Computing virtual acoustics using the 3D finite difference time domain method and Kepler architecture GPUs

Craig J. Webb
Acoustics group / EPCC, University of Edinburgh
C.J.Webb-2@sms.ed.ac.uk

ABSTRACT

The computation of virtual acoustics for physical modelling synthesis using the finite difference time domain is a computationally expensive process, especially at audio rates such as 44.1kHz. However, the high level of data-independence is well suited to parallel architectures such as those provided by graphics processing units. This paper describes the use of the latest Nvidia Kepler cards to accelerate the computation of three-dimensional schemes. The CUDA language and hardware architecture allow many possible approaches to computing even a basic model. Various techniques are considered, such as full tiling, iteration slicing, and the use of shared memory. A standard simulation was used to measure the performance of these different approaches. Benchmark times were compared for the latest Nvidia Tesla K20 GPU against the previous generation cards. Results show the continuing maturity of the hardware, especially in terms of data caching, which allows basic code designs to perform as well as more complex shared memory versions.

1. INTRODUCTION

Virtual acoustics can be approached by direct numerical simulation of the three-dimensional wave equation. This can be used for auralizations, creating a model of a virtual environment, or for physical modelling synthesis by embedding instruments into the space. The finite difference time domain (FDTD) method is an efficient technique for computing such simulations [1]. However, for 3D systems at audio sample rates such an approach is still extremely computationally expensive [2], but can benefit from parallel computing using graphics processing units (GPUs) [3].

This paper examines the use of the latest Nvidia Kepler architecture GPUs to accelerate the computation of the standard FDTD discretisation of the 3D wave equation. The CUDA language and GPU hardware allow many different approaches to this particular computation, each of which produce the same output but with varying efficiency. Finding the optimal solution is a matter of experimentation and tuning of different implementation methods.

This paper examines six different approaches to a standard simulation model, ranging from 2D to 3D threading and paying close attention to the use of shared memory. These solutions are benchmarked on the latest Nvidia Tesla K20 card, as well as the previous generation Fermi Tesla card for comparison.

2. FINITE DIFFERENCE SCHEME

Virtual acoustic simulations using FDTD are based on the 3D wave equation, which in second order form is given by:

$$\frac{\partial^2 \Psi}{\partial t^2} = c^2 \nabla^2 \Psi \tag{1}$$

Here Ψ is the target acoustical field quantity, c is the wave speed in air, ∇^2 is the 3D Laplacian. The standard FDTD discretisation [4] leads to the following update equation for interior grid points:

$$w^{n+1}_{l,m,p} = (2 - 6\lambda^2) w^n_{l,m,p} + \lambda^2 S^n_{l,m,p} - w^{n-1}_{l,m,p} \tag{2}$$

where $w_{l,m,p}$ is the discrete acoustic field, $\lambda = \frac{cT}{X}$, and

$$\begin{aligned} S^n_{l,m,p} = {} & w^n_{l+1,m,p} + w^n_{l-1,m,p} + w^n_{l,m+1,p} \\ & + w^n_{l,m-1,p} + w^n_{l,m,p+1} + w^n_{l,m,p-1} \end{aligned} \tag{3}$$

The stability condition for the scheme can be derived from von Neumann analysis [4], such that for a given time step T the grid spacing X must satisfy:

$$X \geq \sqrt{3c^2T^2} \tag{4}$$

At the Courant limit where $\lambda = 1/\sqrt{3}$, the update equation reduces to:

$$w^{n+1}_{l,m,p} = \frac{1}{3} S^n_{l,m,p} - w^{n-1}_{l,m,p} \tag{5}$$

Fixed boundary conditions were used for testing. The update equation for a given grid point uses the six nearest neighbours from one time step ago, and the centre point from two time steps ago, as shown in figure 1.

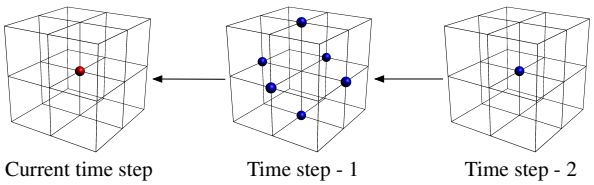

Figure 1. Grid points used for the update equation.

3. GPU COMPUTING USING CUDA

Nvidia's Kepler architecture is the 3rd generation of GPU cards that can be used specifically for general purpose computing using the CUDA language. The original compute 1.x cards were followed in 2010 by the Fermi architecture 2.x cards. With each generation, the hardware has changed significantly, along with the development of features in CUDA. Whilst the 3D FDTD wave equation computation is trivial to parallelize at each time step of the simulation, the downside is that it is clearly memory bandwidth limited. The compute-to-memory access ratio for each update is very low (generally less than two), and so achieving optimal efficiency on the GPU depends solely on the movement of data around the system.

GPU cards provide multiple memory types that can be programmed directly, as well as optimisations such as directing the use of cache lines [5]. These, together with the multiple options for threading the given data set, provide a wide scope of differing approaches to any given problem. Six different methods are considered here for the 3D FDTD scheme.

4. IMPLEMENTATION METHODS

The implementation design for the scheme consists of setup code, followed by a loop over of the time iterations of the simulation. Within this time loop the CUDA kernel threads are launched that update the state of the system, followed by processing the input and output, as shown in figure 2. Only two data grids are required for this basic scheme, as

Figure 2. Time loop with kernel launches.

```
1  for(n=0;n<NF;n++)
2  {
3      UpDate<<<dimGrid,dimBlock>>>(u_d,u1_d,L2);
4      // perform I/O
5      inout<<<1,1>>>(u_d,out_d,ins,n);
6      // update pointers
7      dummy_ptr = u1_d;
8      u1_d = u_d;
9      u_d = dummy_ptr;
10 }
```

the values from two time steps ago can be read from memory before overwriting the new state values.

The first consideration in terms of the threading design is whether to issue enough threads to update the entire 3D state, or to issue threads that cover a 2D slice and then use an iteration within the kernel over the remaining dimension. The latter approach allows greater flexibility in terms of data reuse, whilst limiting the number of threads in use. The kernel code should of course be designed to maximise *memory coalescing* with regard to the decomposition of the 3D data into linear memory.

Both approaches can utilise shared memory, which allows blocks of threads to store and use data in a collaborative manner. The main issue with using shared memory for finite difference schemes is the complication of always needing to access neighbouring grid points when at the edges of a thread block. Two different approaches are considered here, for each of the thread designs.

4.1 3D Tiling

The first method is the 3D tiling approach, where the entire 3D data set is covered by individual threads. So if the data set consists of one million grid points, then one million threads are issued to update the state. Threads are launched in groups known as *blocks*, and multiple blocks then make up the thread *grid*. Each of these objects can be one, two or three dimensional. Experimentation showed that a 32 x 4 x 2 thread block is most efficient here. The kernel code is shown in figure 3. Note that the neighbouring data

Figure 3. 3D Tiling kernel.

```
1  __global__ void UpDate(double *u,double *u1,double L2)
2  {
3      // get X,Y,Z from thread and block Id's
4      int X = blockIdx.x * Bx + threadIdx.x;
5      int Y = blockIdx.y * By + threadIdx.y;
6      int Z = blockIdx.z * Bz + threadIdx.z + 1;
7
8      // Test that not at halo, Z block excludes Z halo
9      if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1)) ){
10
11         // get linear position
12         int cp = Z*area+(Y*Nx+X);
13
14         u[cp] = L2*(u1[cp-1]+u1[cp+1]+u1[cp-Nx]
15                 +u1[cp+Nx]+u1[cp-area]+u1[cp+area])-u[cp];
16     }
17 }
```

points are accessed using shifts, and so only one calculation of the linearly decomposed position is required. Bx, By and Bz define the size of the thread block, with Nx, Ny and Nz defining the size of the 3D data grid, and 'area' is defined as Nx*Ny. This kernel is the simplest possible arrangement, reading data directly from global memory. The next step is to attempt to minimise data movement by using shared memory.

4.2 3D Tiling with shared memory

In order to implement a shared memory version of the above kernel, the main issue is how to deal with data access at the edges of the thread block. A 2D shared memory array is used, and in this version it will be the same size as the 2D thread block that is employed here. A block size of 32 x 8 was found to be most efficient. Figure 4 shows the new kernel code.

Each thread loads one element of data into the shared memory array (at line 16), followed by a thread synchronisation. Having tested that the current position is not a boundary point, the sum of the neighbouring grid points is computed. This requires four conditional statements, which pick up data from global memory if the position is it the edge of a thread block, otherwise it is read from the shared memory array. The final line reads the remaining Z-dimension neighbours, and writes the updated value to global memory.

The overall effect is to reduce the reads from global memory from six to two, when a given thread is not at the edge of a block, which is a significant reduction in memory access.

Figure 4. 3D Tiling with shared memory kernel.

```
1  __global__ void UpDate(double *u,double *u1,double L2)
2  {
3     __shared__ double uS1[BxS][ByS];
4
5     int tdx = threadIdx.x;
6     int tdy = threadIdx.y;
7
8     int X = blockIdx.x * BxS + tdx;
9     int Y = blockIdx.y * ByS + tdy;
10    int Z = blockIdx.z + 1;
11
12    int cp = Z*area+(Y*Nx+X);
13    double sum = 0.0;
14
15    // Load shared
16    uS1[tdx][tdy] = u1[cp];
17    __syncthreads();
18
19    // Test that not at halo, Z block excludes Z halo
20    if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1)) ){
21
22       if(tdx==0) sum+= u1[cp-1];
23       else sum+= uS1[tdx-1][tdy];
24
25       if(tdx==BxS-1) sum+= u1[cp+1];
26       else sum+= uS1[tdx+1][tdy];
27
28       if(tdy==0) sum+= u1[cp-Nx];
29       else sum+= uS1[tdx][tdy-1];
30
31       if(tdy==ByS-1) sum+= u1[cp+Nx];
32       else sum+= uS1[tdx][tdy+1];
33
34       u[cp] = L2*(sum+u1[cp-area]+u1[cp+area]) - u[cp];
35    }
36 }
```

Figure 5. 3D Tiling with extended shared memory kernel.

```
1  __global__ void UpDate(double *u,double *u1,double L2)
2  {
3     __shared__ double uS1[BxS+2][ByS+2];
4
5     int tdx = threadIdx.x;
6     int tdy = threadIdx.y;
7
8     int X = blockIdx.x * BxS + tdx;
9     int Y = blockIdx.y * ByS + tdy;
10    int Z = blockIdx.z + 1;
11
12    // get linear position
13    int cp   = Z*area+(Y*Nx+X);
14
15    // Load shared
16    tdx++; tdy++;
17    uS1[tdx][tdy] = u1[cp];
18
19    if ( (tdy==1) && !(Y==0) ){
20       uS1[tdx][tdy-1] = u1[cp-Nx];
21    }
22    if ( (tdy==ByS) && !(Y==(Ny-1)) ){
23       uS1[tdx][tdy+1] = u1[cp+Nx];
24    }
25    if ( (tdx==1) && !(X==0) ){
26       uS1[tdx-1][tdy] = u1[cp-1];
27    }
28    if ( (tdx==BxS) && !(X==(Nx-1)) ){
29       uS1[tdx+1][tdy] = u1[cp+1];
30    }
31    __syncthreads();
32
33    // Test that not at halo, Z block excludes Z halo
34    if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1)) ){
35
36       u[cp] = L2*(uS1[tdx-1][tdy]+uS1[tdx+1][tdy]
37                  +uS1[tdx][tdy-1]+uS1[tdx][tdy+1]
38                  +u1[cp-area]+u1[cp+area]) - u[cp];
39    }
40 }
```

4.3 3D Tiling with extended shared memory

A second approach to dealing with the problem of picking up data at the edges of a thread block is to use a shared memory array which is larger than the thread block and contains this edge data. Provided that the data is loaded correctly, a shared memory array of size [Bx+2][By+2] will contain all necessary data for the X and Y neighbour points. This 'extended' approach requires a more complicated arrangement for loading the shared memory data, but results in a much cleaner update from the neighbouring values as shown in figure 5.

Instead of four conditional statements around the summing point of the code, there are now four conditionals used in loading the shared memory array. If a thread is at the edge of a block it loads its own point plus one extra edge point. This loads the entire extended shared memory array, and so the update at line 36 reads all X and Y neighbours directly from the array.

Whilst this approach leads to the same reduction in global memory reads as the non-extended version, it does have a major effect on efficiency, as detailed in section six.

4.4 2D Slicing

The first three methods above have all focussed on a 3D tiling approach using the maximum amount of threading. The next three methods will employ a different strategy, namely using a 2D slicing approach. Instead of issuing threads to cover the entire data set, only enough threads are issued to cover a 2D slice of the data, for example a Z slice of size X by Y. Each thread then iterates over the remaining dimension (i.e. the Z dimension), updating all the grid points along that column. This of course requires a loop inside the kernel code itself.

To some extent, this approach is counter-intuitive. The main goal of parallel programming is to perform as much work as possible in parallel. However, there are advantages in using this approach, mainly in that it is easier to re-use data. In particular, it is possible to remove one of the two global memory reads in the Z dimension, as the thread is iterating over that data. By combining this with an X by Y shared memory tile, it is possible to reduce the global memory reads down to one single access per iteration over the Z dimension.

Prior to the Fermi architecture GPU cards, this technique was the most efficient approach to use with 3D data sets due to the lack a effective caching on the early cards [6]. However, this changed significantly with the compute 2.x architecture of Fermi, and made the 3D tiling approach useful. As caching levels continue to improve, it also leads to improved performance of direct global memory accessing, as will be demonstrated.

The kernel code for the basic global memory version of the 2D slicing is shown in figure 6. This closely resembles the kernel for the original 3D tiling method, but with the addition of a FOR loop over the interior of the Z dimension. A 64 x 8 thread block size was found to be the most efficient.

Figure 6. 2D Slicing kernel.

```
__global__ void UpDate(double *u,double *u1,double L2)
{
   // get X,Y,Z from thread and block Id's
   int X = blockIdx.x * BxL + threadIdx.x;
   int Y = blockIdx.y * ByL + threadIdx.y;
   int Z,cp;

   // Test that not at halo
   if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1)) ){

      for(Z=1;Z<(Nz-1);Z++){

        // get linear position
        cp = Z*area+(Y*Nx+X);

        u[cp]=L2*(u1[cp-1]+u1[cp+1]+u1[cp-Nx]+u1[cp+Nx]
                 +u1[cp-area]+u1[cp+area]) - u[cp];
      }
   }
}
```

4.5 2D Slicing with shared memory

As with the 3D tiling methods, the shared memory implementation has to account for the pickup of neighbouring data at the edges of the thread blocks. The same approaches are used. Firstly a method that uses a block size shared memory array with conditional statements around the summing point. Secondly, using an extended size shared memory array, with conditional statements used at the loading point of the code. The kernel code for these methods is shown in figures 7 and 8.

4.6 Cache optimisations

Aside from the design of the kernel code itself and experimenting with the size of the thread block, further speedups can be obtained by optimising the cache usage. On the compute 2.x Fermi cards, using the *cudaFuncSetCacheConfig( )* command to prefer the L1 cache produces efficiency gains for all the above kernels, some by as much as 15%.

An additional feature of the Kepler architecture is the ability to use a read-only data cache which is separate from the standard L1 and L2 cache [5]. The above kernels can make use of this when accessing data from the six neighbour points from one time step ago. In the parameters of the kernel declaration, the data pointer is declared as:

```
const double * __restrict__ u1
```

This functions correctly even though the data pointers are swapped around at each iteration in time. Unlike the L1 cache configuration, this feature does not always provide efficiency gains. Some kernels, such as the first 3D tiling method, benefited from using this cache, whilst others did not.

5. TESTING PROCEDURE

A standard test simulation was used to benchmark both the latest Tesla K20 card, as well as the previous generation Tesla C2050. Each of the six kernel methods was tested on both systems. The simulation models a 3.4 x 4.0 x 2.8 = 38m^3 space. At a sample rate of 44.1kHz this requires data grids of size: 256 x 296 x 212 = 16,064,512 points.

A raised cosine impulse was used as the input to the model, injected as a soft source at a given grid point. The simulation was computed for 44,100 samples, and using double precision floating-point arithmetic. All codes were compiled using CUDA version 5.0, and for compute architectures 2.0 or 3.5 as appropriate for the card. For reference, table 1 shows the key features of both the K20 and C2050 graphics cards.

Description	C2050	K20
Compute capability	2.0	3.5
CUDA cores	448	2,496
Clock speed	1.15 GHz	706 MHz
Memory bandwidth	144 GB/sec	208 GB/sec
Peak double precision	515 Gflops	1.17 Tflops

Table 1. GPU card specifications.

The difference in the hardware architectures is clear, as the Kepler card has five times as many core processors as the Fermi card, but running at a lower clock rate. Whilst the peak double precision performance is twice as high, the memory bandwidth is only 44% greater.

6. RESULTS

Tables 2 and 3 show the resulting computation times for each of the six kernels methods, firstly for the C2050 card, and then the K20. The timing points used were defined directly before the main time iteration loop, and directly after the loop, having performed a *cudaThreadSynchronize( )*. The ratio figure is the percentage time relative to the 3D tiling base case.

Kernel Method	Time (s)	Ratio
3D Tiling	227.1	-
3D Tiling shared	340.8	150.1%
3D Tiling ext shared	272.7	120.1%
2D Slicing	300.8	132.5%
2D Slicing shared	334.3	147.2%
2D Slicing ext shared	227.6	100.2%

Table 2. Computation times for C2050 Fermi card.

Kernel Method	Time (s)	Ratio
3D Tiling	156.9	-
3D Tiling shared	218.7	139.4%
3D Tiling ext shared	198.1	126.3%
2D Slicing	183.3	116.8%
2D Slicing shared	164.6	104.9%
2D Slicing ext shared	150.8	96.1%

Table 3. Computation times for K20 Kepler card.

Table 4 shows a comparison of the timing data for each card, and the relative speedup achieved by the K20 card over the C2050 card.

Method	C2050(s)	K20(s)	Speedup
3D Tiling	227.1	156.9	x1.45
3D Tiling shared	340.8	218.7	x1.56
3D Tiling ext shared	272.7	198.1	x1.38
2D Slicing	300.8	183.3	x1.64
2D Slicing shared	334.3	164.6	x2.03
2D Slicing ext shared	227.6	150.8	x1.51

Table 4. Computation times (seconds) and speedup for K20 card compared to C2050 card.

Starting with the C2050 Fermi card data, the most notable result is that the basic 3D tiling kernel is just as efficient as the best shared memory method, the extended 2D slicing. Despite the major deduction in data reading from global memory, no performance benefit is seen. Indeed, all of the other shared memory kernels delivered worse results, some by up to 50%. The extended shared memory approach outperforms the standard shared memory version in both the 3D and 2D cases.

For the K20 Kepler card, the basic 3D tiling kernel rates second in terms of efficiency, but is only slightly behind the extended 2D slicing by some 4%. The method of using shared memory in the 2D case shows only minor variation. Comparing the two cards, the headline result is a x1.5 speedup for the fastest method running on the K20 card, with a computation time of 150.8 seconds.

This also improves on previously reported test data [7], in which a headline time of 184 seconds was shown using a version of a 2D slicing kernel with shared memory, and running an identical simulation but using the Nvidia GeForce GTX 480 card. Whilst the GeForce cards are designed for the gaming market and do not have same level of double precision support, they do have comparable, or in some cases better, memory bandwidth. For example, the GTX 480 has a bandwidth of 177.4 GB/sec which is greater than the C2050 Fermi card.

7. LARGE-SCALE ROOM MODELS

The K20 graphics card has 5Gb of global memory available, allowing large-scale room models to be simulated. The model detailed above can be extended from a grid size of 16 million up to 310 million points in each of the two grids. At 44.1kHz this gives a volume of 756m^3, and a computation time of 47 minutes per second of output at double precision.

The inclusion of useable boundary conditions, and effects such as viscosity, increase the computation time. The latter effect also requires the use of three data grids rather than the two used here, and so reduces the maximum available volume to around 500m^3. The use of single precision floating-point arithmetic doubles the maximum simulation space, or produces efficiency gains, although this can lead to stability issues for the boundary conditions when running at the Courant limit.

8. CONCLUSIONS

Six different approaches to the kernel design for the basic 3D FDTD scheme were optimised and benchmarked on the Tesla K20 card. What was once considered a 'naive' approach of simply reading directly from global memory now produces efficient kernels, both in the case of the Fermi and Kepler architectures. Making use of cache optimisations allows these basic codes to perform as well as the more complex shared memory versions.

Ultimately, these forms of finite difference schemes are always memory bandwidth limited. The ability to compute double precision floating-point arithmetic, and the amount of parallelisation, is secondary to the speed with which data can be moved around. Further comparisons can be made by benchmarking the latest GeForce Kepler cards, such as the GTX Titan, which has even greater memory bandwidth than the K20 tested here. The simultaneous use of multiple GPU cards is an effective approach to achieving scalable efficiency gains.

Acknowledgments

This work is supported by the European Research Council under Grant StG-2011-279068-NESS.

9. REFERENCES

[1] L. Savioja, "Real-time 3D finite-difference time-domain simulations of low and mid-frequency room acoustics," in *Proceedings of 13th Int. Conf on Digital Audio Effects*. Austria, Sept, 2010.

[2] C. Webb and S. Bilbao, "Computing room acoustics with CUDA - 3D FDTD schemes with boundary losses and viscosity," in *Proc. of the IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, Prague, Czech Republic, May 2011.

[3] L. Savioja, D. Manocha, and M. Lin, "Use of GPUs in room acoustic modeling and auralization," in *Proc. Int. Symposium on Room Acoustics*, Melbourne, Australia, Aug 2010.

[4] J. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*. Pacific Grove, California: Wadsworth and Brooks/Cole Advanced Books and Software, 1989.

[5] Nvidia, "Cuda C programming guide," *CUDA toolkit documentation.[Online][Cited: 24th Mar 2013.] http://docs.nvidia.com/cuda/*, 2012.

[6] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, New York, NY, USA, 2009, pp. 79–84.

[7] J. Lopez, D. Carnicero, N. Ferrando, and J. Escolano, "Parallelization of the finite-difference time-domain method for room acoustics modelling based on CUDA," *Mathematical and Computer Modelling*, vol. 57, no. 78, pp. 1822 – 1831, 2012.

Figure 7. 2D Slicing with shared memory kernel.

```
__global__ void UpDate(double *u,double *u1,double L2)
{
   __shared__ double uS1[BxL][ByL];

   int tdx = threadIdx.x;
   int tdy = threadIdx.y;

   // Get 3D position
   int X = blockIdx.x * BxL + tdx;
   int Y = blockIdx.y * ByL + tdy;
   int Z,cp;

   // Initial variables
   double u1cpm = 0.0;
   double u1cp  = u1[area+(Y*Nx+X)];
   double u1cpp,sum;

   for(Z=1;Z<(Nz-1);Z++){

      // Get linear position
      cp = Z*area+(Y*Nx+X);
      u1cpp  = u1[cp+area];
      // load shared
      uS1[tdx][tdy] = u1cp;
      __syncthreads();

      if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1)) ){

        sum = 0.0;
        if(tdx==0) sum+= u1[cp-1];
        else sum+= uS1[tdx-1][tdy];

        if(tdx==BxL-1) sum+= u1[cp+1];
        else sum+= uS1[tdx+1][tdy];

        if(tdy==0) sum+= u1[cp-Nx];
        else sum+= uS1[tdx][tdy-1];

        if(tdy==ByL-1) sum+= u1[cp+Nx];
        else sum+= uS1[tdx][tdy+1];

        u[cp] = L2*(sum+u1cpm+u1cpp) - u[cp];

        u1cpm = u1cp;
        u1cp  = u1cpp;
        __syncthreads();
      }
   }
}
```

Figure 8. 2D Slicing with extended shared memory kernel.

```
__global__ void UpDate(double *u,double *u1,double L2)
{
   __shared__ double uS1[BxL+2][ByL+2];

   int tdx = threadIdx.x;
   int tdy = threadIdx.y;

   int X = blockIdx.x * BxL + tdx;
   int Y = blockIdx.y * ByL + tdy;

   int Z,cp;
   double u1cpm = 0.0;
   double u1cp  = u1[area+(Y*Nx+X)];
   double u1cpp;
   tdx++; tdy++;

   for(Z=1;Z<(Nz-1);Z++){

      cp = Z*area+(Y*Nx+X);
      u1cpp  = u1[cp+area];
      // load shared
      uS1[tdx][tdy] = u1cp;

      if ( (tdy==1) && !(Y==0) ){
        uS1[tdx][tdy-1] = u1[cp-Nx];
      }
      if ( (tdy==ByL) && !(Y==(Ny-1)) ){
        uS1[tdx][tdy+1] = u1[cp+Nx];
      }
      if ( (tdx==1) && !(X==0) ){
        uS1[tdx-1][tdy] = u1[cp-1];
      }
      if ( (tdx==BxL) && !(X==(Nx-1)) ){
        uS1[tdx+1][tdy] = u1[cp+1];
      }
      __syncthreads();

      if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1)) ){

        u[cp] = L2*(uS1[tdx-1][tdy]+uS1[tdx+1][tdy]
                   +uS1[tdx][tdy-1]+uS1[tdx][tdy+1]
                   +u1cpm+u1cpp) - u[cp];

        u1cpm = u1cp;
        u1cp  = u1cpp;
        __syncthreads();
      }
   }
}
```