GPU Paralleling of 2D Heat Equation FEM solver

Xujin He(xh1131), Zhiheng Wang(zw3518)*
New York University
United States
xh1131,zw3518@nyu.edu

ABSTRACT

This study investigates the computational performance of solving the 2D heat equation in both sequential and CUDA versions. The numerical solver employs a Backward Euler discretization scheme for temporal evolution, and the Compressed Sparse Row (CSR) Matrix is utilized for storing the sparse matrices arising from the implicit time-stepping scheme. The Cholesky decomposition is utilized for solving linear systems. Both sequential and CUDA implementations were developed and systematically tested on several computing systems to evaluate their respective efficiencies.

KEYWORDS

GPU, parallelization, CUDA, 2D heat equation, CSRMatrix, Backward Euler, Cholesky decomposition

1 INTRODUCTION

The numerical solution of partial differential equations (PDEs) is a fundamental aspect of scientific and engineering simulations, with applications ranging from heat conduction to fluid dynamics. In the context of solving the 2D heat equation, the choice of numerical methods and data structures can significantly influence both the accuracy and computational efficiency of the solution. In this study, we focus on the application of the Backward Euler method for temporal discretization and the use of the Compressed Sparse Row (CSR) matrix representation for efficient storage of the associated sparse matrices.

The Backward Euler method is a widely employed implicit time-stepping scheme known for its stability properties in handling parabolic PDEs, such as the heat equation. Its implicit nature necessitates the solution of linear systems at each time step, making the choice of an appropriate matrix storage format crucial for computational efficiency. In this regard, the CSR matrix format offers a compact representation for sparse matrices, reducing memory requirements and enhancing the overall performance of matrix operations.

The motivation behind this research lies in optimizing the numerical solution of the 2D heat equation by leveraging the Backward Euler method and the CSR matrix representation. A key driving force for this investigation is the recognition that the linear systems arising from implicit time-stepping schemes, such as the Backward Euler method, lend themselves to parallel processing. We explore both sequential and CUDA implementations, targeting diverse computing environments. Our focus is on understanding the impact of efficient matrix storage and parallel computing on the overall performance of the solver.

The integration of Cholesky decomposition further adds a layer of complexity to the numerical solution, and its interaction with the CSR matrix format is a key aspect of our investigation. By systematically analyzing the performance of the sequential and CUDA implementations, we aim to provide valuable insights into the interplay between numerical methods, sparse matrix storage, and parallel computing, with implications for a broader range of scientific simulations.

We list here our main contributions.

(1) We explore efficient ways of storing matrix of large dimensions in CUDA memory by using CSRMatrix.
(2) We accelerate initialization and recursion of Backward Euler methods using GPU parallel processing.
(3) We design and implement parallelization of Cholesky LDLT decomposition.

2 MATHEMATICAL PRICIPLES

Explicit methods suffers from stability constraint, that the ratio between Δh and Δt must satisfies the Courant-Friedrichs-Lewy condition for stability; In comparison, implicit methods such as backward Euler method are proven unconditionally stable for 2D heat equation, allowing computation with larger time steps. Also, implicit methods can effectively handle stiff systems of equations, giving it potential to solve a larger variety of problems.

However, the implicit methods are greatly limited by computational and memory requirements. On the computational side, for each time step forward, the implicit methods solves a system of equations, requiring inversion of a matrix which is very computationally intense. On the memory side, for high dimensional equations, the matrix size of the linear system grows large easily and have much higher memory requirements.

What's more, the curse of dimensional affect finite-difference methods on higher dimensions drastically, and we expect that parallelization

By constructing the linear system to be symmetric positive definite, we can apply the Cholesky Decomposition, but still this step will be computationally expensive.

We propose to solve a simple 2D heat equation with Dirichlet boundary conditions, the diffusivity is assumed to be 1 uniformly. We use u as the temperature:

$$\begin{aligned}
&u_t(x,y,t) = \Delta u(x,y,t), 0 \le x \le X, 0 \le y \le Y, t \ge 0 \\
&u((x,0),t) = a(x,t) \\
&u((x,Y),t) = d(x,t) \\
&u((0,y),t) = b(y,t) \\
&u((X,y),t) = c(y,t)
\end{aligned} \tag{1}$$

*

We employ finite difference method (FEM) with the second-order central difference scheme (5-point Laplacian) for spatial discretization with grid size h (X = mh, Y = nh) yielding $O(h^2)$ spatial discretization error.

The system of ODEs follows:

$$u_t(x,t) = \mathbf{A}u + \tilde{f}(x,t) \tag{2}$$

where $\mathbf{A}$ is determined by the finite difference scheme, and is sparse, banded-diagonal matrix; In particular for 5-point laplacian method, A is a matrix with 4 on the diagonal and -1 on the sub-diagonals; u is vector of length $m*n$. $\tilde{f}(x,t)$ is the homogeneous term plus the adapting terms for boundary conditions.

To solve the system of ODEs, we employ backward Euler method as an example of implicit methods, which is characterized by solving a linear system at each time step and yields a total LTE error of order $O(h^2) + O(\Delta t)$. For time step k:

$$(I - \Delta t\mathbf{A})U^{k+1} = U^k + \hat{f}(x,t)\Delta t \tag{3}$$

where $U^k = U(t = k\Delta t)$.

Note $\hat{A} = I - \Delta t\mathbf{A}$ which defines the linear system to solve at each time step.

$$\begin{bmatrix} C & B & & & & \\ B & C & B & & & \\ & B & C & B & & \\ & & & \ddots & & \\ & & & B & C & B \\ & & & & B & C \end{bmatrix}_m$$

where C =

$$\begin{bmatrix} 1+4\frac{\Delta t}{h^2} & -\frac{\Delta t}{h^2} & & & \\ -\frac{\Delta t}{h^2} & 1+4\frac{\Delta t}{h^2} & -\frac{\Delta t}{h^2} & & \\ & & \ddots & & \\ & & -\frac{\Delta t}{h^2} & 1+4\frac{\Delta t}{h^2} & -\frac{\Delta t}{h^2} \\ & & & -\frac{\Delta t}{h^2} & 1+4\frac{\Delta t}{h^2} \end{bmatrix}_n$$

B =

$$\begin{bmatrix} -\frac{\Delta t}{h^2} & & \\ & \ddots & \\ & & -\frac{\Delta t}{h^2} \end{bmatrix}_n$$

The boundary condition term is $\hat{f}$ =

$$\begin{bmatrix} u_{0,1} + u_{1,0} \\ u_{0,2} \\ \vdots \\ u_{0,n-1} \\ u_{0,n} + u_{1,n+1} \\ u_{2,0} \\ 0 \\ \vdots \\ 0 \\ u_{2,n+1} \\ \vdots \\ u_{m+1,1} + u_{m,0} \\ u_{m+1,2} \\ 0 \\ \vdots \\ 0 \\ u_{m+1,n-1} \\ u_{m+1,n} + u_{m,n+1} \end{bmatrix}_{m*n}$$

Factorizing matrix $\hat{A}$ enables us to solve this linear system quickly at each time step. We can show that matrix $\hat{A}$ is a symmetric positive definite matrix of dimension $m*n$, which make it possible to use Cholesky Decomposition. A version of Cholesky Decomposition Algorithm that avoids the computation of square roots is the LDL Cholesky Decomposition $\hat{A} = LDL^T$, where L is lower triangular matrix with 1 on the diagonal and D is a diagonal matrix.

$$D_j = \hat{A}_{jj} - \sum_{k=1}^{j-1} L_{jk}^2 D_k$$

$$L_{ij} = \frac{1}{D_j}\left(\hat{A}_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}D_k\right), \text{for } i > j$$

With the LDLt decomposition given, the linear system that need to be solved at each time step is:

$$LDL^T U^{k+1} = U^k + \hat{f}(x,t)\Delta t \tag{4}$$

We can solve the linear system (4) by simply applying forward substitution, element-wise division, followed by backward substitution.

Forward Substitution:

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} L_{ij}x_j}{L_{ii}}, \quad \text{for } i = 1, 2, \ldots, n$$

Backward Substitution:

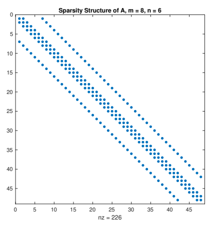
$$x_i = \frac{b_i - \sum_{j=i+1}^{n} L_{ij}^T x_j}{L_{ii}^T}, \quad \text{for } i = n, n-1, \ldots, 1$$

For general cases the boundary conditions also need to be computed at each time step, but since we keep boundary condition to be constant with respect to time this step is not required.

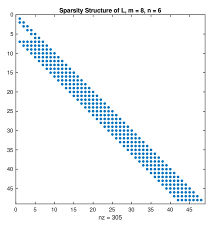
3 SEQUENTIAL IMPLEMENTATION

3.1 CSR Matrix to store sparse matrix

The finite difference method for 2D heat equation suffers from the curse of dimension, that matrix A has dimension $(mn) \times (mn)$, so it is necessary to store the sparse matrices with more efficient memory structure. Since all computation steps in our algorithms requires adding in row direction, CSR format is adequate to store the data information.



(a) A



(b) L

Figure 1: Sparsity structure of A and L

CSRMatrix is defined as:

```
struct CSRMatrix {
  double* values;   // Array storing non-zero values
                       of the matrix
   int* columns;     // Array storing column indices
                       of non-zero values
    int* row_ptr;      // Array storing row pointers
                    (indices where each row starts)
   int rows;         // Number of rows in the matrix
   int capacity;      // maximum number of non-zeros
   int non_zeros;     // Number of non-zero elements
            in the matrix; Essentially "Capacity";
}
```

We also define some help functions such as `get_ij(i,j)` and `set_ij(i,j)` to provide interface when dealing with matrices.

3.2 Initialize CSRMatrix with sparsity structure

Read and write in general CSR format matrices cost $O(n)$, but with observed sparsity structure, it is possible to implement read and write in $O(1)$ just as in array implementation, by pre-allocating all non-zero terms.

From practice, it is observed that L has a fixed sparsity structure, which is a n-banded, lower diagonal matrix. We initialize matrix L by setting all non-zero values accordingly.

3.3 Sequential Algorithm

From the sparsity structure of L: $L_{ij} = 0$ for $i < j$ and for $i > j - n$, we can rewrite the formula from (1, 2)

$$D_j = \hat{A}_{jj} - \sum_{k=j-n}^{j-1} L_{jk}^2 D_k$$

$$L_{ij} = \frac{1}{D_j}\left(\hat{A}_{ij} - \sum_{k=j-n}^{j-1} L_{ik} L_{jk} D_k\right), \text{ for } j + n > i > j$$

Accordingly, we write sequential codes to realize the LDL Decomposition, and the pseudo-code is given in **Algorithm 1**

Solving this linear system can be broke down into three steps: forward substitution, elementwise division, and backward substitution.

Algorithm 1 Cholesky Decomposition (L, Lt, D) = chol(A, m, n)

```
Require: A, m, n
  init(L), init(Lt), init(D)
  $j \leftarrow 0$
  repeat
    $sumL \leftarrow 0$
    $k \leftarrow 0$
    repeat
      $sumL \leftarrow sumL + L_{jk} * L_{jk} * D_k$
      $k \leftarrow k + 1$
    until $k = m * n$
    $D_j \leftarrow A_{jj} - sumL$
    $L_{jj} \leftarrow 1$
    $Lt_{jj} \leftarrow 1$
    $i \leftarrow j + 1$
    repeat
      $sumL2 \leftarrow 0$
      $k \leftarrow max(0, j - n)$
      repeat
        $sumL2 \leftarrow sumL2 + L_{ik} * L_{ik} * D_k$
        $k \leftarrow k + 1$
      until $k = j$
      if $A_{ij} = sumL2$ then
        $L_{ij} \leftarrow (A_{ij} - sumL2)/D_j$
        $Lt_{ji} \leftarrow (A_{ij} - sumL2)/D_j$
      end if
      $i \leftarrow i + 1$
    until $i = j + n + 1$
    $j \leftarrow j + 1$
  until $j = m * n$
```

Algorithm 2 Backward Euler u = be(u, dt, h, endT, m, n)

```
Require: u, dt, h, endT, m, n
  init(A), init(f)
  (L, Lt, D) = chol(A, m, n)
  $t \leftarrow 0$
  while $t < endT$ do
    $b \leftarrow u + dt/h/h * f$
    solve(L, Lt, D, b, u)
    $t \leftarrow t + dt$
  end while
```

4 CUDA IMPLEMENTATION: PARALLELIZATION STRATEGIES

4.1 Initialization:

The initialization is crucial for CSR matrix setup, which eliminate the need of inserting new non-zero values into the CSR structure. Our algorithm first initialize the matrices A and L based on the known sparsity structure, which is trivially parallelizable. Leveraging this inherent structure, we parallelize the initialization process in GPU. This ensures an efficient setup of the CSR matrices, which facilitates subsequent parallel computation.

4.2 Cholesky Decomposition:

The core of our focus lies in the parallelization of the Cholesky decomposition steps, given its significant computational demand. Despite the sequential nature of Cholesky LDL decomposition, we identify a parallelizable pattern within the computation of the sum of elementwise product, as well as the computation of each element in each column.

As illustrated by Figure 2, the known entries are marked yellow, the entry being computed is marked red, and the depended entries are marked in green and blue. The first subfigure at each row indicate computation of diagonal elements, and the following subfigures indicate computation of elements below diagonal.

It is observed that the computation of each element below diagonal is depended on the jth row of L (indicated in blue, which can be shared across all elements) and the ith row of L (indicated in green)

In real practice, the Cholesky decomposition dominates the computational workload, which proves the necessity of our strategy exploiting parallelism to the granular-level calculation.

4.2.1 *Cholesky Decomposition Step: Update Diagonal Elements.*

This is basically a reduction sum of element-wise products. We choose **gridsize = 1** and **constant block_size**, which means we use one thread to compute one product; and store it in a shared memory array; Then we perform a reduction sum and assign the value to L_{jj}, the pseudocode for the algorithm is described below:

```
// gridsize = 1; blocksize = fixed_block_size
// load into shared
__shared__ double sdata[blocksize];
tid = blockDim.x * blockIdx.x + threadIdx.x;
for (int local_k = 0; local_k < J; local_k+=blockDim.x)
{
    global_k = Lcolumns[Lrow_ptr[J]]
        + local_k*blockDim.x+threadIdx.x;
    Ljk = L.get_ij(J, global_k);
    sdata[tid] += Ljk * Ljk * D[global_k];
}
__syncthreads();
// reduction by sequential addressing
for (int s = block_size/2; s > 0; s >>= 1)
{
    if (tid < s) sdata[tid] += sdata[tid + s];
    __syncthreads();
}
// write back
if (tid == 0) {
    // AJJ is a fixed value
    AJJ = 1 + 4 * invhsq_dev * tau_dev;
    D[J] = AJJ - sdata[0];
    Lvalues[Lrow_ptr[J+1]-1] = 1; // Ljj = 1;
}
```

4.2.2 *Cholesky Decomposition Step: Update Elements Below Diagonal.*

Although the overall decomposition steps depend on the results of prior iterations, the calculation of sum within each step is inherently parallel. As the observation above indicated, for the computation of jth column, each element below the diagonal is independent from each other; Also, the computation of reduction sum is also parallelizable, which leads to a two-level parallelize strategy; Depending on the number of n, we proposed two possible designs of grid size and kernel size:

(1) For n value smaller than total blocks available, we can use one block to compute each element under diagonal, and use a fix number of threads per block to compute reduction sum to make sure the shared memory per block does not exceed maximum. In our final application, we applied this algorithm.
(2) For n value larger than the total blocks available, we can use $one2-dimensionalblock$ which covers the $n \times n$ triangle, and the number of threads need to be $n * (n-1)/2$. The indexing in this case is a bit complicated so we haven't finish it yet.
(3) For n value that $n * (n-1)/2$ is larger than the number of maximum threads per block, we need to use a fixed number of block and threads such that each block compute more than one entry. Due to the complexity of indexing in this case, we haven't produce a reasonable parallel solver.

```
// gridsize = n, blocksize fixed
// Compute Elements below Column J
// Assume L and Lt have zeros pre-allocated
    const double DJ = D[J];

// use shared memory for sum
    extern __shared__ double suml2[];
    suml2[threadIdx.x] = 0;
    __syncthreads();

// compute the sum
    const int global_i = J + 1 + blockDimx.x;
    int ith_row_len = row_len - 1 - threadIdx.x;
    if (global_i < n) ith_row_len = n;

    for (int local_k = threadIdx.x;
        local_k < ith_row_len;
        local_k += blockDim.x)
    {
      const int global_k = global_i - col_len + local_k;
        if (global_k >= 0 && global_k < J) {
            Lik = L.get_ij(global_i, global_k);
            Ljk = L.get_ij(J, global_k);
            Dk = D[global_k];
            suml2[threadIdx.x] += Lik * Ljk * Dk;
        }
    }
    __syncthreads();

// reduction by sequential addressing
    for (int s = blockDim.x/2; s > 0; s >>= 1)
    {
        if (threadIdx.x < s)
        {
          suml2[threadIdx.x] += suml2[threadIdx.x + s];
        }
        __syncthreads();
    }

// assign to L_ij and Lt_ij
    if (threadIdx.x == 0)
    {
        const double Aij = A.get_ij(global_i, J);
        double val = (Aij - suml2[0]) / DJ;
        L.set_ij(L, global_i, J, val);
        L.set_ij(Lt, J, global_i, val);
    }
```

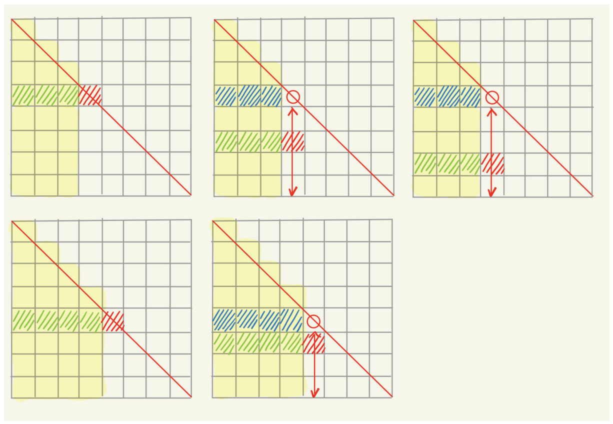


Figure 2: dependency of LDL decomposition

4.3 Backward Euler Steps

In our experiment, we did not include the parallerization of Backward Euler steps, but we recognize the possibility of parallelizing the sum within Forward/Backward substitution;

4.3.1 Backward Euler Steps: Solving Linear system. The parallelization of elementwise division is trivial; Despite the dependency between column-wise update steps, the sum of element-wise product operation in forward and backward substitution are also inherently parallelizable.

4.3.2 Backward Euler Steps: Computation of boundary elements. The update of boundary terms during the Backward Euler steps involve computing the boundary value and adding two vectors, which is clearly a parallelizable task. As each boundary element's update is independent of the others, we can exploit parallelism

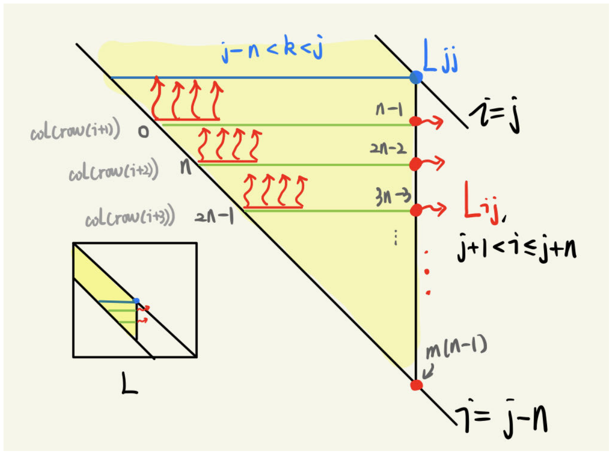


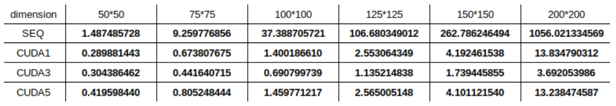
Figure 3: Parallize computation of off-diagonal terms

to concurrently compute these updates. This straightforward parallelization can accelerates the computation algorithm in spatial domain.

5 EXPERIMENT

5.1 Experiment Settings

The sequential code and CUDA code are performed separately on NYU Greene HPC server. We tested by solving the 2D heat equation on heat grids with dimension $50 * 50$, $75 * 75$, $100 * 100$, $125 * 125$, $150 * 150$, $200 * 200$, and performed 100 Backward Euler steps. The total computation time is computed and plotted.



dimension	50*50	75*75	100*100	125*125	150*150	200*200
SEQ	1.487485728	9.259776856	37.388705721	106.680349012	262.786246494	1056.021334569
CUDA1	0.289881443	0.673807675	1.400186610	2.553064349	4.192461538	13.834790312
CUDA3	0.304386462	0.441640715	0.690799739	1.135214838	1.739445855	3.692053986
CUDA5	0.419598440	0.805248444	1.459771217	2.565005148	4.101121540	13.238474587

Figure 4: Table: average runtime

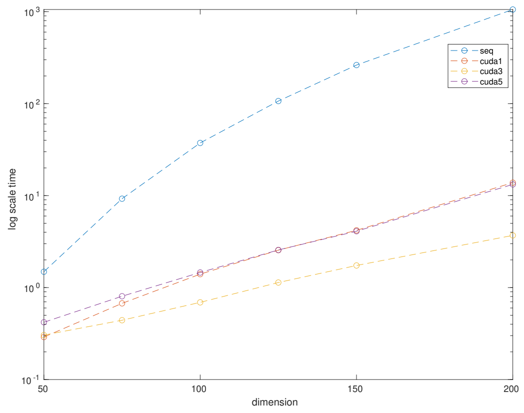


Figure 5: Plot: CPU vs GPU runtime

The sequential code is performed on single CPU on NYU Greene without optimization.

The CUDA code is performed on single GPU on NYU Greene, and the GPU configurations are provided in the appendix.

5.2 Experiment Results

We test three times for sequential version and each cuda cluster and compute the average time. The result is shown in table (4) and plot (5).

5.3 Experiment Analysis

5.3.1 Observation: As the graph indicated, the run time of the sequential code grow exponentially as the dimension of the heat map grows;

In comparison, the performance of the GPU code is growing much slower than sequential code.

5.3.2 Limitation: For larger n value, however, our algorithm that pick **gridDimx** = **n** may be limited by the maximum number of blocks per grid. Also, since we used shared array to compute the sum, the total shared memory available in GPU is also a limiting factor.

6 APPENDIX

6.1 GPU Configuration

6.1.1 NYU Greene cuda1:

```
name: NVIDIA GeForce GTX TITAN Black
Compute capability 3.5
total global memory(KB): 6229696
shared mem per block: 49152
regs per block: 65536
warp size: 32
max threads per block: 1024
max thread dim x:1024 y:1024 z:64
max grid size x:2147483647 y:65535 z:65535
clock rate(KHz): 980000
total constant memory (bytes): 65536
multiprocessor count 15
integrated: 0
async engine count: 1
memory bus width: 384
memory clock rate (KHz): 3500000
L2 cache size (bytes): 1572864
max threads per SM: 2048
```

6.1.2 NYU Greene cuda3:

```
name: NVIDIA TITAN V
Compute capability 7.0
total global memory(KB): 12356288
shared mem per block: 49152
regs per block: 65536
warp size: 32
max threads per block: 1024
max thread dim x:1024 y:1024 z:64
max grid size x:2147483647 y:65535 z:65535
clock rate(KHz): 1455000
total constant memory (bytes): 65536
multiprocessor count 80
integrated: 0
async engine count: 7
memory bus width: 3072
memory clock rate (KHz): 850000
L2 cache size (bytes): 4718592
max threads per SM: 2048
```

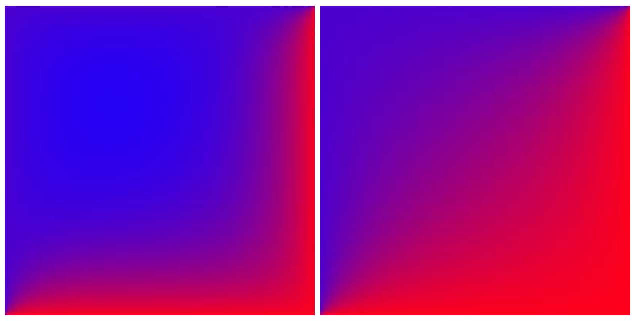
6.1.3 NYU Greene cuda5:

```
name: NVIDIA GeForce GTX TITAN Z
Compute capability 3.5
total global memory(KB): 6229696
shared mem per block: 49152
regs per block: 65536
warp size: 32
max threads per block: 1024
max thread dim x:1024 y:1024 z:64
max grid size x:2147483647 y:65535 z:65535
clock rate(KHz): 915000
total constant memory (bytes): 65536
multiprocessor count 15
integrated: 0
async engine count: 1
memory bus width: 384
memory clock rate (KHz): 3505000
L2 cache size (bytes): 1572864
max threads per SM: 2048
```

6.2 Visualization

Picking heat map grid size $h = 0.01$ and $tau = 0.01$, we visualize the computed solutions at final time $T = 0.1$ and $T = 1$ at resolution $200x200$, setting the top and left boarder boundary condition to be constant 3, right and bottom boundary condition to be constant 10.

The values are linearly interpolated between RGB $(0, 0, 255)$ and $(255, 0, 0)$ color to visualize the heat distribution, which displayed a desired pattern of heat flow.



(a) T=0.1 **(b) T=1**

Figure 6: Visualization of Heat map