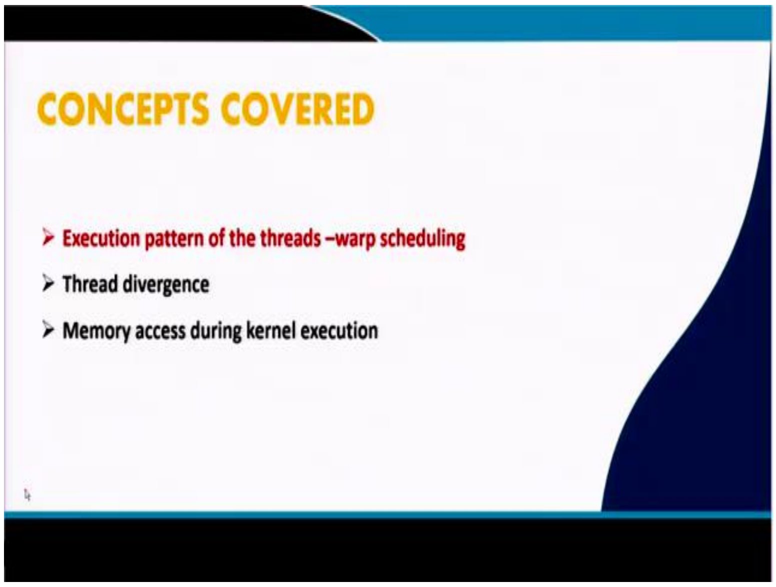
High Performance Computing for Scientists and Engineers

Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Module - 04
GPU Computing
Lecture – 37
Thread execution in CUDA program (continued)

Hello this is the MOOCs course on High Performance Computing for Scientists and Engineers. We are discussing module 4 on GPU Computing, and continuing with previous lecture on Thread execution in a CUDA program. We have looked at thread scheduling, we will see some more details on thread scheduling and synchronization and then we will see how memory access is done by different threads in a CUDA program.

(Refer Slide Time: 00:50)



CONCEPTS COVERED

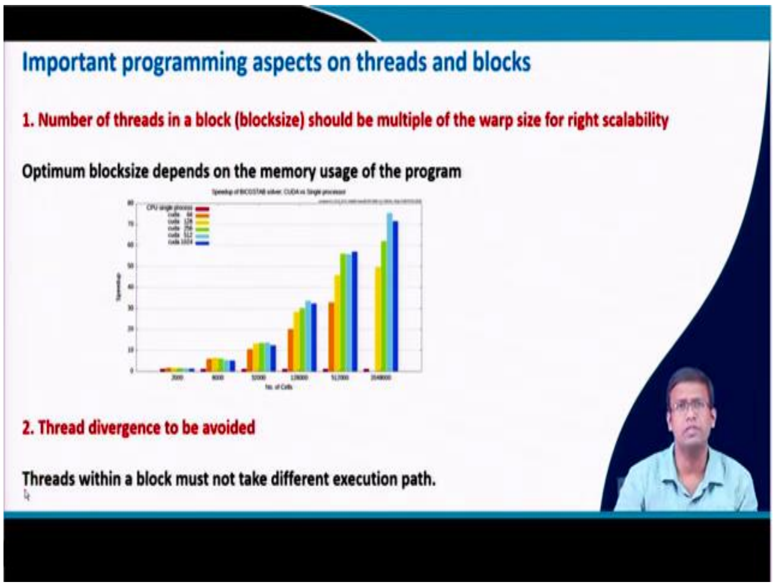
- Execution pattern of the threads –warp scheduling
- Thread divergence
- Memory access during kernel execution

So, you looked about execution patterns of the thread and which involves scheduling at the warp level. Warps are collections of 32 threads in the modern GPUs, which are executed at one go in the streaming multiprocessor. So, every time we launch a kernel, it launches blocks of threads and the collection of the blocks of threads is called a grid, inside the grid there are blocks, inside blocks there are threads.

Each block of thread goes to one streaming multiprocessor and there can be more blocks going to one streaming multiprocessor; but one block will as a total will go to one streaming multiprocessor and the threads inside that block will be executed in that particular streaming multiprocessor. At one go 32 threads among them are chosen and this number is chosen continuously as per the thread id and these 32 threads are called the warps which are executed at one instance.

Once this warp goes to the streaming multiprocessor, 32 codes are allotted from them, they are executed. As soon as they are being executed, during this execution they encounter some of the latency for searching data or for something else; the next warp is activated and that is when it starts executing. Once the first warp latent stage is over, the context goes back to that warp and it is executed again. In that way multiple warps are executed with some dynamic scheduling within a streaming multiprocessor and that is how a large number of threads much more than the number of codes can be executed in GPU with good scalability.

(Refer Slide Time: 02:44)



Important programming aspects on threads and blocks

1. Number of threads in a block (blocksize) should be multiple of the warp size for right scalability

Optimum blocksize depends on the memory usage of the program

2. Thread divergence to be avoided

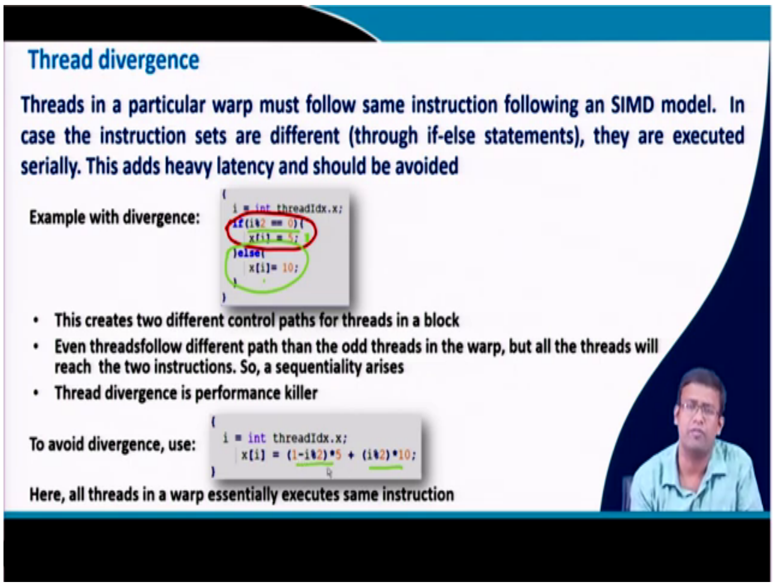
Threads within a block must not take different execution path.

Well, and what we have seen from this scheduling aspect is that as one warp is active at one particular instance; all the threads of the block will be executed as the group of warps, as a collection of warps. Therefore, the block size must be a multiple of warp size .We have seen that there is for one particular case, there is a particular block size which gives better performance.

So, depending on block size, performance also changes; because thread scheduling and latency hiding varies. So, the optimum block size must be multiple of warp size; if the block size is not a multiple of warp size, there will be serious performance degradation. Also, optimum block size depends on memory access patterns.

So, therefore, the same GPU code with changing the block size and the number of blocks inside the grid, we can get different performances and there is an optimum performance level. Thread divergence has to be avoided which tells that threads within a block must not take different execution paths.

(Refer Slide Time: 04:02)



Thread divergence

Threads in a particular warp must follow same instruction following an SIMD model. In case the instruction sets are different (through if-else statements), they are executed serially. This adds heavy latency and should be avoided

Example with divergence:

```
{
  i = int threadIdx.x;
  if(i%2 == 0){
    x[i] = 5; }
  }else{
    x[i]= 10;
  }
}
```

- This creates two different control paths for threads in a block
- Even threadsfollow different path than the odd threads in the warp, but all the threads will reach the two instructions. So, a sequentiality arises
- Thread divergence is performance killer

To avoid divergence, use:

```
{
  i = int threadIdx.x;
    x[i] = (1-i%2)*5 + (i%2)*10;
}
```

Here, all threads in a warp essentially executes same instruction

So, let us see, what is thread divergence? It is recommended that threads in a particular warp must follow the same instruction following a SIMD model. SIMD single instruction multiple data model; that means they must have the same instruction and different threads will operate on different data elements and preferably that should be from a contiguous data set well whatever.

So, the instruction must be the same for all the threads in the warp, and this instruction will be executed in parallel in all of the threads. In case the instruction sets are different; some other threads are asked to do something else and that can be done through an if else statement. In case it is there, the first set of tasks will be operated in the designated threads; the other threads will be in a latent manner. The cores to which these threads are assigned go to an idle phase; this part of execution will be over by the active threads.

The next instruction set will be chosen and the previously active threads; because they are not supposed to do this next step, they will go to and in a latent manner. Therefore, if there are two different instructions set by if else statement, half of the threads are active at any point of time. Although warp scheduling tells us 32 threads are active, this is the unit of parallelization in GPU, 32 threads are active at any time. However, if else is given for 16 threads; 16 will be active, therefore performance will fall by at least a factor of 2.

Let us see an example that, if i is the thread id, if the remainder by dividing i by 2 is zero; that means thread id is or even, then x[i] particular location of the vector x of the memory x will be 5. If it is odd, then x [i] will be 10. So, when this will be executed; when the even threads will come, then the odd threads are in a dormant or latent mode. When the even threads warp will be over; then the odd threads will be active, but even threads will go to a latent mode and this is called thread divergence.

This will create two different control paths for threads in a block and always this is limited within a block. For the other block that is an independent thing; but the same thing will happen, because you are executing the same statement in all blocks. Even threads follow different parts than odd threads in the warp, but all threads will reach the same instruction. So, there will be a sequentiality of the instruction set.

First all threads will look into this instruction set and this instruction set. When the first set of all threads are looking into this instruction set, only these threads are doing this part; but everybody is looking into that.

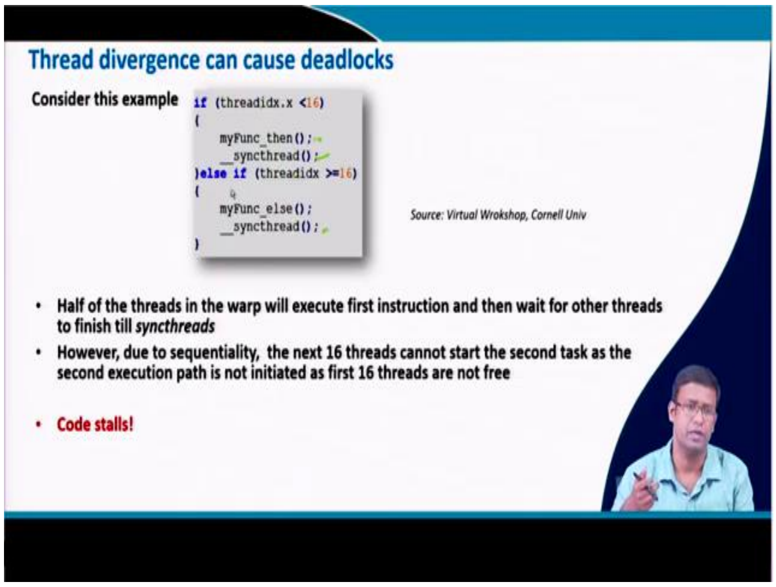
Now, because warp is scheduled together, the threads are active together, so everybody is looking into; they see if and if it is not so, they are waiting. When else is done, then everybody will look into this and only odd threads will look into that, the others will be waiting and this is a performance killer.

The programmer himself is doing something which is putting cores into idle stage and therefore, performance degrades. However, in many cases you cannot avoid if else statements; some threads will do something, some threads will do something else.

To avoid this, you do more computing in each thread and crunch more numbers; because this is a faster process, but do not put any thread into latent state by if else statements. So, write that for even there is some calculation for odd, there will be some other calculation.

So, you combine this if else statement and modify the program execution accordingly. Here all threads essentially execute the same state instruction; but they do calculations differently for even and odd that is not an issue, but none of the threads are waiting or in idle state here. So, that is how thread divergence has to be avoided.

(Refer Slide Time: 08:55)



Thread divergence can cause deadlocks

Consider this example

```
if (threadidx.x <16)
{
    myFunc_then();
    __syncthread();
}else if (threadidx >=16)
{
    myFunc_else();
    __syncthread();
}
```

Source: Virtual Wrokshop, Cornell Univ

- Half of the threads in the warp will execute first instruction and then wait for other threads to finish till *syncthreads*
- However, due to sequentiality, the next 16 threads cannot start the second task as the second execution path is not initiated as first 16 threads are not free
- Code stalls!

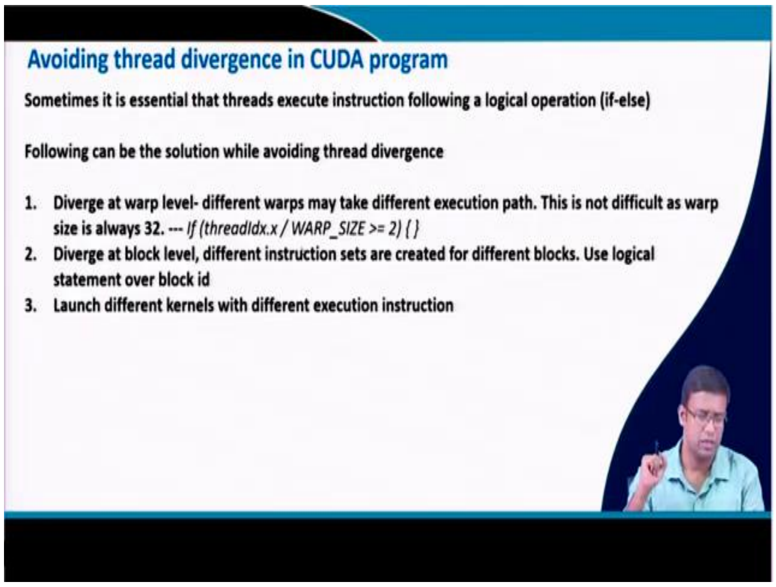
In some cases, it can cause deadlock; say we write some of the threads, thread id less than 16 to run some function, and thread id greater than equal to 16 to run some other function, and put a syncthread in this.

What a syncthread does? Syncthread is a synchronization barrier state; it will not allow the threads to do anything else till all other threads in the block have reached till this point. Now, threads less than 16 will operate this; threads greater than equal to 16 will not even come up here, but the threads less than equal less than 16 are waiting here, they cannot do anything else.

Therefore, it will go to a deadlock system and the code will stall; half of the threads in the warp will execute first instruction and wait for the others to finish till sync threads. But others couldn't finish that, because others are here; others are here cannot operate it till this loop is over, there is a sequentiality in operation due to thread divergence. The next 16 threads cannot start the second task; because the second execution path, the else if has not been activated.

Therefore, the code stalls. So, these are something which has to be avoided; that means if else using thread ids inside a kernel is a strict no. If you do that, your performance will be degraded.

(Refer Slide Time: 10:26)



Avoiding thread divergence in CUDA program

Sometimes it is essential that threads execute instruction following a logical operation (if-else)

Following can be the solution while avoiding thread divergence

1. Diverge at warp level- different warps may take different execution path. This is not difficult as warp size is always 32. --- *If (threadIdx.x / WARP_SIZE >= 2) { }*
2. Diverge at block level, different instruction sets are created for different blocks. Use logical statement over block id
3. Launch different kernels with different execution instruction

However, there are certain cases when you cannot avoid that; when you are writing a complex program, sometimes you cannot avoid using logical statements like if else, but you have to avoid thread divergence. So, you should not put this if else statement based on the thread ids within the kernel; rather few things you can do, you can do at the warp level, you see the warp is scheduled to do some warp at one go.

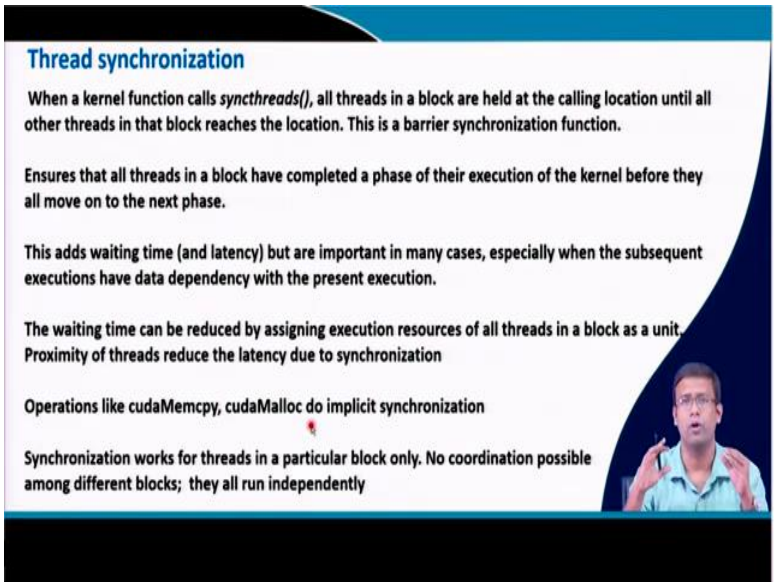
So, instead of odd even branching, say if you club all the threads in the first up for thread id less than the warp size in to do something else, thread id greater than warp size to do something else. So, two different warps will take two different execution paths.

So, instead of doing if else statement in the thread level, do it in the warp level; diverge at the warp level, different warps are doing different warp which is possible. You can diverge at the block level; using block id say that, some block will operate on something some, the other block will operate take the other execution part.

Or you can launch different kernels, identify the warp which is say for even you have to do something; write a kernel combine them, such that all the even threads now become threads of one particular kernel, renumber the threads and rearrange them.

So, launch different kernels. By these few methods, you can avoid thread divergence and improve the performance. Thread divergence kills the performance at least by the factor of the number of divergent paths, different execution paths or if else statements inside the kernel function.

(Refer Slide Time: 12:14)



Thread synchronization

When a kernel function calls *syncthreads()*, all threads in a block are held at the calling location until all other threads in that block reaches the location. This is a barrier synchronization function.

Ensures that all threads in a block have completed a phase of their execution of the kernel before they all move on to the next phase.

This adds waiting time (and latency) but are important in many cases, especially when the subsequent executions have data dependency with the present execution.

The waiting time can be reduced by assigning execution resources of all threads in a block as a unit. Proximity of threads reduce the latency due to synchronization

Operations like cudaMemcpy, cudaMalloc do implicit synchronization

Synchronization works for threads in a particular block only. No coordination possible among different blocks; they all run independently

Now, we have seen earlier thread synchronization, sometimes it is important to synchronize the threads or put up a barrier on the thread activity. Because, say, we can have multiple blocks, there can be some data dependency among the blocks. We can launch multiple kernels, the kernel call is not a synchronized call; once you call the kernel, control goes back to the CPU, it again launches the next kernel, but there can be data dependency among the kernels.

Therefore, if you do not synchronize them, especially for case of data dependency the result can be wrong. So, many times you need to put a barrier and this sync thread is a synchronization command.

It ensures that all the threads in that particular block have completed the phase of that execution, where you put sync threads up to that all the threads in the block have completed that; that means, though you are scheduling at warp level, all the warps are activated till all the threads in the block have done till that particular job. Then the control can go to the next step, then the next executions can be started.

This adds waiting time as well as latency; because the one warp is finished, the thread readily cannot pick up the next warp or the next kernel cannot be launched, there is a waiting time. But it is important in many cases, especially when there is data dependency from the previous and new execution.

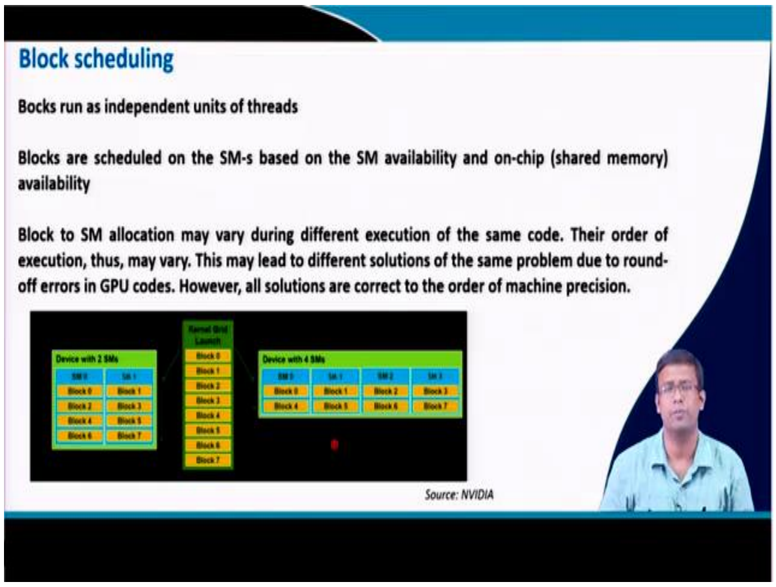
However, this waiting time can be reduced if we allocate threads in a block as a unit in a very contiguous manner that these threads will look into a contiguous set of memory; because threads are picking up memory location as a chunk. If the memory allocation and the thread allocation on the warp has very close proximity; then all of the threads are finishing their job almost at the same time, there is no wait time for one thread for the others. The wait time is anywhere less; so even if you put Syncthreads, they are just synchronizing it, the wait time becomes less.

So, utilizing proximity of threats both in terms of execution and memory, ordering the threads correctly, pointing them to the right memory in an ordered manner; the synchronization time can be reduced. Some operations like cudaMemcpy, cudaMalloc when we write these functions, they are implicit, do implicit synchronization; though we do not need to write sync threads, these operations will implicitly include a sync thread command.

So, all the threads will wait till these operations are finished. So, they are also synchronizations operations. Synchronization works for threads in a particular block only; this is important that we are launching many blocks, but synchronization is done only within a block.

No coordination is possible among different blocks; these blocks run independently; the multiple blocks can be scheduled to a particular streaming multiprocessor. But there is no synchronization among the blocks; at the block level they are running independently cannot be synchronized.

(Refer Slide Time: 15:37)



Block scheduling

Bocks run as independent units of threads

Blocks are scheduled on the SM-s based on the SM availability and on-chip (shared memory) availability

Block to SM allocation may vary during different execution of the same code. Their order of execution, thus, may vary. This may lead to different solutions of the same problem due to round-off errors in GPU codes. However, all solutions are correct to the order of machine precision.

Source: NVIDIA

How are the blocks scheduled? Blocks run as independent unit sub threads; blocks are scheduled on a streaming multiprocessor based on the availability of the streaming multi-processor. So, there were 80 streaming multiprocessors, but more blocks, 80 will go to 80 different streaming multiprocessors.

Then how are the how many cores are available on the streaming multiprocessor based on that more blocks will go to that streaming multiprocessor; also, how the on-chip memory is available that is another factor. Block to SM allocation may vary differently during different execution of the same code and in different GPUs also.

So, same code how block and streaming multiprocessor mapping will be done is kind of arbitrary. Therefore, the order of execution of different blocks; first block will be operated may be at the 7 th streaming multiprocessor; 31 st block may be at the fifth streaming multiprocessor and 130 th block may go to second streaming multiprocessor or it might go another streaming multiprocessor. There can be random arbitrariness in the operation of the blocks, and this creates an interesting situation that the entire job is being executed by threads and these threads are going to block sub threads in different grids. Different groups of threads or different blocks are executed in different order, because blocks are operated in different order.

We do round off error in a calculation; sometimes the calculation rounds off error has some impact on the final solution. As the operation of the blocks are different, the piling up of a round off error becomes different; therefore, different execution in the same GPU will have different amounts of round off error, because the order of execution of the blocks will change during the different instances of the same code. So, you will get different results.

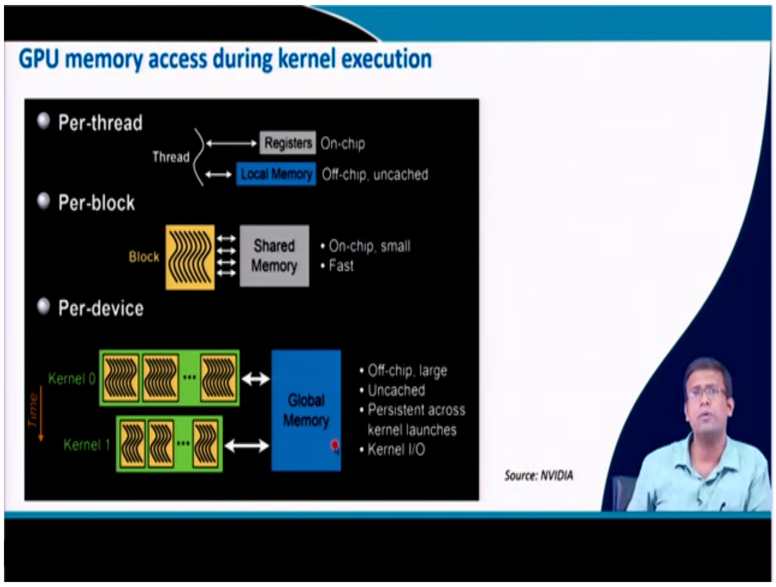
However, these differences are only due to round off error which is less than the machine's machine accuracy or machine precision. So, you will give your solutions will be correct to the order of machine precision; but the round off error makes them different.

So, correct, but different solutions you will get a different run of the same coding GPUs, especially when you run a different GPU also. But all these are correct, the only difference is in terms of the round off error, which is anyway part of numerical calculations. We can see that if you have a device with 2 streaming multiprocessors, there is an allocation of blocks; in case you have a device with 4 streaming multiprocessors, there will be different allocation of blocks.

So, the operation will be very different. As multiple kernels can launch on the GPU; some of the streaming multiprocessor is taken by one previous kernel, then you will see a different allocation or different scheduling of blocks happening.

However, because blocks are populated over all streaming multiprocessors and they are keeping the streaming multiprocessor busy. In case you have blocks more than the streaming multiprocessor number that does not make a problem, which can create a problem is the thread level scheduling or warp level scheduling and that is why the block size has to be optimum which is giving right performance.

(Refer Slide Time: 19:12)


GPU memory access during kernel execution

Now, the next important thing is how memory access is done in GPUs? We have discussed registers, a large number of registers are available in GPU codes and each thread gets memory from the registers, which are on chip memory which are on the streaming multiprocessor. So, it does not need to read the memory from the device ram which is a slow process; because it is connected through an interconnect reading from device ram is slower.

But if that memory requirement is more than what can be supplied by the registers, then the threads read from the local memory and this is in the device RAM. So, a private memory is for all threads that are created in the device RAM; in case it requires more memory than the register and many times it requires so, and this reading becomes slower.

The threads itself can look into registers and local memory, but the block as a whole can look into shared memory which is a small memory 64 kilobyte, 96 kilobyte size of memory attached to the streaming multiprocessor directly. So, it is an on-chip memory, it is fast.

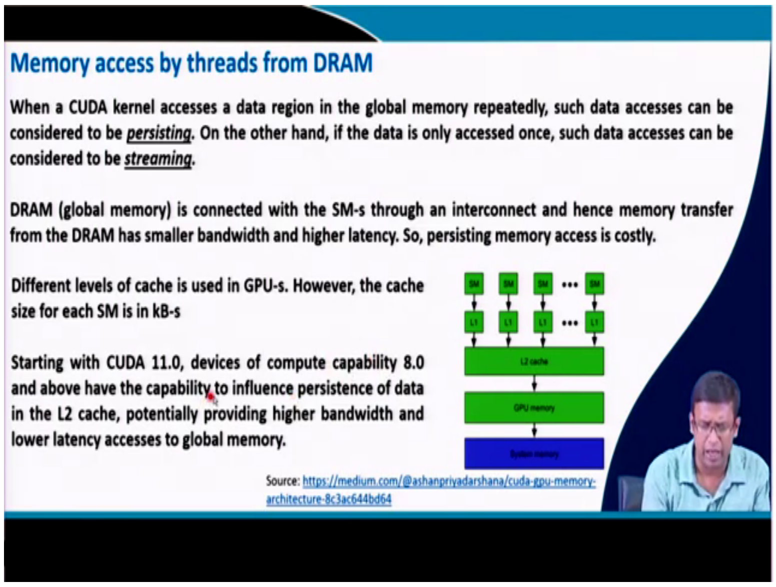
So, we were talking about two terms on chip and off chip; off chip means which is in the device frame or GPU RAM and the GPU RAM is connected via interconnect to the streaming multiprocessors; but on chip means which are directly on the streaming multiprocessors and can be read much faster.

So, off chip memory has a slower access, smaller bandwidth; on chip memory has faster higher bandwidth and fast access. The kernel, when the kernel is launched; the kernel communicates with the global memory.

So, whatever we are copying from the CPU, memory is copying into the global memory and the kernel reads from the global memory. This is usually uncached; but now from kernel to thread level, we can see there are L 1 and L 2 caches in modern GPUs. The kernel can read and write to the global memory, etcetera.

So, we can see that the global memory and local memory are from the device RAM. So, accessing global memory and local memory are slower compared to shared memory or register access; these accesses are faster, but these are small memory units, especially global memory is a large memory’s units.

(Refer Slide Time: 21:38)



Memory access by threads from DRAM

When a CUDA kernel accesses a data region in the global memory repeatedly, such data accesses can be considered to be *persisting*. On the other hand, if the data is only accessed once, such data accesses can be considered to be *streaming*.

DRAM (global memory) is connected with the SM-s through an interconnect and hence memory transfer from the DRAM has smaller bandwidth and higher latency. So, persisting memory access is costly.

Different levels of cache is used in GPU-s. However, the cache size for each SM is in kB-s

Starting with CUDA 11.0, devices of compute capability 8.0 and above have the capability to influence persistence of data in the L2 cache, potentially providing higher bandwidth and lower latency accesses to global memory.

Source: https://medium.com/@ashanpriyadarshana/cuda-gpu-memory-architecture-8c3ac644bd64

When a CUDA kernel accesses a data region in the global memory repeatedly, say it is doing a matrix multiplication; so many times, it needs to access elements of the column. Then this access is called a persisting axis. Many times, during the kernel execution that memory is being accessed. On the other hand, if the data is accessed only once and, in this data, comes and is loaded in the registers and it works, it is considered to be a streaming access.

Persisting access requires many accesses with the global memory and is a slower process, and adds more latency. DRAM global memory is connected to the SM-s through memory interconnects and hence memory transfer from the DRAM has lower bandwidth and higher latency.

So, persisting access is costly. Different levels of caches are used in GPUs; however, the caches size for each streaming multiprocessor cache is not big is of kilobytes, hundreds few kilobytes. So, we can see that because we have a persisting memory access which is costly there from GPU memory. So, this is the CPU memory which copies to GPU memory; from GPU memory to streaming multiprocessor and there are L2 and L1 cache.

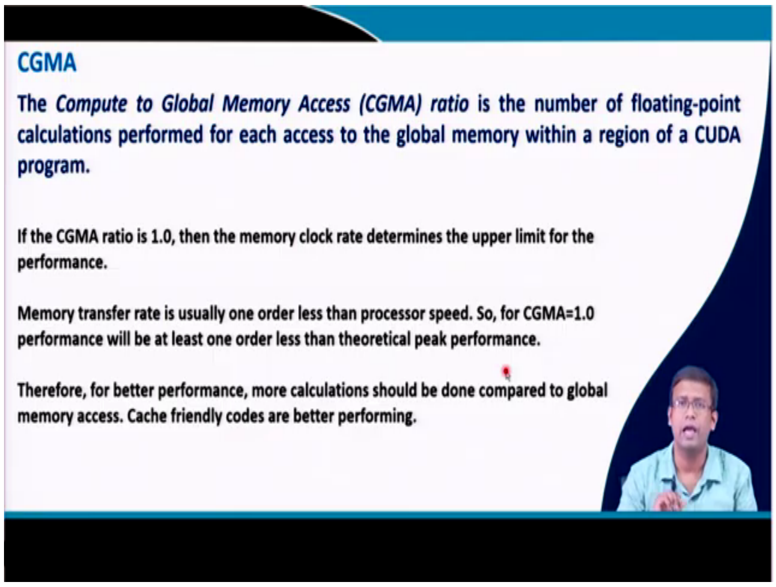
These caches are not large caches, small cache sizes; but with development of modern GPUs, the cache sizes are increasing. Starting with CUDA 11.0 devices of compute capability 8.0 and above have the capability to influence persistence of data in L2 cache, providing higher bandwidth.

So, one is that, making different cache for persisting and streaming memory and providing more spaces for persisting memory. As well as from compute capability 8, data , specially the persisting data is fetched and put into the L 2 cache, so that the read is faster and this adds to the less latency in access to global memory.

So, one important issue also comes here that is cache friendly programming; because data will be now pulled from the GPU memory and put into the L 2 cache. Earlier we had an idea especially for the older version of GPUs, there is very less amount of cache available.

But now as L2 cache and specially for persisting data, there is sufficiently large L2 cache; cache friendly programming is important, so that data can be perfected and resides in the cache. As soon as the streaming multiprocessors are accessing the data, it can directly look into the cache data. So, the bandwidth is not lost.

(Refer Slide Time: 24:39)



CGMA

The *Compute to Global Memory Access (CGMA) ratio* is the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

If the CGMA ratio is 1.0, then the memory clock rate determines the upper limit for the performance.

Memory transfer rate is usually one order less than processor speed. So, for CGMA=1.0 performance will be at least one order less than theoretical peak performance.

Therefore, for better performance, more calculations should be done compared to global memory access. Cache friendly codes are better performing.

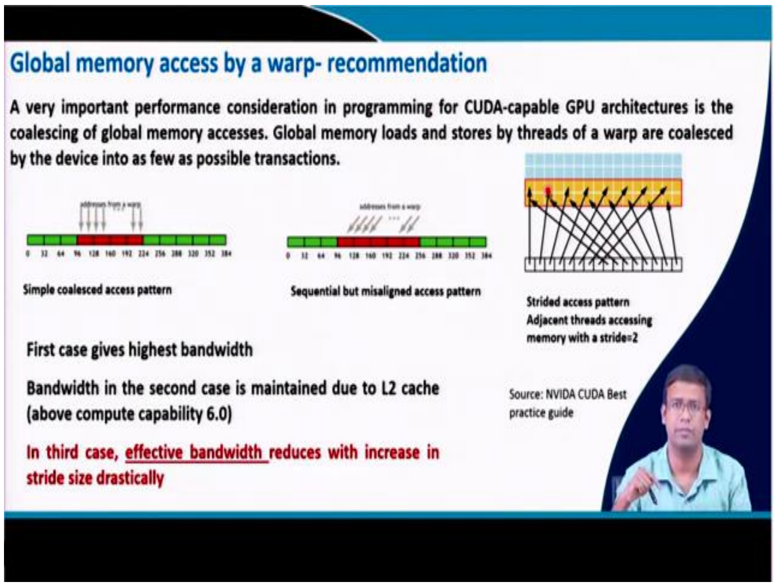
As global memory reads are slower; one important part is compute to global memory access or CGMA, that is how much calculations you are doing for each access to global memory.

How many calculations are done for each access to global memory? If you are loading many global memory data and doing small calculation; then the CGMA value is small. If CGMA ratio, CGMA ratio should be higher for good performance; if CGMA ratio is one, that means for one operation we are doing one memory access. Now, memory access is always slow; so, the computing speed is being limited by the memory access speed.

So, we should try to have higher CGMA access compute to global memory access that, for one global memory access, more computing is being done. Due to this CGMA which can get sometimes close to one even small; we do not get the theoretical peak performance, because theoretical peak performance is based on how many floating-point operations the core can do.

But now if you have one memory access per floating point operation; then your speed is limited by the floating-point operation and the speed reduces in that respect. For better performance, more calculation should be done compared to global memory access and cache friendly codes are better performing.

(Refer Slide Time: 26:03)



Global memory access by a warp- recommendation

A very important performance consideration in programming for CUDA-capable GPU architectures is the coalescing of global memory accesses. Global memory loads and stores by threads of a warp are coalesced by the device into as few as possible transactions.

First case gives highest bandwidth

Bandwidth in the second case is maintained due to L2 cache (above compute capability 6.0)

In third case, effective bandwidth reduces with increase in stride size drastically

Source: NVIDA CUDA Best practice guide

A very important performance consideration for programming CUDA capable GPU architecture is the coalescing of global memory accesses. Why? If you coalesce instead of a 2 D address; if you put into a 1 D contiguous array, coalesce the global memory. When the warp reads the memory, when one's memory element is read, the entire 32 threads requirement is coalesced and put into the cache.

So, all the threads can readily read the memory and the memory access time reduces, the cache can be used efficiently. So, coalesce memory access is an important aspect. Say if you are accessing memory, each thread is accessing a memory which directly points to the memory location. So, each 32 threads can go and utilize the entire bandwidth, fetch the 32 memory element data and can read it; but there can be offsets one thread may need more memory elements.

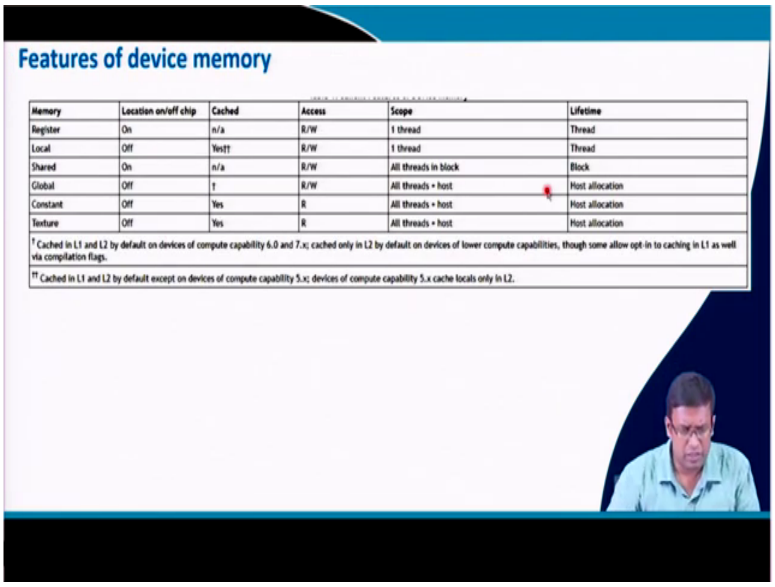
In case there is a misalign, but an offset issue; one thread when it is reading, it is fetching some data in the cache. So, in case there is a regular access, the cache is helping to fetch the next elements of the data and effectively it is getting the same bandwidth. In case the access is not regular, then there is a cache miss, and therefore, many times it has to access global memory and based on the CGMA value, the performance will fall down.

So, the first case gives highest bandwidth, the entire bandwidth can be used, 32 data can be read and accessed. In the second case, though there is a misalignment; whatever is read here, the next data is always sitting in the cache, the offset value is accommodated by the cache. So, effectively the bandwidth is the same; though it requires some off-set data, but that is already residing in the cache, it can utilize the cache probably.

In the third case, we can see that with increase in stride size; the first thread requires first data in first location, second thread requires data in the third location, in case we have more stride size; we need more offset data, the performance can degrade heavily.

We can see that, if data access is of this pattern, performance can fall down and we may get even worse performance than CPUs; because memory access due to smaller cache, memory access is lower here. In CPU you have a large cache; so even if you are using striding memory access, the entire data is sitting in the cache and getting better speed.

(Refer Slide Time: 28:43)



Features of device memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

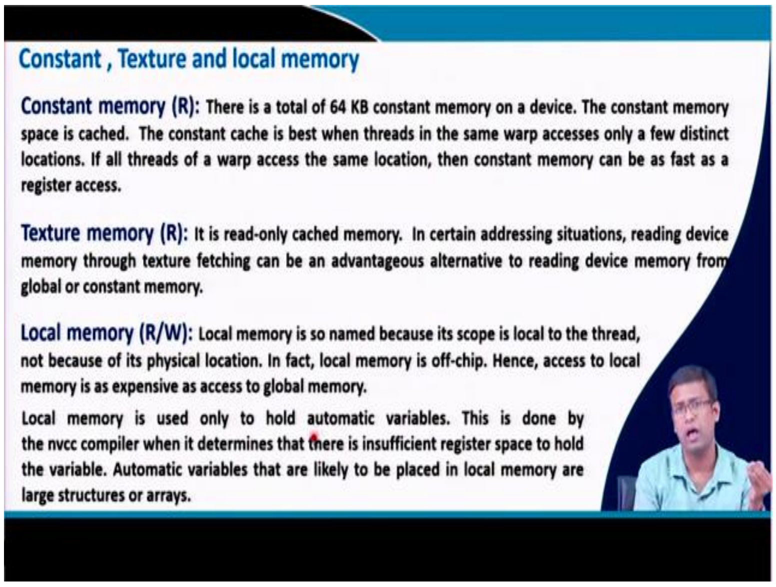
Now, these are the few memory units; register, local, shared, global, constant, texture. You can see from the name texture, it is more related with the graphics processing; because GPUs are graphics processing units, the names have some legacy of their original use.

The register and local memory are assigned to the threads; they are read, write memories, local is cache memory, register is not cache, then this is a very small amount of memory, you do not need to cache it.

The local is an off-chip memory; so, it's a slower memory, register is on chip faster. Shared memory is an on-chip memory which is not cached, which is a small memory; but all threads in the block can look into the shared memory. See even one warp write something in the shared memory that will reside in there and all the threads can utilize that.

Global is the main of chip memory which is now cached in L1 and L2 memory, earlier it was not cached, and these all are read write memory; there are some read only memory which are constant and texture and they are off chip, but these memories can be very first read by the streaming multiprocessor. So, there are some hardware benefits there.

(Refer Slide Time: 30:00)



Constant , Texture and local memory

Constant memory (R): There is a total of 64 KB constant memory on a device. The constant memory space is cached. The constant cache is best when threads in the same warp accesses only a few distinct locations. If all threads of a warp access the same location, then constant memory can be as fast as a register access.

Texture memory (R): It is read-only cached memory. In certain addressing situations, reading device memory through texture fetching can be an advantageous alternative to reading device memory from global or constant memory.

Local memory (R/W): Local memory is so named because its scope is local to the thread, not because of its physical location. In fact, local memory is off-chip. Hence, access to local memory is as expensive as access to global memory.

Local memory is used only to hold automatic variables. This is done by the nvcc compiler when it determines that there is insufficient register space to hold the variable. Automatic variables that are likely to be placed in local memory are large structures or arrays.

Let us look into it in detail. Constant memory is 64 kilobyte memory on the device; this space is cached and constant cache is best when threads in the same warp accesses only a few distinct locations.

If the all threads in the warp access the same location, this constant can be as fast as register access. So, though it is in the off chip, it can be very fast; if it has a constant value, all the warps are trying to thread in the warps are trying to read the same memory, it can be done very fast.

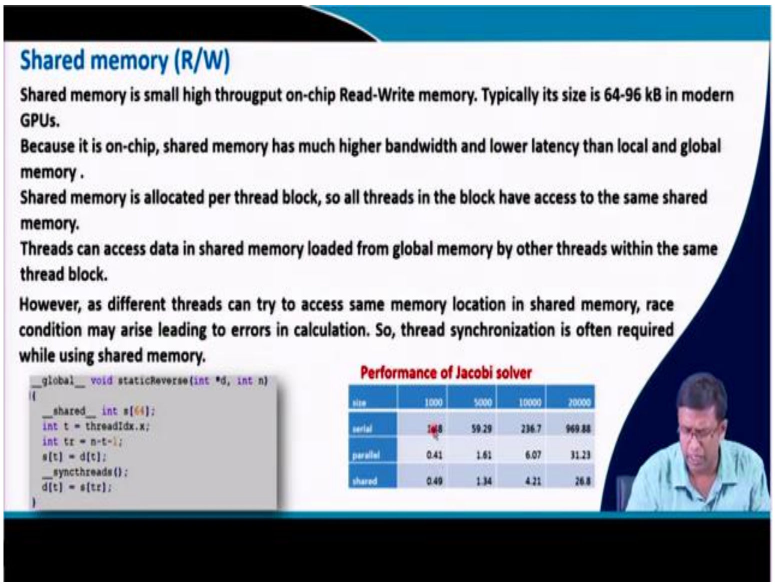
Texture is a read only memory, constant and texture both are read only memory. In certain situations, especially certain addressing situations; reading device memory through texture fetching can be advantageous, then reading device memory through global and constant memory. Some filtering aspects, special mostly for graphics purpose; we will not discuss here the texture memory from scientific computing perspective.

Local is named, because this is local; this is like a private memory to the thread and this does not physically, it is not local memory physically it is not sitting close to the thread, it is sitting in the device, close to the global memory. It is off chip and it is an expensive memory as global memory; but if a thread requires some memory private to it, which is more than the register memory, then it takes from the local memory which is in the device RAM itself and it is a slow memory.

Local memory is used only to hold automatic variables; the variables which have the scope of the thread do not live after the thread is over. This is done by the NVCC compiler, it determines if there is sufficient space in the register to hold the variable; if not then these are placed in the local memory.

So, we should also try to put this automatic or private variables small for each thread, so that local memory is not much required; because local memory is a slower memory, its access will degrade the performance.

(Refer Slide Time: 32:13)



Shared memory (R/W)

Shared memory is small high througput on-chip Read-Write memory. Typically its size is 64-96 kB in modern GPUs.

Because it is on-chip, shared memory has much higher bandwidth and lower latency than local and global memory .

Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory.

Threads can access data in shared memory loaded from global memory by other threads within the same thread block.

However, as different threads can try to access same memory location in shared memory, race condition may arise leading to errors in calculation. So, thread synchronization is often required while using shared memory.

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Performance of Jacobi solver

size	1000	5000	10000	20000
serial	[illegible]	59.29	236.7	969.88
parallel	0.41	1.61	6.07	31.23
shared	0.49	1.34	4.21	26.8

Shared memory is an important memory, it is this memory is shared by all threads in the block; this is the memory sitting on the streaming multiprocessor. So, its size is typically 64 to 96 kilobytes in V 100 96 kilo byte is total given and it can be configured for both; part of it will be shared and part of it will be L 1. Shared memory space shared by an L 1 cache.

However, because this is an on-chip memory it has much higher bandwidth and a lower latency than the local and global memory. This has the scope of the block; once the block is active, its threads can read the shared memory space.

If one warp can fetch data from global memory and put it into the shared memory and the all threads in the blocks are okay with that shared memory; then only with that read, the memory resides in the shared memory location and the next warps can utilize this memory and can be a very fast process.

Shared memory is allocated in the per trade block. So, all threads in the block have access to the same shared memory. Threads can access shared memory loaded from global memory by other threads within the same thread block.

So, once it is loaded in the shared memory, it resides till the block is active and the threads can operate over there. In many cases we can see, we can get optimized performance if we can use the shared memory; but its size is small, the maximum size is 96 kilobytes.

So, the memory requirement has also to be reduced for this activity. However, as different threads can try to access the same memory location in shared memory, there can be race conditions leading to errors in calculation.

Therefore, especially data dependency as it is available to the whole block; one warp is over and the next warp is launched, which is taking care of the next part of the execution and they are trying to look into the same data. However, some data is still not being written and therefore, there can be a race condition among the threads.

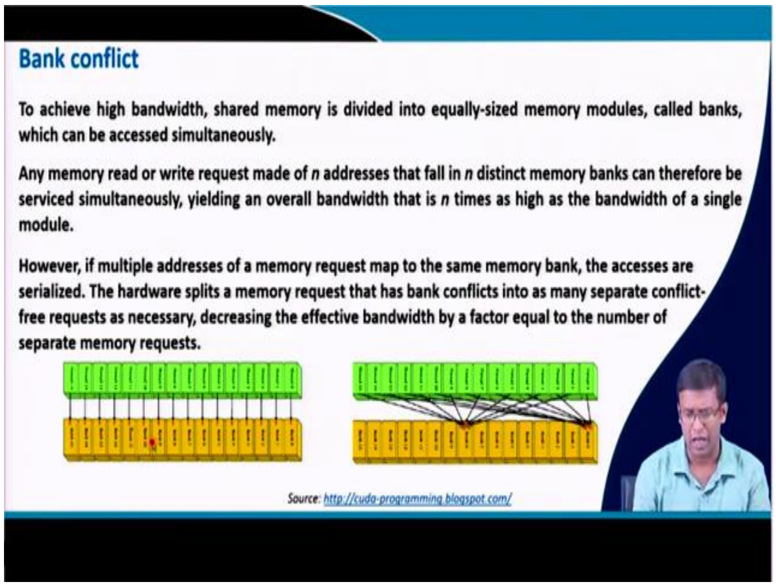
Therefore, syncthread is must ,especially if you have data dependency in a shared memory. So, say you see the example that, you are reading something in shared memory and wrote s [t] = d [t] and you will change the memory locations d[t] will be s [tr],and t is the thread id.

So, you put a sync thread here that, all the threads will finish up to this part and then this next will go. A shared memory has to be declared as shared in the kernel itself; if we looked into a Jacobi solver for matrix solutions and we can see that, when the matrix size is small, shared and without shared performance is similar.

Actually, without using shared memory, performance is better; because matrix size is small, so everything is fitting in the cache and only reading from cache is sufficient enough. But for larger matrices, when we are using a Jacobi solver at least 15, 20 percent improvement in speed happens when we use shared memory.

Because now, the entire matrix cannot fit in cache; so, you fetch the data required by the block and put it into the shared memory and the one warp will do that and the other warps can work on that, other threads can utilize that and operate first.

(Refer Slide Time: 36:01)



Bank conflict

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously.

Any memory read or write request made of *n* addresses that fall in *n* distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is *n* times as high as the bandwidth of a single module.

However, if multiple addresses of a memory request map to the same memory bank, the accesses are serialized. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests.

Source: http://cuda-programming.blogspot.com/

To achieve high bandwidth, shared memory is divided into equally sized memory modulus called banks, which can be accessed simultaneously.

So, shared memory has many equally sized memory modules or banks and each thread can directly access a bank and read from there. In the warp there are 32 threads; so, at a go they can read 32 bank locations and fetch memory from that, this increases the latency. There is no sequentiality in memory reading among the threads or also there is no race condition in memory reading; because 32 locations are directly read by 32 threads.

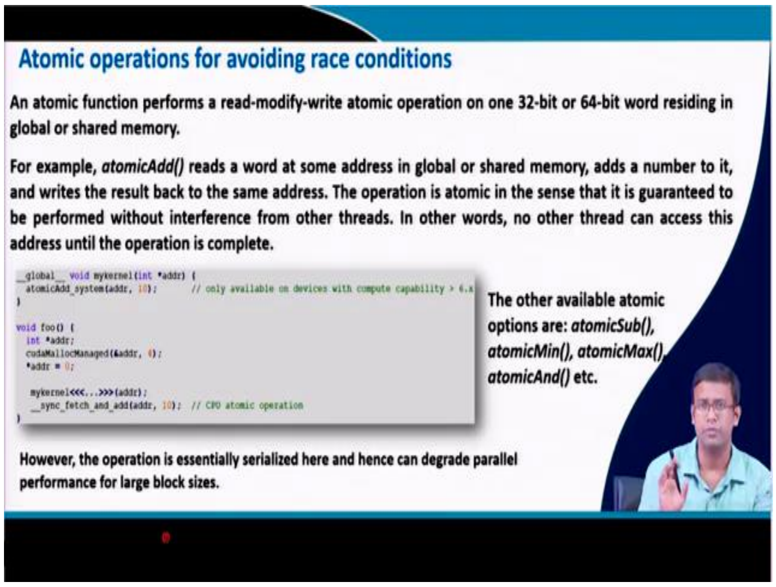
But now programmers have to be smart enough, so that each thread looks into a particular bank; a number of threads do not look into the same bank, there is no bank conflict. Any memory read or write request made to n addresses that fall in n distinct memory banks; therefore, can be served simultaneously, yielding an overall bandwidth which is n time faster than a single module. But now if multiple addresses of memory request map to the same memory bank. The accesses are serialized; if many threads try to read from the same memory bank, now this will be serialized. One thread can look into one memory bank; if many thirds are looking into the same memory bank, this will be serialized. Hardware splits memory requests that have bank conflicts into many separate conflict free requests as necessary and therefore, effective bandwidth reduces. We can see that this has no bank nor bank conflict. So, all the 16 threads here this is the previous GPU where there are 16 threads per warp, now there are 32 threads;

but all the 16 threads are readily reading from this memory, so the effective bandwidth is actually 16 times more.

However, the reverse condition happens here that all the 7, 8 threads read from the same bank. So, one by one the threads are reading their effective bandwidth is reduced by 8 threads here. So, it should not be done, thread should look into distinct memory locations, so that bank conflict is avoided.

They should not look into many threads, should not look into the same memory location, especially when using shared memory. This has to be taken care of by whom? Programmers will take care of how threads are accessing the memory.

(Refer Slide Time: 38:29)



Atomic operations for avoiding race conditions

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

For example, *atomicAdd()* reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete.

```
__global__ void mykernel(int *addr) {
  atomicAdd_system(addr, 10);       // only available on devices with compute capability > 6.x
}

void foo() {
  int *addr;
  cudaMallocManaged(&addr, 4);
  *addr = 0;

   mykernel<<<...>>>(addr);
   __sync_fetch_and_add(addr, 10);  // CPU atomic operation
}
```

The other available atomic options are: *atomicSub(), atomicMin(), atomicMax(), atomicAnd()* etc.

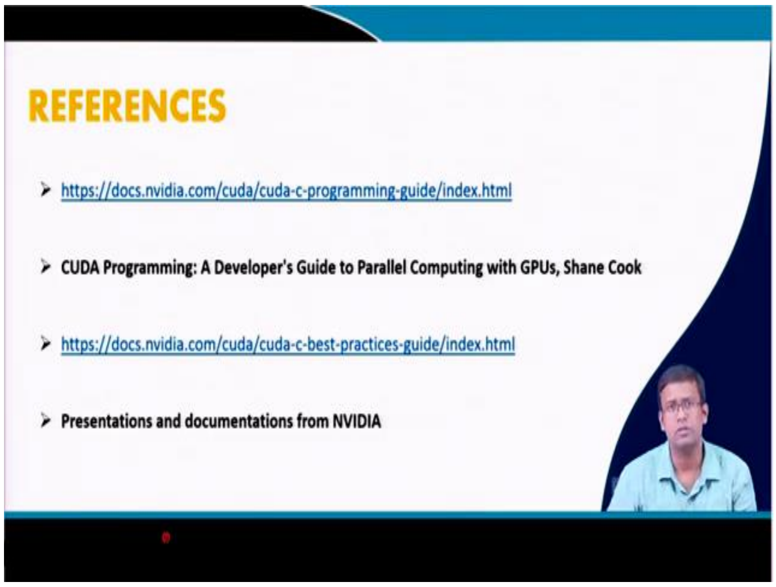
However, the operation is essentially serialized here and hence can degrade parallel performance for large block sizes.

In case many threads are trying to read modify write in the same variable which is often done; one variable is being updated by many threads and this is done through atomic operation. AtomicAdd reads a word at the at some address in a global or shared memory, adds a number to it, and writes the results back to the same address. This avoids race condition and contentions; we have looked into atomic, many threads trying to modify the same memory elements. So, in order to avoid conflict, they are done in an atomic way.

So, there is a sequentiality among the threads; each thread one by one writing to that memory location. Then you can see this atomicAdd system where this address each thread comes and adds 10 with this atomicAdd, and after the kernel is activated, there is a synchronization which says that this atomic add has given them correctly. The other operations are atomicSub, atomicMin, atomicMax, etcetera.

However, this operation is essentially serialized and they can degrade parallel performance for large block sizes. So, when we are doing atomic operations, this can avoid race conditions and can give us right results; but we have to be careful while doing so.

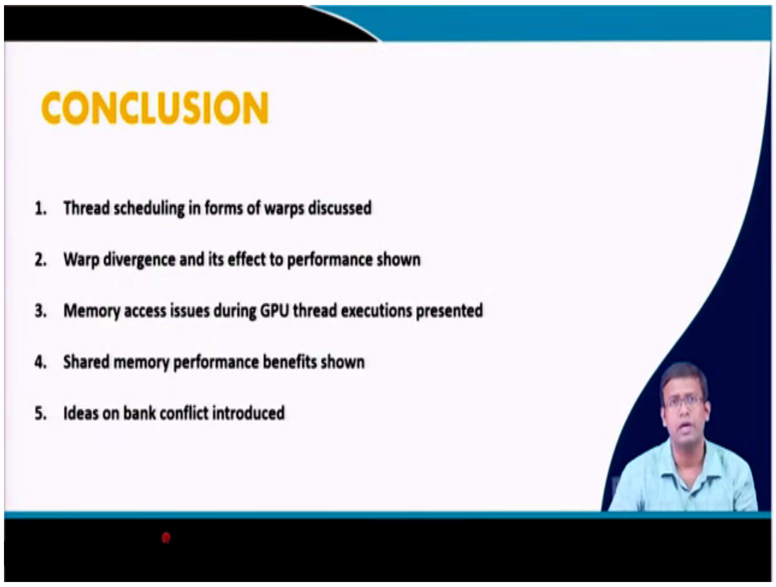
(Refer Slide Time: 39:50)



REFERENCES

- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, Shane Cook
- https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
- Presentations and documentations from NVIDIA

(Refer Slide Time: 39:53)



CONCLUSION

1. Thread scheduling in forms of warps discussed
2. Warp divergence and its effect to performance shown
3. Memory access issues during GPU thread executions presented
4. Shared memory performance benefits shown
5. Ideas on bank conflict introduced

Well, these are the references I have used; in the last two lectures, we looked into thread scheduling in forms of warps. Threads are scheduled as a group of threads within a block or run in the streaming multiprocessor and this group is called warp, this size is 32. Warp divergence or thread divergence reduces the performance, we looked into that.

Memory access issues during GPU thread execution is seen; it is seen that shared memory specially for large problems if we can break down into smaller blocks and utilize shared memory efficiently, then performance improvement is there. Shared memory is accessed in banks; while accessing shared memory, bank conflict has to be avoided.

Well, in the next class we will look into matrix operations and well we will move forward with the concepts which we have discussed till now.