An Introduction to GPGPU Programming - CUDA Architecture

Rafia Inam
Mälardalen Real-Time Research Centre
Mälardalen University, Västerås, Sweden
http://www.mrtc.mdh.se
rafia.inam@mdh.se

CONTENTS

List of Figures

1 Introduction

1.1 Background

At the start of multicore CPUs and GPUs the processor chips have become parallel systems. But speed of the program will be increased if software exploits parallelism provided by the underlying multiprocessor architecture [1]. Hence there is a big need to design and develop the software so that it uses multithreading, each thread running concurrently on a processor, potentially increasing the speed of the program dramatically. To develop such a scalable parallel applications, a parallel programming model is required that supports parallel multicore programming environment.

NVIDIA's graphics processing units (GPUs) are very powerful and highly parallel. GPUs have hundreds of processor cores and thousands of threads running concurrently on these cores, thus because of intensive computing power they are much faster than the CPU as shown in Figure 1.

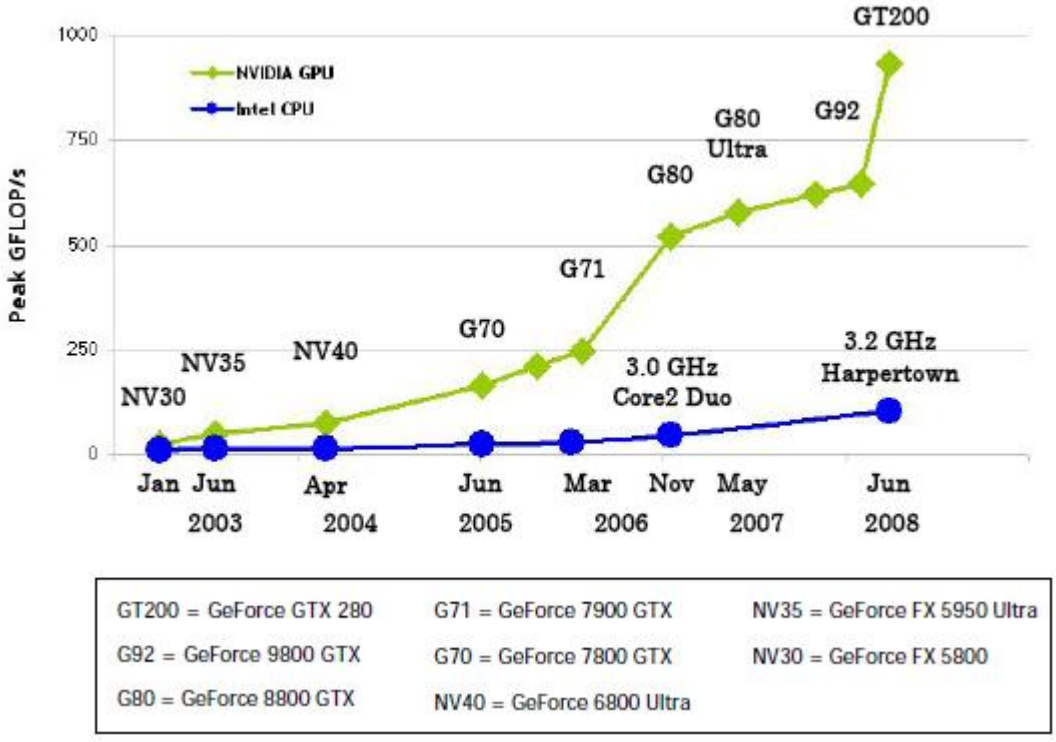

Figure 1: Comparing GPU to CPU [1]

At start, they were used for graphics purposes only. But now GPUs are becoming more and more popular for a variety of general-purpose, non-graphical applications too. For example they are used in the fields of computational chemistry, sparse matrix solvers, physics models, sorting, and searching etc. The programs designed for GPGPU (General Purpose GPU) run on the multi processors using many threads concurrently. As a result, these programs are extremely fast.

Report Outline: This report is organized as follows. The rest of section 1 introduces the CUDA. In section 2, the detailed description of the CUDA system model is given. Some commonly used CUDA API is provided in section 3.

1.2 CUDA

CUDA stands for Compute Unified Device Architecture. It is a parallel programming paradigm released in 2007 by NVIDIA. It is used to develop software for graphics processors and is used to develop a variety of general purpose applications for GPUs that are highly parallel in nature and run on hundreds of GPU's processor cores.

CUDA uses a language that is very similar to C language and has a high learning curve. It has some extensions to that language to use the GPU-specific features that include new API calls, and some new type qualifiers that apply to functions and variables. CUDA has some specific functions, called *kernels*. A kernel can be a function or a full program invoked by the CPU. It is executed N number of times in parallel on GPU by using N number of threads. CUDA also provides shared memory and synchronization among threads.

CUDA is supported only on NVIDIA's GPUs based on Tesla architecture. The graphics cards that support CUDA are GeForce 8-series, Quadro, and Tesla. These graphics cards can be used easily in PCs, laptops, and servers. More details about CUDA programming model are described in the next section.

2 The System Model

2.1 The System Model

Graphics processors were mainly used only for graphics applications in the past. But now modern GPUs are fully programmable, highly parallel architectures that delivers high throughput and hence can be used very efficiently for a variety of general purpose applications.

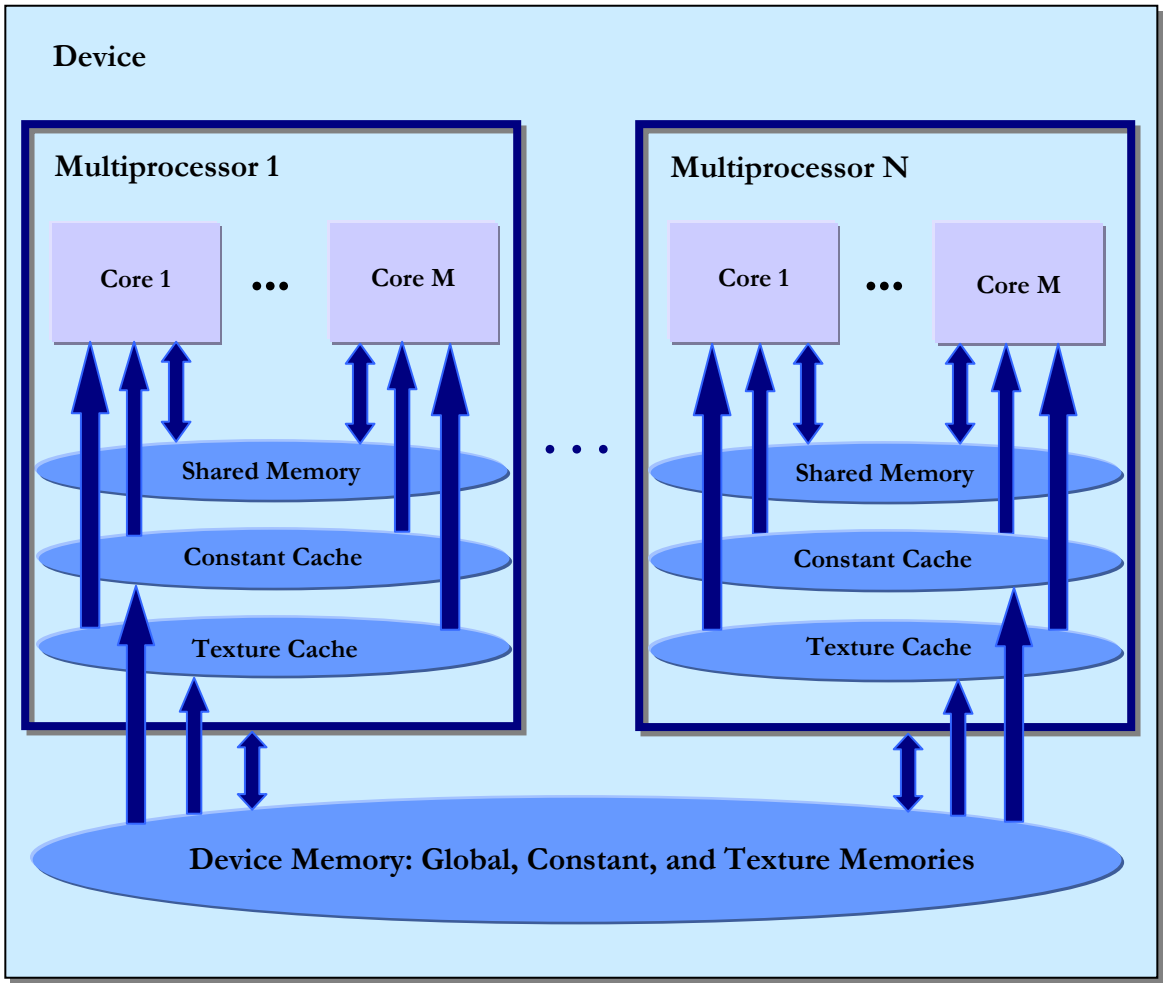

Figure 2: The CUDA Memory Model [3]

NVIDIA's graphics card is a new technology that is extremely multithreaded computing architecture. It consists of a set of parallel multiprocessors, that are further divided into many cores and each core executes instructions from one thread at a time as described in Figure 2.

Hence all those computations in which many threads have to execute the same instruction concurrently, also called data-parallel computations, are well-suited to run on GPU.

NVIDIA has designed a special C-based language CUDA to utilize this massively parallel nature of GPU. CUDA contains a special C function called *kernel*, which is simply a C code that is executed on graphics card on fixed number of threads concurrently. For defining threads, CUDA uses a grid structure.

2.2 Heterogeneous Architecture

CUDA programming paradigm is a combination of serial and parallel executions. Figure 3 shows an example of this heterogeneous type of programming. The simple C code runs serially on CPU also called the *host* [2].

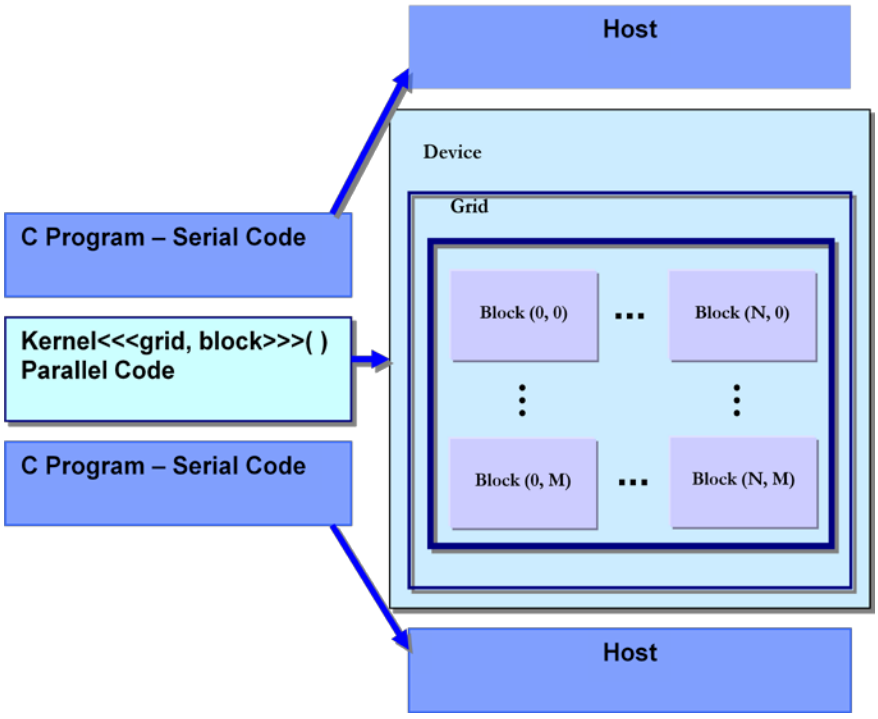

Figure 3: Heterogeneous Architecture [3]

Parallel execution is expressed by the *kernel function* that is executed on a set of threads in parallel on GPU; GPU is also called *device*. This kernel code is a C code for only one thread. The numbers of thread blocks, and the number of threads within those blocks that execute this kernel in parallel are given explicitly when this function is called.

The kernel function can only be invoked by serial code from CPU. To call the kernel function, the execution configuration must be specified, i.e., the number of threads in a thread block and number of threads within a grid. To declare grid and thread blocks CUDA has a predefined data type *dim3*, an integer vector type that specifies the dimensions of the grid and thread blocks. In the kernel function call grid and block variables are written in three angular brackets <<< grid, block >>> as shown in Figure 3Figure 4. In this invocation, grid and thread blocks are created dynamically. The value of this grid and block variables must be less than the allowed sizes which are given in next section. The threads are scheduled in hardware and not in software. Kernel function has always a return type *void*. It has a qualifier __global__ that means this is a kernel function to be executed on GPU. See Figure 4 for a graphical description of grid and thread blocks.

2.3 The Grid and block structures

The Grid consists of one-dimensional, two-dimensional or three-dimensional thread blocks. Each thread block is further divided into one-dimensional or two-dimensional threads. A thread block is a set of threads running on one processor. Figure 4 describes a two-dimensional grid structure and a two-dimensional block structure. Within a thread block, threads are organized together in warps. Normally 32 threads are grouped in one warp. All threads of a warp are scheduled together for execution.

All threads of a single thread block can communicate with each other through shared memory; therefore they are executed on the same multiprocessor. In this way it becomes possible to synchronize these threads.

The CUDA paradigm provides some built-in variables to use this structure efficiently. To access the id of a thread block the *blockIdx* variable (values from 0 to gridDim-1) is used and to access its dimension the *blockDim* variable is used while *gridDim* gives the dimensions of the grid. Each individual thread is identified by *threadIdx* variable, can have values from 0 to blockDim-1. *WarpSize* specifies warp size in the threads. All these variables are built-in in kernel. The maximum allowed sizes of each dimension of grid is 65535, and x, y, and z dimensions of a thread block are 512, 512, and 64, respectively [1] [2].

The allocation of the number of thread blocks to each multiprocessor is dependent on the necessity of the shared memory and registers by each thread block. More memory and registers requirement by each thread block means allocation of less thread blocks to each multiprocessor. In this case the remaining thread blocks have to wait for their turn for execution.

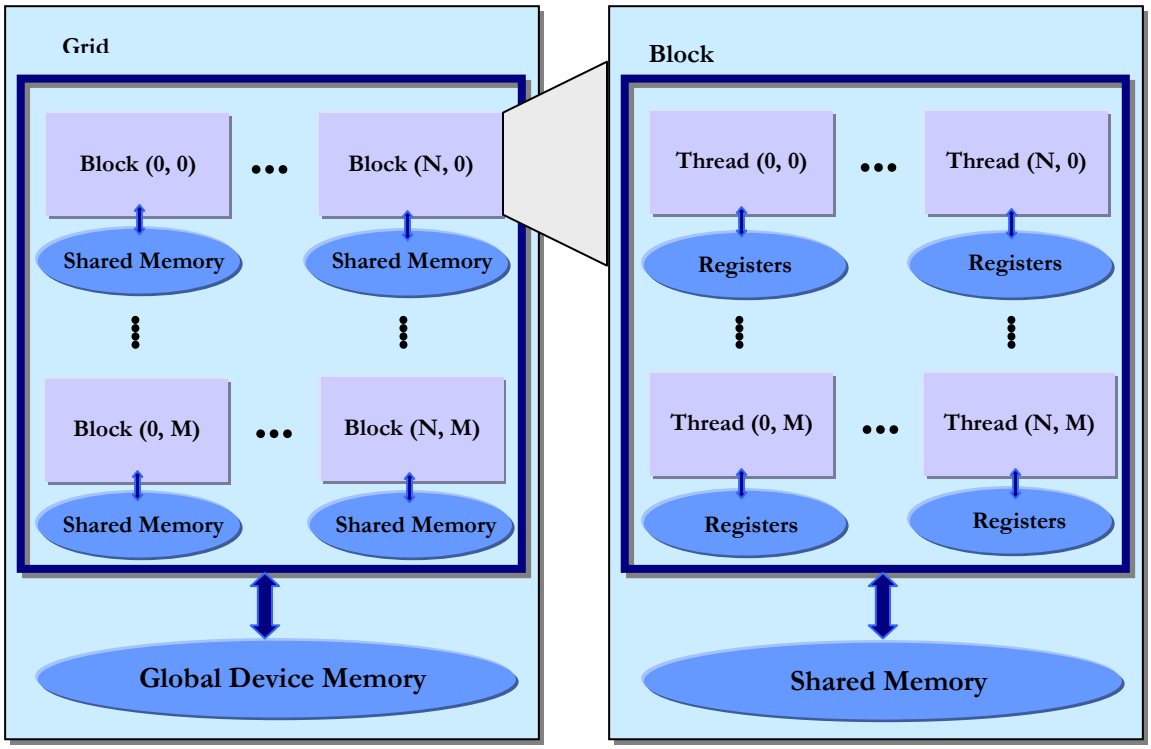

Figure 4: The CUDA Grid Structure and Block Structure.

All this threads creation, their execution, and termination are automatic and handled by the GPU, and is invisible to the programmer. The user only needs to specify the number of threads in a thread block and the number of thread blocks in a grid.

2.4 Memory Model

All multiprocessors access a large global device memory for both gather and scatter operations. Memory model is described graphically in Figure 2. This memory is relatively slow because it does not provide caching.

Shared memory is fast as compared to device memory and normally takes the same amount of time as required to access registers. It is also called parallel data cache (PDC). Shared memory is “local” to each multiprocessor unlike device memory and allows more efficient local synchronization. It is divided into many parts. Each thread block within multiprocessor accesses its own part of shared memory and this part of shared memory is not accessible by any other thread block of this multiprocessor or of some other multiprocessor. All threads within a thread block that have the same life time as of the block, share this part of memory for both read and write operations. As shared memory space is only 16KiB, so it must be used efficiently. To declare variables in shared memory __shared__ qualifier is used and to declare in global memory __device__ qualifier is used.

Each multiprocessor also has its own read only caches to speed up read operation. These are constant cache and texture cache memories.

Each thread also contains its own local memory. Normally local variables of the kernel functions are allocated here. Sometimes they are allocated on global memory.

2.5 Thread Synchronization

For synchronization purpose among threads CUDA API provides a hardware thread-barrier function *syncthreads()* that acts as synchronization point. As threads are scheduled in hardware, this function is implemented in hardware. The threads will wait at the synchronization point until all of the threads have reached at this point. The communication among threads (if required) is possible through per-block shared memory. Hence thread synchronization is possible only at thread block level. Since threads of a thread block may communicate with each other, these threads must execute on same processor. That is why thread block is guaranteed to execute on one processor.

2.6 Number of Threads per Block

To maximize the utilization of available resources, the assignment of the number of threads per block and the number of thread blocks per grid should be done carefully. Less number of threads per block cause load latency in device memory reads and also one block per multiprocessor makes the multiprocessor to idle during thread synchronization. Hence there should be at least twice as many blocks as there are multiprocessors in the device (The number of blocks per grid should be at least 100). Also assign the number of threads per block in multiples of the warp size, because it lessens the under-populated wraps.

2.7 Control Flow

As the kernel function runs on the device, memory must be allocated on device in advance before kernel function invocation and if the kernel function has to execute on some data then the data must be copied from the host memory to the device memory. Device memory can be allocated either as *linear memory* or as *CUDA arrays*. Qualifier __device__ at the start of a variable specifies that space for this variable is allocated on the device memory. CUDA API [2] also has functions to allocate and de-allocate device memory at run time like cudaMalloc(), cudaFree(), etc. Similarly, after the execution of kernel function, data from device memory must be copied back to host memory in order to get results. To copy data to and from the device to host CUDA API provides functions for example cudaMemCpyToSymbol(), cudaMemCpyFromSymbol(), cudaMemCpy(), etc. Keeping all this in view the processing flow is as follows:

1. Allocate memory on host and device separately. Device memory is readable and writable by the host through the memory copy functions.
2. Copy data from host to device using CUDA API if required.
3. Kernel function executes parallel on each core.
4. Copy data back from device to host using CUDA API.

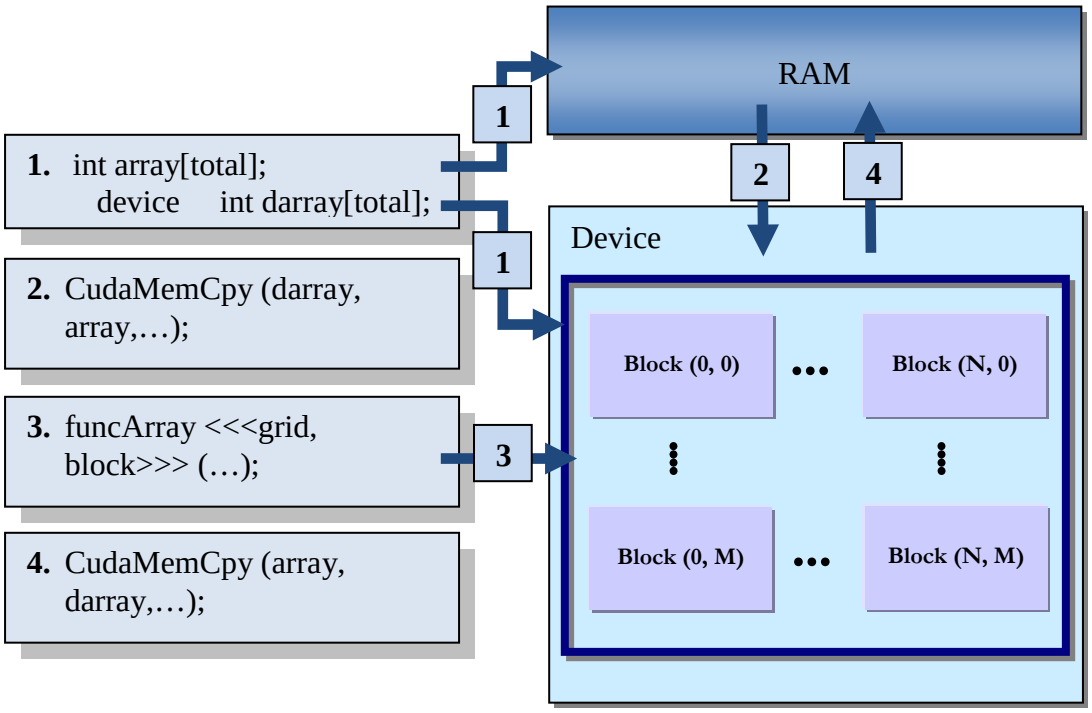

Figure 5: An example of processing flow [3].

Figure 5 illustrates an example of processing flow of CUDA. In first step two arrays of same size are declared, one on the host and one on the device. The data from the host is copied to the device using CUDA API cudaMemCpy(). The kernel function runs in parallel on the device and in last step the results are copied back to the host using cudaMemCpy() function.

2.8 Transferring Data between Host and Device

Since the bandwidth between the device memory and the host memory is much less as compared to the bandwidth between the device and the device memory which is very high, we should try to minimize data transfer between the host and the device. Some of the efforts could be

like moving some code from the host to the device and creating and destroying data structures in the device memory (instead of copying them to the device) and making big transfers by batching up many small transfers to lessen the transfer overheads.

2.9 Restrictions

To use general purpose GPU we must follow the restrictions of the CUDA programming paradigm. Some of the restrictions are given below:

Simple C programming is supported by the CUDA compiler. It lacks the use of object-oriented or C++ features in device code.

Heterogeneous architecture is used to make an interaction between CPU and GPU programming models. Data may be copied from host memory to device memory and the results are copied back to host from the device memory. Heterogeneous programming is discussed in section 2.2 and described graphically in Figure 3.

Kernel function invocation: The grid, thread blocks, and threads are created by the kernel function invocation from the host. This is the only way to create them. They cannot be created inside the kernel function. The grid, and thread blocks are discussed in Figure 4. Moreover the number of grids and thread blocks must not exceed their maximum allowed values.

The kernel functions do not return any results, i.e. its return type is always void. Further the kernel function call is asynchronous. It means that control returns back before the completion of the kernel function on the device. More information can be found in CUDA programming guide [1]. All functions with the __device__ qualifier are by default inline.

Recursion is simply not allowed within kernel functions because of the large amount of memory requirement for the thousands of thread.

The device memory allocation and de-allocation at run-time is possible only when using host code and before calling the device code. It means that within the device code, the device memory cannot be allocated nor de-allocated using the functions like cudaMalloc(), cudaFree(), etc. All the allocations required for a specific kernel function are done before calling that kernel function in the host code and similarly all that allocated device memory is de-allocated after the completion of that kernel function in the host code.

Shared memory is shared among the threads on the same thread block only. Threads from different thread block cannot share it. This concept is discussed in section 2.4 and graphically shown in Figure 4.

Built-in variables such as blockIdx, threadIdx, etc, cannot be assigned any values. Further it is not possible to take their address.

The variables declared with __device__, __shared__, or __constant__ qualifiers also have some restrictions [1]. Address of a variable with any one of these qualifiers can only be used within the device code.

Communication and synchronization among threads are only possible at thread block level. Communication among thread blocks is not allowed. Section 2.5 explains the thread synchronization.

3 Some Commonly used CUDA API

3.1 Function Type Qualifiers

The three main types of the function qualifiers in CUDA are *device, global, and host*.

1. **__device__**

   The functions with device qualifier are executed on the device. These functions are callable from the device only.

2. **__global__**

   The functions with global qualifier are executed on the device but they are callable from the host only.

3. **__host__**

   The functions with host qualifier are executed on the host and are callable from the host only. When no qualifier is used, it means that the function will run on the host; it is equivalent to the function declared with the _host_ qualifier.

3.2 Variable Type Qualifiers

The three main types of the variable qualifiers in CUDA are *device, constant, and shared*.

1. **__device__**
   The variables declared with __device__ reside on the device. Other type qualifiers are optionally used together with __device__. If a variable is declared only with __device__ qualifier then this variable resides in the global memory and it has the lifetime of the application. Since it resides in the global memory, it is accessible from all the threads (within the grid) and host through the runtime library.

2. **__constant__**

   This qualifier is used to allocate constants on the device. It is optionally used together with __device__ qualifier. This constant resides in constant memory, and has the lifetime of an application. It is accessible from all the threads (within grid) and host through the runtime library.

3. **__shared__**

   This qualifier is used to allocate the shared variable. It is optionally used together with __device__ qualifier. Shared variable resides in shared memory of a thread block, and has the lifetime of a block. It is only accessible from all the threads within the block.

3.3 Built-in Variables

Following is a list of some of the built-in variables in CUDA:

1. **gridDim:** is of type **dim3** and contains the dimensions of the grid.
2. **blockIdx:** is of type **uint3** and contains the block index within the grid.
3. **blockDim:** is of type **dim3** and contains the dimensions of the block.
4. **threadIdx:** is of type **uint3** and contains the thread index within the block.
5. **warpSize:** is of type **int** and contains the warp size in threads.

3.4 Memory Management

1. Memory Allocation

```
float* darray;
cudaMalloc((void**)&darray, 1024 * sizeof(float));
```

2. Memory Deallocation

```
cudaFree(darray);
```

3.5 Copying Host to device

1**. Copying host memory array to device memory:**

```
cudaMemcpyToSymbol( const T& symbol, const void* src,
size_t count)
Example:
float cpuArray [1024];
_device_ float dArray [1024];
cudaMemcpyToSymbol (dArray, cpuArray, sizeof(cpuArray));
```

2. **Another method**

```
Example:
float cpuArray[1024];
int size = sizeof(cpuArray);
```

```
float* dArray;
cudaMalloc((void**)&dArray, size);
cudaMemcpy(dArray, cpuArray, size, cudaMemcpyHostToDevice);
```

2. **Copying host memory array to constant memory:**

Example:

```
__constant__ float constArray[1024];
float cpuArray[1024];
cudaMemcpyToSymbol(constArray, &cpuArray, sizeof(constArray));
```

3.6 Copying Device to Host

1. **Copying device memory array to host memory:**

```
cudaMemcpyFromSymbol( void *dst, const T& symbol, size_t
count)
```

Example:

```
float cpuArray [1024];
_device_ float dArray [1024];
cudaMemcpyFromSymbol (&cpuArray, dArray, sizeof(dArray));
```

2. **Another method**

Example:

```
float cpuArray[1024];
int size = sizeof(cpuArray);
float* dArray;
cudaMalloc((void**)&dArray, size);
cudaMemcpy(cpuArray, dArray, size, cudaMemcpyDeviceToHost);
```

3.7 Device Runtime Component

Device runtime components are only be used in the device functions and are prefixed with an underscore symbol __. The following is a short list of these functions:

1. **Mathematical Functions:**

   (e.g. __sinf(x) , __cosf(x), sqrt(x), etc)

2. **Synchronization Function:**

   void __syncthreads();

3. **Atomic Functions:**

   (e.g. atomicAdd(), etc.)

4. **Texture Functions:**

3.8 Device Emulation Mode

A device emulation mode is provided basically for the debugging purpose. –deviceemu option is used with nvcc compile command. It only emulates the device, it is not the simulation. Threads and the thread blocks are created on the host. Host's native debugging (like Microsoft Visual studio's) can be used in setting break points and data inspection. It is especially helpful in input or output operations to the files or to the screen, like the use of printf() function, that is not possible to run on the device.

3.9 An Example

3.9.1 Sequential Code

A sequential program to calculate the distances from a specific point to the all other points in a 2D Matrix of order N × N is given below:

```
const int N=16;
void main (void) {
   int i, j, x, y;
   float hgrid[N][N];

   printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &x);
   printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &y);

  // Code to find distance without using device
  for (i=0; i<N; i++){
            for (j=0; j<N; j++) {
                     n = ((i-x)*(i-x))+((j-y)*(j-y));      // distance formula
                     hgrid[i][j] = sqrt(n);                // distance formula
                     printf("\t%.0lf", hgrid[i][j]);
            }
            printf("\n\n");
   }
}
```

3.9.2 Parallel Code – 1D Grid

Now the same program is converted to the parallel code to run on the device. A one dimensional grid with only one thread block is used. The thread block contains 16 * 16 threads (hence 256 threads in total) in a two dimensional form.

```
 const int N=16;
 __device__ float dgrid[N][N];  // array on device memory

// function on device to calculate distance
  __global__ void findDistance( int x, int y){
      int i = threadIdx.x;
      int j = threadIdx.y;

      float n = ((i-x)*(i-x))+((j-y)*(j-y));
      dgrid[i][j] = sqrt(n);
}



void  main () {
    int i, j;
   float hgrid[N][N];

   dim3 dBlock(N, N);                  // thread block with total 256 threads

   printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
   printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
   printf( "\n\tDistance from a node!\n\n\n" );

   findDistance<<<1, dBlock>>>(i, j);   // Calling kernel function
   cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid));  //copy device memory to host

   printf( "\n\n\tValues in hgrid!\n\n" );
   for (i=0; i<N; i++){
     for (j=0; j<N; j++)
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n");
     }
   }

}
```

3.9.3 Parallel Code – 2D Grid (2 * 2)

Now the same program is converted to the parallel code to run on the device with a two dimensional grid (2 thread blocks in x dimension and 2 in y dimension). The thread block contains 16 * 16 threads (hence 256 threads in total) in a two dimensional form. Hence total 1024 threads will run in parallel in the device.

```
const int N=16;
const int D=2;
__device__ float dgrid[N*D][N*D];   // array on device memory

// function on device to calculate distance
 __global__ void findDistance( int x, int y){
     int i =  blockIdx.x * blockDim.x + threadIdx.x;
     int j =  blockIdx.y * blockDim.y + threadIdx.y;

     float n = ((i-x)*(i-x))+((j-y)*(j-y));
     dgrid[i][j] = sqrt(n);
}


void  main () {
    int i, j;
   float hgrid[N*D][N*D];

   dim3 dGrid(D,D);                  // 2D grid with total 4 thread blocks
   dim3 dBlock(N, N);                // thread block with total 256 threads

   printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
   printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
   printf( "\n\tDistance from a node!\n\n\n" );

   findDistance<<<  dGrid, dBlock>>>(i, j);   // Calling kernel function
   cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid));  //copy device memory to host

   printf( "\n\n\tValues in hgrid!\n\n" );
   for (i=0; i<N*D; i++){
     for (j=0; j<N*D; j++)
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n");
     }
   }

}
```

3.9.4 Parallel Code – 2D Grid (4 * 4)

The same program is converted to the parallel code to run on the device with a two dimensional grid (4 thread blocks in x dimension and 4 in y dimension). The thread block contains 8 * 8 threads (hence 64 threads in total) in a two dimensional form. Hence total 1024 threads will run in parallel in the device.

```
const int N=8;
const int D=4;
__device__ float dgrid[N*D][N*D];   // array on device memory

// function on device to calculate distance
  __global__ void findDistance( int x, int y){
      int i =  blockIdx.x * blockDim.x + threadIdx.x;
      int j =  blockIdx.y * blockDim.y + threadIdx.y;

      float n = ((i-x)*(i-x))+((j-y)*(j-y));
      dgrid[i][j] = sqrt(n);
}


void  main () {
    int i, j;
   float hgrid[N*D][N*D];

   dim3 dGrid(D,D);                  // 2D grid with total 16 thread blocks
   dim3 dBlock(N, N);                // thread block with total 64 threads

   printf( "\n\tEnter the x coordinate of node : " ); scanf_s("%d", &i);
   printf( "\n\tEnter the y coordinate of node : " ); scanf_s("%d", &j);
   printf( "\n\tDistance from a node!\n\n\n" );

   findDistance<<<  dGrid, dBlock>>>(i, j);   // Calling kernel function
   cudaMemcpyFromSymbol( &hgrid, dgrid, sizeof(dgrid));  //copy device memory to host

   printf( "\n\n\tValues in hgrid!\n\n" );
   for (i=0; i<N*D; i++){
     for (j=0; j<N*D; j++)
            printf("\t%.0lf", hgrid[i][j]);
            printf("\n\n");
     }
   }

}
```

References

[1] NVIDIA CORPORATION, CUDA Programming Guide, http://developer.nvidia.com/cuda

[2] NVIDIA CORPORATION, CUDA Reference Manual, http://developer.nvidia.com/cuda

[3] Inam, Rafia, **A* Algorithm for Multi-core Graphics processors**, Master's Thesis, *Chalmers University of Technology, Göteborg, 2010,* http://publications.lib.chalmers.se/cpl/record/index.xsql?pubid=129175