HPC Architectures

HPC: It's the use of **multiple tightly-coupled processors** or computer clusters to run **concurrently** computational-intensive tasks with high throughput and efficiency. Present day HPC are at the top level **distributed memory parallel machines**. They may be thought of as vast clusters of processing nodes (PNs) **interconnected by some network topology**. **Scalability** refers to the capability of a system to handle a growing amount of work (typically by attaching more nodes or improving a node's hardware resources).

Heterogeneous computing: refers to a computing system that utilizes **different types of processor, typically combining traditional CPUs and GPUs** to achieve superior performance. These systems aim to leverage the unique strengths of both CPUs and GPUs. CPUs are optimized for **dynamic workloads** with **short sequences** of computational operations and **unpredictable flow control**, and are responsible for managing the environment, code, and data for the GPU. The GPUs focus on workloads that are dominated by computational tasks with **simple flow control**. This arrangement allows the processing tasks to be distributed based on their nature, maximizing efficiency and performance.

Parallel decomposition: In **fine-grained parallelism**, parallel operations are expressed at the **variable level**, it assumes an instruction is executed simultaneously on multiple data sets, a SIMD (single instruction – multiple data) architecture is considered.
In **medium-grained parallelism,** parallel operations are expressed at the **thread level within a process**, a MIMD (multiple instruction – multiple data) architecture of the **shared memory** type is considered. Finally, in **coarse-grained parallelism,** parallel operations are expressed at the process level, a MIMD architecture of the **distributed memory** type is considered.

A **program** is a set of instructions written to perform a specific task or tasks. It is a **passive** entity stored on the disk until it is loaded into memory to be executed.
A **process** is an **instance of a program** that is **being executed**. It is an active entity with a program counter specifying the next instruction to execute and a set of associated resources. A single program can have multiple processes running from it, each with its own memory space and resources. A **thread** is the **smallest unit of execution within a process.** A single process can have multiple threads, all of which share the process's memory and resources. Each thread has its own program counter, stack, and set of registers. Threads within a process can communicate with each other more easily than separate processes can, as they share the same memory space.

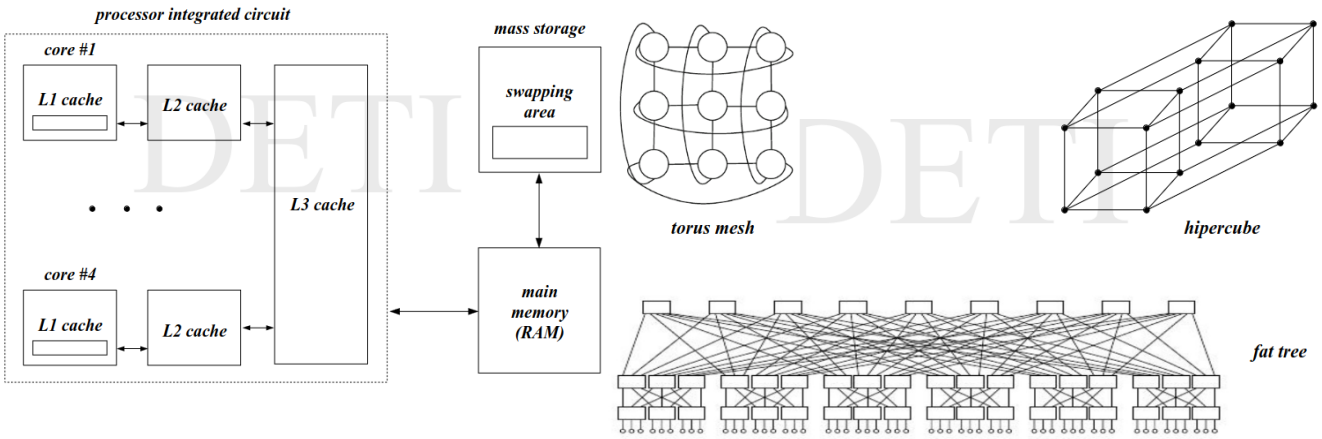
Process/Thread states: created; ready-to-run; run; blocked; terminated.

Main components of a Processing Node:

- **CPU:** is the brain of the computer. It performs the actual **processing** of data, **executing instructions** and **controlling** the **operation** of the computer.
- **Main memory:** It **temporarily** holds the data and instructions that the CPU needs to execute tasks. However, the data in main memory is lost if power is interrupted, as it is **volatile**.
- **Mass memory:** stores data in-between processing runs. This is where files and programs are stored when not in use. Unlike main memory, mass memory is

non-volatile, meaning it **retains information** even when the computer is powered off. This memory typically has a larger capacity but slower access speed compared to main memory.

- **I/O (Input/Output):** This refers to the systems and processes that **move data between the computer and its external environment**. This could include **keyboards** and mice (input devices), or **monitors** and **printers** (output devices).
- **System interconnection:** ensures data communication takes place among the CPU, memory, I/O and other devices. It's usually implemented as a bus.



HPC Architectures:

The main properties that one expects from interconnection topologies are:

1. **Scalability**: The ability for the system performance to increase as new nodes are attached to it.
2. **Connection Simplicity**: Ideally, the number of connections per node should be kept small as the number of processing nodes increases.
3. **Communication Efficiency**: Ideally, communication time and bandwidth should be kept constant as the number of processing nodes increases.

Torus Mesh: A Torus mesh interconnects nodes in a **grid-like fashion** where the grid **wraps around in both dimensions creating a doughnut shape** (hence the name "torus"). Each node is **connected to four others**, so the number of **connections** per node remains **constant** as the network grows, satisfying the **connection simplicity** property. However, **communication time can increase up to** $\sqrt{n}$ (where n is the number of nodes), especially for nodes on opposite sides of the network, which can affect communication efficiency.

Hypercube: A Hypercube connects nodes in a **multidimensional**, **binary logarithmic structure**. Here, each node is connected to **log2 n nodes** (where n = 2k and k is the number of dimensions), so the **number of connections increases slowly** with the number of nodes. The **maximum communication time** between any two nodes is **log2 n**, meaning it **scales better** in terms of communication time than a Torus Mesh.

Fat Tree: A Fat Tree is a hierarchical network with the **ability to keep the same bandwidth at all bisections**, satisfying the **communication efficiency** property. All nodes transmit at the same speed if the packets are uniformly distributed along the available paths. With a **single connection per node** and the usage of k-port switches, k3/4 nodes can be attached, showing **good scalability** properties.

HPC are basically **distributed-memory parallel machines** since they consist of multiple nodes and each node has multiple processors with its own private memory. This contrasts with shared-memory systems, where all processors share access to a common pool of memory. In a distributed-memory system, if a processor needs to access data that's not in its own private memory, it must communicate with the other node that holds that data.

Communication overhead refers to the extra time and resources needed for processors to exchange data which can limit the speedup gained from parallelization. To reduce this problem: **minimize redundant communication**; use **batch communication**; **choose** the **best interconnection topology** for the task at hand; **overlap communication with computation**; balance the computation-to-communication ratio by making tasks large enough that the computation time is significantly larger than the communication time. This is known as increasing the **granularity of** the **tasks**.

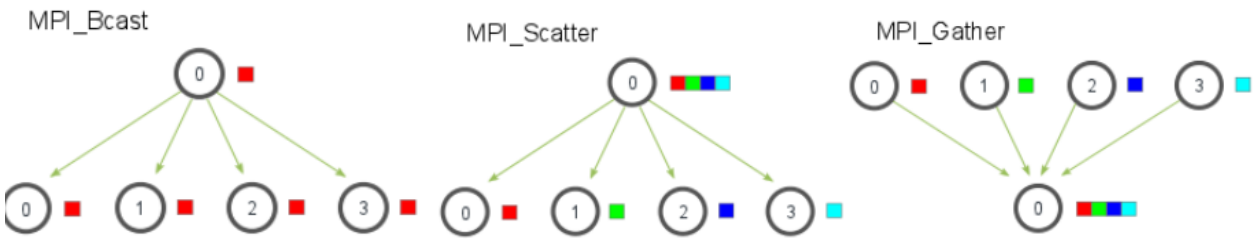
MPI

MPI provides a way for parallel programs to exchange data and synchronize their execution in a **coordinated manner**. It allows programmers to write **distributed-memory parallel programs** by explicitly specifying **communication operations**, such as sending and receiving messages, and defining collective operations, such as reductions and broadcasts.

A communicator allows communication with all processes that are accessible after MPI initialization. Processes are identified by their rank in the group of MPI_COMM_WORLD.

Collective communication addresses the case where multiple processes exchange messages. There are several types of collective communication:

broadcast: a message is sent from a root process to all the processes (processes block until msg is received)), itself included.
scatter: specific messages of similar type are sent from a root process to all the processes (processes block until msg is received), itself included.
gather: specific messages of similar type are sent from all the processes to a root process (root gets blocked).
reduce: allows root process to perform a global calculation by aggregating (reducing) the values of a variable from all processes (root blocks until calculation is concluded).



non-blocking communication is restricted to point-to-point communication (sends and receives). non-blocking operations are not synchronized which means they return immediately without confirming the operation status (if the message was successfully delivered/received). This has the advantage of allowing a program to overlap computation with communication, potentially leading to significant performance improvements.

To assert that a non-blocking operation has completed, MPI provides separate calls, namely **MPI_Wait** and **MPI_Test**. MPI_Wait is used to **wait for the completion** of a non-blocking send or receive operation. It **effectively blocks** until the operation completes. MPI_Test is used to **check** if a non-blocking send or receive **operation has completed**, but unlike MPI_Wait, it **does not block**.

Collective synchronization addresses the problem of ensuring that multiple processes block waiting for one another to reach the same point of program execution. **Barrier** works by only lifting when all processes reach the same point of program execution. All block, except the last one to reach that point, which wakes up the rest and allows all of them to proceed.

Q1. Explain how barrier synchronization works.
Sometimes, a group of processes identified by the MPI_COMM_WORLD need to be synchronized. Each process in the group independently executes its part of the program until it reaches the barrier statement. As each process reaches the barrier, it will block execution. Eventually, the last process in the group will reach the barrier. As soon as the last process reaches the barrier, the barrier is lifted - this is like a signal to all the processes that were blocked waiting at the barrier that they can now proceed with their execution. After the barrier is lifted, all the processes that were blocked at the barrier can now continue executing their instructions. This ensures that all processes in the group are synchronized at the point in the program where the barrier was placed.

Q2. One of the communication paradigms that can be used is the scatter-gatter paradigm. Explain how it works.

1. **Scatter**: In the scatter operation, the root process (say, process 0) has an array of data. This data array is divided into equal parts and then each part is sent to a different process in the system, including itself. For example, if the root process has an array of 20 elements, in the case of 5 processes, it would divide the array into 5 chunks of 4 elements each and send each chunk to one of the 5 processes. So, process 0, process 1, process 2, process 3, and process 4 each get a chunk of 4 elements.
2. **Gather**: In the gather operation, the reverse of scatter happens. Each process, after doing its own computation on its local data, sends its result back to the root process. The root process collects (gathers) all the results from all the processes (including itself) and concatenates them in order, typically in the order of process ranks.

Q3. A basic concept in MPI is a group of processes that will cooperate in running an application. How is the group usually instantiated?

When an MPI application is launched, it creates a group of processes that can communicate with each other. This default group is called MPI_COMM_WORLD. This is a pre-defined communicator that represents all the processes that are launched together when the MPI application starts. Each process in the application has a unique identifier called its rank. The rank is an integer ranging from 0 to N-1, where N is the total number of processes. The rank is assigned by the system, typically in the order that processes are created.

Q4. For the application to run efficiently and to take full advantage of the processing power that is available, one must minimize interprocess communication. Why is it so?

Communication Overhead: Interprocess communication involves transmitting data from one process to another, which carries with it a certain amount of overhead. This overhead includes the time taken to package the data, transmit it, and then unpack it at the receiving end.
Synchronization Costs: When processes need to communicate, they often need to synchronize their operations. If a process is frequently waiting for data to be sent or received, it is not being fully utilized for computation.
Network Bandwidth and Latency: High levels of interprocess communication can consume a significant amount of network bandwidth resulting in network congestion.

Q5. Suppose 8 processes cooperate in computing the product of two order 4 square matrices. Both the operands and the result are stored in memory. Describe how the computation is divided among the processes and what kind of primitives will be used.

Step 1: Divide the work

- Each matrix has 4 rows and 4 columns. We can use 4 out of 8 processes available.
- Assign each of the 4 processes one row from the first matrix and the entire second matrix.

Step 2: Matrix Multiplication

- Each process multiplies its assigned row from the first matrix with every column in the second matrix to compute the corresponding row in the result matrix.

MPI_Scatter: We can use the MPI_Scatter operation to distribute the rows of the first matrix to the 4 processes. The root process that initially holds the entire first matrix, scatters rows of the first matrix across the 4 processes.

MPI_Bcast: We use the MPI_Bcast operation to broadcast the entire second matrix to all 4 processes. The root process that holds the second matrix broadcasts it to all the other processes.

MPI_Gather: After each process has computed its assigned row in the result matrix, we need to gather these rows back to one process (for example, again the root process) to form the complete result matrix.

Multithreading

A **thread** is a unit of execution that the operating system can schedule independently of other threads. It is a subset of a process and multiple threads can exist within the same process, sharing resources such as memory, while executing independently. **Multithreading** is a technique that allows multiple threads to exist within the context of a single process. This means that a single process can have multiple threads running concurrently, with each thread performing a different task. Multithreading is used to maximize utilization of processor time and improve the efficiency of an application.

a resource is something a process needs to execute. Resources may either be physical components or common data structures.

- **preemptable resources:** when they can be taken away from the processes that hold them, without any malfunction resulting from the fact; the processor and regions of the main memory where a process addressing space is stored, are examples of this class in multiprogrammed environments
- **non-preemptable resources:** when it is not possible; the printer or a shared data structure, requiring mutual exclusion for its manipulation, are examples of this class.

A condition variable is a mechanism that allows threads to wait (without wasting CPU cycles) for some even to occur. Several threads may wait on a condition variable, until some other thread signals this condition variable (thus sending a notification). At this time, one of the threads waiting on this condition variable wakes up, and can act on the event.

A mutex is a lock that guarantees three things:

1. Atomicity - Locking a mutex is an atomic operation, meaning that the operating system (or threads library) assures you that if you locked a mutex, no other thread succeeded in locking this mutex at the same time.
2. Singularity - If a thread managed to lock a mutex, it is assured that no other thread will be able to lock the thread until the original thread releases the lock.
3. Non-Busy Wait - If a thread attempts to lock a thread that was locked by a second thread, the first thread will be suspended (and will not consume any CPU resources) until the lock is freed by the second thread. At this time, the first thread will wake up and continue execution, having the mutex locked by it.

Imposing mutual exclusion on access to a resource, or to a shared region, can have, by its restrictive character, two undesirable consequences:

- **deadlock / livelock:** occurs when two or more processes are indefinitely waiting for each other to release resources. Each process is holding a resource that the other needs to complete its operation, resulting in a circular wait. Consequently, all the processes involved are blocked, and no operations can proceed.
- **indefinite postponement:** it happens when one or more processes compete for the access to a critical region and, due to a conjunction of circumstances where new processes come up continuously and compete with the former for this goal, access is successively denied; one is here, therefore, facing a real obstacle to their continuation.

Deadlock prevention: in order to make deadlock impossible to happen, one has only to deny one of the necessary conditions to the occurrence of deadlock:

Mutual exclusion: access to the critical region associated to a given resource can only be allowed to a single process at a time. This condition too restrictive because it can only be denied for non-preemptable resources. Therefore, one of the other three conditions is usually denied to prevent the occurrence of deadlocks.
Waiting with retention: each process, upon requesting a new resource, holds all other previously requested and assigned resources. To deny this condition a process must request at once all the resources it needs for continuation.
Liberation of resources: a resource, once granted, cannot be forcibly taken away from a process until the process voluntarily releases it. To deny this condition a process, when it can not get hold of all the resources it requires for continuation, must release all the resources in its possession and start later on the whole request procedure from the very beginning.
Circular waiting: a circular chain of processes and resources, where each process requests a resource which is being held by the next process in the chain, is formed. To deny this condition we must establish a linear ordering of the resources and to make the process, when it tries to get hold of the resources it needs for continuation, to request them in increasing order of the number associated to each of them.

Indefinite postponement prevention:

- **Fairness Policy**: Implement a scheduling policy that ensures that every process gets a fair chance to execute. For example, the Round Robin scheduling algorithm allows each process to run for a fixed time before moving to the next.
- **Priority Aging**: Increase the priority of waiting processes over time, so that they eventually get served. This prevents a low priority process from being starved by higher priority ones.
- **Resource Reservation**: Reserve a portion of the resource for the processes experiencing postponement, guaranteeing access at some point.

A **monitor** is a synchronization construct that allows threads to have both mutual exclusion (only one thread can execute in the monitor at a time) and the ability to wait (block) for a certain condition to become true. Monitors are a solution to avoid issues like race conditions and ensure safe access to shared resources in concurrent programming.
A monitor is an object that encapsulates shared data structures and the procedures that operate on them, along with synchronization between concurrent method invocations. Only one thread can execute within the monitor at a time, which ensures mutual exclusion.

The primary synchronization primitives of a monitor are condition variables. These primitives allow threads to block when certain conditions are not met and to wake up when the conditions are signaled. They are:

- **Wait**: The 'wait' operation causes the calling thread to be blocked on the specified condition variable. While waiting, the thread is placed outside the monitor to permit another thread to access the shared data.

- **Signal:** if there are blocked threads in the specified condition variable, one of them is woken up; otherwise, nothing happens.

To prevent the coexistence of multiple threads inside a monitor, a rule is needed which states how the contention arisen by a signal execution is resolved:

Hoare Monitors: The thread that calls 'signal' is put outside the monitor, allowing the awakened thread to proceed. This requires a stack to keep track of signaling threads.

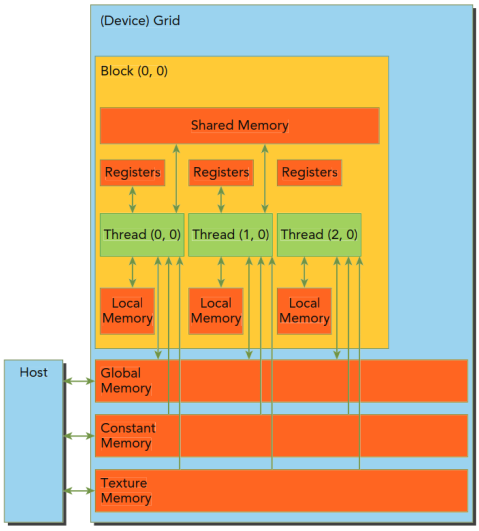
Brinch Hansen Monitors: The thread that calls 'signal' must immediately exit the monitor. This is simpler to implement but restricts the number of 'signal' calls to one.

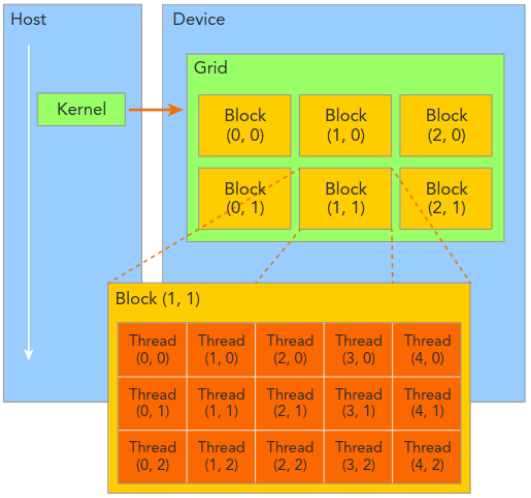
Lampson/Redell Monitors: The thread that calls 'signal' proceeds, while the awakened thread is kept outside the monitor and must compete for access again. This is also simple to implement but can lead to indefinite postponement of some threads.

CUDA

A heterogeneous environment consists of CPUs complemented by GPUs, each with its own memory
separated by a PCI-Express bus. Therefore, you should note the following distinction:
➤ Host: the CPU and its memory (host memory)
➤ Device: the GPU and its memory (device memory)

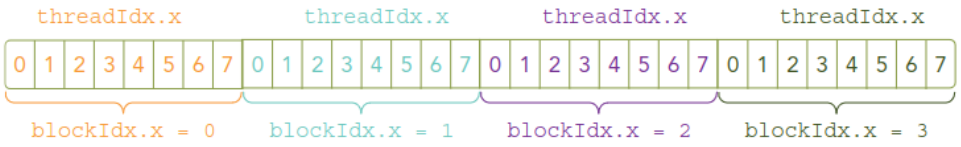
A typical CUDA program structure consists of five main steps:
• allocate GPU memory
• copy data from CPU memory to GPU memory
• invoke CUDA kernel to perform a program-specific computation
• copy data back from GPU memory to CPU memory
• destroy GPU memories.





All threads spawned by a single kernel launch are collectively called a grid. All threads in a grid share the same global memory space. A grid is made up of many thread blocks. A thread block is a group of threads that can cooperate with each other using:
➤ Block-local synchronization
➤ Block-local shared memory

Threads from different blocks cannot cooperate. Threads rely on the following two unique coordinates to distinguish themselves from each other:
➤ blockIdx (block index within a grid)
➤ threadIdx (thread index within a block)



The following restrictions apply for all kernels:
➤ Access to device memory only
➤ Must have void return type
➤ No support for a variable number of arguments

➢ No support for static variables
➢ No support for function pointers
➢ Exhibit an asynchronous behavior

Instructions within a single thread are pipelined to leverage instruction-level parallelism, in addition to the thread-level parallelism

The GPU architecture is built around a scalable array of Streaming Multiprocessors (SM).
CUDA employs a Single Instruction Multiple Thread (SIMT) architecture to manage and execute threads in groups of 32 called warps. All threads in a warp execute the same instruction at the same time. Each thread has its own instruction address counter and register state, and carries out the current instruction on its own data.
SIMT allows multiple threads in the same warp to execute independently.

Warps are the basic unit of execution in an SM. When you launch a grid of thread blocks, the thread blocks in the grid are distributed among SMs. Once a thread block is scheduled to an SM, threads in the thread block are further partitioned into warps. A warp consists of 32 consecutive threads and all threads in a warp are executed in Single Instruction Multiple Thread (SIMT) fashion; that is, all threads execute the same instruction, and each thread carries out that operation on its own private data.

When __syncthreads is called, each thread in the same thread block must wait until all other threads in that thread block have reached this synchronization point. All global and shared memory accesses made by all threads prior to this barrier will be visible to all other threads in the thread block after the barrier. The function is used to coordinate communication between threads in the same block, but it can negatively affect performance by forcing warps to become idle.

Benefits of a Memory Hierarchy
In general, applications do not access arbitrary data or run arbitrary code at any point-in-time. Instead, applications often follow the principle of locality, which suggests that they access a relatively small and localized portion of their address space at any point-in-time. There are two different types of locality:
➢ Temporal locality (locality in time)
➢ Spatial locality (locality in space)
Temporal locality assumes that if a data location is referenced, then it is more likely to be referenced again within a short time period and less likely to be referenced as more and more time passes.
Spatial locality assumes that if a memory location is referenced, nearby locations are likely to be referenced as well. Modern computers use a memory hierarchy of progressively lower-latency but lower-capacity memories to optimize performance. This memory hierarchy is only useful because of the principle of locality.

Constant memory resides in device memory and is cached in a dedicated, per-SM constant cache.
Texture memory resides in device memory and is cached in a per-SM, read-only cache. Texture memory is optimized for 2D spatial locality, so threads in a warp that use texture memory to access 2D data will achieve the best performance.

Global memory is the largest, highest-latency, and most commonly used memory on a GPU. The name global refers to its scope and lifetime. Its state can be accessed on the device from any SM throughout the lifetime of the application.

There is one L1 cache per-SM and one L2 cache shared by all SMs. Both L1 and L2 caches are used to store data in local and global memory.

Aligned memory accesses occur when the first address of a device memory transaction is an even multiple of the cache granularity being used to service the transaction.
Coalesced memory accesses occur when all 32 threads in a warp access a contiguous chunk of memory. To maximize global memory throughput, it is important to organize memory operations to be both aligned and coalesced.

QUESTIONS

Q1. A typical program written in CUDA has part of its operations performed by the host and part by the device. Describe them and signal which are the operations carried out by the host and the device.

1. **Allocate GPU memory (Host operation):** The host (CPU) initiates the process by allocating memory on the GPU device. The host controls the primary system memory (RAM), but for the GPU to perform operations, it needs its own memory space, which is why memory is allocated on the GPU.
2. **Copy data from CPU memory to GPU memory (Host operation):** Once memory has been allocated on the GPU, the host copies data from its memory (RAM) to the GPU's memory. This is necessary because the GPU cannot directly access the CPU's memory, so any data that the GPU needs for its calculations must be explicitly transferred over.
3. **Invoke CUDA kernel to perform a program-specific computation (Device operation):** The host launches a CUDA kernel on the GPU. A kernel is a function that the GPU executes. When this function is called, it is executed in parallel by many threads on the GPU, which allows it to perform a large number of computations simultaneously. The computations that the GPU performs are specific to the particular program or algorithm being run.
4. **Copy data back from GPU memory to CPU memory (Host operation):** After the GPU has finished its computations, the results need to be transferred back to the CPU. The host handles this by copying the data from the GPU memory back to its own memory. Again, this is necessary because the CPU cannot directly access the GPU's memory.
5. **Destroy GPU memories (Host operation):** Once the data has been safely transferred back to the CPU and is no longer needed on the GPU, the host releases the GPU memory that was allocated in step 1. This frees up the memory for other uses and helps to prevent memory leaks, which can cause performance issues and crashes.

This division of labor allows each component to do what it does best: the CPU handles control operations, while the GPU handles computationally intensive tasks.

Q2. When running a computational kernel in a GPU using CUDA, one has to specify a launching geometry for its invocation. How is the computation geometry organized and how does it actually affect code execution?

When launching a CUDA kernel on a GPU, the user has to specify a specific launching geometry. This geometry is primarily concerned with how threads and blocks of threads are organized and executed within the GPU. Here's a detailed explanation of this concept:

1. **Thread Block Organization (1D, 2D, or 3D):** The computational geometry in CUDA is organized in a hierarchy where threads are grouped into blocks, and these blocks are grouped into a grid. Threads within a block can be arranged in one, two, or three dimensions (this is specified when the kernel is launched). The choice of dimensions is often made based on the nature of the problem. For instance, if you are working with a 2D image, you might want to use a 2D block.
2. **Block Grid Organization (1D or 2D):** Similarly, blocks within a grid can be arranged in one or two dimensions (...)
3. **Selection of Thread and Block Sizes:** The size of a thread block (i.e., the number of threads in a block) and the number of blocks in a grid can significantly affect the performance of a CUDA program. These parameters can be tuned to maximize the usage of the GPU's computational resources. The optimal configuration can depend on a variety of factors, including the specific algorithm being implemented, the size and shape of the input data, and the specific GPU architecture being used.
4. **Execution of Threads and Blocks:** Once the launching geometry is defined and the kernel is launched, each block of threads is scheduled for execution on one of the GPU's Streaming Multiprocessors (SMs). The threads within a block are further divided into "warps" (groups of 32 threads), which are the basic unit of execution. All threads in a warp execute the same instruction at the same time in a Single Instruction, Multiple Threads (SIMT) fashion, which allows the GPU to execute operations in parallel across many threads.
5. **Interaction between Threads:** The threads within a block can cooperate by sharing data through shared memory and by synchronizing their execution through the use of __syncthreads() function. This enables fine-grained parallelism and efficient data sharing, but it also means that the choice of block size can affect the performance of the kernel, since a larger block size allows more threads to share data but also requires more shared memory and synchronization.

Q3. Suppose a block of threads has 115 threads. How is it divided for execution? How many blocks of threads may be concurrently executed in the same processor?

When a block of threads is executed on a GPU using CUDA, the threads within a block are divided into "warps". A warp is a group of threads, specifically 32 threads, which is the basic unit of scheduling on a CUDA-enabled GPU. The warps are managed by Streaming Multiprocessors (SMs), which execute the threads in a warp concurrently.

If you have a block of 115 threads, the GPU still divides this block into warps of 32 threads. In this case, you would have:

1. 3 full warps of 32 threads (96 threads total)
2. 1 partial warp of 19 threads

In terms of how many blocks of threads may be concurrently executed in the same processor, it depends on the **number of SMs in the GPU**, **the amount of shared memory used by each block:** Each SM has a certain amount of shared memory. If a block uses too much shared memory, fewer blocks can be scheduled on the same SM.

Q4. Explain how a program running in a processing node may be organized in order to take advantage of a GPU.

1. Identify Parallelizable Tasks
2. Design the program in such a way that it utilizes the CUDA programming model. This includes determining the structure of kernels and organizing threads into a hierarchy of grids and blocks.

Q5. Explain the internal organization of a Streaming Multiprocessor and explain what is the meaning of a WARP in this context.

Internal organization of an SM:

1. **Thread Organization:** Threads in a GPU are organized into a hierarchy: they are initially grouped into thread blocks, and these thread blocks are then organized into a grid, which is the highest level of the thread hierarchy. All threads in a grid share the same global memory space. The threads within a block can cooperate using block-local shared memory and synchronization.
2. **Thread Blocks and SMs:** The thread blocks from the grid are distributed among the available SMs in the GPU. Each SM is capable of running multiple thread blocks simultaneously. This distribution is done to ensure load balancing across the GPU.
3. **Warp Partitioning:** Once a thread block is assigned to an SM, threads within the thread block are further partitioned into smaller groups called warps. A warp consists of 32 consecutive threads.
4. **Single Instruction, Multiple Threads (SIMT):** The SIMT architecture that CUDA employs is a core part of the SM's function. In this architecture, all threads in a warp execute the same instruction at the same time, each on its own set of data. This enables the GPU to handle tasks in parallel, greatly increasing the processing speed for suitable tasks.

Now, to explain the meaning of a warp in this context:

A warp is the basic unit of execution in an SM. It is a set of 32 consecutive threads that are executed concurrently in a SIMT fashion. Essentially, all threads in a warp are given the same instruction to execute, but they each carry out the operation on their own unique data. The fact that all threads in a warp are executing the same instruction allows the SM to manage and execute them as a single entity, which greatly simplifies the scheduling and execution process.

Q6. When writing a computational kernel, one should be aware of two optimization factors to minimize execution time. Which are they?

Two critical optimization factors are:

1. **Memory Access Patterns (Coalescing and Aligning):** The way your program accesses memory can have a significant impact on its performance. Global memory accesses are fastest when they are coalesced. This means that when all threads in a warp (a group of 32 threads) access a contiguous chunk of memory, the memory accesses can be combined into a single transaction, speeding up the operation. This occurs when the threads access adjacent memory locations that fall on the boundary of the cache line (the unit of transfer between cache and memory), which typically corresponds to a certain number of bytes.
   Memory accesses are also optimized when they are aligned. This means the first address of a memory transaction should be a multiple of the cache granularity (size of cache line). Aligned memory access allows the GPU to load or store memory efficiently, improving the memory throughput.
   Coalesced and aligned memory access often go hand-in-hand, and both can significantly reduce the execution time of a CUDA kernel.
2. **Thread Utilization (Maximizing Occupancy):** The GPU's power comes from its ability to execute many threads in parallel. Therefore, to minimize execution time, it's crucial to keep as many of the GPU's cores as possible busy. This is often referred to as maximizing occupancy.
   To achieve high occupancy, you need to organize your threads into blocks and grids in a way that fully utilizes the GPU's streaming multiprocessors (SMs). The size of your thread blocks can affect how well the GPU's cores are utilized, so it's often necessary to experiment with different block sizes to find the one that works best for your specific algorithm.