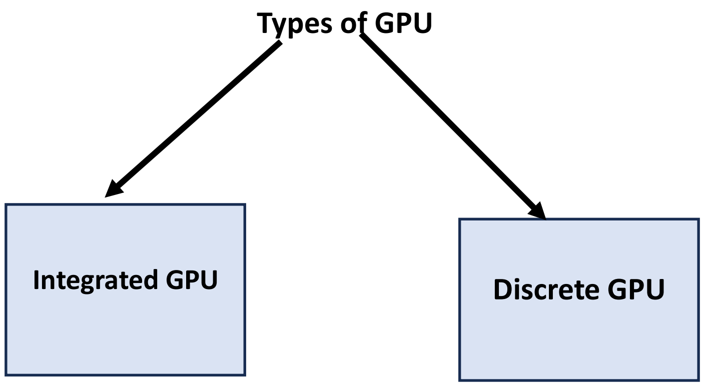
Unit-III

GPU Architectures and CUDA Introduction: The CPU, GPU system as an accelerated computational platform, The GPU and the thread engine, Characteristics of GPU memory spaces, The PCI bus: CPU to GPU data transfer overhead, multi-GPU platforms and MPI, Potential benefits of GPU accelerated platforms. Introduction to CUDA Programming:The history of high-performance computing – Technical requirements, Hello World from CUDA, Thread hierarchy, Vector addition using CUDA, Error reporting in CUDA, Data type support in CUDA.

The CPU, GPU system as an accelerated computational platform

A CPU, or Central Processing Unit, is the primary component of a computer that performs most of the processing inside the computer. It interprets instructions from the computer's memory, processes them, and performs arithmetic and logical operations.

A GPU, or Graphics Processing Unit, is a specialized electronic circuit designed to accelerate the processing of images and videos in a computer. Originally developed for rendering graphics in video games and multimedia applications. GPUs consist of thousands of smaller cores that can handle multiple tasks simultaneously. This parallel architecture makes them highly efficient for tasks that can be parallelized.



Integrated GPUs are built into the same chip as the central processing unit (CPU). They share system memory (RAM) with the CPU and are commonly found in laptops, Ultrabook's, and budget desktop computers. The AMD integrated GPUs are called Accelerated Processing Units (APUs). These are a tightly coupled combination of the CPU and a GPU. In the AMD APU, the CPU and GPU share the same processor memory. Integrated GPUs are suitable for basic tasks like web browsing, office applications, and multimedia playback.

A discrete GPU/Dedicated GPU (Graphics Processing Unit) is a separate graphics card that is installed on a computer's motherboard as an additional hardware component. Unlike integrated GPUs, which are integrated into the same chip as the CPU, discrete GPUs have their own dedicated video memory (VRAM) and are designed to handle graphics-related tasks independently. Discrete GPUs offer significantly higher performance and are suitable for demanding applications such as gaming, video editing, 3D rendering, and professional graphics work.

Communication Between CPU and GPU

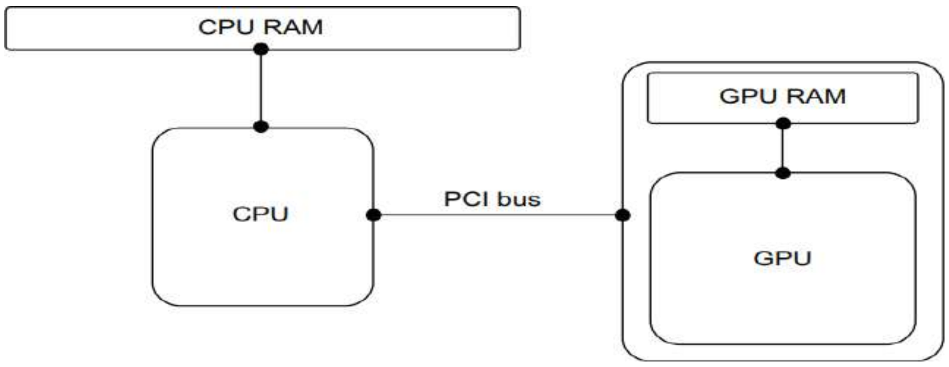


Figure 9.1 Block diagram of GPU-accelerated system using a dedicated GPU. The CPU and GPUeach have their own memory. The CPU and GPU communicate over a PCI bus.

Components of GPU Accelerated System

CPU—The main processor that is installed in the socket of the motherboard.

CPU RAM—The "memory sticks" or dual in-line memory modules (DIMMs) containing Dynamic Random- Access Memory (DRAM) is a type of computer memory module that is used in desktop computers, servers, and workstations that are inserted into the memory slots in the motherboard.

GPU—A large peripheral card installed in a Peripheral Component Interconnect Express (PCIe) slot on the motherboard.

GPU RAM—Memory modules on the GPU peripheral card for exclusive use of the GPU.

PCI bus—The wiring that connects the peripheral cards to the other components on the motherboard.

Figure 9.1 conceptually illustrated a CPU-GPU system with a dedicated GPU. A CPU has access to its own memory space (CPU RAM) and is connected to a GPU via a PCI bus. It is able to send data and instructions over the PCI bus for the GPU to work with. The GPU has its own memory space, separate from the CPU memory space. In order for work to be executed on the GPU, at some point, data must be transferred from the CPU to the GPU. When the work is complete, and the results are going to be written to file, the GPU must send data back to the CPU. The instructions the GPU must execute are also sent from CPU to GPU. Each one of these transactions is mediated by the PCI bus.

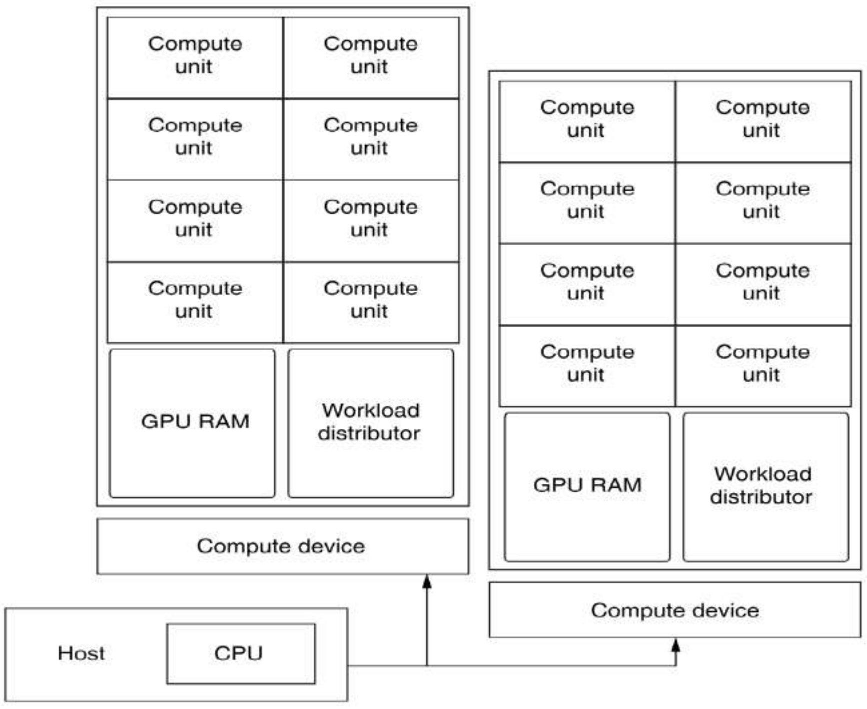
The GPU and the thread engine

The thread engine within a CPU manages the execution of threads, schedules tasks for processing, and ensures efficient utilization of available resources. The graphics processor is like the ideal thread engine.

- **The components of this thread engine are**

- A seemingly infinite number of threads
- Zero-time cost for switching or starting threads: refers to the ideal scenario where the process of initiating or switching between threads occurs instantaneously, without any additional computational overhead.
- Latency hiding of memory accesses through automatic switching between work groups: Memory latency occurs because accessing data from the main memory (RAM) is significantly slower compared to accessing data from the CPU's cache or registers.

- **For Example here we will go through a single node system with a single multiprocessor CPU and two GPUs**



- **Fig 9.2 Simplified Block Diagram of a GPU System consisting of two compute devices each having multiple compute units and separate GPU Memory.**

A GPU is composed of

- **Compute Device :** A compute device in a GPU is a subset of the GPU that is dedicated to general-purpose parallel processing tasks. It consists of multiple compute units
- **GPU RAM** : (also known as global memory) refers to the dedicated memory that is integrated into a graphics processing unit (GPU) or graphics card.
- **Workload distributor**: Instructions and data received from the CPU are processed by the workload distributor. The distributor coordinates instruction execution and data movement onto and off of the Compute Units.
- **Compute units (CU):** Compute units are the fundamental processing units within a compute device. Each compute unit typically consists of multiple ALUs and multiple graphics processors called processing elements (PEs). CUs have their own internal architecture, often referred to as the microarchitecture.

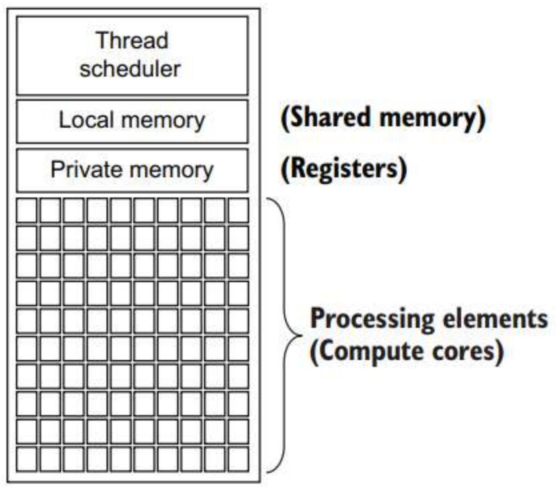
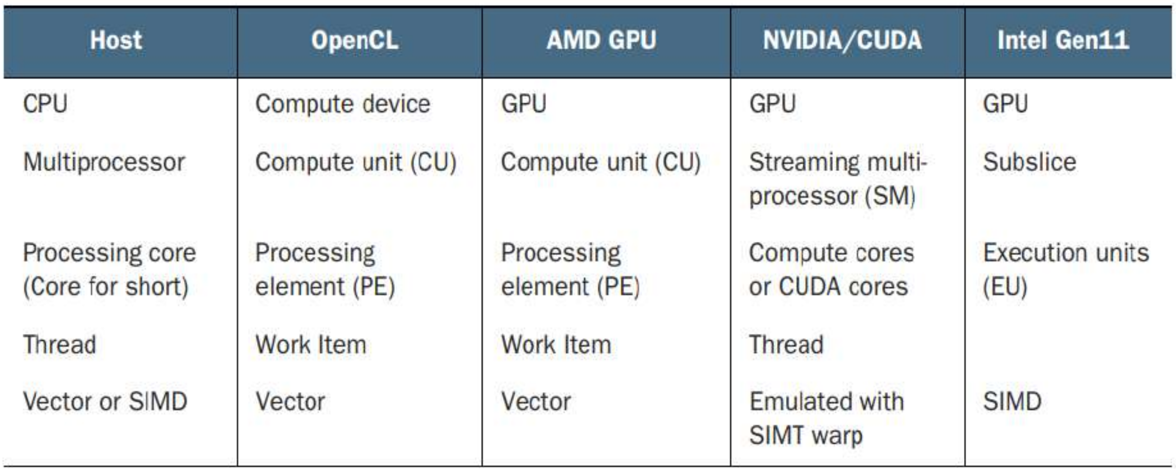


Figure 9.3 Simplified block diagram of a compute unit (CU) with a large number of processing elements (PEs).

- **Processing Element :** PEs within a GPU are designed to execute instructions in parallel. They can handle multiple threads and data elements simultaneously, allowing for massive parallelism.

- **Hardware Terminology**

Table 9.1 Hardware terminology: A rough translation



Host	OpenCL	AMD GPU	NVIDIA/CUDA	Intel Gen11
CPU	Compute device	GPU	GPU	GPU
Multiprocessor	Compute unit (CU)	Compute unit (CU)	Streaming multi-processor (SM)	Subslice
Processing core (Core for short)	Processing element (PE)	Processing element (PE)	Compute cores or CUDA cores	Execution units (EU)
Thread	Work Item	Work Item	Thread	
Vector or SIMD	Vector	Vector	Emulated with SIMT warp	SIMD

Table 9.1 summarizes the rough equivalence of terminology, in different hardware architectures. Example CPU in Host is termed as Compute Device (OpenCL), GPU in (AMD GPU) ,GPU (NVIDIA/CUDA) and GPU in Intel Gen11.

- **Calculating the peak theoretical flops for some leading GPUs**

FLOPS provide a measure of a computer system's processing speed, especially when dealing with numerical and scientific computations. It allows researchers and developers to compare the performance of different hardware architectures and configurations.

The peak theoretical flops can be calculated by taking the clock rate times the number of processors times the number of floating-point operations per cycle. The flops per cycle accounts for the fused-multiply add (FMA), which does two operations in one cycle.

Peak Theoretical Flops (GFlops/s) = Clock rate MHZ × Compute Units × Processing units × Flops/cycle.

Characteristics of GPU memory spaces

GPU memory spaces are essential for efficient data management and processing in parallel computing environments, particularly in the context of graphics processing and high-performance computing. Graphics Processing Units (GPUs) have multiple memory spaces with different characteristics, each optimized for specific types of tasks.

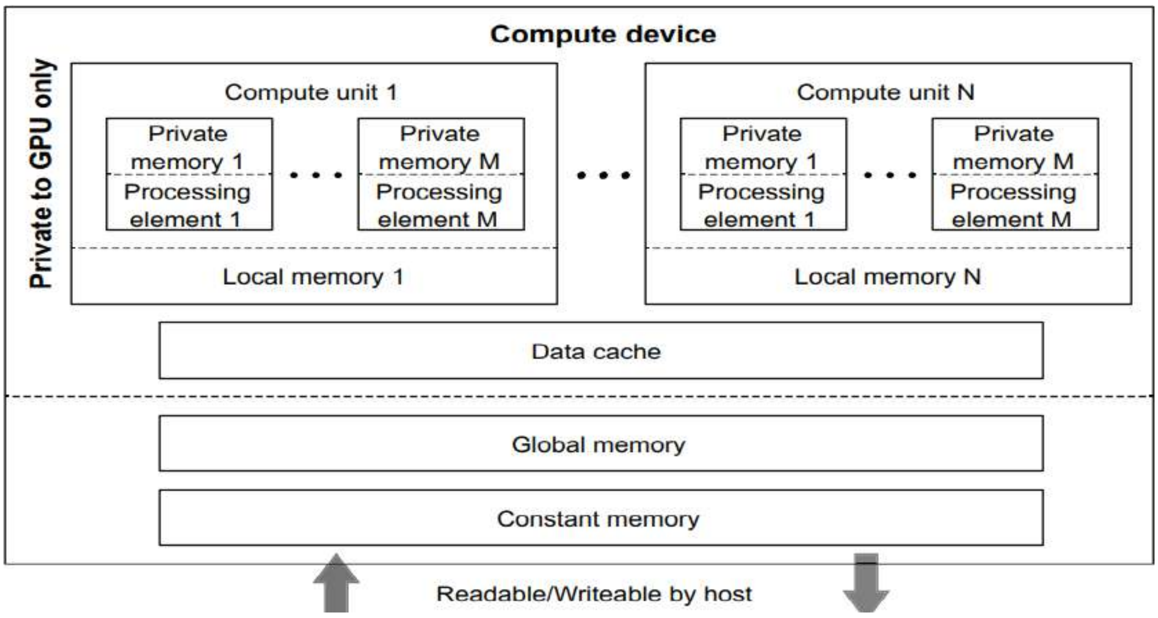


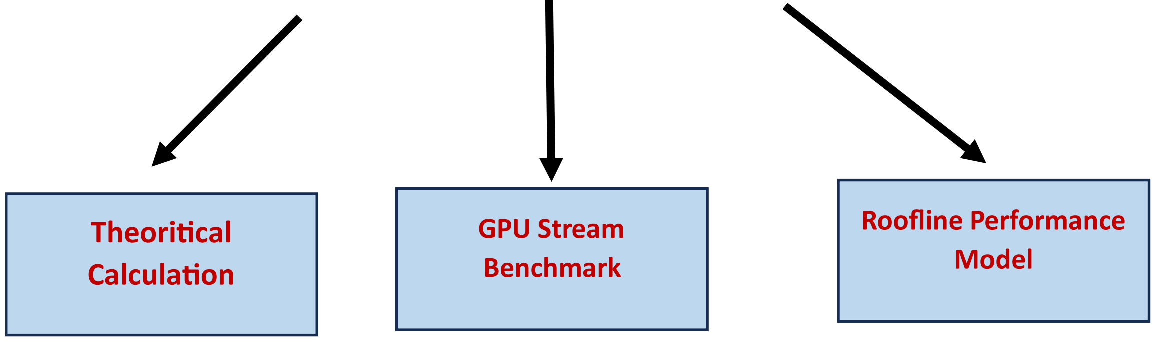
Figure 9.4 Rectangles show each component of the GPU and the memory that is at each hardware level. The host writes and reads the global and constant memory. Each of the CUs can read and write from the global memory and read from the constant memory

The list of the GPU memory types and their characteristics are as follows.

- **Private memory (register memory)—** Private memory in the context of GPUs typically refers to the local memory associated with individual processing element. It is accessible by a single Processing Element (PE) and only by that PE.
- **Local memory—**Accessible to a single Control Unit and all of the Processing Elements on that Control Unit.
- **Constant memory**—Constant memory is read-only, meaning that data stored in constant memory cannot be modified by the GPU kernel during execution. It is primarily designed for read operations and is well-suited for storing constant values, lookup tables, or other data that does not change during the kernel's execution
- **Global memory**—Memory that's located on the GPU and accessible by all of the Control Unit's

The performance of GPU memory, also known as memory subsystem performance, is a critical factor that significantly impacts the overall performance of GPU-accelerated applications.

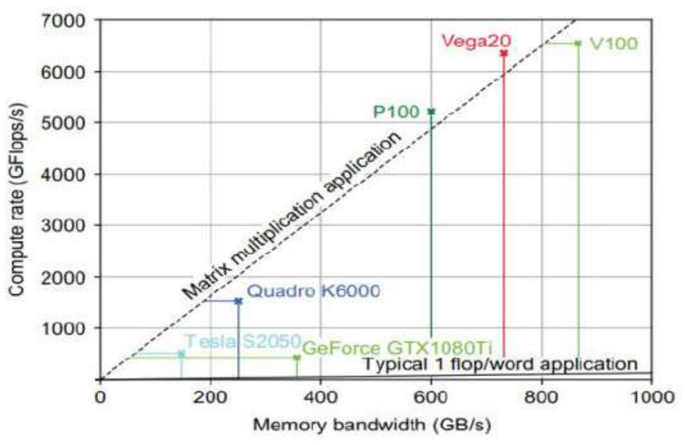
GPU memory performance is influenced by Memory Bandwidth which can be calculated by



Theoretical Calculation: Theoretical Bandwidth = Memory Transaction Rate(Gbps) × Memory bus (bits) × (1 byte/8 bits)

GPU Stream Bench Mark: In the context of GPUs (Graphics Processing Units) and general computing, a benchmark refers to a standardized test or set of tests designed to measure the performance of a GPU or an entire computer system. Benchmarks are used to evaluate various aspects of a GPU's capabilities, such as computational power, memory bandwidth, and graphics rendering performance. For Example, the Babel STREAM Benchmark code measures the bandwidth of a variety of hardware with different programming languages.

The mixbench tool was developed to draw out the differences between the performance of different GPU devices. Using the mixbench performance tool we can choose the best GPU for a workload.

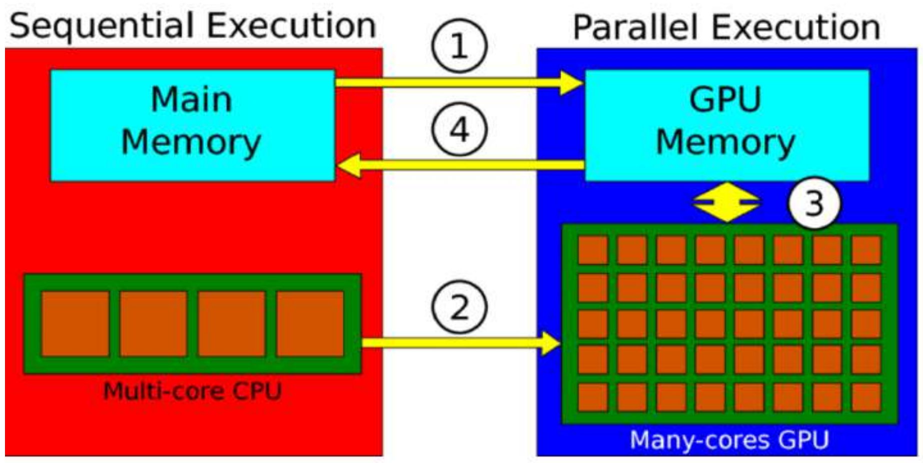


A collection of performance points for GPU devices (shown on the plot on the right) along with the application arithmetic intensity (shown as straight lines). Values above the line indicate that the application is memory-bound and below the line indicates it is compute-bound.

Roofline Performance Model: The Roofline Performance Model is a graphical representation used in high-performance computing (HPC) to analyze and visualize the performance of algorithms on a particular hardware architecture. The Roofline model serves as a visual communication tool that can be easily shared among team members, researchers, and stakeholders. It provides a clear and concise representation of performance characteristics, making it easier to convey insights and optimization recommendations.

The PCI bus: CPU to GPU data transfer overhead

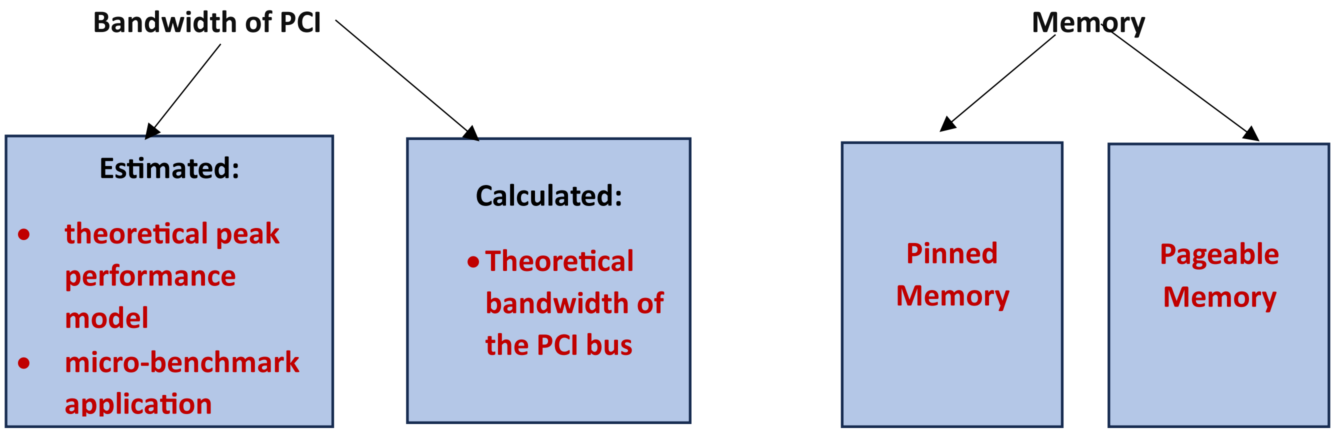
The PCI (Peripheral Component Interconnect) bus is a standard interface that connects various hardware devices, including GPUs (Graphics Processing Units), to a computer's motherboard. The current version of the PCI bus is called PCI Express (PCIe).



The Datatransfer includes the following steps for a typical program exeecution on GPU

1. Copy data to GPU memory
2. CPU instructs the GPU (kernel configuration and launching)
3. Data processed by many cores in parallel
4. Copy result back to main memory.

When transferring data from the CPU to the GPU or vice versa over the PCI bus, there can be overhead associated with the data transfer process. This overhead is influenced **by bandwidth and Memory.**


Bandwidth of PCI
Memory
Estimated:
• theoretical peak performance model
• micro-benchmark application
Calculated:
• Theoretical bandwidth of the PCI bus
Pinned Memory
Pageable Memory

A back-of-the-envelope theoretical peak performance model: The Theoretical Peak Performance Model is a concept used in high-performance computing to estimate the maximum computational performance that a system or a specific hardware component can achieve under ideal conditions. This model provides an upper bound on performance based on the theoretical capabilities of the hardware and is often used as a reference point for evaluating the efficiency of algorithms and applications.

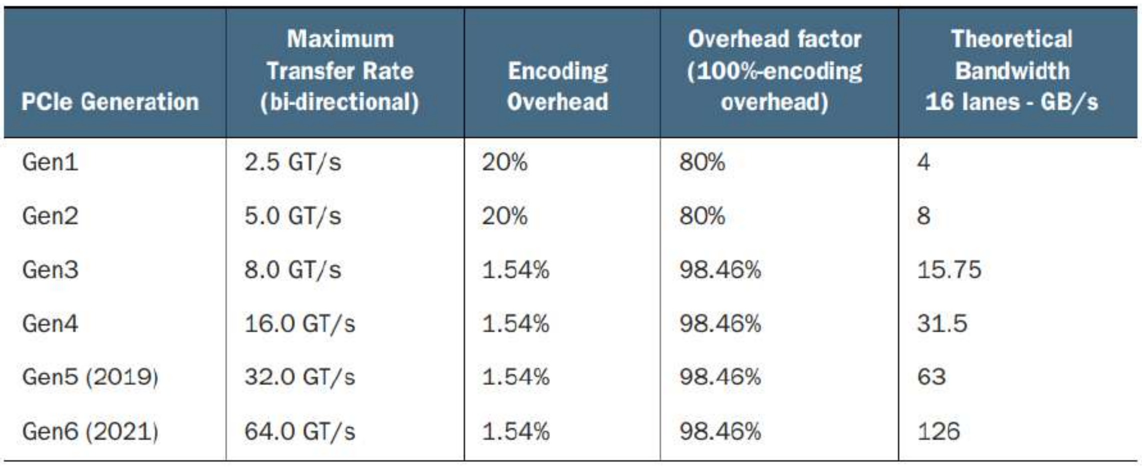
A micro-benchmark application: A microbenchmark is a small, focused benchmark designed to measure the performance of a specific aspect of a system, component, or function. Microbenchmarks are valuable for isolating and evaluating the performance of a particular piece of code or hardware, helping developers optimize and understand the efficiency of specific operations. These benchmarks are often used during the development and testing phases to identify bottlenecks and make targeted improvements.

Theoretical bandwidth of the PCI bus (Calculated):

Theoretical Bandwidth (GB/s) = PCIeLanes × TransferRate (GT/s) × OverheadFactor(Gb/GT) × byte/8 bits

- PCIe lanes refer to the individual data transfer paths within a PCI Express (PCIe) interface. Each lane is a point-to-point connection between two components, typically between a device (e.g., a graphics card, storage device) and the computer's motherboard. The concept of lanes is fundamental to understanding the bandwidth and data transfer capabilities of the PCIe .
- The maximum transfer rates for each lane in a PCIe bus can directly be determined by its design generation. Generation is a specification for the required performance of the hardware, much like 4G is an industry standard for cell-phones. The PCI Special Interest Group (PCI SIG) represents industry partners and establishes a PCIe specification that is commonly referred to as generation or gen for short.
- Transmitting data across the PCI bus requires additional overhead. Generation 1 and 2 standards stipulate that 10 bytes are transmitted for every 8 bytes of useful data. Starting with generation 3, the transfer transmits 130 bytes for every 128 bytes of data. The overhead factor is the ratio of the number of usable bytes over the total bytes transmitted.

PCI Express (PCIe) specifications by generation



PCIe Generation	Maximum Transfer Rate (bi-directional)	Encoding Overhead	Overhead factor (100%-encoding overhead)	Theoretical Bandwidth 16 lanes - GB/s
Gen1	2.5 GT/s	20%	80%	4
Gen2	5.0 GT/s	20%	80%	8
Gen3	8.0 GT/s	1.54%	98.46%	15.75
Gen4	16.0 GT/s	1.54%	98.46%	31.5
Gen5 (2019)	32.0 GT/s	1.54%	98.46%	63
Gen6 (2021)	64.0 GT/s	1.54%	98.46%	126

Pinned memory, also known as locked or page-locked memory, is a type of memory in a computer system that remains fixed in physical RAM and is not subject to swapping to disk by the operating system's virtual memory manager. In GPU programming, pinned memory is often used to facilitate fast data transfers between the CPU and GPU. Pinned memory consumes physical RAM exclusively. Applications using a significant amount of pinned memory may impact overall system resources, so careful consideration is needed to avoid resource exhaustion.

Pageable memory, also known as virtual memory, is a type of memory management strategy used by operating systems to efficiently handle the allocation and deallocation of memory for running processes. In a pageable memory system, the operating system divides the physical memory into fixed-size blocks called pages. These pages can be dynamically moved between the computer's RAM (Random Access Memory) and a secondary storage device, such as a hard disk, to optimize overall system performance.

Multi-GPU platforms and MPI

A multi-GPU (Graphics Processing Unit) platform refers to a system configuration that incorporates more than one GPU. Multi-GPU setups are commonly used in high-performance computing, scientific simulations, and graphics-intensive applications to enhance computational power and graphics rendering capabilities.

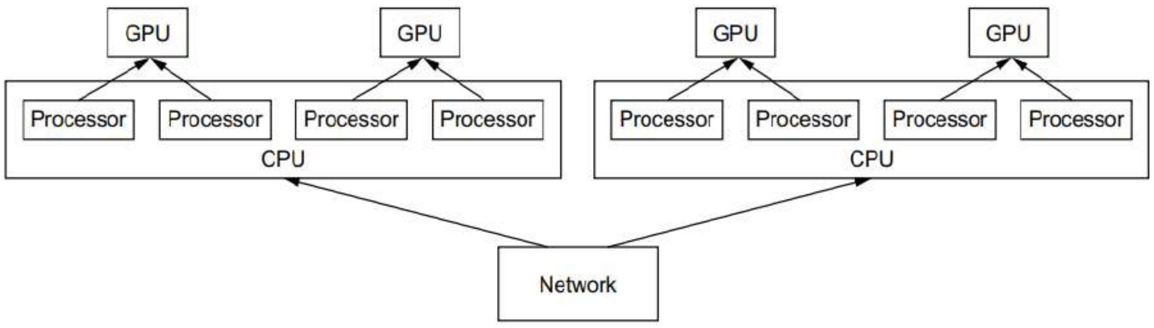
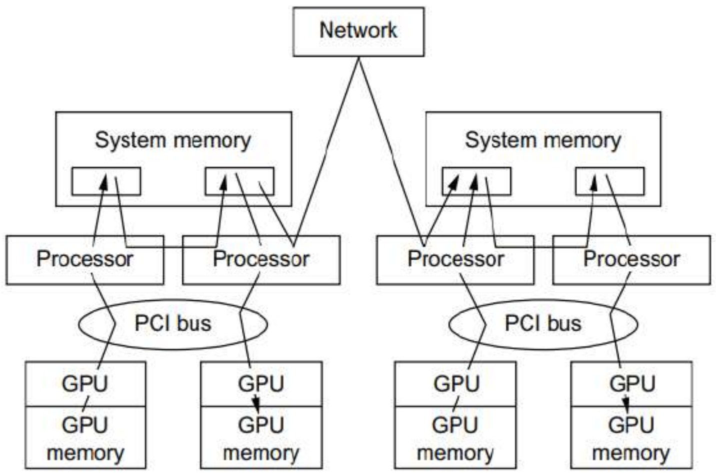


Figure 9.9 Here we illustrate a multi-GPU platform. A single compute node can have multiple GPUs and multiple processors. There can also be multiple nodes connected across a network.

To use multiple GPUs, we have to send data from one GPU to another.

- **Standard data transfer process:** This has a lot of data movement and will be a major limitation to application performance.



1 Copy the data from the GPU to the host processor

a Move the data across the PCI bus to the processor

b Store the data in CPU DRAM memory

2 Send the data in an MPI message to another processor

a Stage the data from CPU memory to the processor

b Move the data across the PCI bus to the network interface card (NIC)

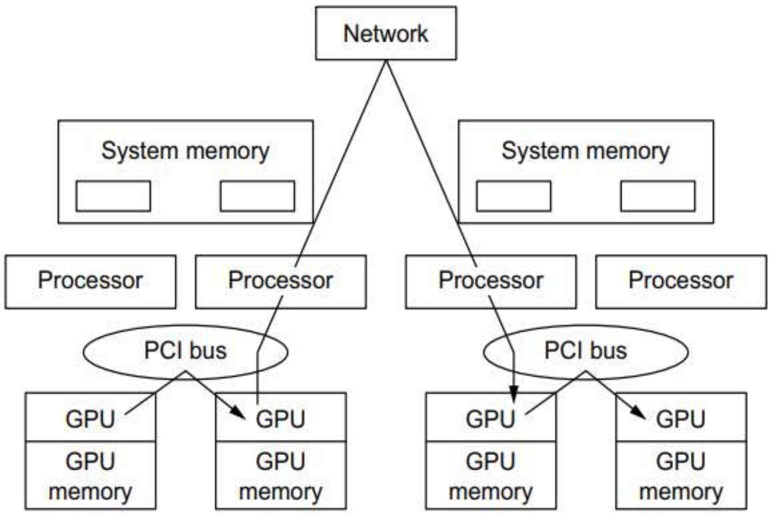
c Store the data from the processor to CPU memory

3 Copy the data from the second processor to the second GPU

a Load the data from CPU memory to the processor

b Send the data across the PCI bus to the GPU

- **Optimizing the data movement between GPUs across the network:** The data movement bypasses the CPU when moving data from one GPU to another



Potential benefits of GPU-accelerated platforms

1. **Reducing time-to-solution**

   Reducing time-to-solution in GPU (Graphics Processing Unit) computing involves optimizing the code, leveraging parallel processing capabilities, and making efficient use of GPU resources. Structure the computations to take advantage of data parallelism, where the same operation is performed on multiple data elements concurrently. This aligns well with the architecture of GPUs, which excel at handling parallel tasks.

2. **Reducing energy use with GPUs**

   Reducing energy use with GPUs involves optimizing your GPU-accelerated applications to achieve computational efficiency while considering power consumption.

   The energy consumption for your application can be estimated using the formula

   Energy = (N Processors) × (R Watts/Processor) × (T hours)

   Achieving a reduction in energy cost through GPU accelerator devices requires that the application expose sufficient parallelism and that the device's resources are efficiently utilized.

3. **Reduction in cloud computing costs with GPUs**

Cloud computing services from Google and Amazon let you match your workloads to a wide range of compute server types and demands.

- If your application is memory bound, you can use a GPU that has a lower flops to-loads ratio at a lower cost.
- If you are more concerned with turnaround time, you can add more GPUs or CPUs.
- If your deadlines are less serious, you can use preemptible resources at a considerable reduction in cost.

As the cost of computing is more visible with cloud computing services, optimizing application's performance becomes a higher priority. Cloud computing has the advantage of giving you access to a wider variety of hardware than you can have on-site and more options to match the hardware to the workload.

When to use GPUs

GPUs are not general-purpose processors. They are most appropriate when the computation workload is similar to a graphics workload—lots of operations that are identical.

There are some areas where GPUs still do not perform well.

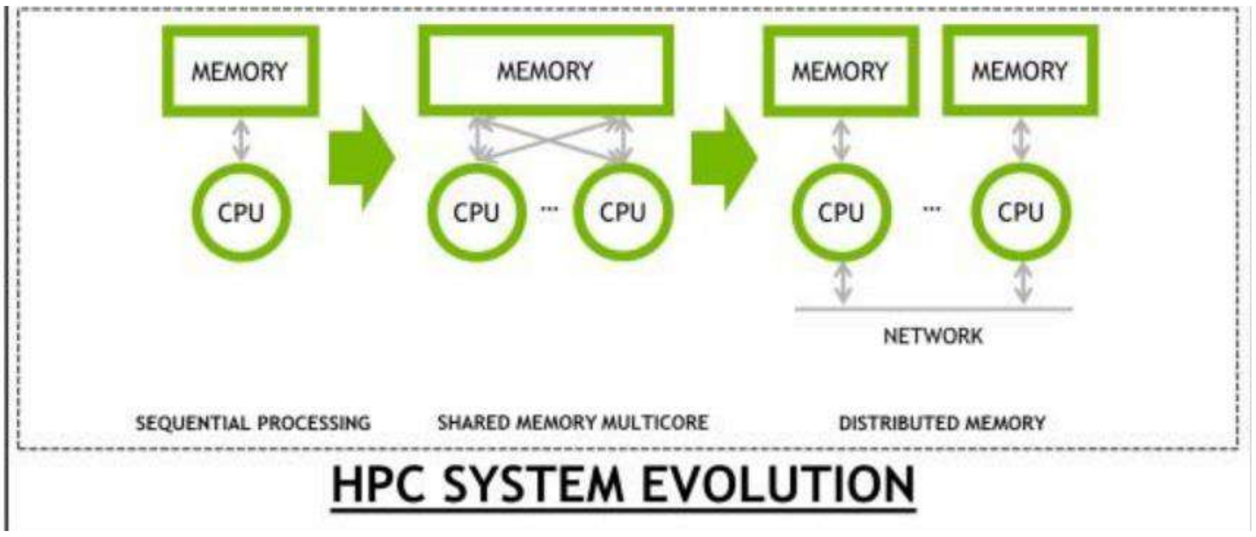
- **Lack of parallelism**— "With great power comes great need for parallelism." If you don't have the parallelism, GPUs can't do a lot for you. This is the first law of GPGPU programming.
- **Irregular memory access**—CPUs also struggle with this. The massive parallelism of GPUs brings no benefit to this situation.
- **Dynamic memory requirements**—Memory allocation is done on the CPU, which severely limits algorithms that require memory sizes determined on the fly.
- **Recursive algorithms**—GPUs have limited stack memory resources, and suppliers often state that recursion is not supported.

The history of high-performance computing

➢ High-Performance Computing (HPC)

High-Performance Computing (HPC) refers to the use of advanced computing techniques and technologies to solve complex problems or perform large-scale simulations and computations at speeds and scales beyond the capabilities of typical desktop or server computers. HPC systems typically involve the use of parallel processing and parallel computing techniques to harness the power of multiple processors or cores working together. In addition to parallel processing within a single machine, HPC can also involve distributed computing, where tasks are distributed across a network of interconnected computers.

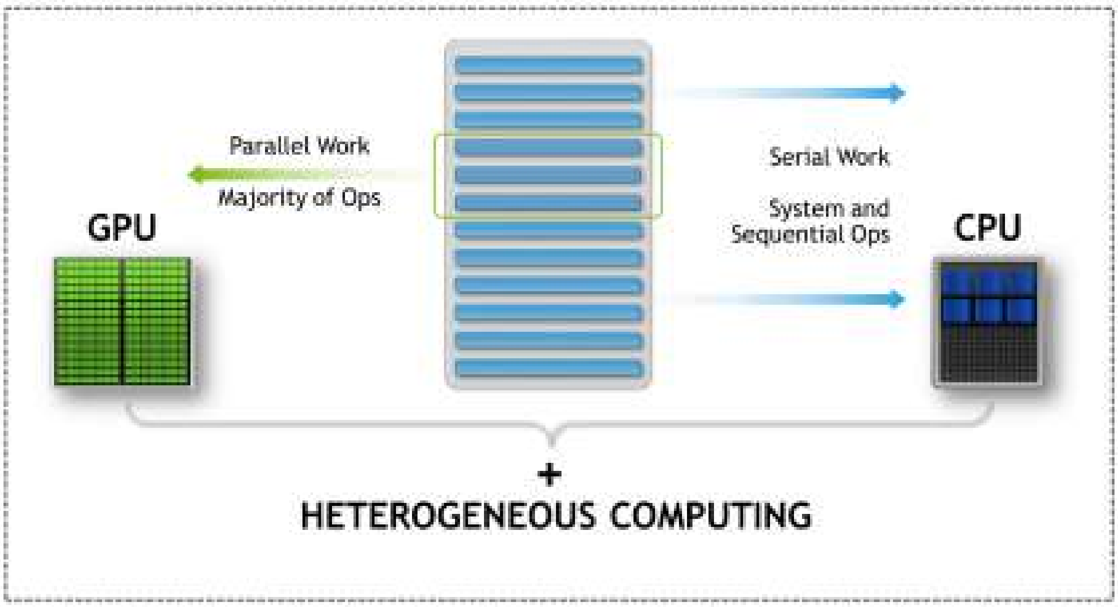
The following diagram shows the evolution of computer architecture from sequential processing to distributed memory



➢ Heterogeneous computing

Heterogeneous computing refers to the use of systems that employ different kinds of processors or co-processors working together to perform a given task. In a heterogeneous computing environment, various types of processing units, such as central processing units (CPUs), graphics processing units (GPUs) or accelerators, collaborate to achieve improved performance and efficiency.

The following diagram represents an application running on multiple processor types:



The key point is that CPU is good for a certain fraction of code that is latency bound, while GPU is good at running the Single Instruction Multiple Data (SIMD) part of the code in parallel. If only one of them, that is, CPU code or GPU code, runs faster after optimization, this won't necessarily result in good speedup for the overall application. It is required that both of the processors, when used optimally, give maximum benefit in terms of performance. This approach of essentially offloading certain types of operations from the processor onto a GPU is called heterogeneous computing.

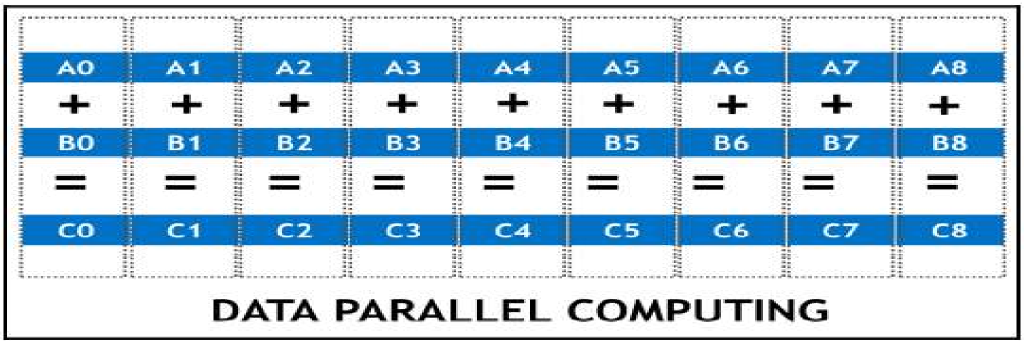
- **Programming paradigm**

A programming paradigm is a fundamental style or approach to programming that provides a set of principles, methods, and practices for designing and implementing software. It encompasses the overall philosophy and methodology that guides the development of computer programs. Different programming paradigms offer distinct ways of organizing and structuring code, handling data, and solving problems.

SIMD is used to describe an architecture where the same instruction is applied in parallel to multiple data points. This description is suitable for processors that have the capability of doing vectorization. In contrast, in Single Instruction Multiple Threads (SIMTs), rather than a single thread issuing the instructions, multiple threads issue the same instruction to different data. The GPU architecture is more suitable in terms of the SIMT category compared to SIMD.

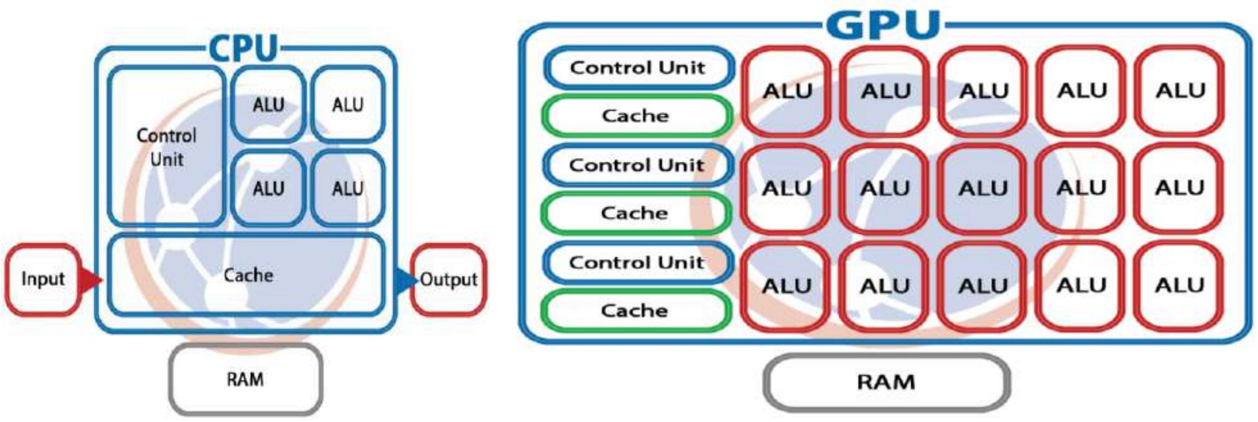
Example : In SIMD, we might have a vector addition operation where a single instruction is applied to corresponding elements of two vectors. In SIMT, we might use CUDA to perform vector addition with multiple threads executing the same instruction on different elements.

The following screenshot shows vector addition, depicting an example of this paradigm:


A0 A1 A2 A3 A4 A5 A6 A7 A8
+
B0 B1 B2 B3 B4 B5 B6 B7 B8
=
C0 C1 C2 C3 C4 C5 C6 C7 C8
DATA PARALLEL COMPUTING

- **Low latency versus higher throughput**

Latency refers to the time delay between the initiation of a process or task and its completion. Throughput is the rate at which a system or network can process or transfer data. It measures the amount of work done in a given period. Low-latency systems prioritize minimizing the time it takes for individual tasks to complete. High-throughput systems aim to maximize the amount of work done over a given time period.



CPU architecture is optimized for low latency access while GPU architecture is optimized for data parallel throughput computation.

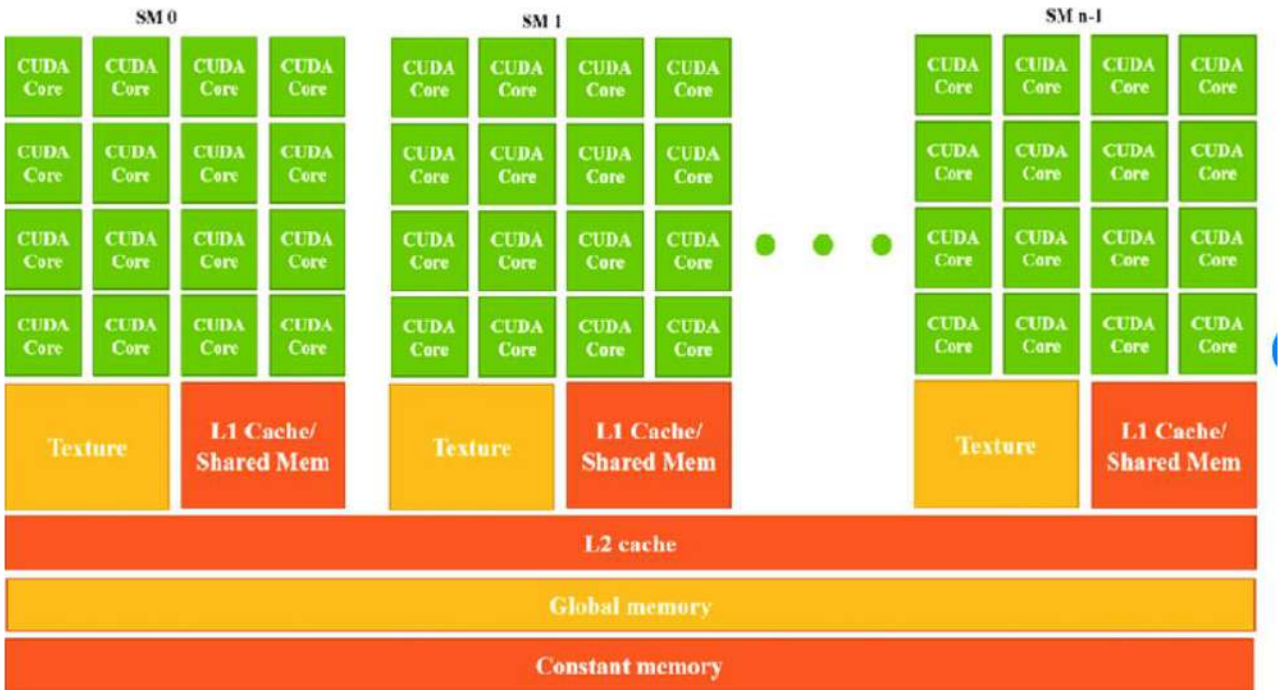
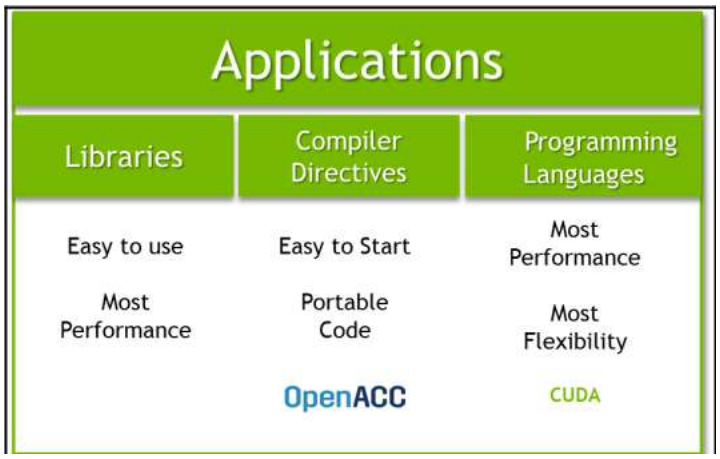


Fig: Schematic of NVIDIA GPU architecture, where SM refers to streaming multiprocessor.

- **Programming approaches to GPU**

Like any other processor, the GPU architecture can be coded using various methods. The easiest method, which provides drop-in acceleration, is making use of existing libraries. Alternatively, developers can choose to make use of OpenACC directives for quick acceleration results and portability. Another option is to choose to dive into CUDA by making use of language constructs in C, C++, Fortran, Python, and more for the highest performance and flexibility.

The following screenshot represents the various ways we can perform GPU programming:



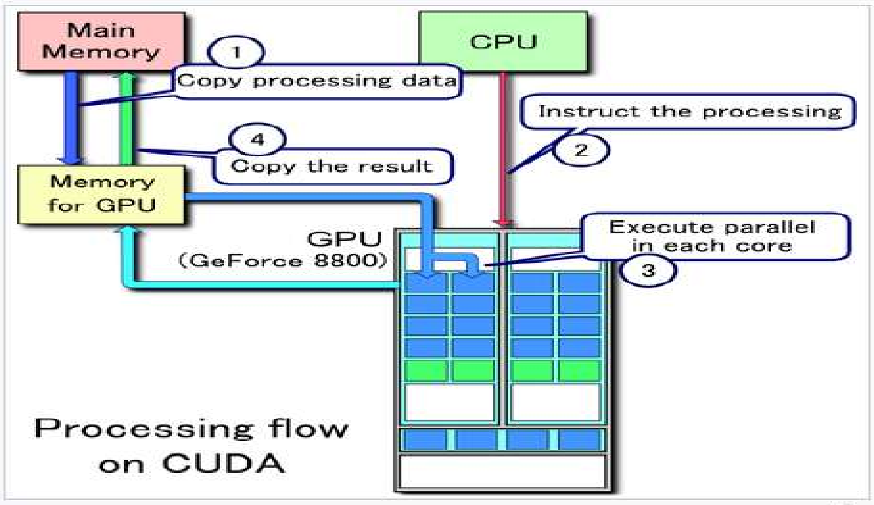
CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for their GPUs (Graphics Processing Units). It allows developers to use NVIDIA GPUs for general-purpose processing (not just graphics) by writing programs making use of kernels. Kernels are functions that run on a GPU. When we launch a kernel, it is executed as a set of Threads. Each thread is mapped to a single CUDA core on a GPU and performs the same operation on a subset of data. According to Flynn‘s taxonomy, it‘s a Single Instruction Multiple Data (SIMD) computation.

CUDA provides a flexible and efficient way to harness the computational power of GPUs for a wide range of applications, including scientific simulations, machine learning, and high-performance computing.

CUDA is a heterogeneous programming model that includes provisions for both CPU and GPU. The CUDA C/C++ programming interface consists of C language extensions so that you can target portions of source code for parallel execution on the device (GPU).

In CUDA, there are two processors that work with each other. The host is usually referred to as the CPU, while the device is usually referred to as the GPU. The host is responsible for calling the device functions. Part of the code that runs on the GPU is called device code, while the serial code that runs on the CPU is called host code. The intention is to take a systematic step wise approach, start with some sequential code, and convert it into CUDA- aware code by adding some additional keywords.



Example of CUDA processing flow

1. Copy data from main memory to GPU memory
2. CPU initiates the GPU compute kernel
3. GPU's CUDA cores execute the kernel in parallel
4. Copy the resulting data from GPU memory to main memory

Hello World from CUDA

A hello world program in C and CUDA .

C Program

```
void c_hello()
{
   printf("Hello World!\n");
}

int main()
{
   c_hello();
   return 0;
}
```

Sample CUDA Program to print a statement

```
__global__ void cuda_hello()
{
   printf("Hello World from GPU!\n");
}

int main()
 {
   cuda_hello<<<1,1>>>();
   cudaDeviceSynchronize();

   return 0;
}
```

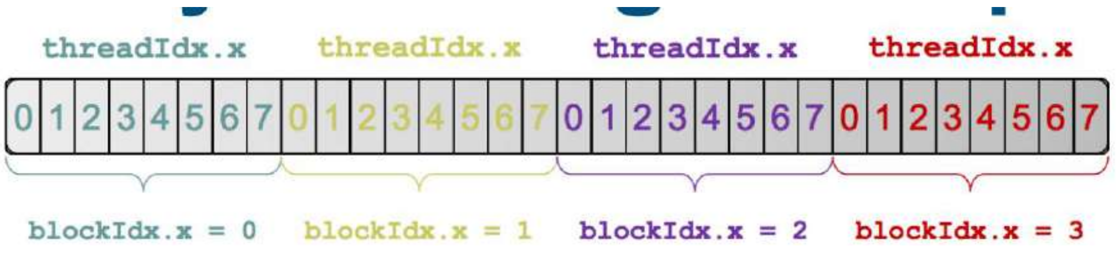
In the preceding code, we added a few constructs and keywords, as follows:

__global__ : (2 underscore symbols before and after global keyword).This keyword, when added before the function, tells the compiler that this is a function that will run on the device and not on the host. However, note that it is called by the host. Another important thing to note here is that the return type of the device function is always "void". Dataparallel portions of an algorithm are executed on the device as kernels.

<<< >>>: This keyword tells the compiler that this is a call to the device function and not the host function. The two parameters are **<<< NUMBER_OF_BLOCKS, NUMBER_OF_THREADS_PER_BLOCK>>>.**

The below figure refers to 4 blocks and 8 threads per block(No.of times the statement/kernel function gets exceuted=No.of bolocks*no.of threads per block)

•



cuda_hello <<<1,1 >>>() is configured to run in a single block which has a single thread and will, therefore, run only once. (1*1=1)

cuda_hello <<<1,5 >>>() is configured to run in a single block which has 5 threads and will, therefore, run 5 times. (1*5=5)

cuda_hello <<< 5,1>>>() is configured to run in 5 blocks where each block have a single thread and will, therefore, run five times. (5*1=5)

cuda_hello <<< 5,5>>>() is configured to run in 5 blocks where each have 5 threads and will, therefore, run 25 times(5*5=25).

cudaDeviceSynchronize(): All of the kernel calls in CUDA are asynchronous in nature. The host becomes free after calling the kernel and starts executing the next instruction afterward. This should come as no big surprise since this is a heterogeneous environment and hence both the host and device can run in parallel to make use of the types of processors that are available. In case the host needs to wait for the device to finish, APIs have been provided as part of CUDA programming that make the host code wait for the device function to finish. One such API is cudaDeviceSynchronize, which waits until all of the previous calls to the device have finished.

CUDA program to print a statement 5 times along with its thread Id and block iD, using single block

```
%%cu
 #include <stdio.h>
#include <cuda.h>
global void print_from_gpu(void)
{
printf("Hello World! from thread [%d,%d] \ From device\n", ,blockIdx.x, threadIdx.x);
}
int main(void)
 {
printf("Hello World from host!\n");
print_from_gpu<< < 1,5>>> ();
cudaDeviceSynchronize();
return 0;
}
```

In this statement of the above program
printf("Hello World! from thread [%d,%d] \ From device\n", ,blockIdx.x, threadIdx.x);
threadIdx.x Index of a thread inside a block
blockIdx.x: Index of a block

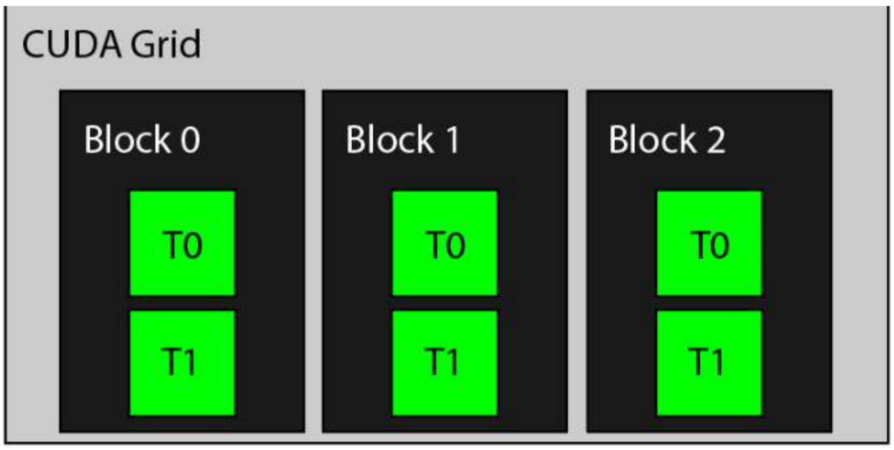
Thread Hierarchy

With a single processor executing a program, it's easy to tell what's going on and where it's happening. With a CUDA device, there seem to be a lot of things going on at once in a lot of different places. CUDA organizes a parallel computation using the abstractions or thread hierarchy consisting of threads, blocks and grids.

Thread: Single execution unit.

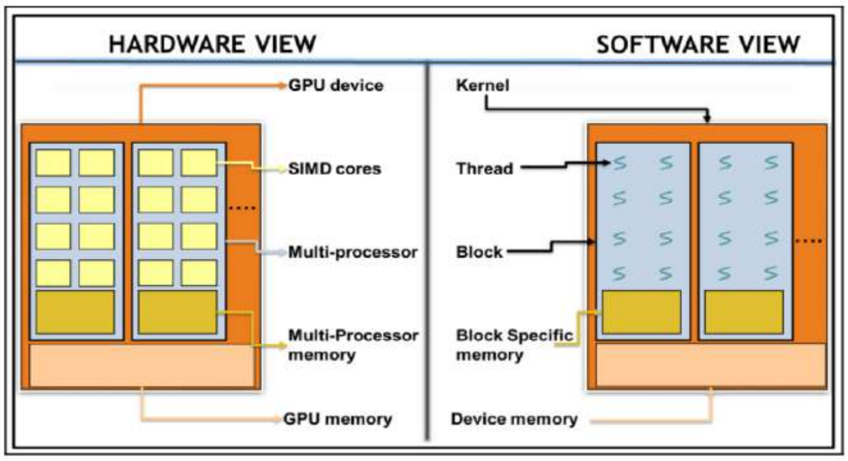
Block: A block contains numerous threads

Grid: A grid contains numerous blocks. A kernel launches a grid



H/W and S/W View GPU architecture

One of the key reasons why CUDA became so popular is because the hardware and software have been designed and tightly bound to get the best performance out of the application. Due to this, it becomes necessary to show the relationship between the software CUDA programming concepts and the hardware design itself. For Example Tesla P100 GPU consists of : 56 Streaming Multiprocessor(Multiprocessor) , 3584 CUDA Cores, 16GB memory

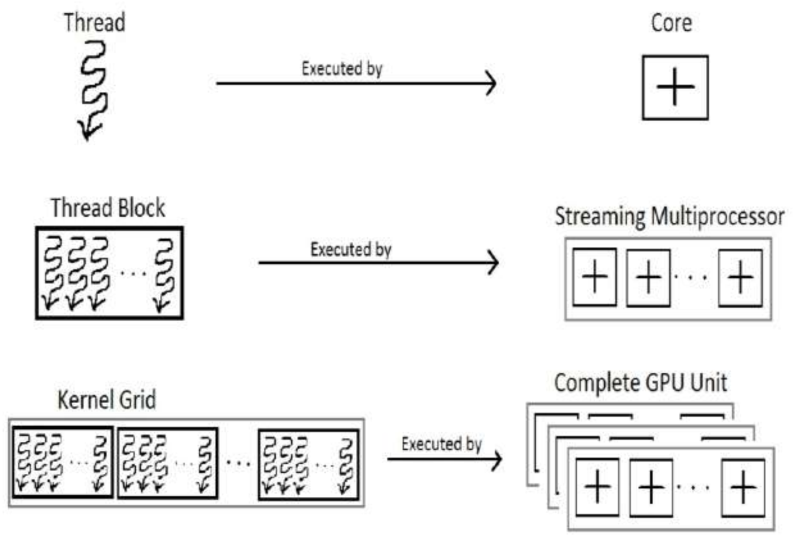


The following figure, in accordance with the preceding screenshot, explains software and hardware mapping in terms of the CUDA programming model:

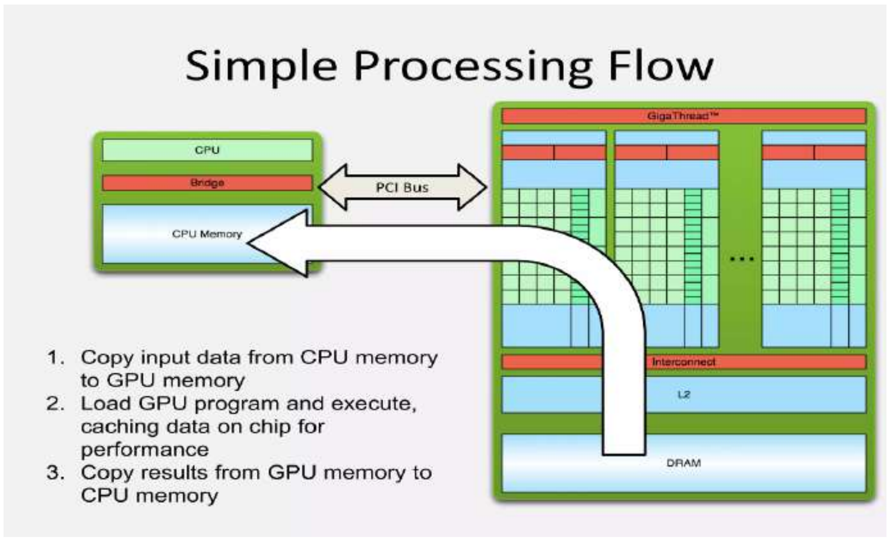
CUDA Threads: CUDA threads execute on a CUDA core. CUDA threads are different from CPU threads. CUDA threads are extremely lightweight and provide fast context switching. The reason for fast context switching is due to the availability of a large register size in a GPU and hardware-based scheduler. The thread context is present in registers compared to CPU, where the thread handle resides in a lower memory hierarchy such as a cache. Hence, when one thread is idle/waiting, another thread that is ready can start executing with almost no delay. Each CUDA thread must execute the same kernel and work independently on different data (SIMT).

CUDA blocks: CUDA threads are grouped together into a logical entity called a CUDA block. CUDA blocks execute on a single Streaming Multiprocessor (SM). One block runs on a single SM, that is, all of the threads within one block can only execute on cores in one SM and do not execute on the cores of other SMs. Each GPU may have one or more SM and hence to effectively make use of the whole GPU; the user needs to divide the parallel computation into blocks and threads.

GRID/kernel: CUDA blocks are grouped together into a logical entity called a CUDA GRID. A CUDA GRID is then executed on the device



CUDA Processing Flow


Simple Processing Flow
CPU
Bridge
CPU Memory
PCI Bus
GigaThread™
Interconnect
L2
DRAM
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

The CUDA processing flow has some steps as shown in the figure above.

1. Copy input data from CPU memory to GPU Memory: This basically has to do three things.

- **Memory allocation on CPU:** This can be done by either static or dynamic memory allocation depending on the application.

- **Memory allocation on GPU:** CPU memory and GPU memory are physically separate memory. malloc allocates memory on the CPU's RAM. The GPU kernel/device function can only access memory that's allocated/pointing to the device memory. To allocate memory on the GPU, we need to use the cudaMalloc API. Unlike the malloc command, cudaMalloc does not return a pointer to allocated memory; instead, it takes a pointer reference as a parameter and updates the same with the allocated memory.
  Function cudaMalloc() is provided by header file cuda.h. It is used to dynamically allocate memory from GPU (global memory) to the pointers.
  Syntax: cudaMalloc(Address of Pointer variable, Size of memory to be allocated)
  e.g. cudaMalloc((void **)&d, 6*sizeof(int)

- **Transfer data from host memory to device memory:** The host data is then copied to the device's memory, which was allocated using the cudaMalloc command used in the previous step. The API that's used to copy the data between the host and device and vice versa is cudaMemcpy.

Like other memcopy commands, this API requires the destination pointer, source pointer, and size. One additional parameter it takes is the direction of copy, that is, whether we are copying from the host to the device or from the device to the host.

Function cudaMemcpy () is provided by header file cuda.h. It is used to transfer some contents from one memory location to another memory location.

Syntax: cudaMemcpy(Destination Pointer Variable, Source Pointer Variable, size of memory, cudaMemcpyHostToDevice / cudaMemcpyDeviceToHost)

e.g. cudaMemcpy(d,a,sizeof(int),cudaMemcpyHostToDevice);

Note:

cudaMemcpyHostToDevice is used to transfer values from Host (CPU) to Device (GPU).

cudaMemcpyDeviceToHost is used to transfer values from Device (GPU) to Host (CPU).

2. **Load GPU Program and Execute**

➢ **Call and execute a CUDA function:** As shown in the Hello World CUDA program, we call a kernel by using <<< >>> brackets, which provide parameters for the block and thread size, respectively.

➢ **Synchronize:** As we mentioned in the Hello World program, kernel calls are asynchronous in nature. In order for the host to make sure that kernel execution has finished, the host calls the cudaDeviceSynchronize function. This makes sure that all of the previously launched device calls have finished.

3. **Copy results from CPU Memory to GPU Memory**

➢ **Transfer data from host memory to device memory:** Use the same cudaMemcpy API to copy the data back from the device to the host for post-processing or validation duties such as printing. The only change here, compared to the first step, is that we reverse the direction of the copy, that is, the destination pointer points to the host while the source pointer points to the device allocated in memory.

➢ **Free the allocated GPU memory:** Finally, free the allocated GPU memory using the cudaFree API. cudaFree (device variable)

With this knowledge we can write a program for vector addition , but before that we have to understand about 1D,2D (threads ,blocks) and the terminology associated with it.

threadIdx = Used to access the index of a thread inside a thread block

threadIdx.x = Index of a thread inside a block in X direction(1D) threadIdx.y = Index of a thread inside a block in Y direction(2D)

threadIdx.z = Index of a thread inside a block in Z direction(3D)

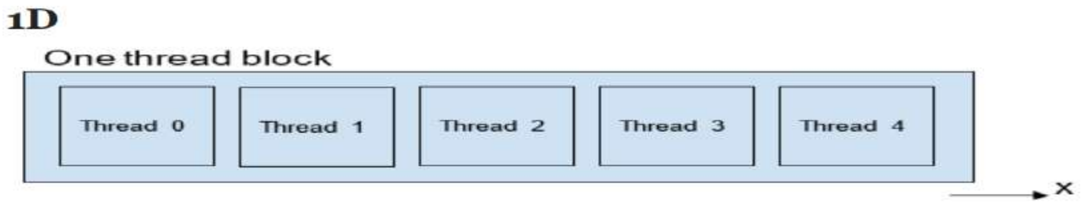
blockDim = Number of threads in the block for a specific direction

blockDim.x = Number of threads in the block for X direction(1D)

blockDim.y = Number of threads in the block for Y direction(2D)

blockDim.z = Number of threads in the block for Z direction(3D)

Let's see how those are being used in the context.



It is 1D block: meaning 1block with 5 threads.

Example:

For **thread 0**, *threadIdx.x = threadIdx.y = threadIdx.z = 0*.

thread 3, *threadIdx.x = 3 and threadIdx.y = threadIdx.z = 0*.

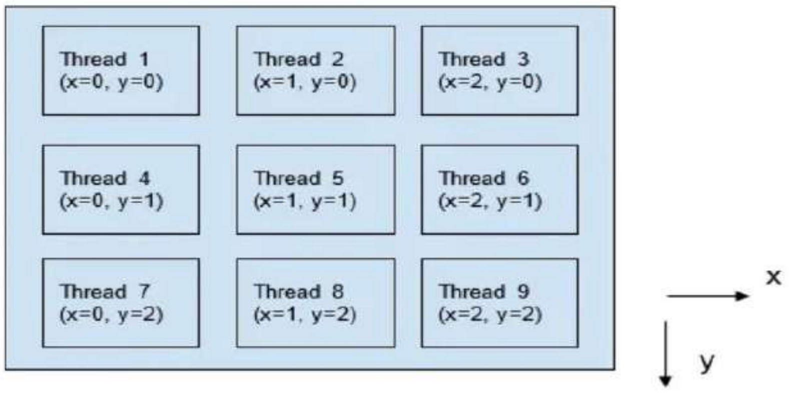
And *blockDim.x = 5*

Vector addition using CUDA

The problem that we are trying to solve is vector addition. As we are aware, vector addition is a data parallel operation. Our dataset consists of three arrays: A, B, and C. The same operation is performed on each element: C[i] = A[i]+ B[i].**Assuming 5 values , which can be implemented either by multiple threads in one block or multiple blocks with one thread each**

```
%%cu
 #include<stdio.h>
#include <cuda.h>
global__ void add(int *d,int *e,int *f)
{
int id;
 id=threadIdx.x; //blockIdx.x for Multiple blocks
 f[id]=d[id]+e[id];
 }
int main()
{
int a[5]={1,2,3,4,5},b[5]={1,1,1,1,1},c[5],*d,*e,*f,i;
cudaMalloc((void **)&d,5*sizeof(int));
cudaMalloc((void **)&e,5*sizeof(int));
cudaMalloc((void **)&f,5*sizeof(int));
cudaMemcpy(d,&a,5*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(e,&b,5*sizeof(int),cudaMemcpyHostToDevice);
add<<1,5>>(d,e,f);  //one block with multiple threads
//add<<<5,1>>>(d,e,f); multiple blocks with one thread each
cudaMemcpy(&c,f,5*sizeof(int),cudaMemcpyDeviceToHost);
 for(i=0;i<5;i++)
printf("%d\t",c[i]);
cudaFree(d);
cudaFree(e);
cudaFree(f);
cudaDeviceSynchronize();
 return 0;
}
```

2D Thread Block(one block with 2D threads)

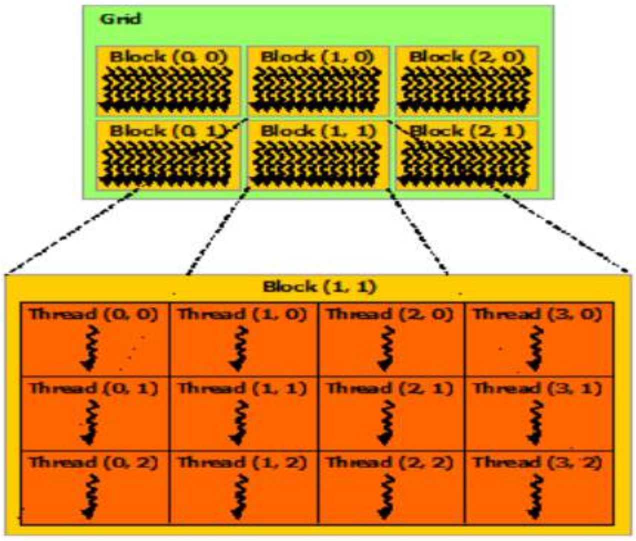


For **thread 1**, *threadIdx.x = threadIdx.y = threadIdx.z = 0.*

For **thread 6**, *threadIdx.x = 2, threadIdx.y = 1 and threadIdx.z = 0.*

And also *blockDim.x=3 and blockDim.y=3.*

Grid: Containing Multiple blocks



Number of blocks in the grid:

gridDim.x — number of blocks in the x dimension of the grid (eg: 3)

gridDim.y — number of blocks in the y dimension of the grid (eg:2)

Number of threads in a block:

blockDim.x — number of threads in the x dimension of the block (eg:4)

blockDim.y — number of threads in the y dimension of the block (eg:3)

Block Index:

blockIdx.x — block's index in x dimension

blockIdx.y — block's index in y dimension

eg: block (0,1) — blockIdx.x = 0 , blockIdx.y = 1

Thread Index:

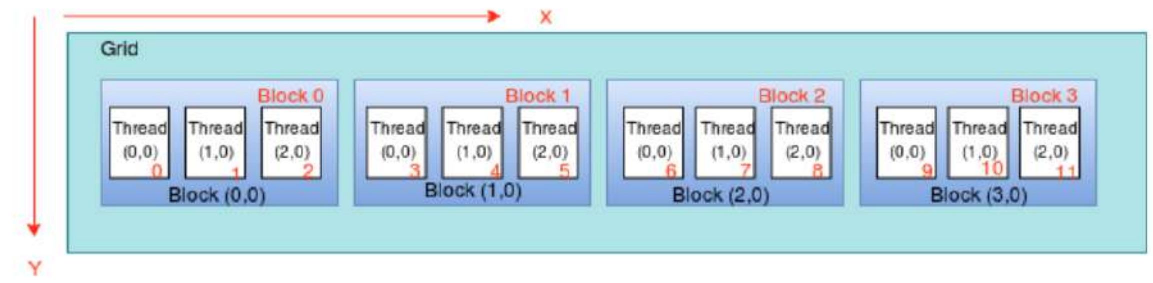
ThreadIdx.x — thread's index in x dimension

ThreadIdx.y — thread's index in y dimension

eg: Thread(2,1) — ThreadIdx.x = 2, ThreadIdx.y = 1

Inorder to get a unique number for each thread and each block in a grid we use the above indexing methods.Lets see an example

1D grid of 1D blocks



Indices given in RED color are the unique numbers for each block and each thread

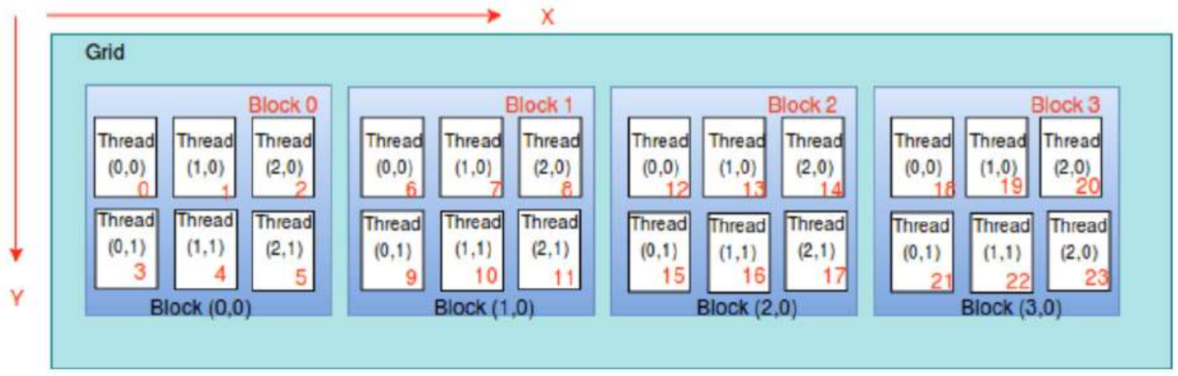
threadId = (blockIdx.x * blockDim.x) + threadIdx.x

Let's check the equation for Thread (2,0) in Block (1,0)

Thread ID = (1 * 3) + 2 =3+2 = 5

1D grid of 2D blocks

Here we have two ways for indexing.



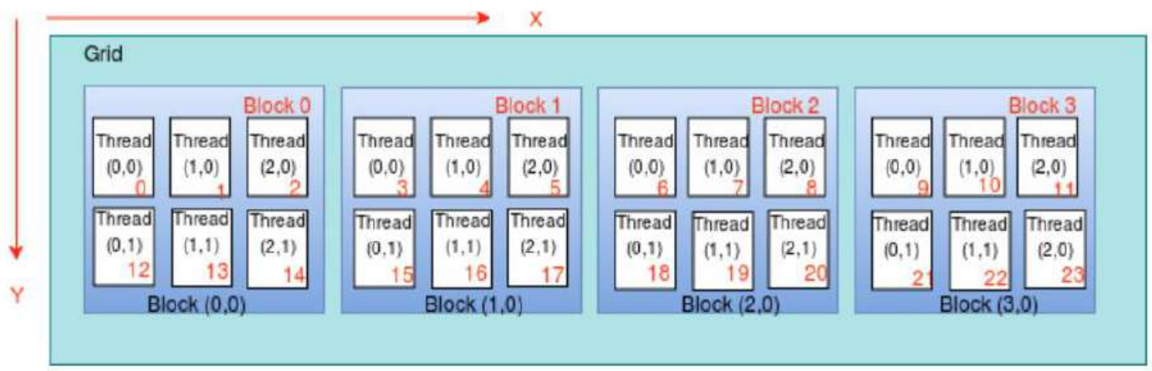
Indices given in RED color are the unique numbers for each block and each thread

threadId = (blockIdx.x * blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x

Let's check the equation for Thread (2,1) in Block (1,0).

Thread ID = (1*3*2)+(1*3)+2 = 6+3+2 =11

Here is the second method.



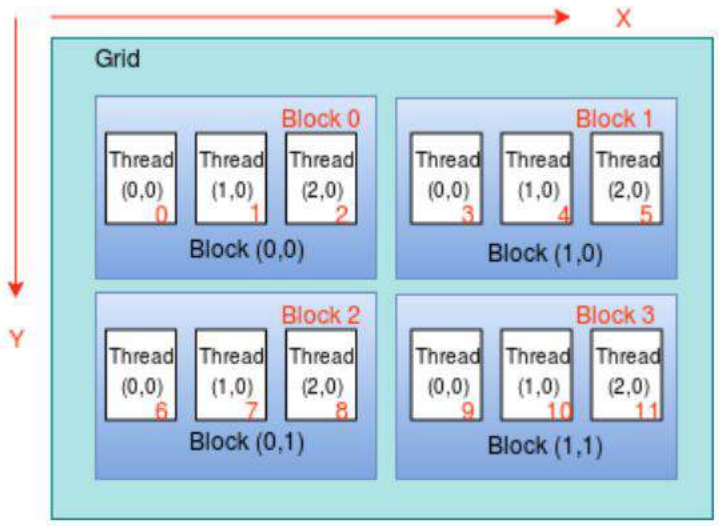
Indices given in RED color are the unique numbers for each block and each thread

threadId = (gridDim.x * blockDim.x * threadIdx.y) + (blockDim.x * blockIdx.x) + threadIdx.x

Let's check the equation for Thread (2,1) in Block (1,0).

Thread ID = (4 * 3 * 1) +(1 * 3)+2 = 12+3+2 =17

2D grid of 1D blocks



Indices given in RED color are the unique numbers for each block and each thread

blockId = (gridDim.x * blockIdx.y) + blockIdx.x

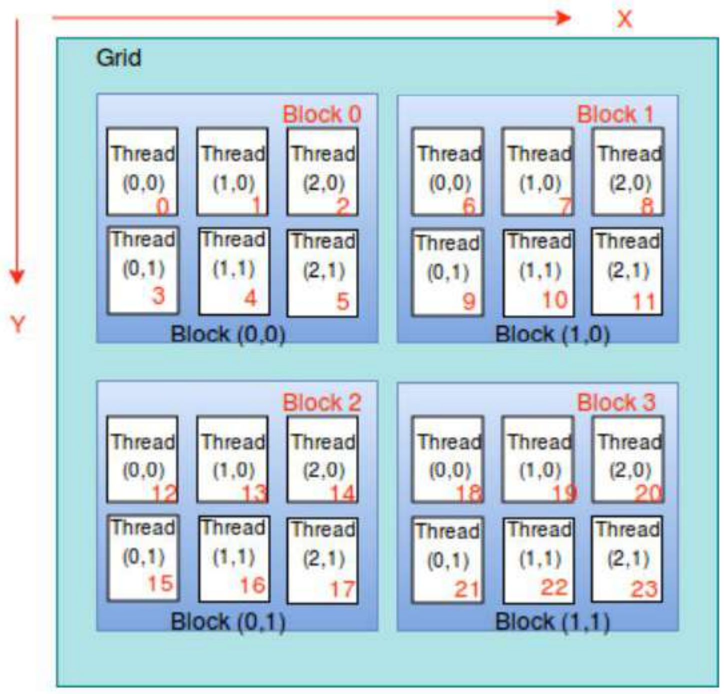
threadId = (blockId * blockDim.x) + threadIdx.x

Let's check the equation for Thread (1,0) in Block (1,1).

blockId = (2*1) + 1 =2+1=3

threadID = (3*3)+1 =9+1=10

2D grid of 2D blocks



Indices given in RED color are the unique numbers for each block and each thread

blockId = (gridDim.x * blockIdx.y) + blockIdx.x

threadId = (blockId * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx.x

Let's check the equation for Thread (2,1) in Block (0,1).

block Id = (2 * 1) + 0 = 2

Thread Id = (2 * (3 * 2))+(1*3) + 2 = 12+3+2 = 17

Matrix addition(2D block with one thread in each block)

```
%%cu
 #include<stdio.h>
#include <cuda.h>
global void add(int *d,int *e,int *f)
{
int x=blockIdx.x;
 int y=blockIdx.y;
int id=gridDim.x*y+x;
f[id]=d[id]+e[id];
}
int main()
{
int a[2][3]={{1,2,3},{4,5,6}},b[2][3]={{1,2,3},{4,5,6}},c[2][3],*d,*e,*f;
cudaMalloc((void**)&d,6*sizeof(int));
cudaMalloc((void**)&e,6*sizeof(int));
cudaMalloc((void **)&f,6*sizeof(int));
cudaMemcpy(d,&a,6*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(e,&b,6*sizeof(int),cudaMemcpyHostToDevice);
dim3 grid(3,2);
add<<<grid,1>>>(d,e,f);
cudaMemcpy(&c,f,6*sizeof(int),cudaMemcpyDeviceToHost);
for(int i=0;i<2;i++)
for(int j=0;j<2;j++)
printf("%d\t",c[i][j]);
printf("\n");
cudaFree(d);
cudaFree(e);
cudaFree(f);
cudaDeviceSynchronize();
return 0;
}
```

In CUDA programming, **dim3** is a built-in data type used to represent three-dimensional sizes or indices. It is often used to specify the dimensions of a grid or block in CUDA kernels.

Matrix addition(one block with 2D thread)

```
%%cu
#include<stdio.h>
#include <cuda.h>
global void add(int *d,int *e,int *f)
 {
int x=threadIdx.x;
int y=threadIdx.y;
int id=blockDim.x*y+x;
 f[id]=d[id]+e[id];
 }
int main()
{
int a[2][3]={{1,2,3},{4,5,6}},b[2][3]={{1,2,3},{4,5,6}},c[2][3],*d,*e,*f;
cudaMalloc((void**)&d,6*sizeof(int));
cudaMalloc((void**)&e,6*sizeof(int));
cudaMalloc((void **)&f,6*sizeof(int));
cudaMemcpy(d,&a,6*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(e,&b,6*sizeof(int),cudaMemcpyHostToDevice);
dim3 threadblock(3,2);//
add<<<1,threadblock>>>(d,e,f);
cudaMemcpy(&c,f,6*sizeof(int),cudaMemcpyDeviceToHost);
for(int i=0;i<2;i++)
for(int j=0;j<2;j++)
printf("%d\t",c[i][j]);
printf("\n");
cudaFree(d);
cudaFree(e);
cudaFree(f);
cudaDeviceSynchronize();
return 0;
}
```

Error reporting in CUDA

When working with CUDA (Compute Unified Device Architecture) for GPU programming in C or C++, error reporting is crucial for debugging and ensuring the correct execution of your GPU-accelerated code. CUDA provides several mechanisms for error reporting.

- ➢ The **cudaGetErrorString** function can be used to obtain a human-readable error message for a given CUDA error code. After each CUDA API call, you can check for errors and print a corresponding error message if an error occurs.
- ➢ **cudaSuccess** is a constant defined in the CUDA API that represents the successful completion of a CUDA API function call. When a CUDA function is executed without encountering any errors, it returns **cudaSuccess** as its result.
- ➢ The **cudaGetLastError** function is used to check for errors that occurred during the execution of the most recent kernel launch.

Example:

```
cudaError_t cudaStatus;
cudaStatus = cudaSomeFunction();
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "CUDA API call failed: %s\n", cudaGetErrorString(cudaStatus));
  // Additional error handling or cleanup can be done here
}
```

Data type support in CUDA

CUDA (Compute Unified Device Architecture) supports several data types for programming on GPUs. These data types are similar to those in C/C++ but are extended to include types specific to GPU programming. CUDA programming supports all of the standard data types that developers are familiar with in terms of their respective languages.

Along with standard data types with different sizes (char is 1 byte, float is 4 bytes, double is 8 bytes, and so on), it also supports vector types such as float2 and float4. It is recommended that the data types are naturally aligned since aligned data access for data types that are 1, 2, 4, 8, or 16 bytes in size ensure that the GPU calls a single memory instruction. If they are not aligned, the compiler generates multiple instructions, which are interleaved, resulting in inefficient utilization of the memory and instruction bus. Due to this, the recommendation is to use types that are naturally aligned for data residing in GPU memory. The alignment requirement is automatically fulfilled for the built-in types of char, short, int, long, long long, float, and double such as float2 and float4.

Also, CUDA programming supports complex data structures such as structures and classes (in the context of C and C++). For complex data structures, the developer can make use of alignment specifiers to the compiler to enforce the alignment requirements, as shown in the following code:

```
struct __align__(16)
{
float r; int g;
}
```

In the above example, the **__attribute__((aligned(16)))** aligns the structure to a 16-byte boundary. This means that the beginning of the structure and each member within the structure will be at an address that is a multiple of 16 bytes.