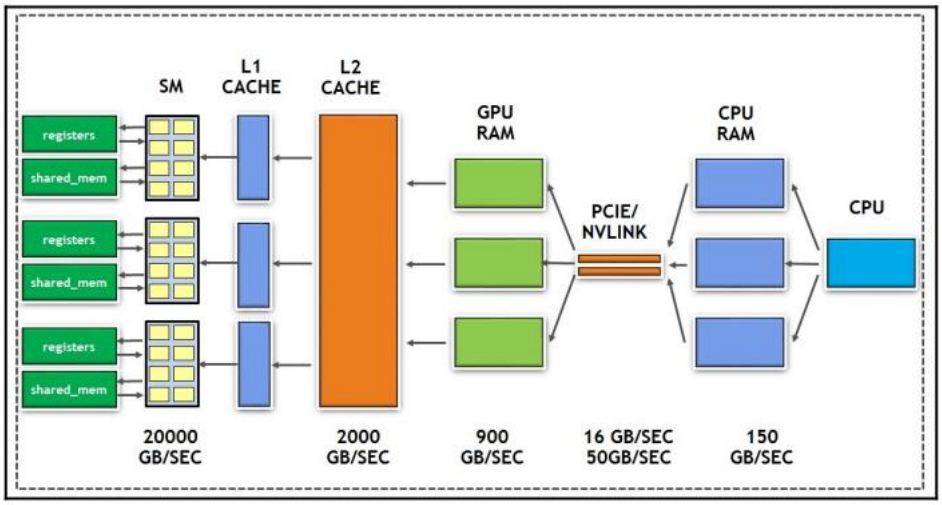
UNIT- IV

CUDA Memory Management-I

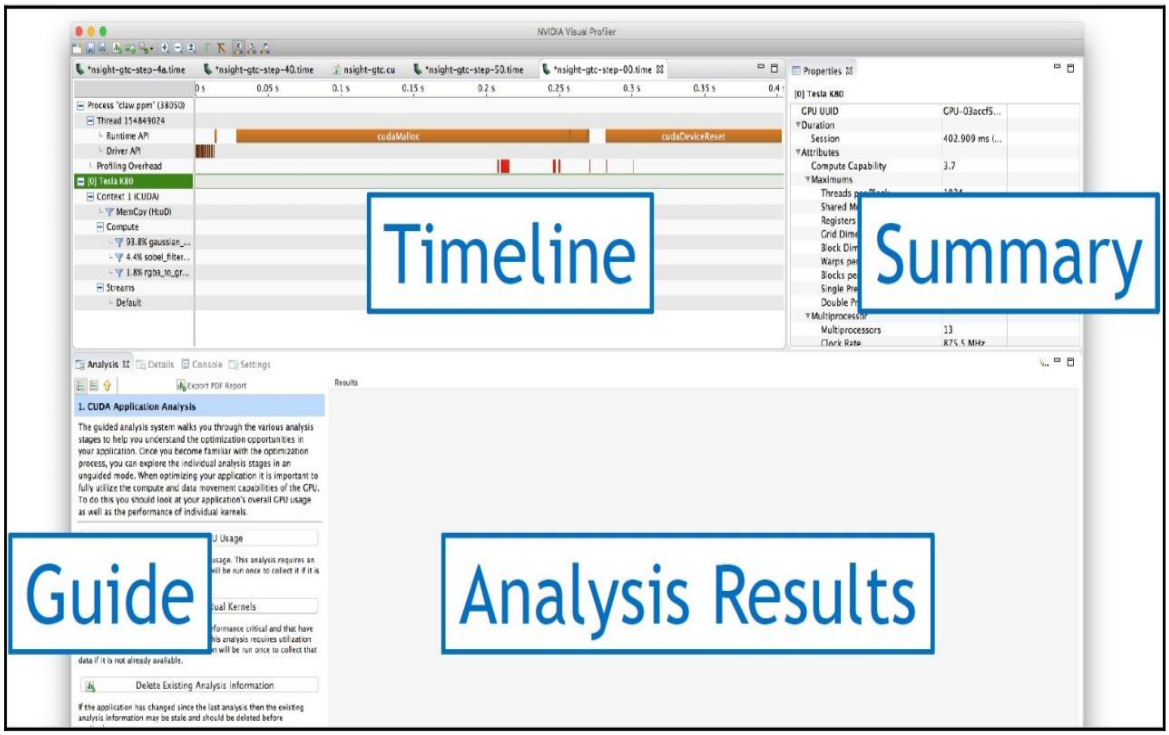
CUDA memory management is a crucial aspect of programming for NVIDIA GPUs using CUDA. It involves allocating, copying, and freeing memory on both the CPU and GPU, as well as understanding the different types of memory available.



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

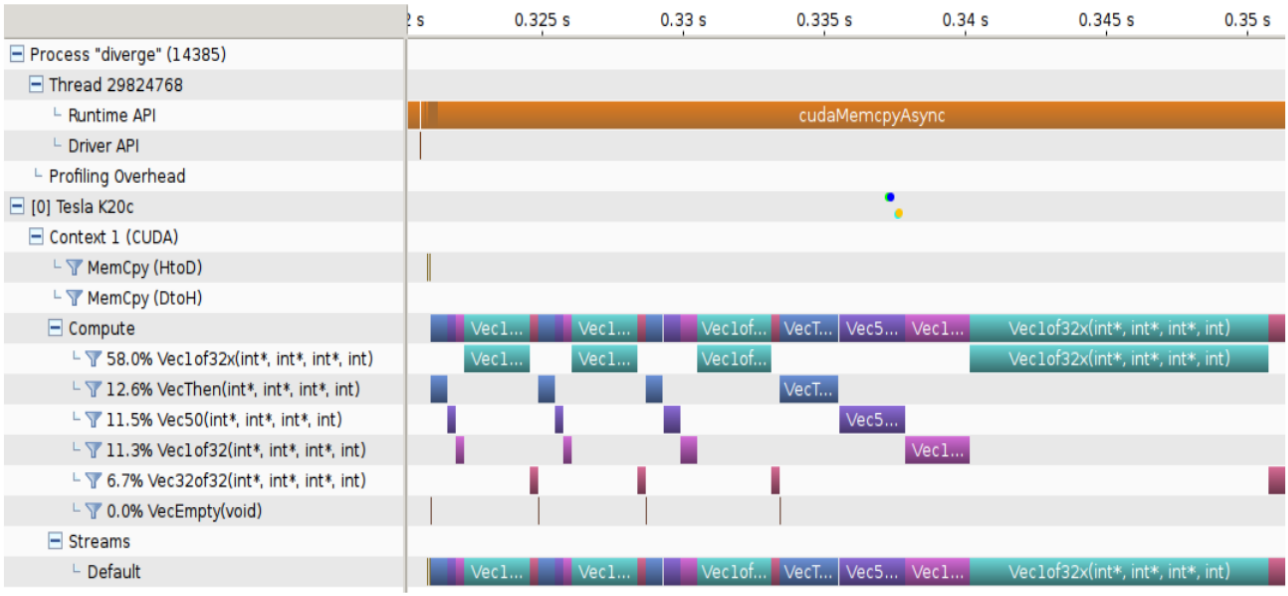
The NVIDIA Visual Profiler is a graphical profiling tool provided by NVIDIA as part of the CUDA Toolkit. It is designed to help developers analyse the performance of CUDA applications by providing insights into GPU utilization, memory usage, and execution timelines. The Visual Profiler assists in identifying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

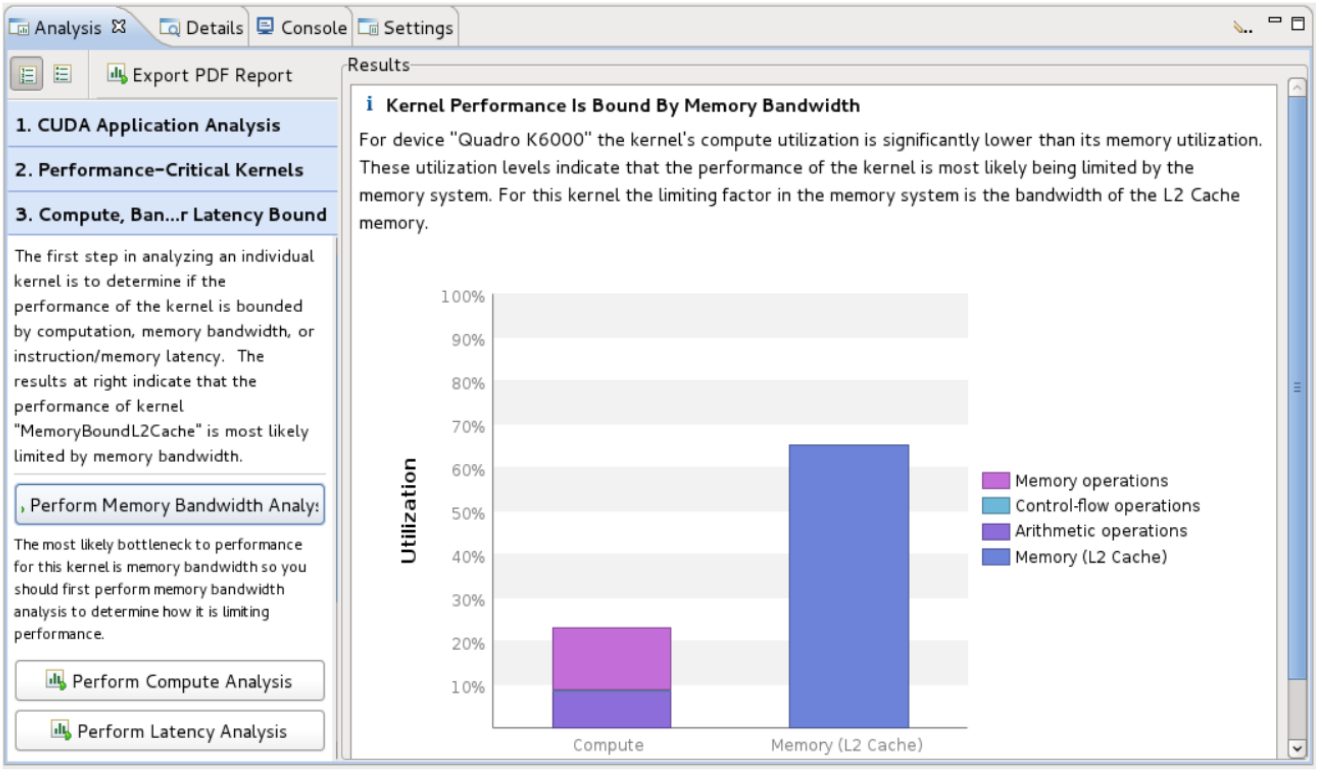


- **Analysis Results:**

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.

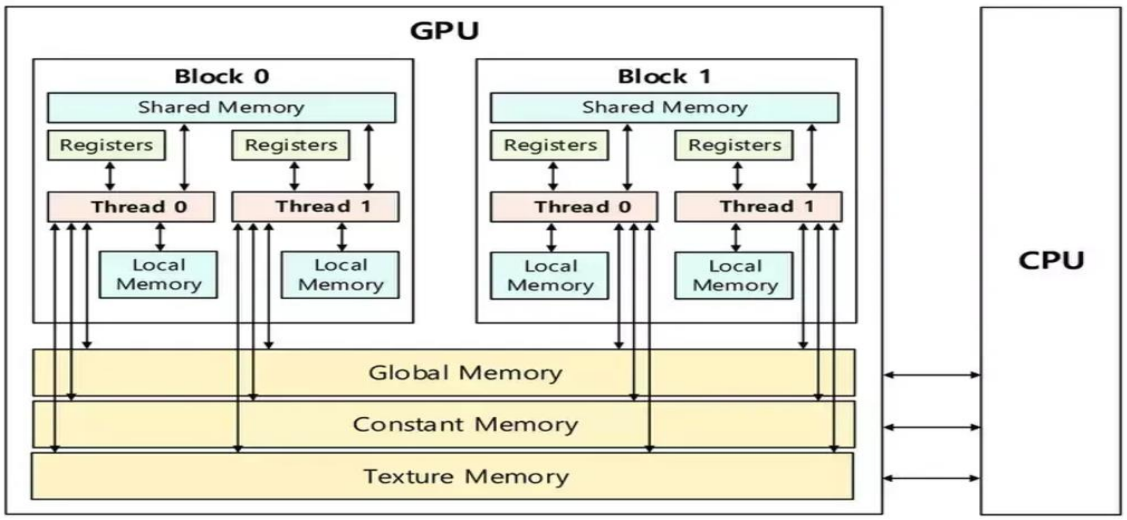
The following figure shows the analysis view in guided analysis mode. The left part of the view provides step-by-step directions to help you analyse and optimize your application. The right part of the view shows detailed analysis results appropriate for each part of the analysis.



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.

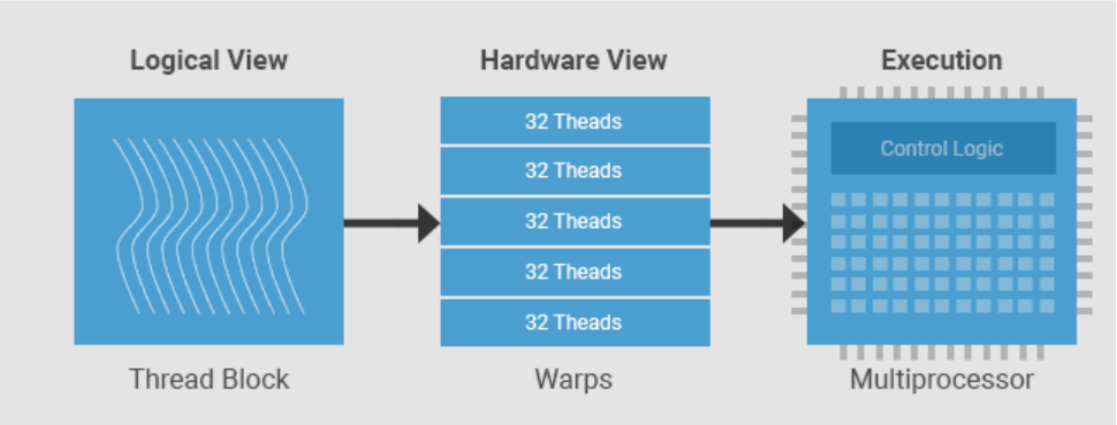


Global memory/device memory

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.



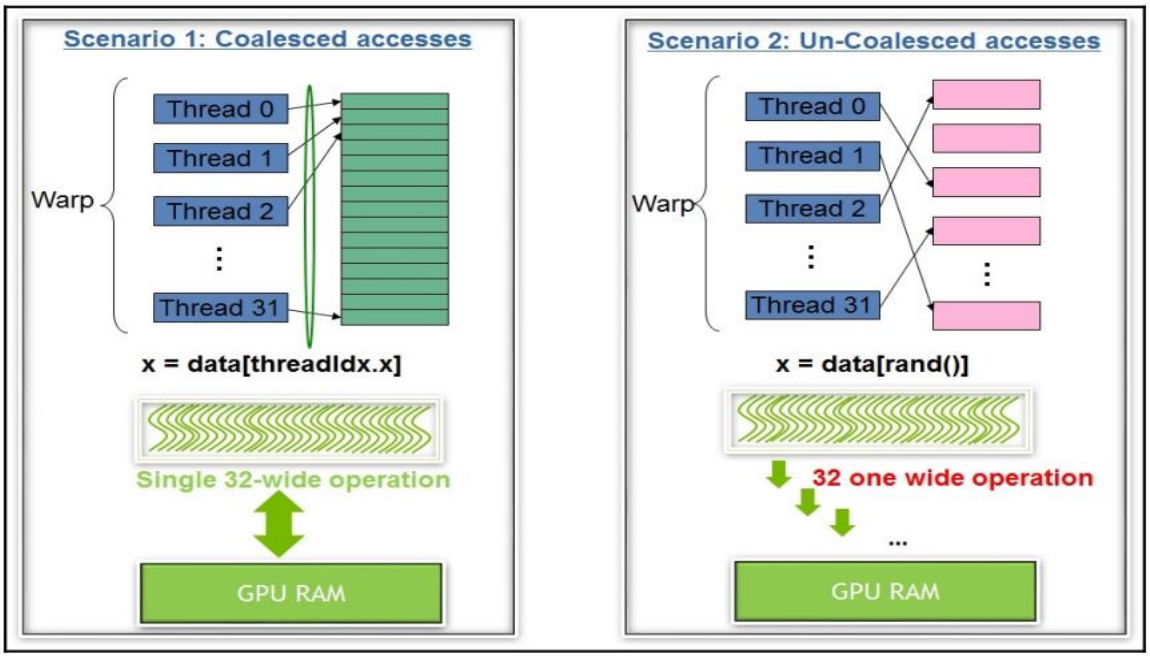
All of the threads in a warp execute the same instruction when selected. CUDA follows the Single Instruction, Multiple Thread (SIMT) model, that is, all threads in a warp fetch and execute the same instruction at one instance in time.

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:

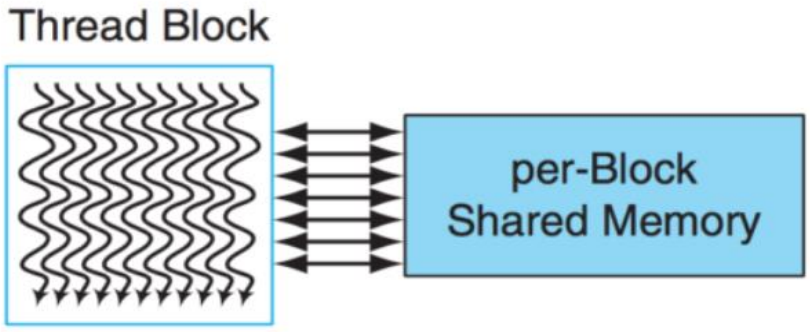
Coalesced access involves accessing consecutive memory locations by threads within a warp in a CUDA kernel.

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

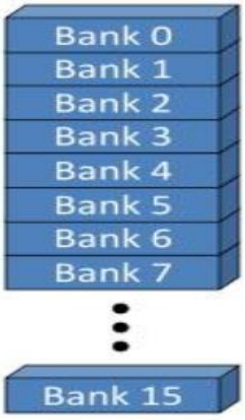


Shared memory

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.



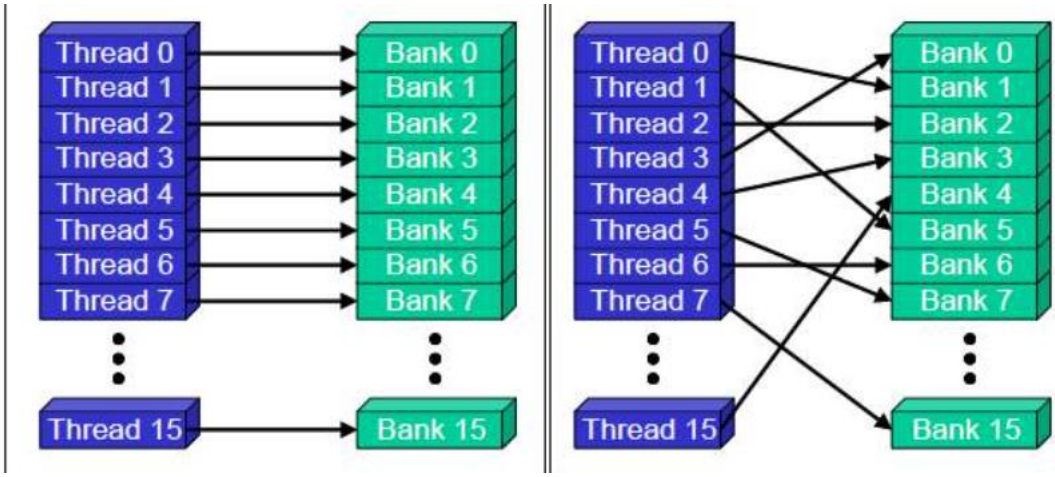
Shared memory is very fast (register speeds). Shared memory is used to enable fast communication between threads in a block. Shared memory only exists for the lifetime of the block. To achieve high memory bandwidth for concurrent accesses, shared memory is divided into **equally sized memory modules, 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.

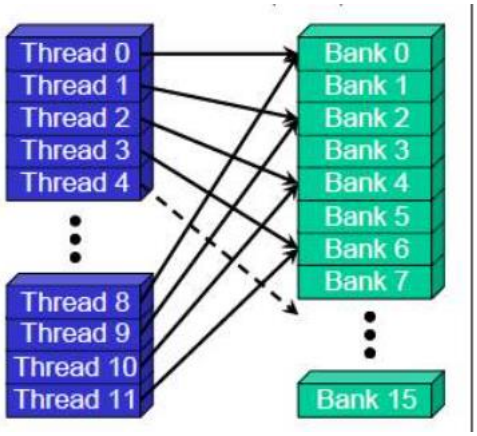
Bank conflicts can lead to serialized access, reducing the overall memory bandwidth and performance.

Example:



No Bank Conflicts

Bank Conflicts

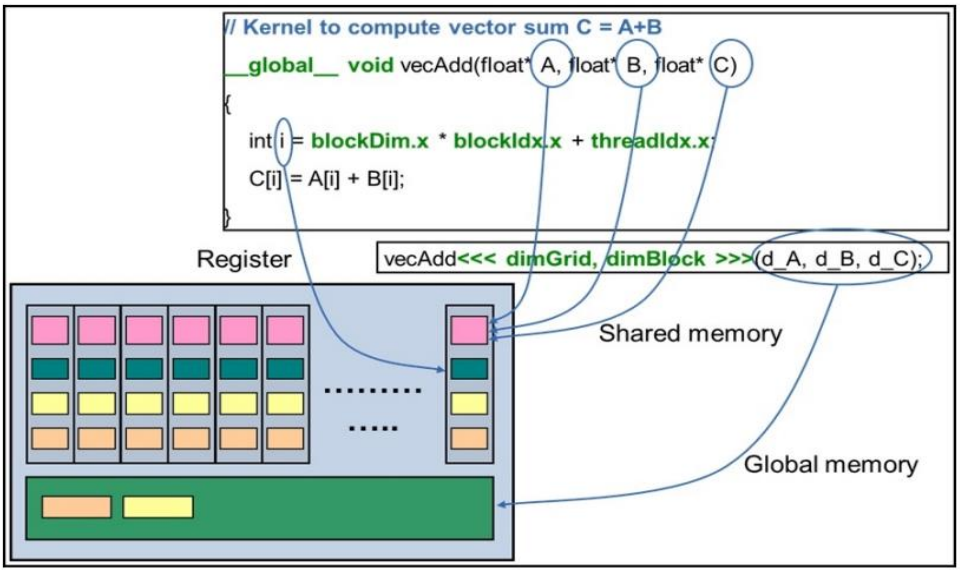


Registers

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

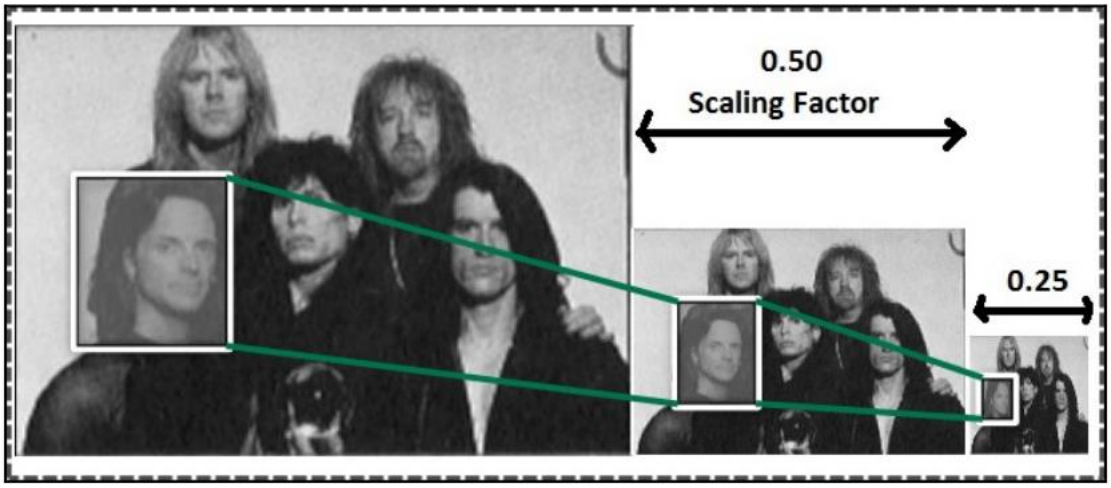
TOPIC 2: Read-only data/cache

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.

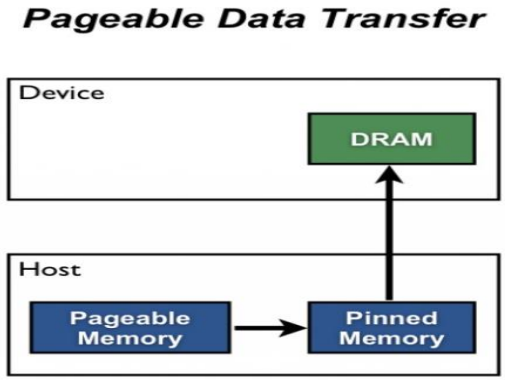
After the kernel execution or when you're done using the texture, it's good practice to unbind the texture.

Ex : cudaUnbindTexture(myTexture);

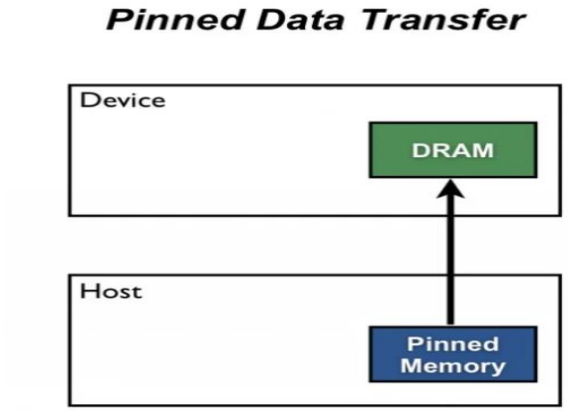
Pinned memory

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().



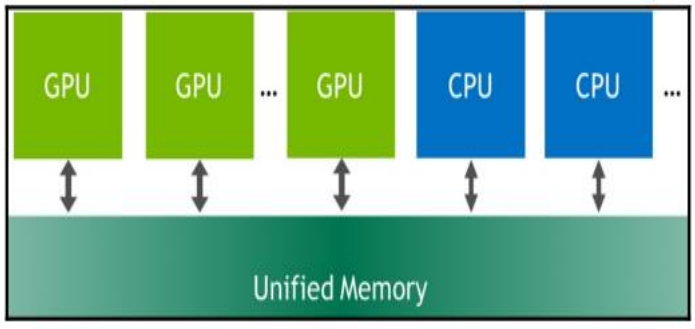
In the context of GPU programming, particularly with CUDA (Compute Unified Device Architecture) developed by NVIDIA, pinned memory is often used to enhance data transfer between the CPU (host) and the GPU (device). Pinned memory allows for asynchronous data transfers between the CPU and GPU, meaning 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:

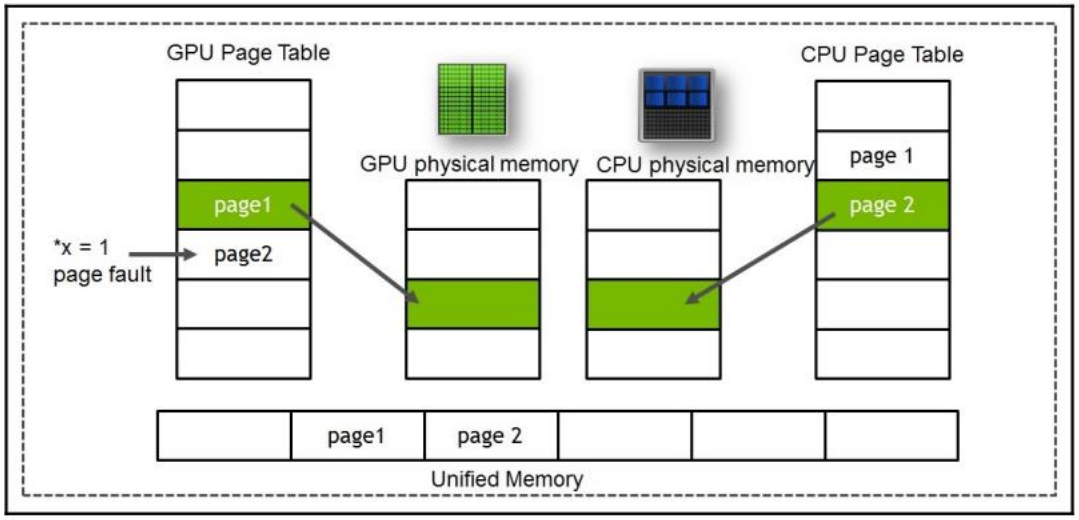


`cudaMallocManaged()` is a CUDA API function used for allocating managed memory in Unified Memory. cudaMallocManaged() does not allocate physical memory but allocates memory based on a first-touch basis. If the GPU first touches the variable, the page will be allocated and mapped in the GPU page table; 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

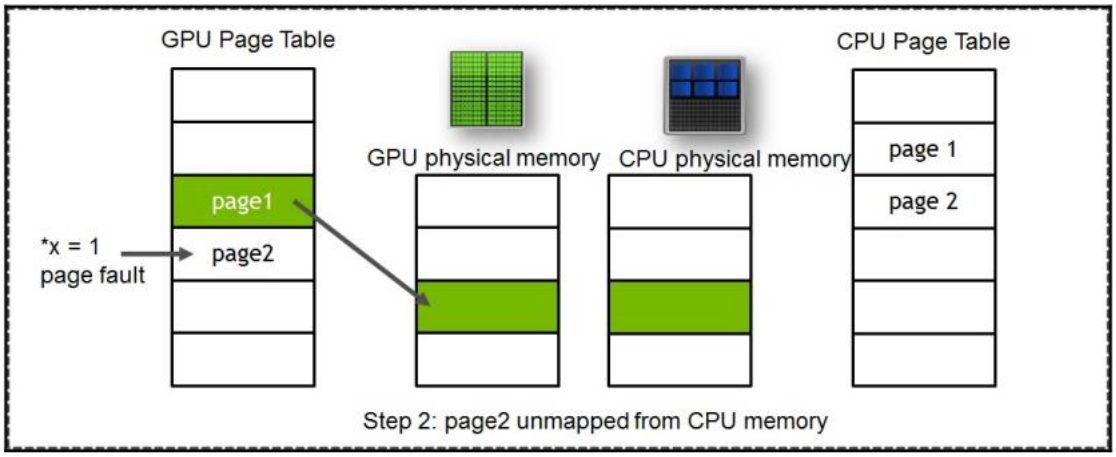
currently resident on the CPU or vice versa, a page fault is triggered, leading to the data being migrated to the appropriate memory space.

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

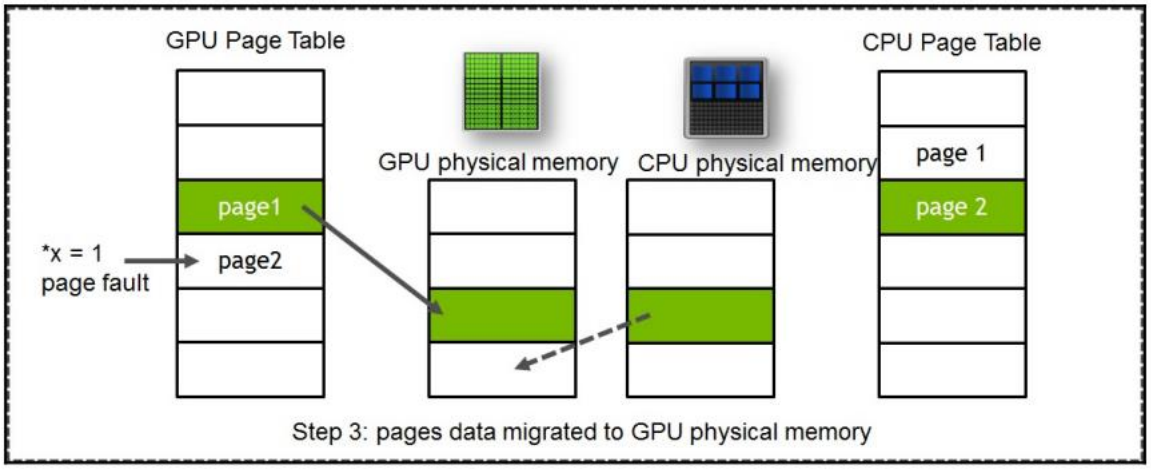
First, we need to allocate new pages on the GPU and CPU (first-touch basis). If the page is not present and mapped to another, a device page table page fault occurs. When *x, which resides in page 2, is accessed in the GPU that is currently mapped to CPU memory, it gets a page fault. Take a look at the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



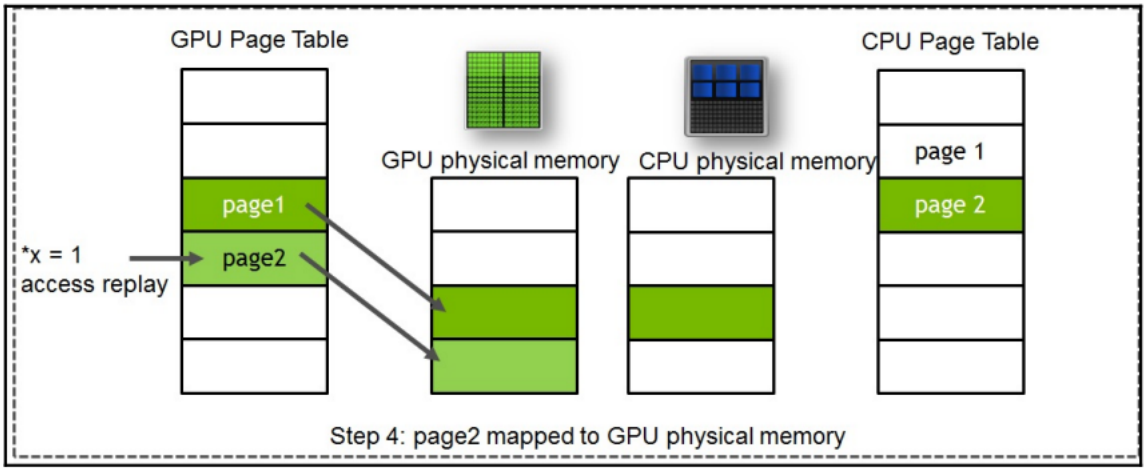
In the next step, the old page on the CPU is unmapped, as shown in the following diagram:



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.

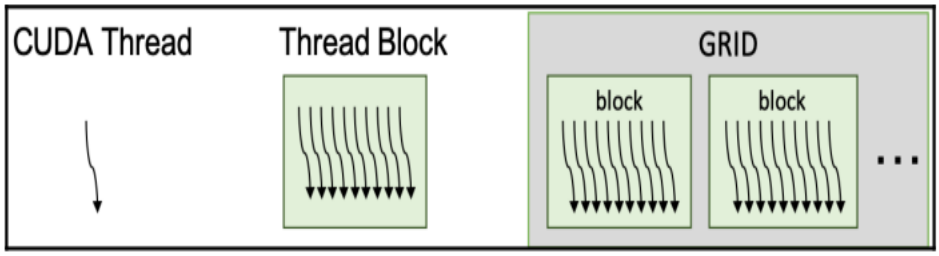
UNIT-IV

Topic 3: CUDA Threads & Occupancy

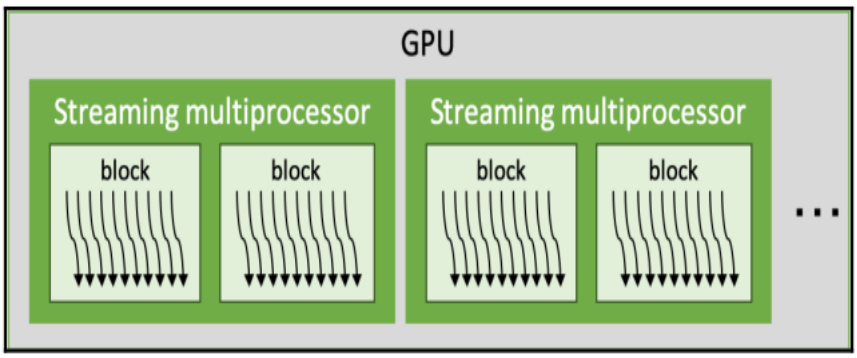
CUDA threads, blocks, and the GPU

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.

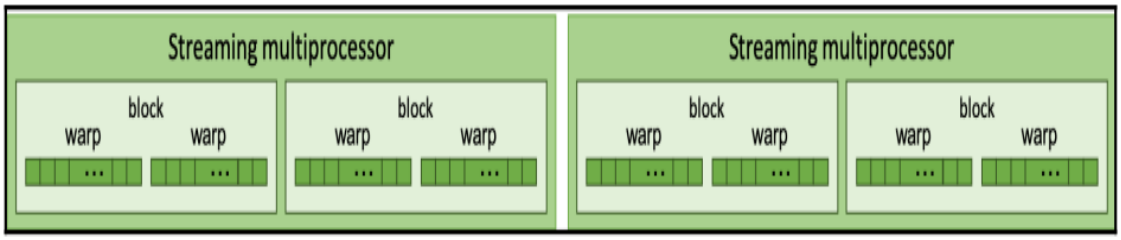
The following diagram shows their relationships:



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.

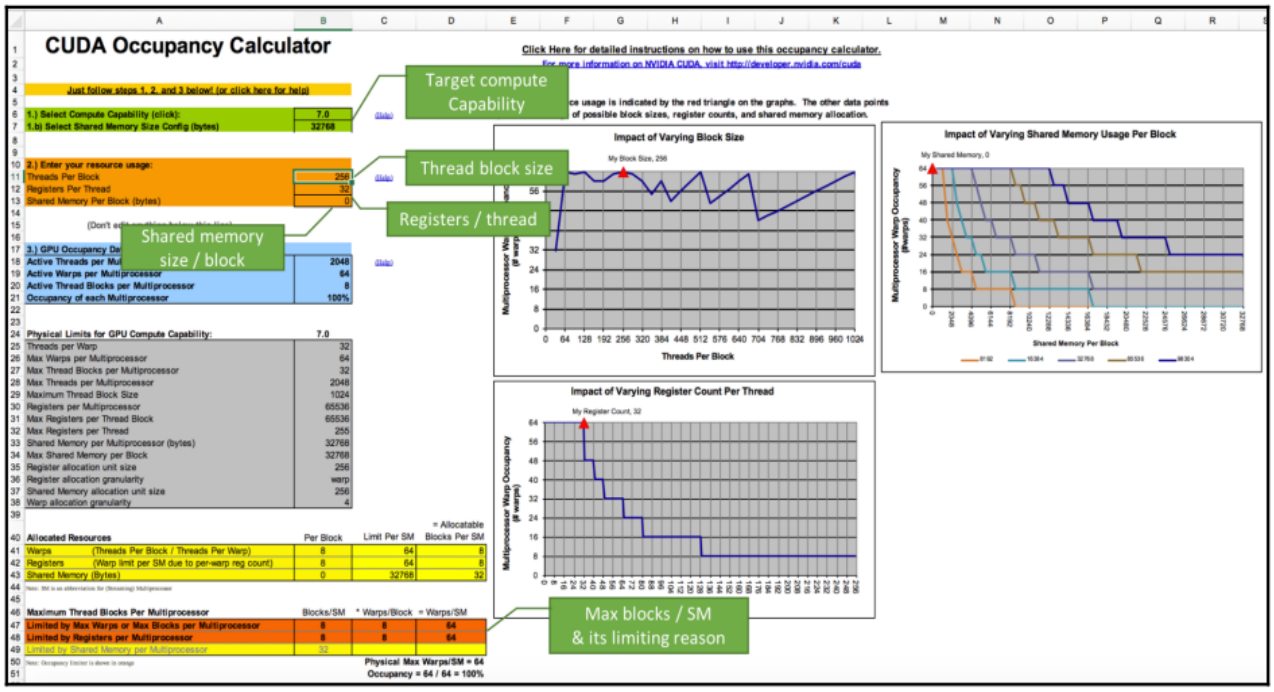
Developers can determine CUDA occupancy using two methods:

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:

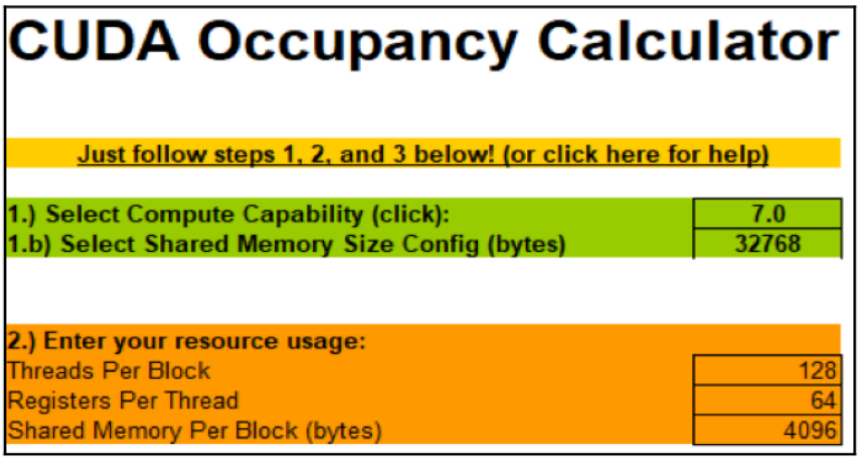
The following is a screenshot of the calculator



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:

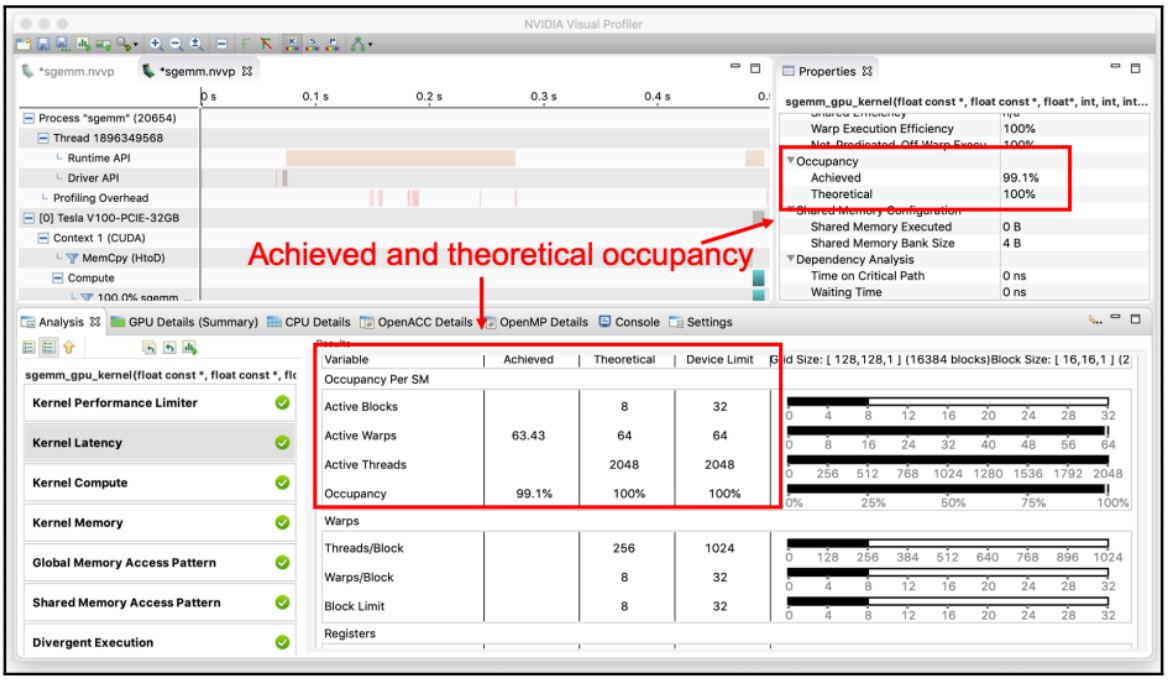
- GPU occupancy data (blue)
- The GPU's physical limitation for GPU compute capability (gray)

- 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.

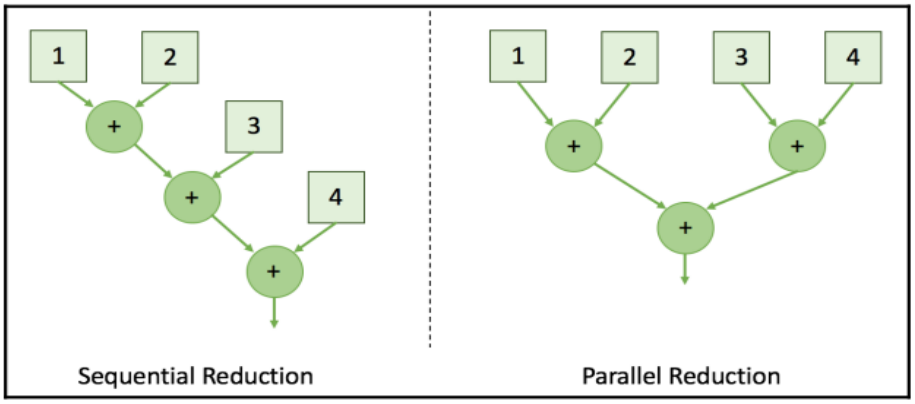


Unit-4

Topic 4: Understanding Parallel reduction

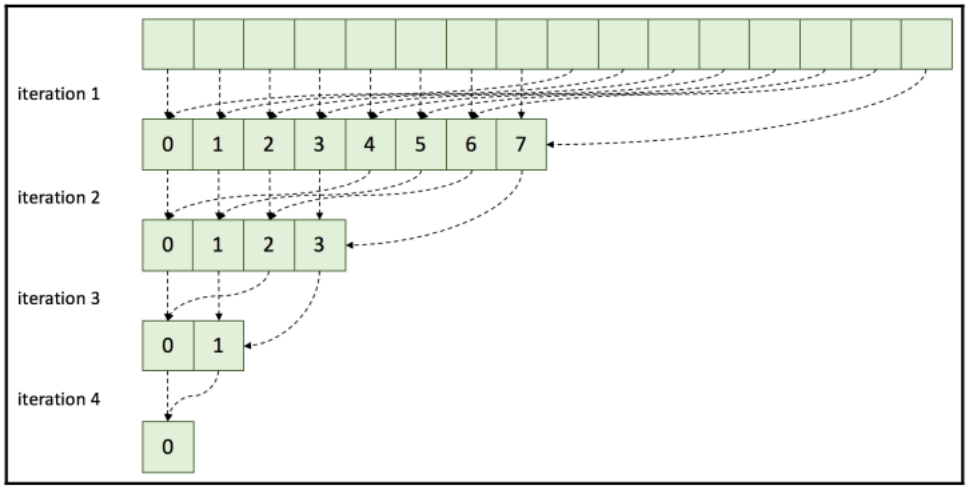
Parallel reduction is a common parallel computing technique used to efficiently compute the sum (or other associative operations) of a large set of values. In CUDA, parallel reduction is often employed to take advantage of the parallel processing capabilities of GPUs. The basic idea is to break down 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

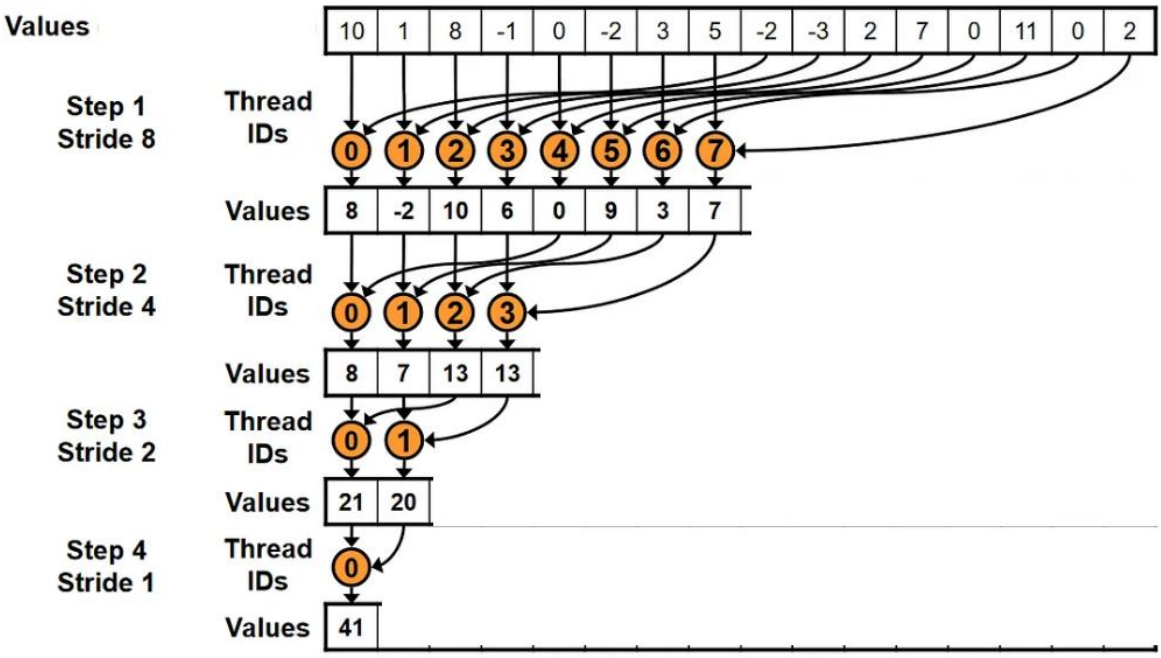


The stride for a vector is an increment that is used to step through array storage to select the vector elements from an array.

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.

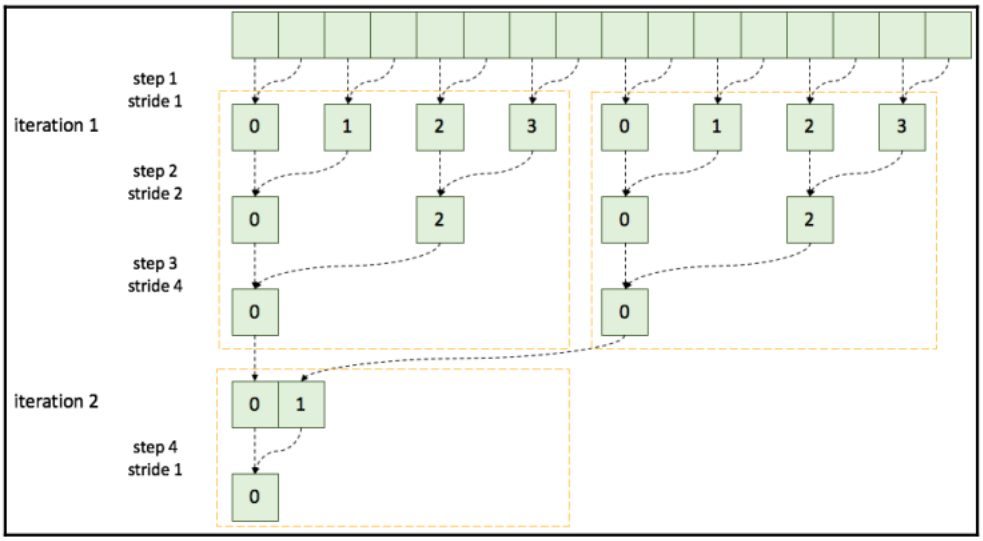
Example



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.

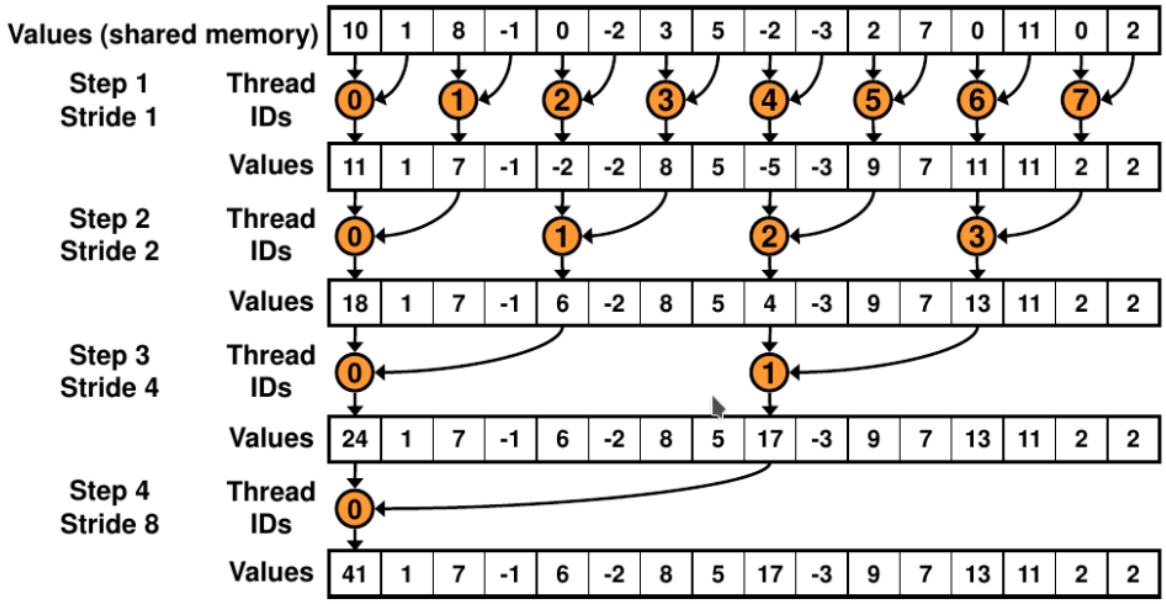
- **Reducing kernels using shared memory**

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:

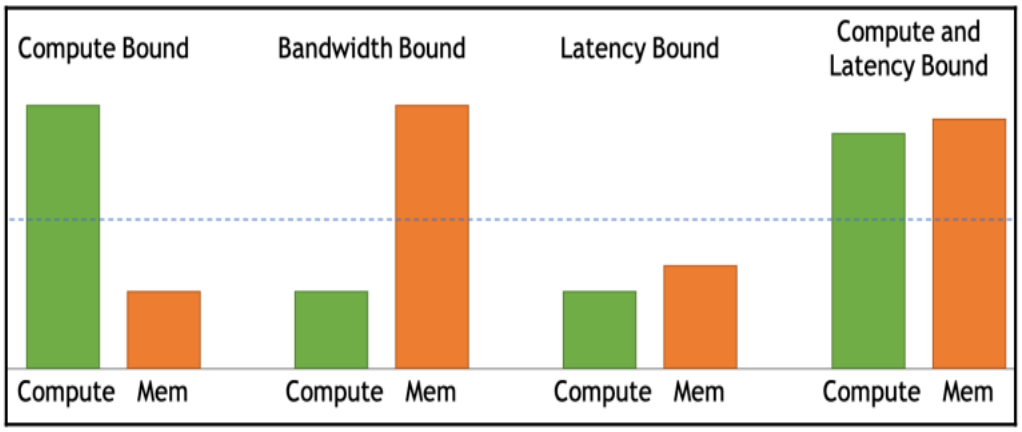
- Utilizes fast shared memory for intermediate results, reducing global memory transactions.
- More complex due to the need for synchronization and a second-level reduction.
- Can be more efficient for large datasets due to reduced memory traffic.

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.

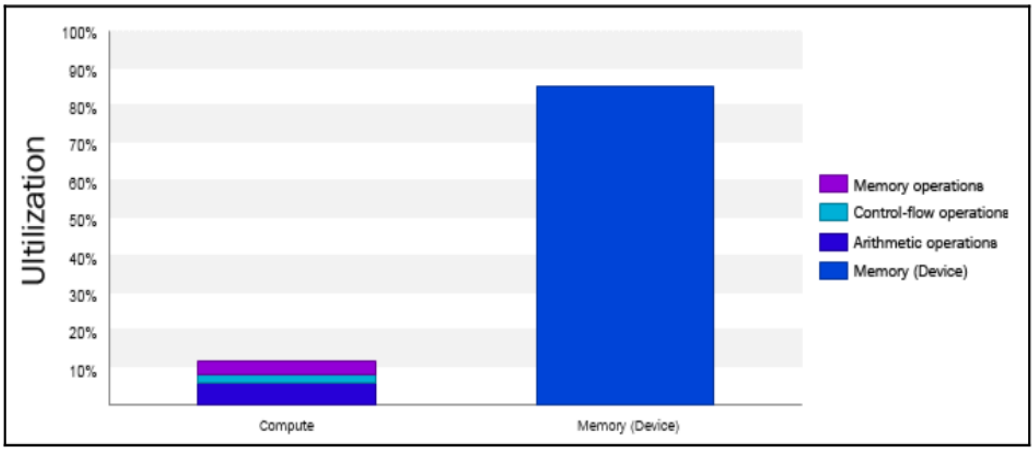
Simulation and Analysis: Compute-bound aspects are crucial for simulations and data analysis.

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

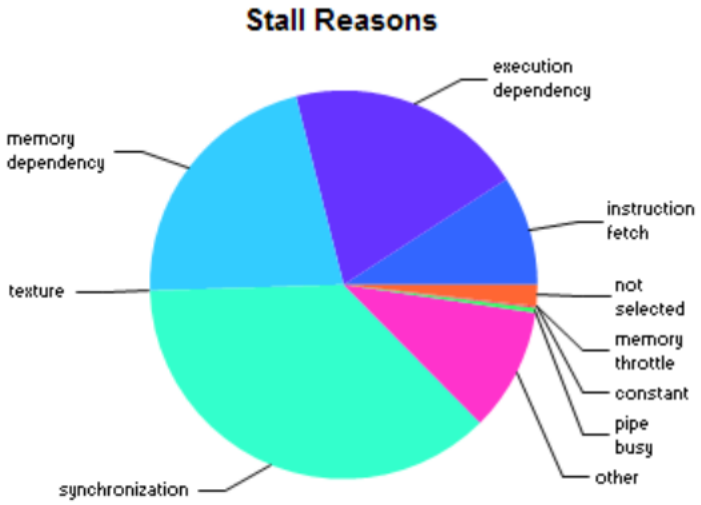
The following chart from NVIDIA profiler, shows the global memory-based reduction's performance limiter:



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