CUDA Coursework 2 – Porting the 2D TLM Algorithm to CUDA

Damir Kadyrzhan 20494312

1. Introduction

The widely used technique in the simulation of electromagnetic wave propagation in complex structures is a two-dimensional transmission line matrix. The computationally intensive nature of this algorithm can make it challenging to run on sequential processing in the CPU. To overcome this limitation, it is an excellent option to port this algorithm into GPU. In this report, the GPU port of the 2D TLM algorithm and a discussion of benefits and boosts in performance will be explored. Also, the performance evaluation of GPU port 2D TLM code is compared with the original CPU version.

2. CPU Code

2.1 CPU Code Description

Provided CPU code implements a two-dimensional Transmission Line Matrix (2D TLM) method, which is a computational technique used to model and simulate the behaviour of electromagnetic (EM) waves in transmission lines and other waveguides.

The 2D TLM method divides the waveguide into a 2D mesh of nodes, where each node is connected to its neighbours through transmission lines. The fields are represented by voltage (V) and current (I) variables at each node, which is updated at each time step (dt) using a set of difference equations.

The main algorithm consists of source, scatter, connect and apply boundaries.

- Source: A time varying voltage source is injected at a specific node (Ein) in the mesh using a Gaussian pulse.
- Scatter: TLM difference equations are used to update voltage and current variables at each node.
- Connect: The voltage and current variables at each node are communicated to its neighbouring nodes. Operation happens by swapping the variables and updating them.
- Apply Boundaries: Special boundary conditions are applied at the edges of the mesh to handle reflections and absorb incoming waves.

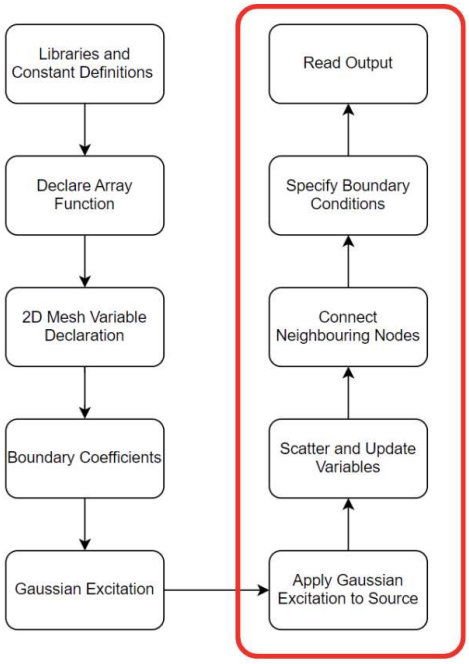

Figure 1. CPU 2D TLM Algorithm Flow Chart

As can be seen from Figure 1, the parts that are parallelisable are source, scatter, connect and apply boundaries. Scatter and connect are the main algorithms that can be parallelised since the operation

happens at each node and it can happen at the same time. At each node gaussian excitation must be applied to the source and boundary conditions must be checked.

2.2 CPU Code Benchmark

Table 1. CPU Benchmark Timing Results

Time Steps	Execution Time
10	0.006
50	0.022
100	0.042
500	0.224
1000	0.44
2000	0.87
5000	2.178
10000	4.328
20000	8.714
30000	13.027
40000	17.773
50000	21.411
100000	43.848

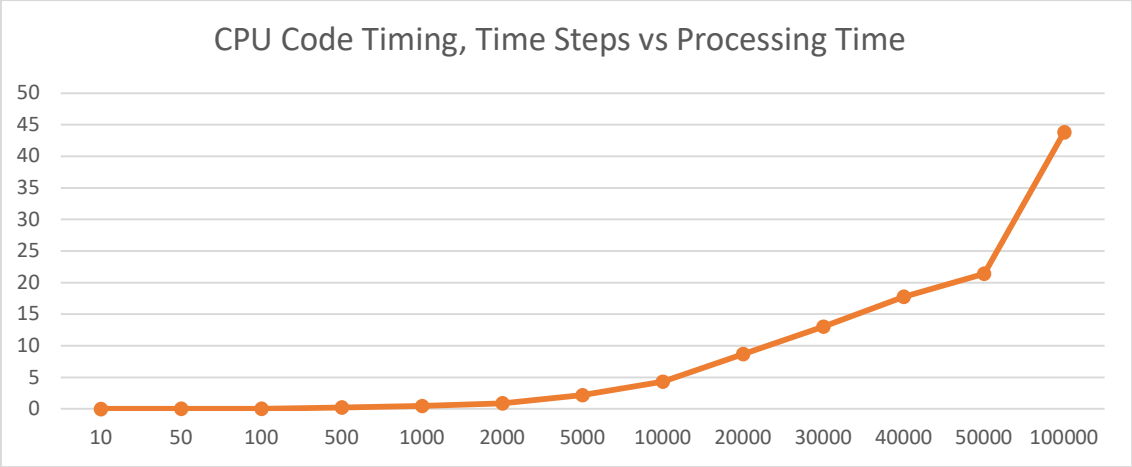

Figure 2. CPU Code Performance Timing

As can be seen from Figure 2, CPU code is fast in processing small amounts of operations due to its powerful single cores. However, sequential operations become irrelevant due to computationally intensive simulation when it comes to large amounts of operations. It leads to long simulation times and may limit the resolution and accuracy of the results. To overcome this limitation, 2D TLM can be implemented in GPU, which offers a higher level of parallelism and can significantly improve the performance of 2D TLM simulations.

3. GPU Code

3.1 GPU Code Description

The proposed 2D TLM algorithm implemented in CUDA utilises the large number of cores of GPU by optimising the ‘Scatter’ and ‘Connect’ functions, which are the main computationally intensive parts.

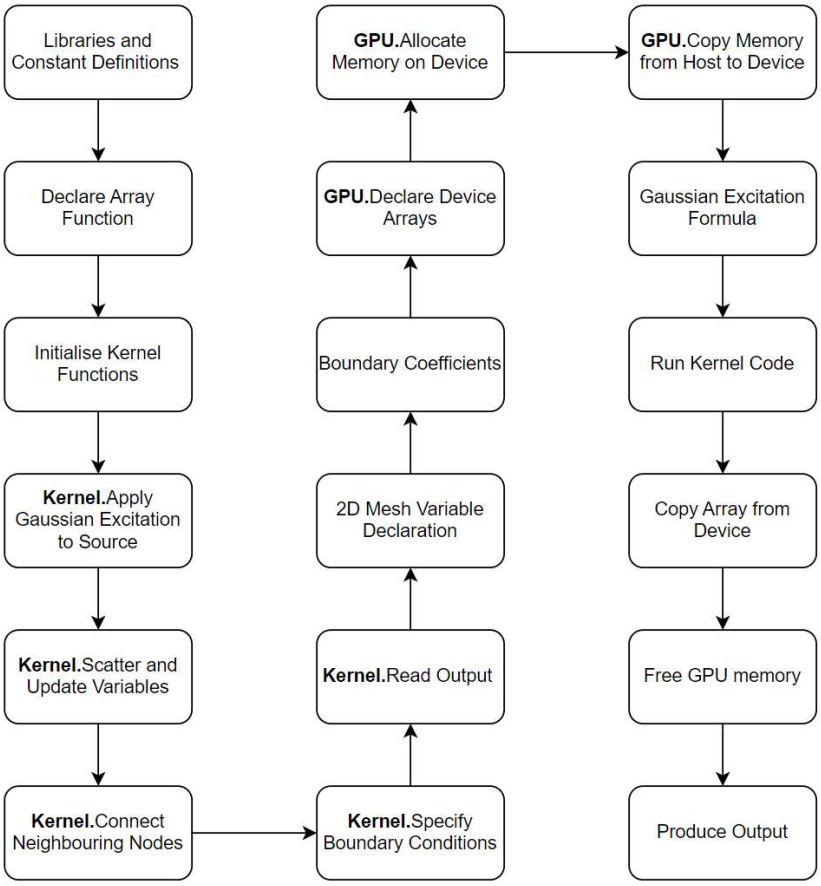

Figure 3. GPU CUDA Code Algorithm Flow Chart

Operations that have been optimised using CUDA consists of the parts of the kernel of source, scatter, connect and apply boundaries. After, the kernel collects and sends the data to the host from the device. Highlighted as 'Kernel' in Figure 3 shows the parallelisable functions executed in GPU's kernel. Then, they are executed after declaring the variables in the main function.

```
__global__ void tlmSource(int* Ein, double* V1, double* V2, double* V3, double* V4, double E0, int N)
{
    auto index = Ein[0] + Ein[1] * N;
    V1[index] = V1[index] + E0;
    V2[index] = V2[index] - E0;
    V3[index] = V3[index] - E0;
    V4[index] = V4[index] + E0;
}
```

Code 1. Source

The first kernel function multiplies and adds the Gaussian Excitation formula (E0) to the pointers to arrays of doubles of 'V1', 'V2', 'V3' and 'V4'. The location of applying this formula is {10, 10} determined by 'Ein [1]' and 'Ein [0]'.

```
__global__ void tlmScatter(int NX, int NY, double* V1, double* V2, double* V3, double* V4, double Z)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < NX && y < NY) {
        double I = (2 * V1[x * NY + y] + 2 * V4[x * NY + y] - 2 * V2[x * NY + y] - 2 * V3[x * NY + y]) / (4 *
Z);
        double V = 2 * V1[x * NY + y] - I * Z;        //port1
        V1[x * NY + y] = V - V1[x * NY + y];
        V = 2 * V2[x * NY + y] + I * Z;        //port2
        V2[x * NY + y] = V - V2[x * NY + y];
        V = 2 * V3[x * NY + y] + I * Z;        //port3
        V3[x * NY + y] = V - V3[x * NY + y];
        V = 2 * V4[x * NY + y] - I * Z;        //port4
        V4[x * NY + y] = V - V4[x * NY + y];
    }
}
```

Code 2. Scatter

The second kernel function ‘Scatter’ does the same operation as the one implemented in CPU, but the positions of ‘x’ and ‘y’ determined by ‘blockIdx’ and ‘threadIdx’ variables. If the ‘x’ and ‘y’ are within the bounds of the arrays, the kernel function updates the values of the voltages at the nodes. The update of the values depends on the current values.

```
__global__ void tlmConnect(int NX, int NY, double* V1, double* V2, double* V3, double* V4, double
rXmin, double rXmax, double rYmin, double rYmax) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    // Connect
    if (x + 1 < NX && y < NY) {
        double tempV = V2[x * NY + y];
        V2[x * NY + y] = V4[(x - 1) * NY + y];
        V4[(x - 1) * NY + y] = tempV;

    }
    if (x < NX && y + 1 < NY) {
        double tempV = V1[x * NY + y];
```

```
        V1[x * NY + y] = V3[(x - 1) * NY + y];
        V3[(x - 1) * NY + y] = tempV;
    }

    // Apply Boundaries
    if (x < NX && y == NY - 1) {
        V3[x * NY + y] = rYmax * V3[x * NY + y];
    }
    if (x < NX && y == 0) {
        V1[x * NY + y] = rYmin * V1[x * NY + y];
    }
    if (x == NX - 1 && y < NY) {
        V4[x * NY + y] = rXmax * V4[x * NY + y];
    }
    if (x == 0 && y < NY) {
        V2[x * NY + y] = rXmin * V2[x * NY + y];
    }
}
```

Code 3. Connect and Apply Boundaries

The third optimised function is 'Connect' and 'Apply Boundaries'. The 'Connect' operation in the kernel function swaps the values of certain elements in the 'V2' to 'V4' and 'V1' to 'V3' every iteration. The elements that are swapped are determined by the coordinated kernel 'x' and 'y'. The 'Apply Boundaries' operation involves modifying elements in the 'V1', 'V2', 'V3', and 'V4' arrays based on the values of constants 'rXmin', 'rXmax', 'rYmin', 'rYmax'. The purpose of this condition is to enforce boundary conditions on the elements of the arrays.

```
__global__ void tlmApplyProbe(int* Eout, double* out, double* V2, double* V4, int n, int N) {
    auto index = Eout[0] + Eout[1] * N;
    out[n] = V2[index] + V4[index];
}
```

Code 4. Apply Probe

The output is read at the point of {15, 15} of 'V2' and 'V4'.

3.2 GPU Code Benchmark

Table 2. GPU Benchmark Timing Results

Time Steps	Time
10	0.234
50	0.274
100	0.282

500	0.376
1000	0.456
2000	0.734
5000	0.922
10000	1.852
20000	3.583
30000	4.65
40000	5.897
50000	7.459
100000	14.279

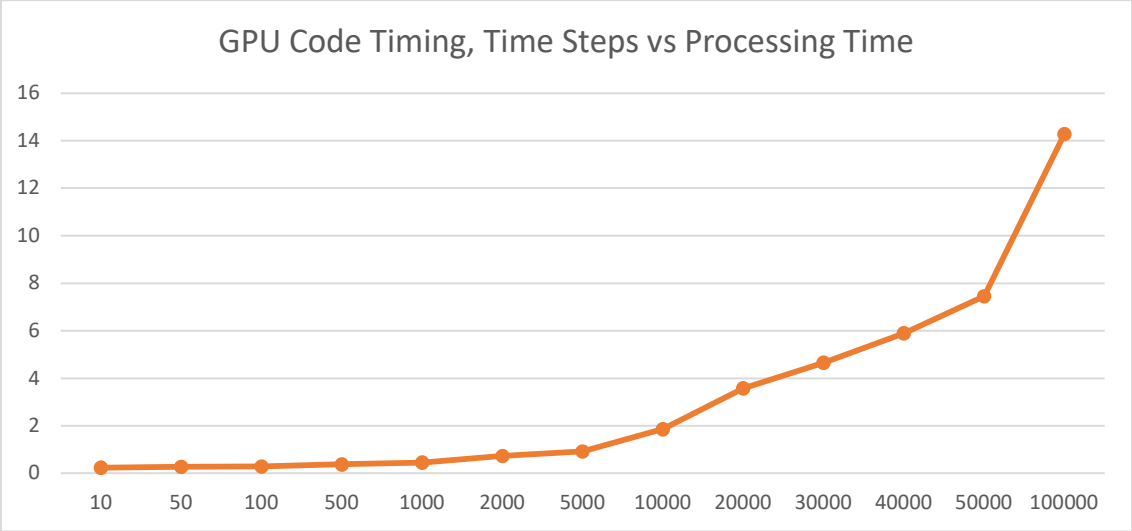

Figure 4. GPU Code Performance Timing

Table 3. GPU vs CPU Benchmark Timing Results

Time Steps	GPU Time	CPU Time	Speedup
10	0.234	0.006	0
50	0.274	0.022	0
100	0.282	0.042	0
500	0.376	0.224	0
1000	0.456	0.44	0
2000	0.734	0.87	1.24
5000	0.922	2.178	2.36
10000	1.852	4.328	2.33
20000	3.583	8.714	2.43
30000	4.65	13.027	2.8
40000	5.897	17.773	3.01
50000	7.459	21.411	2.87
100000	14.279	43.848	3.07

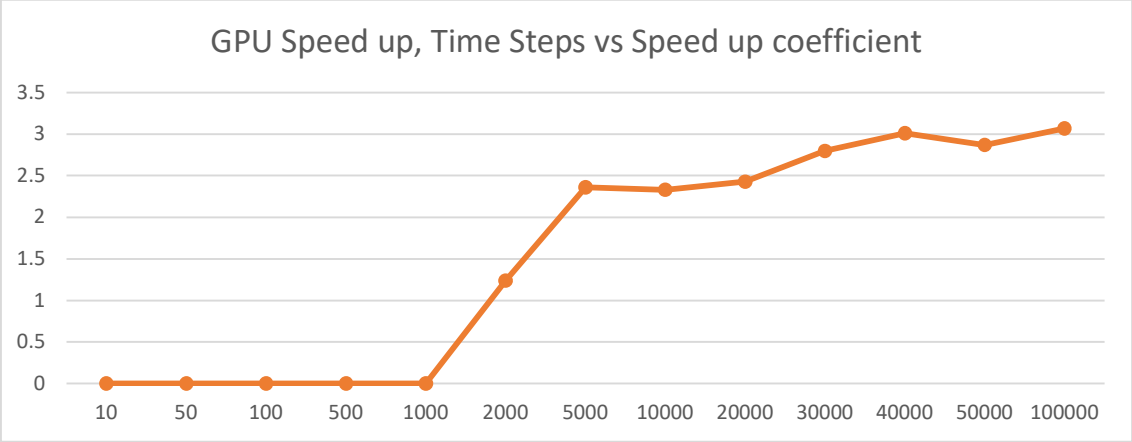

Figure 5. Speed Up Achieved with GPU against CPU

Implementation of the 2D TLM algorithm into CPU has several disadvantages:

- A limited number of cores means that they can only perform a limited number of calculations simultaneously.
- Limited memory bandwidth of CPUs can limit the speed at which data can be accessed and processed.
- Limited cache size can limit the data stored and accessed quickly.

On the other hand, GPU (Graphics Processing Unit) is a specialised chip designed for rapid parallel processing of graphics and other data. As can be seen from Figure 4,5 tested algorithm with 16 blocks and 16 threads has shown a significant boost in performance (up to 3x).

- GPUs have more cores, allowing them to perform a larger number of calculations simultaneously.
- GPUs, unlike CPUs, have higher memory bandwidth, which allows them to access and process data faster.

Performed 2D TLM algorithm in GPU have shown a rapid boost in performance with parallel processing.

4. Conclusion

This project aimed to investigate the concepts of porting CPU code of 2D TLM algorithm to GPU and investigate the difference between CPU algorithm and GPU's parallelism property. A comprehensive understanding of proposed algorithms has been presented alongside GPU port algorithm implementation using CUDA runtime environment. In accordance with the project's aims, the developed GPU 2D TLM algorithm can provide sufficient speed up with a comparison of the CPU counterpart. GPU 2D TLM algorithm has utilised parallelism of GPU by showing a significant speed up in operational speed. Although the resulting output fluctuates a little due to its parallel nature, the algorithm has been shown to work sufficiently to retrieve some information.

Various improvements can be made to the project discussed in the report. Many other CUDA coding techniques have not been investigated, and thus, further research can assist in finding more concrete evidence for producing clear output. Future work would entail extending the research of different CUDA coding techniques and implementing them in various situations. Further work would involve a more precise CUDA algorithm, which utilises modern algorithms. With sufficient time, further corrections to the existing CUDA algorithm can be made. This would require a more profound knowledge of CUDA coding and computational engineering.