University of Nottingham
UK | CHINA | MALAYSIA

EEEE4115 Advanced Computational Engineering

Introduction to GPU Programming with CUDA ®

NVIDIA's CUDA: A Recap

CUDA Programming Model

- The GPU is seen as a 'compute device' to execute all or part of an application that:
  - Can be isolated as a function
  - Has to be executed many times
  - Is highly data parallel
  - Is lightweight
- The function when compiled to run on the 'device' is called a 'Kernel'

NVIDIA's CUDA: A Recap

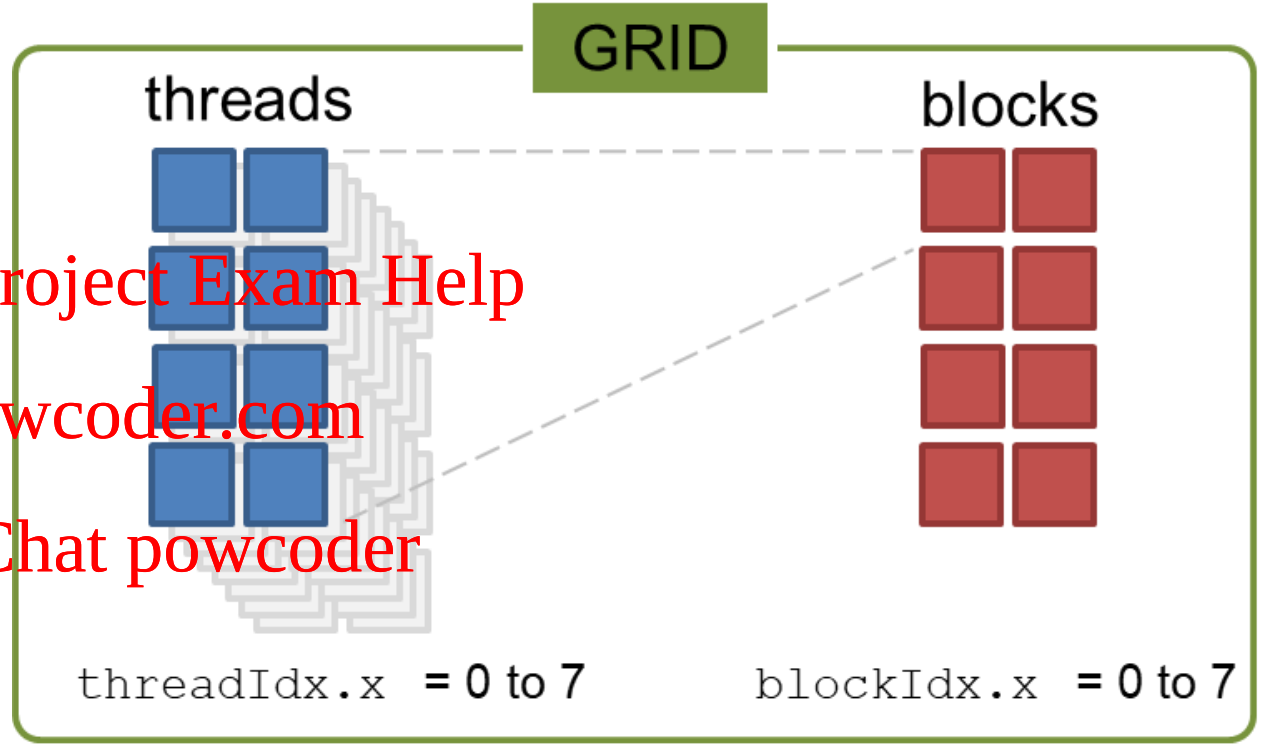
CUDA Programming Model

- **Allocate memory on device**
  - Allocated by cudaMalloc()
- **Copy input arrays from host to device memory**
  - cudaMemcpy()
- **Execute kernel(s) on the device**
  - kernel <<< block size, number of blocks, shared memory size >>>(parameters);
- **Copy output arrays from device to host**
  - cudaMemcpy()

NVIDIA's CUDA: A Recap

Thread/Block Model

- Each thread has a unique local index.
- Each block has a unique local index.
- These indices can be used to index data in arrays

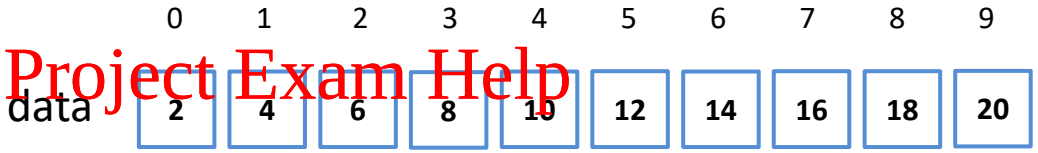



```
kernel<<<8,8>>> (..);
```

Thread/Block Model

int data [10] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20}

creates:



to select i^{th} element:

kernel <<<1,10>>>:
```
int i = threadIdx.x;
do something with data[i];
```

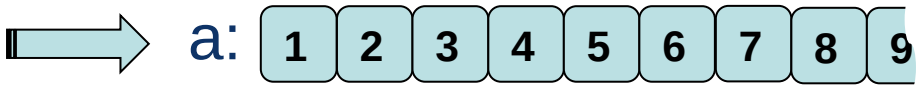
kernel <<<10,1>>>:
```
int i = blockIdx.x;
do something with data[i];
```

kernel <<<2,5>>>:
```
int i = blockDim.x * blockIdx.x  + threadIdx.x;
do something with data[i];
```

```
(kernel <<<number of blocks, number of threads per block>>>(…);  )
```

NVIDIA's CUDA: A Recap

`a[] = {1,2,3,4,5,6,7,8,9,…}` (number of elements = int `length`)

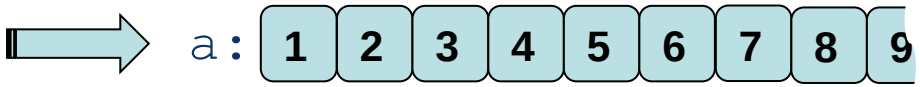


CPU code to access elements of `a[]`:

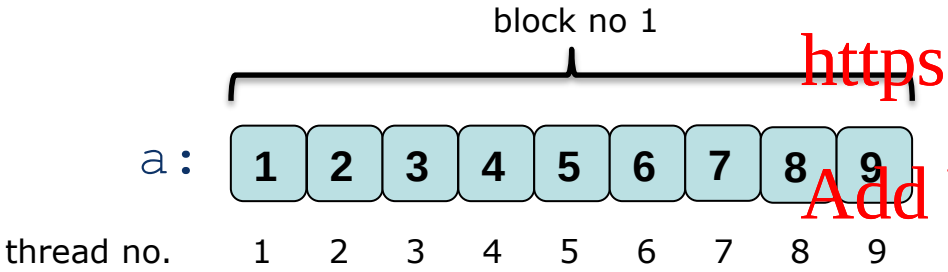
```
void foo (int *a, int *b) {
    for (int i=0; i<length; i++) {
    b[i] = a[i];
    }
}
```

NVIDIA's CUDA: A Recap

`a[] = {1,2,3,4,5,6,7,8,9,…}` (number of elements = int `length`)



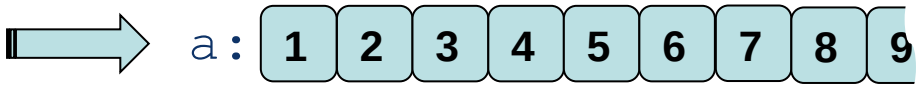
GPU code to access elements of `a[]` using block of threads:



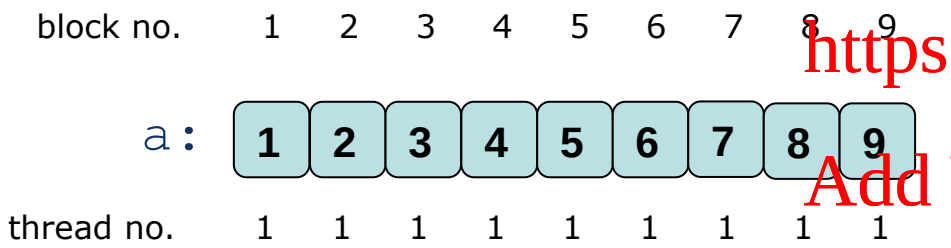

```
__global__ void foo(int *a, int *b) {
    int index = threadIdx.x; //unique thread ID
    if (index < length)
        b[index] = a[index];
}
```

Remember, max. number of threads = 512 or 1024 (hardware dependant)

a[] = {1,2,3,4,5,6,7,8,9,…} (number of elements = int length)



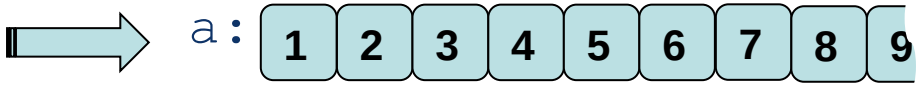
GPU code to access elements of a[] using blocks of single threads:




```
__global__ void foo(int *a, int *b) {
    int index = blockIdx.x; //unique block ID
    if (index < length)
        b[index] = a[index];
}
```

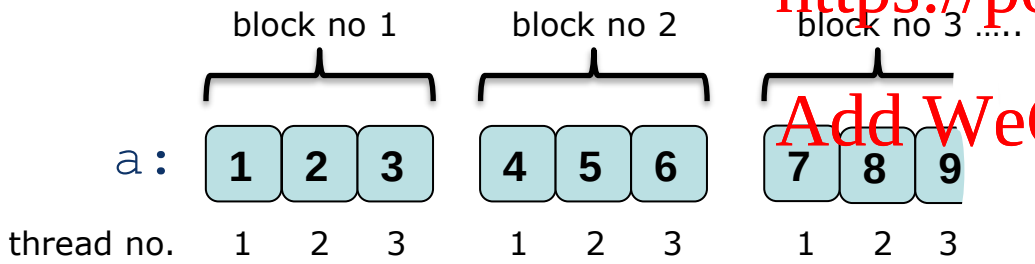
Remember, max. number of blocks = 65535

`a[] = {1,2,3,4,5,6,7,8,9,…}` (number of elements = int `length`)



GPU code to access elements of `a[]` using blocks of n threads:

e.g. n = 3:




```
__global__ void foo(int *a, int *b) {
    int index = threadIdx.x+blockIdx.x*blockDim.x;
    if (index < length)
        b[index] = a[index];
}
```

Example: Computing π

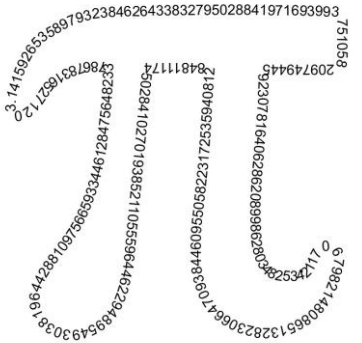
Illustrating memory types

Computing π

Approach

- **3 tier reduction:**
  - Every thread adds a number of terms
  - The first thread in each block adds all results from the block
  - The CPU does the remaining reductions
- **3 types of memory**
  - Each thread uses private memory to store their partial results
  - The first thread of a block needs access to all results in the block using shared memory
  - The block totals need to by copied to CPU global memory

$$\pi = \int_0^1 \frac{1}{1+x^2} \approx 4\Delta x \sum_{i=0}^{N} \frac{1}{1+x_i^2}$$



Computing π - Device Code

```
__global__ void pi(float *blockSums, int stepsPerThread, float dx) {

    __shared__ float threadSums[BLOCKSIZE];

    int id      = threadIdx.x + blockDim.x * blockIdx.x;
    int istart  = id * stepsPerThread;
    int istop   = istart + stepsPerThread;

    float accum = 0.0f;

    for (int i = istart; i < istop; i++) {
        float x = (i + 0.5f) * dx;
        accum += 4.0f / (1.0f + x*x);
    }
    threadSums[threadIdx.x] = accum;

    __syncthreads();

    if (threadIdx.x == 0) {
        float blockSum = 0.0f;
        for (int j = 0; j < blockDim.x; j++) {
            blockSum += threadSums[j];
        }
        blockSums[blockIdx.x] = blockSum;
    }
}
```

allocate work to each thread

each thread calculates their contribution to overall approximation

wait for all threads to finish

calculate sum of contributions from threads in each block

Computing π - Host Code

```
39  int main()
40  {
41      cudaError_t err;
42
43      const int stepsPerThread = 2048;
44      const int blockSize = BLOCKSIZE;
45      const int numBlocks = 256;
46
47      const int numSteps = blockSize * numBlocks * stepsPerThread;
48      const float dx = 1.0f / numSteps;
49
50      float *h_blockSums = (float *)malloc(sizeof(float) * numBlocks);
51      float *d_blockSums;
52
53      err = cudaMalloc((void**)&d_blockSums, sizeof(float) * numBlocks);
54          assert(err == cudaSuccess);
55
56      err = cudaMemcpy(d_blockSums, h_blockSums, sizeof(float) * numBlocks, cudaMemcpyHostToDevice);
57          assert(err == cudaSuccess);
58
59      pi<<<numBlocks, blockSize>>> (d_blockSums, stepsPerThread, dx);
60
61      err = cudaMemcpy(h_blockSums, d_blockSums, sizeof(float) * numBlocks, cudaMemcpyDeviceToHost);
62          assert(err == cudaSuccess);
63
64      float pi = 0.0f;
65      for (int i = 0; i < numBlocks; i++)
66          pi += h_blockSums[i];
67
68      pi *= dx;
69
70      printf("pi approximately equals: %f\n", pi);
71
72      cudaFree(d_blockSums);
73      free(h_blockSums);
74
75      return 0;
76  }
```

CUDA error type

assign total amount of work per thread/block

allocate memory on device, copy arrays and check for errors

call kernel

copy sums from each block

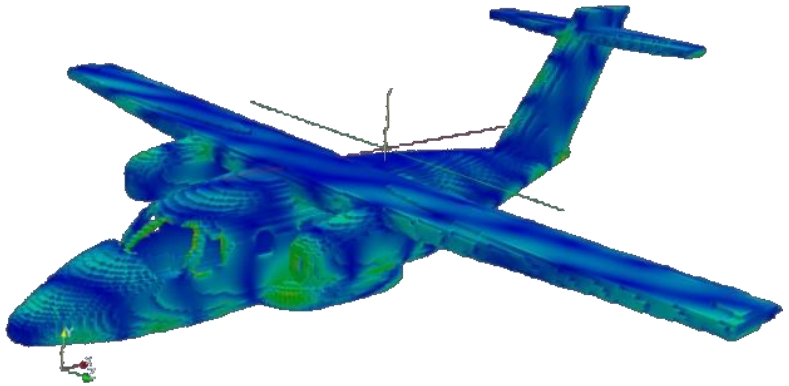
sum all

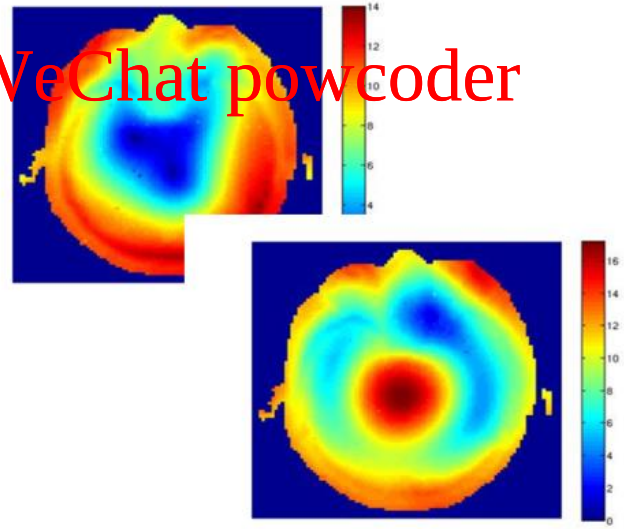
Example: Computational Electromagnetics

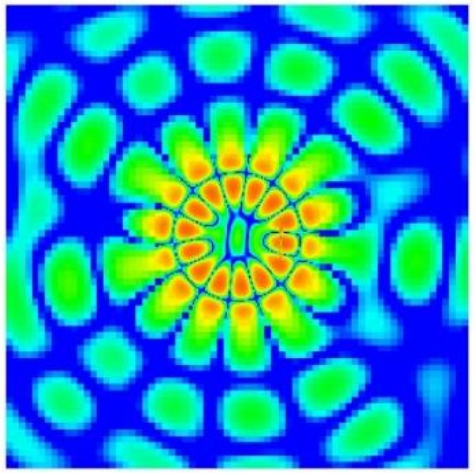
Porting the 1D TLM Algorithm to CUDA

Transmission Line Method

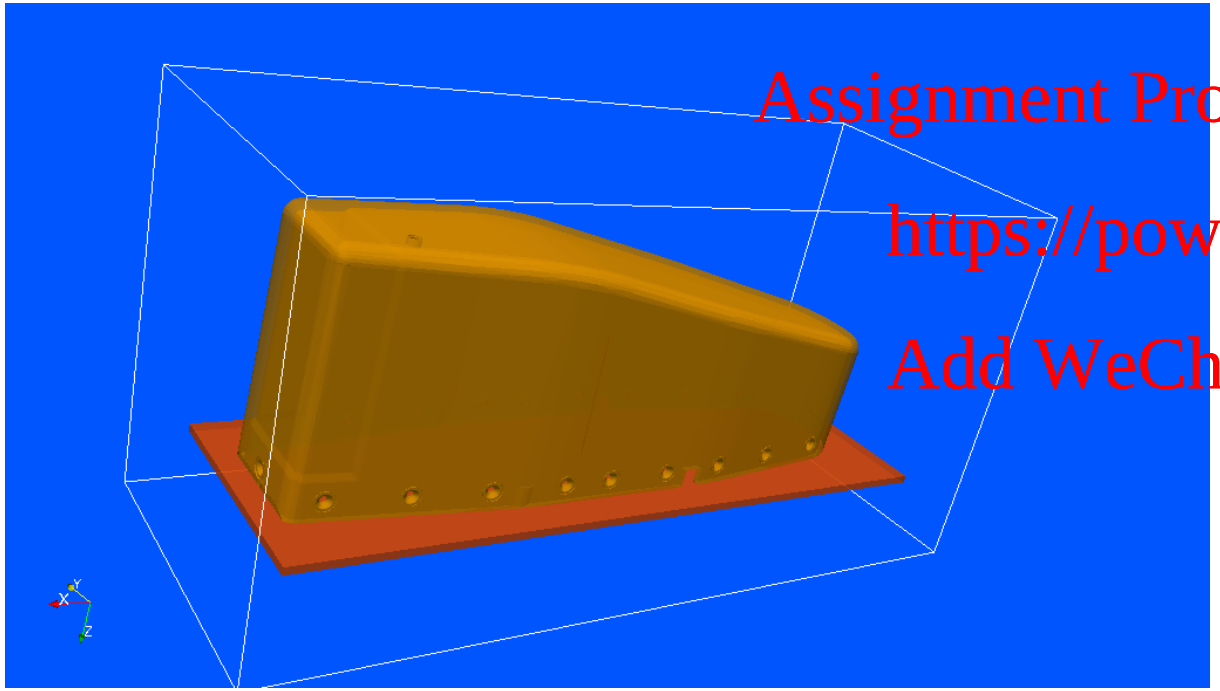
- Founded and developed within the University for over 40 years
- Time domain method for the computation of electromagnetic fields
- Problems are discretised in time and space
- The computational domain being broken down into a mesh of transmission line segments connected at ‘nodes’
- Extremely powerful and underpins the simulation strategies employed commercial software

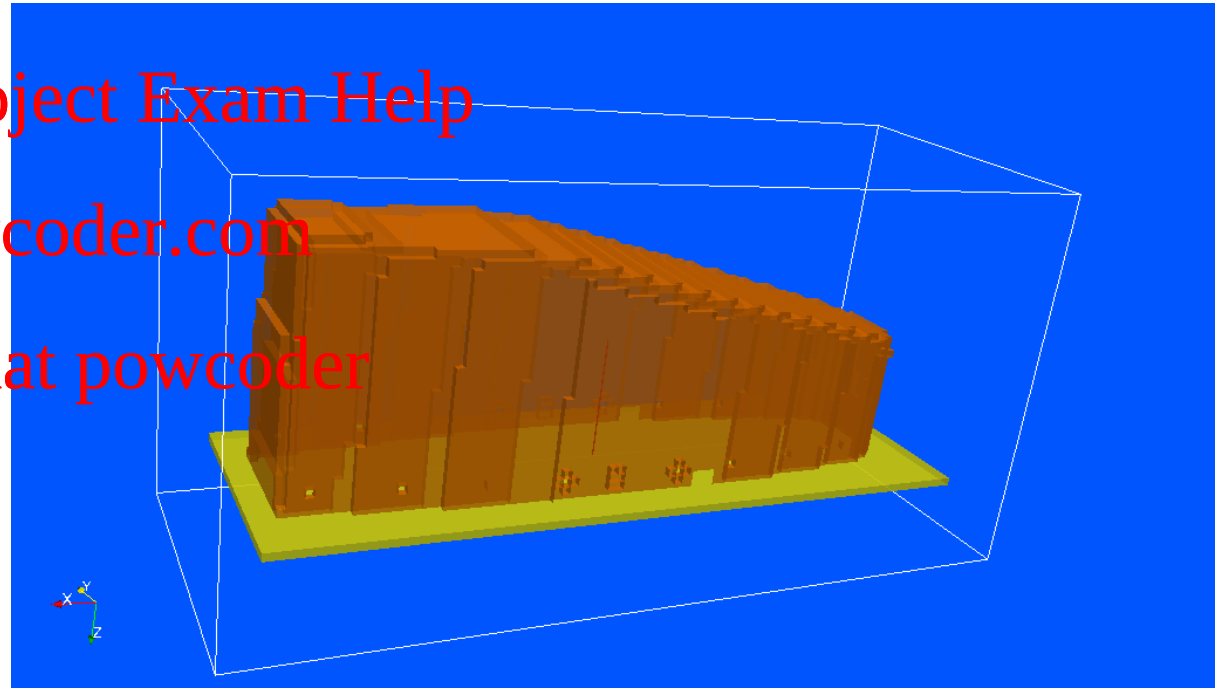






TLM Example: Aircraft Radome





TLM Example: Aircraft Radome

1D TLM Algorithm

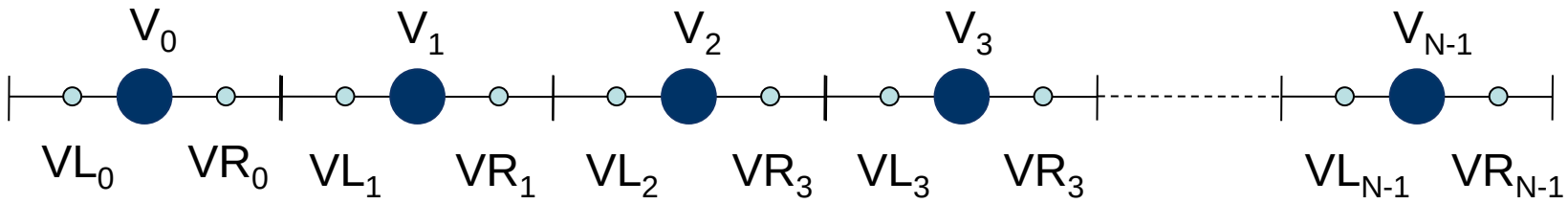
The Algorithm - Consider a line in free space i.e. a wire.

Problem space subdivided into N equal sections (nodes) . Simulation is then run for a total time T, over a number of time steps of Δt.

A simplified description of the process. At each Δt, two operations occur in sequence:

1. VL_n and VR_n are obtained from V_n – The 'scatter' process
2. VR_n and VL_{n+1} and are swapped – The 'connect' process
3. The new V_n values are calculated

Operations are highly data parallel!



1st Things 1st

- **Establish Benchmarks:**
  - Execution Time for CPU application
  - Output from CPU application

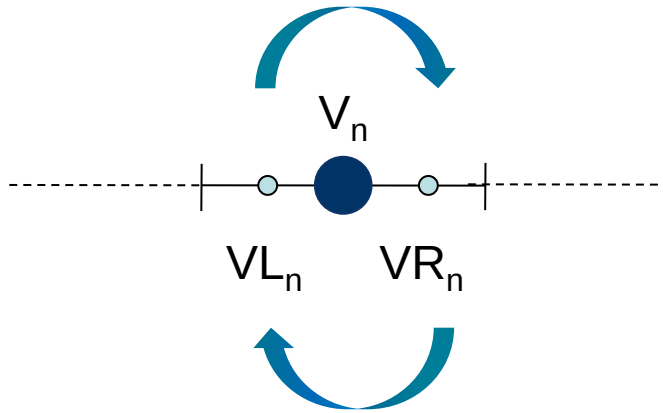
1D TLM Algorithm

```
int main()
{
            .
            .
            .
for (int m=0;m<NT;m++)
{
    source = tlmSource (m*dt, delay,
                                width);
    tlmScatter (VL, VR, N, source);
    tlmConnect (VL, VR, N);
    output << dt*m <<"  "<< VL[2] +
                        VR[2] << endl;
}
```

1D TLM Algorithm

```
int main()
{
            .
            .
            .
for (int m=0;m<NT;m++)
{
    source = tlmSource (m*dt, delay,
                                width);
    tlmScatter (VL, VR, N, source);
    tlmConnect (VL, VR, N);
    output << dt*m <<"  "<< VL[2] +
                        VR[2] <<endl;
}
```

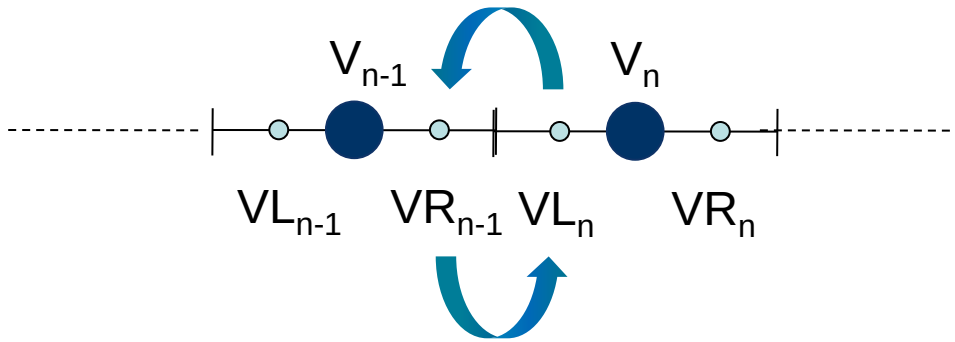
```
void tlmScatter(float* VL, float* VR, int N,
                                      float source)
{
    float V;
    //apply source
    VL[0] += source;
    //scatter
    for(int n=0;n<N;n++)
    {
        V = VL[n] + VR[n];
        VR[n] = V - VR[n];
        VL[n] = V - VL[n];
    }
}
```

1D TLM Algorithm

```
int main()
{
            .
            .
            .
for (int m=0;m<NT;m++)
{
    source = tlmSource (m*dt, delay,
                                width);
    tlmScatter (VL, VR, N, source);
    tlmConnect (VL, VR, N);
    output << dt*m <<"  "<< VL[2] +
                        VR[2] << endl;
}
```

```
void tlmConnect(float* VL, float* VR, int N)
{
float V;
//connect
for(int n=1;n<N;n++)
    {
        V = VR[n-1];
        VR[n-1] = VL[n];
        VL[n] = V;
    }
    //apply boundary conditions
    VR[N-1] = -1*VR[N-1];
    VL[0] = 0*VL[0];
}
```

1D TLM Algorithm

```
//allocate memory on device
cudaMalloc((void**)&dev_VR, N*sizeof(float));
cudaMalloc((void**)&dev_VL, N*sizeof(float));
cudaMalloc((void**)&dev_Vt, NT*sizeof(float));

//zero arrays ......

//copy memory areas from host to device
cudaMemcpy(dev_VR, VR, N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_VL, VL, N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_Vt, Vt,
NT*sizeof(float),cudaMemcpyHostToDevice);

//run TLM algorithm for NT times steps
for (int m = 0; m < NT; m++)
    {
    float source = tlmSource(m*dt, delay, width);
    tlmScatter  <<<1,N>>>( dev_VR, dev_VL, N, m, source);
    tlmConnect<<<1,N>>>( dev_VR, dev_VL, dev_Vt, N, m);
    }
```

Allocate memory on device

1D TLM Algorithm

```
//allocate memory on device
cudaMalloc((void**)&dev_VR, N*sizeof(float));
cudaMalloc((void**)&dev_VL, N*sizeof(float));
cudaMalloc((void**)&dev_Vt, NT*sizeof(float));

//zero arrays ......

//copy memory areas from host to device
cudaMemcpy(dev_VR, VR, N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_VL, VL, N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_Vt, Vt, NT*sizeof(float),cudaMemcpyHostToDevice);

//run TLM algorithm for NT time steps
for (int m = 0; m < NT; m++)
    {
    float source = tlmSource(m, width, 100);
    tlmScatter  <<<1,N>>>( dev_VR, dev_VL, N, m, source);
    tlmConnect<<<1,N>>>( dev_VR, dev_VL, dev_Vt, N, m);
    }
```


Copy memory areas from host to device

1D TLM Algorithm

```
//allocate memory on device
cudaMalloc((void**)&dev_VR, N*sizeof(float));
cudaMalloc((void**)&dev_VL, N*sizeof(float));
cudaMalloc((void**)&dev_Vt, NT*sizeof(float));

//zero arrays ......

//copy memory areas from host to device
cudaMemcpy(dev_VR, VR,
N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_VL, VL,
N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_Vt, Vt,
NT*sizeof(float),cudaMemcpyHostToDevice);

//run TLM algorithm for NT times steps
for (int m = 0; m < NT; m++)
    {
    float source = tlmSource(m*dt, delay, width);
    tlmScatter <<<1,N>>>( dev_VR, dev_VL, N, m, source);
    tlmConnect <<<1,N>>>( dev_VR, dev_VL, dev_Vt, N, m);
    }
```


Launch kernels on device

Note: for generality scatter & connect are launched as two separate kernels to ensure the scatter completes before the connect.

If using single block of threads then both routines could be in the same kernel

1D TLM Algorithm

```
//allocate memory on device
cudaMalloc((void**)&dev_VR, N*sizeof(float));
cudaMalloc((void**)&dev_VL, N*sizeof(float));
cudaMalloc((void**)&dev_Vt, NT*sizeof(float));

//zero arrays .....

//copy memory areas from host to device
cudaMemcpy(dev_VR, VR, N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_VL, VL, N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_Vt, Vt, NT*sizeof(float),cudaMemcpyHostToDevice);

//run TLM algorithm for NT times steps
for (int m = 0; m < NT; m++)
  {
  float source = tlmSource(m*dt, delay, width);
  tlmScatter  <<<1,N>>>( dev_VR, dev_VL, N, m, source);
  tlmConnect<<<1,N>>>( dev_VR, dev_VL, dev_Vt, N, m);
  }
```

```
__global__ void tlmScatter (float* VR,
float* VL,int N, int m, float source)
{
    unsigned int idx = threadIdx.x;
    //apply source
    if (idx == 0)
        VL[0] += source;

    //scatter
    if (idx < N)
    {
        float V = VL[idx] + VR[idx];
        VR[idx] = V - VR[idx];
        VL[idx] = V - VL[idx];
    }
}
```

* Kernel calls for each timestep m are queued on device

1D TLM Algorithm

```
//allocate memory on device
cudaMalloc((void**)&dev_VR, N*sizeof(float));
cudaMalloc((void**)&dev_VL, N*sizeof(float));
cudaMalloc((void**)&dev_Vt, NT*sizeof(float));

//zero arrays ......

//copy memory areas from host to device
cudaMemcpy(dev_VR, VR, N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_VL, VL, N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(dev_Vt, Vt, NT*sizeof(float),cudaMemcpyHostToDevice);

//run TLM algorithm for NT times steps
for (int m = 0; m < NT; m++)
   {
   float source = tlmSource(m*dt, delay, width);
   tlmScatter  <<<1,N>>>( dev_VR, dev_VL, N, m, source);
   tlmConnect<<<1,N>>>( dev_VR, dev_VL, dev_Vt, N, m);
   }
```

```
__global__ void tlmConnect (float* VR,
float* VL, float* Vt, int N, int m)
{
    unsigned int idx = threadIdx.x;

    //Connect
    if (idx > 0 && idx < N)
    {
        float V = VR[idx-1];
        VR[idx-1] = VL[idx];
        VL[idx] = V;
    }

    //apply boundaries
    if (idx == 0)
    {
        VR[N-1] *= -1.f;
        VL[0] = 0.f;

        Vt[m] = VL[2] + VR[2]; // o/p
    }
}
```

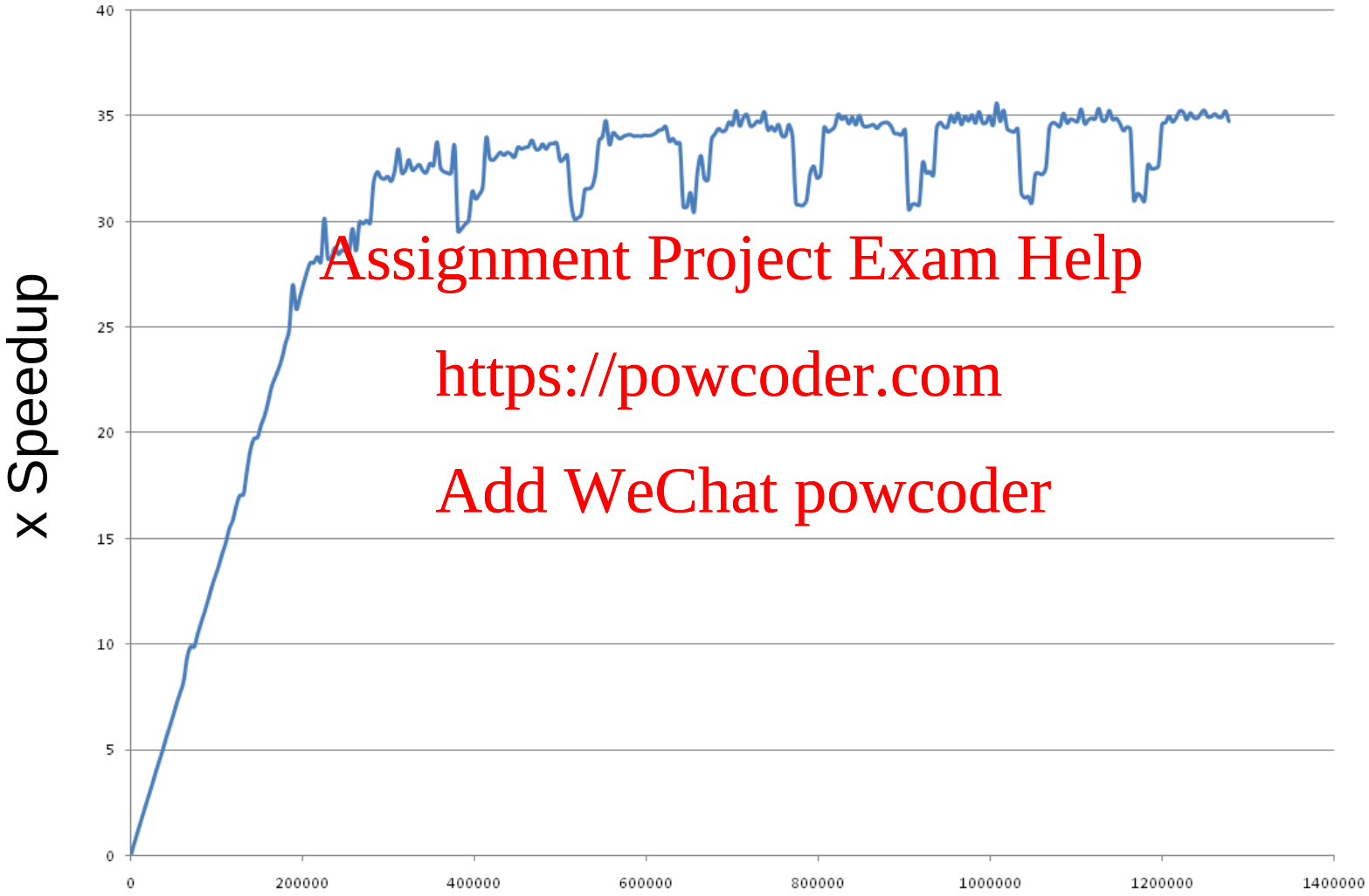
1D TLM Algorithm

```
__global__ void tlmScatter (float* VR, float* VL, int N, int m, float
source) {
    unsigned int idx = threadIdx.x + blockIdx.x*blockDim.x;
    //apply source
    if (idx == 0)  //only one thread to apply source
        VL[0] += source;
    //scatter
    if (idx < N)  {
        float V = VL[idx] + VR[idx];
        VR[idx] = V - VR[idx];
        VL[idx] = V - VL[idx];
    }
}
```



```
int main () {
........
unsigned int nBlocks = (N+BLOCKSIZE-1)/BLOCKSIZE;//Ensure sufficient number of blocks
    for (int m = 0; m < NT; m++)
    {
         float source = tlmSource(m*dt, delay, width);
         tlmScatter<<<nBlocks,BLOCKSIZE>>>( dev_VR, dev_VL, N, m, source);
         tlmConnect<<<nBlocks,BLOCKSIZE>>>( dev_VR, dev_VL, dev_Vt, N, m);
    }
```

1D TLM Algorithm



CUDA Coursework 2

Porting the 1D TLM Algorithm to CUDA

CUDA Coursework 2

Coursework is worth 25% of the Module

- **Full submission to include the full application code as a single .cu file and a brief report (approximately 10 pages) that includes:**
  - A benchmark of the provided 2D TLM code on your platform of choice (illustrative result and run time)
  - A discussion that identifies the areas of the code that are data parallel
  - A description of the scatter and connect functions as implemented on the GPU Obtain performance results
  - A discussion of any routes taken to optimise the application
  - A benchmark of the GPU application (illustrative result, runtime and speed up compared to CPU version)
- **Coursework deadline 11th January 2021 at 15:00**