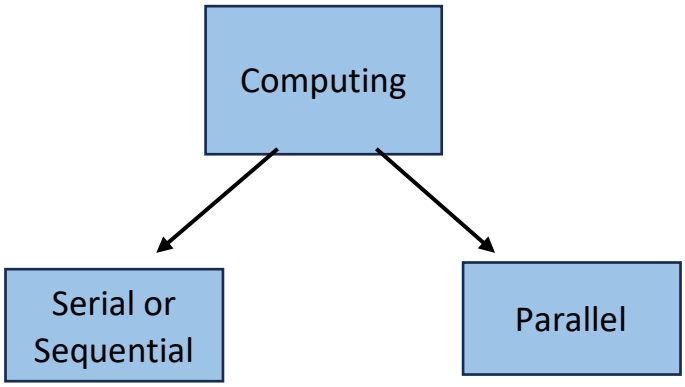
UNIT-1: TOPIC 1

Introduction To Parallel Computing

Basics

"**Computing**" refers to the process of using computers to perform various tasks, such as data processing, information storage, and solving complex problems.

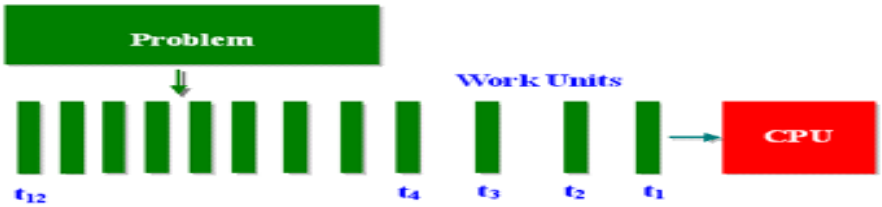
This computing can be either done in Serial Way known as **Serial Computing** or in Parallel Way known as **Parallel Computing**



Serial Computing:

Serial computing refers to traditional computing where tasks are executed sequentially, one after the other, using a single processor. In serial computing, each instruction or task must wait for the previous one to complete before it can be executed. This approach limits the speed and efficiency of processing, especially when dealing with complex or time-consuming tasks.

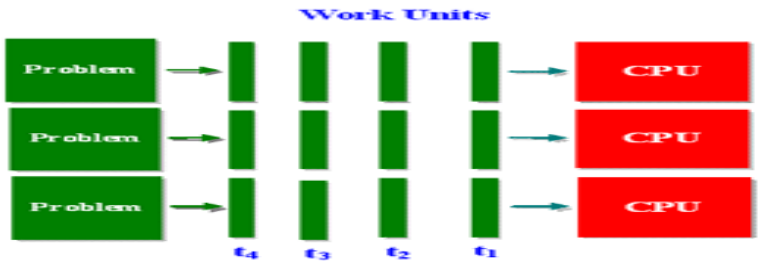
Example of Serial Computing: Consider a task of sorting a large dataset of numbers in ascending order. In a serial computing environment, a single processor would go through the entire dataset, comparing and rearranging numbers one pair at a time until the entire dataset is sorted. This process occurs sequentially, and each comparison and rearrangement must wait for the previous one to finish.



Parallel computing

Parallel computing is a type of computation in which multiple processors or computers work together to solve a problem. Instead of one single processor handling the entire task, parallel computing divides the task into smaller sub-tasks that can be processed simultaneously. This simultaneous processing can lead to significant improvements in computational speed and efficiency.

Example of Parallel Computing: Using the same example of sorting a large dataset, parallel computing would involve dividing the dataset into smaller chunks, and each chunk is sorted independently by a separate processor. These processors work in parallel, sorting their respective chunks simultaneously. Once all processors have completed sorting their portions, the sorted chunks can be combined to produce the final sorted dataset.



DEFINITION Parallel computing is the practice of identifying and exposing parallelism in algorithms, expressing this in our software, and understanding the costs, benefits, and limitations of the chosen implementation.

Benefits of Parallel Computing

- **Faster Run Time with More Compute Cores:** Parallelization involves dividing a task into smaller sub-tasks that can be executed simultaneously, utilizing multiple cores to process the data. This approach can significantly reduce the time required to complete the task, as each core works on a separate portion of the problem concurrently.
- **Larger Problem Sizes with More Compute Nodes:** With more nodes, you can break down your problem into smaller pieces that each node can work on simultaneously, which is especially beneficial for handling larger datasets and more complex simulations.
- **Energy Efficiency by Doing More with Less:** In the context of parallel computing, the concept of "doing more with less" often revolves around optimizing energy efficiency while achieving better computational performance.This can be achieved by making use of dynamic

resource allocation and workload consolidation to ensure that the number of processors used is proportional to the workload. Turn off or put to sleep any unused processors.

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

P = (N Processors) × (R Watts/Processors) × (T hours)

where P is the energy consumption, N is the number of processors, R is the thermal design power,and T is the application run time.

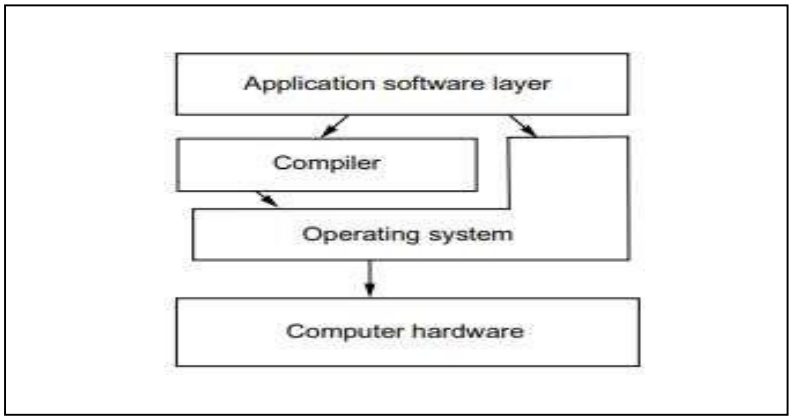
- **Scalability:** Parallel computing can be easily scaled by adding more processors, which further enhances performance. Serial computing does not scale in this manner, as it relies on a single processor.
- **Parallel Computing Can Reduce Costs:** As technology advances, the cost of individual processors and memory decreases. Parallel computing systems can take advantage of these cost reductions, making it more economical to build high-performance computing clusters or data centers.

Applications of Parallel Computing:

- **Scientific Simulations:** Used in fields such as physics, chemistry, and engineering for complex simulations.
- **Big Data Processing:** Parallel computing is crucial in processing vast amounts of data in fields like data analytics and machine learning.
- **Weather Forecasting:** Enables complex weather simulations and predictions.
- **Video and Image Processing:** Parallelism accelerates tasks like video rendering and image recognition.
- **Financial Modelling:** Used for risk analysis, option pricing, and other complex financial calculations.

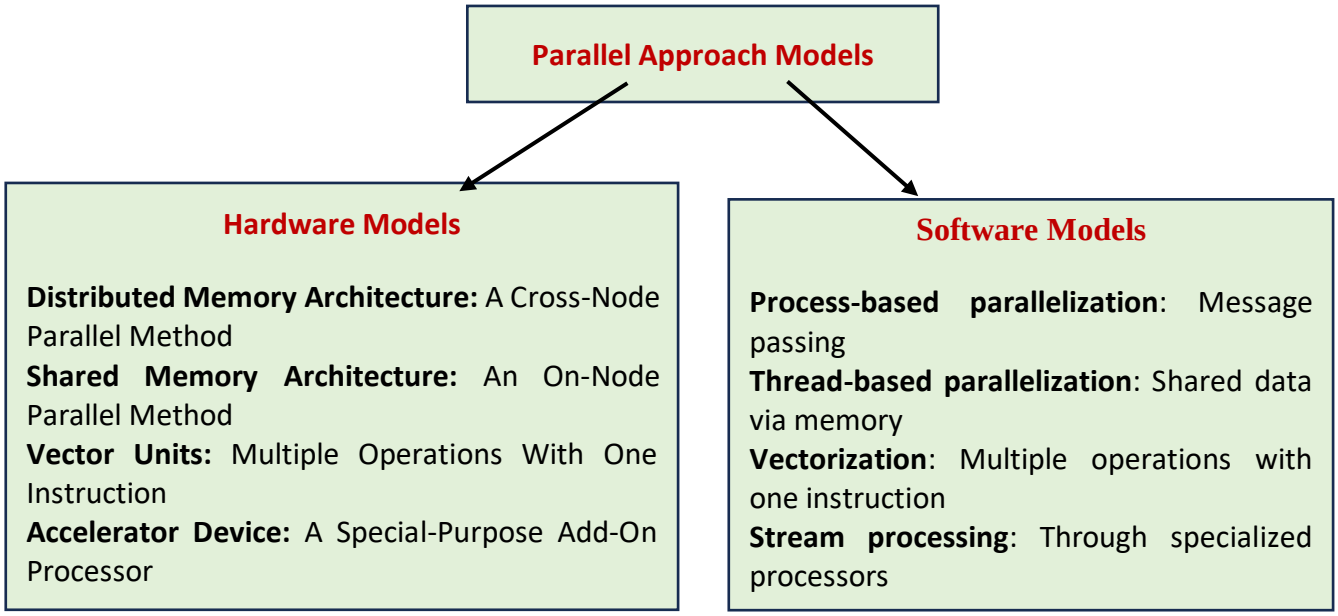
How parallel computing Works

As a developer, you are responsible for the application software layer, which includes your source code.



In the source code, you make choices about the programming language and parallel software interfaces you use to leverage the underlying hardware. Additionally, you decide how to break up your work into parallel units. A compiler is designed to translate your source code into a form the hardware can execute. With these instructions at hand, anOS manages executing these on the computer hardware.

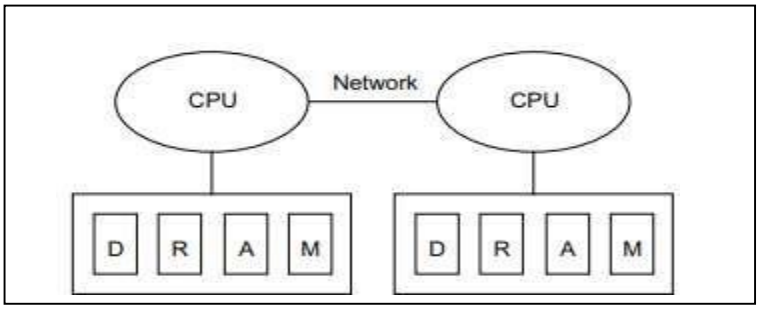
Parallel Approach models are used to express parallelization in an application software layer that gets mapped to the computer hardware through the compiler and the OS. Parallel computing approaches involve various models and paradigms that define how tasks are divided, coordinated, and executed in parallel systems. Here are some common parallel computing approach models:



Hardware Models

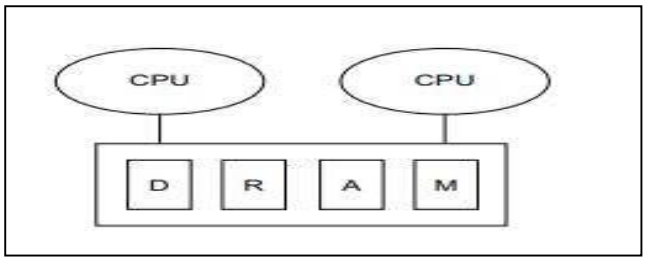
Distributed Memory Architecture: A Cross-Node Parallel Method**:**

Distributed Memory Architecture, also known as distributed memory parallelism, is a parallel computing method where multiple processors or nodes in a cluster have their own private memory. These nodes are connected via a network, and they communicate and coordinate with each other by passing messages. In this architecture, each node operates independently and has its own local memory, and data sharing is achieved explicitly through message passing.In the context of



distributed memory architecture, a "cross-node parallel method" refers to parallel processing techniques that involve distributing tasks across multiple nodes in a cluster. Each node works on its subset of the data or a specific portion of the computation. Communication and coordination between nodes are essential, as tasks often depend on results or data computed on other nodes.

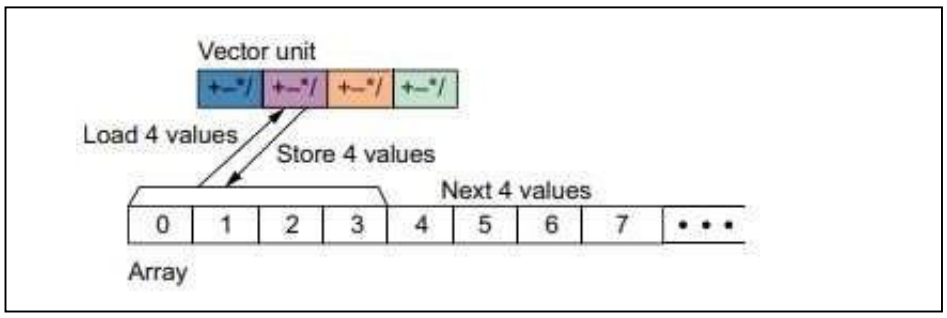
Shared Memory Architecture: An On-Node Parallel Method



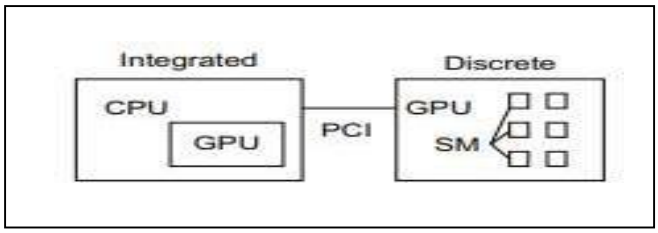
In shared memory architecture, multiple processors or cores share a single, unified memory space. This shared memory can be accessed and modified by any processor within the system. On-node parallelism, within the context of shared memory architecture, refers to parallel processing techniques that occur on a single computing node. In this approach, multiple threads or processes run concurrently on the same node, accessing shared memory to perform computations.

Vector Units: Multiple Operations With One Instruction Vector units, also known as vector processors, are specialized hardware units that can perform multiple operations with a single instruction. These units are designed to process vectors, which are arrays of data elements, simultaneously. Vector processing is particularly useful in scenarios where the same operation needs to be performed on a large set of data elements.

Vector processing example with four array elements operated on simultaneously



Accelerator Device: A Special-Purpose Add-On Processor



GPUs come in two varieties: integrated and discrete. Discrete or dedicated GPUs typically have a large number of streaming multiprocessors and their own DRAM. Accessing data ona discrete GPU requires communication over a PCI bus

An accelerator device, often referred to as an accelerator, is a specialized hardware component (GPU)designed to perform specific types of computational tasks or workloads efficiently. Accelerators are typically used in conjunction with a central processing unit (CPU) and are especially well-suited for workloads that can benefit from parallel processing and offloading certain tasks from the CPU. Accelerators are sometimes called "add-on processors" because they augment the processing capabilities of a system.

General Heterogeneous Parallel Architecture Model

Now let's combine all of these different hardware architectures into one model . Two nodes, each with two CPUs, share the same DRAM memory. Each CPU is a dual-core processor with an integrated GPU. A discrete GPU on the PCI bus also attaches to one of the CPUs. Though the CPUs share main memory, these are commonly in different Non-Uniform Memory Access (NUMA) regions. This means that accessing the second CPU's memory is more expensive than getting at it's own memory

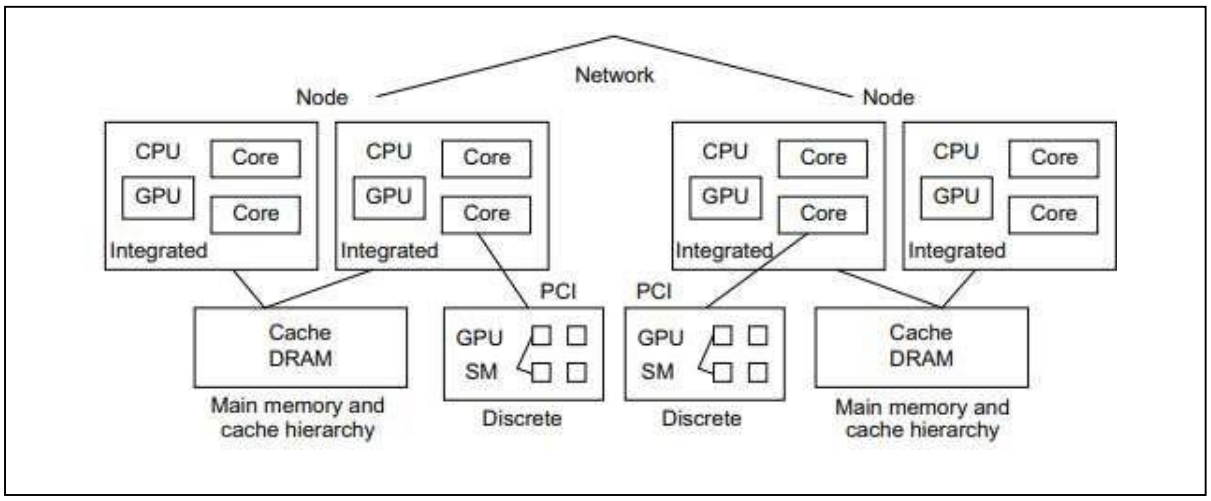
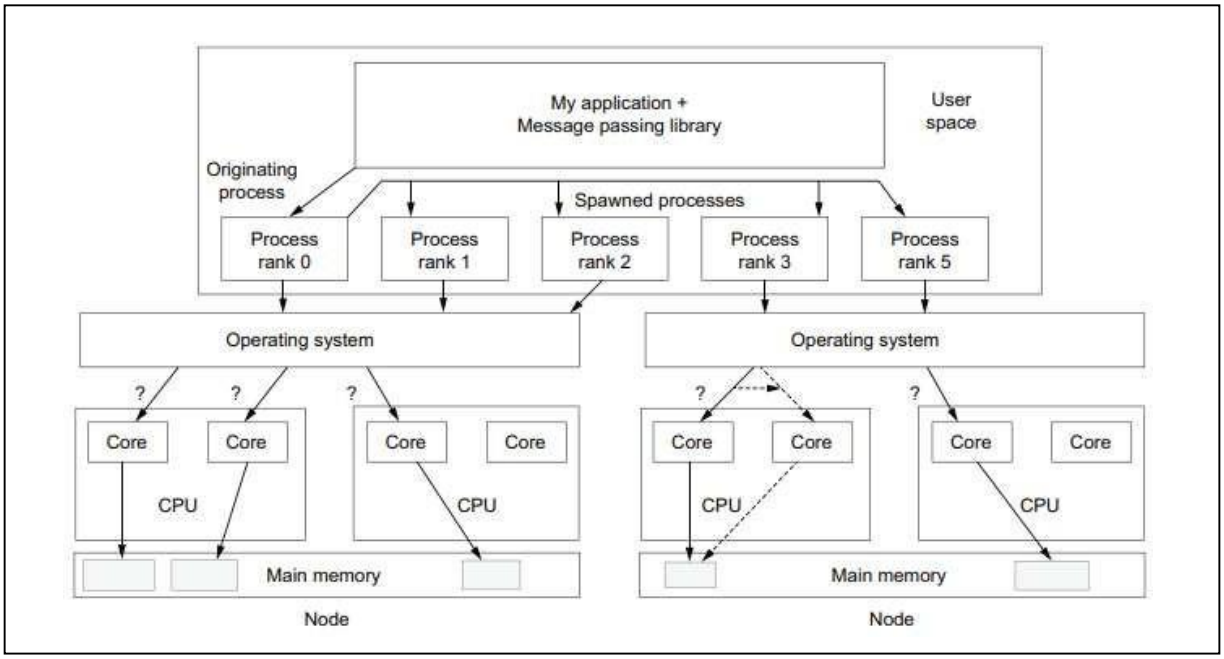


Fig 5: A general heterogeneous parallel architecture model consisting of two nodes connected by a network.Each node has a multi-core CPU with an integrated and discrete GPU and some memory (DRAM).

Software Models

- **Process-based parallelization**: Message passing Process-based parallelization, particularly through message passing, is a common approach in parallel computing. It involves dividing a task into multiple processes or threads that run independently on separate computing nodes or cores. These processes communicate and coordinate with each other by sending and receiving messages. Message passing is a method of inter-process communication where data and instructions are exchanged between processes to synchronize and share information. This approach is widely used in distributed memory systems, such as clusters and supercomputers.

Fig 6 : The message passing library spawns processes. The OS places the processes on the



cores of two nodes.The question marks indicate that the OS controls the placement of the processes and can move these during run time as indicated by the dashed arrows. The OS also allocates memory for each process from the node's main memory

- **Thread-based parallelization**: Shared data via memory

Thread-based parallelization involves dividing a task into multiple threads that share the same memory space within a single process. These threads can run concurrently on multiple CPU cores, and they communicate and coordinate by accessing shared data in the shared memory. This approach is commonly used in multi-core processors and symmetric multiprocessing (SMP) systems.

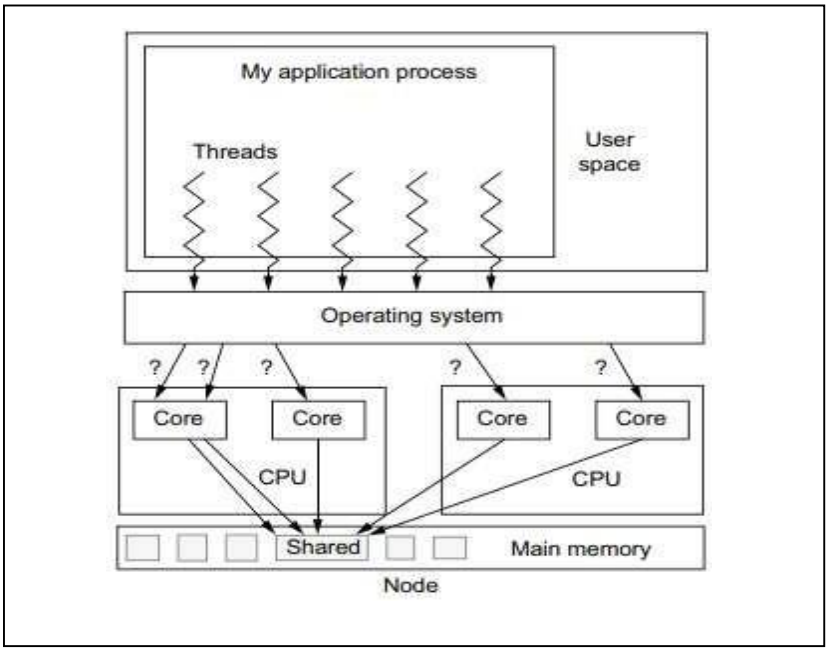


Fig 7:The application process in a thread-based approach to parallelization spawns threads. The threads are restricted to the node's domain. The question marks show that the OS decides where to place thethreads. Some memory is shared between threads.

- **Vectorization**: Multiple operations with one instruction

Vectorization is a parallel computing technique that enables processors to perform multiple operations with a single instruction. It takes advantage of SIMD (Single Instruction, Multiple Data) capabilities found in modern processors, including CPUs and GPUs. SIMD allows a single instruction to operate on multiple data elements simultaneously, which can significantly accelerate computations involving large datasets.

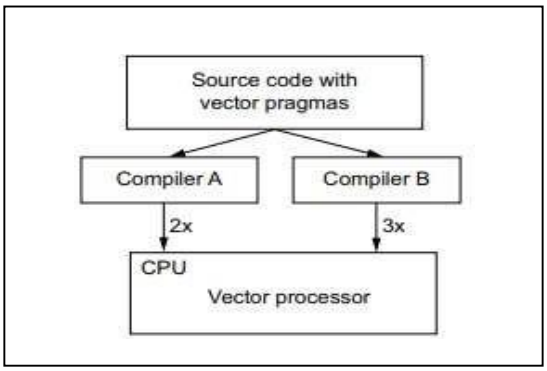


Fig 8 :Vector instructions in source code returning different performance levels from compilers

- **Stream processing**: Through specialized processors

Stream processing, often referred to as stream computing or data stream processing, is a computing paradigm where data is continuously processed as it is generated or ingested, rather than being stored in traditional databases or file systems. Stream processing is particularly useful for handling large volumes of real-time data from various sources, such as sensors, social media, financial transactions, and IoT devices. Specialized processors designed for stream processing accelerate the analysis and manipulation of data streams, ensuring timely and efficient processing

In the stream processing approach, data and compute kernel are offloaded to the GPU and its streaming multiprocessors. Processed data, or output, transfers back to the CPU for file IO or other work

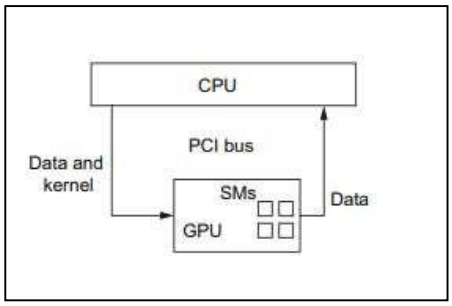


Fig 9 : Stream Processing Through Specialized Processors

UNIT-2

TOPIC 2

Fundamental laws

Fundamental laws in parallel computing, such as Amdahl's Law and Gustafson's Law, are essential for understanding the limitations and possibilities of parallel processing. These laws provide valuable insights into how the speedup of a parallel algorithm is affected by various factors.

What is Speedup?

Speedup in parallel computing refers to the performance improvement achieved by using multiple processors or computing resources to solve a problem compared to using a single processor. It is a measure of how much faster a parallel algorithm or system can complete a task compared to a serial (single-processor) implementation of the same task. Speedup is a crucial metric for evaluating the effectiveness of parallel computing systems.

The speedup (S) can be calculated using the following formula:

$s = T_{serial} / T_{parallel}$

Where:

- Tserial is the execution time of the task using a single processor (serial execution time).
  Tparallel is the execution time of the task using multiple processors (parallel execution time).

A speedup value greater than 1 indicates that the parallel implementation is faster than the serial implementation. Ideally, in a perfectly parallelizable task, doubling the number of processors would ideally halve the execution time, resulting in a speedup of 2. However, achieving perfect linear speedup is rare in real-world scenarios due to factors such as communication overhead, load balancing issues, and synchronization constraints between processors.

Amdahl's Law is a fundamental principle in parallel computing that expresses the potential speedup of a parallel algorithm as a function of the proportion of the algorithm that can be parallelized. It was formulated by Gene Amdahl in 1967 and is represented by the following formula:

$$SpeedUp(N) = \frac{1}{S + \frac{P}{N}}$$

Where:

- **Speedup** is the improvement in performance achieved by parallelizing a computation compared to executing it sequentially.
- **P** is the proportion of the algorithm that can be parallelized (a value between 0 and 1).
- S is the serial fraction
- N – no.of processors/nodes/cores

Amdahl's Law highlights the limitations of parallel computing. It states that the speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. In other words, if only a portion of a program can be parallelized (the rest being inherently sequential), then no matter how many processors are added, there will always be a limit to the speedup that can be achieved.

For example, if 90% of a program can be parallelized (P = 0.9) and the parallel portion runs on 5 processors, the maximum speedup that can be achieved according to Amdahl's Law is:

Speedup=1/(0.1+(0.9/5))=3.57

In this case, even though 90% of the program can be parallelized and runs on 5 processors, the maximum speedup achievable is approximately 3.57 times faster compared to the sequential execution due to the presence of the 10% sequential portion.

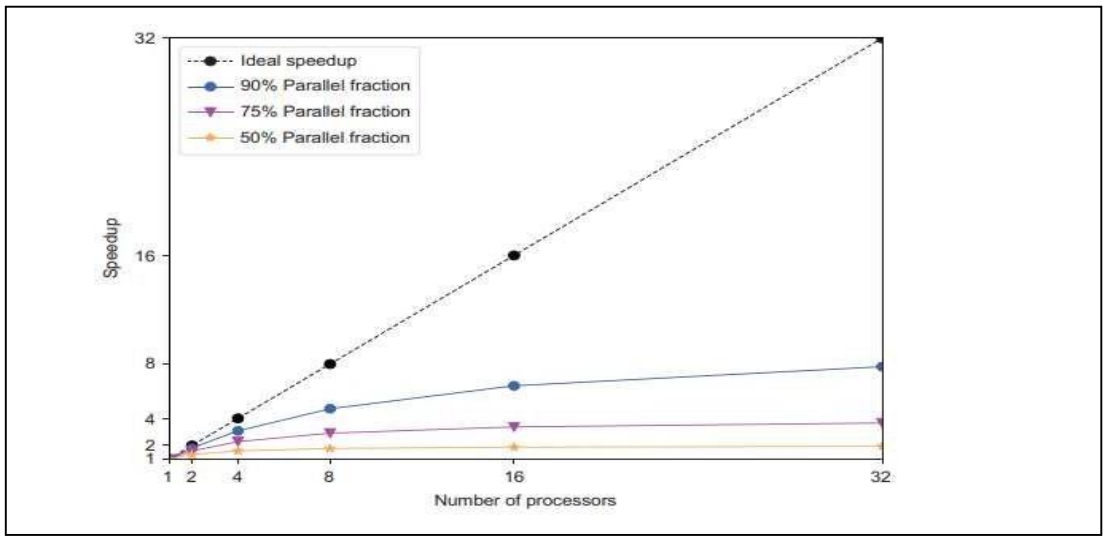


Fig : Speedup for a fixed-size problem according to Amdahl's Law is shown as a function of the number of processors. Lines show ideal speedup when 100% of an algorithm is parallelized, and for 90%, 75%, and 50%. Amdahl's Law states that speedup is limited by the fractions ofcode that remain serial.

Gustafson's Law(Gustafson Barsis Law), formulated by John L. Gustafson, provides a different perspective on parallel computing compared to Amdahl's Law. Unlike Amdahl's Law, which focuses on fixed problem sizes, Gustafson's Law takes into account varying problem

sizes. The basic idea behind Gustafson's Law is that as the size of the problem increases, the impact of the parallelizable portion of the program becomes more significant, leading to better scalability. In other words, with larger problem sizes, parallel systems can achieve higher levels of speedup.

The formula for Gustafson's Law is as follows:

SpeedUp(N) = N – S * (N – 1)where N is the number of processors, and S is the serial fraction

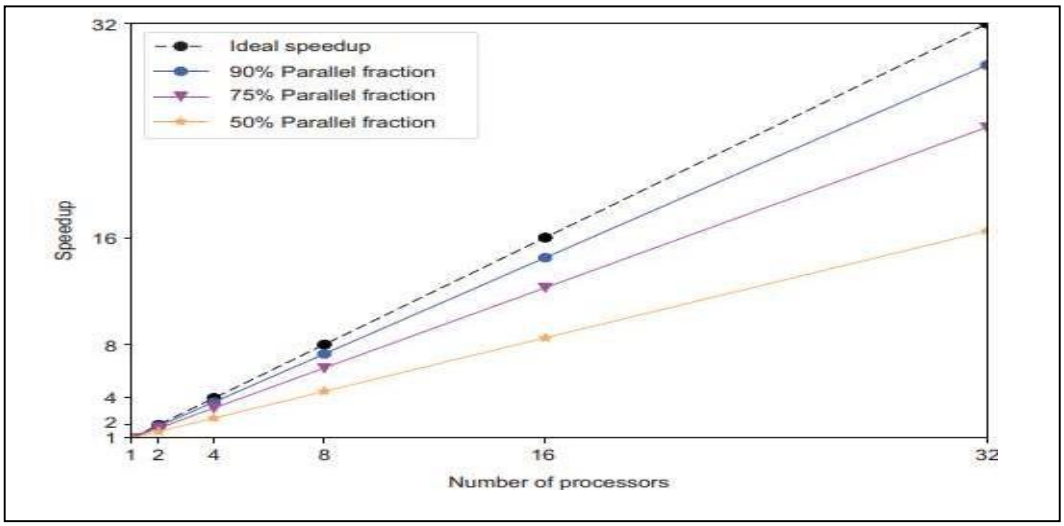


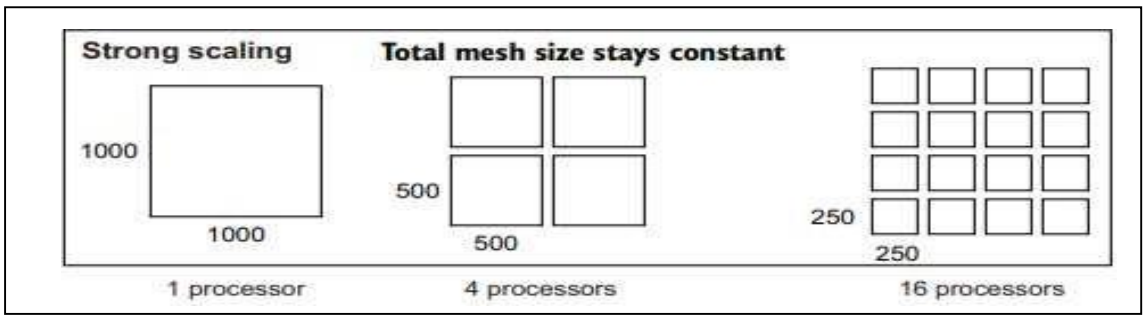
Fig :Speedup for when the size of a problem grows with the number of available processors accordingto Gustafson-Barsis's Law is shown as a function of the number of processors. Lines show ideal speedup when 100% of an algorithm is parallelized, and for 90%, 75%, and 50%

Strong scaling and weak scaling are two different metrics used to evaluate the performance of parallel computing systems, and they provide insights into how well a parallel algorithm or application can handle an increasing workload or an increasing number of processors. Here's a comparison of strong scaling and weak scaling:

Strong Scaling:

Definition: Strong scaling measures how the execution time of a fixed problem size decreases as the number of processors increases. In other words, it assesses how well a parallel system performs when the size of the problem remains constant, but the number of processors used to solve the problem increases.

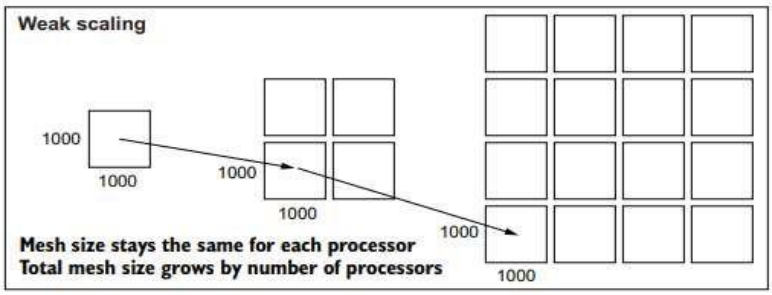
Objective: The goal of strong scaling is to reduce the execution time for a fixed problem size by utilizing more processors. It aims to speed up the solution of a specific problem.



2. Weak Scaling:

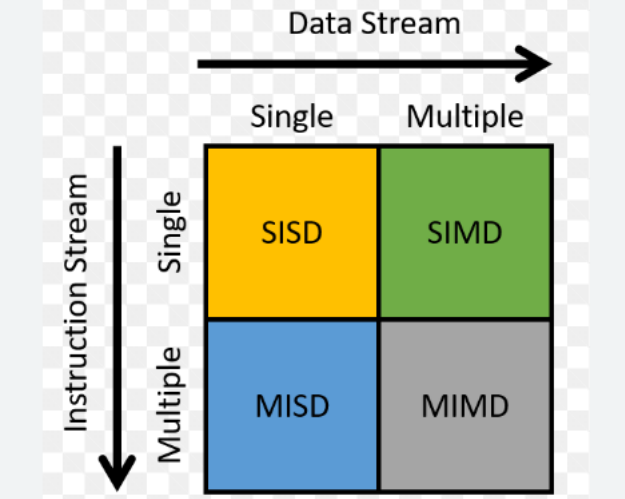
Definition: Weak scaling measures how the execution time changes as both the problem size and the number of processors increase proportionally. In other words, it assesses how well a parallel system can handle larger workloads by adding more processors as the problem size grows.

Objective: The goal of weak scaling is to maintain a constant workload per processor as the size of the problem and the number of processors increase. It aims to solve larger problems in approximately the same amount of time per processor.



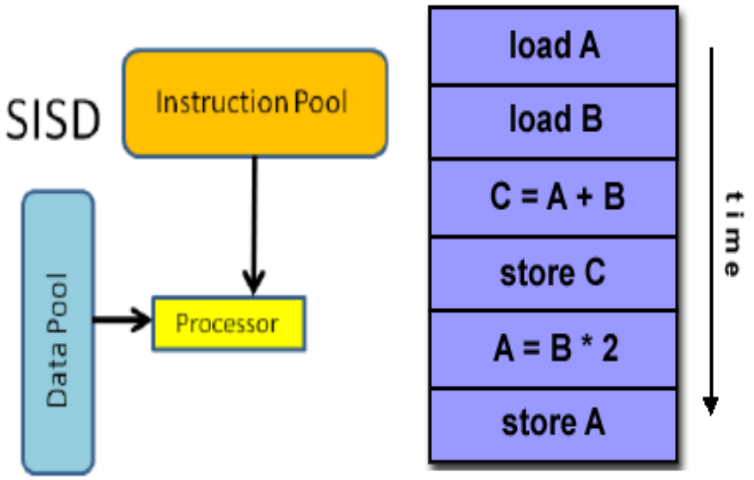
Parallel Approaches (Flynn's Classification)

Flynn's classification is essential in the field of parallel computing because it provides a framework for understanding and categorizing different types of computer architectures based on the number of instruction streams and data streams. This classification is named after Michael J. Flynn, who introduced it in 1966. Flynn's taxonomy is a useful tool for understanding different types of computer architectures and their strengths and weaknesses.

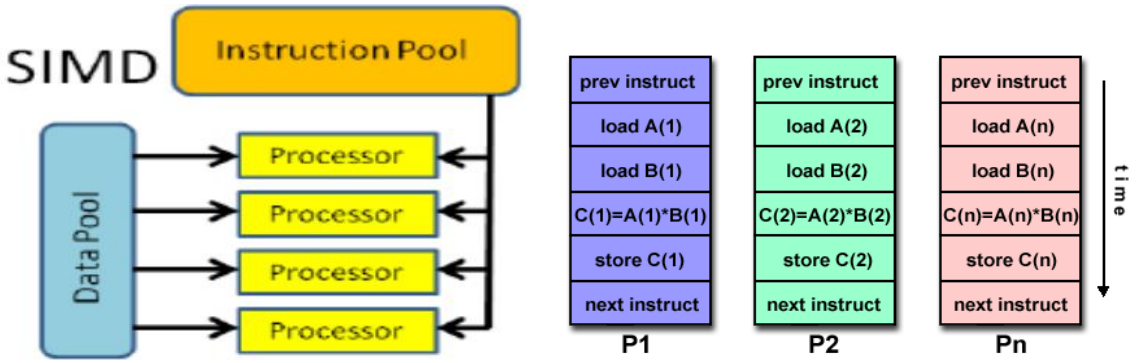


The taxonomy highlights the importance of parallelism in modern computing and shows how different types of parallelism can be exploited to improve performance.It helps in designing and analyzing parallel processing systems

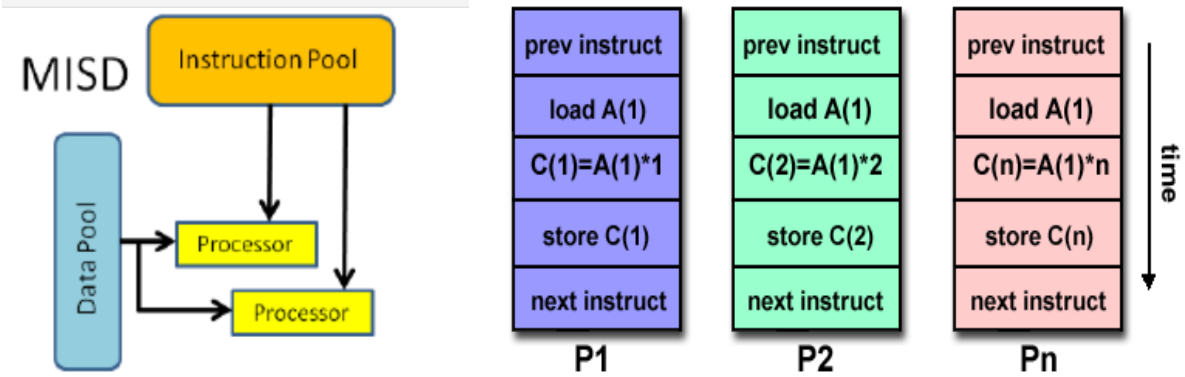
1. Single Instruction Single Data (**SISD**): In a SISD architecture, there is a single processor that executes a single instruction stream and operates on a single data stream. This is the simplest type of computer architecture and is used in most traditional computers.



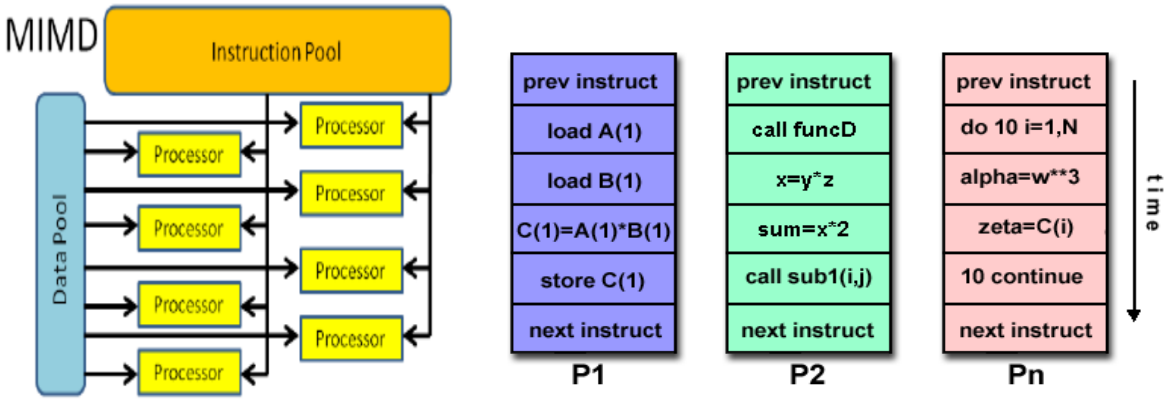
2. Single Instruction Multiple Data (**SIMD**): In a SIMD architecture, there is a single processor that executes the same instruction on multiple data streams in parallel. This type of architecture is used in applications such as image and signal processing.



3. Multiple Instruction Single Data (**MISD**): In a MISD architecture, multiple processors execute different instructions on the same data stream. This type of architecture is not commonly used in practice, as it is difficult to find applications that can be decomposed into independent instruction streams.



4. Multiple Instruction Multiple Data (**MIMD**): In a MIMD architecture, multiple processors execute different instructions on different data streams. This type of architecture is used in distributed computing, parallel processing, and other high-performance computing applications.



UNIT-1

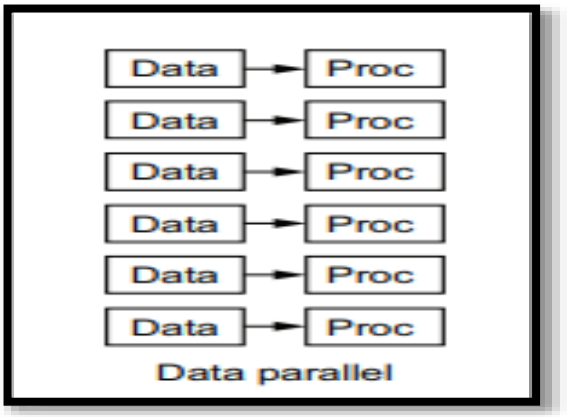
TOPIC 3

Parallel strategies

Parallel strategies" typically refer to techniques and methods for parallel processing, which is the simultaneous execution of multiple tasks or processes to improve the efficiency and performance of a computer system. Parallel strategies are commonly used in various computing domains, such as high-performance computing and distributed systems, to speed up computations and handle large volumes of data. Here are some common parallel strategies:

Data Parallel Approach

Data parallelism involves performing the same operation on multiple data elements simultaneously. This strategy is often used in applications where the same operation can be applied to different pieces of data independently.



Scenario: Imagine you're running a data analysis task on a large dataset of customer reviews for a product. Your goal is to perform sentiment analysis on each review to determine if it's positive, negative, or neutral. The sentiment analysis process is computationally intensive, and you want to speed it up using data parallelism.

Data Parallelism in Sentiment Analysis:

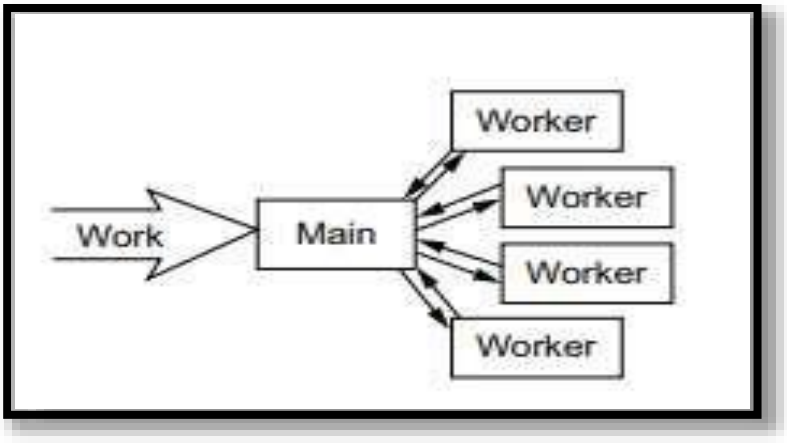
1. **Data Preparation**: You have a dataset of 1,000,000 customer reviews. To apply data parallelism, you divide this dataset into smaller, non-overlapping subsets. Let's say you split it into four subsets, each containing 250,000 reviews.
2. **Parallel Processing**: You have a sentiment analysis model that can analyze reviews. You set up four separate processing units (e.g., CPU cores or machines in a cluster), each responsible for analyzing one subset of reviews. Each processing unit loads its assigned subset of data.

3. **Analysis**: Each processing unit applies the sentiment analysis model to its subset of reviews independently and simultaneously. For instance:
    - Processing Unit 1 analyzes reviews 1 to 250,000.
    - Processing Unit 2 analyzes reviews 250,001 to 500,000.
    - Processing Unit 3 analyzes reviews 500,001 to 750,000.
    - Processing Unit 4 analyzes reviews 750,001 to 1,000,000.
4. **Aggregation**: As each processing unit finishes its analysis, it generates results, such as counts of positive, negative, and neutral reviews within its subset. These results are temporarily stored.
5. **Combining Results**: After all processing units have completed their work, you combine the results. You sum up the counts from each processing unit to get the overall sentiment analysis results for the entire dataset.

Task Parallelism(Main-Worker Approach)

Task parallelism involves executing multiple independent tasks or processes in parallel. Each task can perform different operations and may not necessarily operate on the same data. Task parallelism is common in applications where different tasks can be performed concurrently without dependencies between them.

In the main-worker approach, one processor schedules and distributes the tasks for all the workers, and each worker checks for the next work item as it returns the previous completed



task.

Example: Web Server Handling Requests

Consider a web server handling incoming HTTP requests. Each incoming request is an independent task that can be processed concurrently. The tasks include tasks like parsing the request, querying the database, and generating the response. In a task parallelism scenario:

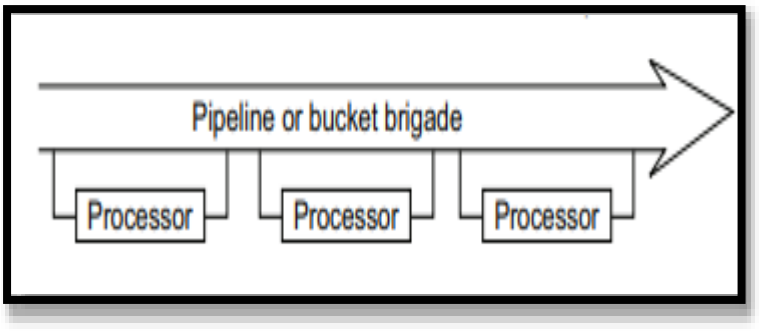
1. **Task 1: Parsing Request**
    - This task involves parsing the incoming HTTP request to extract information like the requested URL, parameters, and headers.
2. **Task 2: Database Query**
    - This task involves querying a database to fetch data related to the request, such as user information or product details.
3. **Task 3: Generating Response**
    - This task involves generating an HTML response based on the parsed request and data retrieved from the database.

In a task parallelism setup, these tasks can be executed concurrently by multiple threads or processes, allowing the server to handle multiple incoming requests simultaneously without waiting for one task to complete before starting the next.

Bucket-brigade Parallelism:

A **bucket brigade** is a method of manually transporting items or materials from one location to another by forming a line of people, each of whom carries an item and passes it to the next person. This technique is similar to how buckets of water might be passed along a line of people to put out a fire, which is where the term "bucket brigade" originated.

In parallel computing, the concept of bucket-brigade parallelism involves breaking down a task into smaller subtasks, where each subtask is processed independently and passed to the next processing unit for further computation. This technique allows for efficient parallel processing of tasks and is often used in scenarios where tasks can be divided into smaller, manageable parts.



Example: Manufacturing Assembly Line

Let's say we have a manufacturing assembly line for producing smartphones. The assembly line consists of three stages: A, B, and C. Each stage represents a specific task in the smartphone assembly process.

1. **Stage A - Component Assembly**:
    - Worker A assembles the basic components of the smartphone, such as the circuit board, battery, and display. Once Worker A finishes assembling a smartphone, it passes it to Stage B.
2. **Stage B - Software Installation**:
    - Worker B installs the operating system and necessary software onto the smartphone assembled by Worker A. After software installation, the smartphone is passed to Stage C.
3. **Stage C - Quality Control and Packaging**:
    - Worker C checks the smartphone for quality control, ensuring that all components are working correctly and the software is functioning as intended. If the smartphone passes quality control, it is packaged and prepared for shipment.

In this example, each stage (A, B, and C) represents a processing step, similar to the stages in a bucket-brigade parallelism scenario.

Parallel speedup versus comparative speedups.

Parallel speedup and comparative speedup are two different metrics used to evaluate the performance improvement achieved by parallel processing.

Parallel speedup measures how much faster a parallel algorithm runs compared to its sequential (single-processor) counterpart. It quantifies the performance improvement gained by using multiple processing units in parallel. Parallel speedup is calculated using the following formula:

Parallel Speedup=Sequential Execution Time/Parallel Execution

In this formula:

- **Sequential Execution Time** is the time taken by the algorithm to execute sequentially on a single processor.
- **Parallel Execution Time** is the time taken by the parallel algorithm to execute on multiple processors.

Comparative speedup: Comparative speedup is between architectures. This is usually a performance comparison between two parallel implementations or other comparison between reasonably constrained sets of hardware. For example, it may be between a parallel MPI implementation on all the cores of the node of a computer versus the GPU(s) on a node

UNIT 1

TOPIC 4

Performance limits and profiling:

Applications Potential Performance Limits & determine your hardware capabilities

In parallel processing, understanding performance limits and profiling the application are crucial steps to optimize the execution of parallel programs.

Performance limits refer to the maximum achievable performance of a computing system or application under specific conditions. These limits are determined by various factors and constraints and play a crucial role in understanding the capabilities and limitations of a system. Understanding these performance limits is essential for designing efficient algorithms, optimizing software, and choosing appropriate hardware configurations. It also guides researchers and engineers in developing new technologies to overcome existing limitations and improve overall computing performance.

Profiling tools are used to gather detailed information about the behavior of a parallel program. By understanding performance limits, utilizing profiling tools, and optimizing the code based on the profiling results, developers can enhance the efficiency of parallel applications, leading to improved speedup and overall performance.

Application's potential performance limits

- Flops (floating-point operations)
- Ops (operations) that include all types of computer instructions
- Memory bandwidth: Rate at which the data is transferred
- Memory latency: Time required for the first byte or word of data to be transferred
- Instruction queue (instruction cache)
- Networks
- Disk
- Machine Balance: Number of flops executed /memory bandwidth
- Arithmetic Intensity: Number of flops executed per memory operation

All of these limitations can be divided into two major categories : Speeds are how fastoperations can be done. It includes all types of computer operations. But to be able to do the operations, you must get the data there. This is where feeds come in. Feeds include the memory bandwidth through the cache hierarchy, as well as network and disk bandwidth.

For many applications, the memory bandwidth limit can be difficult especially dealing with non-contiguous bandwidth.It is also known as strided memory access or non-contiguous memory access, refers to the manner in which data elements are accessed in memory. In contrast to contiguous memory access, where elements are stored in consecutive memory locations, non-contiguous memory access involves accessing elements that are not stored sequentially in memory.

Non-Contiguous Memory Access:

Now, consider a situation where the array elements are scattered in memory with a stride of 2. This is a non-contiguous memory access pattern:

```
Memory: | 1 | x | 2 | x | 3 | x | 4 | x | 5 | ...
Array:  | 10| 20| 30| 40| 50| 60| 70| 80| 90| ...
```

In this case, accessing every second element (stride = 2) would mean accessing memory locations 1, 2, 3, 4, 5, etc., but the elements are not stored sequentially.

When your program needs to access such non-contiguous elements, it may lead to inefficiencies due to increased cache misses and a higher likelihood of accessing data from main memory rather than the faster cache memory.

Determine your Hardware capabilities:

To determine the Performance of hardware the following metrics are used:

- The rate at which floating-point operations can be executed (FLOPs/s)
- The rate at which data can be moved between various levels of memory (GB/s)
- The rate at which energy is used by your application (Watts)

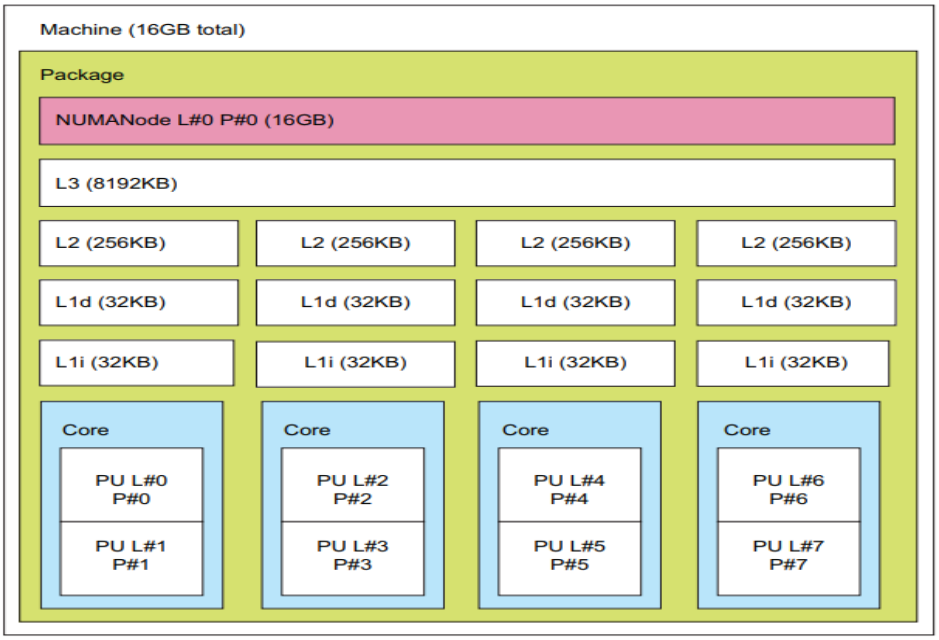
In determining hardware performance and calculating the metrics, we use a mixture of theoretical and empirical measurements.

Theoretical measurements provide an upper limit to what a system can achieve. For instance, in parallel computing, theoretical analysis can reveal the maximum speedup or efficiency that a parallel algorithm can achieve in an ideal scenario**.**

Real-world Validation is done by **Empirical measurements**, they provide concrete evidence of how a system performs under real-world conditions, accounting for various factors like I/O operations, network latency, and concurrency issues.

One of the best tools for understanding the hardware you run is the lstopo program(graphical

view) and lscpu for text view. lstopo is bundled with the hwloc package that comes with nearly every MPI distribution. This command outputs a graphical view of the hardware on your system. Figure below shows the output for a Mac laptop in **graphical view.**



Text view

The information from the lspci command and the /proc/cpuinfo file helps to determine the number of processors, the processor model, the cache sizes, and the clock frequency for the system

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
Stepping:              3
CPU MHz:               871.241
CPU max MHz:           3600.0000
CPU min MHz:           800.0000
BogoMIPS:              6384.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-3
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl
vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx
rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida
arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```

Calculating theoretical maximum flops

Theoretical FLOPS=Number of Cores×Clock Speed×FLOPs per Cycle per Core

Where:

- **Number of Cores:** This represents the total number of processor cores in the computing system.
- **Clock Speed:** This indicates the clock speed of each core in the system, typically measured in Hertz (Hz) or Gigahertz (GHz). It represents the number of cycles the processor can execute per second.
- **FLOPs per Cycle per Core:** This signifies the number of floating-point operations a core can perform in a single clock cycle. Modern processors often perform multiple FLOPs per cycle due to features like SIMD (Single Instruction, Multiple Data) operations.

For example, let's consider a system with 4 cores, each operating at 3.0 GHz, and capable of executing 4 FLOPs per cycle per core (assuming SIMD operations are utilized):

Theoretical FLOPS=4 cores×3.0 GHz×4 FLOPs per cycle per Core

Theoretical FLOPS=48 GFLOPS

The memory hierarchy and theoretical memory bandwidth

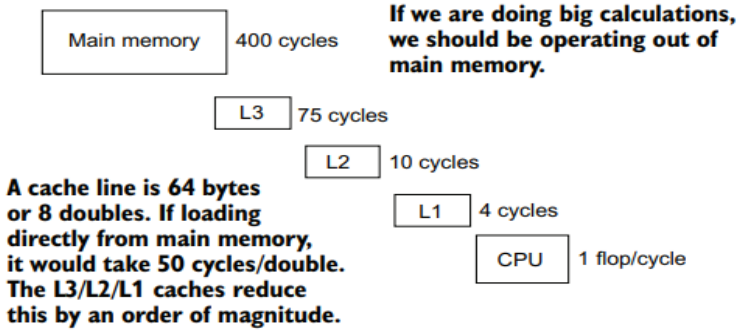


Figure 3.5 Memory hierarchy and access times. Memory is loaded into cache lines and stored at each level of the cache system for reuse.

We can calculate the theoretical memory bandwidth of the main memory using the memory chips specifications.

The general formula is B T = MTR × Mc × Tw × Ns = Data Transfer Rate × Memory Channels × Bytes Per Access × Sockets

Processors are installed in a socket on the motherboard. The motherboard is the main system board of the computer, and the socket is the location where the processor is inserted. Most motherboards are single-socket, where only one processor can be installed. Dual-socket motherboards are more common in high-performance computing systems. Two processors can

be installed in a dual-socket motherboard, giving us more processing cores and more memory bandwidth.

Empirical measurement of bandwidth and flop

The empirical bandwidth is the measurement of the fastest rate that memory can be loaded from main memory into the processor. If a single byte of memory is requested, it takes 1 cycle to retrieve it from a CPU register. If it is not in the CPU register, it comes from the L1 cache. If it is not in the L1 cache, the L1 cache loads it from L2 and so on to main memory. If it goes all the way to main memory, for a single byte of memory, it can take around 400 clock cycles. This time required for the first byte of data from each level of memory is called the memory latency.

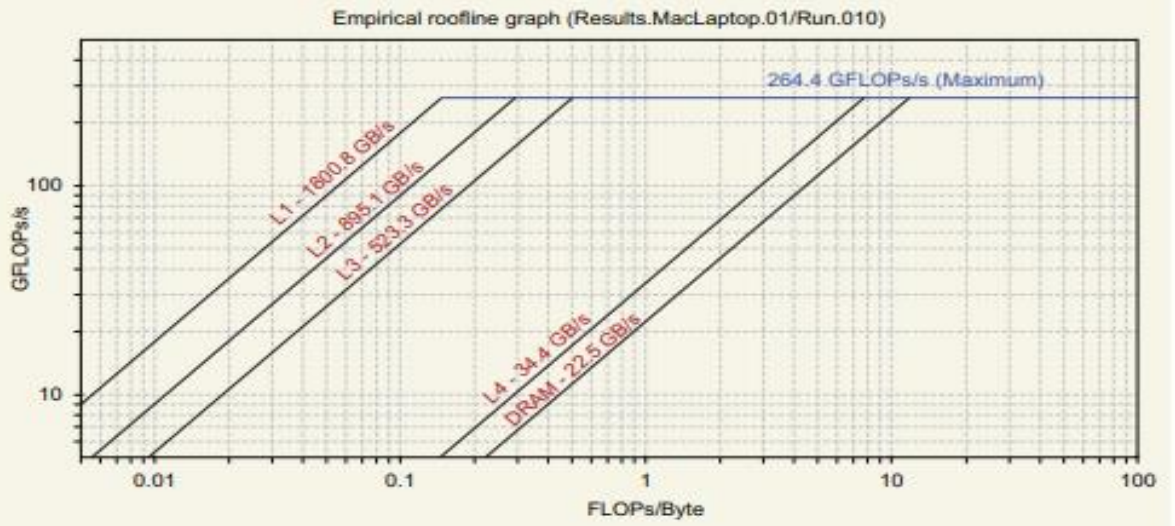
Two different methods are used for measuring the bandwidth: **the STREAM Benchmark** and the **roofline model** measured by the Empirical Roofline Toolkit.

Key Differences:

- **Focus:** STREAM primarily focuses on memory bandwidth, providing quantitative measurements. In contrast, the Roofline Model provides a graphical representation of performance bottlenecks, considering both computational capabilities and memory bandwidth.
- **Representation:** STREAM results in a numerical measurement (memory bandwidth in bytes per second), while the Roofline Model is a graphical representation that helps visualize performance limitations.
- **Insights:** STREAM provides detailed insights into memory subsystem performance, whereas the Roofline Model offers a high-level overview of an application's performance efficiency concerning hardware constraints.

Stream Benchmark

	Bytes	Arithmetic Operations
• Copy: a(i) = b(i)	16	0
• Scale: a(i) = q*b(i)	16	1
• Sum: a(i) = b(i) + c(i)	24	1
• Triad: a(i) = b(i) + q*c(i)	24	2



Roofline for 2017 Mac Laptop shows the maximum FLOPs as a horizontal line, and the maximum bandwidth from various cache and memory levels as sloped lines.

Calculating the machine balance between flops and bandwidth

The machine balance is the flops divided by the memory bandwidth.

We can calculate both a theoretical machine balance (MB_T) and an empirical machine balance (MB_E) like so:

$MB_T = F_T / B_T$

$MB_E = F_E / B_E$

UNIT 1

TOPIC 5

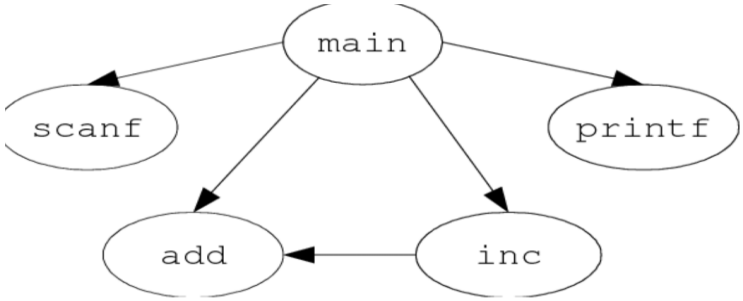
Characterizing your application: Profiling

Now that you have some sense of what performance you can get with the hardware, you need to determine what are the performance characteristics of your application. Additionally, you should develop an understanding of how different subroutines and functions depend on each other.

Profiling tools:

Using call graphs for hot-spot and dependency analysis

In the context of parallel programming, call graphs are diagrams that represent the calling relationships between different functions or methods in a parallel program. They illustrate how functions or tasks invoke each other and provide a visual representation of the program's control flow. Analyzing call graphs in parallel programming can provide valuable insights into the program's structure, dependencies, and potential performance optimizations. By analyzing these call graphs, developers can identify hot-spots—functions or tasks that consume a significant amount of computational time. Optimizing these hot-spots is essential for improving overall parallel program performance.



Empirical measurement of processor clock frequency and energy consumption

Empirical Measurement of Processor Clock Frequency:

1. **Profiling Tools:** Profiling tools like Intel VTune Profiler or AMD CodeXL can provide insights into various performance metrics, including processor clock frequency. These tools often offer visualizations and detailed reports for better analysis.
2. **Benchmarking Suites:** Benchmarking tools like SPEC CPU benchmarks or HPC Challenge benchmarks often include components that measure processor clock frequencies. Running these benchmarks can provide detailed information about the processor's performance characteristics.

Empirical Measurement of Energy Consumption:

1. **Power Measurement Tools:** Use power measurement tools and hardware devices to measure the power consumption of your system. Power meters and sensors can be

attached to the system to measure real-time power usage. Tools like Intel Power Gadget or Linux's `powerstat` can help measure power usage.

2. **Energy Profilers:** Some profiling tools, like Intel VTune Profiler, also offer energy profiling capabilities. They can provide insights into energy consumption patterns at different parts of your code. These tools often correlate energy consumption with specific functions or code regions.

Tracking memory during run time

Tracking memory usage during runtime in parallel computing is crucial for optimizing performance, detecting memory leaks, and ensuring efficient memory management. Several techniques and tools can help you monitor memory usage in parallel applications. Here are some approaches to tracking memory during runtime in parallel computing environments:

Profiling Tools:

1. **Valgrind Massif:** Valgrind is a powerful instrumentation framework. Massif, a Valgrind tool, can profile heap memory usage over time, showing memory consumption patterns. It's particularly useful for detecting memory leaks and understanding how memory usage evolves during program execution.
2. **Intel VTune Profiler:** VTune Profiler provides memory analysis capabilities, including memory usage tracking. It can analyze memory consumption at various levels, from individual functions to entire applications, in both serial and parallel contexts.
3. **OpenMP/MPI Memory Profilers:** Many parallel programming frameworks like OpenMP and MPI provide their memory profiling tools. For example, OpenMP has tools like Score-P, and MPI has memory profiling features integrated into MPI implementations.

UNIT-1

TOPIC-6

Parallel algorithms and patterns

A parallel algorithm is a step-by-step computational procedure or set of rules designed to be executed on parallel computing architectures. These algorithms are specifically crafted to take advantage of parallel processing capabilities, where multiple processors or cores can work together to solve a problem.

Parallel patterns are like reusable blueprints that help programmers apply proven methods to solve specific types of problems efficiently. These patterns guide the decomposition of tasks and data, providing a framework for creating effective parallel algorithms.

Example : Parallel Algorithm for Finding the Maximum Element:

Suppose you have a large array of numbers, and you want to find the maximum element using a parallel algorithm based on the "Divide and Conquer" pattern. In this example, the "Divide and Conquer" pattern is applied to find the maximum element in an array. The array is divided into smaller subarrays, and the maximum values of these subarrays are found in parallel. Finally, the maximum among these partial maximums is selected as the maximum element of the entire array.

Algorithm analysis for parallel computing applications

The goal of algorithm analysis is to compare different algorithms that are used to solve the same problem. One of the more traditional ways to evaluate algorithms is by looking at their algorithmic complexity.

DEFINITION: Algorithmic complexity is a measure of the number of operations that it would take to complete an algorithm. Algorithmic complexity is a property of the algorithm and is a measure of the amount of work or operations in the procedure.

Complexity is usually expressed in asymptotic notation. Using asymptotic notation, you can analyze and compare algorithms efficiently and make informed decisions when choosing the most suitable algorithm for a particular problem, taking into account both time and space complexity

The three main types of asymptotic notation are:

1. **Big O Notation (O-notation):**
    - Big O notation describes the upper bound or worst-case time complexity of an algorithm. It represents an approximation of how an algorithm's running time increases as the input size grows. It provides an upper limit on the number of basic operations an algorithm performs.
2. **Theta Notation (Θ-notation):**

- Theta notation provides a tight bound, expressing both the upper and lower bounds of an algorithm's time complexity. It characterizes the average-case behavior of an algorithm.

3.**Omega Notation (Ω-notation):**

- Omega notation describes the lower bound or best-case time complexity of an algorithm. It provides a way to express how quickly the algorithm can solve a problem in the most favorable circumstances.

Performance models versus algorithmic complexity

Performance models are broader and more practical in nature. They encompass various aspects of system performance, including algorithmic efficiency, but also consider factors related to specific hardware, software, and real-world scenarios.

Algorithmic complexity, often expressed using asymptotic notations like Big O, Theta, and Omega, focuses on analyzing the efficiency of algorithms in terms of their time and space requirements as a function of the input size. It provides a theoretical framework for characterizing how an algorithm's performance scales as the input size grows towards infinity.

Example: finding the sum of all elements in an array.

Algorithmic Complexity (Big O Notation):

- **Time Complexity:** $O(n)$ - Linear time complexity, where n is the size of the input array. The algorithm processes each element once.
- **Space Complexity:** $O(1)$ - Constant space complexity, as it uses only a few variables regardless of the input size.

Performance Model Considerations:

- **Hardware Differences:** Different computers might execute the same algorithm at different speeds due to variations in processor capabilities.
- **Compiler Optimizations:** Compilers can optimize the code differently, affecting the execution time.
- **Parallelization:** Divide the array into chunks and calculate the partial sums concurrently using parallel processing techniques, especially for large arrays.
- **Memory Optimization:** For very large arrays, consider memory-efficient data structures or algorithms to reduce memory usage.

Parallel algorithms

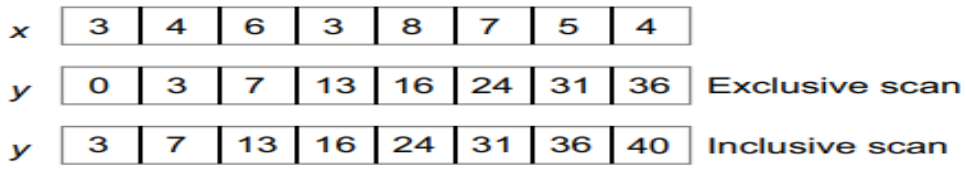
Parallel algorithms are designed to efficiently solve computational problems by utilizing multiple processing units (such as CPU cores, GPUs, or distributed computing nodes) simultaneously. They are crucial in high-performance computing (HPC) and parallel processing environments where large datasets and complex computations need to be handled efficiently. Parallel algorithms aim to break down tasks into smaller subtasks that can be processed independently and concurrently, leading to significant speedup in overall computation time. Here are some common types of parallel algorithms:

- **Parallel Merge Sort:** Divide the sorting task into smaller parts, sort them independently, and then merge the sorted parts in parallel.
- **Parallel QuickSort:** A parallel version of the quicksort algorithm that partitions the data and sorts partitions concurrently.

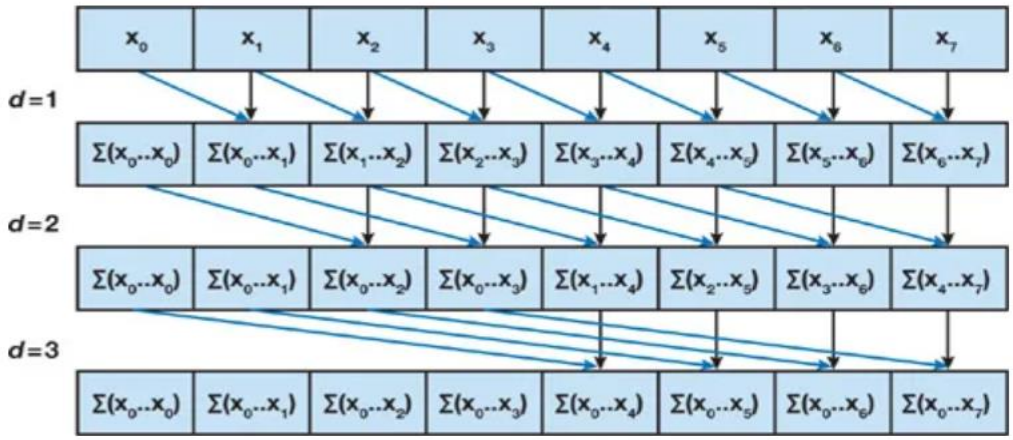
Prefix sum

The prefix sum, also known as the scan operation, is a fundamental parallel pattern in computer science and parallel computing. Given an input array of elements, the prefix sum operation computes a new array where each element is the sum of all elements in the input array up to and including the corresponding element's position. There are two common types of prefix sum operations: exclusive and inclusive.

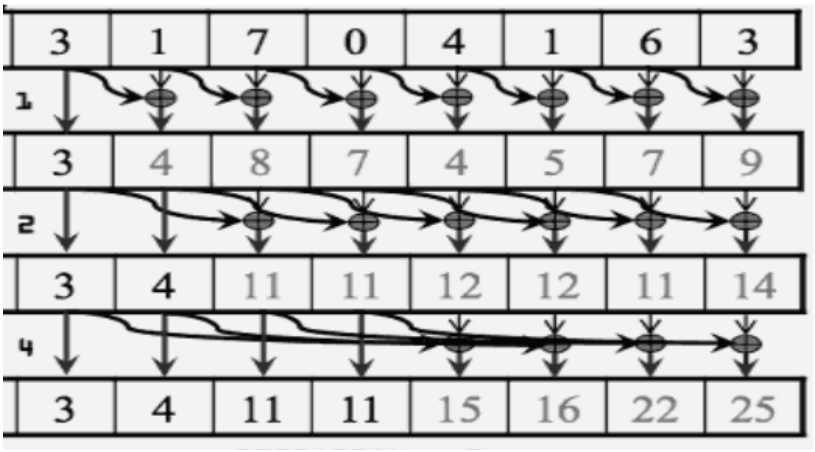
- **Exclusive Prefix Sum:** The result at each position does not include the element at that position.
    - Input: [a, b, c, d]
    - Output (Exclusive): [0, a, a+b, a+b+c]
- **Inclusive Prefix Sum:** The result at each position includes the element at that position.
    - Input: [a, b, c, d]
    - Output (Inclusive): [a, a+b, a+b+c, a+b+c+d]



Step-efficient parallel scan operation(Inclusive Prefix sum)

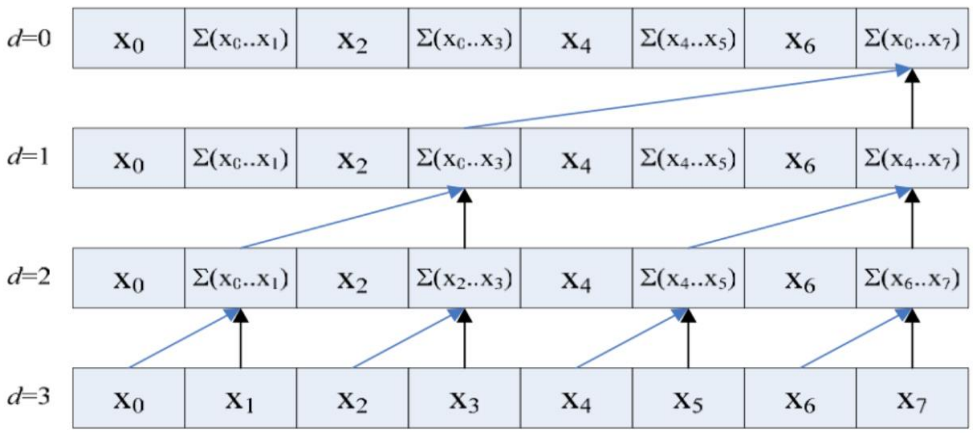


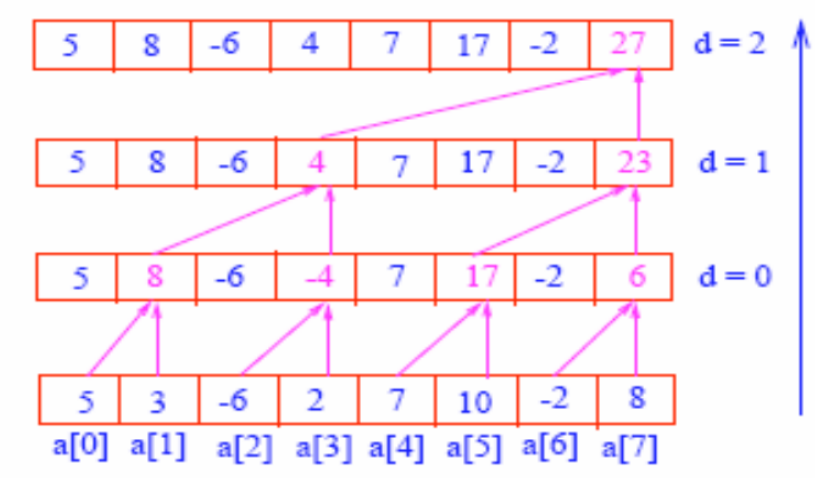
Parallel Prefix Scan



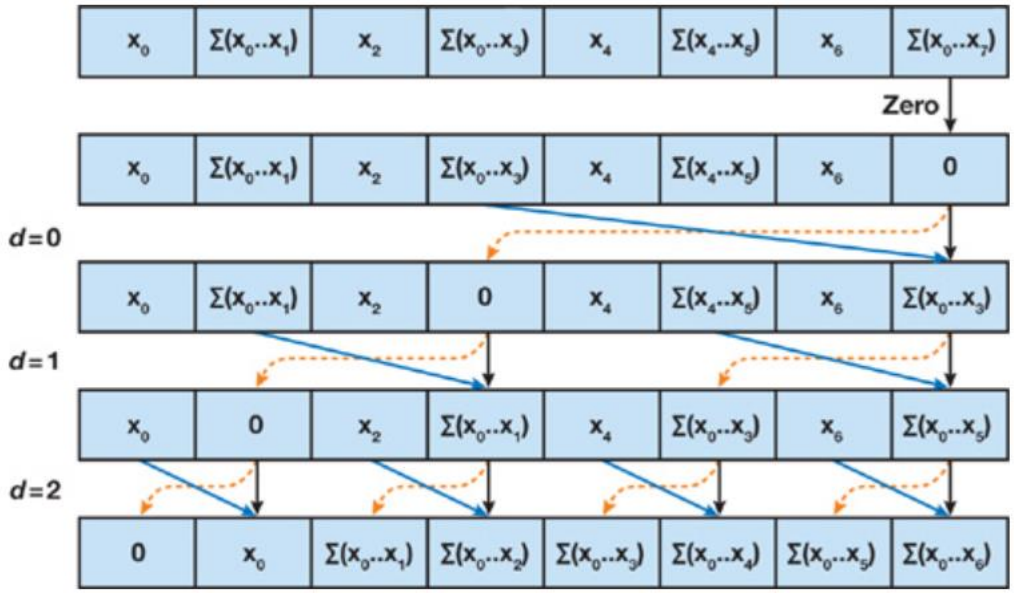
Work-efficient parallel scan operation(Exclusive Prefix sum)

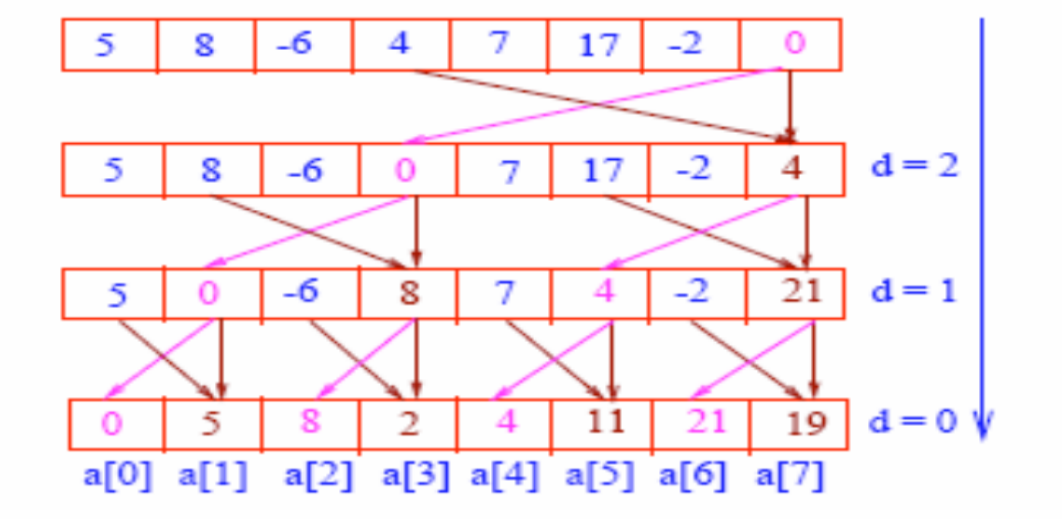
The work-efficient parallel scan operation uses two sweeps through the arrays. The first sweep is called an upsweep, though it is more of a right sweep.





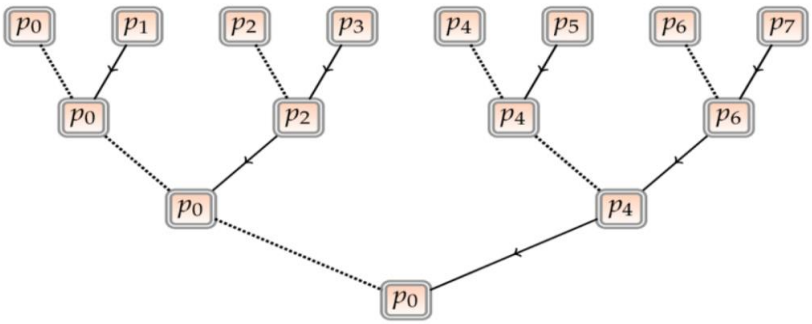
The second phase, known as the downsweep phase, is more of a left sweep. The output of upsweep is provided as input to downsweep.It starts by setting the last value to zero and then does another tree-based sweep to get the final result.





Parallel global sum

The "parallel global sum" refers to the problem of computing the sum of elements across multiple processors or nodes in a parallel or distributed computing environment.



Though the process is simple it has some problems like Changing the order of additions changes the answer in finite-precision arithmetic. This is problematic because a parallel calculation changes the order of the additions. There is even a worse case for additions of finite precision values when adding two values that are almost identical, but of different signs. This subtraction of one value from another when these are nearly the same causes a catastrophic cancellation.

Catastrophic cancellation occurs when the operands are subject to rounding errors.

For example, if there are two measures L1=253.5cm long and the otherL2 =252.5cm long

Approximations could come out to be

L1 = 254cm and L2 = 252cm L1-L2 = 2cm

Actual difference is L1-L2 = 1 cm

There are several solutions for addressing the global sum . The list of possible techniques presented here includes

- Long-double data type
- Pairwise summation
- Kahan summation
- Knuth summation—uses same method of pairwise
- Quad-precision summation

Long-double data type

The easiest solution is to use the long-double data type on a x86 architecture. On this architecture, a long-double is implemented as an 80-bit floating-point number in hardware giving an extra 16-bits of precision. Unfortunately, this is not a portable technique

Listing 5.16 Long-double data type sum on x86 architectures

GlobalSums/do_ldsum.c

```
1 double do_ldsum(double *var, long ncells)
2 {
3     long double ldsum = 0.0;
4     for (long i = 0; i < ncells; i++){
5         ldsum += (long double)var[i];
6     }
7     double dsum = ldsum;
8     return(dsum);
9 }
```

var is an array of doubles, while the accumulator is a long double.

The return type of the function can also be long double and the value of ldsum returned.

Returns a double

Pairwise summation

Pairwise summation, also known as pairwise addition, is a method used to sum a sequence of numbers in a way that reduces the effects of numerical errors, particularly in floating-point arithmetic. This technique is commonly employed in scientific computing and numerical analysis to improve the accuracy of summation operations.

How Pairwise Summation Works:

1. **Pairing the Numbers:**

- Given a sequence of numbers, they are paired up. If there are an odd number of elements, one number is left unpaired.

2. **Pairwise Addition:**
    - Within each pair, the two numbers are added together to create intermediate sums.
3. **Summing the Intermediate Sums:**
    - The intermediate sums obtained from pairwise addition are then summed together using the same pairwise summation method.

- **Kahan summation**

Kahan summation, also known as compensated summation or Kahan summation algorithm, is a method used to reduce the numerical error that accumulates during the summation of a large number of floating-point values.

How Kahan Summation Works:

In standard floating-point summation, when adding a small number to a large number, the small number can be "lost" in the least significant bits of the large number, leading to a loss of precision. Kahan summation addresses this issue by using a compensation term to keep track of the lost precision.

1. **Initialization:**
    - Initialize the sum and the compensation term to zero.
2. **Iterative Addition:**

- For each number to be added:
- Add the number to the current sum.
- Calculate the difference between the updated sum and the original sum (this difference is the lost precision).
- Add this difference to the compensation term.

3. **Final Result:**
    - The final result is the sum adjusted by the compensation term.

Quad-precision summation

Quad-precision summation refers to performing arithmetic operations with numbers represented in quadruple-precision floating-point format. In the IEEE 754 floating-point standard, quadruple-precision is a 128-bit data type, providing higher precision compared to single-precision (32-bit) and double-precision (64-bit) floating-point number.

Listing 5.20 Quad precision global sum

GlobalSums/do_qdsum.c

```
1 double do_qdsum(double *var, long ncells)
2 {
3     __float128 qdsum = 0.0;
4     for (long i = 0; i < ncells; i++){
5        qdsum += (__float128)var[i];
6     }
7     double dsum =qdsum;
8     return(dsum);
9 }
```

Quad precision data type

Casts the input value from array to quad precision

UNIT-2

TOPIC-1

Parallel Programming on CPU-I

Vectorization

Vectorization is a technique used in computer science to perform operations on entire arrays or sequences of data elements simultaneously, instead of processing each element individually. It's commonly used in numerical and scientific computing, as well as in various data analysis and machine learning tasks. In Parallel computing, processors have special vector units that can load and operate on more than one data element at a time.

SIMD overview

Vectorization is an example of single instruction, multiple data (SIMD) processing because it executes a single operation (e.g., addition, division) over a large dataset. A scalar operation, in the context of mathematics and computer science, refers to an operation that is performed on a single scalar value, as opposed to a vector, matrix, or any other data structure. Scalars are single numerical values and can be integers, floating-point numbers, or other numerical types.

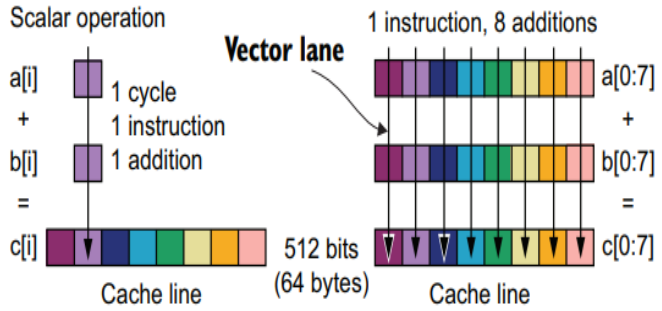


Figure 6.1 A scalar operation does a single double-precision addition in one cycle. It takes eight cycles to process a 64-byte cache line. In comparison, a vector operation on a 512-bit vector unit can process all eight double-precision values in one cycle.

Vectorization Terminology:

Vector (SIMD) lane : A pathway through a vector operation on vector registers for a single data element much like a lane on a multi-lane free way.

Vector width : The width of the vector unit, usually expressed in bits

Vector length : The number of data elements that can be processed by the vector in one operation.

Vector (SIMD) instruction sets: The set of instructions that extend the regular scalar processor instructions to utilize the vector processor.

Vectorization is produced through both a software and a hardware component.

The requirements are

Generate Instruction
The vector instructions must be generated by the compiler or manually specified through intrinsics or assembler codings

Match instructions to the vector unit of the processor.
If there is a mismatch between the instructions and the hardware, newer hardware can usually process

Hardware trends for vectorization

It is helpful to know the historical dates of hardware and instruction set release for selecting which vector instruction set to use.

Release	Functionality
MMX (trademark with no official meaning)	Targeted towards the graphics market, but GPUs soon took over this function. Vector units shifted their focus to computation rather than graphics. AMD released its version under the name 3DNow! with single-precision support.
SSE (Streaming SIMD Extensions)	First Intel vector unit to offer floating-point operations with single-precision support
SSE2	Double-precision support added
AVX (Advanced Vector Extensions)	Twice the vector length. AMD added a fused multiply-add FMA vector instruction in its competing hardware, effectively doubling the performance for some loops.
AVX2	Intel added a fused multiply-add (FMA) to its vector processor.
AVX512	First offered on the Knights Landing processor; it came to the main-line multi-core processor hardware lineup in 2017. From the years 2018 and on, Intel and AMD (Advanced Micro Devices, Inc.) have created multiple variants of AVX512 as incremental improvements to vector hardware architectures.

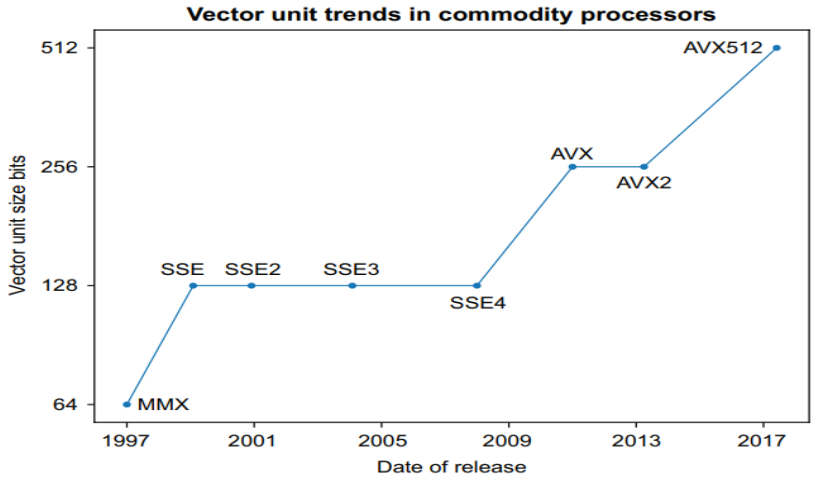


Figure 6.2 The appearance of vector unit hardware for commodity processors began around 1997 and has slowly grown over the last twenty years, both in vector width (size) and in types of operations supported.

UNIT-2

TOPIC-2

Parallel Programming on CPU-II

Vectorization methods

There are several ways to achieve Vectorization in your program.

- **Optimized libraries :**

Optimized libraries play a crucial role in achieving vectorization and improving the performance of software applications, especially in the context of numerical and scientific computing. These libraries provide pre-implemented and highly optimized functions for common mathematical and linear algebra operations. Some of the most commonly used libraries include

BLAS (Basic Linear Algebra System)—A base component of high-performance linear algebra software

LAPACK—A linear algebra package

SCALAPACK—A scalable linear algebra package

FFT (Fast Fourier transform)—Various implementation packages available

Sparse Solvers—various implementations of sparse solvers available

- **Auto-Vectorization**

Auto-vectorization is a compiler optimization technique that transforms scalar code into vectorized code, taking advantage of SIMD (Single Instruction, Multiple Data) instructions available in modern processors. Most modern compilers provide flags or options to enable or enhance auto-vectorization.

- **Hints to the compiler**

Hints to the compiler are annotations or directives provided by the programmer to guide the compiler's optimization decisions. These hints can inform the compiler about specific optimizations that should be applied to certain parts of the code. Different programming languages and compilers have different ways to provide hints to guide vectorization.

1. ***Pragmas can guide the compiler's vectorization process, helping it identify loops that can be safely and efficiently vectorized***.

```
#pragma clang loop vectorize(enable) // Enable loop vectorization
for (int i = 0; i < N; ++i)
{
   // Loop body
}
```

2. In the context of computer programming and compiler optimizations dependencies play a crucial role in determining the order in which instructions or operations can be executed.

Various data Depencies are

A flow dependency, also known as a true dependency (Read After Write-RAW), occurs when an instruction depends on the result of a previous instruction. As a result, the second instruction cannot be executed until the first one completes.

For example, if you have the code **b = a + 1** and then **c = b + 2**, there is a flow dependency from the second instruction to the first because it relies on the result of the first instruction.

An anti-flow dependency, also known as an anti-dependency(Write After Read-WAR), occurs when the order of execution of instructions is crucial to avoid incorrect results.

1. **b = a * 2** (Instruction 1)
2. **a = a + 1** (Instruction 2)

In this example, **Instruction 2** modifies the value of the variable **a**, which is used in **Instruction 1**. If **Instruction 2** were to execute before **Instruction 1**, the value of **a** used in **Instruction 1** would be incorrect, leading to erroneous results. Therefore, the correct order of execution is **Instruction 1** followed by **Instruction 2**.

An output dependency, also known as a write-after-write dependency (WAW dependency), occurs when two instructions write to the same memory location or register.

Consider the following sequence of instructions:

1. **a = 5** (Instruction 1)
2. **a = a + 3** (Instruction 2)

In this example, both **Instruction 1** and **Instruction 2** write to the same variable **a**. If **Instruction 1** and **Instruction 2** are executed out of order, the final value of **a** depends on the order in which the instructions are executed.

- If **Instruction 1** is executed after **Instruction 2**, **a** will be 5.
- If **Instruction 1** is executed before **Instruction 2**, **a** will be 8.

3. Vectorization of Loops:

A "peel loop" is a term used in the context of loop optimization in computer programming and compilers. Loop peeling refers to the process of extracting one or more iterations from the beginning or end of a loop and handling them separately from the main loop.

```
// Original loop
for (int i = 0; i < N; i++) {
   // Loop body
```

```
}
// Peeling the first iteration
// Handle the first iteration separately(This can be beneficial, for example, if the first iteration requires special processing, and the remaining iterations start from index 1.)
// Main loop with iterations from 1 to N-1
for (int i = 1; i < N; i++) {
    // Loop body
}
```

A "remainder loop" refers to a loop that iterates over the remaining elements of a collection or array after a specific condition is met. It is a common programming construct used to process the remaining elements of a data structure once a certain criterion is satisfied within the loop.

For example, if the vectorized loop trip count is 20 and the vector length is 16, it means every time the kernel loop gets executed once, the remainder 4 iterations have to be executed in the remainder-loop.

The peel loop is added to deal with the unaligned data at the start of the loop, and the remainder loop takes care of any extra data at the end of the loop

- **Vector intrinsic:**

  Vector intrinsics are low-level programming constructs used to write explicit vectorized code by directly utilizing the capabilities of SIMD (Single Instruction, Multiple Data) instructions available on modern processors. Vector intrinsics are typically written in assembly or as inline assembly within a higher-level programming language and are often specific to a particular CPU architecture.

  Intrisics provide data types for vectors (i.e. `__m128 a;` would declare the variable `a` to be a vector of 4 floats). They also provide functions that operate directly on vectors (i.e. `_mm_add_ps(a,b)` would add together the two vectors `a` and `b`).

- **Assembler instructions:**

  Using assembly language for vectorization involves writing low-level code that directly employs SIMD (Single Instruction, Multiple Data) instructions to perform operations on multiple data elements in parallel.

  The example below demonstrates vectorization using x86 assembly with SSE (Streaming

SIMD Extensions) instructions:

```
array1 dd 1, 2, 3, 4    ; First array of integers
array2 dd 5, 6, 7, 8    ; Second array of integers
result dd 0, 0, 0, 0    ; Array to store the result
_start:
movaps xmm0, [array1]   ;    Load 128-bit (4x32-bit) values from array1 to xmm0
movaps xmm1, [array2]   ;    Load 128-bit (4x32-bit) values from array2 to xmm1
addps xmm0, xmm1        ;    Perform vectorized addition
movaps [result], xmm0   ;    Store the result back to the result array
Exit
```

Programming style for better Vectorization:

Adopting the following programming styles leads to better performance out of the box and less work needed for optimization efforts.

General suggestions:

- Use the restrict attribute on pointers in function arguments and declarations (C and C++).
- Use pragmas or directives where needed to inform the compiler.
- Be careful with optimizing for the compiler with #pragma unroll and other techniques; you might limit the possible options for the compiler transformations.
- Put exceptions and error checks with print statements in a separate loop.

Concerning data structures:

- Try to use a data structure with a long length for the innermost loop
- Use the smallest data type needed (short rather than int).
- Use contiguous memory accesses.
- Use Structure of Arrays (SOA) rather than Array of Structures (AOS).

Array of Structures (AoS)(structure variable is an array)

```
struct person {
char gender;
int age;
} s[5];
```

Structure of Arrays (SoA)(Structure member is an array)

```
struct person {
char name[60];
```

```
char gender;
int age;
} ;
```

- Use memory-aligned data structures where possible.

Related to loop structures:

- Use simple loops without special exit conditions.
- Make loop bounds a local variable by copying global values and then using them.
- Use the loop index for array addresses when possible.
- Expose the loop bound size so it is known to the compiler. If the loop is only three iterations long, the compiler might unroll the loop rather than generate a four-wide vector instruction.
- Avoid array syntax in performance-critical loops (FORTRAN).

In the loop body:

- Define local variables within a loop so that it is clear that these are not carried to subsequent iterations (C and C++).
- Variables and arrays within a loop should be write-only or read-only (only on the left side of the equal sign or on the right side, except for reductions).
- Don’t reuse local variables for a different purpose in the loop—create a new variable. The memory space you waste is far less important than the confusion this creates for the compiler.

Concerning compiler settings and flags:

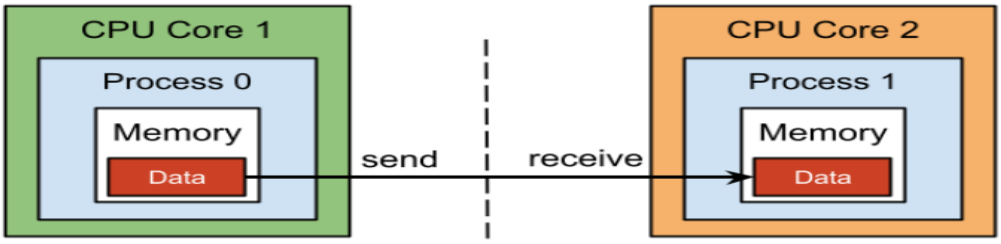
- Use the latest version of a compiler and prefer compilers that do better vectorization.
- Use a strict aliasing compiler flag.
- Generate code for the most powerful vector instruction set you can get away with

UNIT-2

TOPIC 3

The basics for an MPI program

MPI (Message Passing Interface) is a standard communication protocol used in parallel computing environments to enable processes running on different processors or nodes to communicate and coordinate their actions. It is commonly used in high-performance computing (HPC) and cluster computing applications where multiple computing nodes work together to solve a complex problem.



MPI allows programs to be written in a distributed-memory programming model, where each process has its own local memory space and communicates with other processes using message passing. Processes can send and receive messages, making it possible for them to exchange data and synchronize their execution.

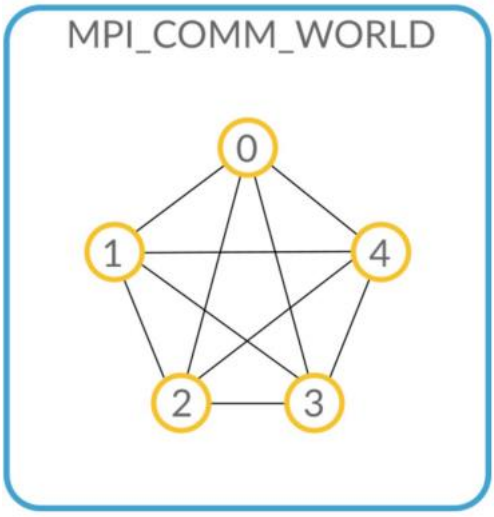
In the context of MPI (Message Passing Interface) programming, the terms "communication world" and "rank" are fundamental concepts used to manage communication and coordination among parallel processes in a parallel computing environment.

1. Communication World:

- A communication world, also known as a communicator, is a group of MPI processes that can communicate with each other.
- MPI_COMM_WORLD is the default communicator that includes all processes created when the MPI application starts.
- Communicators allow processes to be organized into groups, enabling more controlled and specific communication patterns.

2. Rank:

- Rank refers to the unique identifier assigned to each process within a communicator.
- In MPI_COMM_WORLD, ranks range from 0 to (number of processes - 1).
- Ranks are used to distinguish one process from another within the same communicator.
- Processes can communicate with each other using their ranks as identifiers.



The diagram shows a program which runs with five processes. In this example, the size of MPI_COMM_WORLD is 5. The rank of each process is the number inside each circle. The rank of a process always ranges from 0 to 4.

MPI Functions

1. **MPI_Comm_rank(MPI_COMM_WORLD, &rank) :** The rank of a process within a communicator can be obtained using this

Parameters

- `MPI_COMM_WORLD`: This is a predefined communicator in MPI that includes all processes spawned by the MPI program. It is a communicator for the world of all processes.
- `&rank`: This is the address of the variable where the rank of the calling process will be stored. The function retrieves the rank and stores it in the memory location pointed to by the `&rank` variable.

2. **MPI_Comm_size(MPI_COMM_WORLD, &size**) is an MPI function call which retrieves the total number of processes in the communicator `MPI_COMM_WORLD`.

Parameters

- `&size`: This is the address of the variable where the total number of processes in the communicator will be stored. The function retrieves the size and stores it in the memory location pointed to by the `&size` variable.

3. **MPI_Init(&argc, &argv)** is an MPI (Message Passing Interface) function call used to initialize the MPI environment. It is typically the first MPI function called in an MPI program.

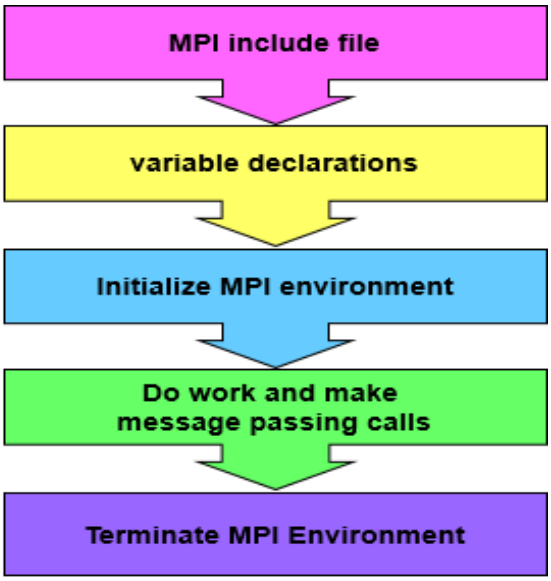
Parameters

- `&argc`: This passes a pointer to the `argc` variable to the MPI library. The `argc` variable holds the number of command-line arguments passed to the program.

- `&argv`: This passes a pointer to the `argv` variable to the MPI library. The `argv` variable is an array of strings containing the command-line arguments. When MPI initializes, it sets up communication channels between the processes, prepares the MPI environment for parallel computation

4. **MPI_Finalize()** is an MPI (Message Passing Interface) function used to finalize the MPI environment. It is typically the last MPI function called in an MPI program, and it performs several important tasks to ensure the proper termination of the MPI application. `MPI_Finalize()` ensures that all communication operations initiated by the program are completed before the program terminates.

MPI Program Structure



MPI Program

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
 {
   MPI_Init(&argc, &argv);
   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get the rank of the current process
   MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the total number of processes in the communicator
   printf("Hello from process %d of %d in MPI_COMM_WORLD\n", rank, size);
   MPI_Finalize();
```

```
    return 0;
}
```

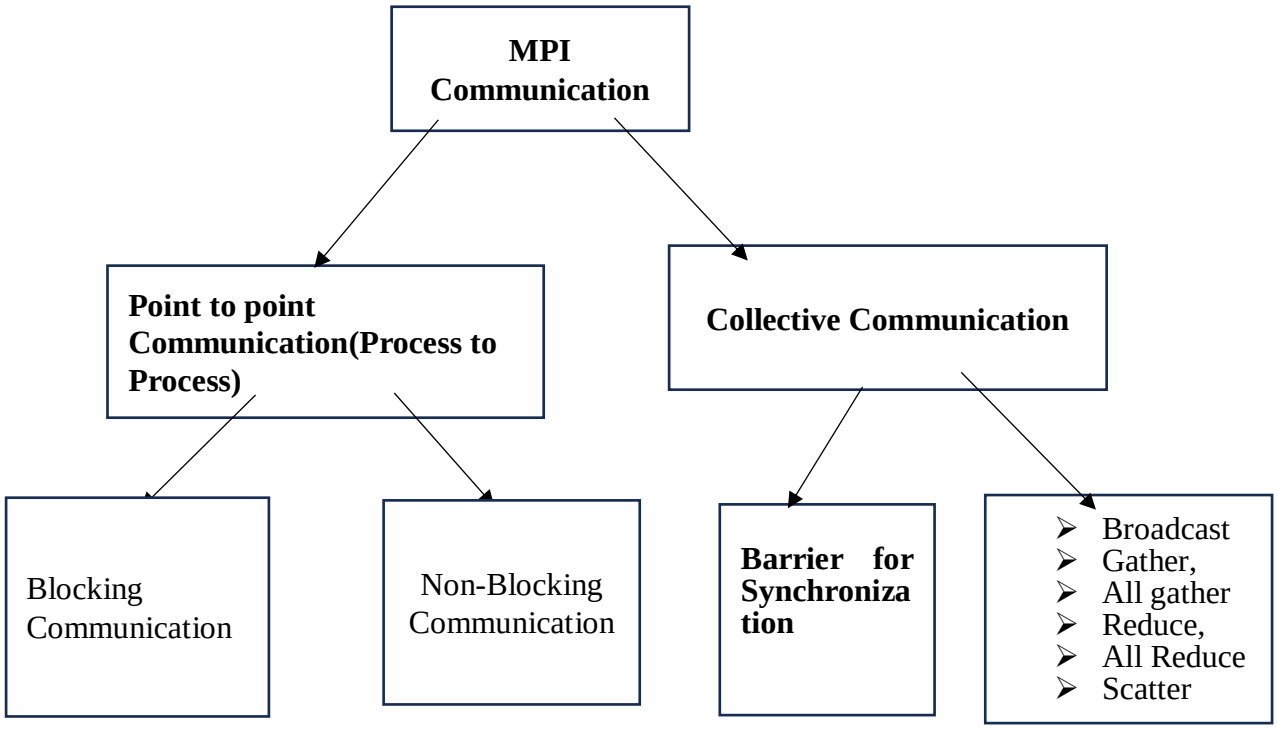
MPI Datatypes: MPI provides its own reference data types corresponding to the various elementarydata types in C.

MPI Datatype	C Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)
MPI_PACKED	(none)

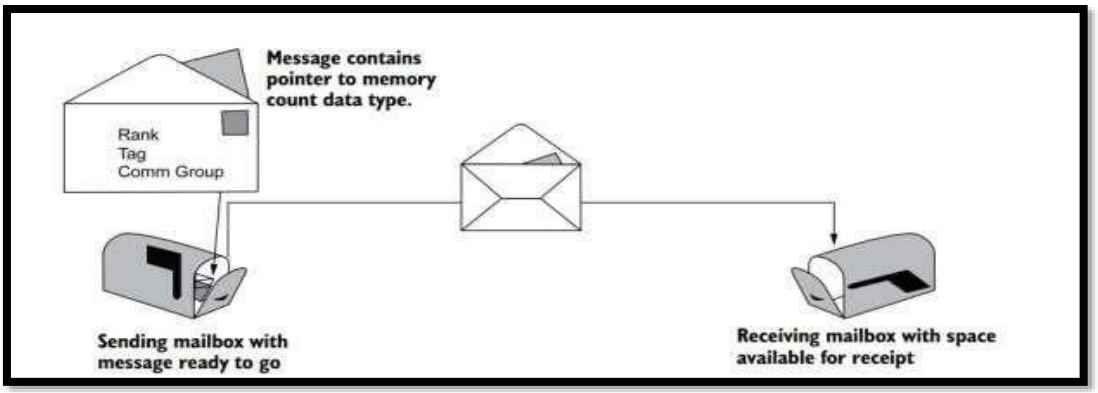
UNIT-2

TOPIC 4

The send and receive commands for process-to-process communication



The core of the message-passing approach is to send a message from point-to-point or, perhaps more precisely, process-to-process. The whole point of parallel processing is to coordinate work.



The Figure shows the three components for process to process communication

- **Mail Box:** There must be a mailbox at either end of the system. The size of the mailbox is important. The sending side knows the size of the message, but the receiving side does not. To make sure there is a place for the message to be stored, it is usually better to post the receive first. This avoids delaying the message by the receiving process having to allocate a

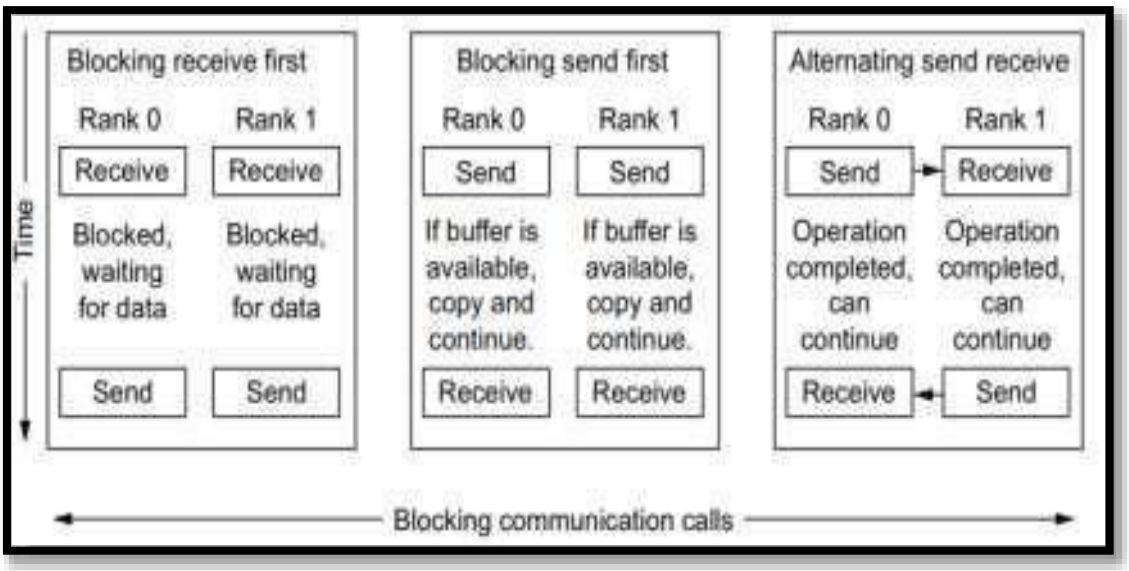
temporary space to store the message until a receive is posted and it can copy it to the right location. For an analogy, if the receive (mailbox) is not posted (not there), the postman has to hangout until someone puts one up. Posting the receive first avoids the possibility of insufficient memory space on the receiving end to allocate a temporary buffer to store the message.

- **Message :**The message itself is always composed of a triplet at both ends: a pointer to a memory buffer, a count, and a type. The type sent and type received can be different types and counts. The rationale for using types and counts is that it allows the conversion of types between the processes at the source and at the destination. This permits a message to be converted to a different form at the receiving end. In a heterogeneous environment, this might mean converting lower-endian to big-endian, a lowlevel difference in the byte order of data stored on different hardware vendors. Also, the receive size can be greater than the amount sent. This permits the receiver to query how much data is sent so it can properly handle the message. But the receiving size cannot be smaller than the sending size because it would cause a write past the end of the buffer.

- **Envelope:** The envelope also is composed of a triplet. It defines who the message is from, who it is sent to, and a message identifier to keep from getting multiple messages confused. The triplet consists of the rank, tag, and communication group. The rank is for the specified communication group. The tag helps the programmer and MPI distinguish which message goes to which receive. In MPI, the tag is a convenience. It can be set to MPI_ANY_TAG if an explicit tag number is not desired.

We have two types of process to process communication: Blocking and non blocking

Blocking communication in MPI refers to the type of communication where a process halts its execution until a specific communication operation is completed. This means that the sending and receiving processes are synchronized also refered as Synchronuous Communication. The sender blocks until the receiver is ready to receive the message, and vice versa. The two most common blocking communication operations in MPI are `MPI_Send` and `MPI_Recv`.



`MPI_Send` is a blocking communication function in MPI (Message Passing Interface) used for sending messages from one process to another. It sends a message from the sender process to the specified destination process. Here is the syntax for `MPI_Send`

MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

- **buf**: A pointer to the send buffer (the data you want to send).
- **count**: The number of elements in the send buffer.
- **datatype**: The data type of the elements in the send buffer.
- **dest**: The rank of the destination process.
- **tag**: A message tag, which can be used by the receiver to distinguish different kinds of messages.
- **comm**: The communicator (usually **MPI_COMM_WORLD** for communication among all processes).

`MPI_Recv` is a blocking communication function in MPI (Message Passing Interface) used for receiving messages from other processes. It receives a message from a specified source process. Here is the syntax for `MPI_Recv`

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);

- **buf**: A pointer to the receive buffer (where the received data will be stored).
- **count**: The number of elements in the receive buffer.
- **datatype**: The data type of the elements in the receive buffer.
- **source**: The rank of the source process from which you want to receive the message. Use **MPI_ANY_SOURCE** if you want to receive a message from any source.

- **tag**: A message tag. If you used tags in **MPI_Send**, you can use the same tag here to filter messages. MPI_ANY_TAG matches any tag comm
- **comm**: The communicator (usually **MPI_COMM_WORLD** for communication among all processes).
- **status**: A pointer to an **MPI_Status** structure that will hold information about the received message, such as the source, tag, and error codes.

Program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
   int rank, size;
   int data_send = 42;
   int data_recv;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (size < 2) {
      printf("This program requires at least 2 processes.\n");
      MPI_Finalize();
      return 1;
   }
   // Blocking Send from process 0 to process 1
   if (rank == 0) {
      MPI_Send(&data_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      printf("Process %d sent data: %d\n", rank, data_send);
   }
   // Blocking Receive at process 1
   else if (rank == 1) {
      MPI_Recv(&data_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Process %d received data: %d\n", rank, data_recv);
   }
   MPI_Finalize();
   return 0;
```

}

Problems with blocking communication:

- A deadlock occurs when a set of processes are blocked because each is waiting for the other to release a resource. For example, if two processes are waiting for each other to send a message before they can receive, they will be deadlocked.
- Processes that are blocked waiting for communication can waste computational resources, such as CPU time and memory, as they are not performing useful work during that time.so we go for non blocking communication

Non blocking communication

Non-blocking communication in MPI allows processes to initiate communication operations and continue their execution without waiting for the communication to complete. This is often referred to as asynchronous or non-blocking calls. Asynchronous means that the call initiates the operation but does not wait for the completion of the work.

MPI provides non-blocking communication functions like `MPI_Isend( I means Immediate)`, `MPI_Irecv`, `MPI_Test`, `MPI_Wait`, and others to facilitate non-blocking communication. **Completion of a non-blocking send operation means that the sender is now free to update the send buffer "message".**

Completion of a non-blocking receive operation means that the receive buffer "message" contains the received data.

- `MPI_Isend` is a non-blocking communication function used for sending messages from one process to another. Unlike `MPI_Send`, which is a blocking operation, `MPI_Isend` returns immediately after initiating the send operation, allowing the sender process to continue its execution without waiting for the message to be delivered.

MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);

- **buf**: A pointer to the send buffer (the data you want to send).
- **count**: The number of elements in the send buffer.
- **datatype**: The data type of the elements in the send buffer.
- **dest**: The rank of the destination process.
- **tag**: A message tag, which can be used by the receiver to distinguish different kinds of messages.

- **comm**: The communicator (usually **MPI_COMM_WORLD** for communication among all processes).
- **request**: A pointer to an **MPI_Request** variable that will be used to identify the send request. You can later use this request to check the status of the communication or wait for its completion.

➢ `MPI_Irecv` is a non-blocking communication function used for receiving messages from other processes. Unlike `MPI_Recv`, which is a blocking operation, `MPI_Irecv` returns immediately after initiating the receive operation, allowing the receiving process to continue its execution without waiting for a message to arrive.

MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);

- **buf**: A pointer to the receive buffer (where the received data will be stored).
- **count**: The number of elements in the receive buffer.
- **datatype**: The data type of the elements in the receive buffer.
- **source**: The rank of the source process from which you want to receive the message. Use **MPI_ANY_SOURCE** if you want to receive a message from any source.
- **tag**: A message tag. If you used tags in **MPI_Send**, you can use the same tag here to filter messages.
- **comm**: The communicator (usually **MPI_COMM_WORLD** for communication among all processes).
- **request**: A pointer to an **MPI_Request** variable that will be used to identify the receive request. You can later use this request to check the status of the communication or wait for its completion.

➢ `MPI_Test` is a non-blocking communication function used to check the completion status of a communication request initiated by non-blocking send (`MPI_Isend`) or receive (`MPI_Irecv`) operations. It allows you to query whether a non-blocking operation has been completed without waiting for its completion. Here is the syntax for `MPI_Test`:

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);

- **request**: A pointer to an **MPI_Request** variable that identifies the communication request.
- **flag**: A pointer to an integer variable that will be set to true (non-zero) if the communication operation associated with the request has completed, and false (zero) otherwise.
- **status**: A pointer to an **MPI_Status** structure that will hold information about the completed communication, such as the source, tag, and error codes.

MPI_Test returns **MPI_SUCCESS** if the operation associated with the request has completed and **flag** is set to true. Otherwise, it returns **MPI_ERR_PENDING** if the operation is still pending, meaning it has not yet completed.

- `MPI_Wait` is a blocking function used to wait for the completion of a specific communication request, such as non-blocking send (`MPI_Isend`) or receive (`MPI_Irecv`) operations. It suspends the execution of the calling process until the specified communication operation is completed. Here is the syntax for `MPI_Wait`:

int MPI_Wait(MPI_Request *request, MPI_Status *status);

- **request**: A pointer to an **MPI_Request** variable that identifies the communication request.
- **status**: A pointer to an **MPI_Status** structure that will hold information about the completed communication, such as the source, tag, and error codes. You can pass **MPI_STATUS_IGNORE** if you don't need this information.

MPI_Wait blocks until the communication associated with the specified request is complete. Once the operation has completed, you can use the information in the **status** object if needed.

Program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
   int rank, size;
   int data_send = 42;
   int data_recv;
   MPI_Request request;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (size < 2) {
      printf("This program requires at least 2 processes.\n");
      MPI_Finalize();
      return 1;
   }
   // Non-blocking Send from process 0 to process 1
   if (rank == 0) {
      MPI_Isend(&data_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
```

```
        printf("Process %d initiated non-blocking send with data: %d\n", rank, data_send);
    }
    // Non-blocking Receive at process 1
    else if (rank == 1) {
        MPI_Irecv(&data_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        printf("Process %d initiated non-blocking receive.\n", rank);
    }
    // Wait for the non-blocking communication to complete
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    if (rank == 1) {
        printf("Process %d received data: %d\n", rank, data_recv);
    }
    MPI_Finalize();
    return 0;
}
```

Advantages of Non Blocking Communication:

1. **Overlapping of Computation and Communication:** Non-blocking operations allow computation and communication to occur concurrently. Processes can initiate communication operations and then continue with other computations without waiting for the communication to complete. This overlap of computation and communication can lead to improved overall performance and better utilization of resources.
2. **Reduced Synchronization Overheads:** Non-blocking operations reduce the need for synchronization points in the code. With blocking communication, processes often need to synchronize at communication points, leading to potential idle time for some processes. Non-blocking communication reduces these synchronization overheads and can lead to more balanced workloads among processes.
3. **Minimized Potential for Deadlocks:** Non-blocking communication reduces the likelihood of encountering deadlocks, which can occur in scenarios where processes are waiting for each other to release resources. Non-blocking operations allow processes to progress independently, reducing the chances of deadlocks.
4. **Better Load Balancing:** Non-blocking operations enable dynamic load balancing techniques. Processes can continue with computation tasks even when waiting for communication, allowing load balancing algorithms to adjust the workload dynamically based on the actual computational needs of the processes.

5. **Optimized Network Utilization:** Overlapping computation and communication can lead to more efficient use of network resources. Processes can perform useful work while waiting for messages, reducing idle time and maximizing the utilization of the communication network.

Other variants of send/receive might be useful in special situations.

The modes are indicated by a one- or two-letter prefix, similar to that seen in the immediate variant, as listed here:

B (buffered)

S (synchronous)

R (ready)

IB (immediate buffered)

IS (immediate synchronous)

IR (immediate ready)

UNIT-2

TOPIC 5

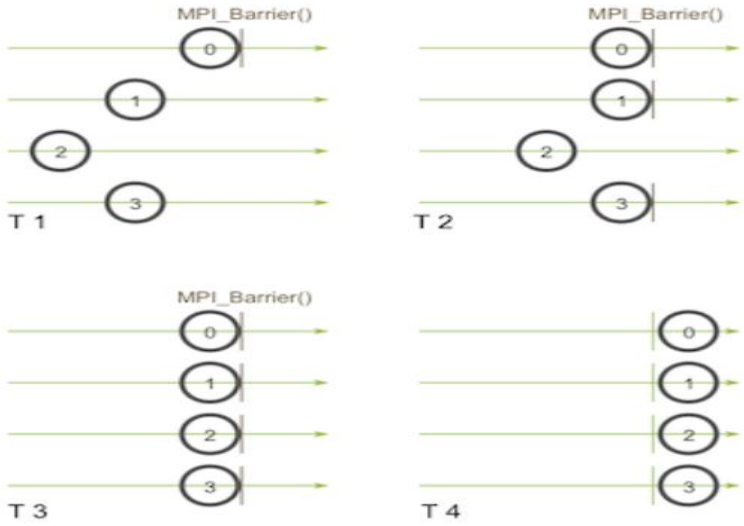
Collective communication

Collective communication refers to a type of communication pattern in parallel and distributed computing, where multiple processes or nodes collaborate to exchange information among themselves. These communication patterns are essential in high-performance computing and distributed systems to efficiently solve problems that require coordinated efforts among multiple participants.

Types of collective communication

Collective communication operations are made of the following types:

1. **Barrier:** A barrier is a synchronization point that forces all processes in a group to wait until they have all reached the barrier before continuing. Barriers are often used to ensure that all processes are at the same point in their execution.



Syntax:

int MPI_Barrier(MPI_Comm communicator);

`communicator`: The communicator that defines the group of processes that synchronize at the barrier.

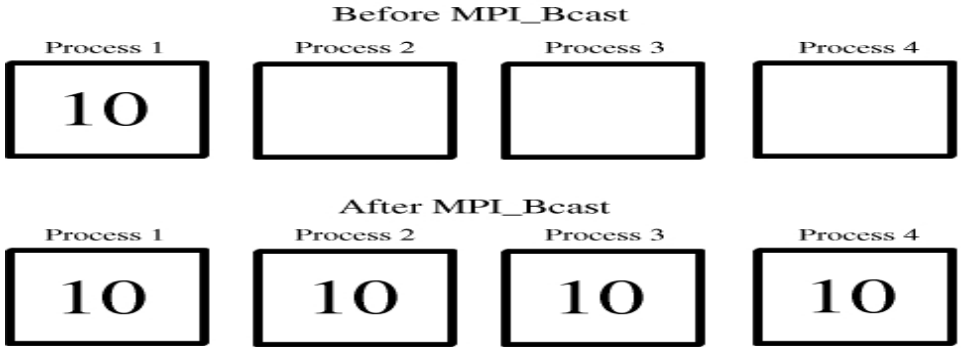
The `MPI_Barrier` function is often used to coordinate the execution of processes in a parallel program. For example, if different processes are performing different parts of a computation and need to ensure that they all reach a certain point before proceeding.

Program

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

```
int main(int argc, char** argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Some computation before the barrier
    printf("Process %d reached the barrier.\n", rank);
    MPI_Barrier(MPI_COMM_WORLD); // All processes wait here until everyone reaches this point
    // Code after the barrier
    MPI_Finalize();
    return 0;
}
```

2. **Data Movement (**or Global Communication):
    - **Broadcast:** In a broadcast operation, one process sends the same data to all other processes in a group. It is often used to distribute information from one process to all others.



Syntax

MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator)

- **data:** A pointer to the data that the root process wants to broadcast. This data is sent by the root process and received by all other processes.
- `count`: The number of data elements in the buffer.
- `datatype`: The datatype of the elements in the buffer.
- `root`: The rank of the process within the communicator that is broadcasting the data.
- `communicator`: The communicator that defines the group of processes over which the broadcast operation is performed.

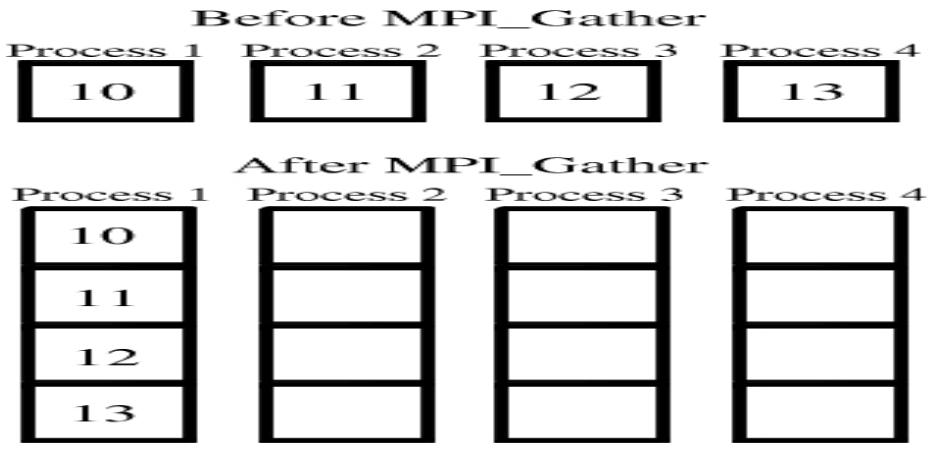
Program

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
int a = 10, r, s;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &s);
MPI_Comm_rank(MPI_COMM_WORLD, &r);
printf("\n data in process %d before bcast=%d", r, a);
MPI_Bcast(&a, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("\n data in process %d after bcast=%d",r,a);
MPI_Finalize(); return 0;
}
```

Output

```
C:\Users\akshith's\source\repos\Project3\x64\Debug>mpiexec -n 4 ./project3.exe

 data in process 0 before bcast=10
 data in process 0 after bcast=10

 data in process 1 before bcast=0
 data in process 1 after bcast=10

 data in process 3 before bcast=0
 data in process 3 after bcast=10

 data in process 2 before bcast=0
 data in process 2 after bcast=10
```

- **Gather:** The gather operation collects data from all processes in a group and sends it to a designated process. This is useful when you want to aggregate data from multiple sources.



Syntax

MPI_Gather(void* sendbuf, int send_count, MPI_Datatype sendtype, void * recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm communicator)

`sendbuf`: A pointer to the send buffer (data to be sent) on each process.

`sendcount`: The number of elements to send from the send buffer.

`sendtype`: The datatype of the elements in the send buffer.

`recvbuf`: A pointer to the receive buffer on the root process. This is where the gathered data will be stored.

`recvcount`: The number of elements to receive from each process.

`recvtype`: The datatype of the elements in the receive buffer.

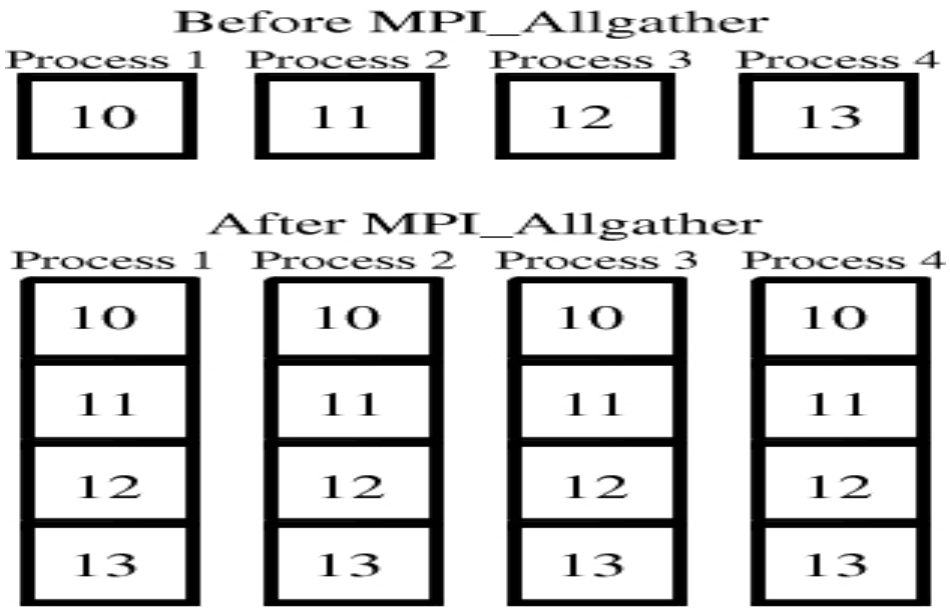
`root`: The rank of the root process, which will receive the gathered data.

`communicator`: The communicator that defines the group of processes.

Program

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
int d = 0, r, s, a[5];
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD,&s);MPI_Comm_rank(MPI_COMM_WORLD, &r);
d = r * 2;
MPI_Gather(&d, 1, MPI_INT, &a, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (r == 0)
{
printf("data received by process o=");
for (int i = 0;i < s;i++)
printf("%d\t", a[i]);
}
MPI_Finalize();return 0;
}
```

- **Allgather :** `MPI_Allgather` distributes the gathered data to all processes in the communicator, not just to the root process. Each process receives the entire gathered dataset.



Syntax

MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype senddatatype, void*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm communicator)

`sendbuf`: A pointer to the send buffer (data to be sent) on each process.

`sendcount`: The number of elements to send from the send buffer on each process.

`sendtype`: The datatype of the elements in the send buffer.

`recvbuf`: A pointer to the receive buffer on each process. This is where the gathered data will be stored.

`recvcount`: The number of elements to receive from each process.

`recvtype`: The datatype of the elements in the receive buffer.

`communicator`: The communicator that defines the group of processes

Here's how `MPI_Allgather` works:

Each process provides data in its send buffer (`sendbuf`).

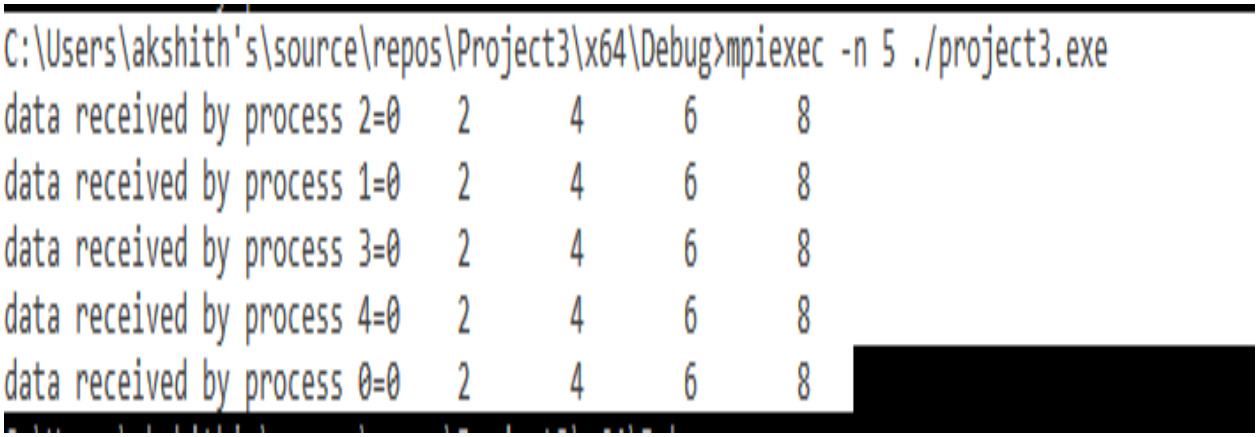
The data from the send buffers of all processes is gathered.

The gathered data is distributed to the receive buffers of all processes (`recvbuf`).

Program

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

```
#include<mpi.h>
int main(int argc, char* argv[])
{
int d = 0, r, s, a[5];
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD,  &s);  MPI_Comm_rank(MPI_COMM_WORLD,
&r);
d = r * 2;
MPI_Allgather (&d, 1, MPI_INT, &a, 1, MPI_INT, MPI_COMM_WORLD);
printf("data received by process %d=",r);
for (int i = 0;i < s;i++)
printf("%d\t", a[i]);
MPI_Finalize(); return 0;
}
```



- **Reduce:** In a reduce operation, data from all processes is combined using an associative and commutative operation (e.g., addition, multiplication) to produce a single result. This is often used for aggregating data or finding global statistics. There are many operations that can be done during the reduction. The most common are

MPI_MAX (maximum value in an array)

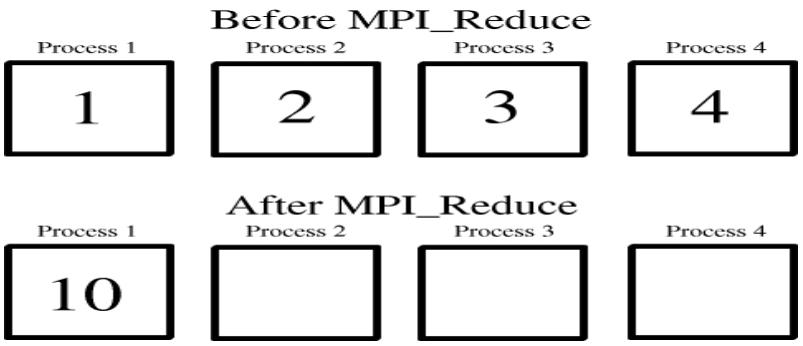
MPI_MIN (minimum value in an array)

MPI_SUM (sum of an array)

MPI_MINLOC (index of minimum value)

MPI_MAXLOC (index of maximum value

Example:



Syntax

• MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Opop, int root, MPI_Comm communicator)

- `sendbuf`: A pointer to the send buffer (data to be reduced) on each process.
- `recvbuf`: A pointer to the receive buffer on the root process. This is where the reduced result will be stored.
- `count`: The number of elements in the send buffer.
- `datatype`: The datatype of the elements in the send buffer.
- `op`: The reduction operation to be performed (e.g., `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`,etc.).
- `root`: The rank of the root process, where the reduced result will be stored.
- `communicator`: The communicator that defines the group of processes

Program

```
int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD, &size); MPI_Comm_rank(MPI_COMM_WORLD,
&rank);
int localsum ;
int globalsum;
localsum = 10 + rank;
printf("process %d value=%d", rank, localsum);
MPI_Reduce(&localsum, &globalsum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0)
```

```
{
printf("\n globalsum = %d", globalsum);
}
MPI_Finalize();
return (0);
}
```

Output for 3 processes: global sum=13

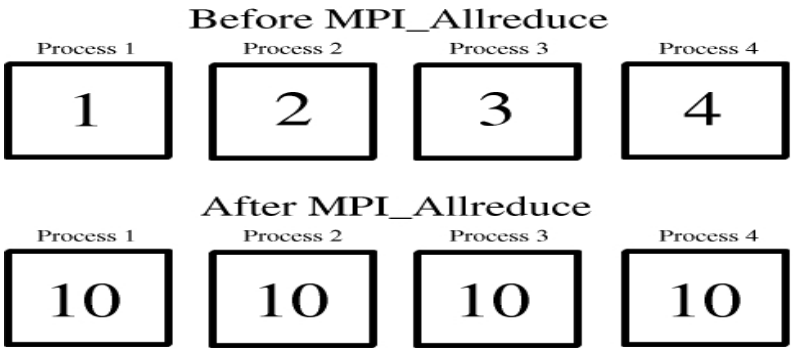
First process local sum=10+0(rank)=10

Second process local sum=10+1(rank)=11

Third process local sum=11+2(rank)=13

- **AllReduce: M**`PI_Allreduce` is a collective communication function in the MPI (Message Passing Interface) standard. It is similar to `MPI_Reduce` in that it performs a reduction operation across all processes in a communicator. However, unlike `MPI_Reduce`, the result of the reduction operation is distributed to all processes, not just the root process. Every process receives the reduced result.

Example



Syntax

- MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm communicator);

`sendbuf`: A pointer to the send buffer (data to be reduced) on each process.

`recvbuf`: A pointer to the receive buffer on each process. This is where the reduced result will be stored.

`count`: The number of elements in the send buffer.

`datatype`: The datatype of the elements in the send buffer.

`op`: The reduction operation to be performed (e.g., `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`, `MPI_LAND`, `MPI_BAND`, `MPI_LOR`, etc.).

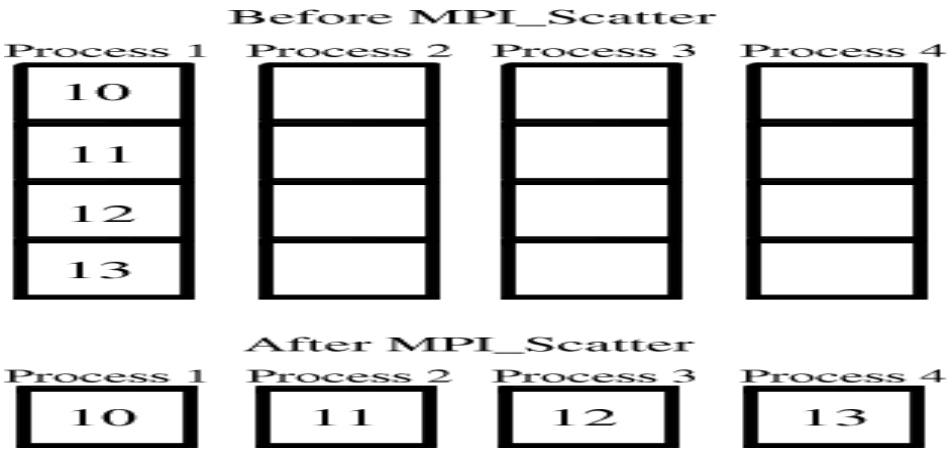
`communicator`: The communicator that defines the group of processes.

`Program`

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
int size, rank;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &size); MPI_Comm_rank(MPI_COMM_WORLD,
&rank);
int globalsum;
printf("process %d value=%d", rank, localsum);
MPI_Allreduce(&rank, &globalsum, 1, MPI_INT, MPI_SUM,  MPI_COMM_WORLD);
printf("\n globalsum = %d", globalsum);
MPI_Finalize();
return (0);
}
```

Output for 3 processes: global sum=13 ,3 times for 3 process

- **Scatter:** Scatter is the opposite of gather. It takes data from one process and distributes it to all other processes in a group. Each process receives a different portion of the data.



Syntax

MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm communicator);

`sendbuf`: A pointer to the send buffer (data to be scattered) on the root process.

`sendcount`: The number of elements to send from the send buffer on the root process.

`sendtype`: The datatype of the elements in the send buffer.

`recvbuf`: A pointer to the receive buffer on each process. This is where the scattered data will be stored.

`recvcount`: The number of elements to receive on each process.

`recvtype`: The datatype of the elements in the receive buffer.

`root`: The rank of the root process, which is the source of the scattered data.

`communicator`: The communicator that defines the group of processes.

Program

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
int d = 0, r, s, * buf=NULL;MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &s);MPI_Comm_rank(MPI_COMM_WORLD, &r);
if(r == 0)
{
int a[5] = { 1,2,3,4,5 };
buf = a;
}
printf("\n data in process %d before scatter=%d", r, d);
MPI_Scatter(buf, 1, MPI_INT, &d, 1, MPI_INT, 0, MPI_COMM_WORLD);
printf("\n data in process %d after scatter=%d", r, d);MPI_Finalize();
return 0;
}
```

Output

```
:\Users\akshith's\source\repos\Project3\x64\Debug>mpiexec -n 5 ./project3.exe

data in process 4 before scatter=0
data in process 4 after scatter=5

data in process 2 before scatter=0
data in process 2 after scatter=3

data in process 1 before scatter=0
data in process 1 after scatter=2

data in process 3 before scatter=0
data in process 3 after scatter=4

data in process 0 before scatter=0
data in process 0 after scatter=1
```

UNIT-2

TOPIC 6

Data Parallel Examples

The data parallel strategy is the most common approach in parallel applications.

First, a simple case of the stream triad where no communication is necessary.

The Stream Triad benchmark measures the memory bandwidth of a computing system. It is a simple yet effective benchmark to assess the memory performance of a node. The Stream Triad benchmark calculates the memory bandwidth by performing a simple operation on arrays in memory.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define ARRAY_SIZE (1 << 20) // 1 million elements (adjust based on your system)
#define SCALAR 2.0
int main() {
    double *a, *b, *c;
    int i;
    double start_time, end_time;
    double bandwidth;

    // Allocate memory for arrays
    a = (double*)malloc(ARRAY_SIZE * sizeof(double));
    b = (double*)malloc(ARRAY_SIZE * sizeof(double));
    c = (double*)malloc(ARRAY_SIZE * sizeof(double));
    // Initialize arrays
    #pragma omp parallel for
    for (i = 0; i < ARRAY_SIZE; i++) {
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }
    // Measure start time
    start_time = omp_get_wtime();
    // Perform Stream Triad operation
```

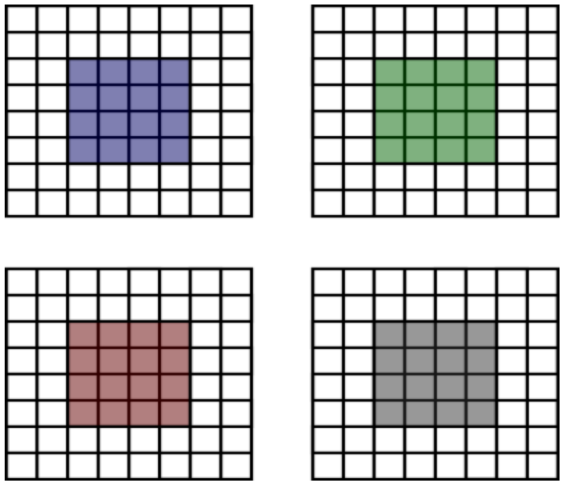
```
    #pragma omp parallel for
    for (i = 0; i < ARRAY_SIZE; i++) {
        c[i] = a[i] + b[i] * SCALAR;
    }
    // Measure end time
    end_time = omp_get_wtime();
    // Calculate bandwidth in GB/s
    bandwidth = (3 * ARRAY_SIZE * sizeof(double)) / ((end_time - start_time) * 1e9);
    // Print bandwidth
    printf("Memory Bandwidth: %f GB/s\n", bandwidth);
    // Free allocated memory
    free(a);
    free(b);
    free(c);
    return 0;
}
```

Second the more typical ghost cell exchange techniques used to link together the subdivided domains distributed to each process

Ghost cell exchange/updates are a common technique in computational science and high-performance computing, particularly in the context of numerical simulations, to manage boundary conditions and ensure accurate results when performing computations on a grid or mesh. Ghost cells, also known as halo cells or boundary cells, are additional grid cells that surround the main computational domain.

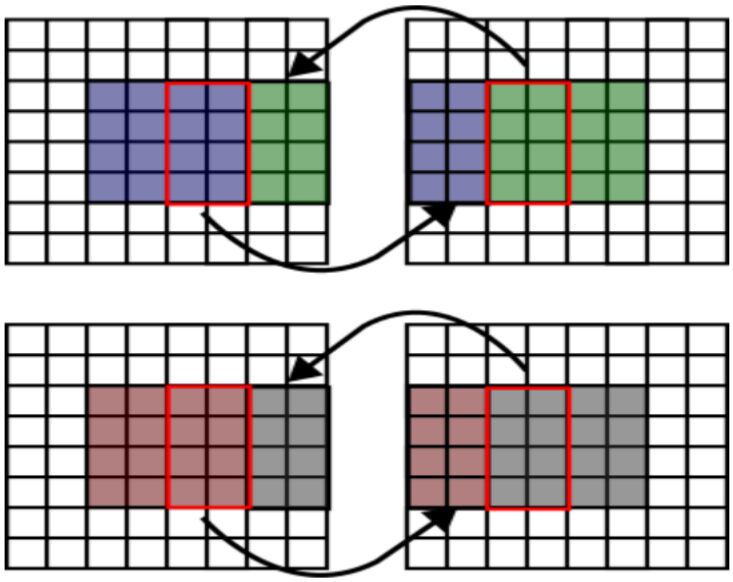
The update of ghost cells involves exchanging data between neighbouring chunks to ensure that each chunk has the correct information about its boundary cells. This communication is necessary because the computation in one chunk often depends on the values of neighbouring cells.

To exchange halo, first we have four block like this:



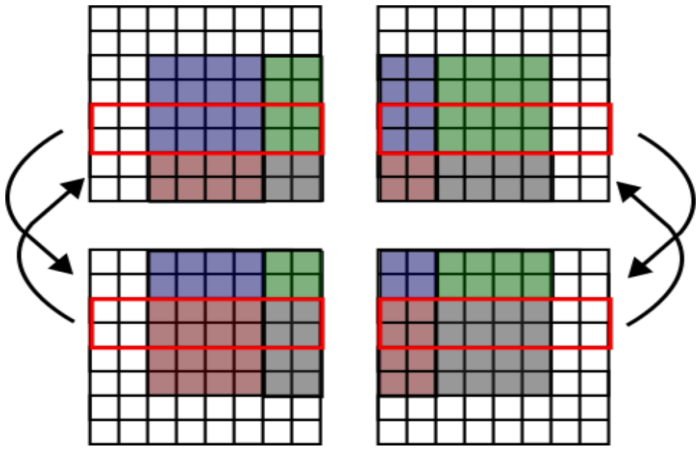
The white cells represeent the halo cells which can be used for temporary storage of values while exchanging the data.Each mesh is assigned to different processors for performing operations.

The first communication happens along x-axis



We can see how the cells are exchanged from one mesh to another with the help of halo cells.(Colors are swapped).

The second communcation happens along y-axis. Note that here the communicated boundary extended to ghosts, this is necessary to have corners transfered correctly:



Advanced MPI functionality to simplify code and enable optimizations

The advanced functions that are useful in common data parallel applications.

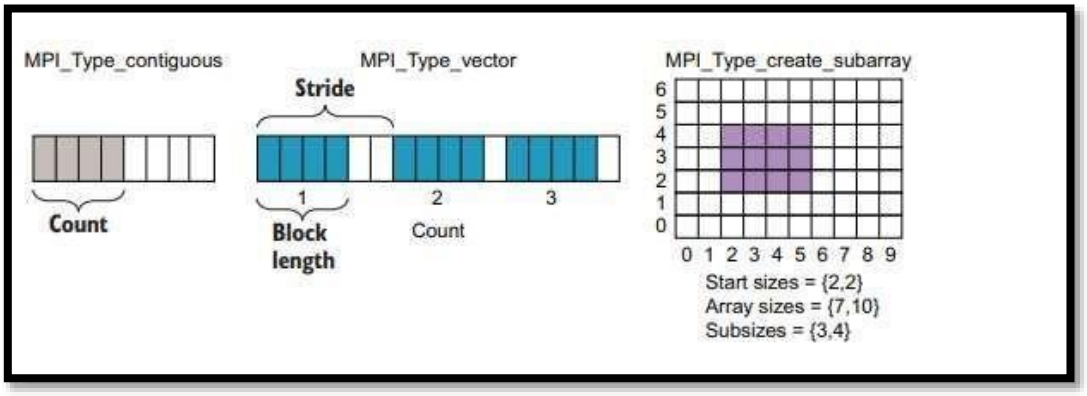
- **MPI custom data types**
- **Topology support**

- **Custom MPI Data Types:**

MPI has a rich set of functions to create new, custom MPI data types from the basic MPI types

- MPI_Type_contiguous—makes a block of contiguous data into a type.
- MPI_Type_vector—creates a type out of blocks of strided data. (elements in strided data are not necessarily contiguous in memory; there are gaps between them).
- MPI_Type_create_subarray—creates a rectangular subset of a larger array.
- MPI_Type_create_struct—Creates a data type encapsulating the data items in a structure in a portable way that accounts for padding by the compiler

Three MPI custom data types with illustrations of the arguments used in their creation



A type must be committed before use and it must be freed to avoid a memory leak. The routines include

- MPI_Type_Commit—Initializes the new custom type with needed memory allocation or other setup
- MPI_Type_Free—Frees any memory or data structure entries from the creation of the data type

- **Topology support**

Cartesian topology support in MPI (Message Passing Interface) allows you to define a logical, multi-dimensional grid or mesh of processes to facilitate communication and coordination in parallel applications. This is particularly useful for simulations, numerical computations, and other scientific computing tasks where data is organized in multi-dimensional arrays or grids.

To create a Cartesian topology in MPI, you typically follow these steps:

1. Initialize MPI and determine the size and rank of your MPI communicator.

```
#include <mpi.h>
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int size, rank;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Finalize();
   return 0;
}
```

2. Define the dimensions and periodicity of the grid using an integer array, and create the Cartesian communicator using `MPI_Cart_create`

```
int dims[ndims];  // Array specifying the number of processes in each dimension
int periods[ndims];  // Array indicating whether the grid is periodic in each dimension
MPI_Comm cart_comm;
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, 0, &cart_comm);
```

Parameters:

MPI_COMM_WORLD:This is the communicator representing all the processes that are involved in the original communication world.

ndims:An integer representing the number of dimensions of the Cartesian grid. For example, ndims = 2 for a 2D grid (like a matrix) or ndims = 3 for a 3D grid.

dims[]:An array of integers specifying the number of processes in each dimension. For example, for a 2D grid with 4 processes in each dimension, dims = {4, 4} would mean a grid with 16 processes in total.

periods[]:An array of integers of size ndims that specifies whether the grid should be periodic in each dimension.

1 indicates periodic (like wrapping around, torus-like),

0 indicates non-periodic (no wrapping).

0(reorderflag):This flag indicates whether process ranks should be reordered to optimize the topology.

If set to 1, processes may be reordered for performance reasons.

If set to 0, the rank assignment remains unchanged.

&cart_comm:A pointer to the new communicator that will be created. This new communicator will include the processes arranged in the specified Cartesian topology. cart_comm will be used in future communication within this topology.

3. Retrieve the coordinates of each process in the Cartesian grid using `MPI_Cart_coords`.

```
int coords[ndims];
MPI_Cart_coords(cart_comm, rank, ndims, coords);
```

Parameters:

comm:The communicator with Cartesian structure (the communicator created by MPI_Cart_create).

rank:The rank of the process whose coordinates you want to determine (within the

Cartesian communicator).

maxdims:The number of dimensions in the Cartesian grid (same as ndims passed to MPI_Cart_create).

coords[]:An integer array that will hold the coordinates of the process. The array should be of size maxdims to hold the Cartesian coordinates in each dimension.

Return Value:

The function returns an array of integers (coords) containing the coordinates of the process in the Cartesian topology. The return value is MPI_SUCCESS if the function completes successfully.

4. Perform point-to-point communication or collective operations specific to your application's grid structure, often using functions like `MPI_Send`, `MPI_Recv`, and collective operations like `MPI_Allreduce`, `MPI_Gather`, or `MPI_Scatter`.
5. After you're done, free the Cartesian communicator using `MPI_Comm_free`
   MPI_Comm_free(&cart_comm);

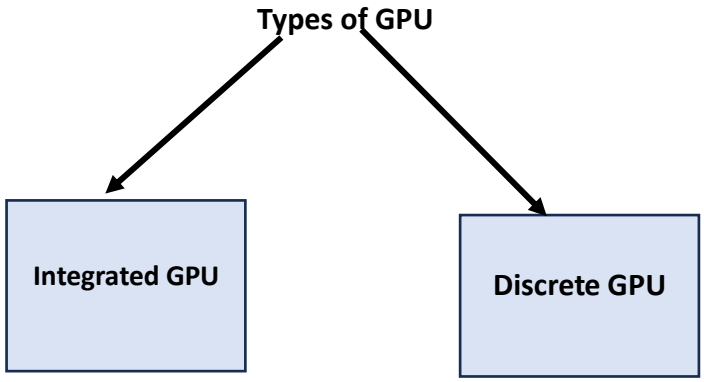
Unit-3

Topic 1

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(Advanced Micro Devices) 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

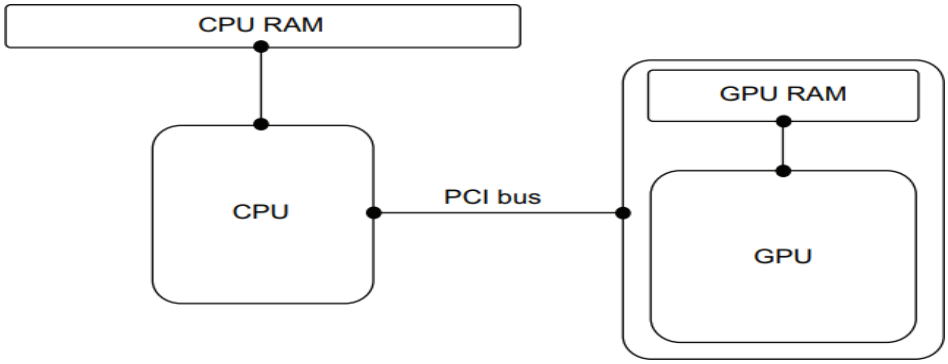


Figure : Block diagram of GPU-accelerated system using a dedicated GPU. The CPU and GPU each have their own memory. The CPU and GPU communicate over a PCI bus.

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 : 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.

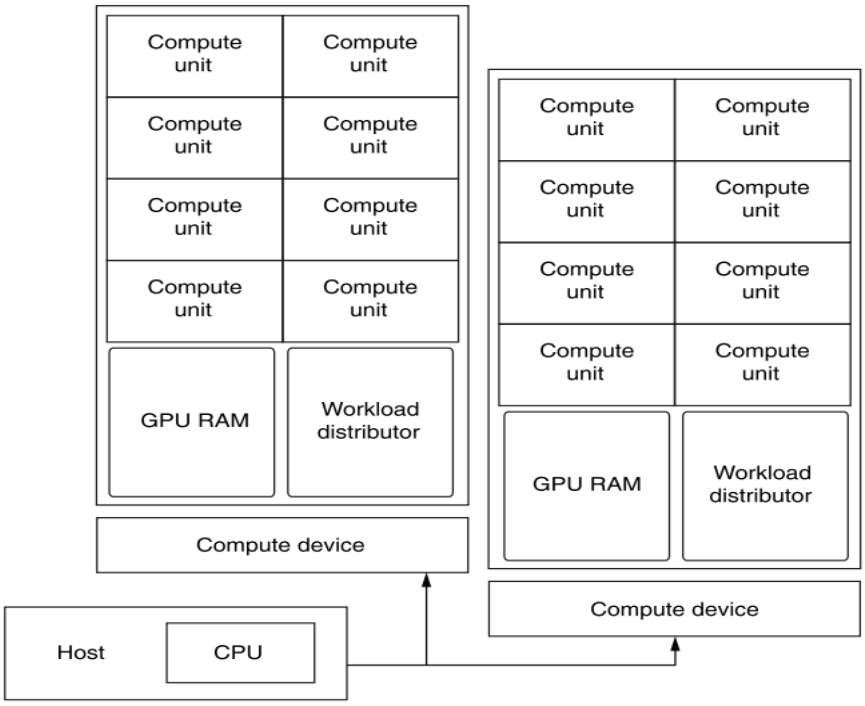
The GPU and the thread engine

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.

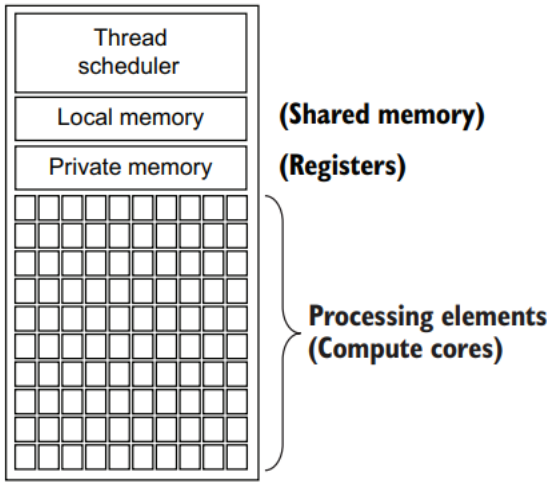


Figure 9.3 Simplified block diagram of a compute unit (CU) with a large number of processing elements (PEs).

- **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 : 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.

UNIT-III

Topic 2: 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.

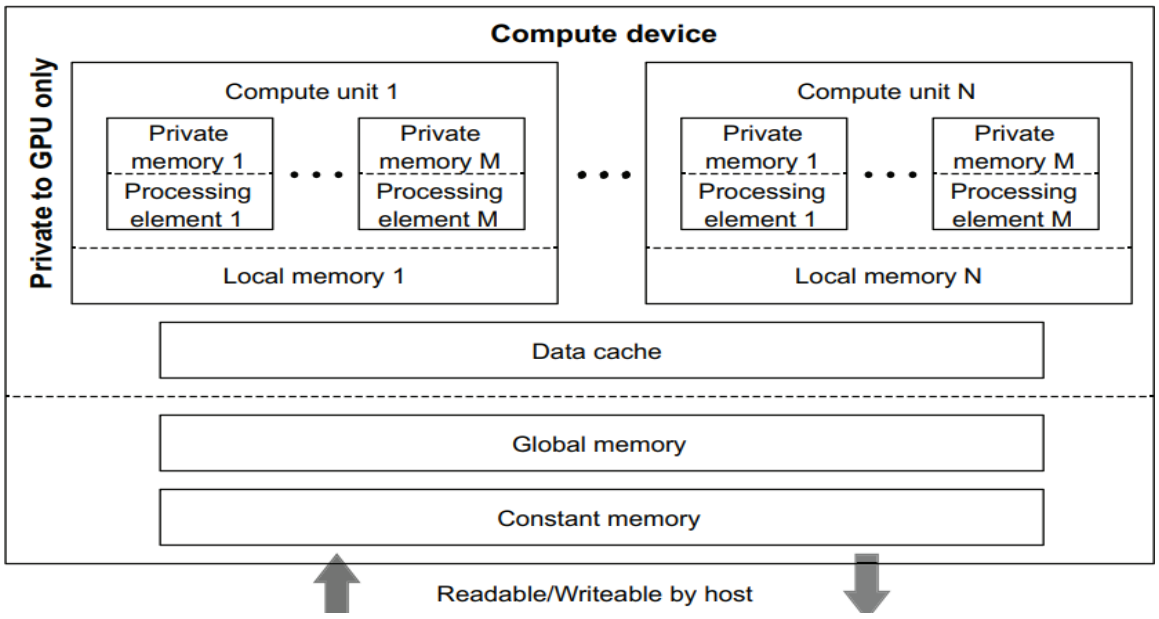


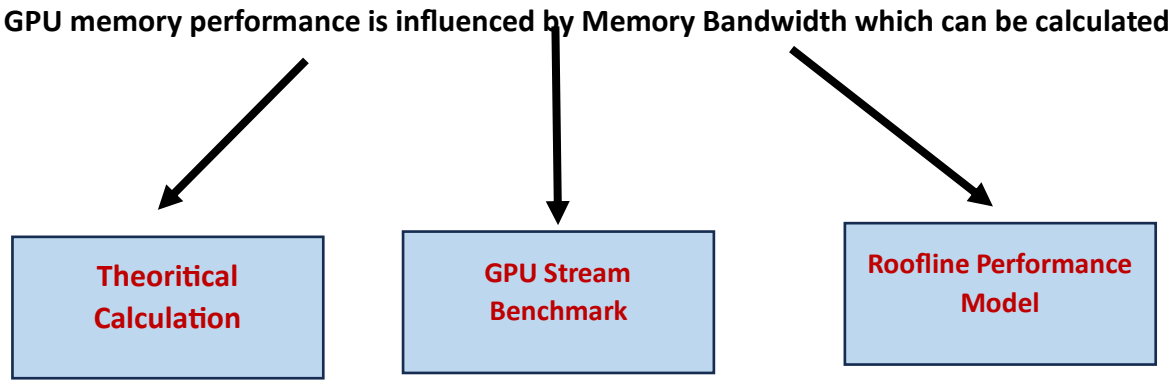
Figure 9.4 Rectangles show each component of the GPU and the memory that is at each hardware level. The host writes and reads the global and constant memory. Each of the CUs can read and write from the global memory and read from the constant memory

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 Compute Unit and all of the Processing Elements on that Compute 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

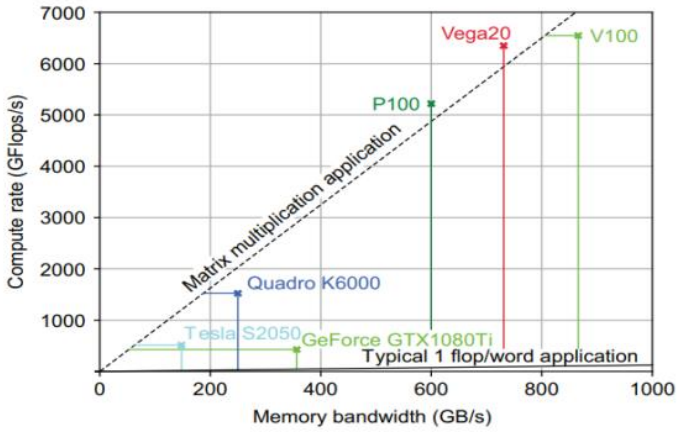
The performance of GPU memory, also known as memory subsystem performance, is a critical factor that significantly impacts the overall performance of GPU-accelerated applications.



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

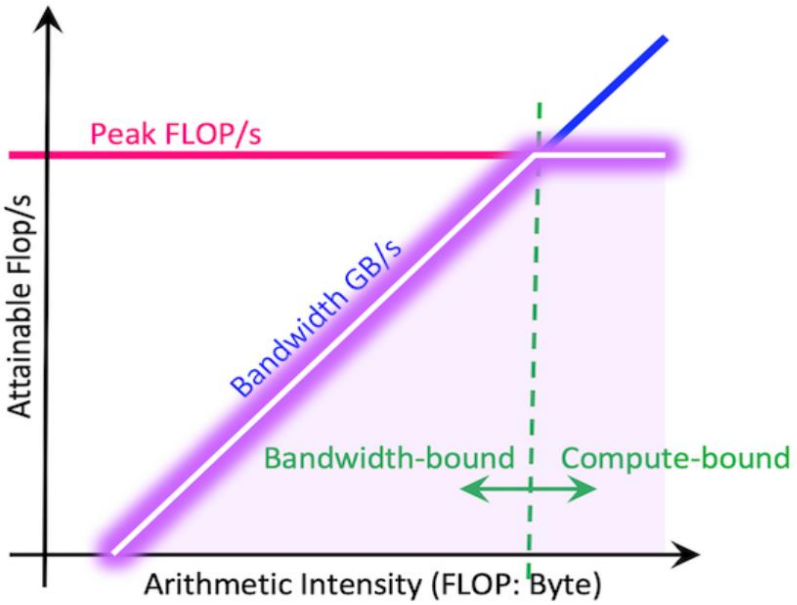
GPU Stream Bench Mark: In the context of GPUs (Graphics Processing Units) and general computing, a benchmark refers to a standardized test or set of tests designed to measure the performance of a GPU or an entire computer system. Benchmarks are used to evaluate various aspects of a GPU's capabilities, 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.

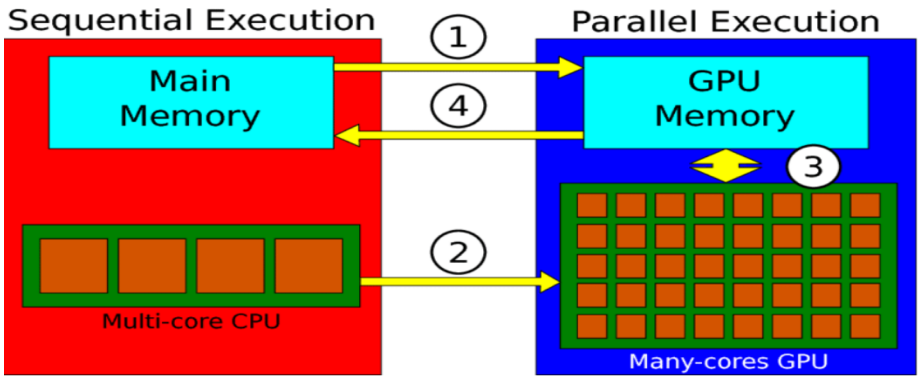
Roofline Performance Model: The Roofline Performance Model is a graphical representation used in high-performance computing (HPC) to analyze and visualize the performance of algorithms on a particular hardware architecture. The Roofline model serves as a visual communication tool that can be easily shared 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.



Topic 3:

The PCI bus: CPU to GPU data transfer overhead

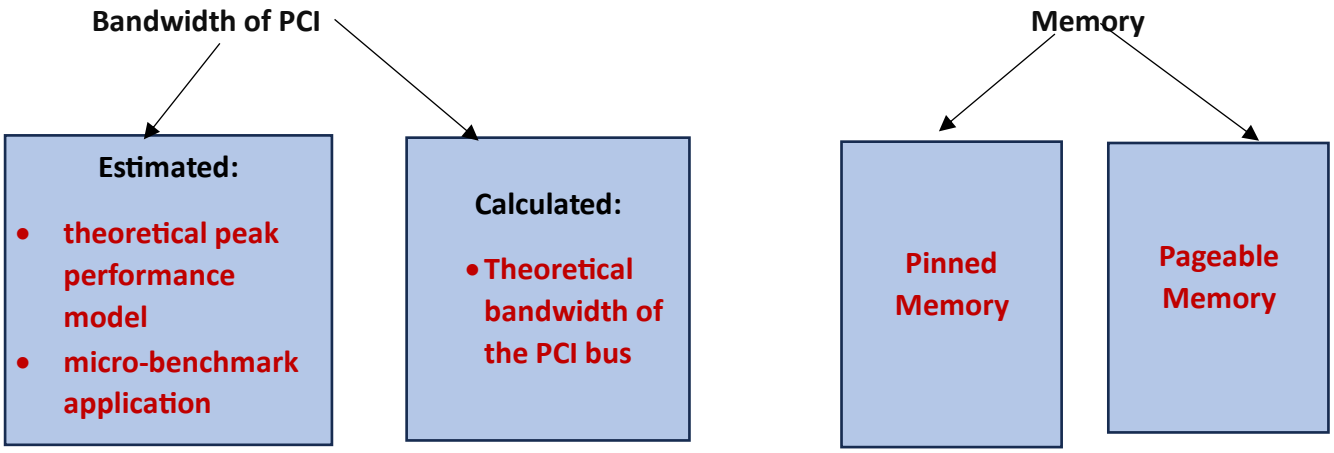
The PCI (Peripheral Component Interconnect) bus is a standard interface that connects various hardware devices, including GPUs (Graphics Processing Units), to a computer's motherboard. The current version of the PCI bus is called PCI Express (PCIe).



The Datatransfer includes the following steps for a typical program execution 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.**



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.

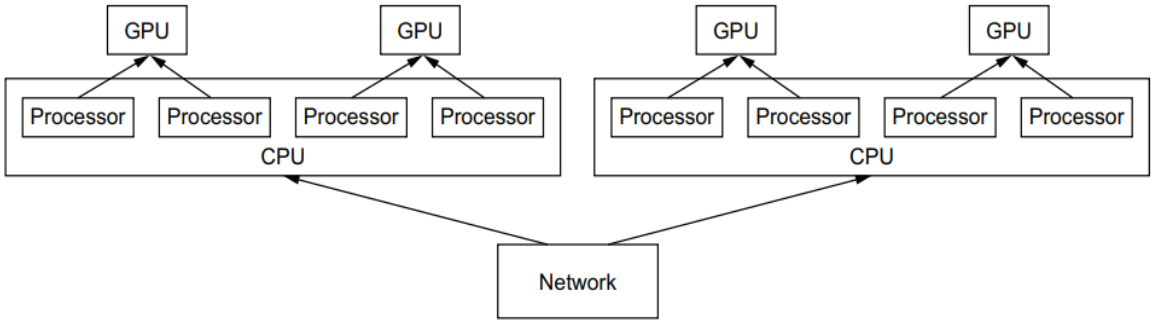
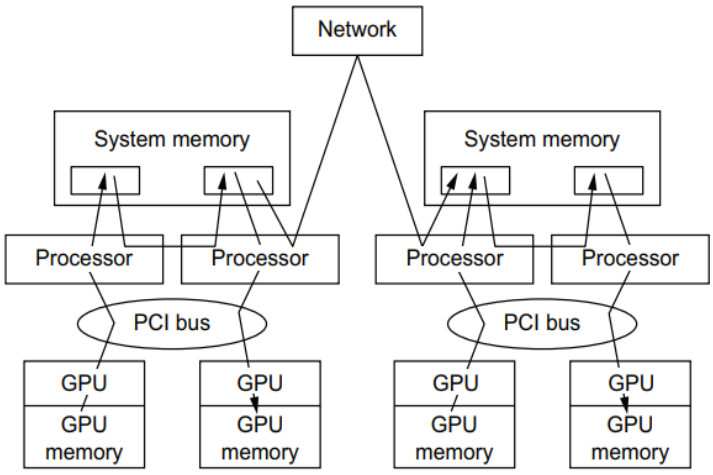


Figure 9.9 Here we illustrate a multi-GPU platform. A single compute node can have multiple GPUs and multiple processors. There can also be multiple nodes connected across a network.

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)

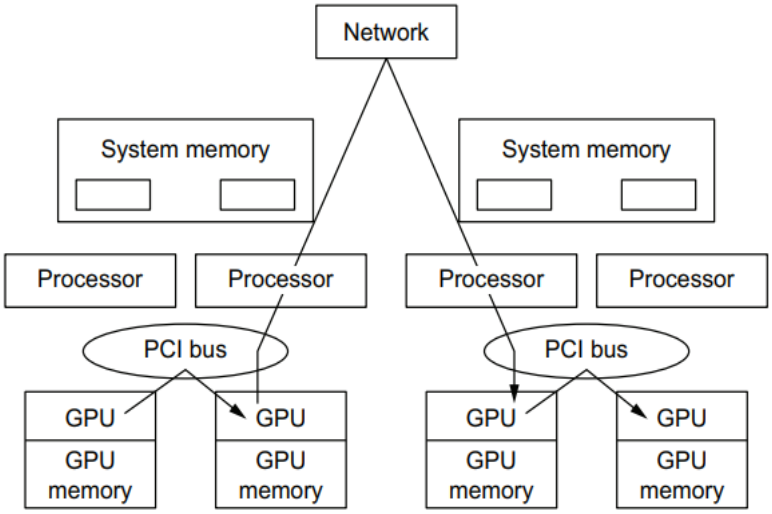
c Store the data from the processor to CPU memory

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 not 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.

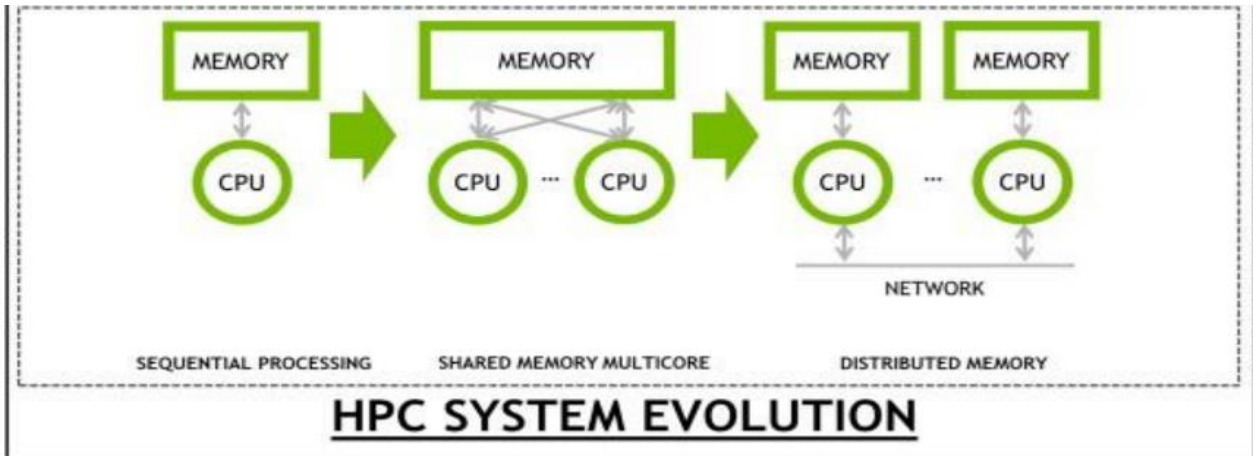
- **Lack of parallelism**— "With great power comes great need for parallelism." If you don't have the parallelism, GPUs can't do a lot for you. This is the first law of GPGPU programming.
- **Irregular memory access**—CPUs also struggle with this. The massive parallelism of GPUs brings no benefit to 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.

Topic 4: 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.

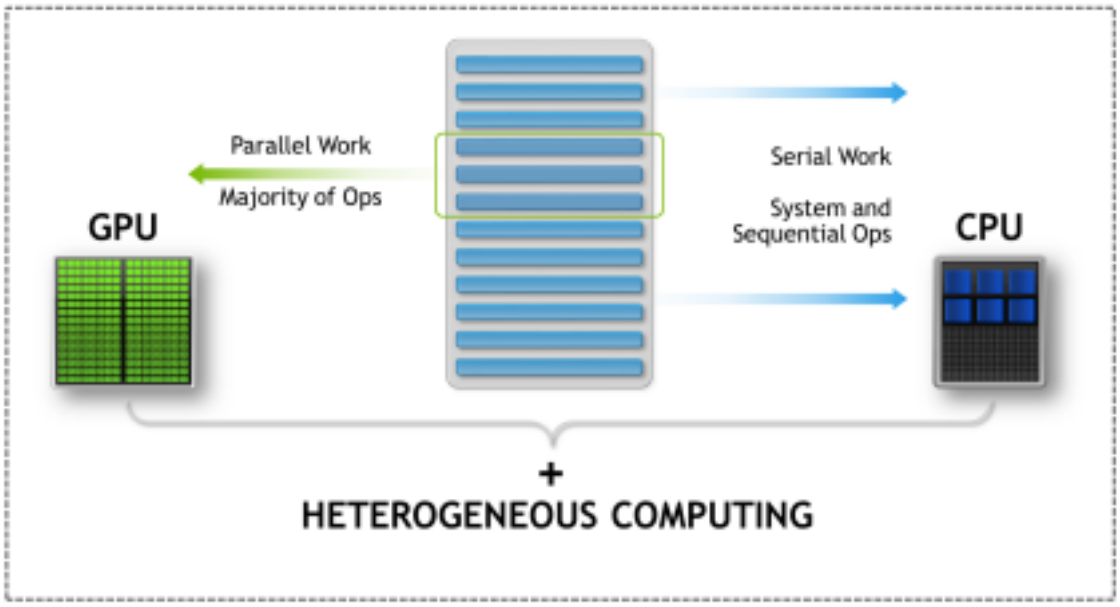
The following diagram shows the evolution of computer architecture from sequential processing to distributed memory



- **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.

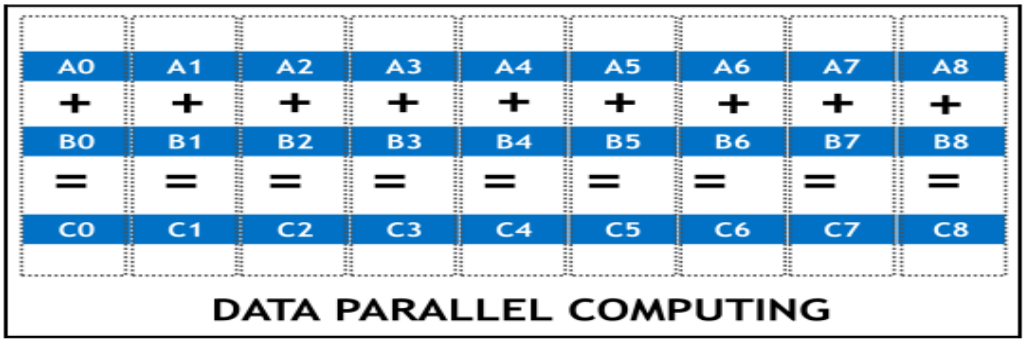
- **Programming paradigm**

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.

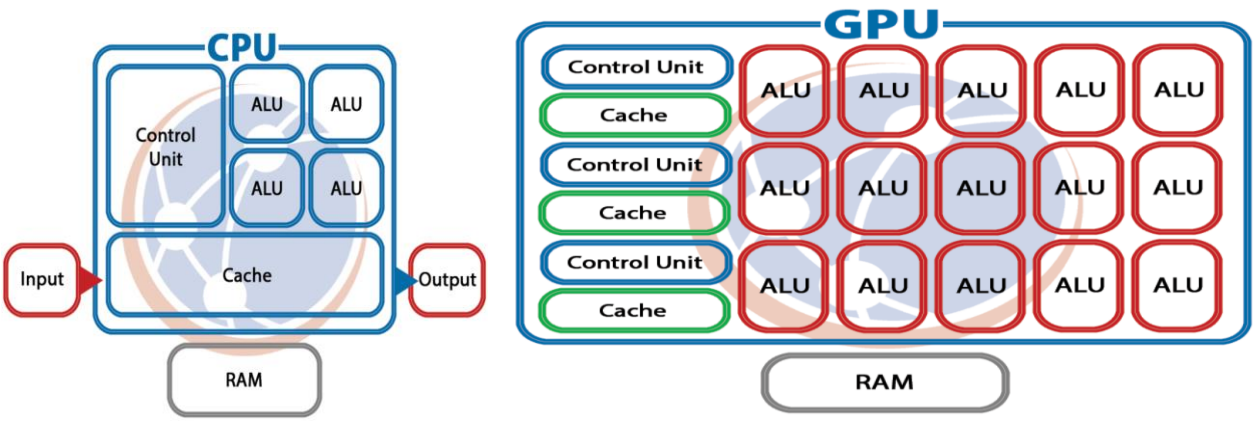
Example : In SIMD, we might have a vector addition operation where a single instruction is applied to corresponding elements of two vectors. In SIMT, we might use CUDA to perform vector addition with multiple threads executing the same instruction on different elements.

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



- **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.

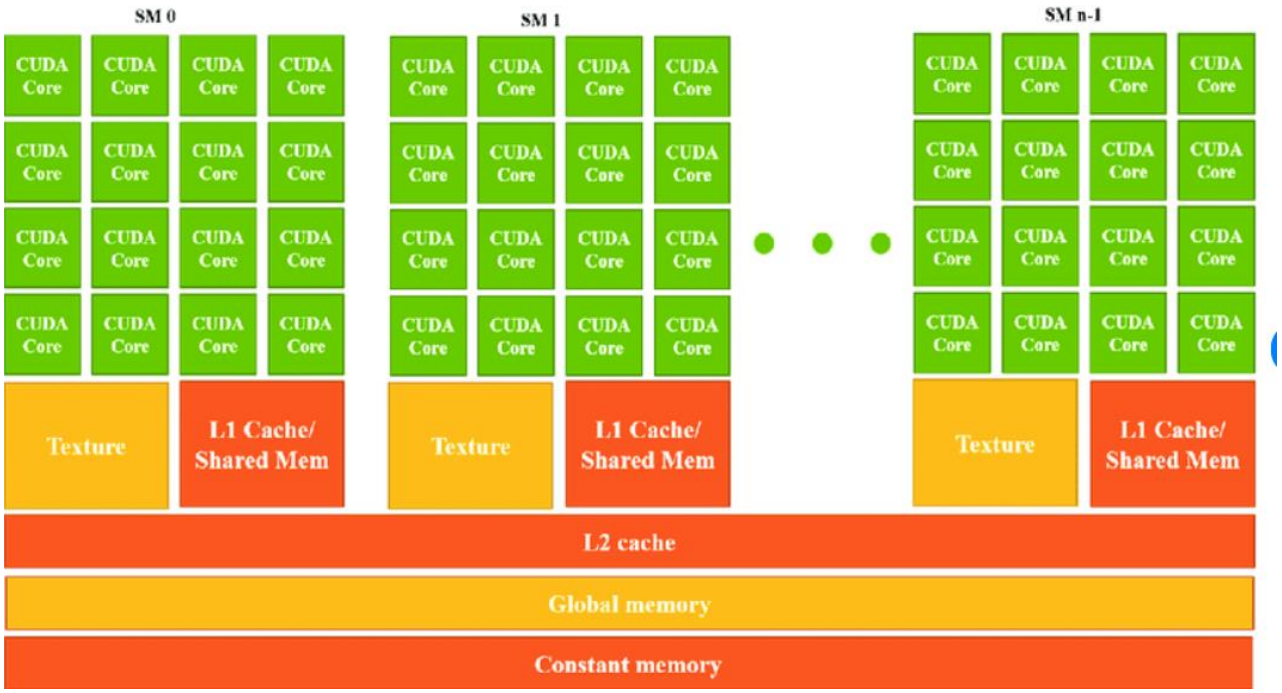
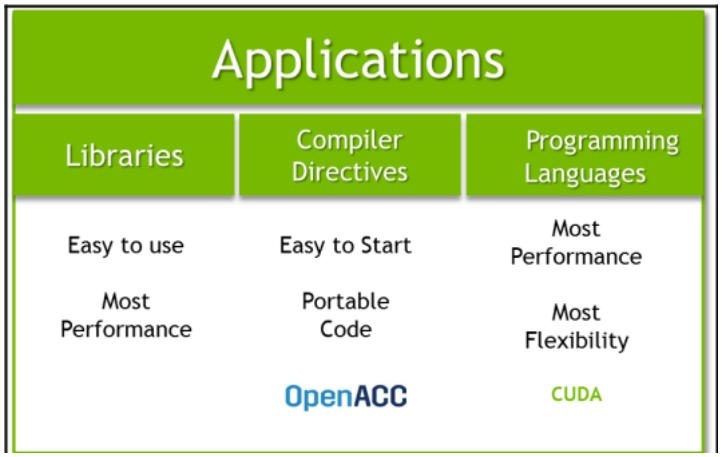


Fig: Schematic of NVIDIA GPU architecture, where SM refers to streaming multiprocessor.

- **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:



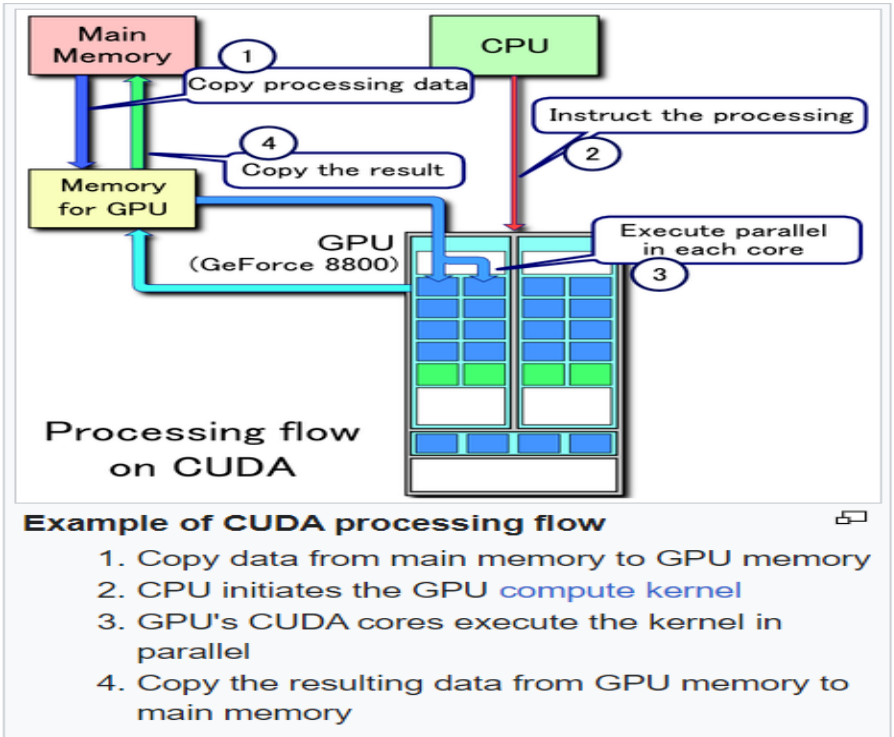
CUDA

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

Topic 5:

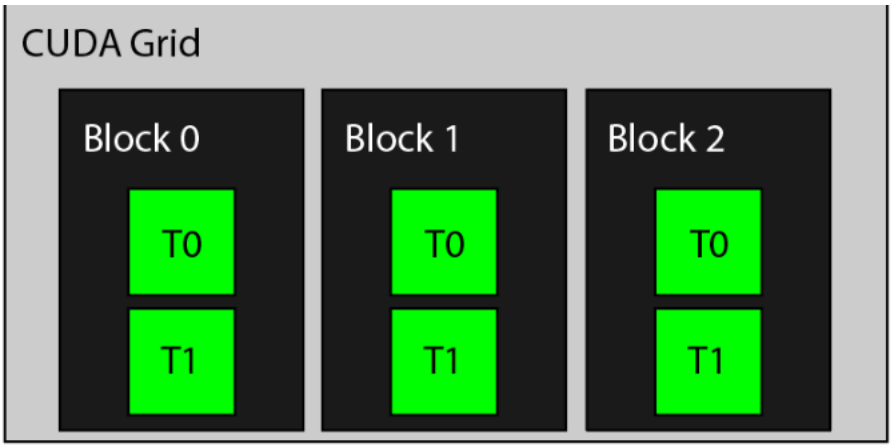
Hello world from CUDA

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.

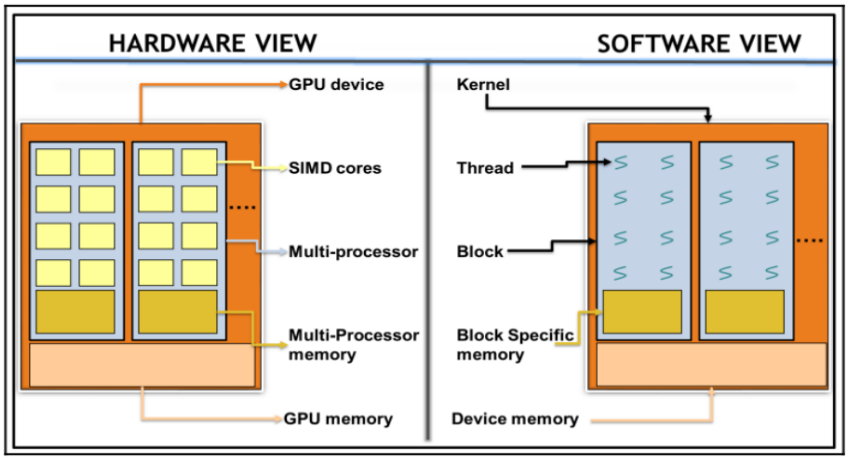
Block: A block contains numerous threads

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

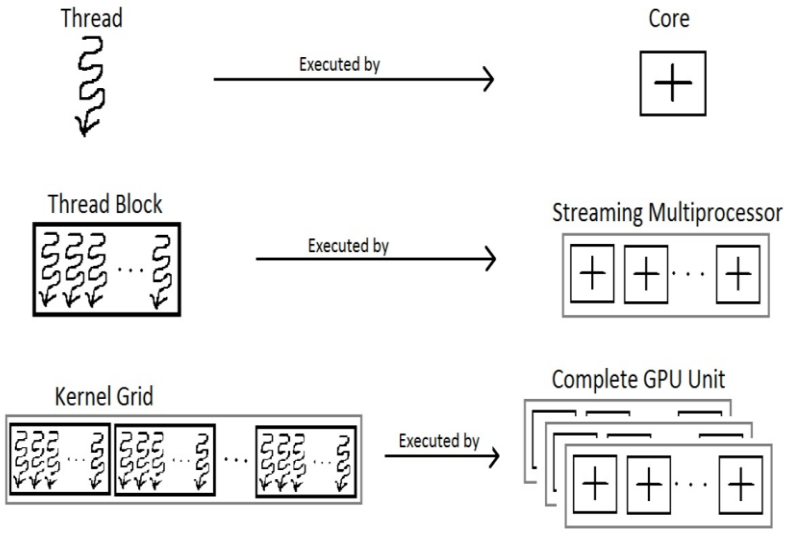


The following figure, in accordance with the preceding screenshot, explains software and hardware mapping in terms of the CUDA programming model:

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



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

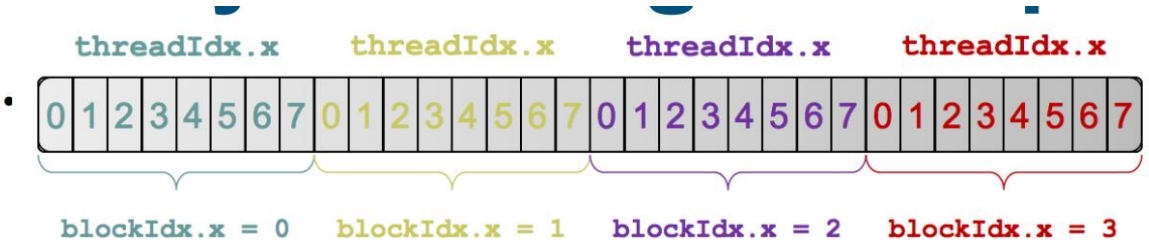
```
__global__ void cuda_hello()
{
   printf("Hello World from GPU!\n");
}

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". Data parallel 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 hello()
{
   printf("hello from GPU  %d \t %d\n",blockIdx.x,threadIdx.x);

}
int main()
{
  printf("hello from CPU\n");
   hello<<<2,2>>>();
 cudaDeviceSynchronize();
 return 0;
}
```

Output

```
hello from CPU
hello from GPU 0 0
 hello from GPU 0 1
hello from GPU 1 0
hello from GPU 1 1
```

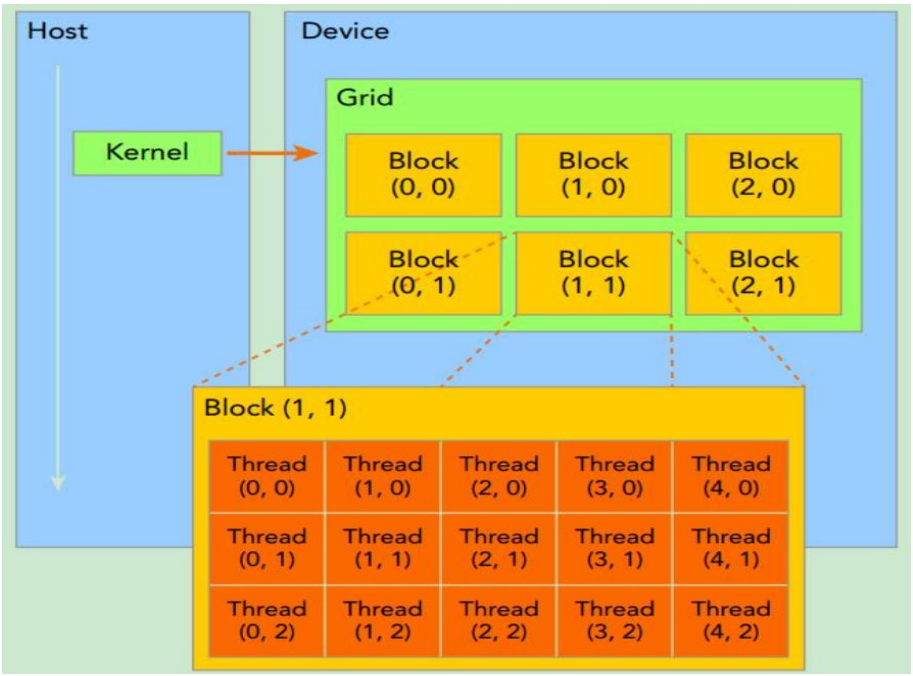
Topic 6:

Thread Hierarchy & Vector Addition

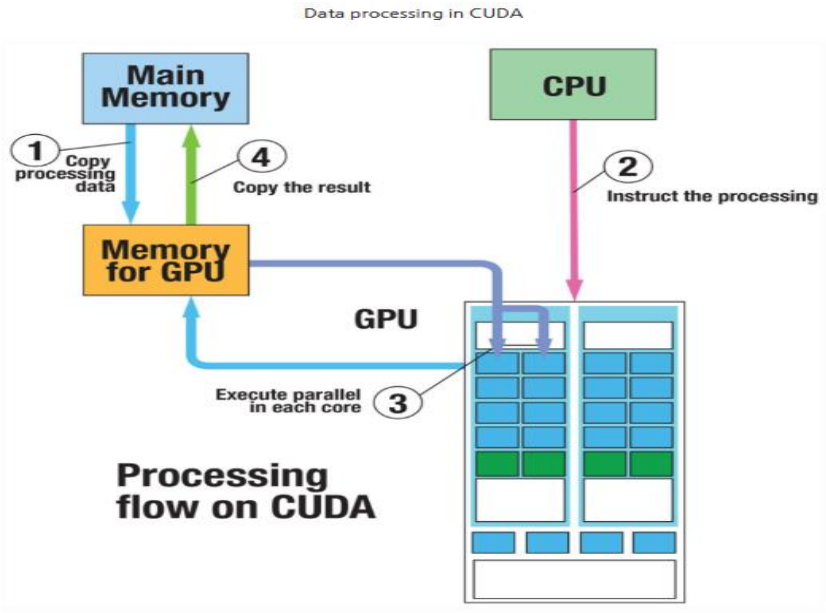
The CUDA thread hierarchy is composed of a grid of thread blocks.

- *Thread block*: A thread block is a set of concurrently executing threads that reside on the same SM; share the resources of that SM, and cooperate among themselves using different hardware mechanisms. Each thread block has a block ID within its grid. A thread block can be one, two, or three dimensional.
- *Grid* : A grid is an array of thread blocks launched by a kernel, that read inputs from global memory; write results to global memory, and synchronize dependency among nested kernel calls. A grid will be described by a user and can be one, two, or three dimensional.

The following image represents an abstract view of the CUDA thread hierarchy.



CUDA Processing Flow



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.

Number of blocks in the grid:

gridDim.x — number of blocks in the x dimension of the grid

gridDim.y — number of blocks in the y dimension of the grid

Number of threads in a block:

blockDim.x — number of threads in the x dimension of the block

blockDim.y — number of threads in the y dimension of the block

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:

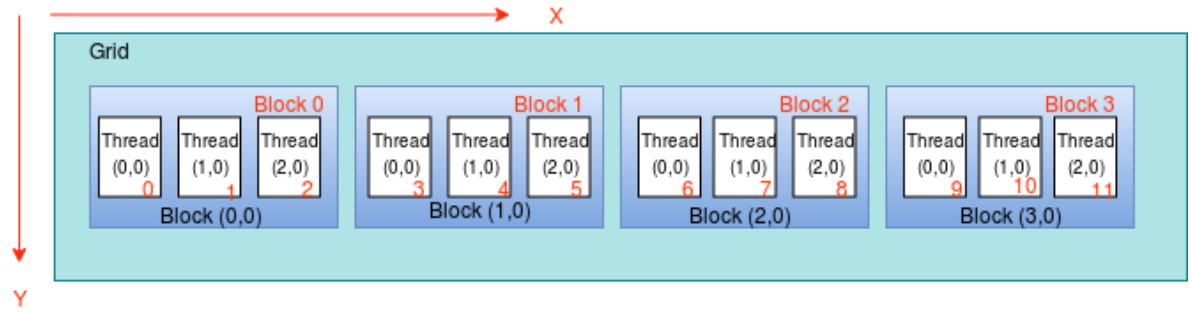
threadIdx.x — thread's index in x dimension

threadIdx.y — thread's index in y dimension

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

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

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

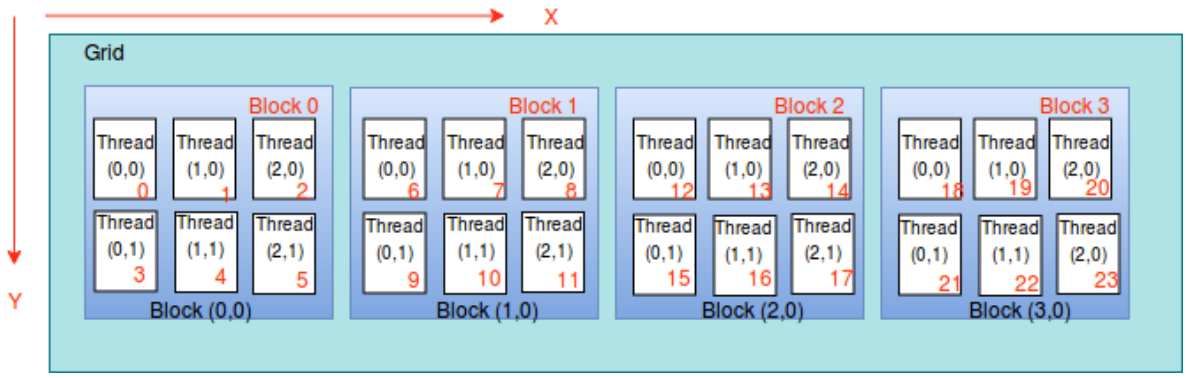
Vector addition using CUDA

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;
}
```

1D grid of 2D blocks



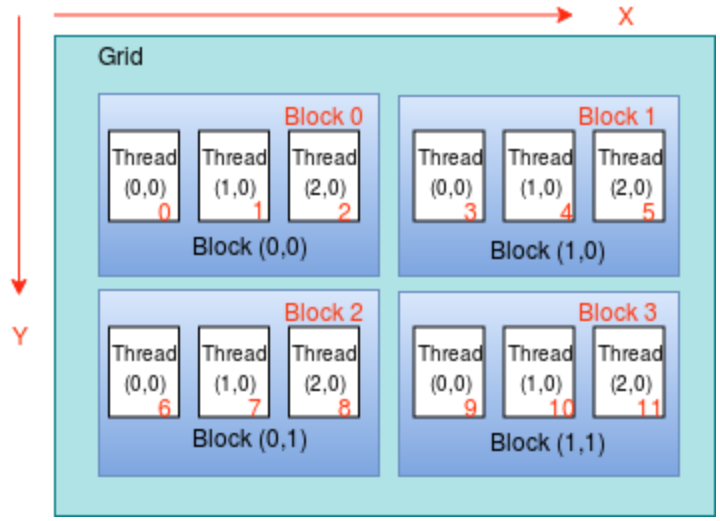
Indices given in RED color are the unique numbers for each block and each thread

*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

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*

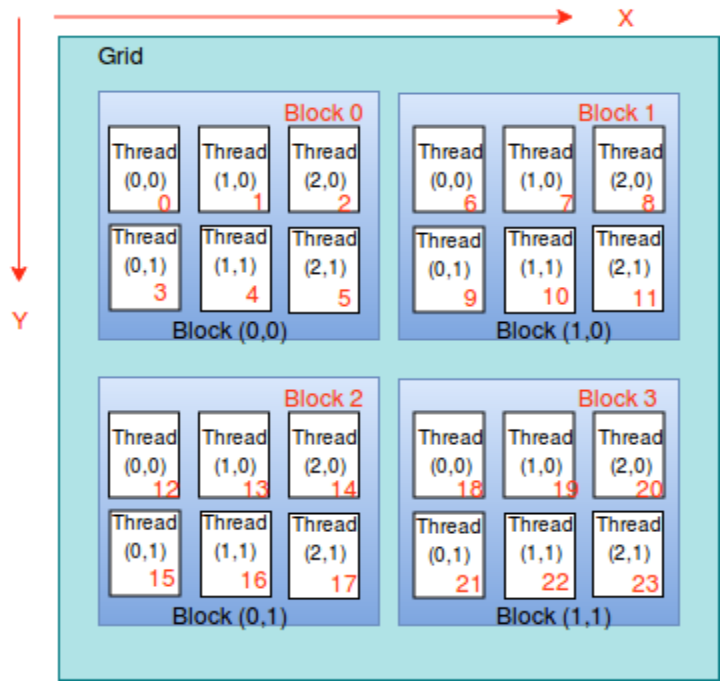
*threadId = (blockId * blockDim.x) + threadIdx.x*

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

Topic 7: Matrix Addition

Matrix addition(2D grid with 1D block(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;
}
```

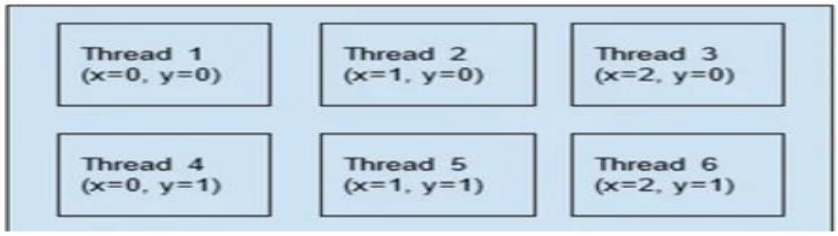
In CUDA programming, **dim3** is a built-in data type used to represent three-dimensional sizes or indices. It is often used to specify the dimensions of a grid or block in CUDA kernels.

Matrix addition(1DGrid(only one block) with 2D block)

```
%%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]={{2,2,2},{3,3,3}},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;
}
```

Here the block is represented as 2D (2*3 matrix) and the values are stored in matrix A & B.But internally they are represented as 1D Array and index values are calculated in the function add.



Matrix A

1	2	3
4	5	6

Matrix B

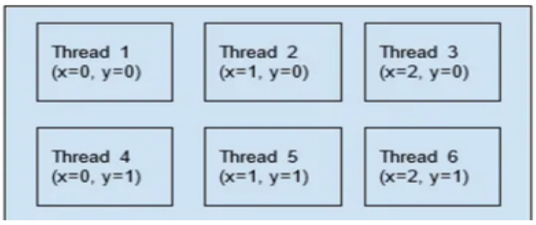
2	2	2
3	3	3

ndex

0	1	2	3	4	5
1	2	3	4	5	6

Value

0	1	2	3	4	4
2	2	2	3	3	3



Thread 1 Id will be 0 and so on

```
int x=threadIdx.x;
int y=threadIdx.y;
int id=blockDim.x*y+x;
f[id]=d[id]+e[id];
```

X=0,y=0
Int x=0
Int y=0
Int id=3*0+0=0

X=1,y=0
Int x=1
Int y=0
Int id=3*0+1=1

X=2,y=0
Int x=2
Int y=0
Int id=3*0+2=2

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:

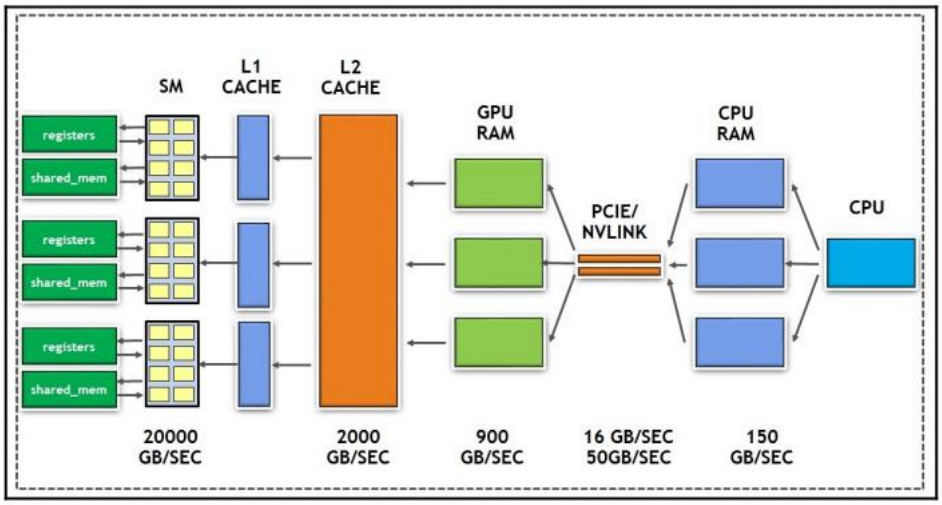
```
struct __align__(16)
{
float r; int g;
}
```

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.

UNIT- IV

CUDA Memory Management-I

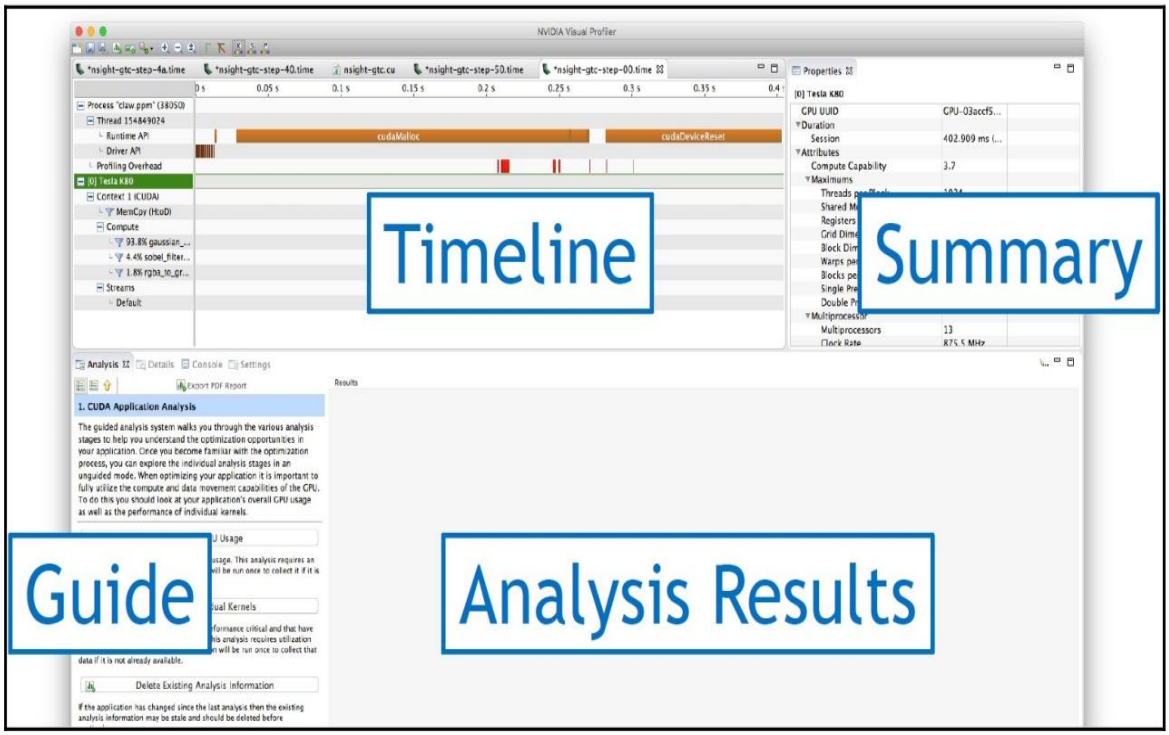
CUDA memory management is a crucial aspect of programming for NVIDIA GPUs using CUDA. It involves allocating, copying, and freeing memory on both the CPU and GPU, as well as understanding the different types of memory available.



In the preceding diagram, we can see the data path from the CPU until it reaches the registers where the final calculation is done by the ALU/cores.

NVIDIA Visual Profiler

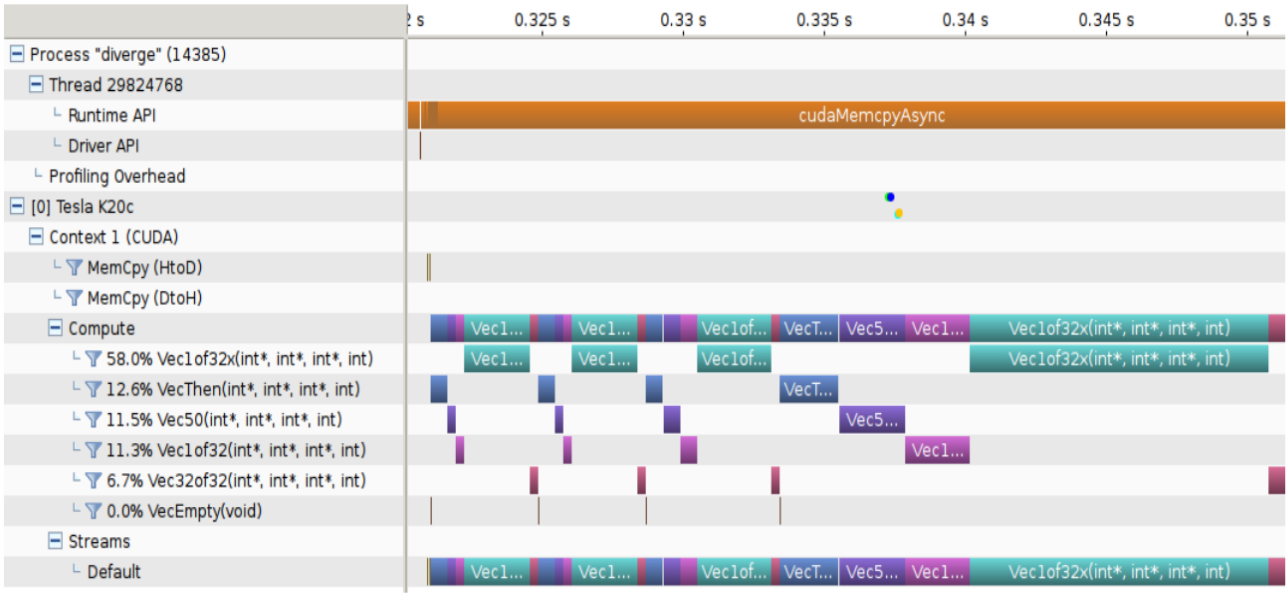
The NVIDIA Visual Profiler is a graphical profiling tool provided by NVIDIA as part of the CUDA Toolkit. It is designed to help developers analyse the performance of CUDA applications by providing insights into GPU utilization, memory usage, and execution timelines. The Visual Profiler assists in identifying performance bottlenecks and optimizing CUDA code for better efficiency.



Here are key features and functions of the NVIDIA Visual Profiler:

- **Timeline View:**

The Timeline View shows CPU and GPU activity that occurred while your application was being profiled. The following figure shows a Timeline View for a CUDA application.

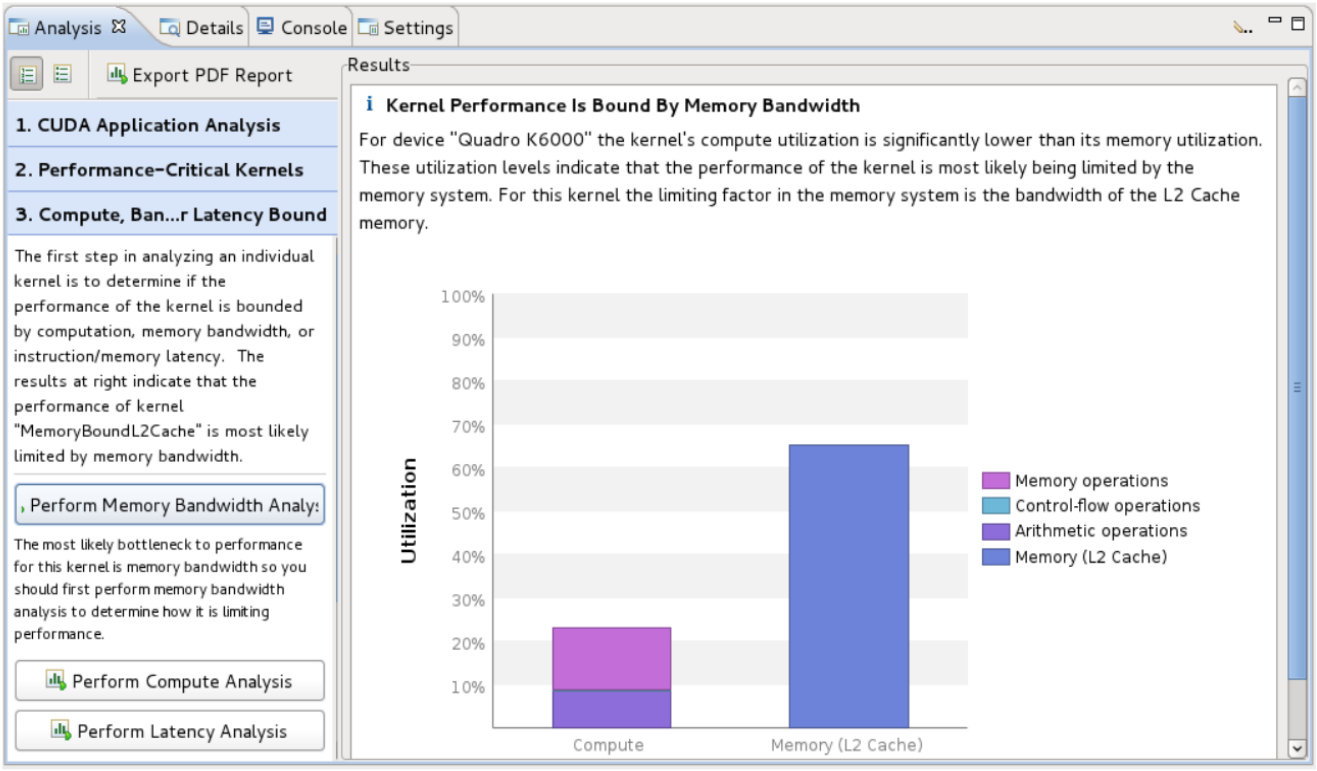


- **Analysis Results:**

The Analysis View is used to control application analysis and to display the analysis results. There are two analysis modes: guided and unguided.

In guided mode the analysis system will guide you through multiple analysis stages to help you understand the likely performance limiters and optimization opportunities in your application. In unguided mode you can manually explore all the analysis results collected for your application.

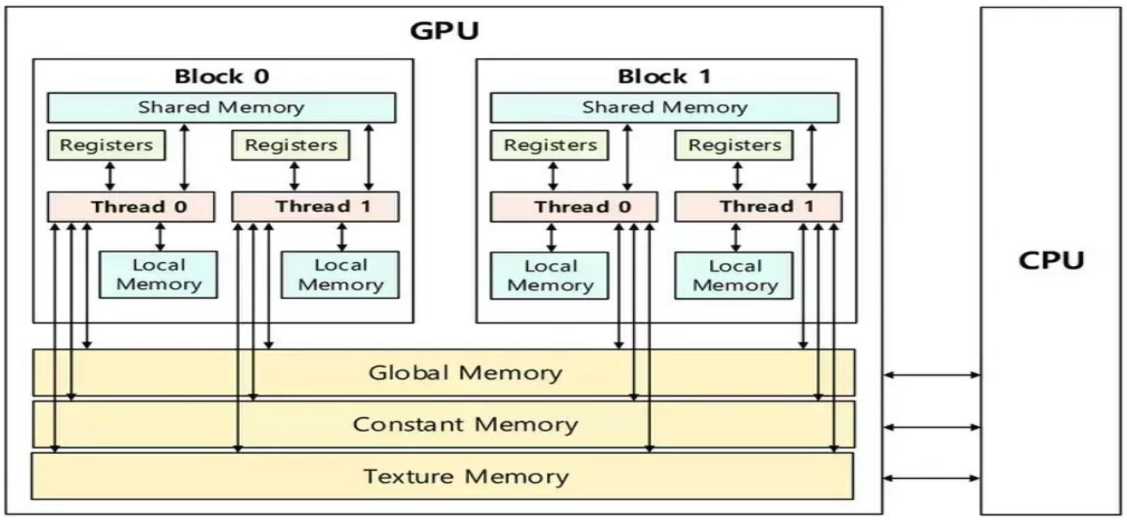
The following figure shows the analysis view in guided analysis mode. The left part of the view provides step-by-step directions to help you analyse and optimize your application. The right part of the view shows detailed analysis results appropriate for each part of the analysis.



CUDA Memories

The CPU and GPU have separate memory spaces. This means that data that is processed by the GPU must be moved from the CPU to the GPU before the computation starts, and the results of the computation must be moved back to the CPU once processing has completed. In CUDA programming, different types of memory are available to facilitate efficient data storage and access for parallel processing on NVIDIA GPUs.

The following diagram shows the different types of memory hierarchies that are present in the latest GPU architecture. Each memory may have a different size, latency, throughput, and visibility for the application developer.

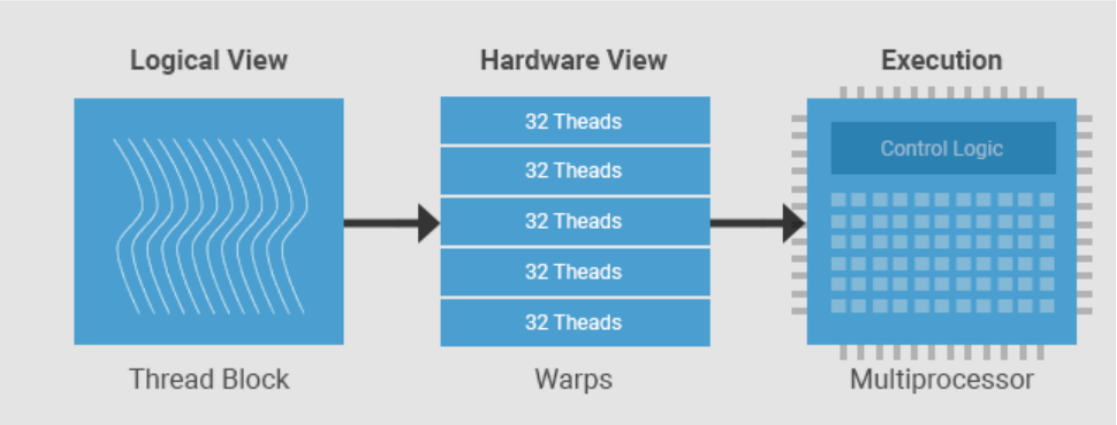


Global memory/device memory

In CUDA programming, global memory is a type of memory accessible by both the CPU (host) and the GPU (device). It serves as the primary memory space for storing data that needs to be shared between the host and the device. Global memory is used for larger data sets and is often employed for input data, output results, and intermediate computations in CUDA kernels. There's a large amount of global memory. It's slower to access than other memories like shared and registers. All running threads can read and write global memory and so can the CPU. The functions `cudaMalloc`, `cudaFree`, `cudaMemcpy` all deal with global memory. Global memory is allocated and deallocated by the host.

To effectively use global memory, we make use of warps in the CUDA programming model.The warp is a unit of thread scheduling/execution in Streaming Multiprocessor(SM). Once a block has been assigned to an SM, it is divided into a 32 -thread unit known as a warp. This is the basic execution unit in CUDA programming.

The following figure represents the relationship between the logical view and hardware view of a thread block, a warp and its mapping to an SM.



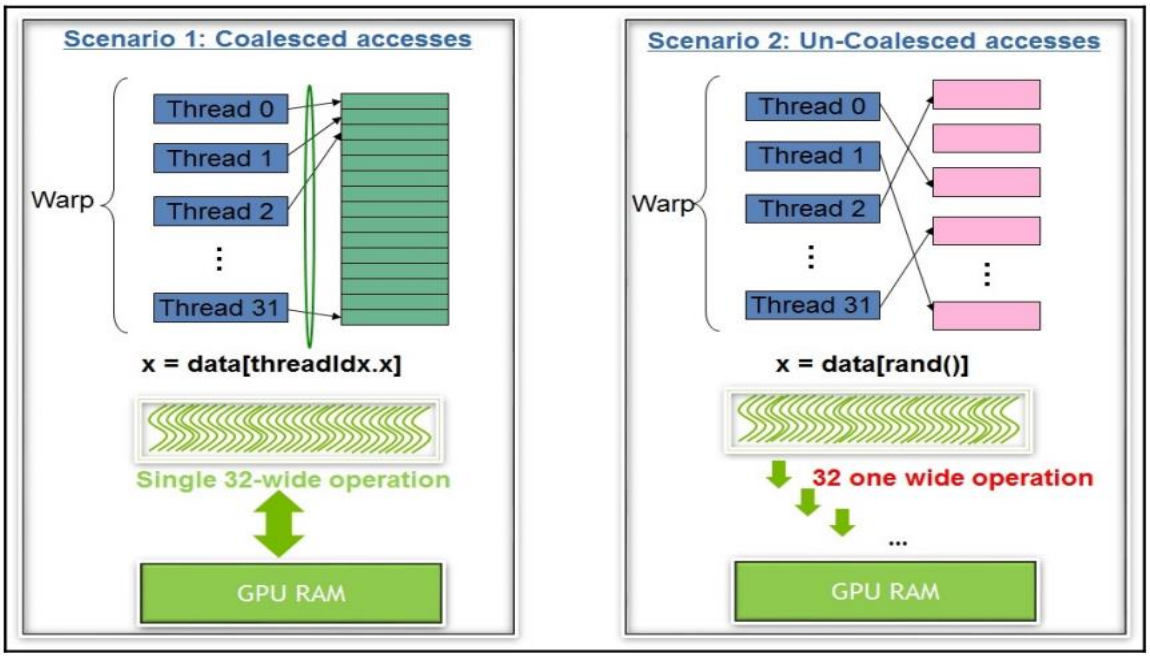
All of the threads in a warp execute the same instruction when selected. CUDA follows the Single Instruction, Multiple Thread (SIMT) model, that is, all threads in a warp fetch and execute the same instruction at one instance in time.

To optimally utilize access from global memory, the access should coalesce. coalesced global memory access is a key optimization strategy in CUDA programming, aiming to maximize memory bandwidth and minimize transfer overhead. Developers should carefully design their memory access patterns to ensure coalesced access, especially in memory-bound scenarios, for optimal GPU performance.

The difference between coalesced and uncoalesced is as follows:

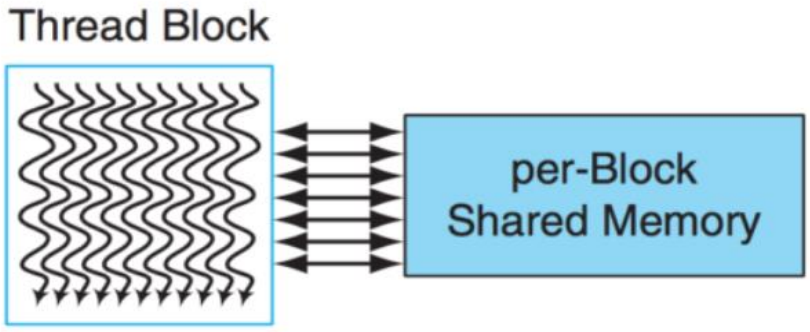
Coalesced access involves accessing consecutive memory locations by threads within a warp in a CUDA kernel.

Uncoalesced access occurs when threads within a warp access non-consecutive memory locations.

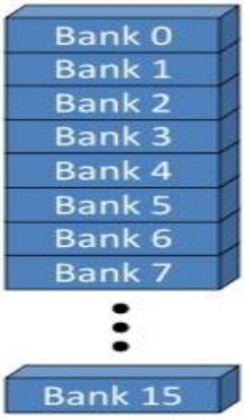


Shared memory

Shared memory in CUDA is a type of memory that is shared among threads within the same thread block during the execution of a CUDA kernel. It is a fast and low-latency memory space that allows threads to cooperatively share data. Shared memory is particularly useful for scenarios where multiple threads need to exchange information or collaborate on a computation.



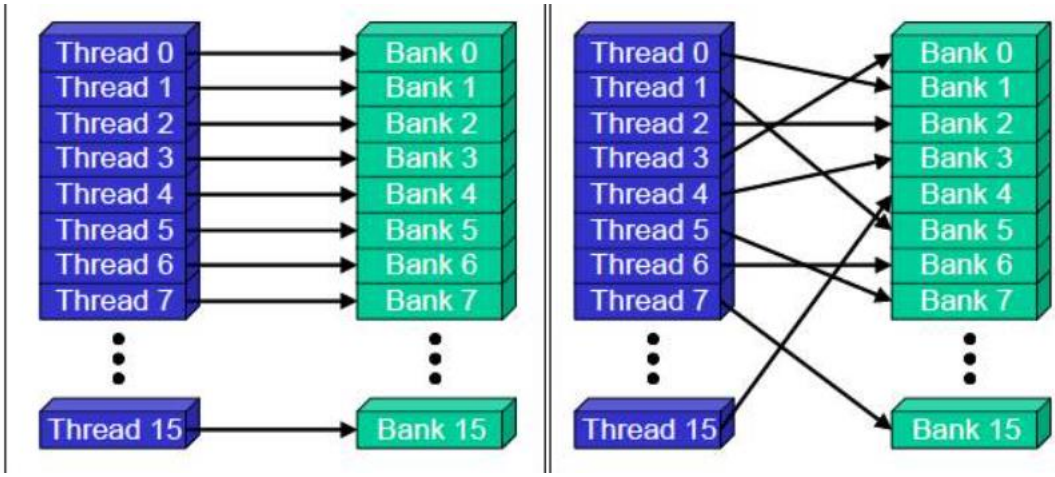
Shared memory is very fast (register speeds). Shared memory is used to enable fast communication between threads in a block. Shared memory only exists for the lifetime of the block. To achieve high memory bandwidth for concurrent accesses, shared memory is divided into **equally sized memory modules, called banks that can be accessed simultaneously**. Typically organized as a power of two (e.g., 16, 32, or 64 banks).



A bank conflict occurs when multiple threads in a warp access the same bank simultaneously, and at least one of the accesses is a write operation.

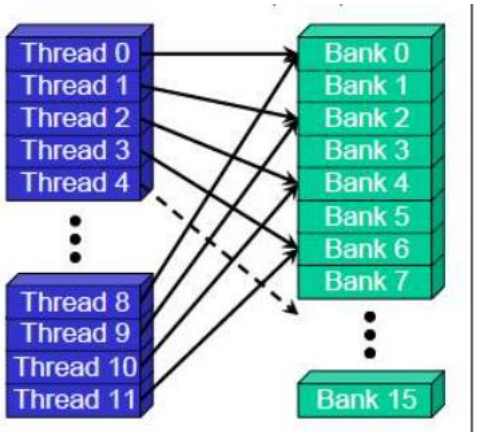
Bank conflicts can lead to serialized access, reducing the overall memory bandwidth and performance.

Example:



No Bank Conflicts

Bank Conflicts

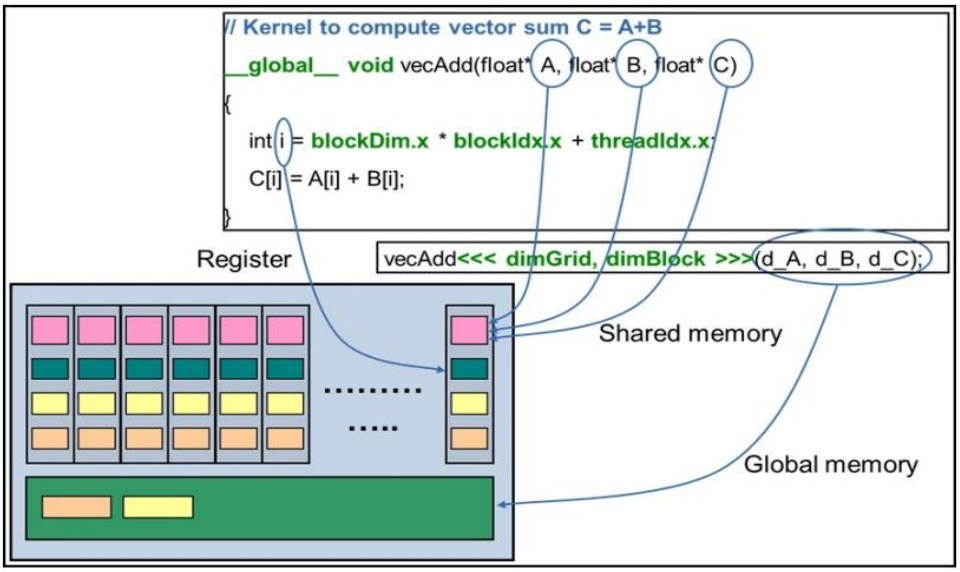


Registers

Registers are the fastest memory on the GPU. Register scope is per thread. Unlike the CPU, there's thousands of registers in a GPU. Carefully selecting a few registers can easily double the number of concurrent blocks the GPU can execute and therefore increase performance substantially.

Local variables that are declared as part of the kernel are stored in the registers. Intermediate values are also stored in registers. Every SM has a fixed set of registers. During compilation, a compiler (nvcc) tries to find the best number of registers per thread. In case the number of registers falls short, which generally happens when the CUDA kernel is large and has a lot of local variables and intermediate calculations, the data gets pushed to local memory, which may reside either in an L1/L2 cache or even lower in the memory hierarchy, such as global memory. This is also referred to as register spills.

A variable that's declared as part of the vecAdd kernel is stored in register memory. The arguments that are passed to the kernel, that is, A, B, and C, point to global memory, but the variable themselves are stored either in the shared memory of registers based on the GPU architecture. The following diagram shows the CUDA memory hierarchy and the default locations of the different variable types:



UNIT-IV

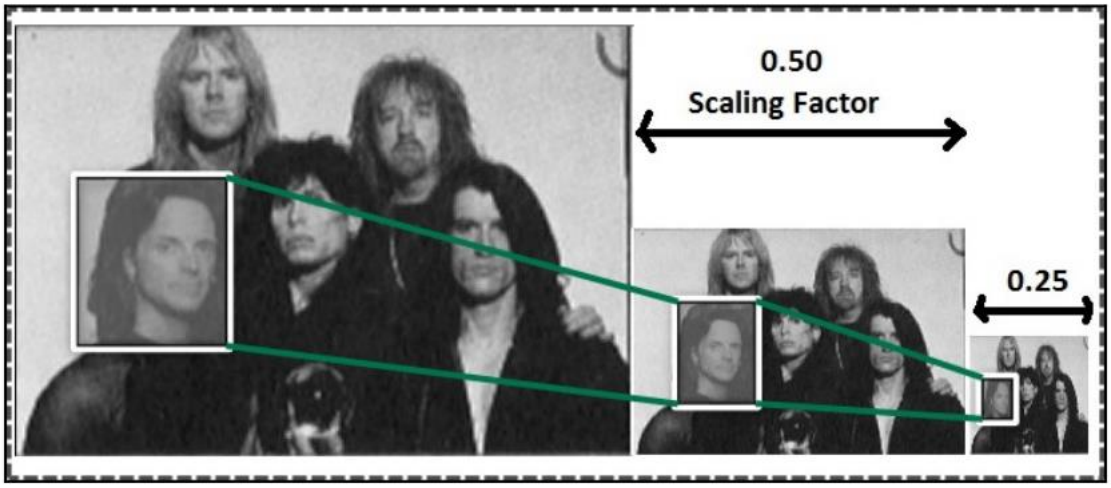
TOPIC 2: Read-only data/cache

In parallel programming, multiple threads or processes may access shared data concurrently, and managing the consistency of this shared data can be challenging. Caches play a crucial role in this scenario, as they store copies of data to reduce the need to access the slower main memory. A read-only cache can be beneficial in situations where certain data is known to be read-only and doesn't change during the parallel execution. By marking this data as read-only, developers can avoid the overhead of synchronization mechanisms, such as locks, that would otherwise be necessary to ensure the consistency of read-write data.

Read only cache is of two types. Constant memory and Texture Memory

In CUDA programming, constant memory is a type of GPU memory that is optimized for read-only access and is shared among all threads in a thread block. It is intended for storing data that remains constant throughout the execution of a kernel. Constant memory is cached and offers low-latency access, making it suitable for scenarios where multiple threads need to access the same read-only data.

Texture memory in CUDA is a type of memory that is optimized for 2D spatial locality, typically used for read-only access patterns. It is designed to provide efficient access to 2D or 3D arrays and is particularly useful for graphics-related operations in GPU programming. **Image scaling is an example to demonstrate the use of texture memory. An example of image scaling is shown in the following screenshot:**



Primarily, there are four steps that are required so that we can make use of texture memory:

1. Declare the texture memory.

In CUDA, you declare a texture using the **texture** qualifier. Specify the data type, texture type (1D, 2D, 3D), and read mode.

The important aspects of texture memory, which act like configurations and are set by the developer, are as follows:

Data Type: The data type stored in the texture (e.g., float, int)

Texture Type: cudaTextureType: The dimensionality of the texture:

- cudaTextureType1D for 1D textures.
- cudaTextureType2D for 2D textures.
- cudaTextureType3D for 3D textures.

Texture Read Mode:

- Specifies how the elements are read.
- Can be in NormalizedFloat (cudaReadModeNormalizedFloat): Data is converted to normalized floats (useful for integer textures). Normalized float mode expects the index within a range, [0.0, 1.0] for unsigned integers and [-1.0, 1.0] for signed integers.
- or ModeElement format. cudaReadModeElementType: Data is returned as-is in its native type.

Ex: texture<float, cudaTextureType2D, cudaReadModeElementType> myTexture;

2. **Bind the texture memory to a texture reference**.

Before you can use a texture in a CUDA kernel, you need to bind it to a CUDA array or linear memory. This associates the texture with the actual data in memory.

Ex: cudaBindTextureToArray(myTexture, myArray);

3. Read the texture memory using a texture reference in the CUDA kernel.

Accessing texture memory in a CUDA kernel is done using functions like **tex1Dfetch**, **tex2D**, **tex3D**, depending on the dimensionality of the texture.These functions automatically perform the necessary interpolation based on the texture coordinates.

```
Ex : __global__ void myKernel() {
   float value = tex2D(myTexture, x, y);
   // ... rest of the kernel code
}
```

4. Unbind the texture memory from your texture reference.

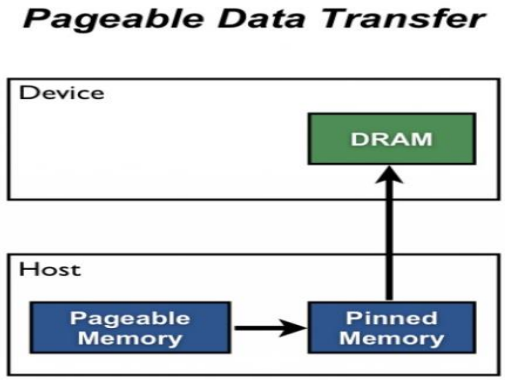
After the kernel execution or when you're done using the texture, it's good practice to unbind the texture.

Ex : cudaUnbindTexture(myTexture);

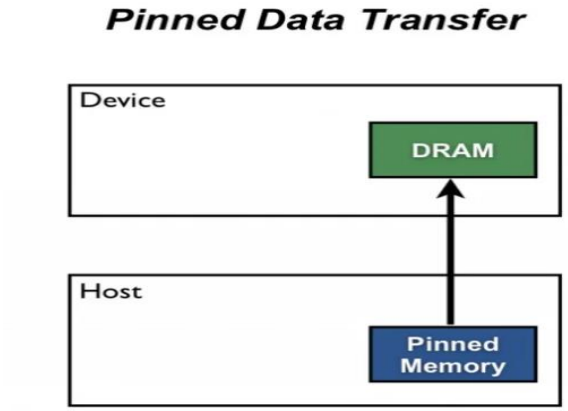
Pinned memory

Pinned memory, also known as page-locked memory or locked memory, refers to a type of memory in which the operating system prevents the pages from being swapped out to disk.

Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or “pinned”, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory, as illustrated below.



As you can see in the figure, pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory. Allocate pinned host memory in CUDA C/C++ using cudaMallocHost() or cudaHostAlloc(), and deallocate it with cudaFreeHost().



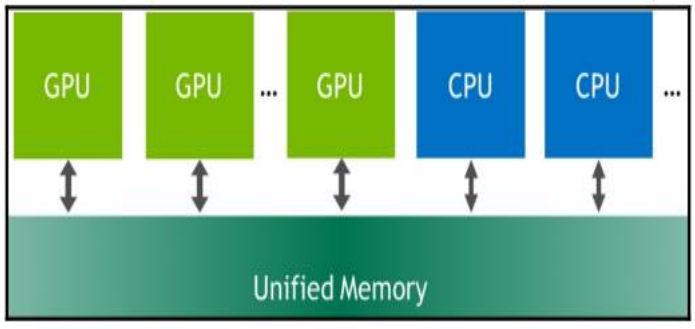
In the context of GPU programming, particularly with CUDA (Compute Unified Device Architecture) developed by NVIDIA, pinned memory is often used to enhance data transfer between the CPU (host) and the GPU (device). Pinned memory allows for asynchronous data transfers between the CPU and GPU, meaning that data can be transferred in the background while the CPU or GPU is performing other tasks. Pinned memory is particularly beneficial when dealing with large data transfers between the host and the GPU.

It is commonly used in applications like real-time video processing, simulations, and other scenarios where low-latency data transfer is critical.

Unified memory

With every new CUDA and GPU architecture release, new features are added. One such important feature that was released from CUDA 6.0 onwards is unified memory from the Kepler GPU architecture.

Unified Memory is a memory management feature introduced by NVIDIA in CUDA to simplify the memory management between the CPU and GPU in heterogeneous computing environments. Unified Memory allows both the CPU and GPU to access a single, unified view of the memory, eliminating the need for explicit data transfers between the host (CPU) and the device (GPU). In simpler words, UM provides the user with a view of single memory space that's accessible by all GPUs and CPUs in the system. This is illustrated in the following diagram:

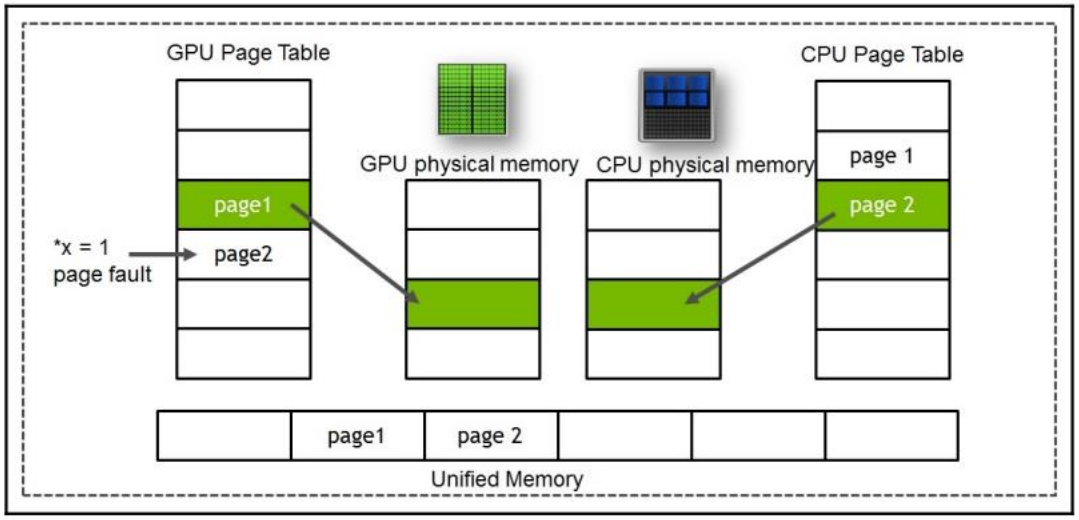


`cudaMallocManaged()` is a CUDA API function used for allocating managed memory in Unified Memory. cudaMallocManaged() does not allocate physical memory but allocates memory based on a first-touch basis. If the GPU first touches the variable, the page will be allocated and mapped in the GPU page table; otherwise, if the CPU first touches the variable, it will be allocated and mapped to the CPU. Unified Memory utilizes a page faulting mechanism. If a GPU attempts to access data that is

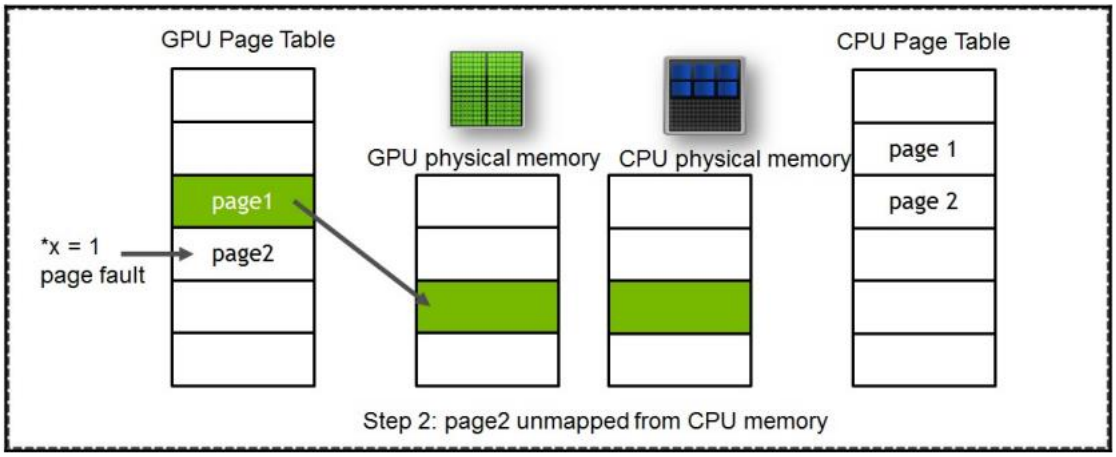
currently resident on the CPU or vice versa, a page fault is triggered, leading to the data being migrated to the appropriate memory space.

The sequence of operations that are completed in a page migration are as follows

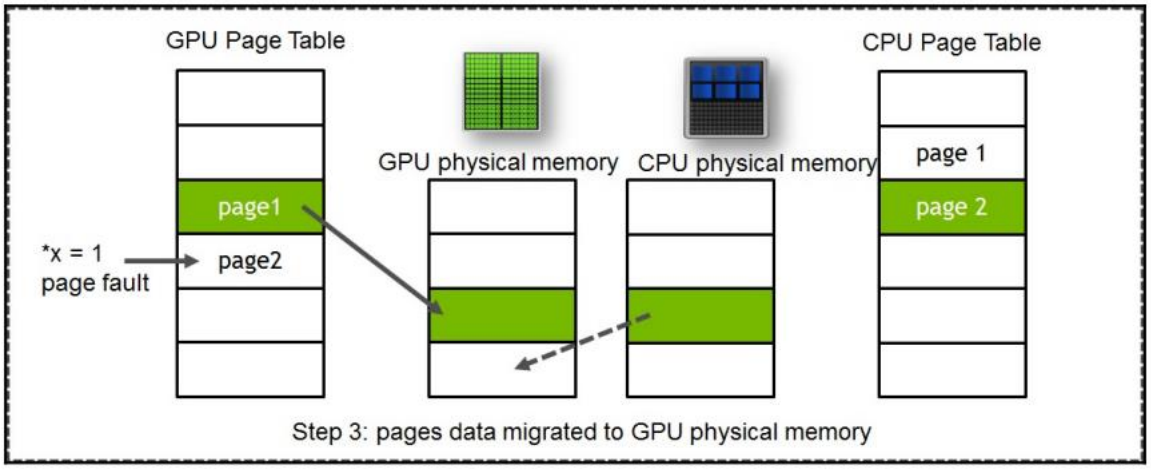
First, we need to allocate new pages on the GPU and CPU (first-touch basis). If the page is not present and mapped to another, a device page table page fault occurs. When *x, which resides in page 2, is accessed in the GPU that is currently mapped to CPU memory, it gets a page fault. Take a look at the following diagram



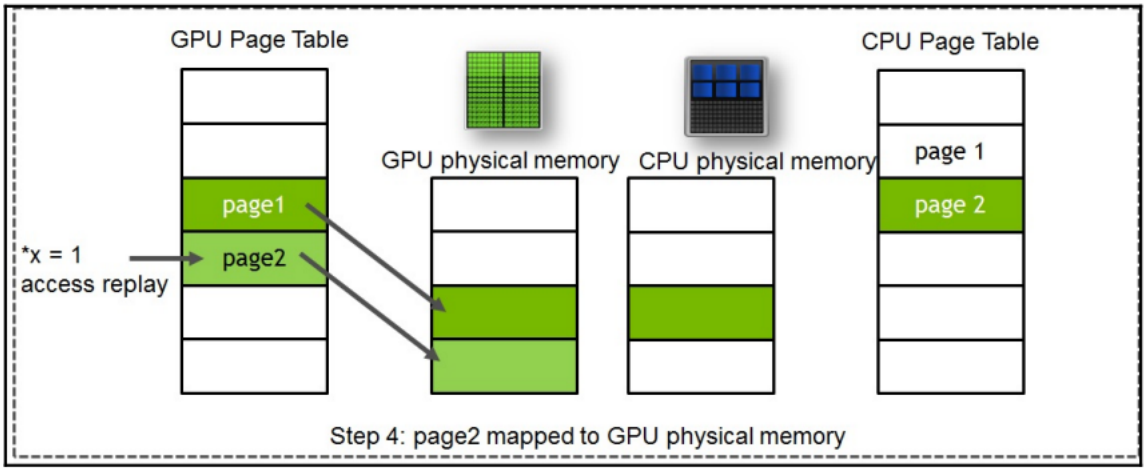
In the next step, the old page on the CPU is unmapped, as shown in the following diagram:



Next, the data is copied from the CPU to the GPU, as shown in the following diagram



Finally, the new pages are mapped on the GPU, while the old pages are freed on the CPU, as shown in the following diagram:



The Translation Lookaside Buffer (TLB) in GPU, much like in the CPU, performs address translation from the physical address to the virtual address. When a page fault occurs, the TLB for the respective SM is locked. This basically means that the new instructions will be stalled until the time the preceding steps are performed and finally unlock the TLB. This is necessary to maintain coherency and maintain a consistent state of memory view within an SM

GPU architectures have evolved over time and memory architectures have changed considerably. If we take a look at the last four generations, there are some common patterns which emerge, some of which are as follows: The memory capacity, in general, has increased in levels. The memory bandwidth and capacity have increased with new generation architectures. The following table shows the properties for the last four generations.

Memory type	Properties	Volta V100	Pascal P100	Maxwell M60	Kepler K80
Register	Size per SM	256 KB	256 KB	256 KB	256 KB
L1	Size	32...128 KiB	24 KiB	24 KiB	16...48 KiB
	Line size	32	32 B	32 B	128 B
L2	Size	6144 KiB	4,096 KiB	2,048 KiB	1,536 Kib
	Line size	64 B	32B	32B	32B
Shared memory	Size per SMX	Up to 96 KiB	64 KiB	64 KiB	48 KiB
	Size per GPU	up to 7,689 KiB	3,584 KiB	1,536 KiB	624 KiB
	Theoretical bandwidth	13,800 GiB/s	9,519 GiB/s	2,410 GiB/s	2,912 GiB/s
Global memory	Memory bus	HBM2	HBM2	GDDR5	GDDR5
	Size	32,152 MiB	16,276 MiB	8,155 MiB	12,237 MiB
	Theoretical bandwidth	900 GiB/s	732 GiB/s	160 GiB/s	240 GiB/s

In general, the preceding observations have helped CUDA applications to run faster with the newer architectures. But in parallel, some fundamental changes were also brought to the CUDA programming model, as well as the memory architecture, to make life easy for CUDA programmers. One such change we observed was for texture memory where, prior to CUDA 5.0, the developer had to manually bind and unbind the textures and had to be declared globally. With CUDA 5.0, it was not necessary to do so. It also removed the restrictions on the number of textures references a developer could have in an application.

In this evolution process, it is also important to understand that CPU and GPU caches are very different and serve a different purpose. As part of the CUDA architecture, we usually launch hundreds to thousands of threads per SM. Tens of thousands of threads share the L2 cache. So, L1 and L2 are small per thread. For example, at 2,048 threads/SM with 80 SM, each thread gets only 64 bytes at L1 and 38 Bytes at L2 per thread. Caches in GPU cache common data that's accessed by many threads. This is sometimes referred to as spatial locality. A typical example of this is when accesses by threads are unaligned and irregular. The GPU cache can help to reduce the effect of register spills and local memory since the CPU cache is primarily for temporal locality. Temporal locality refers to the tendency of a program to access the same memory locations repeatedly over a short period. Spatial locality, on the other hand, refers to the tendency to access nearby memory locations.

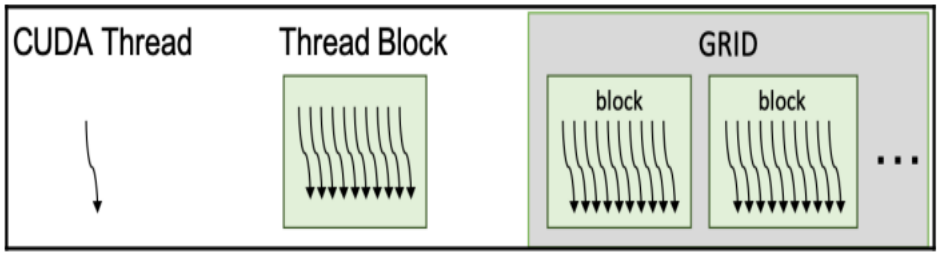
UNIT-IV

Topic 3: CUDA Threads & Occupancy

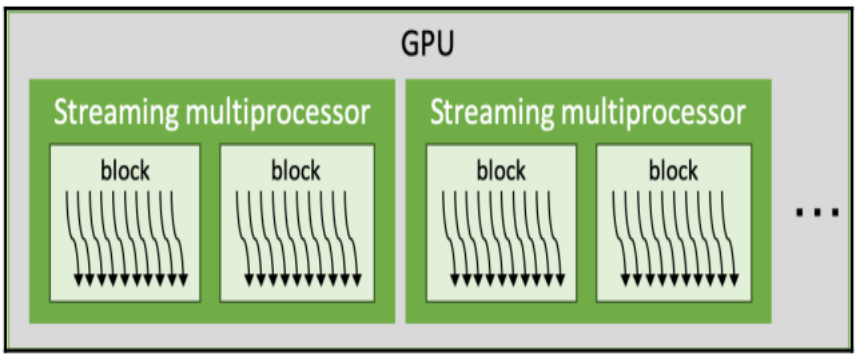
CUDA threads, blocks, and the GPU

The basic working unit in CUDA programming is the CUDA thread. The basic CUDA thread execution model is Single Instruction and Multiple Thread (SIMT). Conceptually, multiple CUDA threads work in parallel in a group. CUDA thread blocks are collections of multiple CUDA threads. Multiple thread blocks operate concurrently with each other. We call a group of thread blocks a grid.

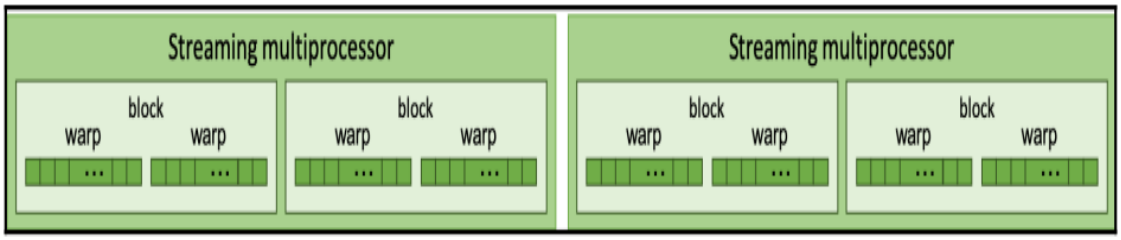
The following diagram shows their relationships:



When we launch a CUDA kernel, one or multiple CUDA thread blocks execute on each streaming multiprocessor in the GPU. Also, a streaming multiprocessor can run multiple thread blocks depending on resource availability. The number of streaming multiprocessors varies depending on the GPU specification. For instance, it is 80 for a Tesla V100, and it is 48 for an RTX 2080 (Ti). The number of threads in a thread block varies, and the number of blocks in a grid does too:



The CUDA streaming multiprocessor controls CUDA threads in terms of Warp. Warp in CUDA refers to a group of threads that execute the same instruction simultaneously. A warp typically consists of 32 threads on most NVIDIA GPUs. One or multiple warps configures a CUDA thread block. The following figure shows the relationship



The output of threads within a warp is not guaranteed to follow a specific order, and the execution of different warps can be interleaved and happen out of order.

CUDA occupancy

CUDA occupancy refers to the measure of how well a GPU is utilized by a specific kernel launch. It quantifies the number of active warps in relation to the total number of warps that a GPU can potentially support. Occupancy is typically expressed as a percentage and is calculated using the following formula:

Occupancy=Active Warps / Maximum Warps

- Active Warps are the number of warps that are actively executing instructions.
- Maximum Warps are the total number of warps that the GPU can accommodate.

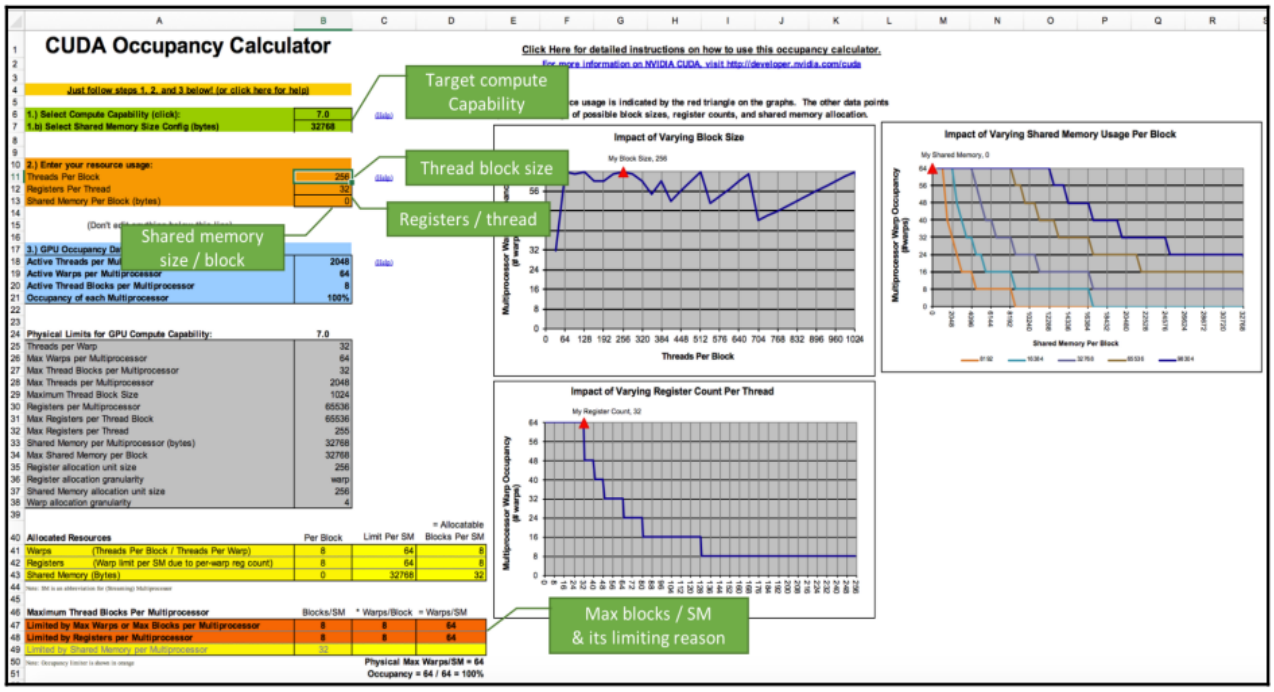
Developers can determine CUDA occupancy using two methods:

Theoretical occupancy determined by the CUDA Occupancy Calculator: This calculator is an Excel sheet provided with the CUDA Toolkit. We can determine each kernel's occupancy theoretically from the kernel resource usages and the GPU's streaming multiprocessor. Theoretical occupancy can be regarded as the maximum upper-bound occupancy because the occupancy number does not consider instructional dependencies or memory bandwidth limitations.

Achieved occupancy determined by the GPU: Achieved occupancy is a measure of how well a kernel is utilizing the available resources on the GPU during actual execution. Unlike theoretical occupancy, which is a calculated value based on hardware specifications and kernel configuration, achieved occupancy is determined through profiling tools and performance analysis during runtime. The achieved occupancy reflects the true number of concurrent executed warps on a streaming multiprocessor and the maximum available warps.

Theoretical occupancy determined by the CUDA Occupancy Calculator:

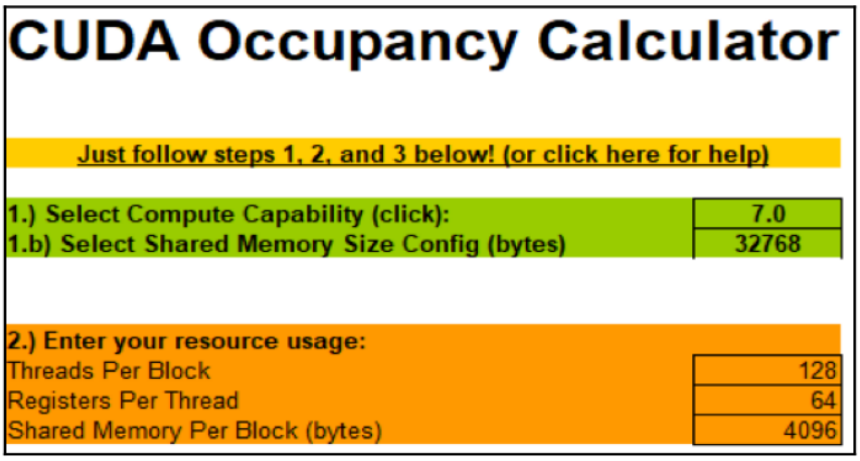
The following is a screenshot of the calculator



This calculator has two parts: **kernel information inputs and occupancy information outputs**.

As input, it requires two kinds of information, as follows:

- **The GPU's compute capability (green)**
- **Thread block resource information (yellow):**
  - Threads per CUDA thread block
  - Registers per CUDA thread
  - Shared memory per bloc



The calculator shows the GPU's occupancy information here:

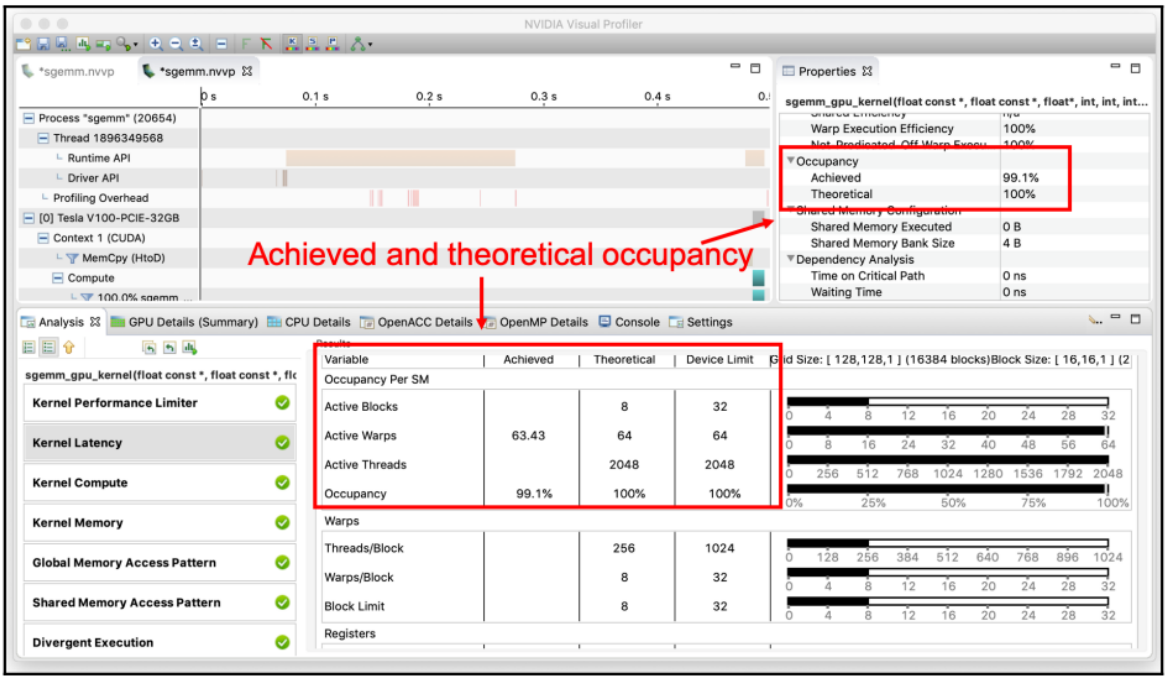
- GPU occupancy data (blue)
- The GPU's physical limitation for GPU compute capability (gray)

- Allocated resources per block (yellow)
- Maximum thread blocks per stream multiprocessor (yellow, orange, and red)
- Occupancy limit graph following three key occupancy resources, which are threads, registers, and shared memory per block
- Red triangles on graphs, which show the current occupancy data

Occupancy tuning in CUDA involves optimizing the number of active warps on a multiprocessor (SM) to achieve better utilization of GPU resources. One important aspect of occupancy tuning is managing the usage of registers per thread, as the number of registers can impact the number of warps that can be accommodated on an SM. Each thread in a CUDA kernel is allocated a certain number of registers for storing variables and intermediate values.The more registers a thread uses, the fewer threads can be accommodated on a multiprocessor, which can reduce occupancy. Strategies include minimizing the use of complex data types, reducing the use of local variables, and avoiding unnecessary register spills.

Getting the achieved occupancy from the profiler

We can obtain the achieved occupancy from the profiled metric data using the Visual Profiler. Click the target kernel timeline bar. Then, we can see the theoretical and achieved occupancy in the Properties panel.

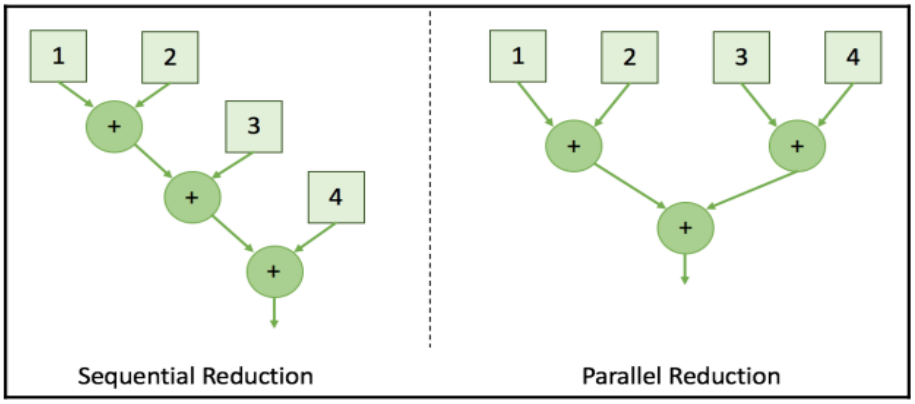


Unit-4

Topic 4: Understanding Parallel reduction

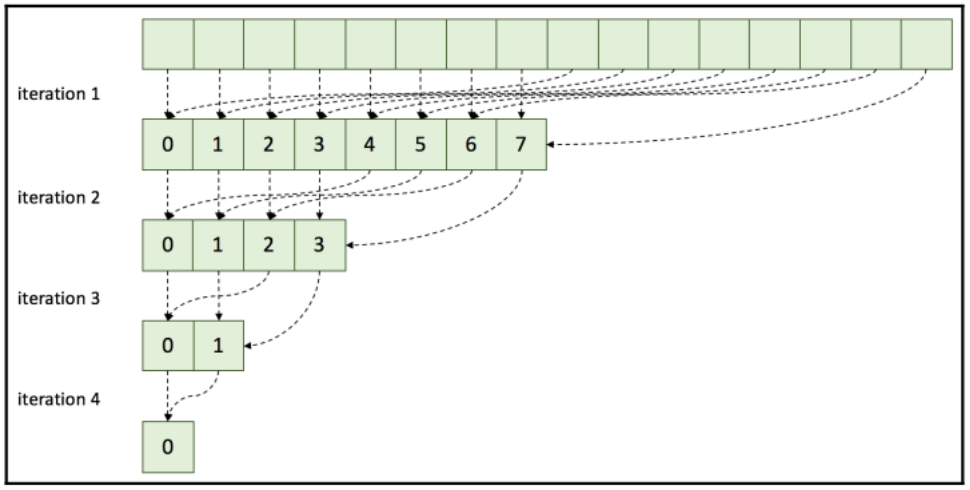
Parallel reduction is a common parallel computing technique used to efficiently compute the sum (or other associative operations) of a large set of values. In CUDA, parallel reduction is often employed to take advantage of the parallel processing capabilities of GPUs. The basic idea is to break down the problem into smaller pieces that can be computed concurrently, reducing the overall computation time. This task can be done in sequence or in parallel.

The following diagram shows the difference between sequential reduction and parallel reduction:



Parallel reduction can be implemented using two approaches

- Naive parallel reduction using global memory

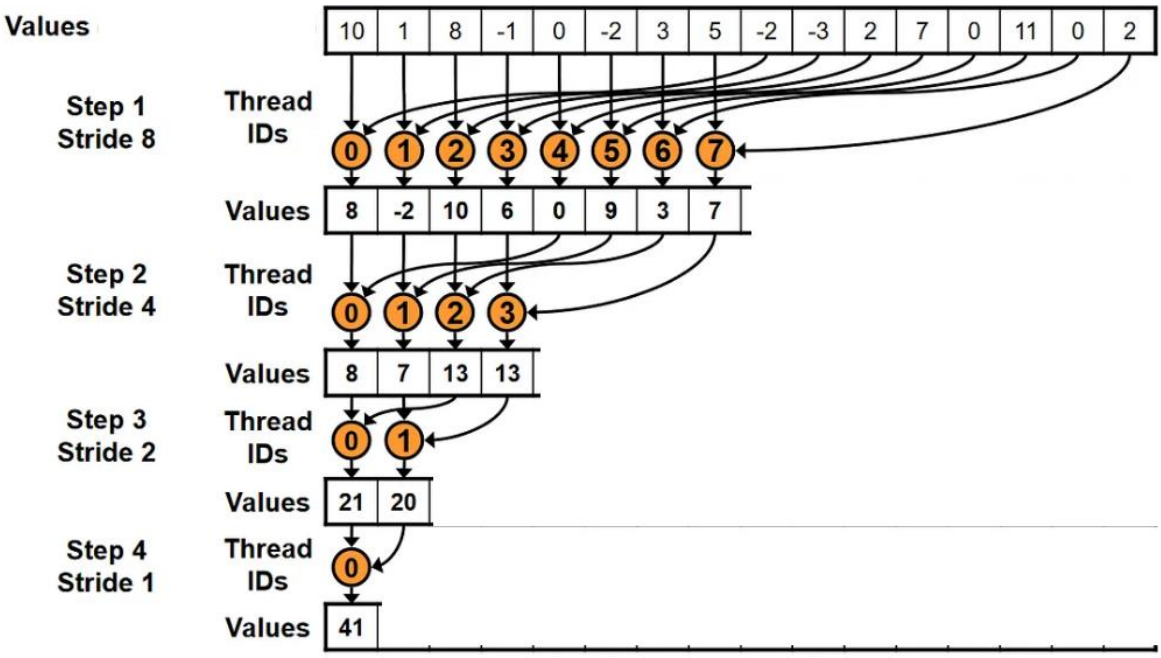


The stride for a vector is an increment that is used to step through array storage to select the vector elements from an array.

Steps:

1. **Input Loading**:
    - Each thread processes an element from the input array stored in global memory.
2. **Reduction**:
    - Threads combine data in successive iterations. For example, a sum reduction would involve adding pairs of elements.
    - At each step, the size of the problem is halved (e.g., 1024 → 512 → 256 → ...).
3. **Storage**:
    - Intermediate results are written back to global memory after each step.
4. **Final Output**:
    - The final result is stored in global memory.

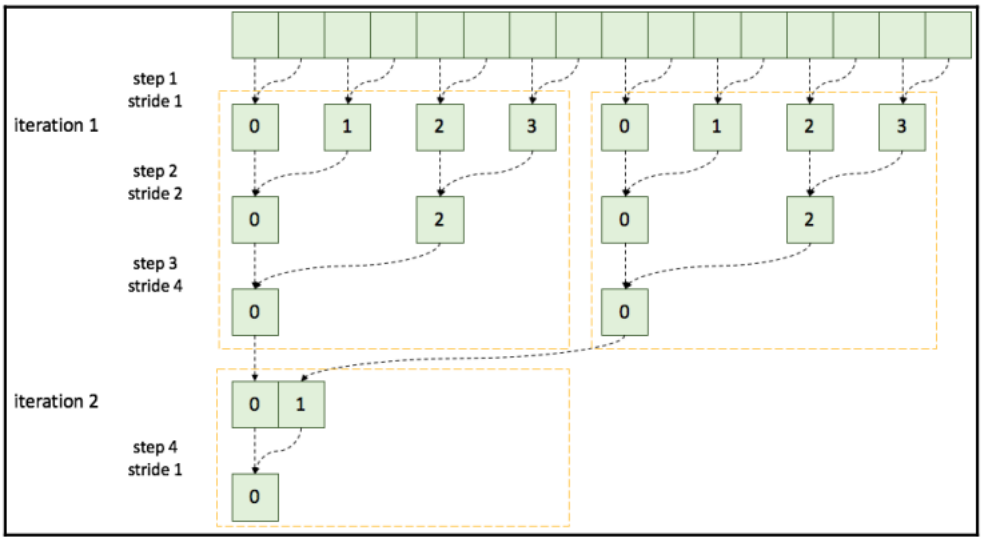
Example



This approach is slow in CUDA because it wastes the global memory's bandwidth and does not utilize any faster on-chip memory. For better performance, it is recommended to use shared memory to save global memory bandwidth and reduce memory-fetch latency.

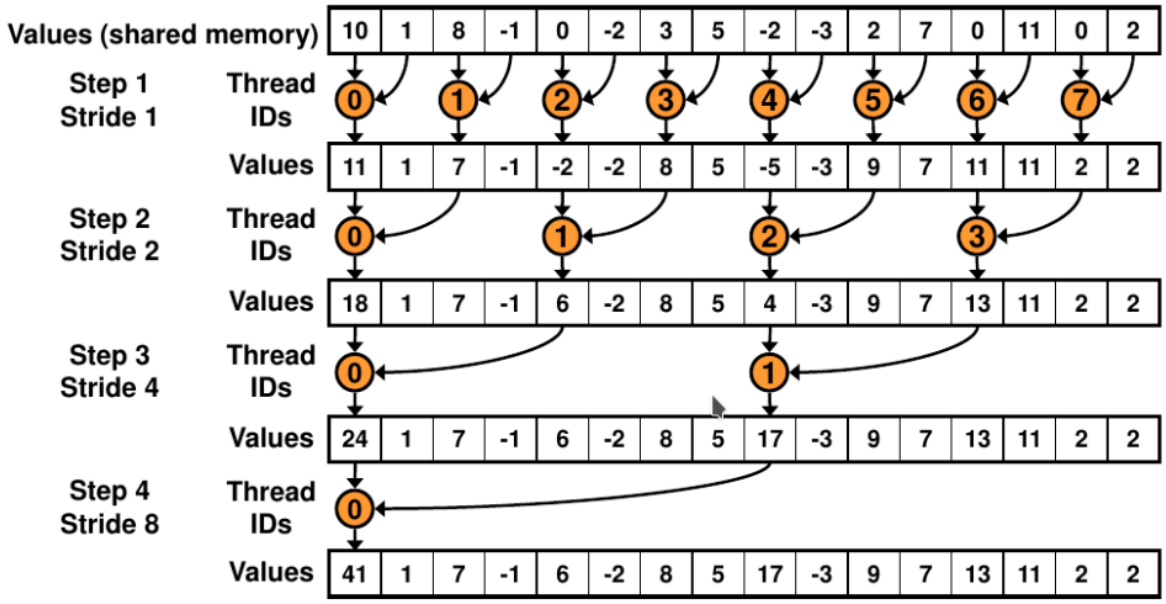
- **Reducing kernels using shared memory**

In this reduction, each CUDA thread block reduces input values, and the CUDA threads share data using shared memory. Each iteration operates on the previous reduction result. Its design is shown in the following diagram, which displays parallel reduction using shared memory.



Steps:

1. Load to Shared Memory:
    - Each thread loads its data from global memory into shared memory.
2. Reduction in Shared Memory:
    - Threads perform the reduction step within shared memory, minimizing global memory access.
3. Write Back:
    - The final result is written back to global memory.



The yellow-dotted boxes represent a CUDA thread block's operation coverage. In this design, each CUDA thread block outputs one reduction result. Block-level reduction lets each CUDA thread block conduct reduction and outputs a single reduction output. Since it does not require us to save the intermediate result in the global memory, the CUDA kernel can store the transitional value in the shared memory. This design helps to save global memory bandwidth and reduce memory latency.

Performance comparison for the two reductions – global and shared memory

Global Memory:

- Simple to implement.
- Requires more global memory transactions, which can be a performance bottleneck.
- Limited by the bandwidth of the global memory.

Shared Memory:

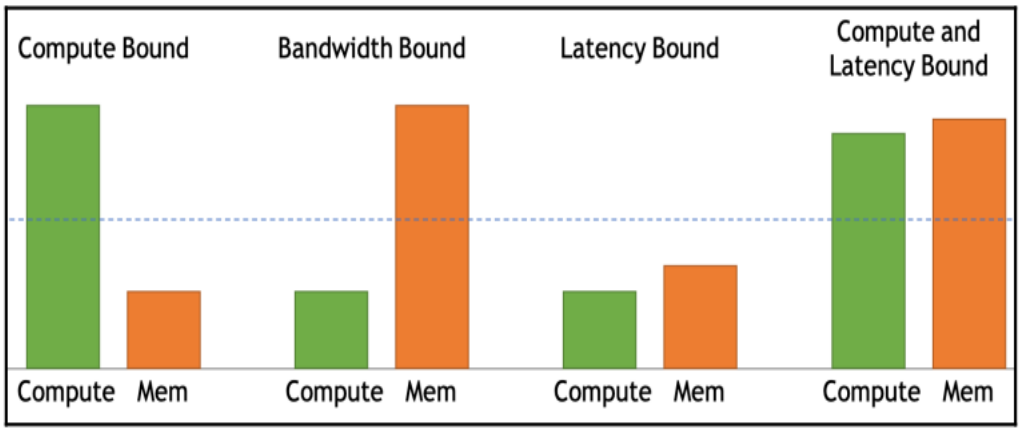
- Utilizes fast shared memory for intermediate results, reducing global memory transactions.
- More complex due to the need for synchronization and a second-level reduction.
- Can be more efficient for large datasets due to reduced memory traffic.

The choice between global and shared memory depends on the size of the dataset, the specific GPU architecture, and the nature of the reduction operation.

Identifying the application's performance limiter

The performance limiter shows the bounding factor, which limits the performance of an application most significantly. Identifying performance limiters in CUDA applications involves analyzing various aspects of your code, hardware, and the interactions between them. Based on its profiling information, it analyzes performance limiting factors among computing and memory bandwidth.

Based on these resources' utilization, an application can be categorized into four types: Compute Bound, Bandwidth Bound, Latency Bound, and Compute and Latency Bound. The following graph shows these categories related to compute and memory utilization:



Compute Bound: A program or application is considered "compute-bound" if the program spends a significant amount of time performing calculations and computations rather than waiting for data to be fetched from memory or performing other I/O operations. The goal in optimizing a compute-bound program is to maximize the utilization of the computational resources available.

Bandwidth-bound: A program or application is considered "bandwidth-bound" when its performance is primarily limited by the rate at which data can be transferred between different components, such as between the CPU and RAM or between the CPU/GPU and global memory on a GPU. In a bandwidth-bound scenario, the processing units (CPU or GPU) spend a significant amount of time waiting for data to be fetched from or written to memory.

Latency Bound: A program or application is considered "latency-bound" when its performance is primarily limited by the time it takes to complete individual operations or tasks. Latency-bound scenarios often involve minimizing the response time or completion time for critical operations.

When dealing with a scenario where an application is both compute and latency bound, the need arises to strike a balance between maximizing computational throughput and minimizing the time it takes for critical operations to complete.

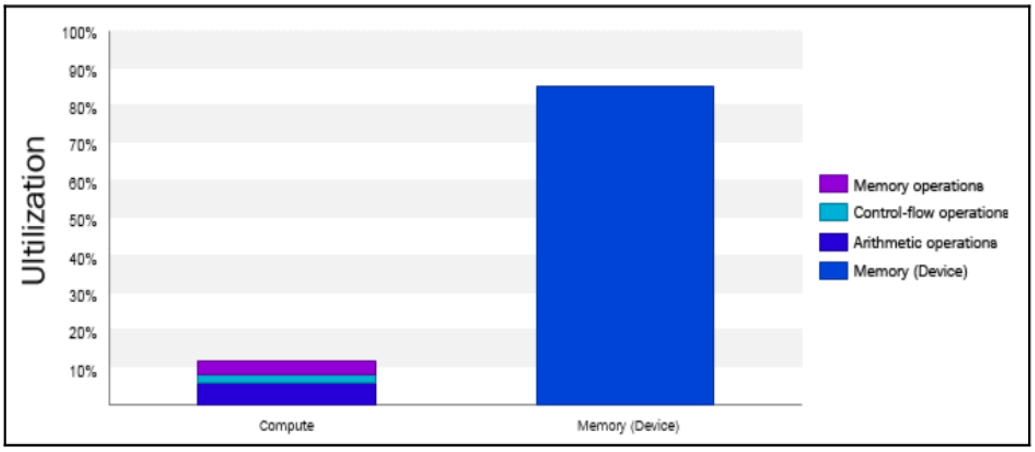
Simulation and Analysis: Compute-bound aspects are crucial for simulations and data analysis.

Real-time Response: Latency-bound aspects are essential for applications requiring real-time responses, such as control systems or interactive user interfaces.

After we identify the limiter, we can use the next optimization strategy. If either resource's utilization is high, we can focus on the optimization of the resource. If both are under utilized, we can apply latency optimization from I/O aspects of the system. If both are high, we can investigate whether there is a memory operation stalling issue and computing related issue.

Finding the performance limiter and optimization

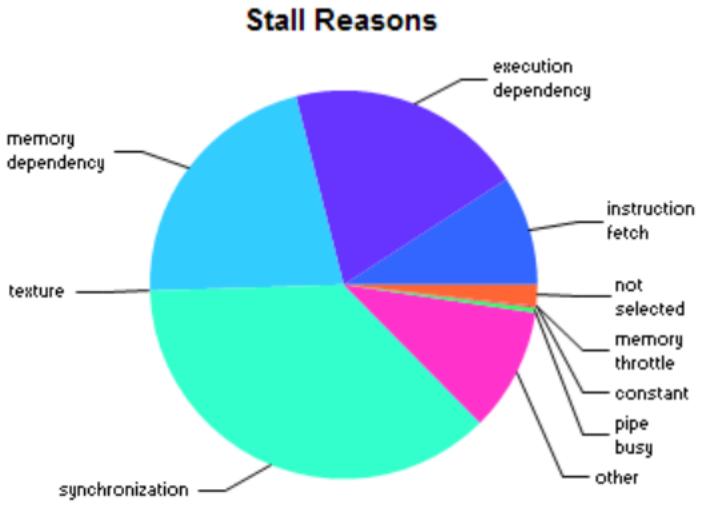
The following chart from NVIDIA profiler, shows the global memory-based reduction's performance limiter:



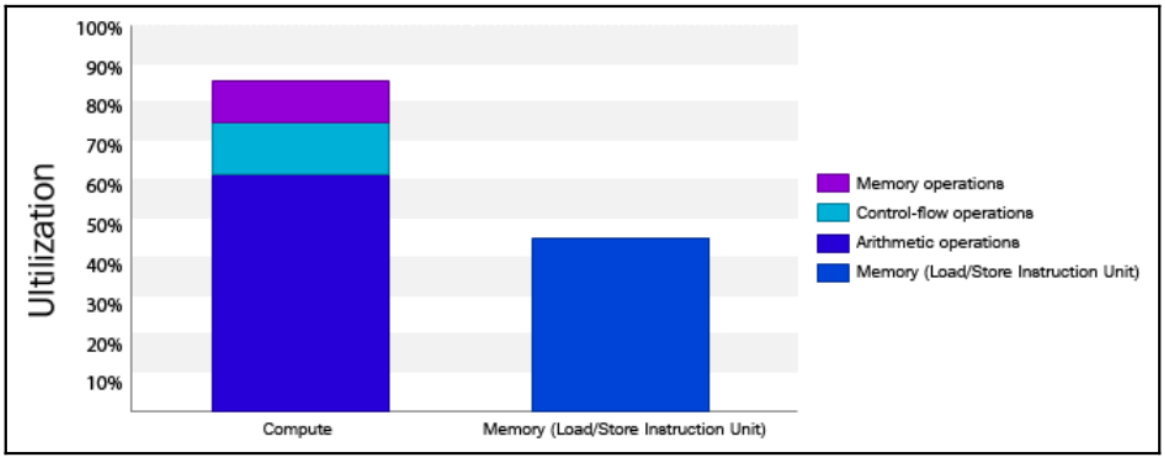
As you can see in the preceding chart, the utilization gap between Compute and Memory is large and this could mean there will be a lot of latency in compute due to memory bottleneck. This is because of Memory Stall.

A "memory stall" refers to a situation where a processor or a computational unit is waiting for data to be fetched from memory, and during this waiting period, the processor is effectively idle. Memory stalls can significantly impact the overall performance of a program, leading to lower efficiency and throughput. Memory stalls can be caused by various factors, and addressing them is crucial for

optimizing the performance of an application.



Then, we will obtain the following chart, which shows the second shared memory-based reduction's performance limiter: We can determine that it is compute-bounded and memory does not starve the CUDA cores

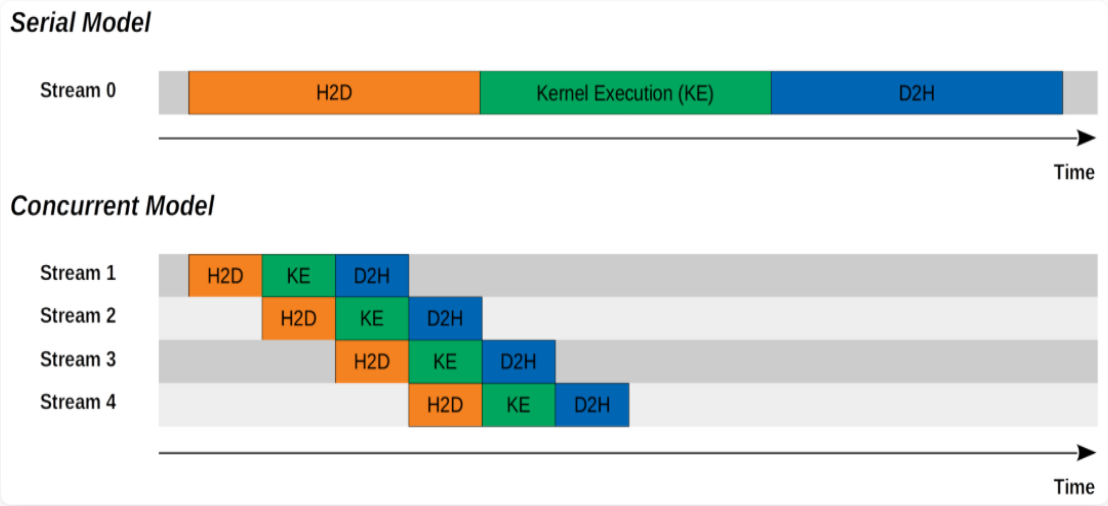


UNIT-V

TOPIC 1: CUDA Streams & Pipelining in GPU

Kernel execution with CUDA streams

CUDA streams are a fundamental concept in NVIDIA's CUDA programming model. CUDA streams are sequences of operations (such as kernel launches, memory transfers, and other tasks) that are executed on a GPU in the order they are issued. By default, all operations in CUDA are executed in a single, default stream, ensuring serial execution.They provide a way to overlap the execution of multiple operations on a GPU, enabling concurrent processing and improved performance. CUDA streams allow developers to organize and manage asynchronous execution of tasks, such as kernel launches and memory transfers, on the GPU.



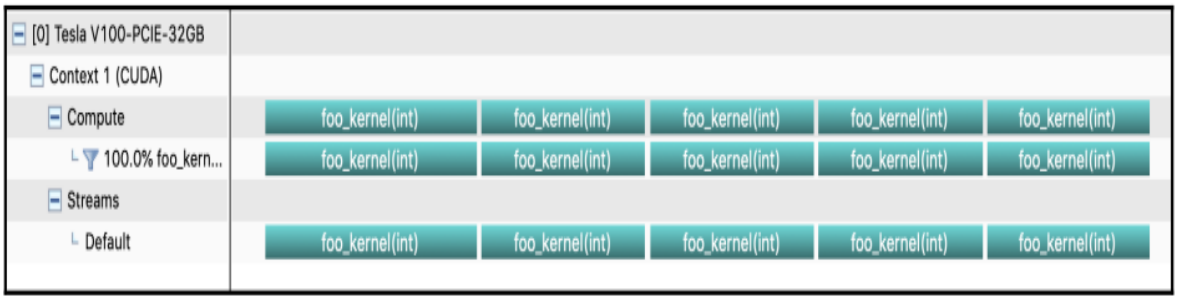
The usage of CUDA streams

Default Stream

First, let's write an application that uses the default CUDA stream, as follows:

```
__global__ void foo_kernel(int step)
{
printf("loop: %d\n", step);
}
int main()
{
for (int i = 0; i < 5; i++) // CUDA kernel call with the default stream
foo_kernel<<< 1, 1, 0, 0 >>>(i);
cudaDeviceSynchronize();
return 0;
}
```

As you can see in the code, we call the kernel function with the stream ID as 0, because the identification value of the default stream is 0



Multiple CUDA Streams

- Streams are created using the **cudaStreamCreate** function

  cudaStream_t stream;

  cudaStreamCreate(&stream);

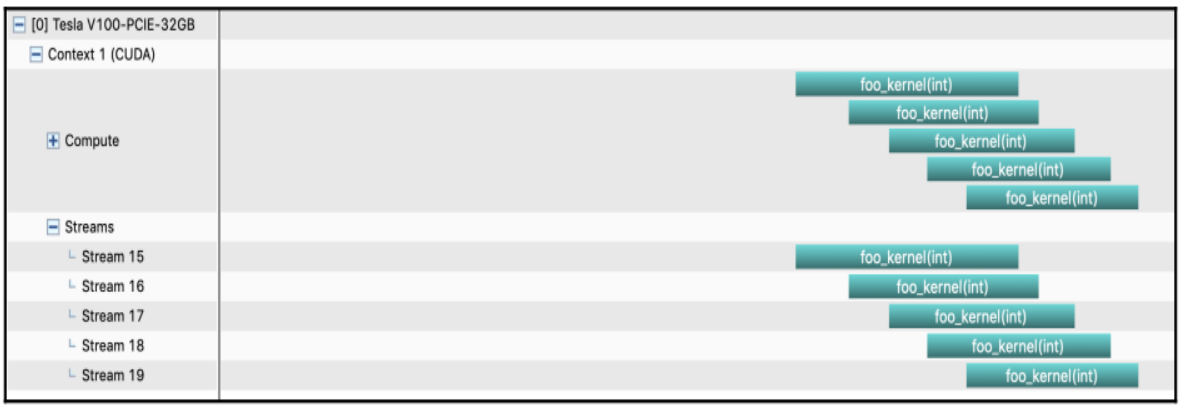
- Kernel Launch with Streams: When launching a kernel, you can specify the stream in which the kernel should execute.

  kernelName<<<gridDim, blockDim, sharedMem, stream>>>(arg1, arg2, ...);

  kernelName: The name of the kernel function you want to launch.
  <<<gridDim, blockDim, sharedMem, stream>>>:
  gridDim: The dimensions of the grid. It can be a 1D, 2D, or 3D structure.
  blockDim: The dimensions of each block within the grid. Similar to gridDim, it can be 1D, 2D, or 3D.
  sharedMem: An integer specifying the amount of dynamic shared memory to be allocated per block in bytes. This is optional and can be omitted if your kernel does not use shared memory.
  stream: A CUDA stream in which the kernel should be executed. This is an optional parameter and can be omitted if you are not working with streams.
  arg1, arg2, ...: The arguments to the kernel function.

- **Destroying Streams:** Streams should be destroyed after use.

```
cudaStreamDestroy(stream);

__global__ void foo_kernel(int step)
{
printf("loop: %d\n", step);
}
 int main()
{
int n_stream = 5;
cudaStream_t *ls_stream;
ls_stream = (cudaStream_t*) new cudaStream_t[n_stream];
// create multiple streams
 for (int i = 0; i < n_stream; i++)
cudaStreamCreate(&ls_stream[i]);
// execute kernels with the CUDA stream each
 for (int i = 0; i < n_stream; i++)
foo_kernel<<< 1, 1, 0, ls_stream[i] >>>(i);
// synchronize the host and GPU
cudaDeviceSynchronize();
 // terminates all the created CUDA streams
for (int i = 0; i < n_stream; i++)
cudaStreamDestroy(ls_stream[i]);
delete [] ls_stream;
 return 0;
 }
```



As you can see at the bottom of the screenshot, five individual streams execute the same kernel function concurrently and their operations are overlapped with each other. From this, we can discern two features of the streams, as follows:

1. Kernel executions are asynchronous with the host.

2. CUDA operations in different streams are independent of each other

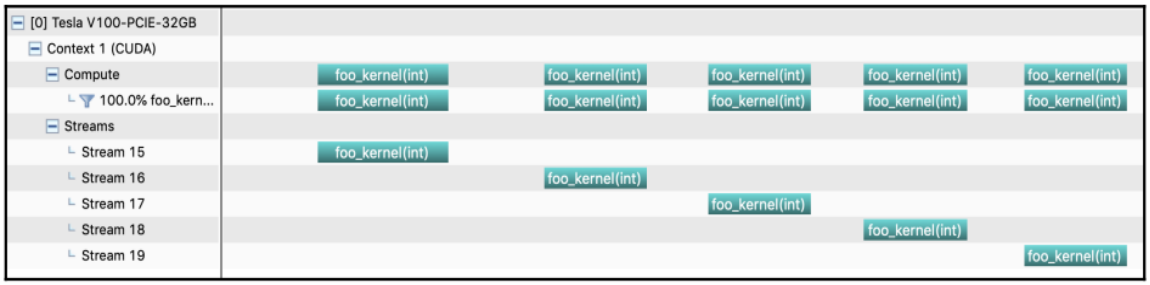
Stream-level synchronization:

In the context of CUDA (Compute Unified Device Architecture), stream-level synchronization refers to coordinating the execution of multiple CUDA streams. CUDA streams are sequences of operations that execute on a GPU. Each stream can contain a series of kernel launches, memory transfers, and other GPU operations.

The previous example shows concurrent operations without synchronization within the loop. However, we can halt the host to execute the next kernel execution by using the cudaStreamSynchronize() function. The following code shows an example of using stream synchronization at the end of the kernel execution:

```
// execute kernels with the CUDA stream each
 for (int i = 0; i < n_stream; i++)
 {
foo_kernel<<< 1, 1, 0, ls_stream[i] >>>(i);
cudaStreamSynchronize(ls_stream[i]);
 }
```

The following screenshot shows the result:



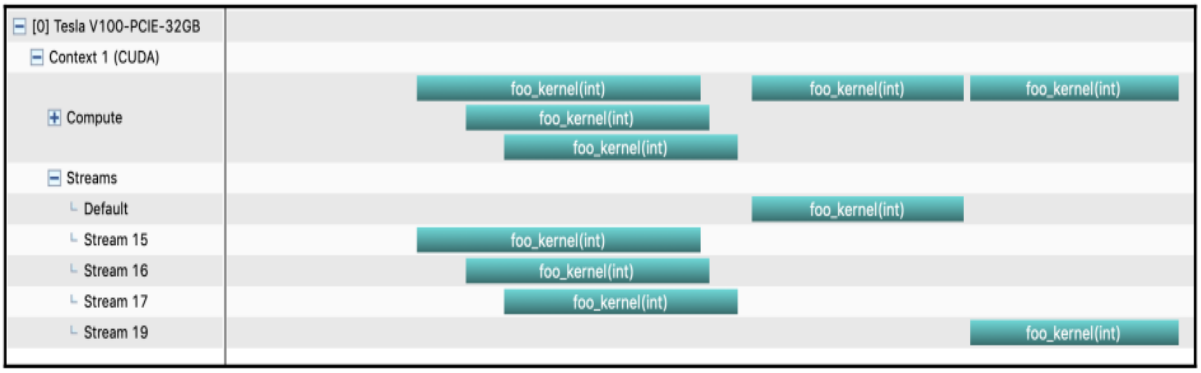
As you can see, all the kernel executions have no overlapping points, although they are executed with the different streams. Using this feature, we can let the host wait for the specific stream operation to start with the result.

Working with the default stream

In CUDA, you can use multiple streams to achieve concurrent execution of GPU operations. Each stream operates independently, allowing you to overlap the execution of kernels and memory transfers. However, the default stream (stream 0) operates synchronously, meaning that operations within the default stream are executed sequentially.

To use multiple streams in CUDA, you typically create and manage them explicitly in your code. Here's a simple example that demonstrates the use of multiple streams:

```
for (int i = 0; i < n_stream; i++)
  if (i == 3)
       foo_kernel<<< 1, 1, 0, 0 >>>(i);
else
     foo_kernel<<< 1, 1, 0, ls_stream[i] >>>(i)
```



Pipelining the GPU execution

In a typical CUDA program, the CPU sends data to the GPU, the GPU processes the data, and the results are then sent back to the CPU. This can result in idle time where the CPU is waiting for the GPU to finish processing or vice versa. **Pipelining** can be implemented to overlap data transfers and kernel executions, improving overall performance by ensuring that the CPU and GPU are both working efficiently without unnecessary idle times. **CUDA Streams** are particularly useful for this purpose, as they allow for asynchronous execution of memory copies and kernel launches, enabling pipelining of tasks.

In the context of **CUDA**, pipelining can involve overlapping:

1. **Data transfer** between the host (CPU) and device (GPU),
2. **Kernel execution** on the GPU, and
3. **Memory access** operations (reading/writing data in and out of global, shared, or constant memory).

Example:

While the GPU is executing the first kernel, the CPU can be preparing the next batch of data and transferring it to the GPU. Similarly, when the first kernel finishes, the CPU can start retrieving the results, while the GPU begins executing the next kernel.

To enable such a pipelining operation, CUDA has three prerequisites:

1. The host memory should be allocated as pinned memory—CUDA provides the cudaMallocHost() and cudaFreeHost() functions for this purpose. Or static memory(Normal Array declaration).

2. Transfer data between the host and GPUs without blocking the host—CUDA provides the cudaMemcpyAsync() function for this purpose.

3. Manage each operation along with the different CUDA streams to have concurrent operations

Building a pipelining execution

```
const int N = 5; // size of the vectors

// CUDA kernel for vector addition

__global__ void vectorAdd(float *a, float *b, float *c, int size)
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
      c[tid] = a[tid] + b[tid];
}

int main()
{
   const int blockSize = 256;

   const int numBlocks = (N + blockSize - 1) / blockSize;

   // Allocate and initialize host vectors
  int h_a[5] = {1, 1, 1, 1, 1}, h_b[5] = {2, 2, 2, 2, 2}, h_c[5];

   // Allocate device vectors
   int *d_a, *d_b, *d_c;
   cudaMalloc((void**)&d_a, N * sizeof(int));
   cudaMalloc((void**)&d_b, N * sizeof(int));
   cudaMalloc((void**)&d_c, N * sizeof(int));

   // Create CUDA streams
   cudaStream_t stream1, stream2;
   cudaStreamCreate(&stream1);
   cudaStreamCreate(&stream2);

   // Copy data to device asynchronously using streams
```

```
cudaMemcpyAsync(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(d_b, h_b, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

// Launch the kernel asynchronously using stream1
vectorAdd<<<numBlocks, blockSize, 0, stream1>>>(d_a, d_b, d_c, N);

// Copy data back to host asynchronously using stream2
cudaMemcpyAsync(h_c, d_c, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

// Synchronize streams to ensure all operations are completed
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);

// Verify the result
for (int i = 0; i < N; ++i) {
   printf("%f ", h_c[i]);
}

// Free resources
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

// Destroy streams
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);

return 0;}
```

The CUDA callback function

CUDA callback functions are user-defined functions executed on the host (CPU) after certain CUDA operations complete on the device (GPU). They can be used for tasks like synchronization, logging, or resource management. CUDA provides the callback functionality through its asynchronous streams.

Scenario: Real-Time Image Processing Pipeline

Imagine you're building an application that processes a series of images to enhance their brightness and then displays them on a screen.

Steps in the Workflow:

1. **Image Processing on GPU**:
    - An image is uploaded to the GPU.
    - The GPU adjusts the brightness of the image using a CUDA kernel.
2. **Host-Side Task**:
    - Once the GPU finishes processing the image, a callback is triggered on the host (CPU).
    - The callback takes the processed image and sends it to a display routine (e.g., rendering on the screen).
3. **Repeat**:
    - The next image is processed while the host handles displaying the current image, ensuring parallelism between CPU and GPU tasks.

The main CUDA callback functions include:

- **Error Callbacks (**cudaSetErrorCallback**):** You can register a callback function to be called whenever a CUDA runtime API call returns an error. This can be useful for custom error handling or logging.

  cudaSetErrorCallback(errorCallback, nullptr);

  errorCallback: This is a pointer to the user-defined error callback function.The function takes three parameters:
- **Event Callbacks (**cudaEventCreateWithCallback**):** You can register a callback function to be called when a CUDA event is recorded. This is useful for synchronizing CPU and GPU tasks.

cudaEventCreateWithCallback(&event, cudaEventDefault, eventCallback, nullptr);

&event: This is the address of the cudaEvent_t variable where the newly created event will be stored.

cudaEventDefault: This parameter specifies the flags associated with the event. cudaEventDefault is a default flag value, and you can customize it based on your requirements.

eventCallback: This is the callback function that will be called when the event reaches the specified state.

```
void errorCallback(cudaError_t error, const char *description, void *userData)
{
   printf( "CUDA error: %s ", description);
}
// Event callback function
void eventCallback(cudaStream_t stream, cudaError_t status, void *userData)
 {

Printf("CUDA Event Callback: Event completed with status %s", cudaGetErrorString(status) );

}
__global__ void myKernel() {
   // Some kernel code
}
int main()
{
   // Set error callback
   cudaSetErrorCallback(errorCallback, nullptr);

   // Create event with callback
   cudaEvent_t event;
   cudaEventCreateWithCallback(&event, cudaEventDefault, eventCallback, nullptr);

   // Allocate device data
   int *d_data;
   cudaMalloc((void**)&d_data, sizeof(int));

   // Launch kernel asynchronously
   myKernel<<<1, 1, 0, nullptr>>>(d_data);

   // Cleanup
   cudaFree(d_data);
   cudaEventDestroy(event);

   return 0;}
```

CUDA streams with priority

By default, all CUDA streams have equal priority so they can execute their operations in the right order. However, starting with CUDA 11.0, NVIDIA introduced the concept of stream priorities. Stream priorities allow you to assign different priorities to CUDA streams, influencing the order in which they are scheduled for execution. Higher priority streams are scheduled to run before lower priority streams.

Priorities in CUDA

- To use streams with priorities, we need to obtain the available priorities from the GPU first. We can obtain these using the cudaDeviceGetStreamPriorityRange() function. Its output is two numeric values, which are the lowest and highest priority values.

cudaError_t cudaDeviceGetStreamPriorityRange(int* leastPriority, int* greatestPriority);

- **leastPriority**: Pointer to an integer in which the function stores the minimum supported stream priority.
- **greatestPriority**: Pointer to an integer in which the function stores the maximum supported stream priority.

The `cudaDeviceGetStreamPriorityRange` function returns `cudaSuccess` on success or an error code if there's an issue.

- Then, we can create a priority stream using the cudaStreamCreateWithPriority() function .This function is part of the CUDA runtime API, and it allows you to create a CUDA stream with a specified priority. This function was introduced in CUDA 10.0 and is used to create streams with different priorities, influencing the order in which they are scheduled for execution on the GPU.

  cudaError_t cudaStreamCreateWithPriority(cudaStream_t* pStream, unsigned int flags, int priority);

  - **pStream**: Pointer to a variable that will receive the newly created stream.
  - **flags**: Flags that can be used to specify additional stream creation options. Typically, this is set to **cudaStreamDefault**.
  - **priority**: The priority of the stream. A higher priority value indicates a higher-priority stream.

Stream execution with priorities

```
void checkCudaError(cudaError_t cudaStatus, const char* errorMessage)
{
```

```
    if (cudaStatus != cudaSuccess) {
printf(“%s”, cudaGetErrorString(cudaStatus) );
    }
}

int main() {
    // Get the priority range supported by the device
    int leastPriority, greatestPriority;
    cudaError_t cudaStatus = cudaDeviceGetStreamPriorityRange(&leastPriority, &greatestPriority);
    checkCudaError(cudaStatus, "cudaDeviceGetStreamPriorityRange failed");
    // Create streams with different priorities
    int highPriority = greatestPriority;
    int lowPriority = leastPriority;
    cudaStream_t  streamHigh, streamLow;
    cudaStatus = cudaStreamCreateWithPriority(&streamHigh, cudaStreamDefault, highPriority);
    checkCudaError(cudaStatus, "cudaStreamCreateWithPriority (High Priority) failed");
    cudaStatus = cudaStreamCreateWithPriority(&streamLow, cudaStreamDefault, lowPriority);
    checkCudaError(cudaStatus, "cudaStreamCreateWithPriority (Low Priority) failed");
    // Use the created streams for kernel launches or memory transfers
    // Destroy the streams when done
    cudaStatus = cudaStreamDestroy(streamHigh);
    checkCudaError(cudaStatus, "cudaStreamDestroy (High Priority) failed");
    cudaStatus = cudaStreamDestroy(streamLow);
    checkCudaError(cudaStatus, "cudaStreamDestroy (Low Priority) failed");
    return 0;
}
```

UNIT-V

Topic 3: CUDA events & Dynamic Parallelism

Kernel execution time estimation using CUDA events

CUDA events are a mechanism provided by NVIDIA's CUDA programming model to measure elapsed time and synchronize tasks on the GPU. CUDA events are particularly useful for performance measurement, debugging, and controlling task execution order.

Example : Imagine a scenario where we want to measure the time taken by a CUDA kernel to process an image filter operation. We also need to ensure that one operation (e.g., applying a filter) completes before another operation (e.g., normalizing the result) begins

Here's an overview of how CUDA events work:

- **Creating Events:** To use CUDA events, you need to create them. This is done using the `cudaEventCreate` function:

  ```
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  ```

- **Recording Events:** You can record events at specific points in your code, typically before and after the region of code you want to measure. This is done using the `cudaEventRecord` function:

  ```
  cudaEventRecord(start);
  // Code to be timed
  cudaEventRecord(stop);
  ```

- **Synchronizing Events:** To ensure that the recorded events have completed, you can use the `cudaEventSynchronize` function:

  ```
  cudaEventSynchronize(stop);
  ```

  This is especially important when measuring the execution time of a kernel to make sure that the GPU has finished executing the kernel.

- **Calculating Elapsed Time:** The elapsed time between two events can be calculated using the `cudaEventElapsedTime` function:

  ```
  float elapsedTime;
  cudaEventElapsedTime(&elapsedTime, start, stop);
  ```

  The elapsedTime variable will contain the time between the start and stop events in milliseconds.

- **Destroying Events:** Once you're done using the events, it's important to destroy them to free up resources:

  cudaEventDestroy(start);

  cudaEventDestroy(stop);

Sample Program

```
__global__ void myKernel()
{
  // Some computation
}

int main()
{
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  // Record the start event
  cudaEventRecord(start);

  // Launch the kernel
  myKernel<<<1, 1>>>();

  // Record the stop event
  cudaEventRecord(stop);

  // Synchronize to ensure accurate timing
  cudaEventSynchronize(stop);

  // Calculate and print the elapsed time
  float elapsedTime;
  cudaEventElapsedTime(&elapsedTime, start, stop);

  // Destroy the events
  cudaEventDestroy(start);
  cudaEventDestroy(stop);

  return 0;
}
```

Multiple stream estimation

When dealing with multiple CUDA streams, you might want to estimate the overall execution time, taking into account the asynchronous nature of stream execution. The basic idea is to use events to

measure the time taken by each stream and synchronize with the CPU to ensure accurate measurements

```
// Example kernel
__global__ void myKernel(int* data, int size)
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;

      data[tid] = data[tid] * 2;

}

int main()
{
   const int dataSize = 5;
   const int blockSize= 256;
   const int gridSize = (dataSize + blockSize - 1) / blockSize;

   int h_data[5] = {1, 2, 3, 4, 5};
   int* d_data[2];  // Two streams

   for (int i = 0; i < 2; ++i) {
      // Allocate device memory for each stream
      cudaMalloc((void**)&d_data[i], dataSize * sizeof(int));
      // Copy data from host to device
      cudaMemcpy(d_data[i], h_data, dataSize * sizeof(int), cudaMemcpyHostToDevice);
   }

   cudaStream_t stream[2];

   for (int i = 0; i < 2; ++i) {
      // Create streams
      cudaStreamCreate(&stream[i]);
   }

   cudaEvent_t start[2], stop[2];

   for (int i = 0; i < 2; ++i) {
      // Create events
      cudaEventCreate(&start[i]);
      cudaEventCreate(&stop[i]);
   }

   // Record the start events
   for (int i = 0; i < 2; ++i) {
      cudaEventRecord(start[i], stream[i]);
```

```
    }

    // Launch the kernels in parallel streams
    for (int i = 0; i < 2; ++i) {
        myKernel<<<gridSize, blockSize, 0, stream[i]>>>(d_data[i], dataSize);
    }

    // Record the stop events
    for (int i = 0; i < 2; ++i) {
        cudaEventRecord(stop[i], stream[i]);
    }

    // Synchronize to ensure accurate timing
    for (int i = 0; i < 2; ++i) {
        cudaStreamSynchronize(stream[i]);
    }

    // Calculate and print the elapsed time for each stream
    for (int i = 0; i < 2; ++i) {
        float elapsedTime;
        cudaEventElapsedTime(&elapsedTime, start[i], stop[i]);
        printf("Stream %d execution time is %f ms\n", i, elapsedTime);
    }

    // Clean up
    for (int i = 0; i < 2; ++i) {
        cudaFree(d_data[i]);
        cudaStreamDestroy(stream[i]);
        cudaEventDestroy(start[i]);
        cudaEventDestroy(stop[i]);
    }

    return 0;
}
```

CUDA dynamic parallelism

Dynamic parallelism in CUDA refers to the ability of a CUDA kernel to launch other kernels. In traditional CUDA programming, kernels are launched from the host, and they run on the GPU. With dynamic parallelism, a kernel running on the GPU can itself launch additional kernels. CUDA dynamic parallelism (CDP) is a device runtime feature that enables nested calls from device functions

```
__global__ void childKernel(int* data, int index)
{
    data[index] = data[index] + 1;
}
```

```
__global__ void parentKernel(int* data, int size)
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
      childKernel<<<1, 1>>>(data, tid);
      cudaDeviceSynchronize();  // Ensure child kernel completes
}

int main() {
   const int dataSize = 5;
   const int blockSize = 256;
   const int gridSize = (dataSize + blockSize - 1) / blockSize;

   int h_data[dataSize] = {1, 2, 3, 4, 5};
   int* d_data;

   cudaMalloc((void**)&d_data, dataSize * sizeof(int));
   cudaMemcpy(d_data, h_data, dataSize * sizeof(int), cudaMemcpyHostToDevice);

   // Launch parent kernel
   parentKernel<<<gridSize, blockSize>>>(d_data, dataSize);
   cudaDeviceSynchronize();

   // Copy data back to host
   cudaMemcpy(h_data, d_data, dataSize * sizeof(int), cudaMemcpyDeviceToHost);

   // Print modified values
   for (int i = 0; i < dataSize; ++i)
{
      printf("Modified value at index %d: %d\n", i, h_data[i]);
   }

   cudaFree(d_data);

   return 0;
}
```

Recursion using Dynamic Parallelism

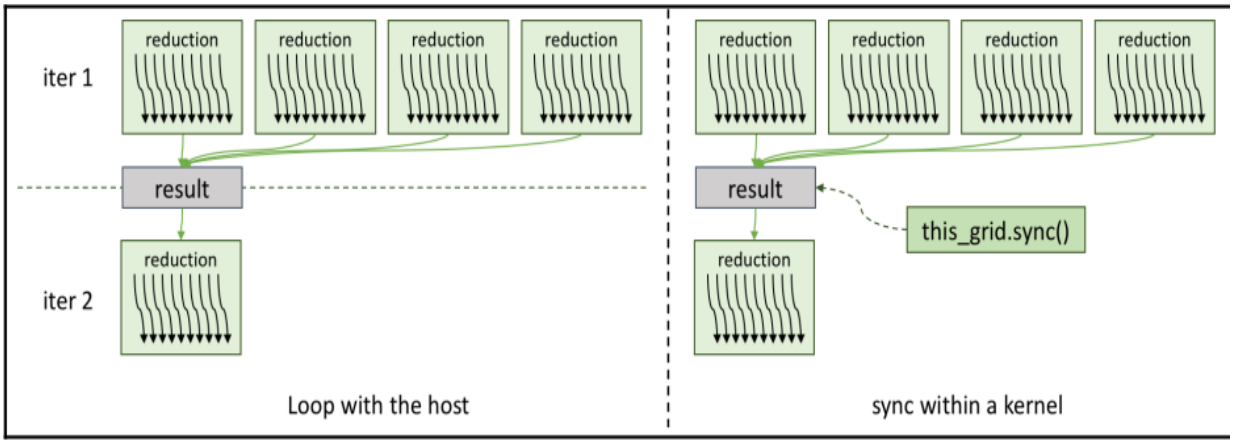
```
// GPU kernel for recursive factorial calculation
__global__ void factorialKernel(int n, int *result) {
   if (n == 1 || n == 0) {
      *result = 1;  // Base case
      return;
   }

   if (n > 1) {
```

```
        int partialResult = n;
        int *childResult;

        // Allocate memory for the result of the child kernel
        cudaMalloc(&childResult, sizeof(int));

        // Launch child kernel to calculate factorial(n-1)
        factorialKernel<<<1, 1>>>(n - 1, childResult);

        // Wait for child kernel to complete
        cudaDeviceSynchronize();

        // Retrieve result from child kernel
        cudaMemcpy(&partialResult,    childResult,     sizeof(int), cudaMemcpyDeviceToHost);

        // Combine result
        *result = n * partialResult;

        // Free the memory allocated for the child result
        cudaFree(childResult);
    }
}

// Host function
int main() {
    int n = 5;  // Example: Calculate factorial of 5
    int result;
    int  *d_result;

    // Allocate memory on the device for the result
    cudaMalloc(&d_result, sizeof(int));

    // Launch the factorial kernel
    factorialKernel<<<1, 1>>>(n, d_result);

    // Wait for the kernel to complete
    cudaDeviceSynchronize();

    // Copy the result back to the host
    cudaMemcpy(&result, d_result, sizeof(int), cudaMemcpyDeviceToHost);

    // Print the result
    // Free device memory
    cudaFree(d_result);

    return 0;
}
```

UNIT-V
TOPIC 4:
Grid level cooperative groups

Grid-level **cooperative groups** in CUDA are a powerful feature that enable collaboration and synchronization across an entire grid of thread blocks, not just within a single block. This allows for more advanced parallel programming patterns in CUDA, particularly for workloads requiring fine-grained inter-block synchronization.

Reduction example with grid.sync()



For implementing cooperative groups we require two functions: grid_group & grid.sync().

➢ The **grid_group** is part of the **cooperative_groups** library in CUDA, introduced to provide a more flexible way for threads in different thread blocks to synchronize and collaborate within a grid. This is particularly useful when you need cooperation or communication between different thread blocks.

To make all the thread blocks in grid_group synchronize, the total number of active thread blocks in the grid should not exceed the number of maximum active blocks for the kernel function and the device. The maximum active block size on a GPU is a multiplication of the maximum amount of active blocks per SM and the number of streaming multiprocessors. The violation of this rule can result in deadlock or undefined behavior. We can obtain the maximum amount of active thread blocks of a kernel function per SM using the cudaOccupancyMaxActiveBlocksPerMultiprocessor() function, by passing the kernel function and block size information.

- **grid.sync()** is used to synchronize all threads within the grid. All threads within the grid must reach the **grid.sync()** statement. No thread in any block is allowed to proceed beyond the **grid.sync()** until all threads in all blocks have reached it.Once all threads in the grid have reached this point, they will be synchronized.This synchronization allows you to perform collective operations or share data between blocks.

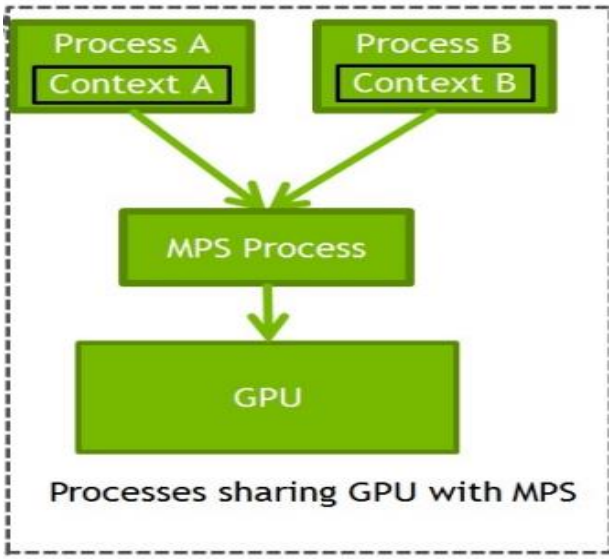
Example

```
namespace cg = cooperative_groups;
__global__ void gridLevelKernel(int* data, int size)
{
   cg::grid_group grid = cg::this_grid();
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   // Work on data assigned to the block
   // Synchronize all threads within the grid
   grid.sync();
   // Perform some computation collectively across all blocks
   // Synchronize again
   grid.sync();
}
int main() {
   const int blockSize = 256;
   const int gridSize = 4;
   const int dataSize = blockSize * gridSize;
   int* d_data;
   cudaMalloc((void**)&d_data, dataSize * sizeof(int));
   // Launch the kernel with multiple blocks
   gridLevelKernel<<<gridSize, blockSize>>>(d_data, dataSize);
   // Wait for the kernel to finish
```

```
    cudaDeviceSynchronize();
    // Cleanup
    cudaFree(d_data);
    return 0;
}
```

Multi-Process Service

- Multi-Process Service is a feature provided by NVIDIA GPUs that allows concurrent execution of kernels from different CPU processes on the same GPU. This is particularly beneficial when multiple applications or processes are trying to utilize the GPU simultaneously.
- By default, when multiple CPU processes try to use the GPU concurrently, the GPU time-slices between them. This means that the GPU switches between executing kernels from different processes over time.The default time-slicing approach can lead to underutilization of GPU resources because even if a kernel doesn't fully use the GPU's compute resources, it has to wait for its turn in the time slice.
- To address this issue and fully utilize GPU resources, MPS mode is introduced. In MPS mode, different CPU processes can execute their kernels simultaneously on the GPU. This allows for better parallelism and resource utilization.It is particularly useful in scenarios where you have multiple applications running concurrently, each using the GPU.
- While MPS mode can be beneficial, it's essential to consider factors such as memory usage, context switching, and potential contention for resources. Proper management of MPS mode is required to avoid resource conflicts between concurrent processes.



To have the multi-processes application scenario to a single GPU we will use MPI.MPI is commonly used for parallel computing, where multiple processes work together to solve a larger computational problem. These processes can run on different CPU cores, GPUs, or even on different nodes within a cluster.

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

int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   printf("Hello from process %d of %d\n", rank, size);

   MPI_Finalize();
   return 0;
}
```

UNIT-V

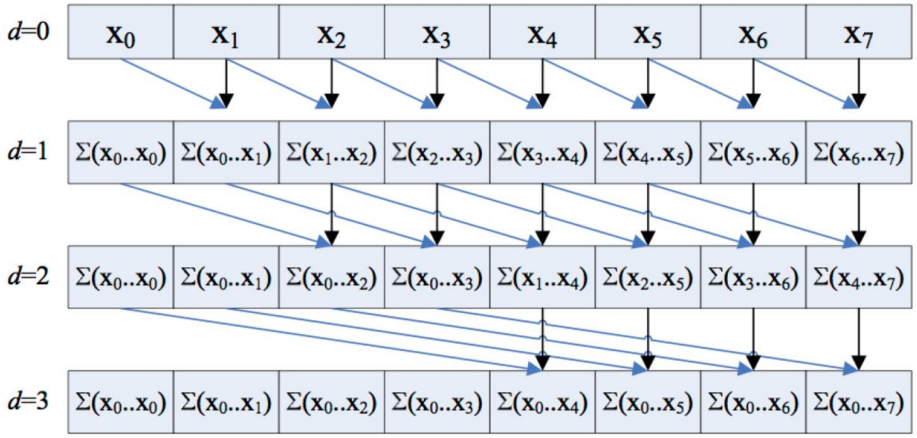
TOPIC 5

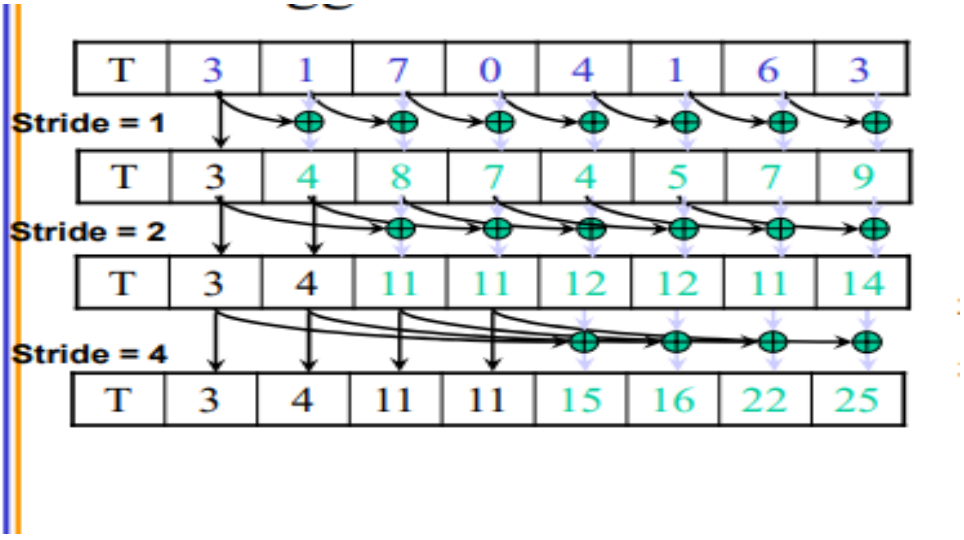
Prefix sum

Prefix sum (scan) is used to obtain a cumulative number array from the given input numbers array. For example, we can make a prefix-sum sequence as follows:

Input numbers	1	2	3	4	5	6
Prefix sums	1	3	6	10	15	21

It differs from parallel reduction since reduction just generates the total operation output from the given input data. On the other hand, scan generates outputs from each operation. The easiest way to solve this problem is to iterate all the inputs to generate the output. However, it would take a long time and would be inefficient in GPUs. Hence, the mild approach can parallelize the prefix-sum operation, as follows:



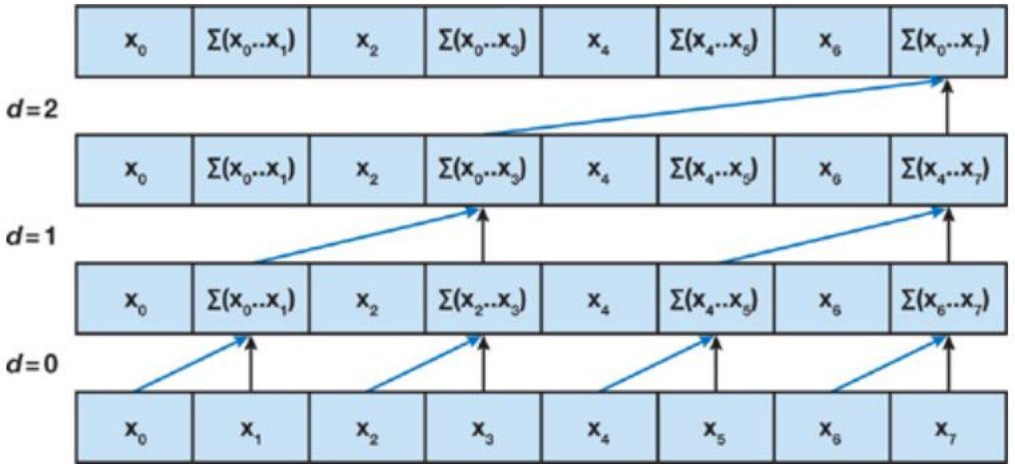



```
#define N 1024

__global__ void prefixSum(int *input, int *output, int n)
```

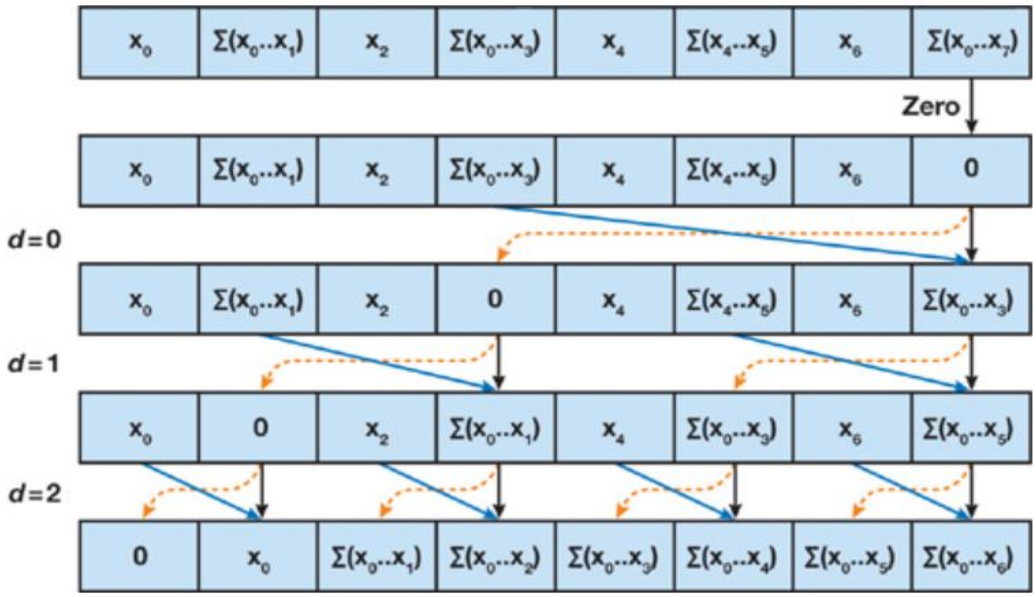
```
{
   int thid = threadIdx.x;

   // Perform parallel reduction to calculate partial sums
   for (int offset = 1; offset < n; offset *= 2)
{
      if (thid + offset < n) {
         output[thid + offset] += output[thid];
      }
      __syncthreads();
   }
}

int main()
{
   int *h_input = new int[N];
   int *h_output = new int[N];
   int *d_input, *d_output;

   // Initialize input data (for simplicity, using sequential values)
   for (int i = 0; i < N; ++i)
{
      h_input[i] = i;
   }

   // Allocate device memory
   cudaMalloc((void**)&d_input, N * sizeof(int));
   cudaMalloc((void**)&d_output, N * sizeof(int));

   // Copy input data to device
   cudaMemcpy(d_input, h_input, N * sizeof(int), cudaMemcpyHostToDevice);

   // Define block and grid dimensions
   dim3 dimBlock(N, 1);
   dim3 dimGrid(1, 1);

   // Launch the prefix sum kernel
   prefixSum<<<dimGrid, dimBlock>>>(d_input, d_output, N);

   // Copy the result back to the host
   cudaMemcpy(h_output, d_output, N * sizeof(int), cudaMemcpyDeviceToHost);

   // Print some elements of the result array for verification
   for (int i = 0; i < 10; ++i) {
      printf("%d\t",h_output[i]);
   }
```

```
    // Free device memory
    cudaFree(d_input);    cudaFree(d_output);
    delete[] h_input;    delete[] h_output;
    return 0;
}
```

There is another optimized approach named. This method generates prefixsum outputs by increasing and decreasing the strides exponentially. This method's procedure is shown in the following diagram:



Upsweep(Reduce Phase)



Downsweep

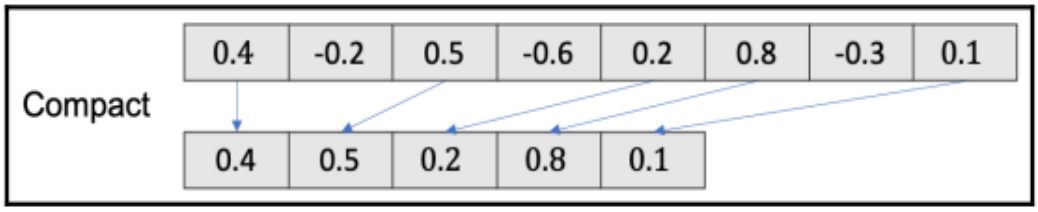
```
#define N 1024
__global__ void upsweep(int *data, int stride) {
    int thid = threadIdx.x;
    int offset = 2 * stride * thid;
    data[offset + 2 * stride - 1] += data[offset + stride - 1];
```

```
}

__global__ void downsweep(int *data, int stride) {
   int thid = threadIdx.x;
   int offset = 2 * stride * thid;

   int temp = data[offset + stride - 1];
   data[offset + stride - 1] = data[offset + 2 * stride - 1];
   data[offset + 2 * stride - 1] += temp;
}

__global__ void initializeLastElement(int *data, int n) {
   // Set the last element to 0 for inclusive scan
   data[n - 1] = 0;
}
int main() {
   int *h_data = new int[N];
   int *d_data;
   // Initialize input data (for simplicity, using sequential values)
   for (int i = 0; i < N; ++i) {
      h_data[i] = i;
   }
   // Allocate device memory
   cudaMalloc((void**)&d_data, N * sizeof(int));

   // Copy input data to device
   cudaMemcpy(d_data, h_data, N * sizeof(int), cudaMemcpyHostToDevice);

   // Define block dimensions
   dim3 dimBlock(N / 2, 1);

   // Upsweep (reduce) phase
   for (int stride = 1; stride < N; stride *= 2) {
      dim3 dimGrid(1, 1);
      upsweep<<<dimGrid, dimBlock>>>(d_data, stride);
   }
   // Initialize last element to 0
   initializeLastElement<<<1, 1>>>(d_data, N);

   // Downsweep (inclusive scan) phase
   for (int stride = N / 2; stride > 0; stride /= 2) {
      dim3 dimGrid(1, 1);
      downsweep<<<dimGrid, dimBlock>>>(d_data, stride);
   }
   // Copy the result back to the host
   cudaMemcpy(h_data, d_data, N * sizeof(int), cudaMemcpyDeviceToHost);

   // Print some elements of the result array for verification
   // Free device memory
```

```
    cudaFree(d_data);    delete[] h_data;
    return 0;
}
```

Compact and split

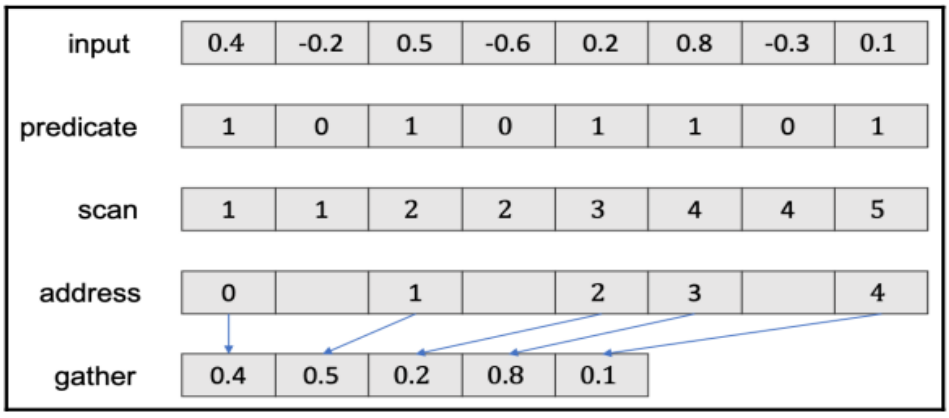
Compaction removes unwanted elements (e.g., zeros or negative numbers) from an array while preserving the order of the remaining elements. This often accompanies prefix sums to help generate indices for the compacted output.



Steps for Compaction:

1. Use a **predicate function** to determine which elements to keep.
2. Use prefix sums on the predicate results to compute indices for the output array.
3. Scatter the valid elements into the compacted array using the computed indices.

The following diagram shows an example of a compact operation:



To implement a compact operation, we will write several kernel functions that can do the required operation for each step and call those last:

1.Let's write a kernel function that can make a predicate array by checking whether each element's value is greater than zero or not:

```
__global__ void predicate_kernel(float *d_predicates, float *d_input, int length)
{
int idx = blockDim.x * blockIdx.x + threadIdx.x;
if (idx >= length) return;
```

```
d_predicates[idx] = d_input[idx] > FLT_ZERO;
}
```

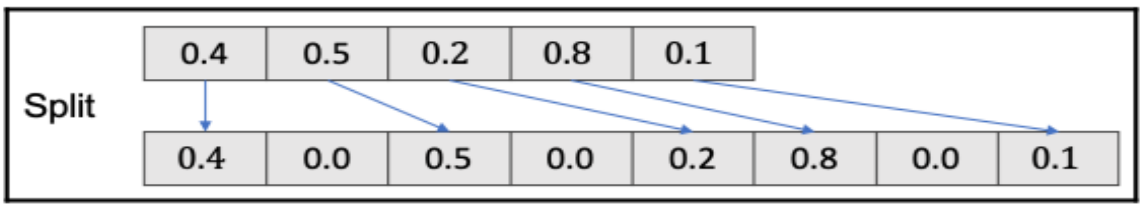
2. Then, we have to perform a prefix-sum operation for that predicate array. We will reuse the previous implementation here. After that, we can write a kernel function that can detect the address of the scanned array and gather the target elements as output:

```
__global__ void pack_kernel(float *d_output, float *d_input, float *d_predicates, float *d_scanned,
int length)
{
 int idx = blockDim.x * blockIdx.x + threadIdx.x;
 if (idx >= length) return;
if (d_predicates[idx] != 0.f)
{ // addressing
 int address = d_scanned[idx] - 1;
// gather
d_output[address] = d_input[idx];
} }
```

3. Now, let's call them all together to make a compact operation:

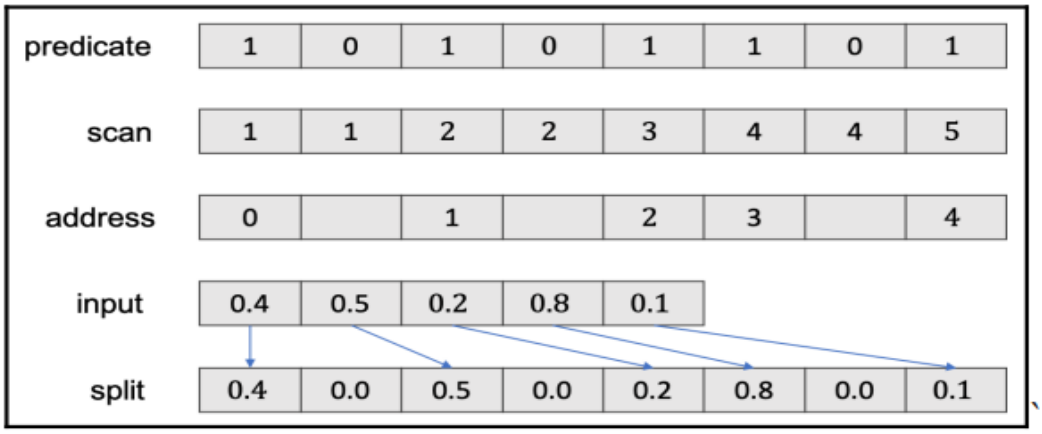
```
// predicates
predicate_kernel<<< GRID_DIM, BLOCK_DIM >>>(d_predicates, d_input, length);
// scan
 scan_v2(d_scanned, d_predicates, length);
// addressing & gather (pack)
pack_kernel<<< GRID_DIM, BLOCK_DIM >>>(d_output, d_input, d_predicates, d_scanned, length);
```

A split operation typically refers to dividing data into multiple parts, often based on a certain criterion. This can be useful when you want to perform parallel processing on different subsets of the data.



We can also do this in parallel using prefix-sum. Firstly, we refer to the predicate array and do the

prefix-sum. Since the outputs are each element's address, we can distribute them easily. The following diagram shows how this operation can be done.



Let's write the split kernel function, as follows: _

_global__ void split_kernel(float *d_output, float *d_input, float *d_predicates, float *d_scanned, int length)

{

int idx = blockDim.x * blockIdx.x + threadIdx.x;

if (idx >= length) return;

if (d_predicates[idx] != 0.f)

{ // address

int address = d_scanned[idx] - 1;

// split

d_output[idx] = d_input[address];

} }

2. Now, we can call the kernel function, as follows:

cudaMemcpy(d_input, d_output, sizeof(float) * length, cudaMemcpyDeviceToDevice);

cudaMemset(d_output, 0, sizeof(float) * length);

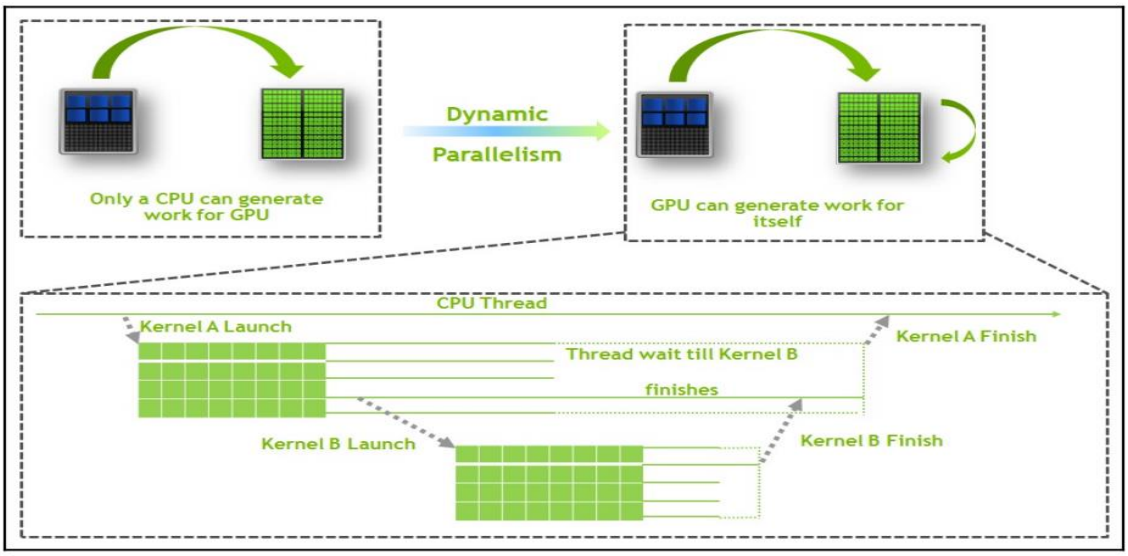
split_kernel<<< GRID_DIM, BLOCK_DIM >>>(d_output, d_input, d_predicates, d_scanned, length);

UNIT V
Topic 6
Quicksort in CUDA using dynamic parallelism

QuickSort is a divide-and-conquer sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

The Quicksort algorithm demands launching kernels recursively. So far, the algorithms we have seen call the kernel once via the CPU. After the kernel has finished executing, we return to the CPU thread and then relaunch it. Doing this results in giving back control to the CPU, and may also result in data transfer between CPU and GPU, which is a costly operation. It used to be very difficult to efficiently implement algorithms such as Quicksort on GPUs that demand features such as recursion. With the GPU architecture 3.5 and CUDA 5.0 onwards, a new feature was introduced called dynamic parallelism.

Dynamic parallelism allows the threads within a kernel to launch new kernels from the GPU without returning control back to the CPU. The word dynamic comes from the fact that it is dynamically based on the runtime data. Multiple kernels can be launched by threads at once. The following diagram simplifies this explanation.




```
const int MAX_DEPTH = 16; // Maximum recursion depth
__global__ void quicksort(int* array, int left, int right, int depth);
__device__ int partition(int* array, int left, int right) {
   int pivot = array[right];
   int i = left - 1;
   for (int j = left; j < right; j++) {
      if (array[j] <= pivot) {
```

```
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
    int temp = array[i + 1];
    array[i + 1] = array[right];
    array[right] = temp;

    return i + 1;
}
__global__ void quicksort(int* array, int left, int right, int depth) {
    if (depth <= 0 || left >= right) {
        // Use a simple sorting algorithm for small chunks or when recursion is deep enough
        for (int i = left + 1; i <= right; i++) {
            int j = i;
            while (j > left && array[j - 1] > array[j]) {
                int temp = array[j];
                array[j] = array[j - 1];
                array[j - 1] = temp;
                j--;
            }
        }
        return;
    }
    int pivotIndex = partition(array, left, right);
    // Launch two child kernels with new partitions
    if (left < pivotIndex) {
        quicksort<<<1, 1>>>(array, left, pivotIndex - 1, depth - 1);
    }
    if (pivotIndex < right) {
        quicksort<<<1, 1>>>(array, pivotIndex + 1, right, depth - 1);
    }
}
int main() {
    const int arraySize = 1000; // Adjust the size as needed
    const int arrayBytes = arraySize * sizeof(int);
    int* hostArray = new int[arraySize];
    int* deviceArray;
    // Initialize array with random values
    for (int i = 0; i < arraySize; i++) {
        hostArray[i] = rand() % 1000;
    }
    // Allocate and copy array to the device
    cudaMalloc((void**)&deviceArray, arrayBytes);
    cudaMemcpy(deviceArray, hostArray, arrayBytes, cudaMemcpyHostToDevice);
```

```
    // Launch quicksort kernel
    quicksort<<<1, 1>>>(deviceArray, 0, arraySize - 1, MAX_DEPTH);
    cudaDeviceSynchronize();
    // Copy the sorted array back to host
    cudaMemcpy(hostArray, deviceArray, arrayBytes, cudaMemcpyDeviceToHost);
    // Cleanup
    delete[] hostArray;
    cudaFree(deviceArray);
    return 0;
}
```

Radix Sort

Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys. Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

The detailed steps are as follows −

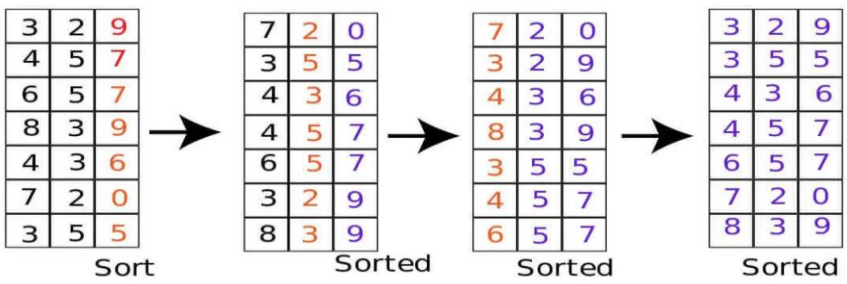
Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits using counting sort logic and change the order of elements based on the output achieved. For example, if the input elements are decimal numbers, the possible values each digit can take would be 0-9, so index the digits based on these values.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

Step 5 − The final list of elements achieved after kth loop is the sorted output.



Implementation using warp level primitives

```
#define NUM_BITS 8
#define NUM_BUCKETS 1 << NUM_BITS
__device__ int getDigit(int num, int digit, int numBits)
{
  return (num >> digit) & ((1 << numBits) - 1);
}

__global__ void radixSort(int* data, int numElements)
{
  extern __shared__ int sharedData[];
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  int laneId = threadIdx.x % warpSize;
  for (int bit = 0; bit < sizeof(int) * 8; bit += NUM_BITS)
{
    int digitMask = (1 << NUM_BITS) - 1;
    int shift = bit;

    // Warp-level radix sort
    for (int i = 0; i < NUM_BITS; ++i)
{
      int digit = getDigit(data[tid], shift, NUM_BITS);

      // Create a histogram using warp-level primitives
      unsigned int mask = (digitMask << laneId);
      int warpHistogram = __popc(__ballot_sync(mask, (digitMask & digit) == laneId));

      // Compute the prefix sum
      int warpPrefixSum = __popc(__ballot_sync(mask, laneId < digit));

      // Update the position of the current element
      int position = warpPrefixSum + warpHistogram;
      sharedData[position] = data[tid];

      __syncthreads();

      // Copy data back to global memory
      data[tid] = sharedData[threadIdx.x];

      shift += NUM_BITS;
    }
  }
}

int main() {
  // Initialize data on the host
```

```
    const int numElements = 1024;
    int data[numElements];
    for (int i = 0; i < numElements; ++i) {
        data[i] = rand() % 1000;
    }

    // Allocate and copy data to the device
    int* d_data;
    cudaMalloc((void**)&d_data, numElements * sizeof(int));
    cudaMemcpy(d_data, data, numElements * sizeof(int), cudaMemcpyHostToDevice);

    // Set grid and block dimensions
    dim3 gridSize(1, 1, 1);
    dim3 blockSize(1024, 1, 1);

    // Run the radix sort kernel
    radixSort<<<gridSize, blockSize, blockSize.x * sizeof(int)>>>(d_data, numElements);

    // Copy data back to the host
    cudaMemcpy(data, d_data, numElements * sizeof(int), cudaMemcpyDeviceToHost);

    // Free allocated memory on the device
    cudaFree(d_data);

    // Print sorted data
    for (int i = 0; i < numElements; ++i) {
        printf("%d ", data[i]);
    }

    return 0;
}
```

Thrust-based radix sort

Thrust is a parallel template library for CUDA, and it provides a high-level interface for GPU programming. It includes various parallel algorithms, and it has a radix sort implementation that you can use directly without having to write low-level CUDA code.

The steps involved in making use of Thrust for radix sort are as follows:

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
int main() {
    // Generate random data on the host
    thrust::host_vector<int> hostData(1024);
    for (int i = 0; i < 1024; ++i) {
        hostData[i] = rand() % 1000;
```

```
    }

    // Transfer data to the device
    thrust::device_vector<int> deviceData = hostData;

    // Use Thrust to perform radix sort on the device
    thrust::sort(deviceData.begin(), deviceData.end());

    thrust::copy(deviceData.begin(), deviceData.end(), hostData.begin());
    // Transfer data back to the host

    // Print sorted data
    return 0;
}
```