UNIT VI

CHAPTER 6

CUDA Architecture

Syllabus Topics

CUDA Architecture, Using the CUDA Architecture, Applications of CUDA, Introduction to CUDA C-Write and launch CUDA C kernels, Manage GPU memory, Manage communication and synchronization, Parallel programming in CUDA-C.

What is CUDA ?

- **CUDA** (Compute Unified Device Architecture) is a programming interface provided by NVIDIA. The basic purpose of CUDA is to allow direct programming of NVIDIA hardware.
- NVIDIA released CUDA in 2007 and announced it is a parallel programming paradigm. CUDA is used to develop software for high - end graphics processors.
- The motivation for CUDA is to develop general purpose applications for Graphics Processor Units (GPUs).
- The language used in CUDA is very similar to C programming language.
- It has some extensions included to use GPU specific features in the form of new APIs and type qualifiers apply to functions and variables.
- CUDA contains some specific functions are called as kernels. Actually in CUDA a kernel can be a full program or function invoked by the CPU. CUDA provided highly parallel execution facilities in terms of threads.
- A kernel 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.

Syllabus Topic : CUDA Architecture

6.1 CUDA Architecture

Q. 6.1.1 Explain CUDA Architecture with schematic diagram. (Refer section 6.1) **(5 Marks)**

- As described previously that CUDA is Compute Unified Device Architecture.
- Basically the CUDA paradigm is used for Single Program Multiple Data (SPMD) computing device.

The architecture of CUDA is depicted in Fig. 6.1.1.

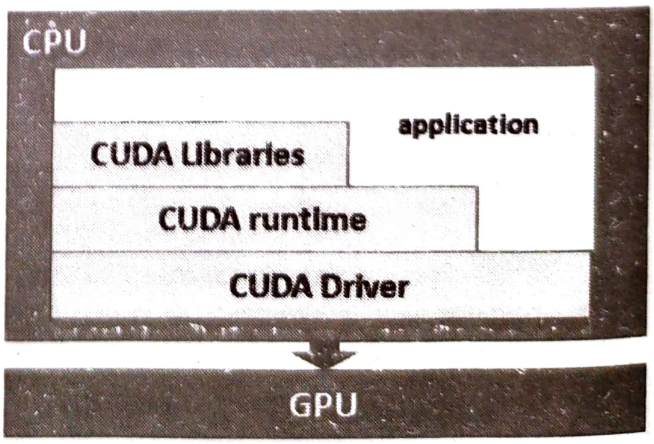


Fig. 6.1.1 : CUDA architecture

- NVIDIA presented CUDA architecture as a paradigm for GPUs. In this architecture several new components are included; these components are specially designed for GPU computing.
- In previous GPUs there were some limitations because of that GPUs were not used for non - graphical calculations.
- The main aim of newly included components is to remove these limitations. After providing these features the GPU execution units are able to read and write arbitrary memory locations.
- In addition it is also allowed to access a cache maintained in software called shared memory.
- The inclusion of additional architectural features allowed using CUDA GPU for general purpose calculations as well as in traditional graphics tasks.

The division of space between the various components of a graphics processing unit (GPU) and a central processing unit (CPU) are depicted in Fig. 6.1.2.

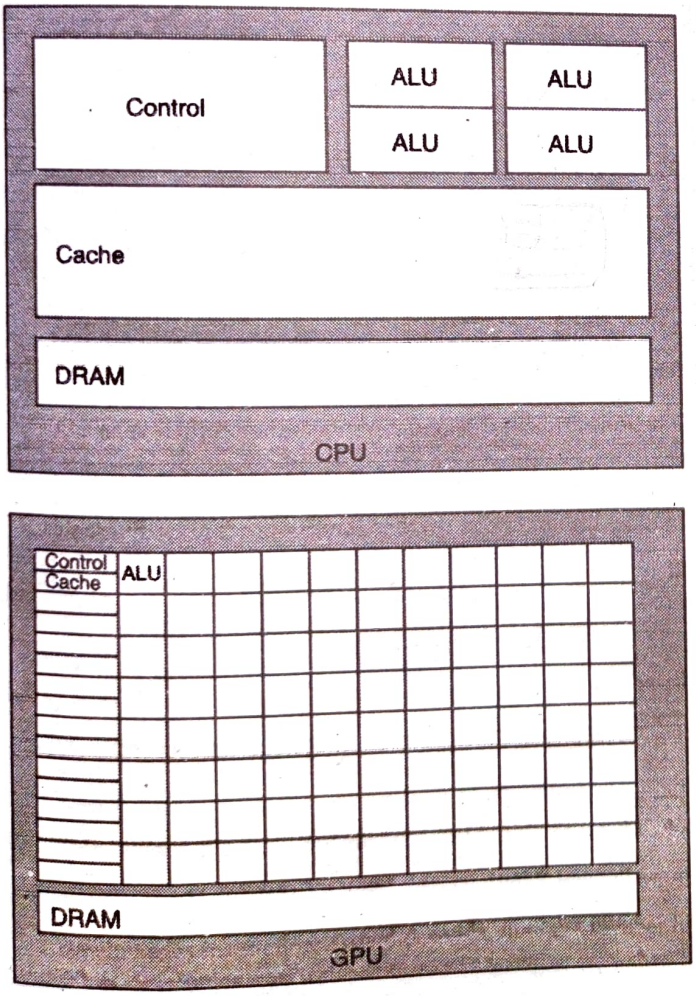


Fig. 6.1.2 : CPU versus GPU architecture

- As seen clearly that more transistors are devoted for data processing in GPU as compare to CPU. The GPU is a highly parallel, multithreaded, and many core processor.

Syllabus Topic : Using the CUDA Architecture

6.2 Using the CUDA Architecture

- NVIDIA used industry standard C language to be used to access features of CUDA architecture and develop high - end graphics as well as general purpose applications.
- The aim of choosing C programming language with additional features was to attract maximum number of developers for the application development in this field.
- Actually a relatively small number of keywords require din access of special features of CUDA architecture and these are added in the CUDA C.
- NVIDIA made available a compiler for CUDA C immediately after launching of its architecture named as GeForce 8800 GTX.
- In this way CUDA C became the first language created for developing general purpose language used in the field of GPU computing.
- The CUDA architecture provides massively computational power for creating highly - compute intensive tasks. The specialized hardware drivers are provided by NVIDIA to access facilities of CUDA architecture specially provided for massively computational power.
- These facilities provided flexibility for the user to develop applications without require knowing OpenGL or DirectX graphics programming interfaces.

Syllabus Topic : Applications of CUDA

6.3 Applications of CUDA

Q. 6.3.1 Describe about the applications of CUDA. (Refer section 6.3) **(5 Marks)**

- The main aim and strong point of CUDA is to provide highly parallel computing platform.

- In many high performance computing problems this is actually a common problem to get and access highly parallel environments.
- So, CUDA provided features are used to solve many HPC based problems.
- Here is a list of applications created using CUDA to achieve maximum performance which were not possible on a CPU alone.

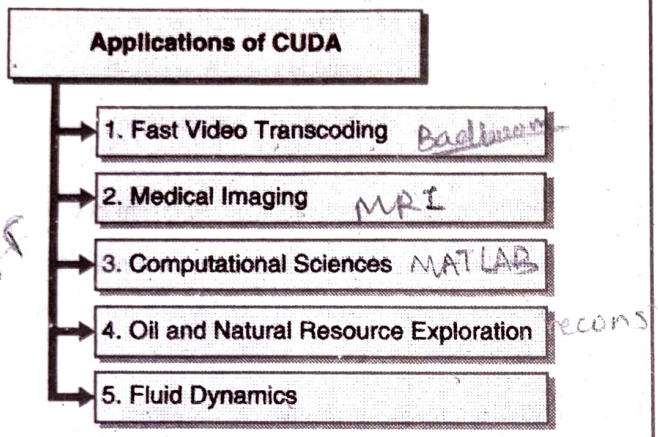


Fig. 6.3.1 : Applications of CUDA

➜ 1. Fast Video Transcoding

- The process of converting a file from one encoding format to another is termed as transcoding.
- The incompatible data are converted to a better - supported format using the transcoding technique.
- Usually transcoding is performed for devices with less storage capacity and does not support the format.
- The transcoding procedure is used most often for conversion of video formats.
- Graphics and HTML files can be used in mobile devices and other web - enabled small screen, lower bandwidth and less memory devices using transcoding.
- In transcoding procedure, trillions of parallel computations are involved and many of such computations are based on floating point operations.
- Badaboom is an example application which is created to harness the raw computing power of GPUs to transcode videos much faster than ever before.
- Badaboom can be used to transcode the DVD to be played on iPod. In the similar fashion it can be used to transcode an entire movie faster than real time.

➜ 2. Medical Imaging

- In medical imaging techniques and processes the images of various human body parts are captured for diagnosis and further treatments.
- Imaging techniques encompass the fields of radiology, nuclear medicine and optical imaging and image-guided intervention.
- The use of CUDA in the medical imaging provides significant advancements for image capturing and diagnosis. MRI machines can compute images faster than before using CUDA with significant reduce in price.
- This means CUDA provides considerable performance and cost effective computing of medical images.
- As an example it is used to take entire day to make a diagnosis of breast cancer before introducing CUDA in this field.
- The same thing takes only 30 minutes by the use of CUDA. In fact, patients no longer need to wait 24 hours for the results, which will benefit many people.

➜ 3. Computational Sciences

- CUDA also play an important role in the field of raw computational sciences. As an example, after interfacing CUDA with MATLAB provides a great amount of increase in computations.
- In the similar fashion CUDA is used to speedups other computations like matrix mathematics, eigen values computations, or SVD decompositions.

➜ 4. Oil and Natural Resource Exploration

- It is very difficult to construct a 3D view of underground things. It becomes more difficult when the ground is deeply submerged in a sea.
- The possible sources of oil and other natural resources are detected by the scientists by analyzing small sample sources.
- The technologies used for oil and gas exploration use CUDA because the ground reconstruction algorithms are highly parallel and CUDA perfectly suited for the parallel computations used in such algorithms.
- In the current scenario CUDA is used to find oil and other natural resources quicker.

5. Fluid Dynamics

- The movement of fluids, including their interactions as two fluids come into contact with each other are carefully studied through some principles comes under fluid dynamics. In this context, the term "fluid" refers to either liquid or gases.
- It is a macroscopic, statistical approach to analyzing these interactions at a large scale, viewing the fluids as a continuum of matter and generally ignoring the fact that the liquid or gas is composed of individual atoms.
- CUDA based Fluid dynamics simulations have also been created.
- These simulations require a huge number of calculations, and are useful for wing design and other engineering tasks.

6.4 Heterogeneous Architecture

Q. 6.4.1 Explain Heterogeneous system Architecture. (Refer section 6.4) **(5 Marks)**

- In current scenarios heterogeneous compute node contains two multicore processors and two or more many-core GPUs. GPU is used as a co-processor to a CPU instead of using as a standalone platform.
- This way it is clear that GPUs operate in conjunction with a CPU based host.
- The Fig. 6.4.1 depicts this arrangement. This is the basic reason why CPU is called the host and GPU is called the device in GPU computing terms.

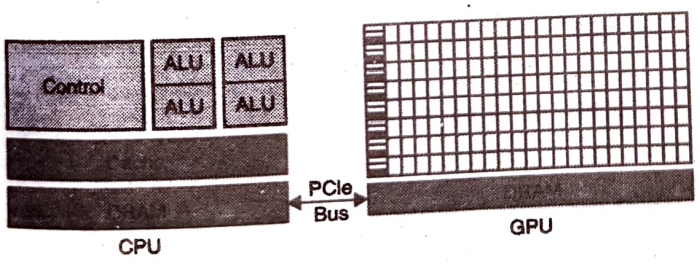


Fig. 6.4.1

- The applications developed for heterogeneous architecture are termed as heterogeneous applications. Such applications consist of two parts namely.
- Host code and Device code. The CPU executes host code and GPU executes device code.
- The environment, code and data are managed by the CPU before loading of compute intensive task on the device.
- GPUs are used to accelerate the execution of data parallelism in program section because it contains a rich amount of data parallelism.
- Hardware accelerator is the component which is separate from the CPU and used to provide acceleration on compute intensive sections of an application.
- GPU is most commonly and widely used hardware accelerator for compute intensive tasks.

6.4.1 Paradigm of Heterogeneous Computing

- CPU computing and GPU computing each having its advantages and shortcoming. In fact GPU computing is not a replacement for CPU computing.
- The control-intensive tasks are better taken care by CPU computing whereas GPU computing is good for data parallel computation - intensive tasks.
- The Fig. 6.4.2 shows about the two dimensions used to differentiate the scope for CPU applications and GPU applications : parallelism level and data size.

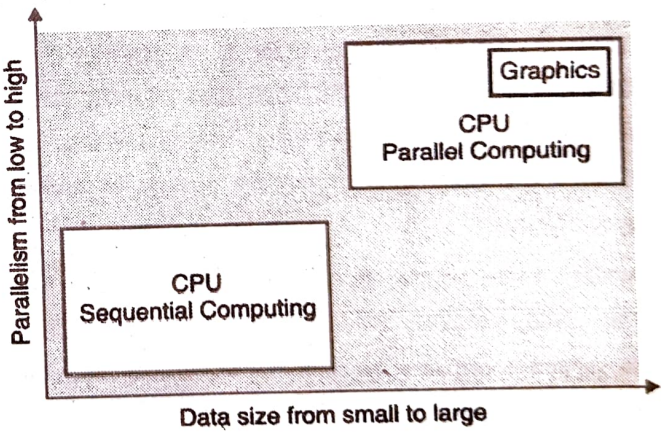


Fig. 6.4.2

- The CPU is a good choice when the problem has a small data size because of providing sophisticated control logic and low level parallelism.
- The ability of CPU to handle complex logic and instruction level parallelism suits in the situation where data size is small. The GPU is suitable if the problem has huge amount of data and has opportunity for massively parallel processing activities.

- The optimal performance can be obtained for applications when executing the sequential parts or task parallel parts with CPU and intensive data parallel parts with the GPU. This is depicted in the Fig. 6.4.3.

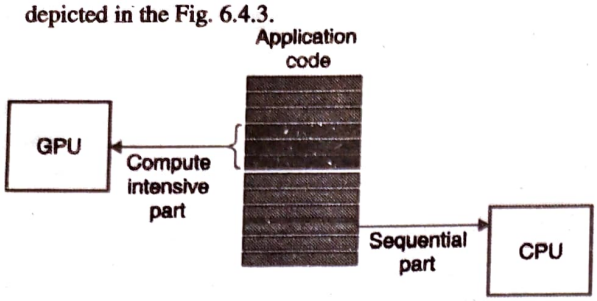


Fig. 6.4.3 : GPU Vs CPU code assignments

- This way we can write the code and ensures that CPU and GPU characteristics complement each other. This way the system combined with GPU and CPU is fully utilized.
- CUDA is the programming model designed by NVIDIA for providing support for joint utilization of GPU and CPU computations.

6.4.1(A) Processor Architecture

Q. 6.4.2 Explain processor Architecture for heterogeneous computing paradigm. (Refer section 6.4.1(A)) **(5 Marks)**

- The Tesla architecture is used for the construction of the CUDA capable GPUs. The applications written can be executed on any card which supports this architecture. Each GPU has slightly different features because of different specifications.
- When a kernel is invoked, each thread block executes on a 'multiprocessor'. This multiprocessor contains the resources to support a certain number of threads. Specifically, each multiprocessor consists of :
  - 8 Scalar Processor cores
  - 2 special function units for transcendental
  - 1 multithreaded instruction unit
  - On-chip shared memory
- One or more thread blocks are assigned to a multiprocessor during the execution of a kernel. The CUDA runtime handles the dynamic scheduling of thread blocks on a group of multiprocessors.
- The scheduler will only assign a thread block to a multiprocessor when enough resources are available to support the thread block.
- Each block is split into SIMD (Single-Instruction Multiple-Data) groups of threads called 'warps'. The SIMD unit creates, manages, schedules and executes 32 threads simultaneously to create a warp.
- Every warp is synchronous, and therefore care must be taken to ensure that certain threads within a warp do not take abnormally longer compared to other threads in that same warp, because the warp will only execute as fast as the slowest thread.
- There are a number of programming hints provided in the CUDA programming guide to help prevent warp divergence.

Interconnect

- The multiple GPU enabled systems are designed because of the following two basic reasons :
- **Problem domain size :** In the case where data sets are too large and unable to fit into the memory of a single GPU.
- **Throughput and efficiency :** The throughput of an application can be increased by processing it in the multiple CPUs concurrently.
- The inter - GPU communication is required to design properly while converting an application into multiple GPU systems. The efficiency of the data transfers among the GPUs depends on how GPUs are connected together within a node or across a cluster. There are two types of connectivity in multi-GPU systems:
- Multiple GPUs connected over the PCIe bus in a single node.
- Multiple GPUs connected over a network switch in a cluster.
- Fig. 6.4.4 shows the topology for a cluster with two compute nodes. In Fig. 6.4.4, GPU1 and GPU2 are connected via the PCIe bus on node1 and GPU3 and GPU4 are connected via the PCIe bus on node2. The two nodes (node1 and node2) are connected to each other through *InfiniBand* Switch.

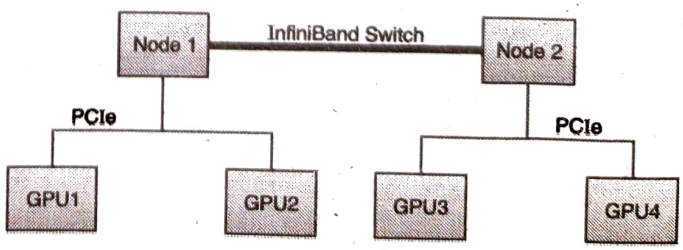


Fig. 6.4.4

Communication

- The communication in the platform is in terms of peer-to-peer communication. The CUDA peer-to-peer API is used to enable direct inter - device communication.
- In this system the devices with compute capability 2.0 and higher can directly access the global memory of any GPU connected with the same root node.
- The CUDA 4.0 or higher version is required for peer-to-peer communications. The CUDA P2P API supports two modes that allows direct communication between GPUs.
- **Peer-to-peer Access :** Directly load and store addresses within a CUDA kernel and across GPUs.
- **Peer-to-peer Transfer :** Directly copy data between GPUs.
- The direct peer-to-peer access is not supported when two GPUs are connected to different root nodes within a system. The CUDA P2P API still can be used to perform peer - to - peer transfer between devices. In this case the driver transparently transfers data through host memory for those transactions rather than directly across the PCIe bus.

Memory Organization

- CUDA uses a segmented memory architecture that allows applications to access data in global, local, shared, constant, and texture memory.
- In terms of organization of memory in GPU devices several levels of memory is considered. The several levels of memory are arranged in the hierarchical fashion and each level has distinct read and write characteristics. Every primitive thread has access to private 'local' memory as well as registers.
- Every thread in a thread block also has access to a unified 'shared memory', shared among all threads for the life of that thread block. Finally, all threads have read/write access to 'global memory', which is located off-chip on the main GDDR memory module which therefore has the largest capacity but is the most costly to interact with.
- There also exists a read-only 'constant' and 'texture' memory, in the same location as the global memory.
- The scope of each memory segments used in the CUDA memory hierarchy is shown in Fig. 6.4.5
- According to the CUDA memory architecture a thread contains its unique registers and the local memory, a block has unique shared memory, and global, constant, and texture memories exist across all blocks.

  The global, constant and texture memory are optimized for different memory usage models.

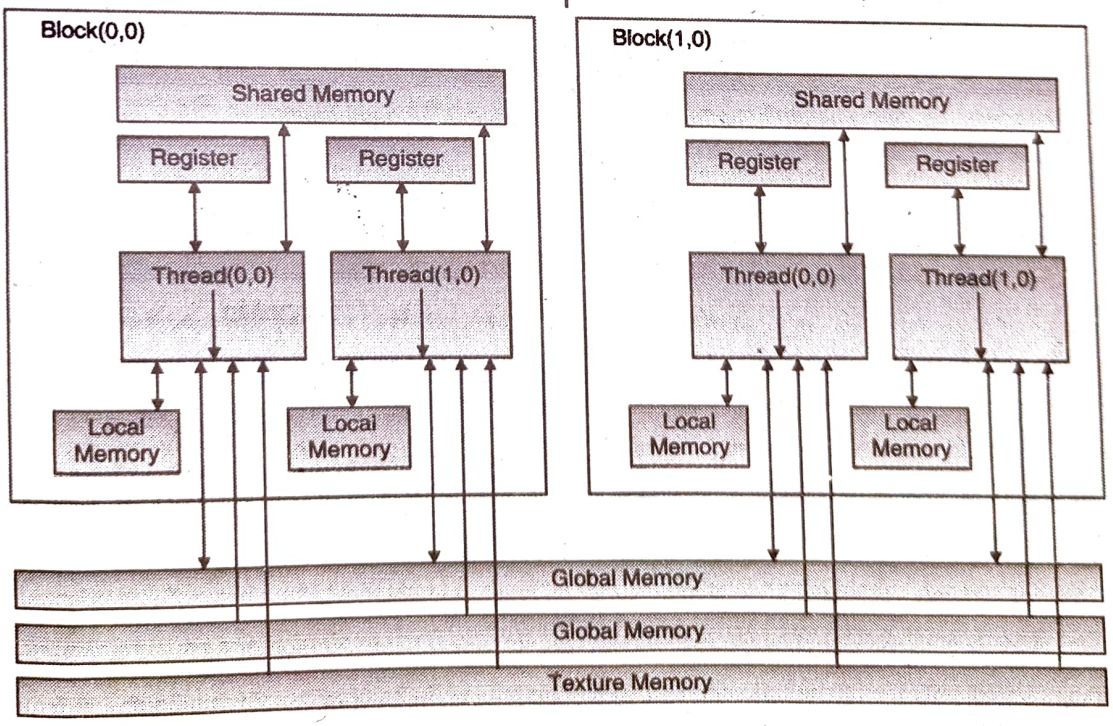


Fig. 6.4.5

- Global memory is not cached, though memory transactions may be 'coalesced' to hide the high memory access latency. These coalescence rules and behaviours are dependent on the particular device used.
- The read-only constant memory resides in the same location as global memory, but this memory may be cached. On a cache hit, regardless of the number of threads reading, the access time is that of a register access for each address being read.
- The read-only texture memory also resides in the same location as global memory, and is also cached. Texture memory differs from constant memory in that its caching policy specifically exploits 2D spatial locality.
- This is due to the use of 'textures' in 3D graphics; the use of 2D images to 'texture' the surface of 3D polygons are frequently read and benefit from caching the texture spatially.

6.4.2 CUDA Programming Models for High Performance Computing Architectures

Q. 6.4.3 Explain CUDA Programming Models for High Performance computing Architectures. (Refer section 6.4.2) **(5 Marks)**

- The model used for programming in a particular architecture acts as a bridge between an application and its implementation on available hardware.
- Fig. 6.4.6 shows a layer of abstraction lies between the program and the programming model implementation.
- The boundary between the program and the programming model implementation is the communication abstraction.
- A compiler or libraries using system calls to access hardware services are used to provide such a communication abstraction.
- The components of the program share information and coordinate their activities by the instructions provided through program for a particular programming model.
- The logical view of specific computer architecture is provided by the programming model. This is typically embodied in a programming environment or language.
- The CUDA programming model share many abstractions with other parallel programming models. The CUDA programming model has some supports for fully utilization of GPU architectures.

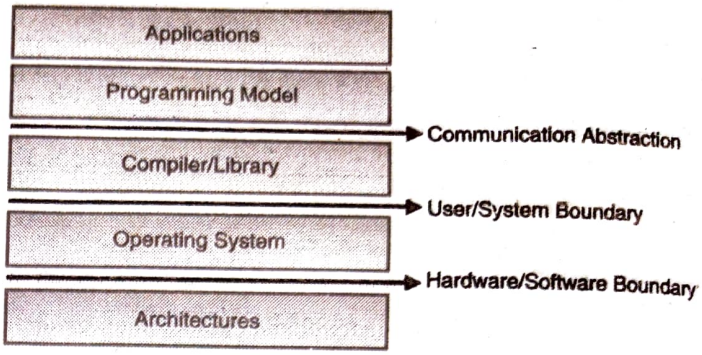


Fig. 6.4.6

- **Anatomy of a CUDA C/C++ Application :** The application for the architecture contains serial and parallel code shown in Fig. 6.4.7.
- The serial code executes in a Host (CPU) thread whereas the parallel code executes in many Device (GPU) threads across multiple processing elements.

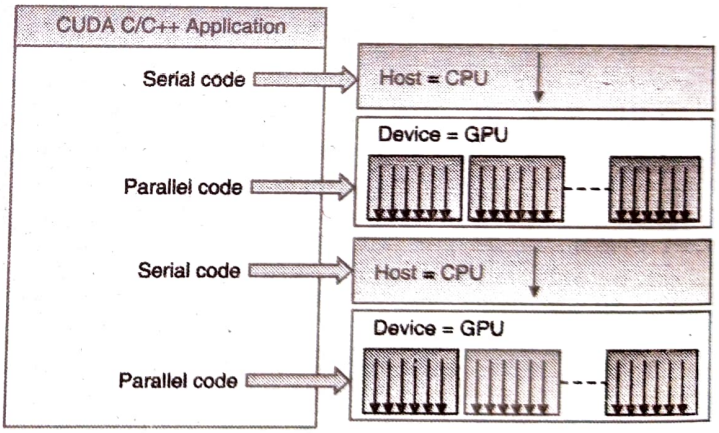


Fig. 6.4.7 : Block of threads

6.4.3 GPU Programming Model

Q. 6.4.4 Write short note on GPU Programming Model. (Refer section 6.4.3) **(5 Marks)**

- As we have discussed the distinction between the host and the device used in the CUDA programming model.
- The code executed in the host side is the part of code executed on the CPU and this will also include the RAM and the hard disk.

- However, the code executed on the device is automatically loaded on the graphic card and run on the latter.
- Another important concept is the kernel; it stands for a function performed on the device and launched from the host.
- The code defined in the kernel will be performed in parallel by an array of threads. The Fig. 6.4.8 summarizes how the GPU programming model works.
- The running program will have source code to run on CPU and code to run on GPU.
- CPU and GPU have separated memories.
- The data is transferred from CPU to GPU to be computed.
- The data output from GPU computation is copied back to CPU memory.

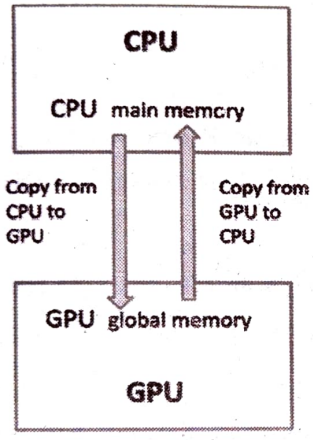


Fig. 6.4.8

6.5 Memory Hierarchy

Q. 6.5.1 Write short note on Memory Hierarchy. (Refer section 6.5) **(4 Marks)**

- There are several threads are considered to be included during execution to provide parallel computing activities.
- The memory references by running threads handled by providing multiple memory accesses, in fact data accessed by these threads done from multiple memory spaces during their executions.
- The Fig. 6.5.1 shows memory hierarchy in CUDA model.
- Each and every thread stored their data in private memory associated with it. These data are called as thread specific data and stored in private memory of a thread termed as local memory.

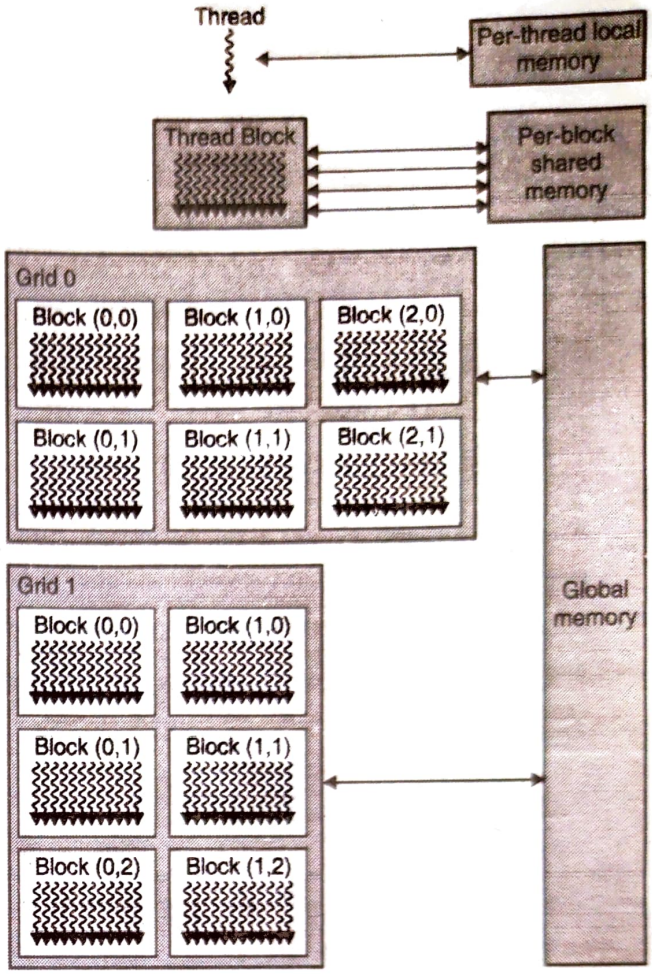


Fig. 6.5.1 : Memory Hierarchy

- In addition to local memory each thread in a block can access common memory to be shared among all threads in the block called as shared memory.
- Global memory space is also available for all threads and all can access the same global memory.
- The two additional read only memory named as : constant and texture memory spaces are also included in the system. These read only memory spaces are accessible by all threads.
- The different types of memory namely global, constant and texture memory spaces are actually optimized to be used for different usage.
- The global, constant, and texture memory spaces are persistent across kernel launches by the same application.
- Fig. 6.5.2 shows the CPU and GPU memory hierarchy and its brief discussion about these memory structures are also described thereafter.

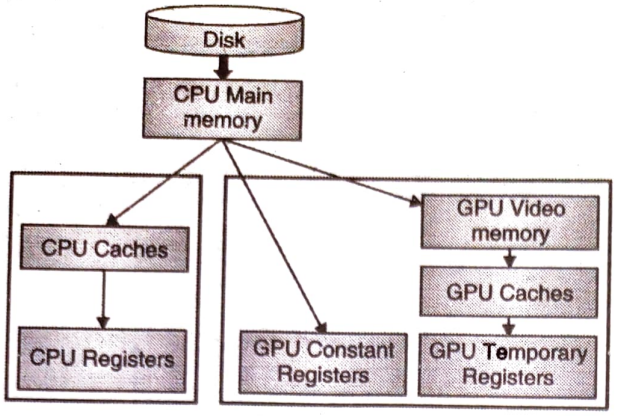


Fig. 6.5.2 : CPU and GPU Memory Hierarchy

- As we have already covered about the multiple types of memory included in the CUDA processor and these memory are available to the programmer and to each thread during their executions.

Register memory

- Scalar variables (e.g., **inti**, **float f**, etc.) are stored in fast registers.
- There are a limited number of registers per thread (although each SIMD process or has 32,768 32-bit registers!) Kernel-declared arrays can also be in registers, but only if the indexing is known at compile-time.
- Registers are private to the thread : no sharing Read/Write, no synchronization necessary.

Local Memory

- Registers can spill into "local memory," which is just (slow) global memory set aside for each thread.
- New GPUs do have caches for local memory.

Shared Memory

- Variables declared with_ shared_ are stored in shared memory, which is very fast. It is, however, limited (48KB per multiprocessor on our GPU).
- Has the lifetime of a block - can be shared between threads in a block.
- Cannot be shared between blocks. Read/Write, must be synchronized with _ syncth reads().
- **Good practice :** Copy global to shared for best use (but only if you can reduce global memory usage).

Global Memory

- Variables declared with_ device_ are stored in global memory, which is very slow. We have lots of global memory, though (up to 2GB on our GPU).
- **cudaMemcpy** reads and writes from the CPU to GPU memory.
- Lifetime of the program, but sometimes tricky because you can't synchronize across blocks. If you need synchronization, you must have multiple kernel in vocations.
- Global memory is declared on the host process using **cuda Malloc** and freed in the host process using **cuda Free**. Pointers are passed from the CPU to the GPU. Reducing global accesses is a goal, but an art form. Judicious use of shared memory is helpful.

Constant Memory

- Variables that are declared with the **__constant__** attribute are declared in constant memory (which is part of global memory).
- There is only a limited amount of constant memory (64 KB per kernel), but it is much faster than regular global memory, because it is always cached.
- Constant memory can be written to by the host process using the **cudaMemcpy To Symbol** function and read-from using the cudaMemcpy From Symbol function. Lifetime of program, but can only be changed by the CPU.
- The different types of memory space available during operations are concluded as follows :

Global	Large address space, high latency (100xslower than cache)
Shared	Small, low latency
Texture/Constant	Read-only
Registers/Local	Only available to one thread

The comparison of CPU memory model and GPU memory model is given in Table 6.5.1.

Table 6.5.1

CPU Memory Model	GPU Memory Model
– At any program point	– Much more restricted memory access.
– Allocate/free local or global memory	– Allocate/free memory only before computation
– Random memory access	– Limited memory access during computation (kernel)
– Registers o Read/write	– Registers o Read/write
– Local memory o Read/write to stack	– Local memory o Read/write
– Global memory	– Shared Memory
– Read/write to heap	– Only available in GPGPU not Graphics pipeline
Disk – Read/write to disk	– Global memory o Read-only during computation o Write-only at end of computation (pre -computed address) o Read/write in GPGPU world only.
	– Virtual Memory o Does not exist
	– Disk access o Does not exist

6.5.1 CUDA Memory and Cache Architecture

Q. 6.5.2 Explain CUDA Memory and cache architecture. (Refer section 6.5.1) **(4 Marks)**

- The understanding of the basic memory architecture of the system for which we are working to create high – performance applications is always required.
- As far as desktop systems are concerned most of the systems consist of large amounts of system memory connected with a single CPU.
- These systems may contain two or three levels or fully coherent cache.
- While developing applications for CUDA systems one has to understand the memory hierarchy of the CUDA capable devices.

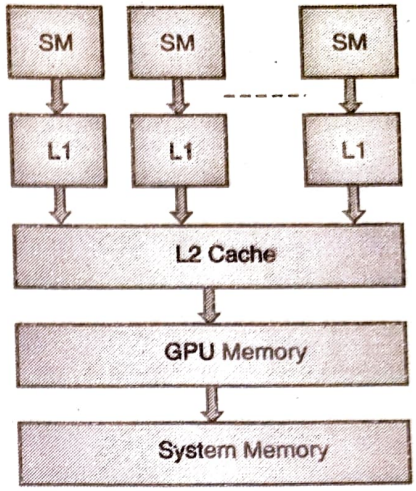


Fig. 6.5.3 : Basic Memory Hierarchy

- The Fig. 6.5.3 shows the memory structure in a modern system using NVIDIA's Fermi architecture. The GPU has on- board memory of its own called as device memory.
- This can be ranges from 768 megabytes all the way up to 6 GBs of GDDR5 on mid to high - end workstations.
- On the other hand system memory for modern computers ranges from 6 GBs to 64 GBs or more of DDR3 on high – end workstations.
- In conclusion GPUs do not have much memory as compare to CPUs but the memory bandwidth for GPUs is usually substantially higher than that of even the highest end Intel or AMD CPUs.
- All modern CUDA capable cards (Fermi architecture and later) have a fully coherent L2 Cache.
- As with memory, the GPU's L2 cache is much smaller than a typical CPU's L2 or L3 cache, but has much higher bandwidth available.
- Finally, the L1 cache onboard a GPU is smaller than L1 cache in a CPU, but again, it has much higher bandwidth.

6.6 Introduction to CUDA C

Q. 6.6.1 Write short note on CUDA C. (Refer section 6.6) **(4 Marks)**

- CUDA basically provides a parallel computing platform and it has its own style of programming model for parallel application developments.

- CUDA contains a small set of extensions of C programming language. Parallel applications can be developed using CUDA on NVIDIA GPUs.
- CUDA is used to develop applications for various devices like, tablet devices, laptops, desktops, and high performance cluster systems etc.
- The existing C programming development tools have been extended to edit, debug and for overall programming development under different devices.

CUDA Programming Models for High Performance

Computing Architectures

- The model used for programming in a particular architecture acts as a bridge between an application and its implementation on available hardware.
- Fig. 6.6.1 shows a layer of abstraction lies between the program and the programming model implementation.
- The boundary between the program and the programming model implementation is the communication abstraction.
- A compiler or libraries using system calls to access hardware services are used to provide such a communication abstraction.
- The components of the program share information and coordinate their activities by the instructions provided through program for a particular programming model.
- The logical view of specific computer architecture is provided by the programming model. This is typically embodied in a programming environment or language.

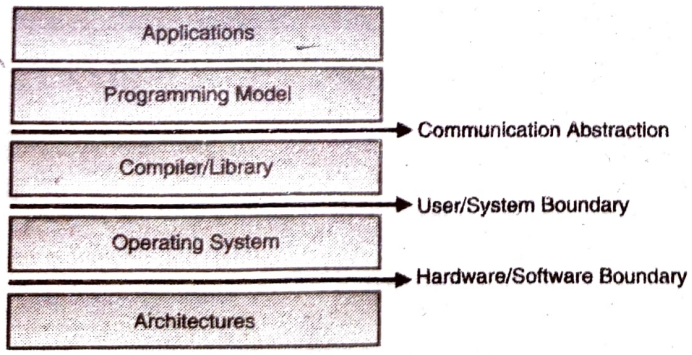


Fig. 6.6.1

- The CUDA programming model share many abstractions with other parallel programming models. The CUDA programming model has some supports for fully utilization of GPU architectures.
- **Anatomy of a CUDA C/C++ Application :** The application for the architecture contains serial and parallel code shown in Fig. 6.6.2. The serial code executes in a Host (CPU) thread whereas the parallel code executes in many Device (GPU) threads across multiple processing elements.

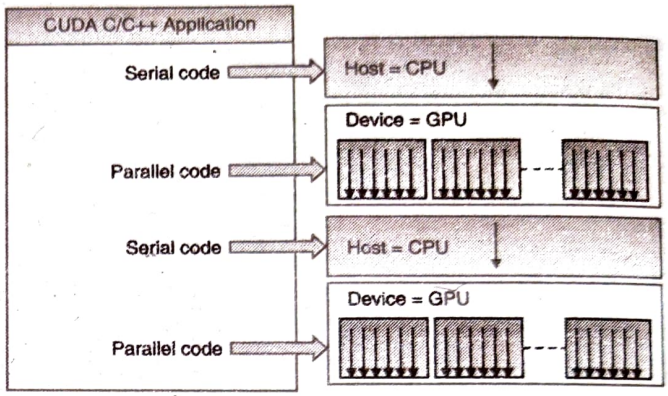


Fig. 6.6.2 : Block of threads

CUDA Programming Structure

- The CUDA programming model provides approaches through which applications are executed on heterogeneous computing systems.
- This is possible by using the simple annotation codes with small extensions to the C programming language.
- The heterogeneous computing environment consists of Host (CPU and its memory) and device (GPU and its memory) so that different activities can be performed separately by CPU and GPU.
- The code that executes on the GPU device is considered as a kernel, and it is a key component of the CUDA programming model.
- As a programmer we write kernel code as a sequential program but in the background CUDA manages scheduling programmer - written kernels on GPU threads.
- In the CUDA programming model the GPU computation performed on the GPU is overlapped with the host - device communication because of its asynchronous nature.
- Typically a CUDA program contains serial code complemented by parallel code as Fig. 6.6.3.
- The host provides execution facilities for serial code (as well as task parallel code) whereas the GPU executes the parallel code.

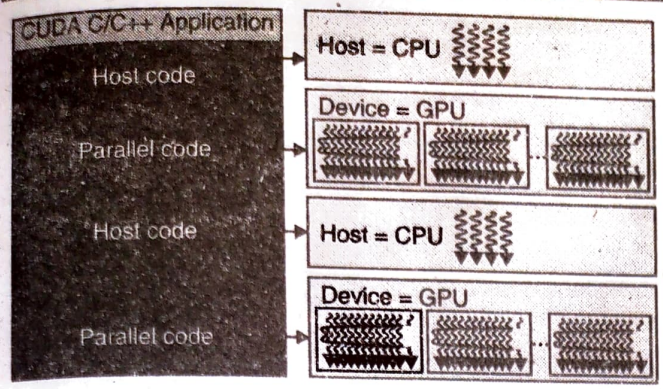


Fig. 6.6.3

- The device code and host code are written separately using CUDA C and ANSI C respectively. In fact one can use single source file to put all codes or multiple source files can be used to build applications or libraries.
- The executable code for both host and device are generated by the NVIDIA C Compiler (NVCC).
- The following pattern is followed typically in processing of a CUDA program :
  1. First copy data from CPU memory to GPU memory.
  2. Kernel is invoked to execute on the data stored in GPU memory.
  3. Now data is copied back from GPU memory to CPU memory.
- In this way it is clearly described that CUDA programming is for heterogeneous platform and that is the reason CUDA programming model is a heterogeneous model.
- In CUDA programming both CPU and GPU are used for performing several activities.
- As we have already discussed that in CUDA environment CPU and its memory is called as host whereas GPU and its memory is called as a device.
- The code execute on the host can manage memory on CPU and GPU and it also launches the kernel to be executed on the device. These kernels are executed in parallel by many GPU threads.
- A typical sequence of operations for a CUDA program is listed below and it is based on the heterogeneous nature of the CUDA programming model.
  - Host and device memory declaration and allocation.
  - Host data initialization.
  - Transfer data from host to device.
  - Start executions of one or more kernels.
  - Transfer results from device to the host.

6.6.1 CUDA Kernels

- While dealing with programming in CUDA environment we should not get confuse with the word kernel it is not actually operating system kernel.
- A Kernel in CUDA terminology is just a name given for a function that runs on the GPU.
- In CUDA environment C programming language is extended, so that the programmer is allowed to define C functions and these functions are actually called as kernels.
- When a kernel is called it is executed N times in parallel by N different threads, whereas a regular C function executes only once.

A kernel can be invoked by using the following syntax:

```
kernel_function<<<num_blocks,
num_threads>>>(param1, param2, .)
```

- There are two important parameters **num_blocks** and **num_threads** used in the kernel function invocation.
- These parameters can be either variables or literal values, but most of the time we consider to use variables because we refer it later in the program.
- Apart from these two parameters other parameters can also be required but we will stick up to these two parameters for time being.
- The number of threads one intends to launch into the kernel is represented by **num_threads** parameter.
- The other parameter is used to represent number of blocks of threads can be used for executions.
- The **__global__** declaration specifier is used to define a kernel and the new execution configuration syntax **<<<...>>>** is used to specify the number of CUDA threads that execute that kernel for a given kernel call.
- A unique thread ID is given to each thread that executes the kernel. The built-in variable **threadidx** is used to access thread ID within the kernel.

- For example, the following code segment adds two vectors A and B in parallel and store the resultant value in third vector C.
- In this example, each thread that executes the kernel ArrSum() performs one pair – wise summation.

```
// Kernel definition
__global__ void ArrSum (float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    ...
    // Kernel invocation with N threads
     ArrSum<<<1, N>>>(A, B, C);
    ...
}
```

6.6.1(A) Hello World Program in CUDA

- The most common hello world program using C is shown below, it executes entirely on the host.
- The name host is used to refer CPU and the system memory. On the other side GPU and its memory is called as device.

```
#include <stdio.h>
int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

- This program can be compiled using standard C compiler like gcc under Linux environment and executed thereafter by running the created executed file.

```
gcc -o hello hello.c
./hello
```

- Now let's add some statements in the above program code so that it can be executed on the device (or GPU).
- The function which executes on the device is called kernel, and the following segment of code is referred as an example of kernel code in CUDA.

```
#include <iostream>
__global__ voiddemo_kernel(void)
{
}
int main( void )
{
    demo_kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- There are two additional steps added in this program as compare to the simple hello world program:
  1. A function named as demo_kernel() with qualifier __global__
  2. In the main function demo_kernel() function is called using demo_kernel<<<1,1>>>()
- The normal program is compiled using the standard C compiler like GNU gcc under Linux whereas the NVIDIA compiler nvcc is used to compile a program with extended C statement added.

```
nvcc -o hello_cuda  hello_cuda.cu
./hello_cuda
```

6.6.1(B) Parameter Passing in Kernel

Q. 6.6.2 How parameter passing takes place in CUDA kernel, explain with example code. (Refer section 6.6.1(B)) **(4 Marks)**

- The number of parameters can be passed to the kernel in the similar fashion as normal function call in standard C programming.
- Consider the following program which has included the parameter passing in the kernel call.

```
__global__ void sqrNum( int num, int* result )
{
*result = num*num;
}
int main( void )
{
    int c;
    int *dev_c;
    cudaMalloc( (void**)&dev_c, sizeof(int) );
    sqrNum<<<1,1>>>( 2, dev_c );
    cudaMemcpy( &c, dev_c, sizeof(int),
    cudaMemcpyDeviceToHost );
    printf( "sqr of 2 is = %d\n", c );
    cudaFree( dev_c );
    return 0;
}
```

- There are two notable points related to the above program, first is we can pass parameters to the kernel like any other C functions and the second important point is related to memory allocation. It is required to allocate memory to do anything useful on a device.
- The angle - bracket syntax is used to pass parameters to the kernel very similar to other C language function calls. The complexity related to the copy of parameters from host to device is taken care by the CUDA runtime system.

6.6.2 Organizing Threads

- A kernel function is launched by the host and its execution is moved to a device in which a large number of threads are generated.
- Each generated thread on the device executes the statements specified in the kernel function.
- The crucial part in CUDA programming is to know about the organization of threads so that these threads can execute efficiently in parallel.
- A thread hierarchy abstraction is supported by the CUDA so that programmer can enable and organize threads.
- This hierarchy is actually two level hierarchies and it is decomposed into blocks of threads and grids of blocks as shown in Fig. 6.6.4.

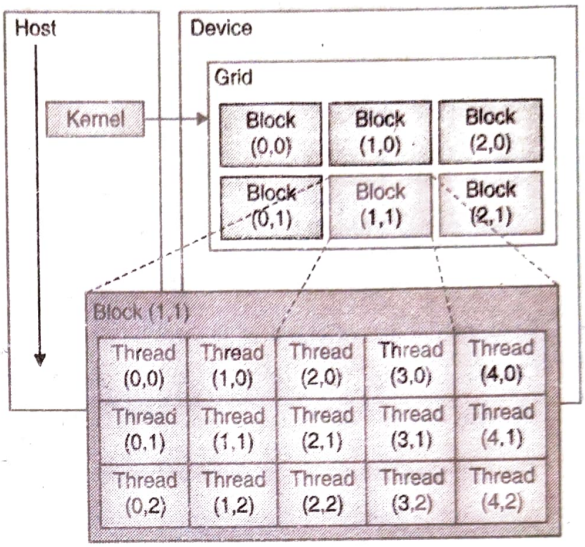


Fig. 6.6.4

- A grid is the collection of all threads spawned by a single kernel launch. The same global memory space is shared among all the threads in a grid. In terms of thread blocks, a grid is considered as it is made up of many thread blocks.
- A group of threads cooperate with each other using the two techniques named as ***block – local synchronization*** and ***block – local shared memory*** is termed as a thread block.
- This means a block includes a number of threads cooperate with each other through specified approaches of communications.
- In this description it is clear that threads from the different blocks cannot cooperate with each other.

Coordinate variables : blockIdx and threadIdx

- These are the two unique coordinates used by threads to distinguish themselves from each other. The CUDA runtime assigns the coordinate variables **blockIdx** and **threadIdx** to each thread when a kernel function is executed.
- In the program design a programmer can assign a portion of the data to different threads based on the coordinates.
- The type of these coordinate variables is unit 3, which is a CUDA built – in vector type.
- The unit 3 type is derived from the basic integer type. This is actually a structure which contains three unsigned integers as members.

- The fields x, y and z are used to access the first, second and third components of the structure respectively.

```
blockIdx.x
blockIdx.y
blockIdx.z
```

```
threadIdx.x
threadIdx.y
threadIdx.z
```

- Grids and blocks are organized in three dimensions by the CUDA. The Fig. 6.6.4 describes an example of thread hierarchy structure with a 2D grid containing 2D blocks.
- The following two built - in variables are used to specify dimensions of a grid and a block.
- **Block Dim (block dimension, measured in threads)**
  grid Dim (grid dimension, measured in blocks).
- The type of these variables is dim3, which is an integer vector type based on unit 3 that is used for dimension specification.
- When a variable is defined of type dim3 and any unspecified variable is automatically initialized to 1.
- Each component in a variable of type dim3 is accessible through its x, y, and z fields, respectively, as shown in the following example:

```
blockDim.x
blockDim.y
blockDim.z
```

Syllabus Topic : Launching a CUDA Kernel

6.6.3 Launching a CUDA Kernel

- The familiar C programming syntax for function call is extended in CUDA for kernel call. The syntax of the kernel call is shown below. The kernel's execution context is specified within triple angle brackets.

```
kernel_name <<<grid, block>>>(argument list);
```

- It is already described in the previous section, that the CUDA programming model exposes thread hierarchy.
- It is possible to specify how threads are scheduled to be executed on the GPU using the execution configuration. There are two values specified in the execution configuration.
- The first value is the grid dimension which is used to specify the number of blocks to launch. The second value in the execution configuration is used to specify the block dimension which is describes the the number of threads within each block.
- The following two things can be configured using the grid and block dimension specification in the execution configuration :
  (i) A kernel will be executed by the total number of thread.
  (ii) What layout to be employed for a kernel ?
- Threads communications are based on the blocks they belong; in fact threads included in the same block can communicate with each other whereas threads in different blocks cannot cooperate.
- The communication approaches among the threads belonging to different blocks are tough and time consuming. In the consideration situation for a given problem one can organize threads using the different grid and block layouts.
- As an example we can have a group of 8 elements in each block and can launch 4 blocks if the number of data elements is 32 shown below.

```
kernel_name<<<4, 8>>>( argument list);
```

- The layout of threads is illustrated in Fig. 6.6.5 for the above configuration.

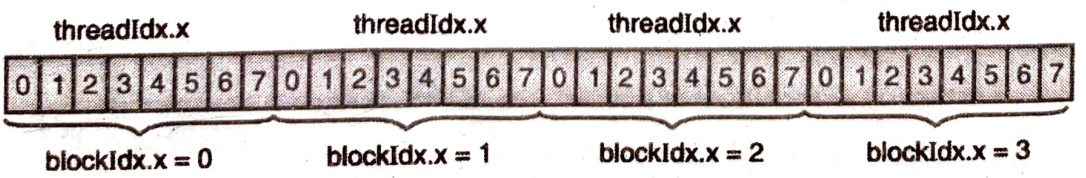


Fig. 6.6.5

- The built – in variables **blockIdx.x** and **threadIdx.x** are used for identifying a unique thread in the grid and establishing a mapping between threads and the data elements.
- It is possible because data is stored linearly in a global memory.

☞ **Example kernel - call statements**

```
kernel_name<<<1, 32>>>(argument list);
```

- This call is invoked when all 32 elements are grouped into one block.

```
kernel_name<<<32, 1>>>(argument list);
```

- This call is required in the case where each block will have one element and in total 32 blocks are needed.

☞ **About host and device synchronization**

- The separate threads created in host and on device but the kernel call runs in asynchronous nature with respect to the host thread.
- The control returns immediately to the host after kernel call is invoked.
- The function shown below is used in a case where it is required to provide waiting status to the host application for all kernels to complete executions.

```
cudaError_t cudaDeviceSynchronize(void);
```

- Some of the CUDA APIs automatically provides implicit synchronization between the host and the device.
- For example data copy is performed between the host and the device using ***cudaMemcpy()*** function.
- When this function is used implicit synchronization is handled and host application is forced to wait for the completion of data copy operation.

```
cudaError_t cudaMemcpy(void* dst, const void* src,
size_t count, cudaMemcpyKind kind);
```

- Once all previous kernel calls have completed this function starts to perform copy operation. After completion of the copy operation control immediately transfer back to the host side.

Syllabus Topic : Writing a CUDA Kernel

6.6.4 Writing a CUDA Kernel

- As we have already discussed about a kernel function that it executes on the device side only. A kernel function definition contains the computation for a single thread and the data access for that thread.
- When a kernel is called, many CUDA threads are created to perform the same computation in parallel.
- A kernel is defined using the **__global__** declaration specification as shown below. One has to remember that a kernel function must have **void** as its return type.

```
__global__ voidkernel_name(argument list);
```

- Table 6.6.1 shows brief about function type qualifiers used in CUDA C programming.
- These function type qualifiers are used to specify whether a function executes on the host or on the device.
- It also indicates about a function is callable from the host or from the device.

Table 6.6.1

Qualifiers	Execution	Callable	Notes
__global__	On the device	From the host	Return type must be void
__host__	On the host	From the host only	It is possible to omit return type
__device__	On the device	From the device only	

- When it is required to compile a function for both host and the device the qualifiers __device__ and __host__ are used together.
- In conclusion the following points are considered as restrictions in a CUDA kernel functions:
  - CUDA kernel function must have void as its return type.
  - It can only access the device memory.
  - Automatically shows an asynchronous behavior
  - Kernel function does not support the variable number of arguments.

- It does not provide support for the static variables and the function pointers.

- For example, consider an example to perform summation of two arrays. The code to perform array addition using C is shown below :

```
void arraysSumOnCPU(float *A, float *B, float *C, const int Num)
{
        for (int i = 0; i < Num; i++)
                C[i] = A[i] + B[i];
}
```

- This is a sequential code and it iterates Num times, the kernel function will be written after removal of the loop as shown below :

```
__global__ voidarraysSumOnGPU(float *A, float *B, float *C)
{
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}
```

- The different between the C function and the kernel is that the loop required for iterations is not present in the kernel function.

- This is because the built – in thread coordinate variables are used to replace the array index, and no reference required for Num because it is implicitly defined by only Num threads.

- For example, we can invoke the kernel with 32 threads for an array with the length of 32 elements as below:

```
arraysSumOnGPU<<<1,32>>>(float *A, float *B, float *C);
```

Syllabus Topic : Manage Communication and Synchronization

6.6.5 Manage Communication and Synchronization

- CUDA API provides a hardware thread – barrier function named as **syncthreads()** for synchronization purpose among threads. This function acts as a synchronization point.

- This function is implemented at hardware level because threads are scheduled in hardware. The threads belonging to one kernel will wait at the synchronization point until all of the threads have reached at this point.

- If need arises to provide communication among threads it is handled using the shared - memory used per block. This means thread synchronization is provided only at thread block level.

- Threads of a thread block must execute on the same processor because threads belonging to one thread block may communicate with each other. That is why thread block is guaranteed to execute on one processor.

Syllabus Topic : Writing a CUDA Kernel

6.6.6 Manage GPU Memory

Q. 6.6.3 How to manage GPU memory? (Refer section 6.6.6) (4 Marks)

- A large global device memory is available and accessible by all the multiprocessors for both gather and scatter operations. This memory does not provide caching and hence it is relatively slow memory.

- As compare to device memory shared memory is fast and it takes same amount of time as required to access registers.

- Shared memory is also called as parallel data cache (PDC) and is local to each multiprocessor unlike device memory. Shared memory is divided into many parts and it allows more efficient local synchronization.

- Shared memory access is restricted up to a thread block; this means each thread block in a multiprocessor accesses its own part of shared memory.

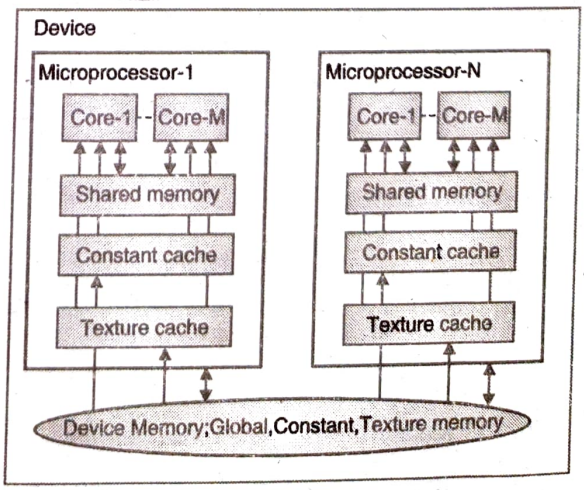


Fig. 6.6.6 : CUDA memory model

- This clearly indicates that a part of shared memory associated with one thread block will not be accessible by any other thread block in the same multiprocessor or from other multiprocessor.
- Shared memory space is limited and should be used efficiently. In most of the time system supports shared memory up to only 16 KB. Qualifiers **__shared__** and **__device__** are used to declare variables in shared memory and in global memory respectively.
- All threads within a thread block use shared memory for both read and write operations. The read operation is performed quickly because local read – only cache is included in each multiprocessor.
- These are constant cache and texture cache memories. Each thread may contain thread specific data in their own local memory. Normally local variables of the kernel functions are allocated here. Sometimes they are allocated on global memory.

Syllabus Topic : Parallel Programming in CUDA- C

6.7 Parallel Programming in CUDA- C

- In this section the parallel programming using CUDA is described with example codes.
- The kernel function is presented with the specific comment to write parallel code for the given problem.

6.7.1 Parallel Programming Pattern in CUDA

- The parallel programming pattern used by CUDA is grids or tiling pattern.
- In this approach the overall problem is brake down into smaller parts by the programmer. For example, the problem of climate modeling is broken down into hundreds or thousands of blocks.
- Thereafter each block is assigned to one of the thousands of the processing elements present in the machine. This type of parallel decomposition provides advantages in terms of scaling.
- A GPU is similar to symmetric multiprocessing system on a single processor in the way of its working. In a GPU each Symmetric Multiprocessing unit (SMs) is considered as a full processor in its own rights and has the capability to run multiple blocks of threads. Typically each block may contain 256 or 512 threads to perform the task.
- The CUDA provided model is the simple two – dimensional grid model. For a significant number of problems this is entirely sufficient.
- If you have a linear distribution of work within a single block, you have an ideal decomposition into CUDA blocks.
- As we can assign up to sixteen blocks per SM and we can have up to 16 SMs (32 on some GPUs), any number of blocks of 256 or larger is fine.
- In practice, we'd like to limit the number of elements within the block to 128, 256, or 512, so this in itself may drive much larger numbers of blocks with a typical dataset.

6.7.2 A Simple Example

- Before writing proper parallel programs we will start with writing simple programs but using the CUDA programming constructs.
- Let us write a simple kernel which will add two integer numbers.

```
__global__ void add(int*n1,int*n2,int*result)
{
    *result = *n1 + *n2;
}
```

- Here the keyword **__global__** is used to indicate that the function **add()** will run on the device as a kernel when it is called from the host code.
- In this program we have used pointers and because this function will run on the device these pointers must point to device memory.
- As we have discussed host and device memory are distinct entities because device pointer points to the GPU memory whereas host pointer points to the CPU memory.

- Here one has to note that device pointers may be passed to and from the host code but may not be dereferenced from the host code.
- There are three basic CUDA APIs, **cudaMalloc()**, **cudaFree()** and **cudaMemcpy()** provided by the platform for dealing with the device memory.
- These APIs are similar to their C equivalents malloc(), free() and memcpy().

The main function for the simple example is shown below :

```
int main( void )
{
    int n1, n2, result;                    // host copies of variables
    int *dev_n1, *dev_n2, *dev_result;
                                           //device copies of variables
    int size = sizeof( int );              //space created for an integer

    //memory allocation for device copies of variables
    cudaMalloc( (void**)&dev_n1, size);
    cudaMalloc( (void**)&dev_n2, size);
    cudaMalloc( (void**)&dev_result, size);

    n1 = 5;
    n2 = 6;

    // copy inputs to device
cudaMemcpy( dev_n1, &n1, cudaMemcpyHostToDevice);
cudaMemcpy( dev_n2, &n2, cudaMemcpyHostToDevice);

    // launch function add() as a kernel on GPU, and also passing parameters
    add<<<1,1 >>> (dev_n1, dev_n2, dev_result);

    //now copy result back from device to host copy of result variable
cudaMemcpy(&result, dev_result, cudaMemcpyDeviceToHost);

    cudaFree(dev_n1);
    cudaFree(dev_n2);
    cudaFree(dev_result);
return 0;
}
```

- Suppose the name of this program is add_nums.cu, when this program is compiled and executed the synchronous and asynchronous nature can be described using the diagrammatic representations.

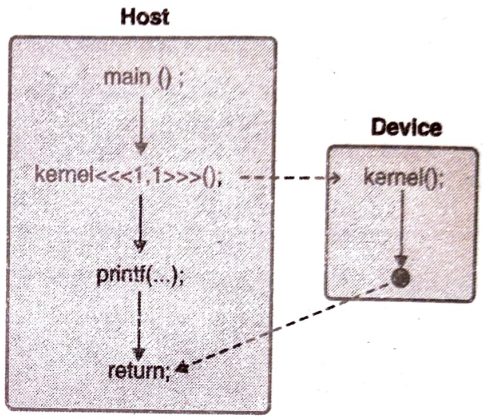


Fig. 6.7.1 : kernel invocation is asynchronous

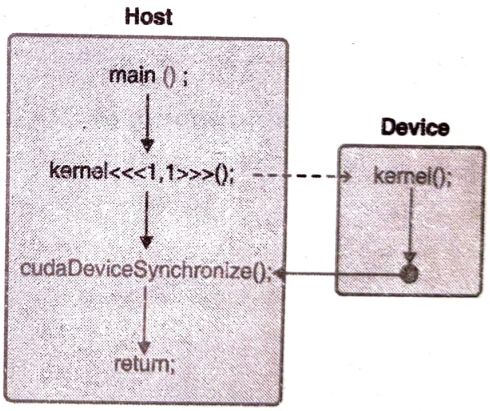
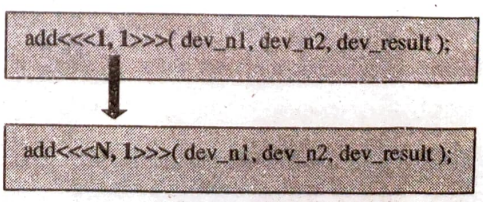


Fig. 6.7.2 : kernel invocation is synchronous

Parallel execution

- How we can execute code in parallel on the device because GPU computing is meant for massively parallel computing.
- The parameters used within the triple angle bracket are used for this purpose.



- Here this modification in first parameter from 1 to N indicates that add() function executed N times instead of once.

6.7.3 Vector Addition in Parallel

- Now we will deal with the parallel implementation of the vector addition program in CUDA because at this stage we dealt with the function **add()** running in parallel.
- In CUDA terminology each parallel invocation of **add()** is called as a block. The built – in variable **blockIdx.x** is used by the kernel to refer its block's index.
- Each block adds a value from the vectors n1[] and n2[] and the result is stored in result[] as shown below :

```
__global__ voidadd(int*n1,int* n2,int* result )
{
    result[blockIdx.x] =n1[blockIdx.x] +n2[blockIdx.x];
}
```

- The variable blockIdx.x is used to index arrays and each block handles different indices. The contents of different blocks may look like shown in the Fig. 6.7.3 diagrammatic description when runs in parallel on the device.

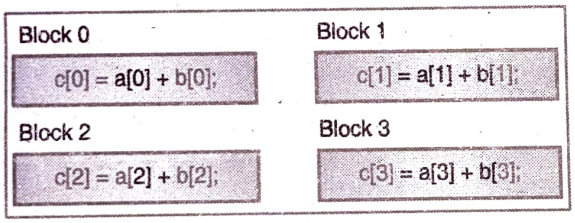


Fig. 6.7.3

- The main function for the parallel addition program is written below. It contains memory creation statements and kernel launching statement with N number of blocks.

```
#define N 512
int main( void )
{
    int*n1, *n2,*result; // host copies
    int*dev_n1, *dev_n2, *dev_result;// device copies
    int size =N *sizeof(int );
                    // required memory space for 512 integer elements
// allocate n1, n2, result on device
cudaMalloc( (void**)&dev_n1, size );
cudaMalloc( (void**)&dev_n2, size );
cudaMalloc( (void**)&dev_result, size );

//allocate n1, n2, result on host
n1 = (int*)malloc( size );
n2 = (int*)malloc( size );
result  = (int*)malloc( size );

//get N random numbers and initialize n1 and n2
getRandomInt(n1, N);
getRandomInt(n2, N);

// copy inputs values from host to device
cudaMemcpy( dev_n1, n1, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_n2, n2, size, cudaMemcpyHostToDevice );

// Now launch add() kernel with N parallel blocks
add<<<N, 1 >>>( dev_n1, dev_n2, dev_result );

// copy device result back to host copy of result
cudaMemcpy( result, dev_result, size, cudaMemcpyDeviceToHost );

//free host memory
free( n1 ); free( n2 ); free( result );

//free device memory
cudaFree( dev_n1 );
cudaFree( dev_n2 );
cudaFree( dev_result );
```

```
return 0;
}
void getRandomInt(int *n1, int N)
{
// statements to generate N random numbers and store in vector n1
}
```

6.7.4 Vector Addition using Parallel Threads

- Now we will rewrite the previous program to use parallel threads instead of parallel blocks because a block can be split into parallel threads.
- In this example we use built – in variable **threadIdx.x** instead of **blockIdx.x** in the kernel add().

```
__global__ voidadd(int*n1,int* n2,int* result )
{
    result[threadIdx.x] =n1[threadIdx.x] +n2[threadIdx.x];
}
```

The main function will also need one change as well.

```
#define N 512
int main( void )
{
    int*n1, *n2,*result; // host copies
    int*dev_n1,*dev_n2, *dev_result;// device copies
    int size =N *sizeof(int );
                // require memory space for 512 integer elements

// allocate n1, n2, result on device
cudaMalloc( (void**)&dev_n1, size );
cudaMalloc( (void**)&dev_n2, size );
cudaMalloc( (void**)&dev_result, size );

//allocate n1, n2, result on host
n1 = (int*)malloc( size );
n2 = (int*)malloc( size );
result = (int*)malloc( size );

//get N random numbers and initialize n1 and n2
getRandomInt(n1, N);
getRandomInt(n2, N);

// copy inputs values from host to device
cudaMemcpy( dev_n1, n1, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_n2, n2, size, cudaMemcpyHostToDevice );

// Now launch add() kernel with N parallel threads
add<<<1,N >>>( dev_n1, dev_n2, dev_result );

// copy device result back to host copy of result
cudaMemcpy( result, dev_result, size, cudaMemcpyDeviceToHost );

//free host memory
free( n1 ); free( n2 ); free( result );

//free device memory
cudaFree( dev_n1 );
cudaFree( dev_n2 );
cudaFree( dev_result );
return 0;
}
void getRandomInt(int *n1, int N)
{
// statements to generate N random numbers and store in vector n1
}
```

6.7.5 Parallel Dot Product Using CUDA

- Before writing the kernel for dot product we will have a bit discussion on dot product of two vectors in general.
- After performing dot product the resultant value is a number, it is scalar not a vector. The notation to represent the dot product is using a central dot for example a.b. This means the dot product of 'a' and 'b' is represented as **a.b**.

The calculation for the dot product is given as follows:

$$a.b = |a| \times |b| \times \cos(\theta)$$

It is based on the representation as shown in Fig. 6.7.4.

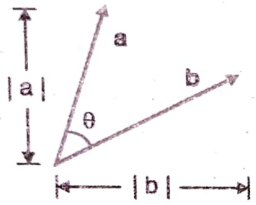


Fig. 6.7.4

Where :

|a| is the magnitude (length) of vector **a**

|b| is the magnitude (length) of vector **b**

θ is the angle between **a** and **b**

- So we multiply the length of **a** times the length of **b**, then multiply by the cosine of the angle between **a** and **b**
- Another way we can calculate the dot product of two vectors using the following approach:

$$a \cdot b = a_x \times b_x + a_y \times b_y$$

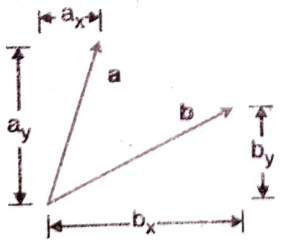


Fig. 6.7.5

So we multiply the x's, multiply the y's, then add.

- This clearly indicates that dot product is a reduction from vectors to a scalar not like vector addition.
- The diagrammatic representation is shown below along with the steps performed for dot product of two vectors let us say 'a' and 'b'.

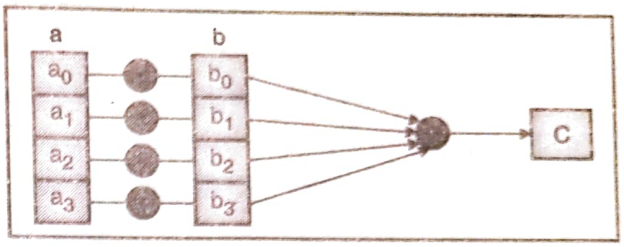


Fig. 6.7.6 : dot product of vectors a and b

Steps for dot product calculation of vectors a and b

$$c = a \cdot b$$

$$c = (a_0, a_1, a_2, a_3) \cdot (b_0, b_1, b_2, b_3)$$

$$c = a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3$$

- The number of parallel threads can be created and perform the computation on pairwise products as shown in Fig. 6.7.7:

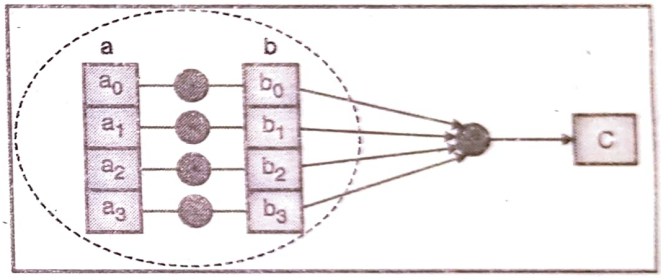


Fig. 6.7.7

- This way it is clear that we can start writing a CUDA kernel by performing just a pairwise computation as listed in the following code segment.

```
__global__ voiddot(int*a,int*b,int*result)
{
// Each thread computes a pairwise product
inttemp = a[threadIdx.x] * b[threadIdx.x];
.....
```

- The final sum can be performed only when we share the data between threads this is described in the Fig. 6.7.8.
- As we know each and every thread contains its private data known as thread specific data so private data of these threads cannot be shared. We need to discuss the task about sharing of data among threads.

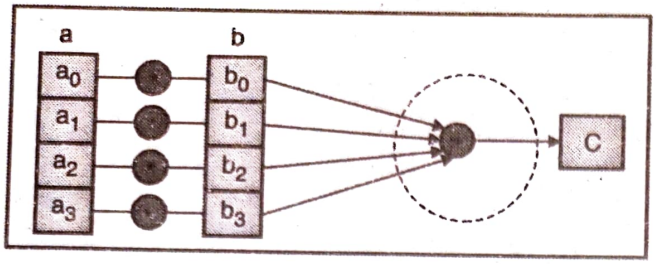


Fig. 6.7.8 : require sharing of thread's data for final sum

```
__global__voiddot(int*a,int*b,int*result)
{
// Each thread computes a pairwise product
int temp = a[threadIdx.x] * b[threadIdx.x];

// It is not possible here to compute the final sum because
each thread's copy of temp is private
```

☞ Sharing between threads

- Shared memory is a portion of memory in a block that can be shared by all the threads of that block. This memory is extremely fast on – chip user -- managed cache memory.
- This is declared using the qualifier **__shared__** in CUDA. This is not visible to threads in other blocks running in paraliel.

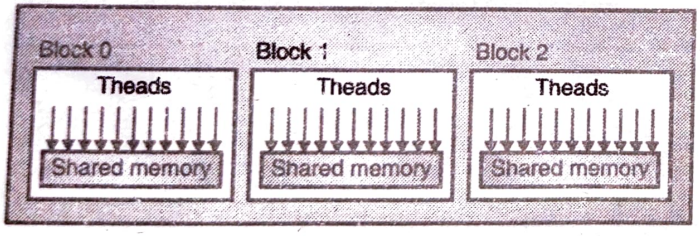


Fig. 6.7.9 : Shared memory in different blocks

- Now we can rewrite the kernel for doing parallel multiplication and serial addition as shown below.
- In this code segment pairwise multiplication is performed in parallel and the result of each thread's computation is stored in shared memory.
- Finally a single thread is used to perform summation of these pairwise products.
- This is just an example code to perform dot product using CUDA it may show wrong results sometimes.
- The debugging is an option to resolve the issues, mostly synchronization among threads required.
- The thread synchronization will solve the problem. This code is error – pronc because we have left thread synchronization as an exercise for the reader.

```
#define N 512
__global__void dot(int*a,int*b,int*result)
{
    // Declare temp as a shared memory to store results of
multiplication __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x] ;

// sums the pairwise products by thread 0
if( 0 == threadIdx.x )
{
    int sum = 0;
    for(int i = 0; i < N; i++)
    {
        sum = sum + temp[i];
    }
    *result = sum;
}
```