CHAPTER 16

Graphics processing unit acceleration of finite-difference time-domain method

Recent developments in the design of graphics processing units (GPUs) have been occurring at a much greater pace than with central processor units (CPUs) and very powerful processing units have been designed solely for the processing of computer graphics. For instance, the Tesla® K40 GPU includes 2,880 cores and 12 GB memory, and it can reach to 4.29 Tflops peak single precision floating point performance. Due to this potential in faster computations, the GPUs have received the attention of the scientific computing community. Initially these cards were designed for computer graphics and floating precision arithmetic has been sufficient for such applications. Due to the demand of higher precision arithmetic from the scientific community, the vendors have started to develop graphics cards that support double precision arithmetic as well by introducing a new generation of graphical computation cards.

The computational electromagnetics community as well has started to utilize the computational power of graphics cards, and in particular, several implementations of finite-difference time-domain (FDTD) [79–89] method have been reported in the literature. For instance, Brook is used as the programming language in [79–85], High Level Shader Language (HLSL) is reported in [86], while the FDTD implementations in [87–89] are based on OpenGL. Relatively recently, the introduction of the Compute Unified Device Architecture (CUDA) [90] development environment from NVIDIA made GPU computing much easier. CUDA is a general purpose parallel computing architecture. To program the CUDA architecture, developers can use C, which can then be run at great performance on a CUDA-enabled processor [91]. CUDA has been reported as the programming environment for implementation of FDTD in several publications, while [92] and [93] illustrate methods to improve the efficiency of FDTD using CUDA, which can be used as guidelines while programming FDTD using CUDA.

It should be noted that OpenCL [94] has been introduced recently as a programming platform to develop codes on parallel devices. It enables programming on parallel architectures including GPU. It has been used to develop FDTD implementations as well [95]. Although OpenCL provides an alternative platform to CUDA, there is a large community that uses CUDA and the technical support for CUDA is well established, therefore, CUDA will keep its position as the prominent GPU programming platform for the near future.

16.1 GPU programming using CUDA

CUDA, as other languages or programming platforms, has its advantages and disadvantages. As the major advantages: CUDA is easier to learn compared with other alternatives and NVIDIA provides extensive support to developers and users. One major disadvantage is that CUDA can run only on CUDA-enabled NVIDIA cards. While OpenCL and DirectCompute are becoming the new alternatives to CUDA as programming languages on modern GPU-based architectures, CUDA may keep its popularity for scientific computing due to vast learning resources available for developers. In this chapter we will illustrate an implementation of a two-dimensional FDTD program using CUDA.

In order to start programming with CUDA, one need to install CUDA drivers, CUDA Toolkit that would include C/C++ compiler, Visual Profiler, and GPU-accelerated BLAS, FFT, Sparse Matrix, and RNG libraries. Another crucial component is GPU Computing Software Development Kit (SDK), which includes several tools and code samples. All these components are available at NVIDIA's web portal for Windows, Linux, and Mac OS X operating systems.

In order to program using CUDA, one needs to know the CUDA terminology and architecture. NVIDIA CUDA Programming Guide [96], available at NVIDIA's web portal, is a good resource to learn CUDA. In this section, brief descriptions of some concepts in CUDA are summarized from the Programming Guide in order to prepare the reader for the discussions that follow.

16.1.1 Host and device

CUDA provides an extended version of C language for programming. Thus it is relatively easy for a C programmer to develop codes for CUDA. In CUDA's programming model the part of the computer architecture which includes CPU is referred to as "host", while the GPU based card is referred to as "device". A CUDA program would consist of parts that would run on the host and the device: the main C program would run on the host, while the code including parallel computations would execute on the device. Basically, the "device" is a physically separate device on which the CUDA threads execute. These threads can be launched by the program running on the host.

Both the host and the device maintain their own DRAM, referred to as host memory and device memory, respectively. Generally, for data parallel computations, data is transferred from the host memory to the device memory, computations are performed on the device, and results are transferred back to host memory. For instance, Listing 16.1 shows a small program which adds two vectors. First, vectors A, B, and C are allocated on host (CPU) memory, and A and B are initialized with some numbers. Then corresponding vectors are allocated on device (GPU) memory. Vectors A and B are copied to device memory. Then a function is executed on the device which adds these vectors and stores the results in a vector on the device memory. The resulting vector is then copied back to vector C on the host memory to provide the expected results.

Listing 16.1 Addition of two vectors in CUOA.

```
#include <stdlib.h>
#include <stdio.h>

// Kernel definition
__global__ void AddVectors(int* A, int* B, int* C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}

int main()
{
        int N = 256;
        int mem_size = N*sizeof(int);

        // Allocate memory for 1D arrays on host memory
        int* A = (int*) malloc(mem_size);
        int* B = (int*) malloc(mem_size);
        int* C = (int*) malloc(mem_size);

        // initialize arrays
        for (int i=0; i<N; i++) A[i] = 2*i;
        for (int i=0; i<N; i++) B[i] = 3*i;

        // allocate device memory
   int* d_A;
   int* d_B;
   int* d_C;
        cudaMalloc( (void**) &d_A, mem_size);
        cudaMalloc( (void**) &d_B, mem_size);
        cudaMalloc( (void**) &d_C, mem_size);

   // copy host memory to device
        cudaMemcpy( d_A, A, mem_size, cudaMemcpyHostToDevice);
        cudaMemcpy( d_B, B, mem_size, cudaMemcpyHostToDevice);

        // Add vectors on device
        AddVectors<<<1, N>>>(d_A, d_B, d_C);

   // copy result from device to host
   cudaMemcpy( C, d_C, mem_size, cudaMemcpyDeviceToHost);

        // dsiplay results
        for (int i=0;i<N;i++) printf("%d ",C[i]);

   // cleanup memory
   free(A);     free(B);        free(C);
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
}
```

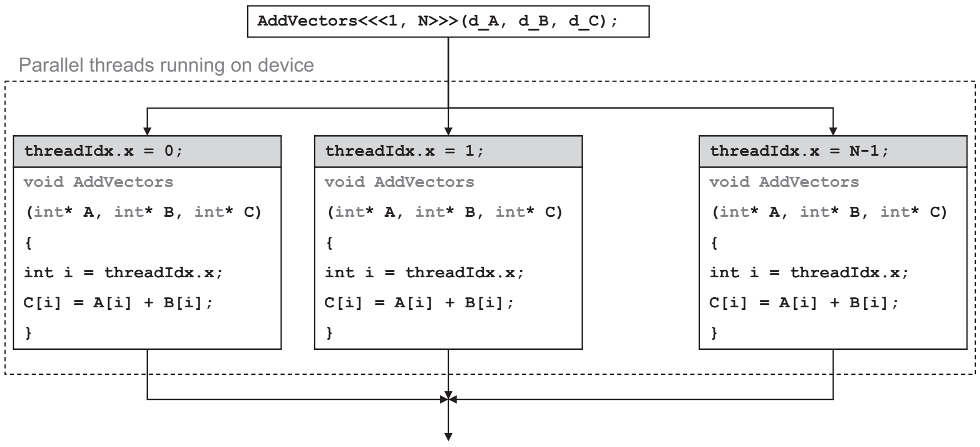

Figure 16.1 Parallel threads executing the same code on different set of data identified by the threadIdx.x value.

16.1.1.1 Kernels

A C function defined to run on the device is called a kernel. A kernel is defined using the __global__ declaration specifier as shown in Listing 16.1. A kernel, when called, as illustrated in Figure 16.1, executes *N* times in parallel by *N* different CUDA threads, as opposed to only once like regular C functions. Each CUDA thread executes the same kernel code, but for a different section of data accessed by an index identifying the thread. The number of CUDA threads for each call is specified using <<<...>>> syntax. In the example code, <<<1, *N*>>> indicates that *N* threads are executed in parallel. In each thread a pair-wise addition of elements in arrays A and B are performed. Here each thread has a unique thread ID, which is an integer number between 0 and $N-1$ in the given example. The thread ID is accessible within the kernel through the built-in threadIdx variable. A thread uses its specific thread ID, and uses it to access the array elements with the same index as the thread ID, and perform calculations on them. If the number of threads is the same as the number of array elements, then the entire array is processed through one-to-one mapping between array elements and the threads.

16.1.2 Thread hierarchy

The thread ID threadIdx can be used to map threads to data elements in arrays that these threads process. This mapping enables data parallel computations, i.e., running the same code for different sets of data. The variable threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector (one-dimensional array), matrix (two-dimensional array), or field (three-dimensional array). As an example, the following code in Listing 16.2 adds two three-dimensional arrays A and B of size $M \times N \times P$ and stores the result into array C. Here there is a one-to-one mapping between array elements and the threads in a three-dimensional thread block.

Listing 16.2 Addition of three dimensional arrays.

```
__global__ void AddArrays3D (int A[M][N][P], int B[M][N][P], int
C[M][N][P])
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        int k = threadIdx.z;
        C[i][j][k] = A[i][j][k] + B[i][j][k];
}
int main()
{
...
        int M = 8;
        int N = 8;
        int P = 4;
        dim3 block_size(M, N, P);
        AddArrays3D<<<1, block_size>>>(A, B, C);
...
}
```

In CUDA, a number of threads form a thread block. For instance, in this example code, a thread block of M × N × P threads is defined by the statement "`dim3 block_size (M, N, P);`". However, there is a limit for the number of threads that can form a block: in NVIDIA graphics cards with compute capability 1.x, the maximum number of threads in a block is 512, while it is 1024 in cards with compute capability 2.x. In order to map larger number of threads, one can define a grid which consists of a number of equally-shaped thread blocks, so that the total number of threads in the grid is equal to the number of threads per block times the number of blocks. The grid of thread blocks can be one-, two-, or three-dimensional. For instance, Figure 16.2 illustrates a two-dimensional grid of 2×2 blocks, where each block is a two-dimensional array of 2×2 threads. The given grid thus includes 16 threads.

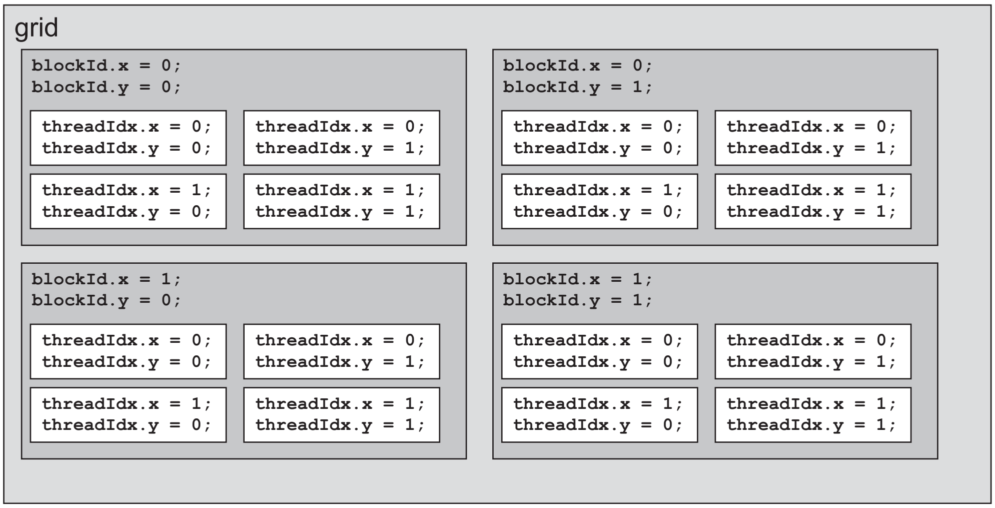
grid
blockId.x = 0; blockId.y = 0;
threadIdx.x = 0; threadIdx.y = 0;
threadIdx.x = 0; threadIdx.y = 1;
threadIdx.x = 1; threadIdx.y = 0;
threadIdx.x = 1; threadIdx.y = 1;
blockId.x = 0; blockId.y = 1;
threadIdx.x = 0; threadIdx.y = 0;
threadIdx.x = 0; threadIdx.y = 1;
threadIdx.x = 1; threadIdx.y = 0;
threadIdx.x = 1; threadIdx.y = 1;
blockId.x = 1; blockId.y = 0;
threadIdx.x = 0; threadIdx.y = 0;
threadIdx.x = 0; threadIdx.y = 1;
threadIdx.x = 1; threadIdx.y = 0;
threadIdx.x = 1; threadIdx.y = 1;
blockId.x = 1; blockId.y = 1;
threadIdx.x = 0; threadIdx.y = 0;
threadIdx.x = 0; threadIdx.y = 1;
threadIdx.x = 1; threadIdx.y = 0;
threadIdx.x = 1; threadIdx.y = 1;

Figure 16.2 Grid of thread blocks.

Listing 16.3 Addition of two dimensional arrays.

```
__global__ void AddArrays2D(int A[M][N], int B[M][N], int C[M][N])
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
C[i][j] = A[i][j] + B[i][j];
}
int main()
{
...
        dim3 grid_size(3, 4);
        dim3 block_size(16,32);
        AddArrays2D<<< grid_size, block_size>>>(A, B, C);
...
}
```

The dimension of the grid is specified by the first parameter of the <<<...>>> syntax. Listing 16.3 shows a code that performs pair-wise addition of two-dimensional arrays. In this example code below, a 3×4 grid is defined, where each thread block in the grid includes 16×32 threads. Thus the threads can be easily mapped to two-dimensional 48×128 size arrays. Each block within the grid can be identified by a one-dimensional or two-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable. It is straightforward to map threads to data elements in arrays using the blockDim, blockIdx, and threadIdx internal variables as shown in the kernel of Listing 16.3.

It should be noted that thread blocks execute independent from each other; therefore, one should consider this independence while developing an algorithm in CUDA. The number of thread blocks in a grid is generally determined by the size of the data being processed.

For instance, if an array of size 256×128 is to be processed by thread blocks of 16×16, one can use a grid of size 16×8.

16.1.3 Memory hierarchy

CUDA threads may access data from multiple memory spaces on the GPU card.

Global memory:

1. The main memory space on the device is called global memory.
2. All threads can access the global memory to read and write data.
3. The read and write operations to the global memory are long latency, thus should be minimized.

Shared memory:

1. This is a memory space that is available to a block throughout the lifetime as the block.
2. All threads in the block can access the shared memory of the block.
3. Shared memory is a cached memory space; it is computationally much more efficient to access the shared memory.
4. It is better to copy data from global memory to shared memory and process the data while it resides on the shared memory if the data are reused.

Two other memory spaces are called “constant memory” and “texture memory”, which are read-only memory spaces on the device. For instance, if threads will access some data only in read-only mode, the data can be copied to the “constant memory” before the execution of the threads, and can be accessed by the threads during execution. Moreover, each thread has a private local memory, which is a set of local 32-bit registers. It is important to use these memory spaces effectively according to the capabilities they provide in order to achieve an algorithm with high computational efficiency in CUDA.

16.1.4 Performance optimization in CUDA

One should be familiar with the CUDA architecture to some extent in order to develop a program with optimum performance. CUDA Best Practices Guide, available at NVIDIA’s web portal, is a good reference for programmers; it provides recommendations for optimization and a list of best practices for programming with CUDA. While not all of these recommendations are applicable to the case of FDTD programming; the following list of recommendations is taken into consideration to develop an FDTD implementation discussed in the next section:

(R1) structure the algorithm in a way that exposes as much data parallelism as possible. Once the parallelism of the algorithm has been exposed, it needs to be mapped to the hardware as efficiently as possible.
(R2) ensure global memory accesses are coalesced whenever possible.
(R3) minimize the use of global memory. Prefer shared memory access where possible.
(R4) use shared memory to avoid redundant transfers from global memory.
(R5) hide latency arising from register dependencies, maintain at least 25 percent occupancy on devices with CUDA compute capability 1.1 and lower, and 18.75 percent occupancy on later devices.
(R6) use a multiple of 32 threads for the number of threads per block as this provides optimal computing efficiency and facilitates coalescing.

16.1.5 Achieving parallelism

At every iteration of the FDTD loop, new values of three magnetic field components are calculated at every cell simultaneously using the past values of electric field components. After magnetic field updates are completed, values of three electric field components are updated at every cell simultaneously in a separate function using the past values of magnetic field components. Since the calculations for each cell can be performed independent from the other cells, a CUDA algorithm can be developed by assigning each cell calculation to a separate thread, and the highest level of parallelism can be achieved to satisfy the recommendation R1. Therefore, the data parallelism required to benefit from CUDA is inherently satisfied by FDTD.

16.2 CUDA implementation of two-dimensional FDTD

In this section we will present an implementation of a two-dimensional FDTD program in CUDA. The program consists of two parts: the first is a MATLAB® code that is based on the two-dimensional FDTD code that has been illustrated in Chapters 7 and 8 while discussing absorbing boundary conditions, and the second is a C code using CUDA that is developed to run FDTD time-marching loop calculations on a graphics card. These codes are available on the CD accompanying this book in folders "fdtd_2d_with_cuda" and "fdtd_2d_cuda_code", respectively.

The two-dimensional FDTD code that has been presented in Chapter 8 is modified such that problem definitions, initializations, and results postprocessing is done in MATLAB as usual, while the program launches an executable that runs the time-marching loop on a graphics card. The executable, "fdtd2d.exe", is implemented in C and it is compiled and built using CUDA's nvcc compiler. The C code can perform FDTD time-marching loop in one of two modes: either on CPU or on GPU. A parameter, named as **computation_platform**, is added to ***define_problem_space_parameters_2d*** to set the computation platform as shown in Listing 16.4. If this parameter is set as 1, the program will run as usual in MATLAB. If the parameter is set as 2 or 3, then the "fdtd2d.exe" will be launched. If the parameter is 2, the executable will perform FDTD computations on CPU, otherwise on GPU. Listing 16.5 illustrates the modified code in the subroutine ***run_fdtd_time_marching_loop_2d***.

Listing 16.6 shows the subroutine ***launch_executable***. Since the parameters and arrays that FDTD calculations require are initialized in MATLAB, they need to be transferred to the executable for use. This transfer is performed through a binary file defined as "`exe_data_file_name`" in Listing 16.6: All required arrays and parameters are adjusted and written to this data file, the data file is read by "fdtd2d.exe", and FDTD calculations are performed either on CPU or on GPU. The intermediate data file is accessed in binary format, instead of ASCII format, to transfer data as is without loss of precision between the MATLAB and the C programs, and binary files require less space for storage. Once the

Listing 16.4 define_problem_space_parameters_2d

```
% the platform on which the calculations will be performed
% 1: matlab on cpu, 2: C executable on cpu, 3: C executable on gpu
computation_platform = 3;
```

Listing 16.5 run_fdtd_time_marching_loop_2d

```
if computation_platform ~=1
    launch_executable;
else
    for time_step = 1:number_of_time_steps
        update_magnetic_fields_2d;
        update_impressed_M;
        update_magnetic_fields_for_CPML_2d;
        capture_sampled_magnetic_fields_2d;
        update_electric_fields_2d;
        update_impressed_J;
        update_electric_fields_for_CPML_2d;
        capture_sampled_electric_fields_2d;
        display_sampled_parameters_2d;
    end
end
```

Listing 16.6 launch_executable

```
% save the data to a file to process on gpu
exe_data_file_name = 'exe_data_file.dat';
save_project_data_to_file;

cmd = ['fdtd2d_on_gpu.exe ' exe_data_file_name];
status = system(cmd);
if status
   disp('Calculations are not successful!');
   return;
end

time_step = number_of_time_steps;

result_data_file_name = ['result_' exe_data_file_name];
fid = fopen(result_data_file_name, 'r');

for ind = 1:number_of_sampled_electric_fields
   sampled_electric_fields(ind).sampled_value = ...
      fread(fid, number_of_time_steps, 'float32').';
end

for ind = 1:number_of_sampled_magnetic_fields
   sampled_magnetic_fields(ind).sampled_value = ...
      fread(fid, number_of_time_steps, 'float32').';
end

fclose(fid);
```

calculations are completed by “fdtd2d.exe” results are written to an output data file, which then is read by MATLAB for postprocessing and displaying the results. In Listing 16.6, MATLAB ***system*** command is used to launch the executable.

16.2.1 Coalesced global memory access

In CUDA, memory instructions are the instructions that read from or write to shared, constant, or global memory. When accessing global memory, there are 400 to 600 clock cycles of memory latency. Much of this global memory latency can be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete [96]. Unfortunately when FDTD calculations are performed on a computer the operations are dominated by memory accesses rather than arithmetic instructions. Hence, the memory access inefficiency is the main bottle neck that reduces efficiency of FDTD on GPU. Global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be coalesced into a single memory transaction of 32, 64, or 128 bytes [96].

If the size of two-dimensional arrays, thus the size of the FDTD domain in number of cells, in the x and y directions, is a multiple of 16, then the coalesced memory access is ensured. In general an FDTD domain size would be an arbitrary number. In order to achieve

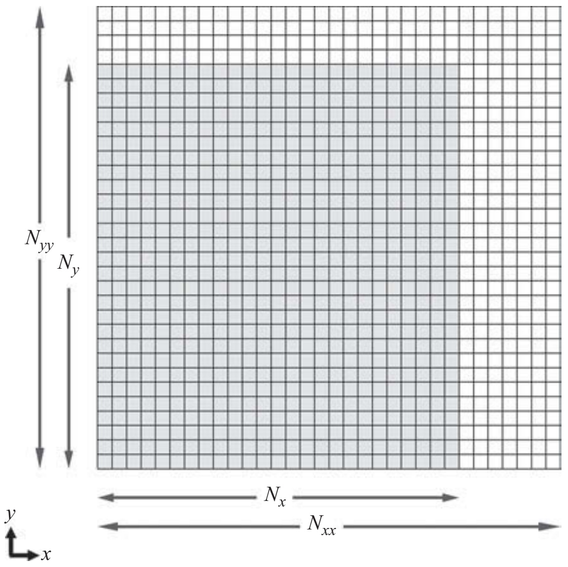

Figure 16.3 An extended two-dimensional computational domain.

coalesced memory access, the FDTD domain can be extended by padded cells such that the number of cells in x and y directions is an integer multiple of 16 as illustrated in Figure 16.3. Although, padding these cells increases the amount of memory needed to store arrays, it improves the efficiency of the kernel functions tremendously. Thus the recommendation R2 is satisfied.

It should be noted that, these padded cells are beyond the boundaries of the original domain and they should be electrically isolated from the original domain. As long as the original domain boundaries are kept as PEC, such as in the case of CPML boundaries, the fields in the original domain will be isolated from the padded cells, thus padded cells can be assumed to be filled with any type of material. The easiest way is to set these padded cells as free space.

If the size of a two-dimensional FDTD domain is $Nx \times Ny$ cells, where Nx and Ny are the numbers of cells in x and y directions, respectively, then the modified size of the FDTD domain becomes $Nxx \times Nyy$, where Nxx and Nyy are the new numbers of cells. In the presented example code, the extended size of the problem space is determined in the subroutine ***save_project_data_to_file*** as

```
% extend the domain and adjust number of cells for gpu
nxx = (floor(nx/16)+1)*16;
nyy = (floor(ny/16)+1)*16;
```

Then the two-dimensional arrays of fields and coefficients can be extended to $Nxx \times Nyy$. However, in fact, the two-dimensional arrays are extended to $Nxx \times (Nyy + 2)$ in the subroutine ***save_project_data_to_file*** before they are copied to the binary data file. The reason for the additional padded cells in the y direction will be explained in the subsequent sections.

16.2.2 Thread to cell mapping

The “fdtd2d_on_gpu.exe” program starts with reading the binary data file; first arrays are allocated on the CPU memory for two-dimensional arrays, then the coefficient and field data are read into these arrays. These arrays initially reside on the host (CPU) memory and they need to be copied to device (GPU) global memory. The `cudaMalloc()` function of CUDA is used to allocate memory on the device global memory for these arrays and `cudaMemcpy()` function is used to copy the data to the global memory. Once these arrays are ready on the global memory, they are ready for processing on the graphics card by kernel functions.

Listing 16.7 shows a function that performs an iteration of the time-marching loop on GPU. Similarly, Listing 16.8 shows a function that performs an iteration on CPU. It can be seen that main components of an iteration, such as electric and magnetic field updates, source updates, boundary condition updates, etc., are performed by calls to associated functions. We will discuss the electric and magnetic field updating functions in detail for the TMz case to illustrate an implementation using CUDA.

Listing 16.7 FDTD iteration on GPU

```
bool fdtdIterationOnGpu()
{
        int n_bx = (nxx/TILE_SIZE) + (nxx%TILE_SIZE == 0 ? 0 : 1);
        int n_by = (nyy/TILE_SIZE) + (nyy%TILE_SIZE == 0 ? 0 : 1);
        dim3 threads = dim3(TILE_SIZE, TILE_SIZE, 1);
        dim3 grid = dim3(n_bx, n_by, 1);

        dim3 threads_sef = dim3(number_of_sampled_electric_fields, 1, 1);
        dim3 threads_smf = dim3(number_of_sampled_magnetic_fields, 1, 1);
        dim3 grid_sef = dim3(1, 1, 1);
        dim3 grid_smf = dim3(1, 1, 1);

        if (is_TEz)
             update_magnetic_fields_on_gpu_TEz<<<grid,
threads>>>(dvChzh, dvChzex, dvChzey, dvHz, dvEx, dvEy, nxx);

        if (is_TMz)
             update_magnetic_fields_on_gpu_TMz<<<grid,
threads>>>(dvChxh, dvChxez, dvChyh, dvChyez, dvHx,  dvHy, dvEz, nxx);

        update_impressed_magnetic_currents_on_gpu(time_step);

        update_magnetic_fields_for_CPML_on_gpu();

        capture_sampled_magnetic_fields_on_gpu<<<grid_smf, threads_smf>>>
                (dvHx, dvHy, dvHz, dvsampled_magnetic_fields_component,
dvsampled_magnetic_fields_is,
                 dvsampled_magnetic_fields_js,
dvsampled_magnetic_fields_sampled_value, time_step,
number_of_time_steps, nxx);

        if (is_TEz)
        update_electric_fields_on_gpu_TEz<<<grid, threads>>>
(dvCexe, dvCexhz, dvCeye, dvCeyhz,  dvEx, dvEy, dvHz, nxx);

        if (is_TMz)
        update_electric_fields_on_gpu_TMz<<<grid, threads>>>
(dvCeze, dvCezhy, dvCezhx, dvHx,  dvHy, dvEz, nxx);
```

```
        update_impressed_electric_currents_on_gpu(time_step);

        update_electric_fields_for_CPML_on_gpu();

        capture_sampled_electric_fields_on_gpu<<<grid_sef, threads_sef>>>
                (dvEz, dvEy, dvEz, dvsampled_electric_fields_component,
dvsampled_electric_fields_is,
             dvsampled_electric_fields_js,
dvsampled_electric_fields_sampled_value, time_step,
number_of_time_steps, nxx);

        printf("timestep: %d \n", time_step);
        time_step++;

        return true;
}
```

Listing 16.8 FDTD iteration on CPU

```
bool fdtdIterationOnCpu()
{
        if (is_TEz) update_magnetic_fields_on_cpu_TEz();

        if (is_TMz) update_magnetic_fields_on_cpu_TMz();

        update_impressed_magnetic_currents_on_cpu(time_step);

        update_magnetic_fields_for_CPML_on_cpu();

        capture_sampled_magnetic_fields_on_cpu(time_step);

        if (is_TEz)     update_electric_fields_on_cpu_TEz();

        if (is_TMz)     update_electric_fields_on_cpu_TMz();

        update_impressed_electric_currents_on_cpu(time_step);

        update_electric_fields_for_CPML_on_cpu();

        capture_sampled_electric_fields_on_cpu(time_step);

        printf("timestep: %d \n", time_step);
        time_step++;

        return true;
}
```

The kernel function that updates the magnetic field components is called

```
update_magnetic_fields_on_gpu_TMz<<<grid, threads>>>
(dvChxh, dvChxez, dvChyh, dvChyez, dvHx, dvHy, dvEz, nxx);
```

Here the parameter names preceded with “dv” are pointers to respective arrays on the global memory of the graphics card. The CUDA notation `<<<grid, threads>>>`

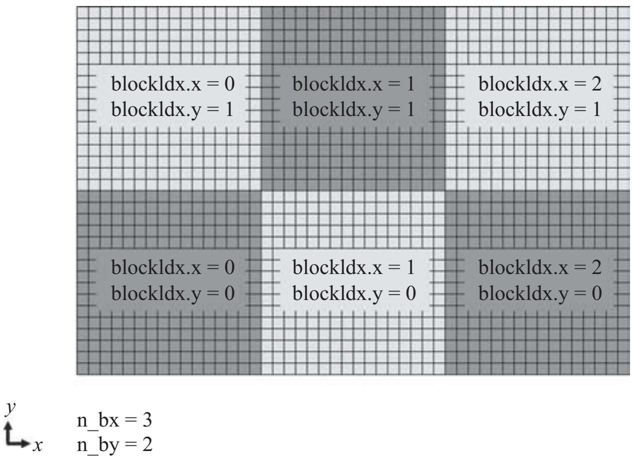

Figure 16.4 A grid of thread blocks that spans a 48×32 cells two-dimensional problem space.

describes the grid of threads that will be launched for this kernel. The parameter `threads` is initialized as

```
dim3 threads = dim3(TILE_SIZE, TILE_SIZE, 1);
```

which means that a thread block consists of `TILE_SIZExTILE_SIZE` square shaped array of threads. In this implementation the `TILE_SIZE` is set as 16, thus the recommendation R6 in Section 16.1 is satisfied. The parameter `grid` is initialized as

```
int n_bx = (nxx/TILE_SIZE) + (nxx%TILE_SIZE == 0 ? 0 : 1);
int n_by = (nyy/TILE_SIZE) + (nyy%TILE_SIZE == 0 ? 0 : 1);
dim3 grid = dim3(n_bx, n_by, 1);
```

which means that the grid is composed of *n_bx* × *n_by* thread blocks. This thread grid scheme helps to conveniently map threads to cells. Figure 16.4 illustrates such a grid that spans a problem space.

Listing 16.9 shows the kernel function that updates the magnetic field components for the TMz case, whereas Listing 16.10 shows the kernel that updates the electric field components. Respective functions that perform magnetic and electric field updates on CPU are shown in Listings 16.11 and 16.12 for comparison. In this implementation for GPU each cell is processed by an associated thread. The CUDA internal variables `threadIdx` and `blockIdx` are used to identify the threads and data elements which they will update. The variable `blockIdx` is used to identify the thread blocks as illsutrated in Figure 16.4. Similarly, the variable `threadIdx` is used to identify threads within a thread block as shown in Figure 16.5. Thus, one can identify any thread, and also cell, in a two-dimensional array with indices *i* and *j* such that

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

At this point it should be noted that although the arrays that are being processed are two-dimensional, they are stored in device (GPU) global memory as one-dimensional arrays and elements of these arrays are accessed in kernel functions in a linear fashion. Thus a conversion from *i* and *j* indices to the linear index, denoted as "ci" in the Listing 16.4, is required.

Listing 16.9 update_magnetic_fields_on_gpu_TMz

```
__global__ void
update_magnetic_fields_on_gpu_TMz(float* Chxh, float* Chxez, float*
Chyh, float* Chyez, float* Hx,  float* Hy, float* Ez, int nxx)
{
        __shared__ float sEz[TILE_SIZE][2*TILE_SIZE+1];

        int tx = threadIdx.x;
        int ty = threadIdx.y;
        int i = blockIdx.x * blockDim.x + tx;
        int j = blockIdx.y * blockDim.y + ty;

        int ci = (j+1)*nxx+i;

        sEz[ty][tx] = Ez[ci];
        sEz[ty][tx+TILE_SIZE] = Ez[ci+TILE_SIZE];

        __syncthreads();

        Hx[ci] = Chxh[ci] * Hx[ci] + Chxez[ci] * (Ez[ci+nxx]-sEz[ty][tx]);
        Hy[ci] = Chyh[ci] * Hy[ci] + Chyez[ci] * (sEz[ty][tx+1] - sEz[ty][tx])
}
```

Listing 16.10 update_electric_fields_on_gpu_TEz

```
__global__ void
update_electric_fields_on_gpu_TEz(float* Cexe, float* Cexhz, float*
Ceye, float* Ceyhz,  float* Ex, float* Ey, float* Hz, int nxx)
{
        __shared__ float sHz[TILE_SIZE][2*TILE_SIZE+1];

        int tx = threadIdx.x;
        int ty = threadIdx.y;
        int i = blockIdx.x * blockDim.x + tx;
        int j = blockIdx.y * blockDim.y + ty;

        int ci = (j+1)*nxx+i;

        sHz[ty][tx] = Hz[ci-TILE_SIZE];
        sHz[ty][tx+TILE_SIZE] = Hz[ci];

        __syncthreads();

        Ex[ci] = Cexe[ci] * Ex[ci] + Cexhz[ci] * (Hz[ci]-Hz[ci-nxx]);
        Ey[ci] = Ceye[ci] * Ey[ci] + Ceyhz[ci] * (sHz[ty][tx+TILE_SIZE]-
sHz[ty][tx+TILE_SIZE-1]);
  }
```

Listing 16.11 update_magnetic_fields_on_gpu_TMz

```
void update_magnetic_fields_on_cpu_TMz()
{
       for (int i=0;i<number_of_cells-nxx;i++)
        Hx[i] = Chxh[i] * Hx[i] + Chxez[i] * (Ez[i+nxx]-Ez[i]);

       for (int i=0;i<number_of_cells-nxx;i++)
        Hy[i] = Chyh[i] * Hy[i] + Chyez[i] * (Ez[i+1] - Ez[i]);
}
```

Listing 16.12 update_electric_fields_on_gpu_TMz

```
void update_electric_fields_on_cpu_TMz()
{
        for (int i=nxx;i<number_of_cells;i++)
               Ez[i] = Ceze[i] * Ez[i] + Cezhy[i] * (Hy[i]-Hy[i-1]) +
Cezhx[i] * (Hx[i]-Hx[i-nxx]);
}
```

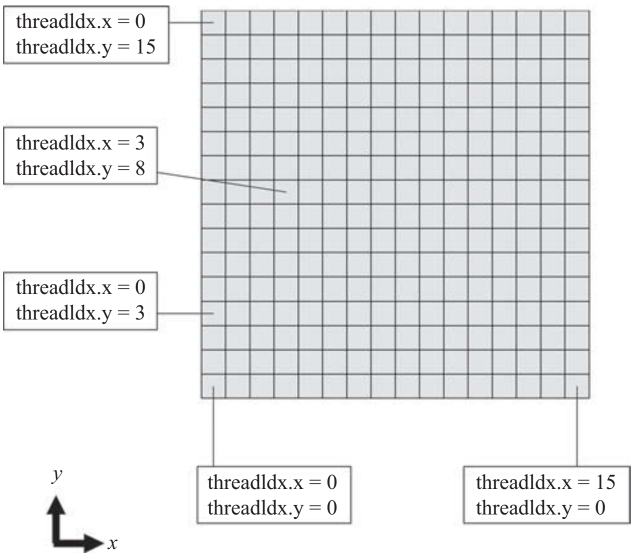

Figure 16.5 Threads in a thread block.

In MATLAB the index i runs faster than the index j, i.e., the data stored at (i, j) is adjacent to $(i+1, j)$ on the physical memory, thus the conversion to linear index can be done simply as

```
int ci = j*nxx+i;
```

However, in Listings 16.9 and 16.10 one can notice that the linear index is calculated as

```
int ci = (j+1)*nxx+i;
```

The reason is as follows: Consider the equation used to update E_z as shown in (1.36). In order to update $E_z(i,j)$ one needs $H_x(i,j-1)$, hence one cannot update $E_z(i,1)$ since $H_x(i,0)$ does not exist. Here MATLAB array indexing scheme is used which starts from 1, unlike C, which starts from 0. The update of $E_z(i,j)$, therefore, shall be performed starting from $j=2$, which can be controlled in the kernel code using an `if` statement. An `if` statement would reduce the efficiency of a kernel function considerably, thus to avoid such a situation, in the given FDTD algorithm one layer of cells is padded to the two-dimensional arrays in the $-y$ direction. This means that all field components that are being processed are shifted by one cell in the $+y$ direction, so final results would not be affected. Hence the calculation of `ci` is adjusted. Similar situation happens when updating H_z. In order to update $H_z(i,nyy)$ one needs $E_x(i,nyy+1)$, which does not exist on the physical memory if the arrays are of size $nxx \times nyy$. A straightforward solution is to pad another layer of cells to the two-dimensional arrays in the $+y$ direction. These two additional layers of cells are appended to the arrays in the subroutine ***save_project_data_to_file***.

16.2.3 Use of shared memory

Because it is on the GPU chip, the access to shared memory is much faster than the local and global memory. Parameters that reside in the shared memory space of a thread block have the lifetime of the block, and are accessible from all the threads within the block [96]. Therefore if a data block on global memory is going to be used frequently in a kernel, it is better to load the data to shared memory and reuse the data from the shared memory. Here it should be reminded that, shared memory is available only through the lifetime of a thread, thus the relevant field data have to be reloaded to the shared memory every time a kernel function is called.

Shared memory is especially useful when threads need to access to unaligned data. For instance, examining (1.38) reveals that in order to calculate $H_y(i,j)$, a thread mapped to the cell (i,j) needs E_z at (i,j) as well as E_z at $(i+1,j)$. In the kernel code the index of a thread that is processing the cell (i,j) is indexed as "ci". A cell with index $(i+1,j)$ can be accessed by `ci+1`, while a cell with index $(i,j+1)$ can be accessed by `ci+nxx`. Access to $(i,j+1)$ is coalesced, however $(i+1,j)$ is not. If an access to a field component at a neighboring cell in the x direction is needed, i.e. $(i+1,j)$, then shared memory can be used to load the data block mapped by the thread block, and then the neighboring field value is accessed from the shared memory. At this point one need to use the CUDA function `__syncthreads()` to ensure that all threads in the block are synchronized; thus all necessary data are loaded to the shared memory before they are used by the neighboring threads.

As discussed above, uncoalesced memory accesses can be prevented by using shared memory. However, a problem arises when accessing the neighboring cells' data through shared memory. While loading the shared memory, each thread copies one element from the global memory to the shared memory. If the thread on the boundary of the thread block needs to access the data in the neighboring cell, this data will not be available since it has not been loaded to the shared memory. One way to overcome this problem is to load another set of data, which includes the neighboring cell's data, to the shared memory. In the presented implementation shown in Listing 16.9, two square blocks of data are copied from global memory to the shared memory as

```
sEz[ty][tx] = Ez[ci];
sEz[ty][tx+TILE_SIZE] = Ez[ci+TILE_SIZE];
```

Then E_z at $(i+1,j)$ is safely accessed from the shared memory to update $H_y(i,j)$ as

```
Hy[ci] = Chyh[ci] * Hy[ci] + Chyez[ci] * (sEz[ty][tx+1] -
sEz[ty][tx]);
```

A similar treatment is shown in Listing 16.10, where $E_y(i,j)$ is updated by $H_z(i-1,j)$ through an effective use of shared memory: First two blocks of data is from global memory to the shared memory as

```
sHz[ty][tx] = Hz[ci-TILE_SIZE];
sHz[ty][tx+TILE_SIZE] = Hz[ci];
```

Then $E_y(i,j)$ is updated as

```
Ey[ci] = Ceye[ci]  *  Ey[ci] + Ceyhz[ci]  *  (sHz[ty][tx+TILE_
SIZE]-sHz[ty][tx+TILE_SIZE-1]);
```

16.2.4 Optimization of number of threads

As pointed out in recommendations R5 and R6 in Section 16.1, occupancy of the microprocessors and number of threads in a block are two other important parameters that affect the performance of a CUDA program. Number of threads and occupancy are tightly connected. It is possible to set the number of threads as a desired value while it may not be possible to control the occupancy; it is a function of number of threads, number of registers used in the kernel, amount of shared memory used by the kernel, compute capability of the device, etc. A good practice is to optimize the number of threads while keeping the occupancy at a reasonable value. As a rule of thumb, it is better to keep kernel functions small such that they do not use many registers.

CUDA Visual Profiler is a graphical user interface based profiling tool provided by NVIDIA that can be used to measure performance and find potential opportunities for optimization in order to achieve maximum performance of kernels in a CUDA program. It can show information such as total GPU and CPU times of a thread, memory read-write throughputs, global memory load and store efficiencies, occupancy, etc. Therefore, the kernel functions can be profiled using the Visual Profiler and, if not satisfactory, their efficiencies can be improved.

In the presented code the block size is chosen as $16 \times 16 = 256$. One can choose a different configuration and test with Visual Profiler to evaluate the efficiency. It should also be noted that the presented grid scheme, i.e., two-dimensional array of square thread blocks, is not the only way to map threads to cells. It is possible to use other grid configurations, such as using one-dimensional thread blocks in a one-dimensional grid, develop algorithms based on these configurations, and evaluate if the developed algorithms are superior.

16.3 Performance of two-dimensional FDTD on CUDA

The main purpose of developing a code to run FDTD program on a graphics card is to run it faster by utilizing the parallel processing capability of GPU. The performance of the developed CUDA code is examined as a function of problem size. The analysis is performed on an NVIDIA® Tesla™ C1060 Computing Processor installed on a 64-bit Windows XP computer. This card has 240 streaming processor cores operating at 1.3 GHz. Size of a two-dimensional FDTD problem domain has been swept and the throughputs of the program are calculated as number of million cells processed per second (*NMCPS*) as a measure of the performance of the CUDA program. Number of million cells is calculated as [97]

$$NMCPS = \frac{n_{steps} \times Nxx \times Nyy}{t_s} \times 10^{-6}, \tag{16.1}$$

where n_{steps} is the number of time steps the program has been run and t_s is the total time of program run in seconds. Here, the problem that is discussed in Section 7.5.2, illustrated in Figure 7.15, is simulated for performance evaluation. The problem space is terminated by PEC boundaries. The results, shown in Figure 16.6, reveal that as the problem size is increased, the throughput of the CUDA calculations increase to about 900 million cells per second. Figure 16.6 also shows the throughput of the FDTD program when all calculations are performed on the CPU. The average throughput on CPU is about 25 million cells per second. For the presented implementations, GPU calculations are more than 36 times faster than the CPU calculations.

CUDA and OpenGL are two software platforms both of which operate on the GPU hardware, while their intended use are different; CUDA is to improve the performance of data parallel computations, while OpenGL is to manipulate data to produce 2D and 3D computer graphics. While running an FDTD simulation, it is possible to capture electromagnetic fields and display them as an on-the-fly animation. If the FDTD calculations are performed on a graphics card, which is used also to perform the OpenGL operations to display the field, one can copy the field data from graphics card memory (device global memory) to computer's main memory that is processed by CPU (host memory), process the data, and copy back to GPU memory to display via OpenGL. Thanks to CUDA/OpenGL

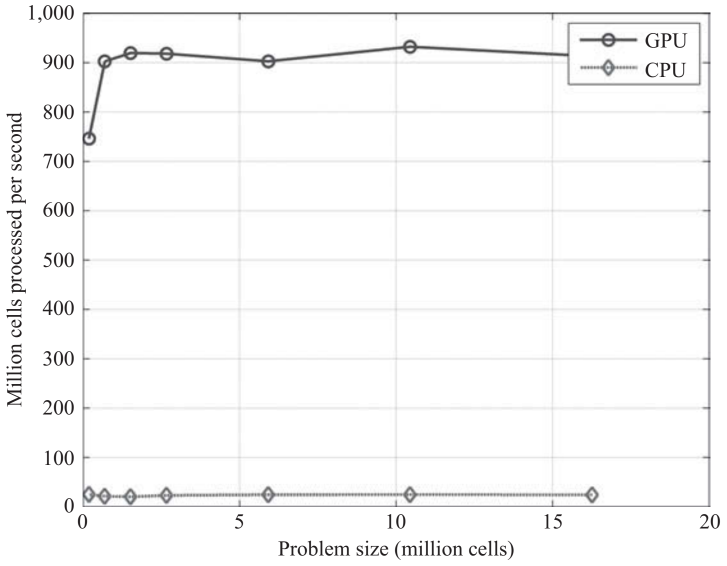

Figure 16.6 Throughput of CUDA FDTD calculations on a Tesla C1060 card.

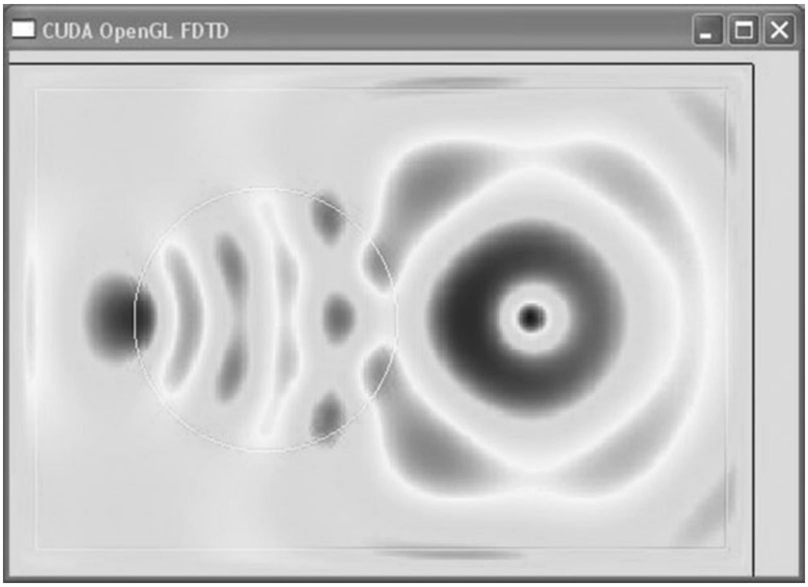

Figure 16.7 A snapshot of electric field distribution scattered from a dielectric cylinder due to a line source with sinusoidal excitation.

interoperability provided by CUDA, it is possible to avoid the back and forth data transfer between the host and device memories, and perform all processing required for the display on the graphics card.

The presented CUDA code also utilizes the CUDA/OpenGL interoperability to capture and display electromagnetic fields. The test case presented above is run with electromagnetic field animation enabled, and electric field distribution is captured and displayed on a window while the program is running. The snapshot of the display window is shown in Figure 16.7. The details for CUDA-OpenGL interoperability can be found in [98].