# Introduction and problem specification

Graphics Processing Units (GPUs) are specialized processors originally designed to accelerate graphics rendering through its high bandwidth and massive parallel execution resources. GPUs optimise performance by parallelizing the execution of compute-intensive sections within applications making them more efficient than general purpose Central Processing Units (CPUs). However, most computer applications contain sequential elements which makes it impractical to run entirely on a GPU. To overcome this, CPUs usually handle the sequential operations while the GPU handles data parallel operations, this method is known as hybrid processing.

This report will demonstrate how a CPU application can be optimised through parallelisation. This is achieved through porting it to CUDA, which is a parallel computing platform and programming model that harnesses the power of GPUs enabling dramatic increases in computing performance. The CPU application at hand is a 2-Dimensional Transmission-Line Matrix (2D TLM) method, a method of analysis which represents a true computer simulation of wave propagation in the time domain within transmission line networks. The following sections will further describe the algorithm and which parts of it are parallelizable, then will illustrate porting it, and graphically represent any performance improvements observed.

# CPU Code

The 2D TLM CPU code at hand can be divided into the following sections:

- The 2D TLM algorithm

It can be seen in the first figure that the 2D TLM algorithm exists entirely within a counter controlled loop in main(), and thus is executed many times. This algorithm consists of 4 main operations: Sourcing, Scattering, Connecting and Applying boundaries. These operations are highly data parallel and can each be isolated as an external functions to be run on the GPU kernel when porting the application to CUDA. A breakdown of the stages of operation of the algorithm is presented below.

- Source Operation

```
59      for (int n = 0; n < NT; n++) {
60
61          //source
62          E0 = (1 / sqrt(2.)) * exp(-(n * dt - delay) * (n * dt - delay) / (width * width));
63          V1[Ein[0]][Ein[1]] = V1[Ein[0]][Ein[1]] + E0;
64          V2[Ein[0]][Ein[1]] = V2[Ein[0]][Ein[1]] - E0;
65          V3[Ein[0]][Ein[1]] = V3[Ein[0]][Ein[1]] - E0;
66          V4[Ein[0]][Ein[1]] = V4[Ein[0]][Ein[1]] + E0;
67
68          .
69          .
```

*Figure 1: Source Process of 2D TLM (CPU).*

The Source operation calculates the value of the gaussian excitation voltages at given points in time and stores it in the variable E0, this value is then used in the following lines to calculate the new voltages of the current iteration and stores it in the 2D arrays (V1 – V4). This is a parallelisable operation and can be isolated into an external function for use by the GPU kernel.

- Scatter Operation

```
68          //scatter
69          for (int x = 0; x < NX; x++) {
70              for (int y = 0; y < NY; y++) {
71                  I = (2 * V1[x][y] + 2 * V4[x][y] - 2 * V2[x][y] - 2 * V3[x][y]) / (4 * Z);
72
73                  V = 2 * V1[x][y] - I * Z;          //port1
74                  V1[x][y] = V - V1[x][y];
75                  V = 2 * V2[x][y] + I * Z;          //port2
76                  V2[x][y] = V - V2[x][y];
77                  V = 2 * V3[x][y] + I * Z;          //port3
78                  V3[x][y] = V - V3[x][y];
79                  V = 2 * V4[x][y] - I * Z;          //port4
80                  V4[x][y] = V - V4[x][y];
81              }
82          }
```

*Figure 2: Scatter Process of 2D TLM (CPU).*

The Scatter operation consists of 2 nested loops, making it highly inefficient in its current state and in turn, highly parallelisable. The operation is dependant on the completion of the previous stage and therefore, should be launched on a separate kernel, as to respect the sequential aspect of the operations.

- Connect Operation

```
84          //connect
85          for (int x = 1; x < NX; x++) {
86              for (int y = 0; y < NY; y++) {
87                  tempV = V2[x][y];
88                  V2[x][y] = V4[x - 1][y];
89                  V4[x - 1][y] = tempV;
90              }
91          }
92          for (int x = 0; x < NX; x++) {
93              for (int y = 1; y < NY; y++) {
94                  tempV = V1[x][y];
95                  V1[x][y] = V3[x][y - 1];
96                  V3[x][y - 1] = tempV;
97              }
98          }
```

*Figure 3: Connect Process of 2D TLM (CPU).*

Similar to the previous operation, the Connect operation also consists of nested loops which makes it highly inefficient to run on a CPU, especially if a large number of iterations are to be used. Moreover, this operation is also dependant on the completion of the Scatter operation and so must be launched on a separate kernel to ensure the scattering is complete before the connecting starts.

- Apply boundaries

```
100         //boundary
101         for (int x = 0; x < NX; x++) {
102             V3[x][NY - 1] = rYmax * V3[x][NY - 1];
103             V1[x][0] = rYmin * V1[x][0];
104         }
105         for (int y = 0; y < NY; y++) {
106             V4[NX - 1][y] = rXmax * V4[NX - 1][y];
107             V2[0][y] = rXmin * V2[0][y];
108         }
```

*Figure 4: Boundary Process of 2D TLM (CPU).*

Finally, the final stage of the 2D TLM algorithm is applying the boundaries, once again this stage contains nested loops which would require parallel execution to make the code more efficient. However, unlike the previous operations, applying the boundaries is not dependant on the completion of the previous stage, and therefore, can run on the same kernel as the Connect operation, or better yet, in the same loop. This will reduce the total number of iterations happening during program execution and will further optimise the code.

- External function " declare_array2D "

```
124
125 double** declare_array2D(int NX, int NY) {
126     double** V = new double* [NX];
127     for (int x = 0; x < NX; x++) {
128         V[x] = new double[NY];
129     }
130
131     for (int x = 0; x < NX; x++) {
132         for (int y = 0; y < NY; y++) {
133             V[x][y] = 0;
134         }
135     }
136     return V;
137 }
138
```

*Figure 5: External function for 2D array declaration.*

Another section of the program that is of crucial importance is this function. This function simply declares, initiates, and returns a 2 dimensional array. However, due to the fact that this function has a data type " double** ", it cannot be parallelised to run on a GPU kernel, this is because __global__ functions must have a void return type.

Please note: These snippets do not account for the inclusion and declaration of necessary headers and variables.

## Benchmark Results

### Measured Voltage

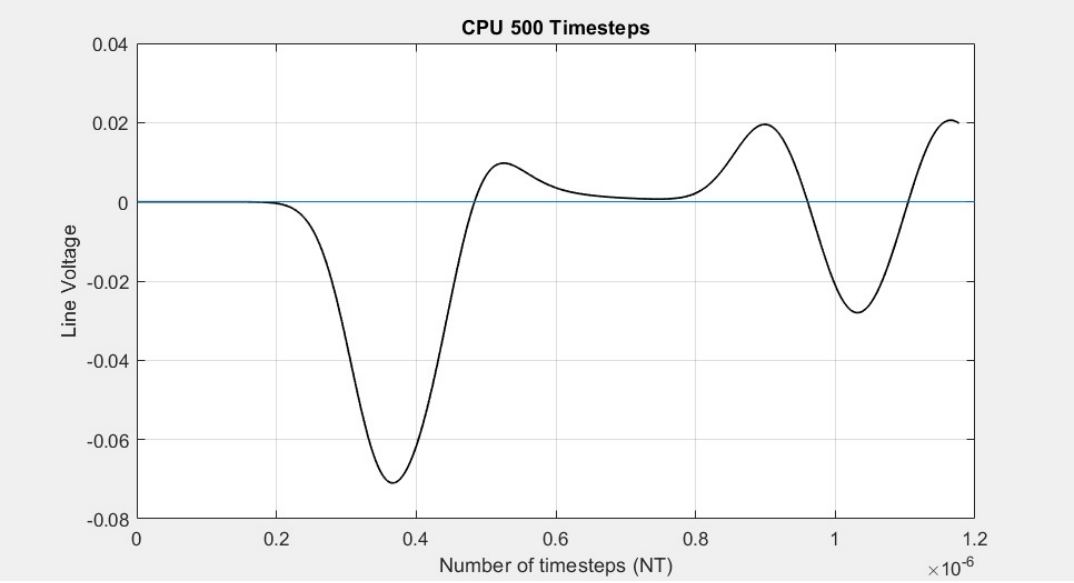


*Figure 6: Runtime = 0.214s*

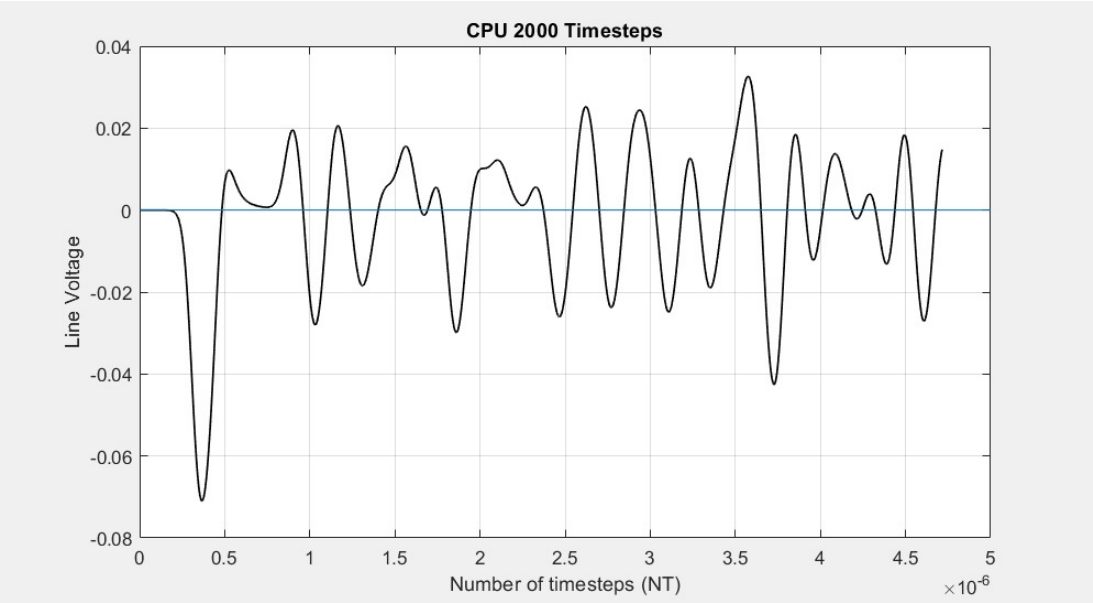


*Figure 7: Runtime = 0.843s*

## Performance

| Number of Iterations (NT) | CPU Computational Time |
|---|---|
| 800 | 0.351 |
| 1600 | 0.662 |
| 3200 | 1.324 |
| 6400 | 2.725 |
| 10000 | 4.238 |
| 12800 | 5.351 |
| 20000 | 8.327 |
| 25600 | 10.539 |
| 50000 | 20.76 |

*Table 1: CPU code execution time.*

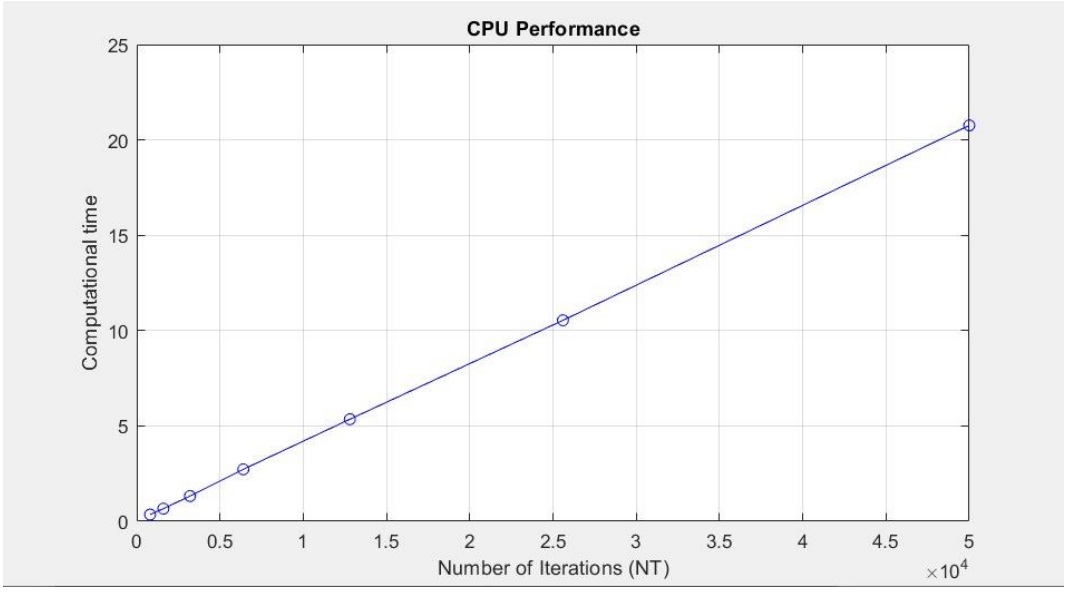


*Figure 8: CPU code execution speed.*

## Porting application to CUDA (GPU code)

In order to successfully port the 2D TLM program to CUDA, each of the stages of operation must first be isolated into a separate external function which can be ported onto a GPU kernel. As such, the first step taken to parallelise the program was to modularise the code. The impact modularisation had on the main() part of the program is shown in the figure below.

```
123  int main() {
124                  .
125                  .
126                  .
127
128    for (int n = 0; n < NT; n++) {
129
130        //source
131        tlmSource(V1, V2, V3, V4, dt, delay, width, n, Ein);
132
133        //shatter
134        tlmScatter(V1, V2, V3, V4, NX, NY, I, Z);
135
136        //connect
137        tlmConnect(V1, V2, V3, V4, NX, NY, tempV);
138
139        //boundary
140        tlmBoundary(V1, V2, V3, V4, NX, NY, rYmax, rYmin, rXmax, rXmin);
141
142                  .
143                  .
```

*Figure 9: Impact of Modularisation on main().*

Following this, the porting of the program to CUDA can commence. Porting a CPU application to CUDA requires the inclusion of certain header files as well as the declaration of device arrays and allocation of memory to them. Relevant snippets are shown in the following figures.

```
2  #include "cuda_runtime.h"
3  #include "device_launch_parameters.h"
4  #define BLOCKSIZE 1024
```

*Figure 10: Additional header files and definitions required for porting.*

```
int main() {

            .
            .
            .

    //device arrays
    double** dev_V1 = declare_array2D(NX, NY);
    double** dev_V2 = declare_array2D(NX, NY);
    double** dev_V3 = declare_array2D(NX, NY);
    double** dev_V4 = declare_array2D(NX, NY);

    //allocate memory on device
    cudaMalloc((void**)&dev_V1, NX * NY * sizeof(double));
    cudaMalloc((void**)&dev_V2, NX * NY * sizeof(double));
    cudaMalloc((void**)&dev_V3, NX * NY * sizeof(double));
    cudaMalloc((void**)&dev_V4, NX * NY * sizeof(double));

    //copy memory from host to device
    cudaMemcpy(dev_V1, V1, NX * NY * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_V2, V2, NX * NY * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_V3, V3, NX * NY * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_V4, V4, NX * NY * sizeof(double), cudaMemcpyHostToDevice);

    // Determination of number of blocks required for the given number of time steps
    // Taking into account the max number of threads per block
    int blockSize = BLOCKSIZE;
    int numBlocks = (NT + blockSize - 1) / blockSize;

            .
            .
```

*Figure 11: Device declaration, memory allocation and other essential additions.*

After these initial stages are complete, each of the operations previously isolated into functions are ported as shown in the following subsections.

One thing to note is the importing of variable declarations from main() into the functions themselves. Although this does not have any notable effect in terms of speed of execution, it does however optimise the use of memory, this is because variables declared within a function are stored temporarily in the stack for the duration of function execution and are deleted at the end of it. Doing this allows for better memory use than when declaring variables in main(), as variables declared in main() are permanently stored at a memory address for the duration of program execution and will each have to be passed to their respective functions with every iteration.

- Source Operation

```
__global__ void tlmSource(double** V1, double** V2, double** V3, double** V4, double time, int n)
{
    // Variable declarations imported from main()
    double width = 20 * time * sqrt(2.);
    double delay = 100 * time * sqrt(2.);
    int Ein[] = { 10,10 };

    //Calculate value of gaussian voltage at time point
    double source;
    source = (1 / sqrt(2.)) * exp(-(n * time - delay) * (n * time - delay) / (width * width));
    V1[Ein[0]][Ein[1]] = V1[Ein[0]][Ein[1]] + source;
    V2[Ein[0]][Ein[1]] = V2[Ein[0]][Ein[1]] - source;
    V3[Ein[0]][Ein[1]] = V3[Ein[0]][Ein[1]] - source;
    V4[Ein[0]][Ein[1]] = V4[Ein[0]][Ein[1]] + source;
}
```

*Figure 12: Source Process of 2D TLM (GPU).*

- Scatter Operation

It can be seen in Figure 13 that minor changes were made to the scatter operation that was previously shown in Figure 2, these include the declaration of variables “idx” and “idy” which are used to store the current thread in the current block ID for both dimensions, as well as changing the counter controlled loop into a condition controlled one.
A combination of threads and blocks is used as this allows for more operations to be carried out simultaneously, this is because blocks are limited to a maximum of 1024 threads and each dimension is limited to 65535 blocks so if only threads were to be used a maximum of 1024 operations can happen at a time instead of utilizing the full potential of a GPU kernel (≈ 1024 * 65535).

```
__global__ void tlmScatter(double** V1, double** V2, double** V3, double** V4, int NX, int NY)
{
    // Variable declarations imported from main()
    double Z = eta0 / sqrt(2.);
    double I = 0;

    // Variable storing the current thread in the current block ID for each dimension
    unsigned int idx = threadIdx.x + blockDim.x + blockIdx.x;
    unsigned int idy = threadIdx.y + blockDim.y + blockIdx.y;
    double V;

    // Scatter Operation
    if (idx < NX) {
        if (idy < NY) {
            I = (2 * V1[idx][idy] + 2 * V4[idx][idy] - 2 * V2[idx][idy] - 2 * V3[idx][idy]) / (4 * Z);

            V = 2 * V1[idx][idy] - I * Z;         //port1
            V1[idx][idy] = V - V1[idx][idy];
            V = 2 * V2[idx][idy] + I * Z;         //port2
            V2[idx][idy] = V - V2[idx][idy];
            V = 2 * V3[idx][idy] + I * Z;         //port3
            V3[idx][idy] = V - V3[idx][idy];
            V = 2 * V4[idx][idy] - I * Z;         //port4
            V4[idx][idy] = V - V4[idx][idy];
        }
    }
}
```

*Figure 13: Scatter Process of 2D TLM (GPU).*

- Connect Operation & Apply Boundaries

In this stage of porting the application, both the Connect and Boundary operation were merged into one decreasing the overall number of iterations happening during program execution, the same minor changes discussed in the previous stage were also required for this stage along with the introduction of temporary variables “tempNX” and “tempNY” which were used to replace [NX - 1] and [NY - 1] seen previously in Figure 4. This is important as it removes the effect the nested loop would have had on the boundary operation. Compared to the previous stages, this stage had the greatest speed up, as it not only parallelises the operations, but it also merges them first into one function, and then into one loop.

```
__global__ void tlmConnect(double** V1, double** V2, double** V3, double** V4, int NX, int NY)
{
    // Variable declarations imported from main()
    double tempV = 0;
    //boundary coefficients
    double rXmin = -1;
    double rXmax = -1;
    double rYmin = -1;
    double rYmax = -1;

    // Variables storing the current thread in the current block ID for each dimension
    unsigned int idx = threadIdx.x + blockDim.x + blockIdx.x;
    unsigned int idy = threadIdx.y + blockDim.y + blockIdx.y;

    int tempNX, tempNY; // Temporary variables to allow merging of Connect and Boundary operations
    tempNX = NX - 1;
    tempNY = NY - 1;

    // Connect Operation
    if (idx < NX) {
        if (idy < NY) {
            tempV = V2[idx][idy];
            V2[idx][idy] = V4[idx - 1][idy];
            V4[idx - 1][idy] = tempV;
        }
    }
    if (idx < NX) {
        if (idy < NY) {
            tempV = V1[idx][idy];
            V1[idx][idy] = V3[idx][idy - 1];
            V3[idx][idy - 1] = tempV;

            V4[tempNX][idy] = rXmax * V4[tempNX][idy];  // Start of boundary operation
            V2[0][idy] = rXmin * V2[0][idy];            //....
        }
        V3[idx][tempNY] = rYmax * V3[idx][tempNY];      //....
        V1[idx][0] = rYmin * V1[idx][0];                // End of boundary operation
    }
}
```

*Figure 14: Merged Connect and Boundary Processes of 2D TLM (GPU).*

The final impact this had on main() is shown below in Figure 15.

```
int main() {

            .
            .
            .

    for (int n = 0; n < NT; n++) {

        //source
        tlmSource << <numBlocks, blockSize >> > (dev_V1, dev_V2, dev_V3, dev_V4, dt, n);

        //shatter
        tlmScatter << <numBlocks, blockSize >> > (dev_V1, dev_V2, dev_V3, dev_V4, NX, NY);

        //connect
        tlmConnect << <numBlocks, blockSize >> > (dev_V1, dev_V2, dev_V3, dev_V4, NX, NY);

            .
            .

    }
```

*Figure 15: Impact of GPU porting on main().*

The last step required to end porting of the application to CUDA is returning the arrays from the devices back to the host and freeing the memory initially allocated to them as shown in the figure below.

```
172                 .
173                 .
174
175         // copy memory from device to host
176         cudaMemcpy(V2, dev_V2, NX * NY * sizeof(double), cudaMemcpyDeviceToHost);
177         cudaMemcpy(V4, dev_V4, NX * NY * sizeof(double), cudaMemcpyDeviceToHost);
178
179         // write output to text file
180         for (int n = 0; n < NT; n++)
181         {
182         output << n * dt << "  " << V2[Eout[0]][Eout[1]] + V4[Eout[0]][Eout[1]] << endl;
183         if (n % 100 == 0)
184             cout << n << endl;
185         }
186
187         // free memory allocated on devices
188         cudaFree(dev_V1);
189         cudaFree(dev_V2);
190         cudaFree(dev_V3);
191         cudaFree(dev_V4);
192
193                 .
194                 .
```

*Figure 16: Final stages of porting.*

Furthermore, another way of optimising this program is by utilizing the different types of memories available, these include the shared memory, which is located on chip and is shared by all threads in a thread block, thus allowing a much faster memory access compared to that of global memory. As such, by declaring the 2D arrays used across all kernels as shared memory variables, a significant improvement in performance can be observed in terms of both memory usage and execution speed.

## Benchmark Results

### Measured Voltage

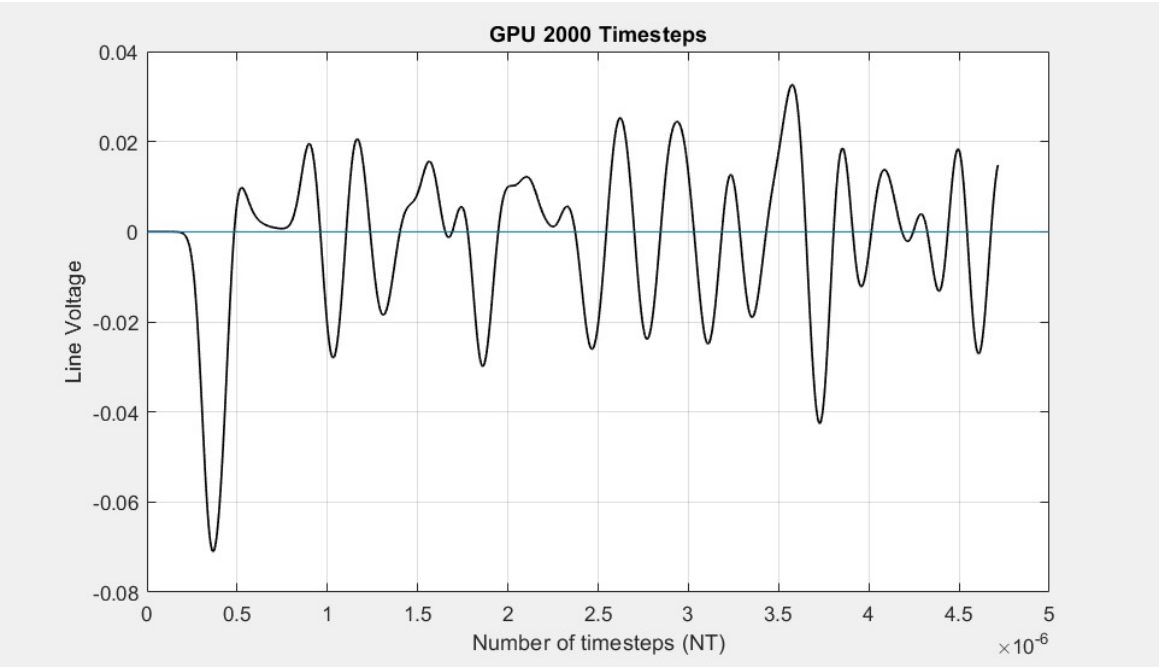


*Figure 17: Runtime = 0.349s*

## Performance

| Number of Iterations (NT) | GPU Computational Time |
|---|---|
| 800 | 0.31 |
| 1600 | 0.318 |
| 3200 | 0.361 |
| 6400 | 0.413 |
| 10000 | 0.545 |
| 12800 | 0.586 |
| 20000 | 0.744 |
| 25600 | 0.92 |
| 50000 | 1.467 |

*Table 2: GPU code execution time.*

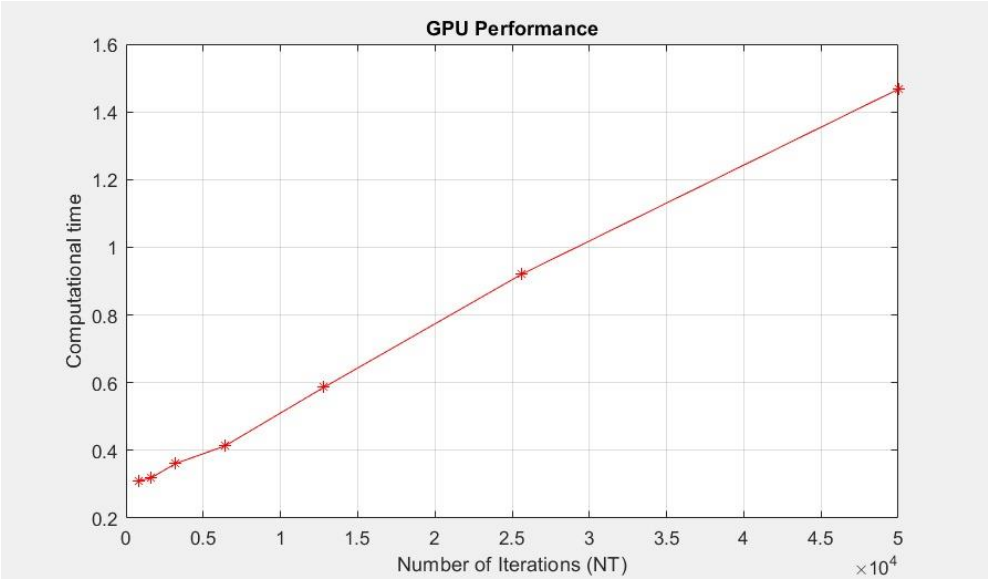


*Figure 18: GPU code execution time.*

## Comparison to CPU Performance

| Number of Iterations (NT) | CPU Computational Time | GPU Computational Time |
|---|---|---|
| 800 | 0.351 | 0.31 |
| 1600 | 0.662 | 0.318 |
| 3200 | 1.324 | 0.361 |
| 6400 | 2.725 | 0.413 |
| 10000 | 4.238 | 0.545 |
| 12800 | 5.351 | 0.586 |
| 20000 | 8.327 | 0.744 |
| 25600 | 10.539 | 0.92 |
| 50000 | 20.76 | 1.467 |

*Table 3: CPU code vs. GPU code execution time.*

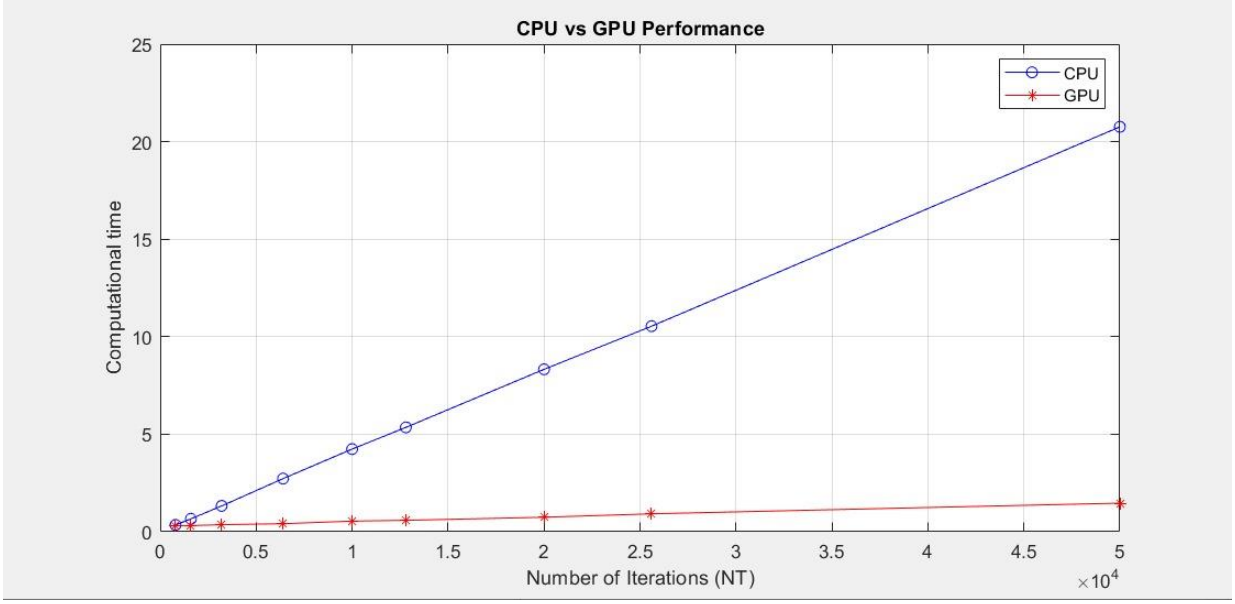


*Figure 19: CPU code vs. GPU code execution time.*

## Speed Up Achieved

| Number of Iterations (NT) | Speed up |
|---|---|
| 800 | 0.041 |
| 1600 | 0.344 |
| 3200 | 0.963 |
| 6400 | 2.312 |
| 10000 | 3.693 |
| 12800 | 4.765 |
| 20000 | 7.583 |
| 25600 | 9.619 |
| 50000 | 19.293 |
| 75000 | 30.6 |
| 100000 | 38.811 |
| 150000 | 57.884 |
| 200000 | 77.45 |
| 250000 | 96.896 |
| 300000 | 116.472 |
| 350000 | 136.125 |
| 400000 | 155.796 |
| 450000 | 179.816 |
| 500000 | 194.772 |
| 550000 | 212.656 |
| 600000 | 233.79 |
| 650000 | 252.911 |
| 700000 | 278.373 |
| 750000 | 290.428 |
| 800000 | 311.159 |
| 850000 | 330.747 |
| 900000 | 350.504 |
| 950000 | 369.796 |
| 1000000 | 388.724 |

*Table 4: Speed up achieved through porting to GPU.*

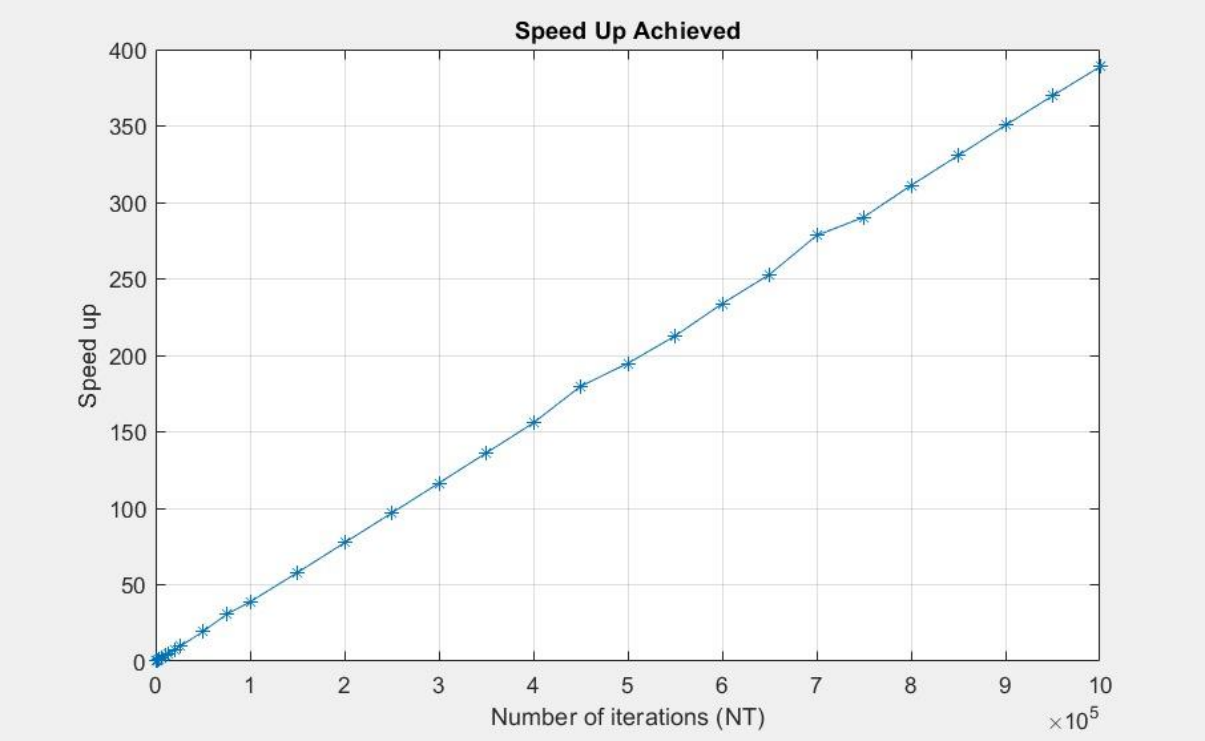


*Figure 20: Speed Up Achieved.*

It can be observed above, in Table 4 and Figure 20, that porting the application to CUDA achieves a steady speed up even for a very high number of iterations, this is due to the fact that work is being divided amongst threads across multiple blocks therefore allowing up to **67 million**[1] simultaneous operations. So even when the program was required to execute one million operations in parallel, the GPU managed to complete it within 25 seconds while the CPU took approximately 7 minutes.

## Conclusion

In conclusion, parallelising a program properly can achieve a huge speed up, especially when it comes to large datasets which include thousands or millions of processes. The best way to optimize code using the GPU's parallel resources is by utilising both the threads and blocks in a kernel. As each block has 1024 threads, and there are 65535 blocks per dimension, enabling the use of multiple blocks would make parallelising programs with datasets that require more than 1024 processes more efficient as multiple blocks can run at max capacity simultaneously executing up to 67 million processes in parallel per dimension.

The speed up achieved is dependant on the nature of the operations performed by the application, making some applications more parallelizable than others. The 2D TLM was an example of an application whose operations are highly data parallel, hence the speed up achieved. A different program may consist mainly of sequential operations, and so attempting to port such a program to CUDA may not speed up execution at all, and is more likely to run even slower than it did on the CPU.

[1] 1024 x 65535 = 67107840 ≈ 67 million