GPU Computing Using C/C++ CUDA: Scientific Analysis of the 2-D Convection Equation

Alovya Chowdhury

17th September, 2019

Contents

1 Introduction 3
2 General-purpose GPU computing 3
3 Serialised code 3
3.1 1-D convection equation: theory 3
3.2 1-D convection equation: Python 4
4 Parallelising using C/C++ CUDA 6
4.1 1-D convection equation 6
4.1.1 Time reduction analysis of 1-D version 9
4.2 2-D convection equation 10
4.2.1 Time reduction analysis of 2-D version 15
5 Concluding remarks and areas of improvement 18

1 Introduction

The purpose of this document is to demonstrate the author's technical capacity and potential to parallelise the code for a flood simulator being developed under the SEAMLESS-WAVE project. To achieve this, coding projects written by the author in Python, C++ and C++ CUDA will be showcased. The structure of the document will be as follows: a very brief outline of GPU computing, its history and its link with scientific applications. Afterwards, code solving the 1-D convection equation in Python will be displayed; this will be accompanied with relevant theory. The aforementioned code, written in series, will be followed by a section showing instead a parallel implementation in C/C++ CUDA solving the 1-D convection equation. Finally, a parallelised solver code in C/C++ CUDA for the 2-D convection equation will also be presented.

2 General-purpose GPU computing

GPU computing harnesses the power of graphical processing units, which act as "co-processors" to accelerate CPU performance for general-purpose scientific and engineering computing. This is achieved by off-loading a portion of the compute-intensive and time consuming sections of the code from the CPU to the GPU; from the user's point of view, the application completes more quickly because it taps into the GPU's massively parallel architecture. Such a combination of CPU and GPU is known as "heterogeneous" or "hybrid" computing.

General-purpose GPU computing began with graphics chips that contained fixed functions. These chips gradually became increasingly programmable and the scientific community started to take advantage of them for scientific applications. However, early GPUs required the use of graphics programming languages, such as OpenGL or Cg, and developers were forced to make their scientific applications look like graphics applications to access the parallel processing power of GPUs.

NVIDIA eventually realised the huge potential of bringing this power to the scientific community at large and developed the CUDA parallel computing platform. This platform is fully interfaceable with high level languages such as C, C++ and Fortran. As a result, GPUs can now readily be exploited for scientific applications, with huge swathes of calculations, computations and numerical operations, which are by nature parallelisable, being delegated to GPUs instead of being held back by (mostly) serial CPUs.

3 Serialised code

3.1 1-D convection equation: theory

The following two subsections will show serialised code that numerically solves the 1-D convection equation using the finite difference method, in Python. The 1-D convection equation is as follows:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0$$

This equation can be discretised; the subscript "i" represents the x-direction and the subscript "n" refers to the time step. Discretising each term one obtains:

$$\frac{\partial u}{\partial t} = \frac{u_i^{n+1} - u_i^n}{\Delta t}$$

$$\frac{\partial u}{\partial x} = \frac{u_i^n - u_{i-1}^n}{\Delta x}$$

The time derivative has been discretised with a forward scheme and the spatial derivative in x has been discretised with a backward scheme. The delta values refer to the numerical spacing between each grid point. The final discretised equation is:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + c\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0$$

Which can be rearranged to solve for u across the spatial domain and "march forward" in time:

$$u_i^{n+1} = u_i^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$$

In the 1-D case, we therefore have a vector of u values that represent the function u at a given time step. Then it is possible to perform numerical calculations according to the above equation and solve it to the desired time duration. One final term that must be considered is the Courant number:

$$Co_i \equiv \frac{u_i \Delta t}{\Delta x} < C_{max}$$

Where Co_i is the Courant number at a given stencil point and u_i is the corresponding velocity (in this case consider u to be the velocity). Formally, this is known as the Courant–Friedrichs–Lewy condition and for many applications C_{max} has a value of one. This discussion will not go into the derivation of this condition, but heuristically it will be stated the meaning of the condition is that the numerical domain must be fine enough to include a sufficient amount of information from analytical domain. When values of Δx and Δt are chosen in the code, it must be remembered that the CFL condition has to be met to promote stability of the solution as it marches forward in time.

3.2 1-D convection equation: Python

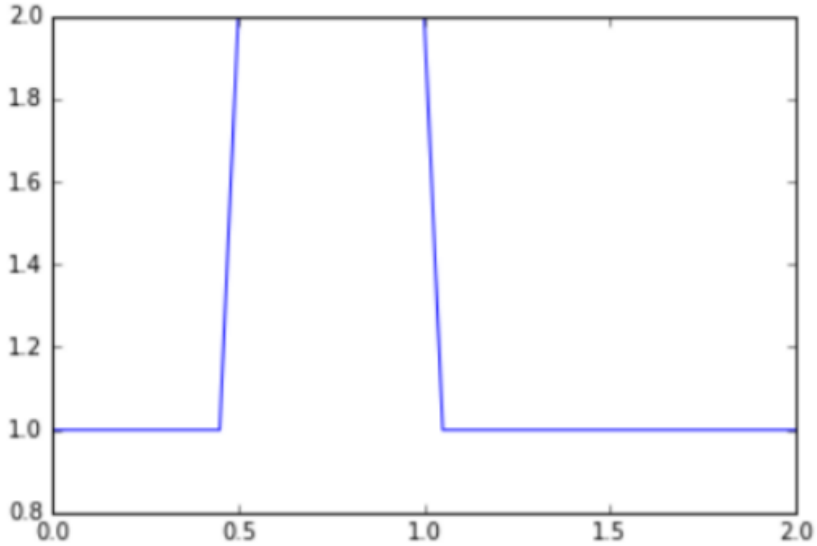
This section will walk the reader through Python code that solves the convection in equation in one dimension. First, necessary parameters are declared:

```
nx = 41            # number of pencil points in spatial domain
dx = 2 / (nx-1)   # numerical spacing delta x

nt = 25           # number of time steps
dt = .025        # seconds per time step delta t

c = 1           # take wavespeed to be = 1
```

Next, the vector u is initialised and a step function is used as the initial condition:

```
u = numpy.ones(nx)
u[int(.5 / dx):int(1 / dx + 1)] = 2  # u = 2 between x = 0.5 and x = 1
pyplot.plot(numpy.linspace(0, 2, nx), u);
```

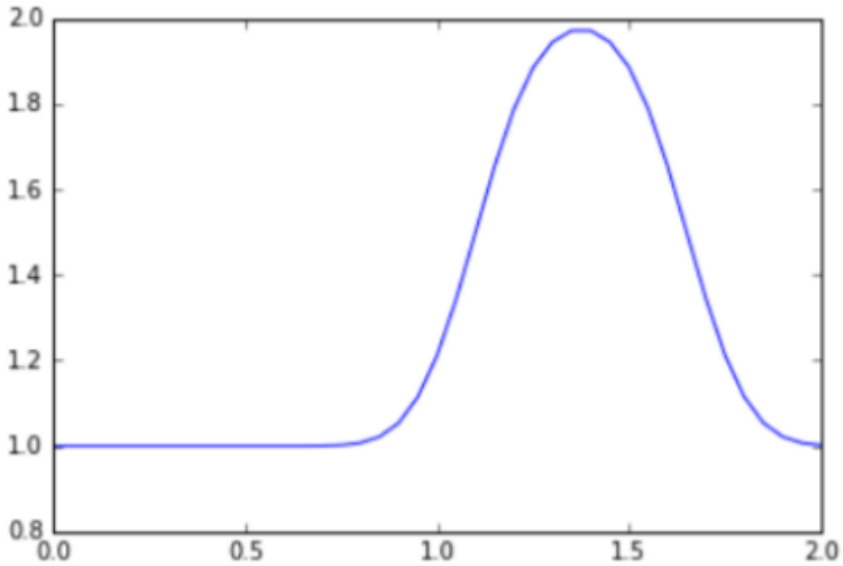
The output, used for visualisation purposes, looks like:



The finite difference operation is carried out in the below for-loop:

```
un = numpy.ones(nx) # initialise temporary array

for n in range(nt):  # loop over time range
    un = u.copy()
    for i in range(1, nx):
        u[i] = un[i] - c * dt / dx * (un[i] - un[i-1])
```

This code is neither Pythonic nor very efficient; instead of carrying out matrix subtraction, the code loops through every individual element in the vector, accessing it twice. In plotting the u vector after this loop is completed the following is obtained:



The step function has convected to the right in the x-direction as expected. However, some numerical diffusion is also observed: this is not a behaviour of the exact solution but rather an artefact of the differencing schemes, which in this case is only first-order accurate.

4 Parallelising using C/C++ CUDA

The below subsections deal with a parallelised implementation of the finite difference method, initially in 1-D then in 2-D.

4.1 1-D convection equation

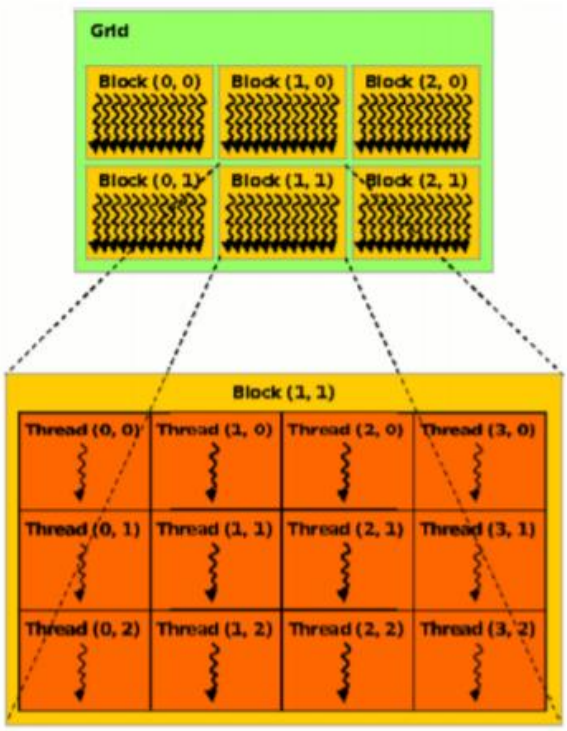
First, the code will simply be showcased to highlight how it implements the finite difference method, with the details of parallelisation coming later. Two important functions are defined:

```
__global__ void finiteDiff(const int c, const double dt, const double dx, const int
nt, const int nx, double *u, double *un) {

	int index = blockIdx.x * blockDim.x + threadIdx.x;
	int stride = blockDim.x * gridDim.x;

	for (int t = 0; t < nt; t++) {

		for (int i = index; i < nx; i += stride) {
			un[i] = u[i];
		}

		for (int i = index + 1; i < nx; i += stride) {
			u[i] = un[i] - c * dt / dx * (un[i] - un[i - 1]);
		}
	}
}

__global__ void stepFunction(const double dx, const int nx, double *u) {

	int index = blockIdx.x * blockDim.x + threadIdx.x;
	int stride = blockDim.x * gridDim.x;

	for (int i = index; i < nx; i += stride) {

		if (i * dx >= 0.5 && i * dx <= 1) {
			u[i] = 2;
		}
		else {
			u[i] = 1;
		}
	}
}
```

The first is stepFunction, which is simply a function that initialises the step function on the u vector. The finiteDiff function looks very similar to the Python code in the previous section: it is a nested for-loop for the time steps with two for-loops, one to create deep copies to carry out the subtraction and another one actually carrying out this subtraction.

Bearing this in mind, the discussion as to how GPU computing is implemented in this code can begin. First of all one should notice the "__global__" term in front of the function declaration and definition. In C CUDA, what this means is that the function has been defined to be a *CUDA kernel*. In other words, instead of running on the CPU, the program now knows that this *kernel* should instead be *launched* on the GPU processors.

Next, one needs explanations of the terms *gridDim, blockIdx, blockDim* and *threadIdx*. The concept of grids, blocks and threads is best illustrated with a picture:



Essentially, the grid can be divided into 2 dimensions and blocks into 3 dimensions; threads make up a block. In this case however our problem maps very neatly to a 1 dimensional block, hence in the code we only ever access the x index. In the 2-D problem below it will be shown one can take advantage of dividing the grid into 2 dimensions by using a *dim3* object. Loosely speaking, the grid maps to GPUs, the blocks map to the multiprocessors and the threads map to stream processors.

The expression *blockDim.x * blockIdx.x + threadIdx.x* is idiomatic CUDA, and allows us to access each individual thread in grid of blocks. The stride length *gridDim.x * blockDim.x* allows us to loop over the length of the grid; these two terms combine to form what is known as a "*grid-stride*" loop. There are many benefits to using grid-stride loops. In using one, it is ensured that the indexing within warps is unit-stride and maximum memory coalescing is achieved. The expressions are only evaluated when the loop condition is true, therefore there is in the worst case the same instruction cost as when using an if-statement; such conditions are in place to ensure that the thread index does not attempt to access an out-of-range value from an array.

There will be further discussion about the benefits of using grid-stride loops at the end of this section, at the moment we are ready to discuss the int main() portion of our code. First the necessary variables are initialised and memory is allocated:

```
const int nx = 2000;
double dx = double(2) / (nx - 1);

const int nt = 250;
double dt = 0.001;

int c = 1;

double *u, *un;
cudaMallocManaged(&u, nx * sizeof(double));
cudaMallocManaged(&un, nx * sizeof(double));
```

Notice that the number of stencil points nx is much greater which leads to a smaller spatial spacing dx. To ensure that the CFL condition is met, a much smaller time dt of 0.001 is used. Two pointers to the u and u copy vectors are initialised, similar to the Python code, however unlike Python there is a question of memory management. Therefore, the cudaMallocManaged() function is called; from the CUDA user guide it states that this function is useful when developers want to prototype a new kernel or application quickly. There are other, more efficient ways to allocate memory which is included in the 2-D code.

cudaMallocManaged() allocates a buffer for variables in *Unified Memory*, a memory which is accessible by both the host and the device (the CPU and GPU). Since there will be an array of nx doubles, memory of this amount is allocated using the argument nx * sizeof(double). Continuing through the code:

```
int threads = 256;
int blocks = (nx + threads - 1) / (threads);

// Step function initial condition
stepFunction<<<blocks, threads>>>(dx, nx, u);

cudaDeviceSynchronize();

// Visualise step function
ofstream myfile("data.txt");

for (int i = 0; i < nx; i++) {
	myfile << u[i] << " ";
}

myfile << endl << endl << endl;

// Finite difference
finiteDiff<<<blocks, threads>>>(c, dt, dx, nt, nx, u, un);

cudaDeviceSynchronize();

// Visualise final solution
for (int i = 0; i < nx; i++) {
	myfile << u[i] << " ";
}

cout << "Solved";
```

Usually the number of threads per block on a given GPU architecture is a multiple of 32; the author found after tedious trial-and-error, debugging and Googling that their NVIDIA GTX 1050 mobile GPU is limited to a maximum of 256 threads per block. To ensure there are enough threads and in cases where nx is odd, the expression for "*blocks*" is crafted such that number of blocks * threads per block equals the number of elements in the array. When launching the stepFunction kernel, in the brackets <<<x, y>>> x specifies the number of blocks and y the number of threads per block.

After having launched the stepFunction kernel, it is necessary to call cudaDeviceSynchronize() to ensure the CPU waits until the GPU has finished running before proceeding with the rest of the program. Data is written to file for visualisation purposes, and then finiteDiff, also in parallel, is launched, calling cudaDeviceSynchronize() to sync and writing data to file. Finally the buffer in unified memory is freed as follows to ensure there are no memory leaks:

```
cudaFree(u);
cudaFree(un);
```

Now it is possible to go into a bit more detail regarding the benefits of using grid-stride loops. This can be broken down into primarily three areas:

- **Scalability**: by using a loop, problem sizes that are larger than maximum grid size of a given CUDA device can be supported;
- **Readability**: the code looks very similar to the sequential loop implementation, making it easier for other users to understand;
- **Debugging**: it is very simple to switch to serial processing (use <<<1, 1>>> during kernel launch) and emulate host implementation in order to compare results from parallelisation.

4.1.1 Time reduction analysis of 1-D version

In fact, we will take advantage of this very last point to compare sequential processing with parallel processing. We will time the kernels using the command line GPU profiler *nvprof*. The results of launching the kernel with one block and one thread (i.e. sequentially) is as follows:

```
==3764== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.86%  217.54ms         1  217.54ms  217.54ms  217.54ms  finiteDiff(int, double, double, int, int, double*, double*)
                    0.14%  300.48us         1  300.48us  300.48us  300.48us  stepFunction(double, int, double*)
```

Serialised implementation

Compare this with launching the kernel using 256 threads and the appropriate number of blocks (<<<blocks, threads>>>) the following is obtained:

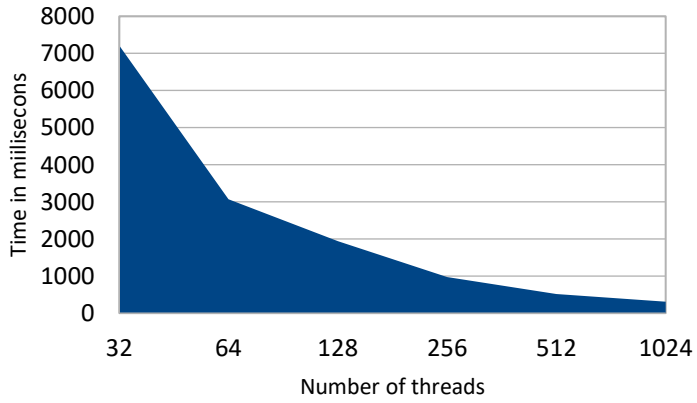
```
==10748== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.65%  149.60us         1  149.60us  149.60us  149.60us  finiteDiff(int, double, double, int, int, double*, double*)
                    1.35%  2.0480us         1  2.0480us  2.0480us  2.0480us  stepFunction(double, int, double*)
```

Parallelised implementation

Attention should be drawn to the highlighted sections: in the serialised implementation, the finiteDiff() function takes 217.54 milliseconds (number emphasised in red) to run. Contrast this with the 149.60 microsecond execution time in parallelised implementation.

$$\frac{Serialised\ run\ time}{Parallelised\ run\ time} = \frac{217{,}540\ \mu s}{149.60\ \mu s} \approx 1500 \times speed\ up$$

In other words, parallel processing has given us a huge advantage by reducing the time the *kernel* takes to run by a factor of 1500!



Execution time vs number of threads

In six test cases, a similar set of simulations have been carried out, but only one block has been used and the number of threads are varied in multiples of 32 to observe the effect of thread number on the scalability i.e. execution time of the kernel. The below results were obtained:

From the graph, it is evident that the kernel execution time is inversely proportional to the number of threads i.e. the greater the number of threads the quicker the execution time. Furthermore, it ought to be mentioned that the maximum number of threads per block is 1024, so in this instance where only one block has been used, the limit is also 1024 threads. Had more blocks been used, then it would have been possible to increase the number of threads e.g. if 2 blocks had been used, up to 2048 threads could have been implemented and hence, the kernel execution time could have been reduced even further.

One final note is that so far, this sub-discussion has been about the influence of the numbers of threads on GPU kernel execution time. In reality, the speed of the simulation framework does not depend only on the duration of operations being carried out in the GPU. There are many other processes, both on the GPU and CPU, that may hinder or be a bottleneck with it comes to increasing the simulation speed (and therefore reducing time and costs) of a programme. Take for instance the fully detailed timing profile of 1D_convection_CUDA provided by *nvprof:*

```
==14292== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.59%  149.70us         1  149.70us  149.70us  149.70us  finiteDiff(int, double, double, int, int, double*, double*)
                    1.41%  2.1440us         1  2.1440us  2.1440us  2.1440us  stepFunction(double, int, double*)
      API calls:   76.51%  322.44ms         2  161.22ms  108.10us  322.33ms  cudaMallocManaged
                   22.61%  95.291ms         1  95.291ms  95.291ms  95.291ms  cuDevicePrimaryCtxRelease
                    0.22%  909.40us         2  454.70us  197.20us  712.20us  cudaFree
                    0.19%  813.20us         2  406.60us  288.10us  525.10us  cudaDeviceSynchronize
                    0.19%  782.80us         1  782.80us  782.80us  782.80us  cuModuleUnload
                    0.18%  742.50us        97  7.6540us     300ns  341.10us  cuDeviceGetAttribute
                    0.09%  379.50us         2  189.75us  119.10us  260.40us  cudaLaunchKernel
                    0.01%  50.300us         1  50.300us  50.300us  50.300us  cuDeviceTotalMem
                    0.00%  20.500us         1  20.500us  20.500us  20.500us  cuDeviceGetPCIBusId
                    0.00%  3.6000us         3  1.2000us     600ns  2.1000us  cuDeviceGetCount
                    0.00%  3.6000us         2  1.8000us     400ns  3.2000us  cuDeviceGet
                    0.00%  1.7000us         1  1.7000us  1.7000us  1.7000us  cuDeviceGetName
                    0.00%     800ns         1     800ns     800ns     800ns  cuDeviceGetUuid
                    0.00%     700ns         1     700ns     700ns     700ns  cuDeviceGetLuid
```

Consider for example the highlighted API call cudaMallocManaged, which consumes nearly 77% of the profile time on the CPU. If the simulation tool is to be made as efficient and accurate as possible, then while speeding up the process by parallelising certain sections of the programme by developing GPU CUDA kernels is effective, attention must also be paid to other sections of the code that are called on the CPU.

4.2 2-D convection equation

The 2-D convection equation is as follows:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} + c\frac{\partial u}{\partial y} = 0$$

However, this time the equation can be discretised spatially in the y-direction as well, giving rise to the subscript "j". The discretised equation looks like:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} + c\frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\Delta x} + c\frac{u_{i,j}^{n} - u_{i,j-1}^{n}}{\Delta y} = 0$$

And therefore we can solve for u marching forward in time by:

$$u_{i,j}^{n+1} = u_{i,j}^{n} - c\frac{\Delta t}{\Delta x}(u_{i,j}^{n} - u_{i-1,j}^{n}) - c\frac{\Delta t}{\Delta y}(u_{i,j}^{n} - u_{i,j-1}^{n})$$

Evidently, this 2-D system requires 2-D discretisation; therefore there is a matrix whose rows are the y coordinates and whose columns are the x coordinates. For ease of implementation, a square domain is used i.e. a square matrix and the numerical spacing of the both the x- and y-direction are the same. This means the above equation looks like:

$$u_{i,j}^{n+1} = u_{i,j}^{n} - c\frac{\Delta t}{\Delta d}(u_{i,j}^{n} - u_{i-1,j}^{n}) - c\frac{\Delta t}{\Delta d}(u_{i,j}^{n} - u_{i,j-1}^{n})$$

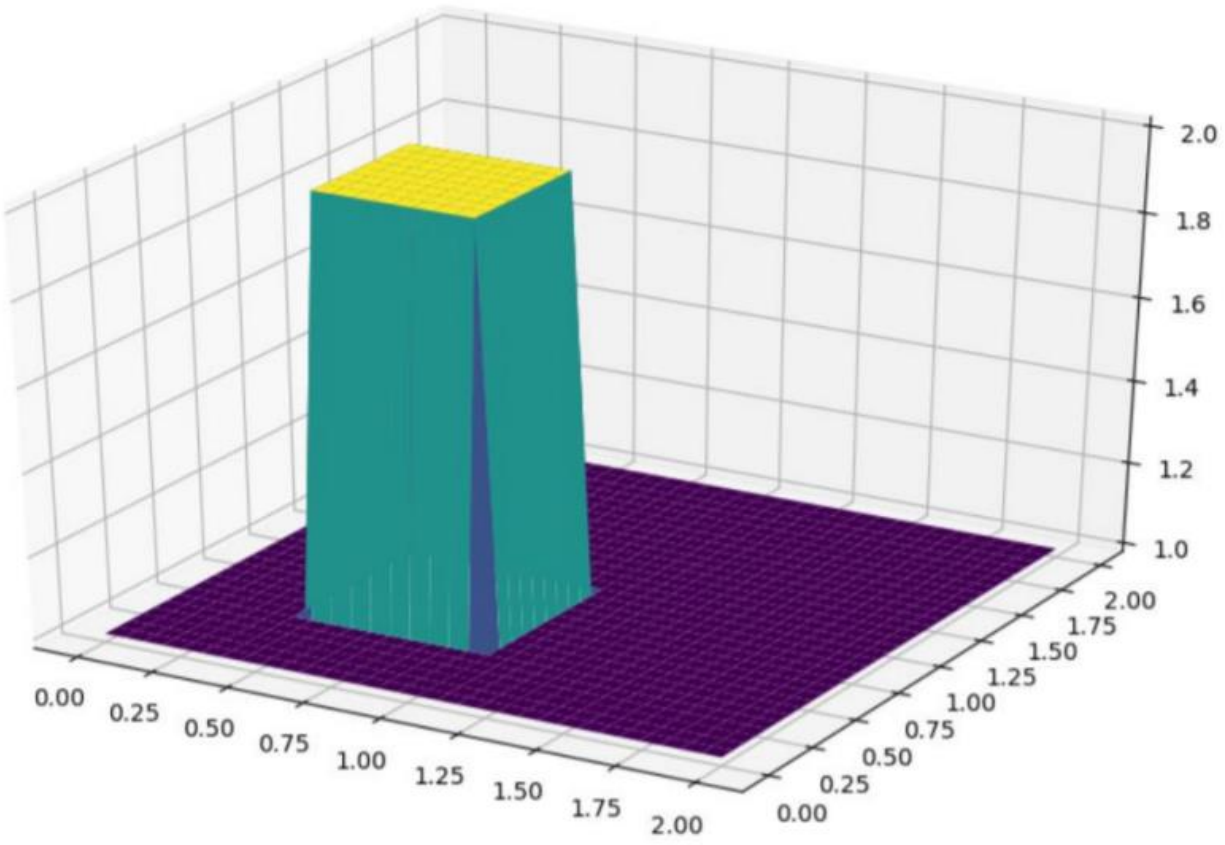
Now we can begin to step through the code. In this case, a *monolithic* kernel will be used, which means the whole array is passed only once and there is only one element per thread. Since it is known from previous coding attempts that there can only be 256 threads per block, and only one block is being used (in this case), then for a square matrix a side length of 32 elements is defined, giving 32*32 = 256 elements in total:

```
// MAX THREADS PER BLOCK FOR GTX 1050 MOBILE GPU CARD IS 256
#define N 32
```

For the sake of brevity, in contrast to the previous section only selected portions of the code will be showcased here in order to demonstrate necessary concepts. As in the 1-D code, there are kernels for initialising a step, but this time in 2 dimensions:

```
__global__ void stepFunction2D(double d_A[N][N], double d) {
	int ix = blockIdx.x * blockDim.x + threadIdx.x;
	int iy = blockIdx.y * blockDim.y + threadIdx.y;

	if (ix * d >= 0.5 && ix * d <= 1 && iy * d >= 0.5 && iy * d <= 1) {
		d_A[ix][iy] = 2;
	}
	else {
		d_A[ix][iy] = 1;
	}
}
```

The output of which looks like*:



**Note: visualisation was done in Python as the libraries are integrated into the environment, unlike in C++*

Similarly, there is a kernel implementing the finite difference method and carrying out matrix addition/subtraction:

```
__global__ void finiteDiff2D(double d_A[N][N], double d_B[N][N], const double d, const
double dt, const double c) {
	int ix = blockIdx.x * blockDim.x + threadIdx.x;
	int iy = blockIdx.y * blockDim.y + threadIdx.y;

	if (ix < N - 1 && iy < N - 1) {
		d_A[ix + 1][iy + 1] = d_B[ix + 1][iy + 1]
			- c * dt / d * (d_B[ix + 1][iy + 1] - d_B[ix + 1][iy])
			- c * dt / d * (d_B[ix + 1][iy + 1] - d_B[ix][iy + 1]);
	}

	// Boundary conditions
	if (ix == 0 || ix == N || iy == 0 || iy == N) {
		d_A[ix][iy] = 1;
	}
}
```

Notice that for the thread indexes, there is now both an x- and y-direction. This will be discussed further as we step through the code. For now, it is possible to start looking at our int main() function:

```
	// Declaring on host
	double A[N][N];
	double B[N][N];
	double C[N][N];
```

A very clear concept: simply declare square matrices of N x N size, which holds just enough elements for the computation. In the next bits of code, one can observe more efficient memory management and greater attention to error-checking:

```
cudaError_t cudaStatus;

// Device pointers
double (*d_A)[N], (*d_B)[N], (*d_C)[N];

// Allocate buffer for GPU
cudaStatus = cudaMalloc((void**)& d_A, N * N * sizeof(double));
if (cudaStatus != cudaSuccess) {
	fprintf(stderr, "cudaMalloc failed!");
}
```

Device pointers are declared which will be moved from the host memory to the device memory. The cudaMalloc() function is called, which allocates a buffer for the pointer address in the GPU. Since the declared pointers are of double**, and cudaMalloc expects a void** pointer we must cast it using (void**).

```
// Copy to device from host
cudaStatus = cudaMemcpy(d_A, A, N * N * sizeof(double),
cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {
	fprintf(stderr, "cudaMemcpy failed!");
}
```

A fairly self-explanatory section, here data is copied from the host to the device by calling cudaMemcpy(). The syntax is as follows: copy to d_A, the contents of A, giving N * N * sizeof(double) space, from host to device.

```
// One block only
int blocks = 1;

// One thread per element
dim3 threads(N, N);

stepFunction2D<<<blocks, threads>>>(d_A, d);
```

It is worthwhile to discuss the penultimate line. The *dim3* object *threads* is used in the kernel launch to specify the number of dimensions per block. Up to 3 can be defined; if no input is given then it is taken to be 1. Therefore, using (N, N) as arguments translates to inputting (N, N, 1) i.e. there will be N threads in the x-direction and N threads in the y-direction, giving a total of 32 * 32 = 256 threads per block. Since we have only one block (again for simplification in implementation) this means there is one thread per element, as was discussed before.

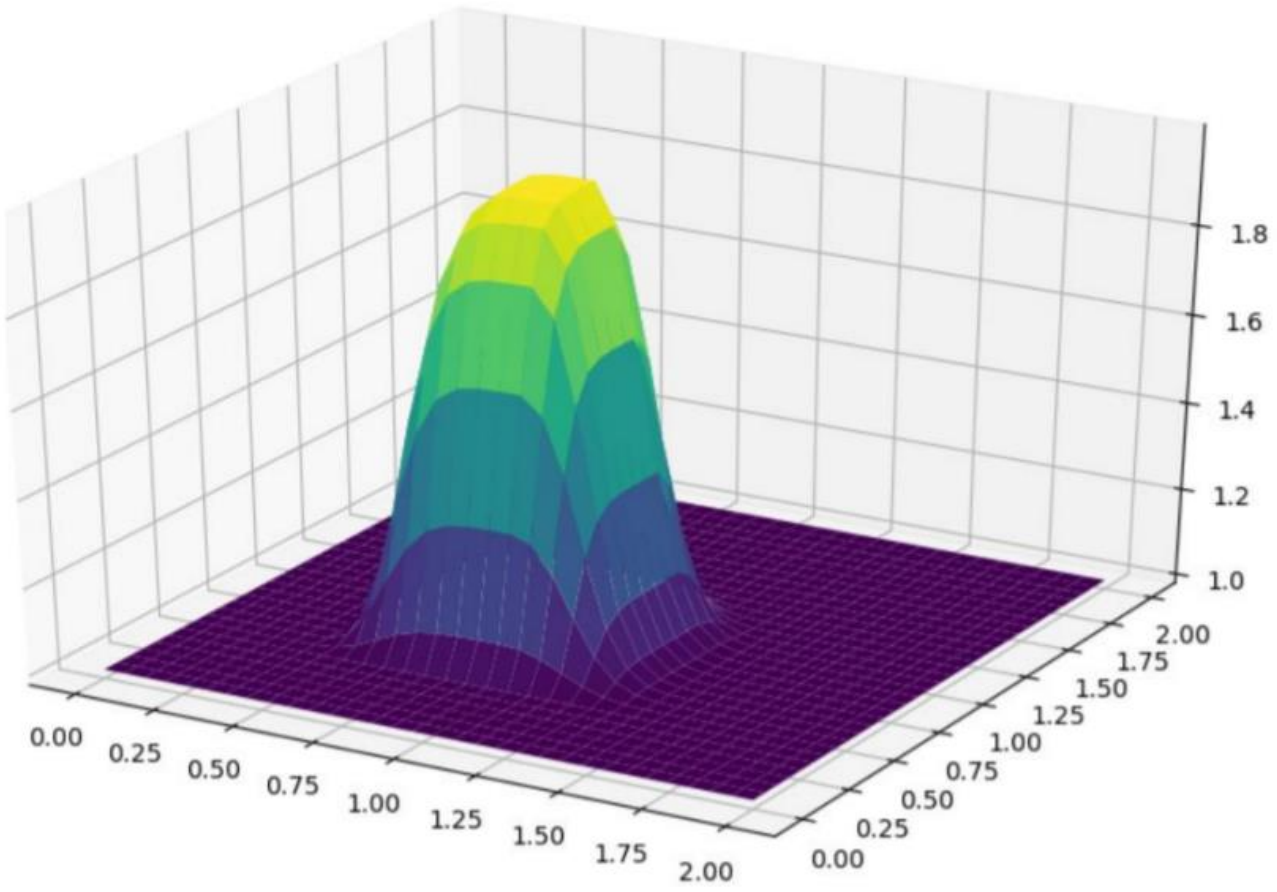
```
	for (int t = 0; t < nt; t++) {
		matrixCopy<<<blocks, threads>>>(d_A, d_B);
		finiteDiff2D<<<blocks, threads>>>(d_A, d_B, d, dt, c);
	}

	cudaStatus = cudaGetLastError();
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "Kernel launch failed: %s\n",
cudaGetErrorString(cudaStatus));
	}

	cudaStatus = cudaDeviceSynchronize();
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
launching kernel!\n", cudaStatus);
	}

	// Copy back to host from device
	cudaStatus = cudaMemcpy(C, d_C, N * N * sizeof(double),
cudaMemcpyDeviceToHost);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
	}
```

The for-loop in time was removed from the finiteDiff2D() kernel (in contrast to the 1-D version where it was inside the kernel) to separate tasks and write cleaner code. The matrixCopy() kernel is launched to create deep copies for use in finiteDiff2D(). At the end of the loop, cudaGetLastError() is called to ensure everything launched properly; it was through this error-checking procedure that the author was able to identify that they were exceeding the maximum number of threads per block! Hence, implementing such error-checking is fundamental in being able to develop quality CUDA kernels on time.

cudaDeviceSynchronize() is again called to make sure the CPU waits for the GPU to finish running its kernels. Then cudaMemcpy() is again called but this time data from the GPU memory (device) is copied to the CPU (host). Finally, in line with standard memory management practice, memory is freed using cudaFree() to ensure there are no memory leaks:

```
	cudaFree(d_A);
	cudaFree(d_B);
	cudaFree(d_C);
```

Some visualisation of the final output after launching the parallelised finite difference kernel, again done in Python:



As in the 1-D case, we note that there is numerical diffusion due to the differencing scheme being only first-order accurate.

4.2.1 Time reduction analysis of 2-D version

Given that the 2-D parallelised implementation has been completed, it is possible to compare the execution time of the parallelised version with that of the serial version (as was done with the 1-D version). First, the benchmark run times for the monolithic kernel from the profiler *nvprof* are listed below:

```
==11324== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   52.67%  47.264us        10  4.7260us  4.6400us  5.3120us  finiteDiff2D(double[32]*, double[32]*, double, double, double)
                   31.35%  28.128us        10  2.8120us  2.7520us  3.3280us  matrixCopy(double[32]*, double[32]*)
```

With a run time of **47.26 microseconds**. So now the question remains, how is it possible to rank the performance of this parallelised programme? In the 1-D case, this was straightforward as the kernel could simply be launched with one block and one thread to emulate serial processing. However, for the 2-D version i.e. a monolithic version, this is not possible hence an implementation using for-loops has been programmed instead. The resulting function's run time can therefore be compared to the monolithic kernel. Below is the code for the version implementing for-loops; it should be stressed that the function has only been programmed as a CUDA kernel so as to be able to take advantage of *nvprof's* profiling capabilities:

```
__global__ void stepFunction2D(double d_A[N][N], double d) {
	int ix = blockIdx.x * blockDim.x + threadIdx.x;
	int iy = blockIdx.y * blockDim.y + threadIdx.y;

	for (iy; iy < N; iy++) {
		for (ix; ix < N; ix++) {

			if (ix * d >= 0.5 && ix * d <= 1 && iy * d >= 0.5 && iy * d <=
1) {
				d_A[ix][iy] = 2;
			}
			else {
				d_A[ix][iy] = 1;
			}
		}
	}
}

__global__ void finiteDiff2D(double d_A[N][N], double d_B[N][N], const double d,
const double dt, const double c) {
	int ix = blockIdx.x * blockDim.x + threadIdx.x;
	int iy = blockIdx.y * blockDim.y + threadIdx.y;

	for (iy; iy < N; iy++) {
		for (ix; ix < N; ix++) {

			if (ix < N - 1 && iy < N - 1) {
				d_A[ix + 1][iy + 1] = d_B[ix + 1][iy + 1]
					- c * dt / d * (d_B[ix + 1][iy + 1] - d_B[ix +
1][iy])
					- c * dt / d * (d_B[ix + 1][iy + 1] - d_B[ix][iy
+ 1]);
			}

			// Boundary conditions
			if (ix == 0 || ix == N || iy == 0 || iy == N) {
				d_A[ix][iy] = 1;
			}
		}
	}
}
```

Only the relevant functions, stepFunction2D() to set the initial conditions, and finiteDiff2D(), to perform the finite differencing, have been shown for the sake of succinctness. The complete code can be found at the author's GitHub by following this link. *nvprof* gives the below output:

```
==11288== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   80.14%  314.95us        10  31.494us  31.201us  31.872us  finiteDiff2D(double[32]*, double[32]*,
double, double, double)
                  11.20%  44.000us         1  44.000us  44.000us  44.000us  stepFunction2D(double[32]*, double)
```

As highlighted and in red, the for-loop function execution time is **314.95 microseconds**.

It is possible to conclude this discussion having shown that the for-loop function takes longer to execute than the monolithic kernel. However, other examples can be also be shown, written in Python, that also demonstrate how C++ CUDA GPU computing drastically speeds up code execution time. The below Python code shows that all the parameters have been initialised using the same values:

```
# New Library required for projected 3D plots

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import pyplot, cm

import time
import numpy

%matplotlib inline

# variable declarations

nx = 32
ny = 32
nt = 10

dx = 2 / (nx - 1)
dy = 2 / (ny - 1)

c = 1
sigma = .2
dt = sigma * dx

x = numpy.linspace(0, 2, nx)
y = numpy.linspace(0, 2, ny)

u = numpy.ones((ny, nx)) # create a 1 x n vector of 1's
un = numpy.ones((ny, nx)

# Assign hat function initial conditions
u[int(.5 / dy):int(1 / dy + 1),int(.5 / dx):int(1 / dx + 1)] = 2

# Plot ICs
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
surf = ax.plot_surface(X, Y, u[:], cmap=cm.viridis)
```

Then again a for-loop finite difference solver is implemented, but in Python:

```
start = time.time()

for n in range(nt + 1): # loop across number of time steps
    un = u.copy()
    row, col = u.shape
    for j in range(1, row):
        for i in range(1, col):
            u[j, i] = (un[j, i] - (c * dt / dx * (un[j, i] - un[j, i -
1])) -
                                   (c * dt / dy * (un[j, i] - un[j - 1,
i])))
            u[0, :] = 1
            u[-1, :] = 1
            u[:, 0] = 1
            u[:, -1] = 1

end = time.time()

print(end - start)
```

The result from "print(end – start)" outputs how long it takes for the for-loop to run, which in this case was **73.23 milliseconds**. Conversely, a more Pythonic version was also coded which uses array operations instead of for-loops using the finite difference method, which is more akin to the monolithic version:

```
start = time.time()

for n in range(nt + 1): # loop across number of time steps
    un = u.copy()
    u[1:, 1:] = (un[1:, 1:] - (c * dt / dx * (un[1:, 1:] - un[1:, :-1])) -
                              (c * dt / dy * (un[1:, 1:] - un[:-1, 1:])))
    u[0, :] = 1
    u[-1, :] = 1
    u[:, 0] = 1
    u[:, -1] = 1

end = time.time()

print(end - start)
```

Which gives a run time of **4.84 milliseconds.** Available to us now are four different finite difference code run times, implemented in both serial & parallel and C++ & Python. These are tabulated on the right, in ascending order of execution time.

Rank	Code (implementation type)	Run time [microseconds]
1	C++ CUDA (monolithic kernel)	47.26
2	C++ CUDA (for-loops)	314.95
3	Python (array operations)	4840
4	Python (for-loops)	73230

It is clear from these results that as per this study, the C++ CUDA monolithic kernel is the best choice as it is the most rapid.

5 Concluding remarks and areas of improvement

This report set out to showcase the author's coding capacity. This has been done by showcasing both serialised and parallelised code solving the convection equation, using Python as a starting point and transitioning to C/C++ CUDA. Besides discussing some limited theory, the focus was on GPU computing. The author stepped through the code, identifying and describing where exactly parallelisation is useful. The interaction between CPU and GPU (host and device), what is loaded to the GPU and how improved parallelisation had been achieved was also discussed. Furthermore, it was demonstrated and quantified through several examples how parallelised C++ CUDA kernels run faster than their serialised counterparts, whether these are in C++ or Python. This results have been tabulated, and it was established that the monolithic kernel was the fastest option, with a run time of **47.26 microseconds**.

There still lie some areas of improvement however. For example, __shared__ memory is available to threads that are all in the same block. Since in this case each element is accessed twice, some further optimisation could be done in this area to improve memory coalescence. Or for example, there is an instruction cost when using if-statements (including the implicit if-statements in for-loops). A method implementing a grid-stride loop in 2-D could also be implemented, for instance. Furthermore, code run time measurements could be improved and made more consistent by being "closer to the metal" e.g. the Python code was timed in a Jupyter notebook. Overall, through grasping how parallelisation is useful in this exercise, it might be possible for the author to do so as well for the SEAMLESS-WAVE framework, by applying similar optimisation concepts during the project.