the heat equation in 2D. We support various problem types and the algorithms are able tackle a wide range of grid sizes. With the provided analysis, we can choose a suitable method depending on the exact parameters of the problem.