Demonstration of the 2D Heat Distribution Problem using CUDA programming model

Ninaad Joshi

3rd December, 2018

Abstract

Benefits of the parallel programming model can be seen in solving differential equations with applications in many streams like Physics, Mathematics, Biology, etc. One of the popular problems is the Heat Distribution problem for a 2-Dimensional thin plate where the boundaries or specific regions are subjected to different but continuous temperatures and the effects of the heat are observed as the temperature in the central region of the plate change continuously until an equilibrium condition is achieved i.e. the temperature of the central region of the plate stabilizes. In order to solve the heat equation, we discretize the plate into unit segments(elements) which we assume will have the same temperature from edges to edges. The temperature for an element can be calculated by averaging the neighbouring segment temperatures. Thus, parallelism can be achieved by keeping track of the temperatures of neighbouring elements and computing the temperatures for each element independently from other elements. This will make the computation of the results much faster than the sequential implementation. Nvidia provides the CUDA programming model to achieve parallelization over GPUs which can perform parallel computations across thousands of cores, thus, considerably reducing the time to solution than the sequential program and generate the final heat map of the plate. Thus, we propose to use the CUDA programming model to parallelize the sequential heat distribution equation.

1 Introduction

When the single processor speedup started stagnating, then the only way to improve the throughput of a system was using parallelism. Thus, parallel computing has been the norm for high throughput computing where problems are solved using low powered parallel processors, shared memory, low latency networks, etc. The parallelism has been made available in the hardware, and to utilize the hardware efficiently, there is a need to understand parallel programming models. Parallel programming models like OpenMP, OpenAcc, CUDA, MPI, etc. are some of the popular models. These models can be used very effectively to increase the efficiency and throughput and at the same time reduce the Time To Solution of a variety of applications. These models can be used in applications which involve scientific equations in mathematics, physics, chemistry, etc. to generate simulations and solve the problems. One of the popular equations to parallelize is the Heat Equation where a constant temperature is applied to a part of a 2D heat plate, and the heat transfer is observed over a period of time. We demonstrate such an example in this project using the CUDA programming model where there is a heat plate with the temperature applied to a specific point on the plate, and we simulate the temperature shift across the plate.

2 Background

The general formula for the heat equation is described as:

$$\frac{\partial u}{\partial t} = k\nabla^2 u \tag{1}$$

The formula in 2D can be described as:

$$u_t = k(u_{xx} + u_{yy}) \tag{2}$$

We discretize the equation by dividing the heat plate into smaller square regions where we assume that the temperature of the square is constant. Thus, to calculate the temperature of the individual square element of the plate we calculate the average of four neighboring elements and then assign the result to the element. This is also called the Jacobi Heat distribution method. The formula for the Jacobi Heat Distribution can be given as:

$$x_c(t+1) = \frac{x_n(t) + x_e(t) + x_s(t) + x_w(t)}{4.0} \tag{3}$$

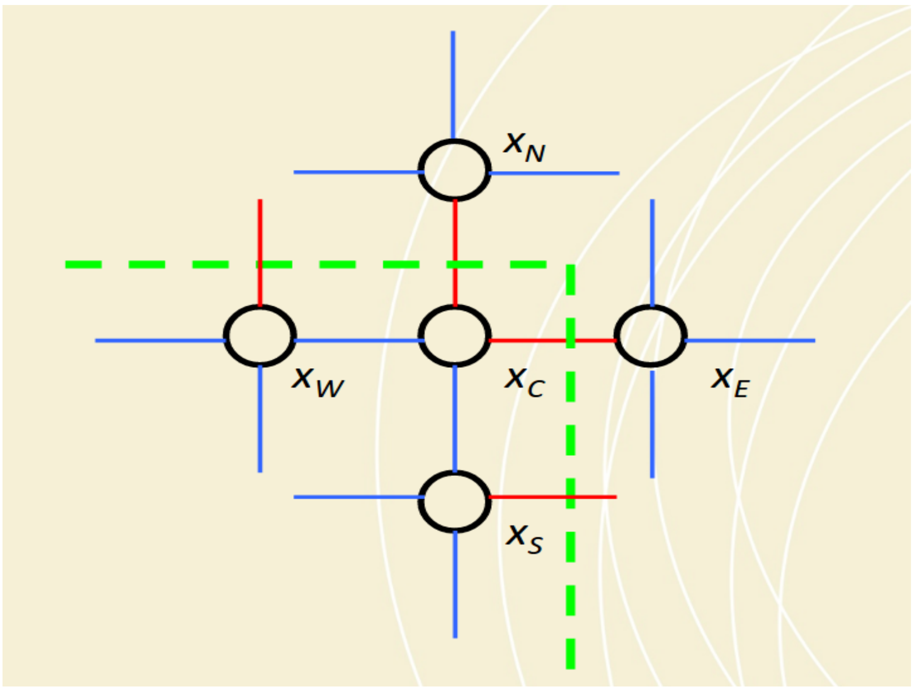


Figure 1: Calculation of the temperature for an individual element considering the neighboring four elements

The boundary conditions are locked and only the elements within the boundaries are considered for the calculations.

The formula (3) is the basis of the simulation for the heat equation. We consider the fact that the temperature of each element can be calculated independently of each other during a single iteration. This allows us to harness the necessary parallelism from the CUDA architecture.

3 CUDA Programming Model

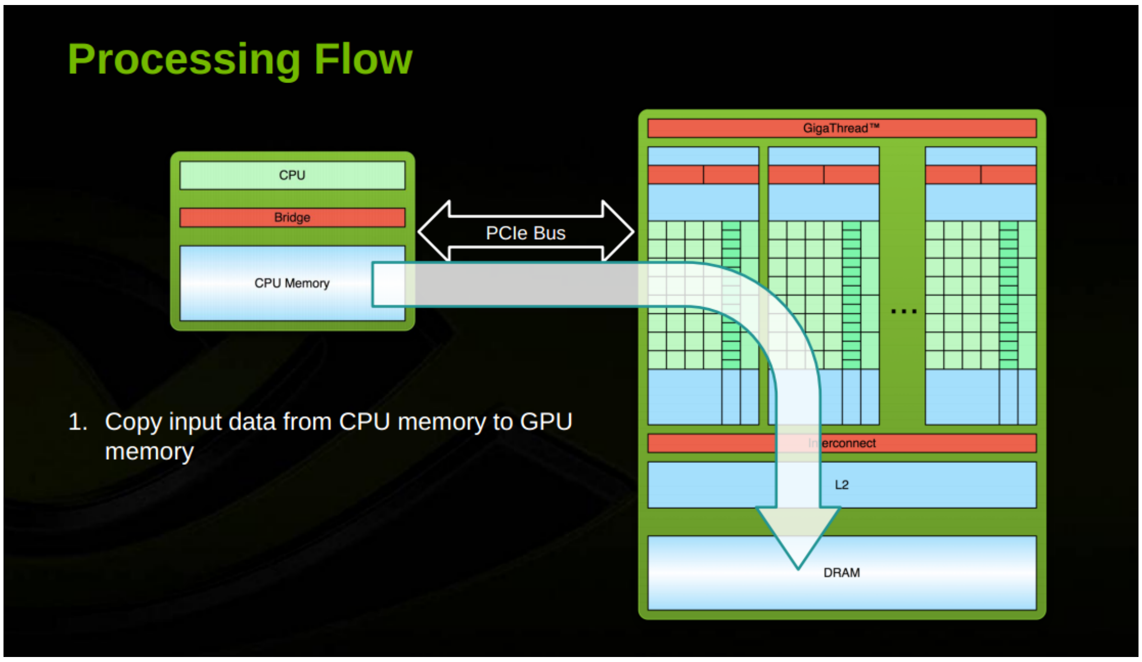
The CUDA programming model is very efficient at creating and executing a lot of threads simultaneously. In the CUDA implementation each element in the heat plate is calculated by an individual thread. The threads all have a unique id and are associated with blocks. Each block can consist of multiple threads. The blocks are further associated with a grid and have unique block ids. Each grid can consist of multiple blocks. Thus, the indexing of each element of the heat plate is done with the help of the block dimensions, block ids and the thread ids. Thus, we can achieve high levels of parallelism for large problem sizes. If OpenMP is used for the same problem, there can be a problem of oversubscription of the threads which has adverse effects on performance. However, even with oversubscription, high performance speedup can be achieved on GPUs.

The CUDA architecture can be divided into two entities - the host and the device. There are three main types of data exchanges between these entities - cudaMemCpyHostToDevice, cudaMemCpyDeviceToDevice and cudaMemCpyDeviceToHost. The data exchange between the Host and the Device are the most expensive operations in terms of performance, as the devices are physically separated in hardware and are connected via the PCI-X bus. Thus, the main objective is to reduce the data traffic between the device and host as much as possible.

The dimensions of the grid and the blocks are calculated based on the number of elements in the heat plate. The CUDA architecture allows a maximum of 512 or 1024(on newer GPUs) threads per block, thus, we choose the number of blocks accordingly for achieving high performance.

The data transfer in between the host and the device is explained in the following stages:

a. Host memory to device memory

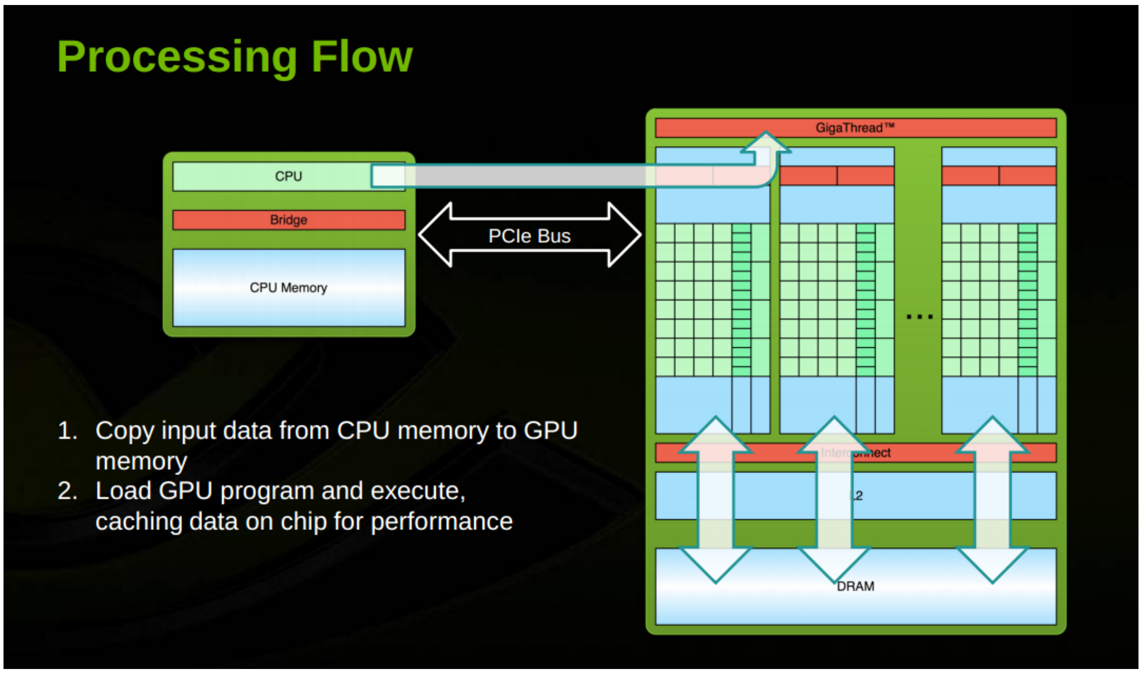


Processing Flow

1. Copy input data from CPU memory to GPU memory

Figure 2: The data is copied from the CPU's main memory to the GPU's dedicated memory

b. CPU to GPU

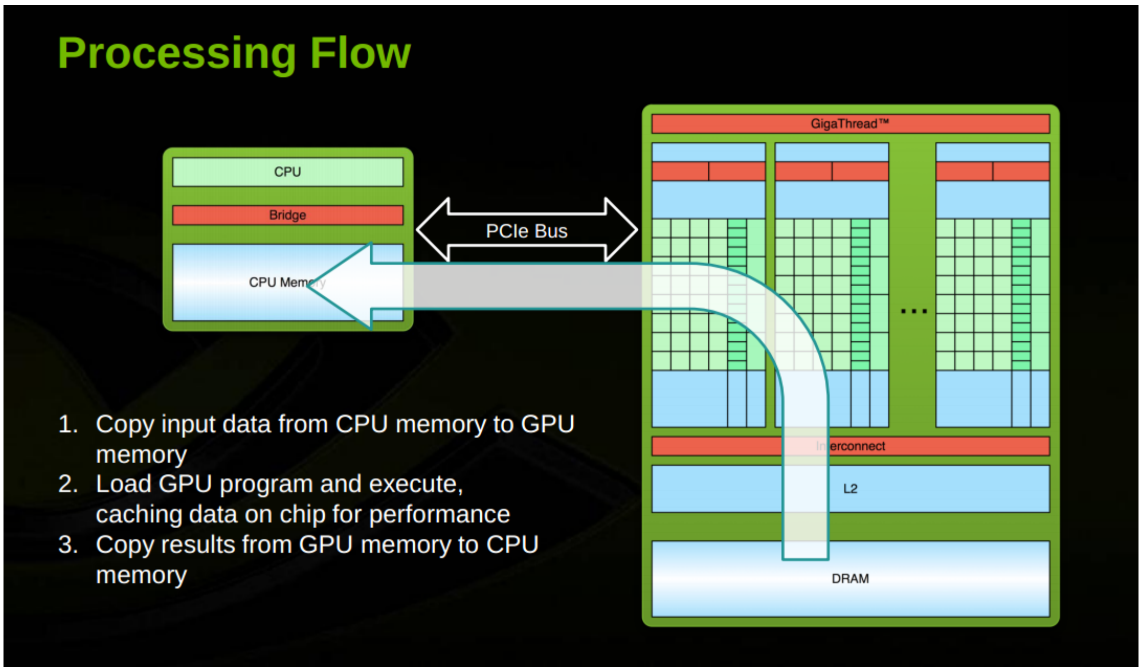


Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Figure 3: The CPU loads the program on the GPU for execution

c. Device memory to host memory



Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

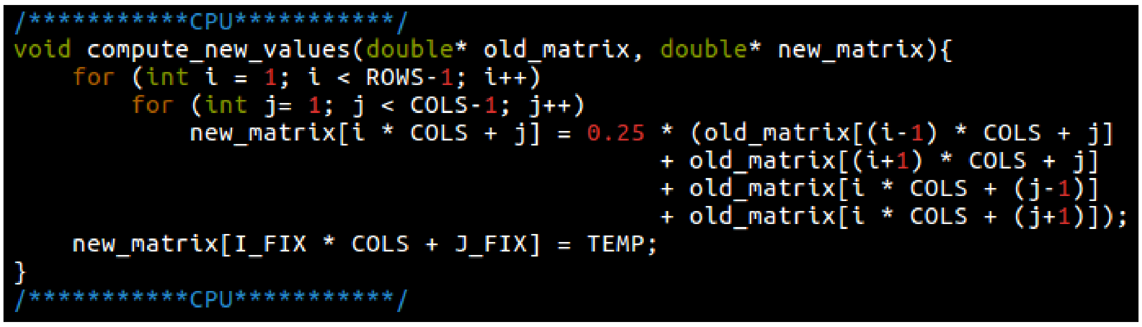
Figure 4: The results are copied from the GPU's memory to the CPU's main memory

4 Implementation

a. Serial Implementation

The sequential implementation of the Jacobi Heat Distribution example is carried out using two 1D matrices for old and new values respectively. The matrices are initialized before the simulation is executed. Then, simulation is started and the new value is fed back into the serial kernel to calculate the new temperature values. The iterations are terminated when the difference between the values of the input plate and the output plate is less than a threshold value called EPS which is the minumum allowable difference between iterations. The number of iterations are counted for achieving this difference.

The following kernel is used for the serial implementation of the Jacobi heat distribution equation as follows:



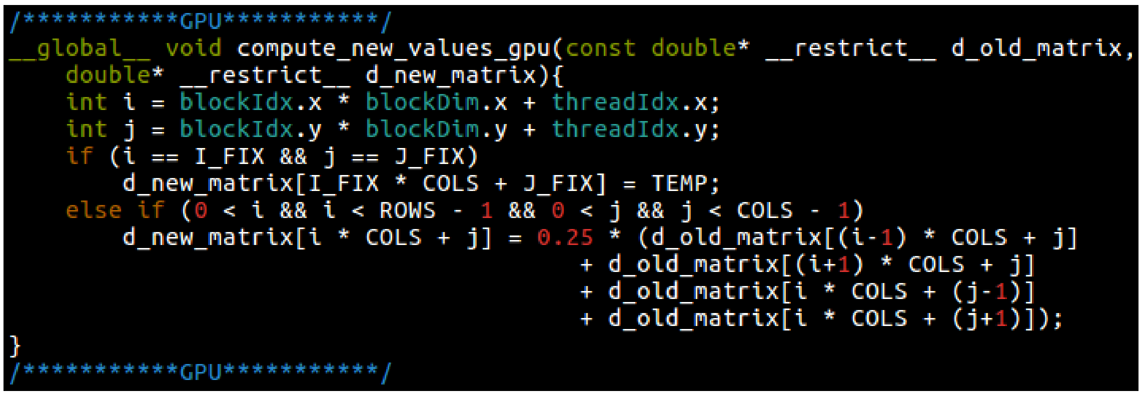
```
/***********CPU************/
void compute_new_values(double* old_matrix, double* new_matrix){
    for (int i = 1; i < ROWS-1; i++)
        for (int j= 1; j < COLS-1; j++)
            new_matrix[i * COLS + j] = 0.25 * (old_matrix[(i-1) * COLS + j]
                                            + old_matrix[(i+1) * COLS + j]
                                            + old_matrix[i * COLS + (j-1)]
                                            + old_matrix[i * COLS + (j+1)]);
    new_matrix[I_FIX * COLS + J_FIX] = TEMP;
}
/***********CPU***********/
```

Figure 5: Serial Implementation of the Jacobi Heat Distribution method

b. CUDA Implementation

In the CUDA implementation, we have access to a lot of threads which can parallelize the computation. Generally, the block dimensions are set to constants and the number of blocks i.e. grid dimensions, are calculated with respected to the block dimensions. For example, in our code, the block dimensions considered are 32 in the x-dimension, 32 in the y-dimension and 1 in the z-dimension(default) as it is a 2D problem. Thus, the number of blocks or grid dimensions are calculated to be the ceiling values of 'ROWS/BLOCK_SIZE_X' in the x-dimension, 'COLS/BLOCK_SIZE_Y' in the y-dimension and the z-dimension is again defaulted to 1.

For the parallel implementation, we have used the Jacobi Heat Distribution method for calculating the temperature for each element as follows:



```
/***********GPU***********/
__global__ void compute_new_values_gpu(const double* __restrict__ d_old_matrix,
    double* __restrict__ d_new_matrix){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i == I_FIX && j == J_FIX)
        d_new_matrix[I_FIX * COLS + J_FIX] = TEMP;
    else if (0 < i && i < ROWS - 1 && 0 < j && j < COLS - 1)
        d_new_matrix[i * COLS + j] = 0.25 * (d_old_matrix[(i-1) * COLS + j]
                                   + d_old_matrix[(i+1) * COLS + j]
                                   + d_old_matrix[i * COLS + (j-1)]
                                   + d_old_matrix[i * COLS + (j+1)]);
}
/***********GPU***********/
```

Figure 6: Serial Implementation of the Jacobi Heat Distribution method

5 Results

The calculations for resultant matrix are shown as outputs which can be enabled by setting the DISPLAY macro in the Makefile provided. The time difference is calculated for the serial and the parallel CUDA execution, and the speedup is shown as the output of the program. For debugging purposes, we have kept additional steps which can result in poor performance as there is a lot of data exchange between the CPU and the GPU. Thus, the speedup should be considered only when the DEBUG flag is cleared. For a small problem size like the 10x10 heat plate used to calculate the following heat maps, the overhead of moving the data to and from the GPU is very high, hence, the speedup was computed to be 0.198.
The following are the heat maps for the initial and the final heat plates for a 10x10 size:

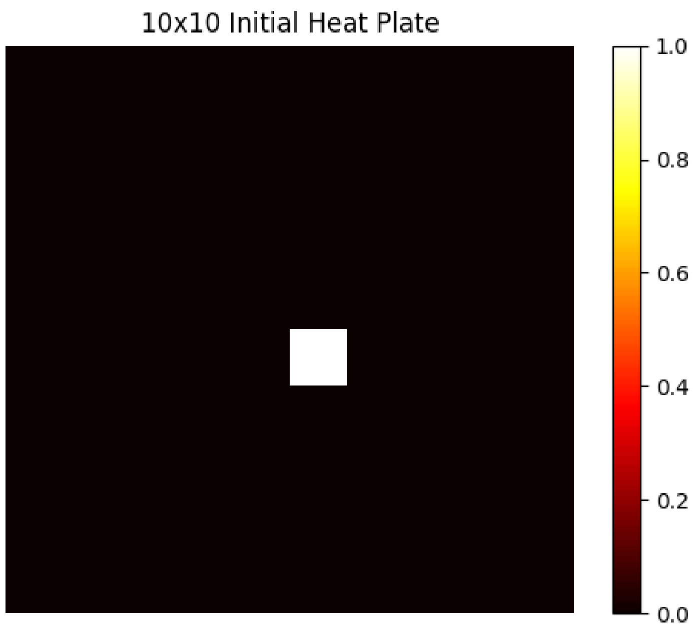


Figure 7: Initial stage of the heat plate

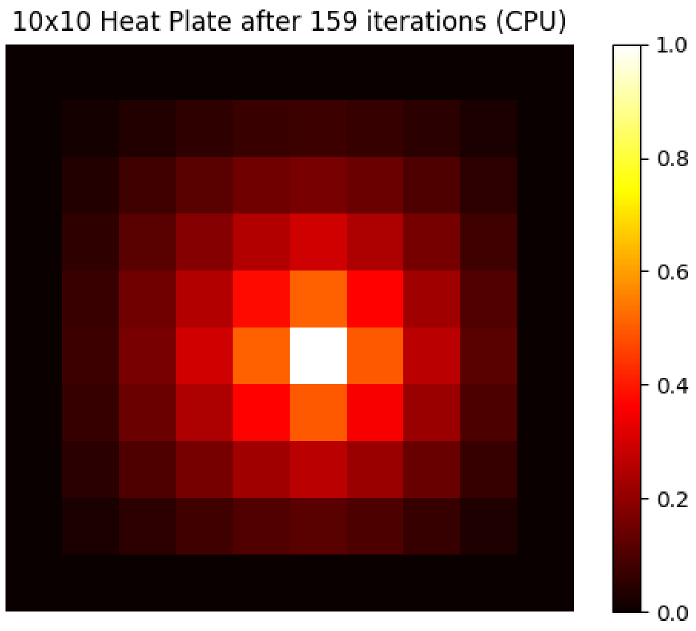


Figure 8: Final state of the heat plate after 159 iterations on CPU

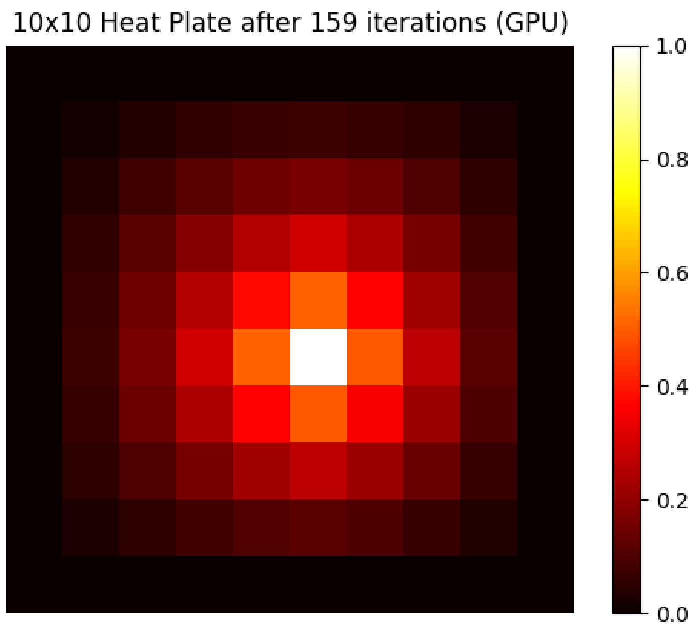


Figure 9: Final state of the heat plate after 159 iterations on GPU

For larger problems, the speedup is non-trivial and extreme. The graphs of the time to solution vs the problem size are shown below. From the graph, we can infer that there is a very high speedup as the problem size increases.

a. One Dimensional Scaling Results

The graph for one dimensional scaling is show below.

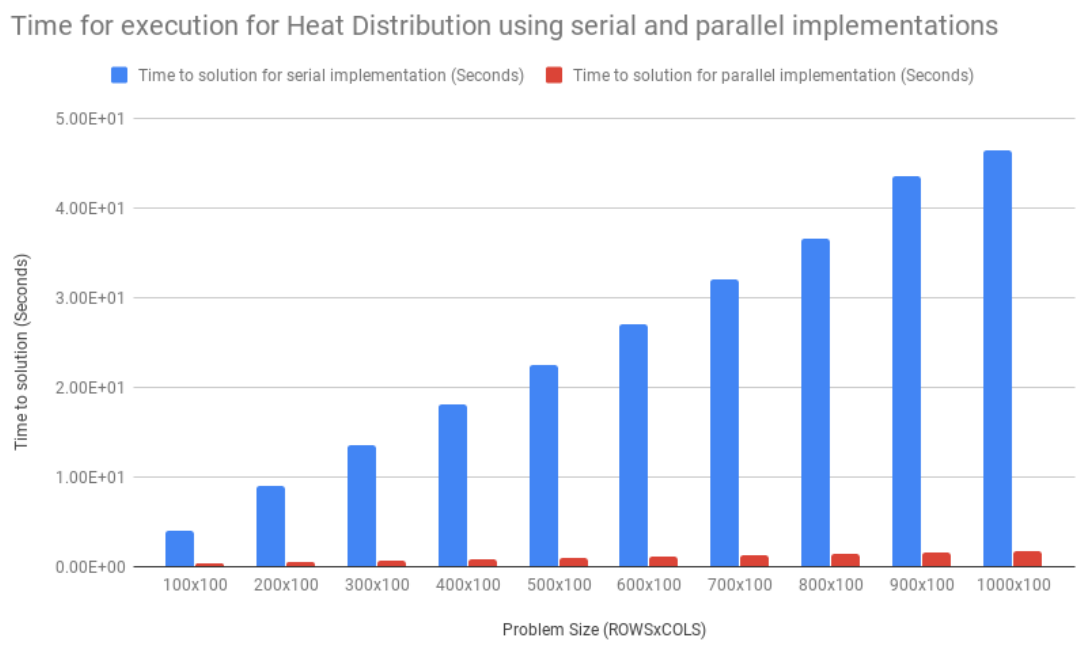


Figure 10: The Graph of time to solution (TOS) vs the problem size. Here the problem is scaled in one dimension i.e. the ROWS dimension

The table of readings for one dimensional scaling is shown below.

Problem Size (ROWSxCOLS)	Time to solution for serial implementation (Seconds)	Time to solution for parallel implementation (Seconds)	Number of Iterations	Speedup
100x100	3.91E+00	2.91E-01	12101	13.449
200x100	8.99E+00	5.35E-01	15061	16.798
300x100	1.35E+01	6.42E-01	15091	21.058
400x100	1.82E+01	7.77E-01	15091	23.365
500x100	2.25E+01	9.61E-01	15091	23.392
600x100	2.71E+01	1.12E+00	15091	24.257
700x100	3.20E+01	1.20E+00	15091	26.57
800x100	3.67E+01	1.40E+00	15091	26.119
900x100	4.36E+01	1.52E+00	15091	28.674
1000x100	4.65E+01	1.65E+00	15091	28.146

Figure 11: Table containing one dimensional scaling results

b. Two Dimensional Scaling Results

The graph for two dimensional scaling is show below.

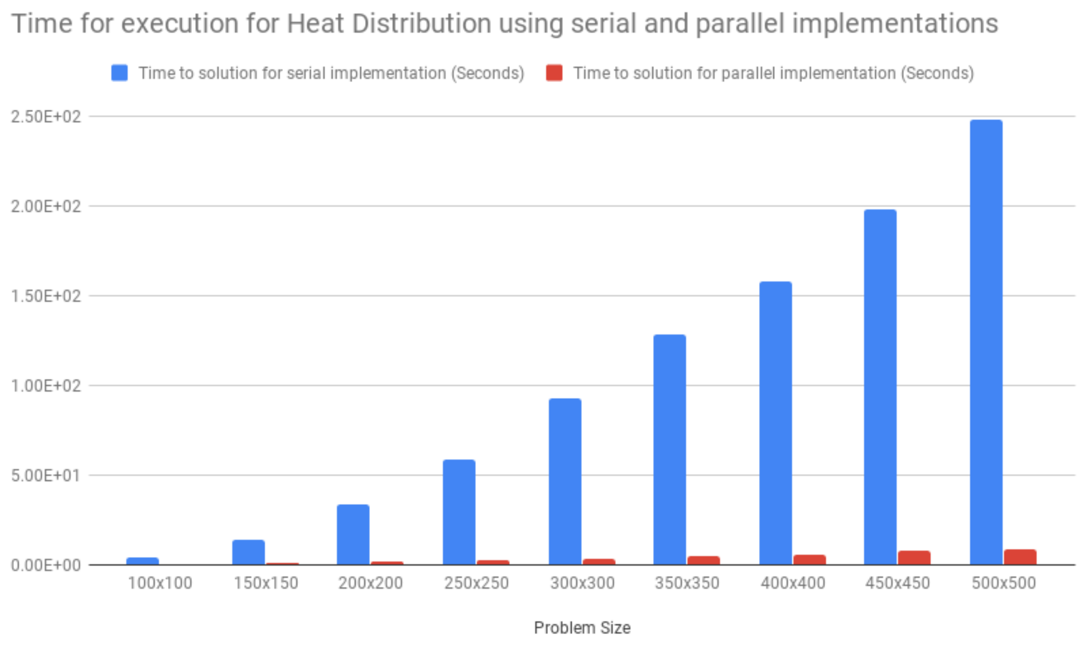
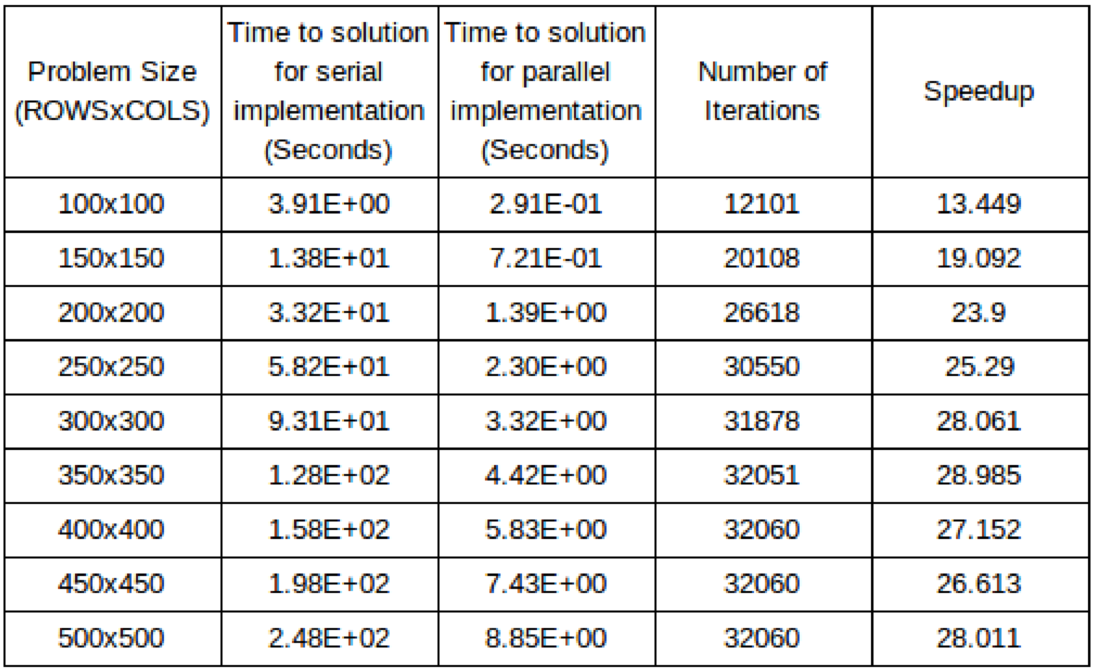


Figure 12: The Graph of time to solution (TOS) vs the problem size. Here the problem is scaled in both dimensions i.e. the ROWS dimension and the COLS dimension

The table of readings for two dimensional scaling is shown below.



Problem Size (ROWSxCOLS)	Time to solution for serial implementation (Seconds)	Time to solution for parallel implementation (Seconds)	Number of Iterations	Speedup
100x100	3.91E+00	2.91E-01	12101	13.449
150x150	1.38E+01	7.21E-01	20108	19.092
200x200	3.32E+01	1.39E+00	26618	23.9
250x250	5.82E+01	2.30E+00	30550	25.29
300x300	9.31E+01	3.32E+00	31878	28.061
350x350	1.28E+02	4.42E+00	32051	28.985
400x400	1.58E+02	5.83E+00	32060	27.152
450x450	1.98E+02	7.43E+00	32060	26.613
500x500	2.48E+02	8.85E+00	32060	28.011

Figure 13: Table containing two dimensional scaling results

6 Correctness

To check the correctness of CUDA implementation, the outputs of both the serial and the CUDA implementation are compared and the maximum difference is stored in a 'max_val' variable. The tolerance is set by defining the 'EPS' variable in the code. Initially, the sequential code is used to generate the number of iterations to get the required level of temperature difference precision using 'EPS'. Then, the CUDA implementation is run for the same number of iterations and then the outputs of the serial and the parallel codes are compared and the maximum difference is stored in the 'max_val' variable. The value of the 'max_val' variable is printed in the output. The difference, if other than zero, indicates that the computations are producing inconsistent output plates. However, the difference is zero for our example.

7 Conclusion

We have demonstrated the 2D Heat Distribution problem by solving the problem using serial and CUDA programming models. The observation is that the CUDA implementation provides and extreme level of speedup over the serial implementation using the CUDA architecture to its advantage. The problem is visualized over multiple problem sizes and the speedup is compared accordingly.

8 Future Scope

The project can be further extended to include implementations in OpenMP, OpenAcc and MPI programming models and the results can be compared to check the most efficient solution to the problem. The results can be further improved using techniques like cache blocking, etc. Also, other problems like simulation of a water drop landing on the water surface can be simulated, or the effects of nuclear reactions at atomic levels can be simulated using the equations of Physics.

References

1. https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
2. https://en.wikipedia.org/wiki/Heat_equation
3. https://devblogs.nvidia.com/using-shared-memory-cuda-cc/
4. https://devtalk.nvidia.com/default/topic/411575/gpu-vs-cpu-gpu-is-always-much-slower/
5. http://web.mit.edu/pocky/www/cudaworkshop/HeatDist/HeatCUDA-G_A.cu
6. https://stackoverflow.com/questions/9985912/how-do-i-choose-grid-and-block-dimensions-for-cuda-kernels
7. Heat equation image borrowed from IU HPC slides for Communicating Sequential Processes (https://iu.app.box.com/file/311811426363)
8. Data flow images borrowed from http://frankenthespian.com/SEMINAR/Seminar.html