High Performance Computing for Scientists and Engineers
Prof. Somnath Roy
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

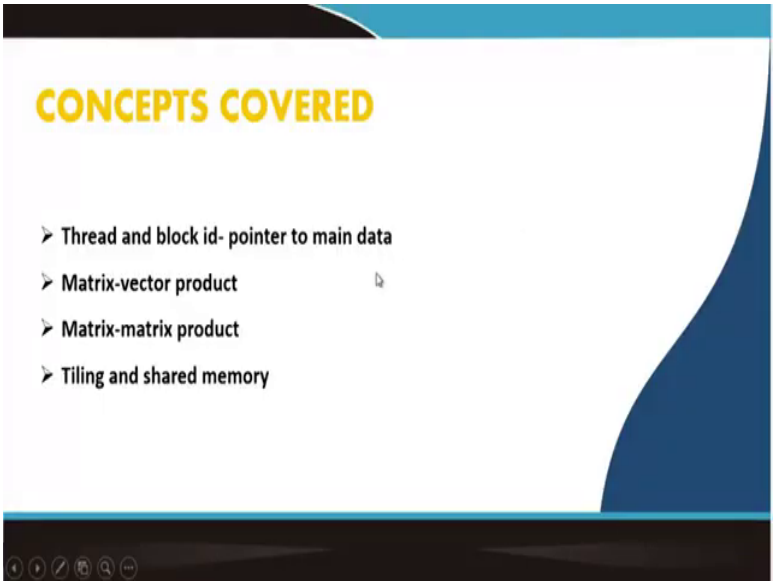
Module – 04
GPU Computing
Lecture – 38
Matrix multiplications in CUDA

Hello, welcome to the class of High-Performance Computing for Scientists and Engineers and we are discussing the 4th module of this class which is GPU Computing. Today, we will discuss Matrix multiplications in CUDA. In the last few lectures over the last couple of weeks, we are learning about different nuances of CUDA programming.

We looked into how to parallelize a section to run in GPU in CUDA, how to take care of memory management in CUDA, what are different off-chip and on-chip memories and how to utilize these memories efficiently in CUDA programming. Now, with that background, we will see examples which are very much related with scientific computing and these are matrix multiplication operations.

So, we understand that dealing with large scientific or engineering problems often requires utilizing matrices, doing matrix vector products, matrix-matrix multiplications and we will see how CUDA can be efficiently used for that.

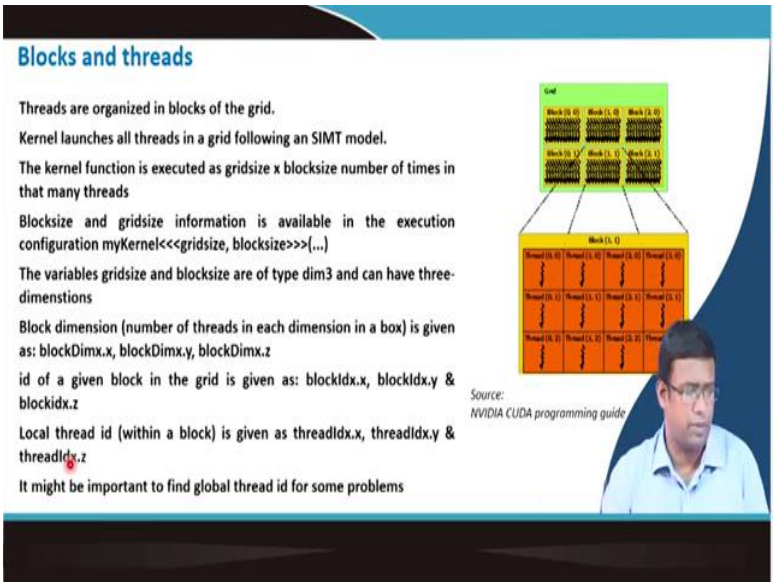
(Refer Slide Time: 01:35)



So, in this class, we will see how using the thread and block id we can point to different locations of main data because we are using a single instruction multiple data modules essentially same instructions will be run over different locations of the data and how thread and block id can be utilized for that. We will see examples of matrix-vector products. We will see the programs on matrix-matrix products again using CUDA to run on GPU's.

We will see how tiling, if you remember we have discussed matrix blocks or matrix tiles when discussing matrix products while discussing OpenMP programming. So, how this tiling and how utilization of shared memory can improve the performance of matrix-matrix products, there are certain complicated memory access in matrix-matrix product, cache unfriendliness can be there which has to be taken care of and we will also see some examples related with matrix solvers and how the performance is.

(Refer Slide Time: 02:42)



If we look into a CUDA kernel, when it is executed, it launches a number of threads and these threads are launched as a grid, inside each grid there are multiple blocks and these blocks are composed of a number of threads. So, kernel launch is essentially a single instruction multiple thread model or single instruction multiple data model parallelization using multiple threads.

The kernel function is executed on grid size into block size number of times in that many threads. So, if a kernel is launched, grid size the number of blocks and block size is the number of threads in each block. So, the total number of threads will be grid size* block size because each block has the same number of threads.

So, that will be the total number of threads launching when a kernel is launched. And, this information is available in the execution kernel, what is the block size and grid size programmer have to specify a block size and a grid size and in the execution kernel which is given by the variables grid size and block size when you call the kernel.

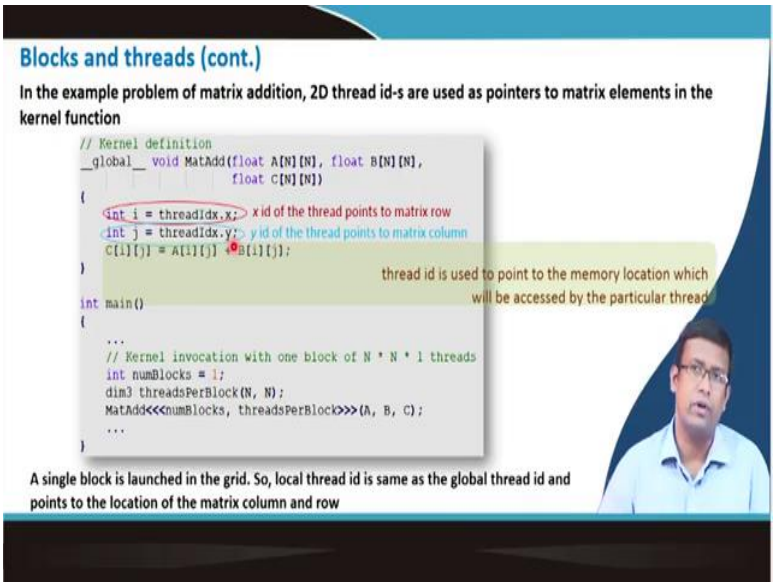
Then, these variables are of type dim3 so, you can put three integers in grid size and say that this is the number of blocks in x, y and z directions. Similarly, because there is a dim3 variable, three integers in block size and say that this many are the number of threads in each block in x, y, z direction. So, in total, you can launch a large number of threads.

You can find out what is the total dimension of block when the code is being executed , say you have specifying grid size and block size as dim3 variables, but when you are inside the kernel, the kernel might need to know what is the total number of blocks and what are the total number of threads inside the block etcetera, which can be found out by these variables.

A thread is also required to know its identity, what is the specific number of the thread. Now, if we think of MPI and OpenMP programs, there are function calls by which a thread or a processor knows it's rank or identity. In GPU's because there will be a large number of threads already in a GPU kernel, if you call functions to identify the number of threads that will have to be high overhead. So, there are some automatic variables which are blockIdx. x, blockIdx . y and blockidx. z which gives the local id of the block. In the grid, there are many blocks, and these are three-dimensional arrangements so, when you look into this variable, it will tell you about the coordinate of the block or the location of the block inside the grid. Similarly, if you find the thread id inside a particular block, you will get threadIdx.x, threadIdx.y, threadIdx .z these are like C++ structures.

But this thread id is the local thread id, is the id of the thread within a particular block. In many times, it might be required for one to know about the global thread id; that means, say block number p, there is a thread number q so, what is the global number of threads? How many threads have already been passed before that? Considering the blocks which I counted before this block, what becomes a global id of this thread and we will see when this will be important in a while. It may be important to find the global id of the thread for some problems.

(Refer Slide Time: 06:47)



If we look into a matrix addition problem, when this global id can be important, we will see here. What we are doing here, we are adding two matrices. If we see when we have launched the kernel, the total number of blocks inside the grid is 1, grid size is 1 so, there is only 1 block. Inside 1 block, there are N* N threads, N in x direction, N in y direction. So, the thread id inside that block will have two components; one is its x coordinate and another is 1 coordinate.

Now, when we call the kernel, we call the kernel function number of blocks which is 1, threads plus block because these are dim3 variables, it will take in N, N into 1 and in input parameters. Now, when the kernel function is called it will take matrix A, take matrix B, the output will be matrix C. So, each element of matrix A will be added to the corresponding element of matrix B and the output will be an element of matrix C.

How is it doing that? It is identifying the location of an element in matrix A or B or C by the thread id. So, if I am looking in a particular thread, this thread is taking one element of the matrix A, taking a corresponding element of matrix B and adding them and getting the corresponding element of matrix C. This is being done by; this is being done by all the threads simultaneously. So, which element of thread will pick up that if you look into this code that is found out by the coordinate of the threads.

Inside the matrix, the element that is being operated by a thread has an index or has indices the same as the indexes of the thread or the coordinate of the thread. So, far this is simple in a sense that each thread will do same operation, one thread will pick up one particular location of

matrix A, similar corresponding location of matrix B, add these two variables, put in the corresponding location of matrix C. So, x id of the thread points to the matrix row, y id of the thread points to the matrix column and we get the product.

Thread id is used to point to the memory location which will be accessed by the particular thread. Now, this is simply obtained here because we have launched only one block. Because we have launched only one block, the thread id x and thread id y can directly correspond to the element location on the indices of the matrix, each thread can be directly mapped to one element of the matrix. Well, so we can go ahead with one block, I mean within a block there can be a large number of threads which can be operated. So, you can go ahead with one block, but if we now think of it in terms of (Refer Time: 10:04) of the blocks, each block will go to one streaming multiprocessor. Inside one streaming multiprocessor at one go, one warp of threads will be active; that means, 32 threads will be active.

So, even we are launching large number of threads, if we are operating over $10^6 * 10^6$ matrix so, we are launching $10^6 * 10^6$ threads, but we are getting a parallelism only by the factor 32, because at one go only 32 threads are active. So, we are not getting the right scalability.

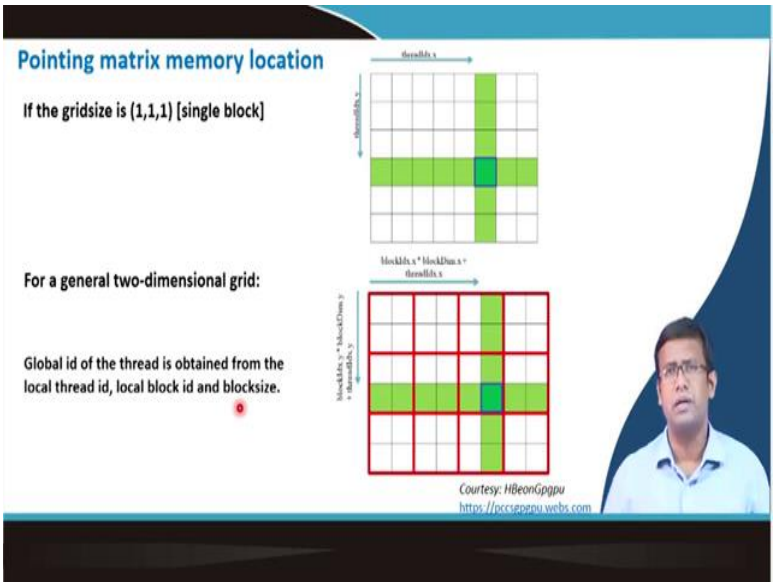
Therefore, if we are working on V 100 which has 80 streaming multiprocessors, we can think of at least 80 blocks as a matter of factor, we have done more than 80 blocks. So, all the streaming multiprocessors are working together, and we are getting parallelism not only for the 32 threads in the warp, but also multiple streaming processors are active and all they act to the scalability.

So, you need to launch number of blocks and I think I am clear that if we run in one block, we will get less scalability, you have to run number of blocks and we have seen less class that what is optimum block size that depends on a multiple of warp that also has an effect on the performance. If you just keep on increasing the block size even by multiples of warp, you may not get the right performance, there is an optimum block size depending on the memory utilization of the program.

So, we need to launch this kernel in multiple numbers of blocks. So, we need to have multiple blocks. Therefore, we cannot just pick up one block and say that its x id will correspond to the particular location of the thread, we need to use the global ids of the threads.

We have to consider that it is like multiple arrangements of threads are there, we have to count what is the global x id, we have to count what is a global y id and point that thread accordingly. So, as a single thread is launched, local id and global ids are the same, but when multiple threads will be launched, they will be different. We have to take care of that.

(Refer Slide Time: 12:19)



This gives us the issue that my threads will be utilized also to point to the memory location. I will not use any other table or any other pointer to the memory location, the thread ids are the pointers for the memory location. Each thread will do a small amount of work. All the threads will simultaneously carry the work therefore, a large problem can be addressed. This is precisely a single instruction multiple thread model or single instruction multiple data model. Here, we have a large number of threads and each thread is pointing to one location of the matrix and therefore, at a go the large matrix problem is solved.

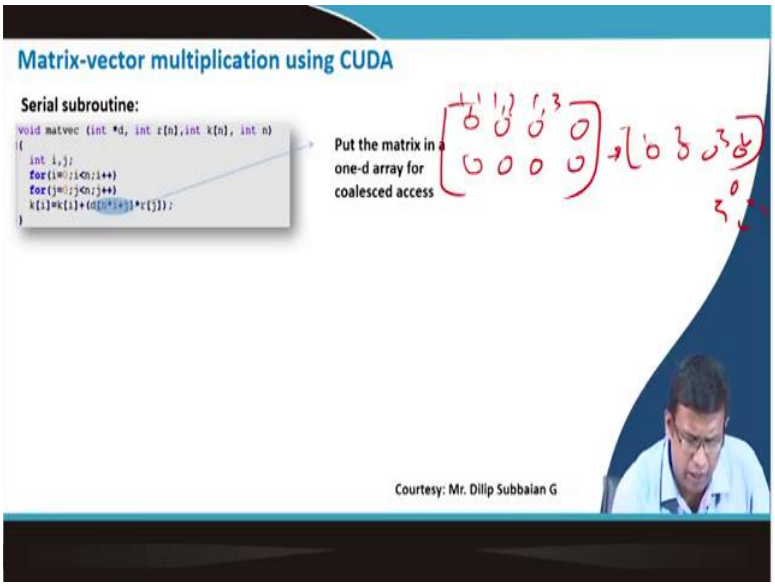
So, if we have only one block, we can use the threadId x as the x coordinate, threadId y as the y coordinate and locate the location of the matrix. If we have a general two-dimensional grid, we can extend it for three dimensional also, you need to think of the thread algebra there.

We have to use that threadId x is the local threadId, blockId x is the id of the block and blockDim x is the dimension of the blocks in x direction; that means, in x direction total each block has that many number of threads in each direction. Block ids, thread ids everything starts from 0 to the number of threads-1.

So, if a block has an id blockIdx. x in x direction, there are total blockIdx .x blocks which (Refer Time: 04:04) before that. So, each block has blockDim.x threads in x direction. So, the total number of threads in x direction will be there multiple and then, we add the local thread id.

Similarly, in x direction, similarly in y id, we do that and we find out the global thread id using this. Global id of the thread is obtained from the local thread id, local block id and block size. Using this three information, whatever the block arrangement be 2D, 3D we can find out the global thread id and local thread id. In certain cases, there can be some more complications, but I mean using simple algebra that can be found out.

(Refer Slide Time: 14:44)



Now we will look into a multi block matrix vector product and this program that will show is written by one of my interns Mister Dilip Subbaian was a B-Tech from Shiv Nadar University.

So, we think of a matrix vector product. What is in a matrix product? Matrix row will be chosen that will be multiplied with the vector as a dot product with corresponding column element will be multiplied with the corresponding row element of the vector and we will get one row element of the matrix vector, vector product.

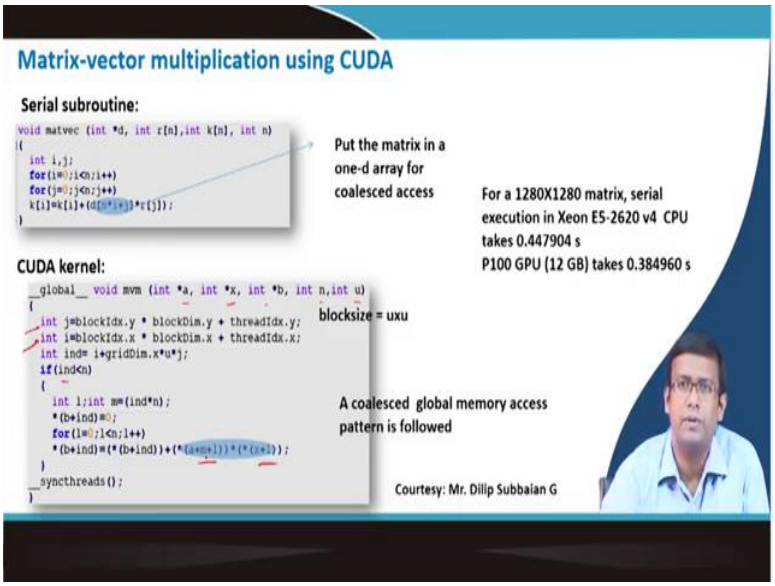
So, we can see that the matrix d is multiplied with vector r. Now, while looking into this product, this is a serial subroutine, a simple C code. One thing we can see is that this matrix

instead of d being a 2D matrix it has been put into a vector order. So, we have somehow coalesced the memory access.

Instead of writing the matrix, in this order, 1, 1, 1, 2, 1, 3 we have coalesced the entire matrix into a one-d array. So, 1, 2, 3, 4, 5, 6 so on. So, the entire matrix is coalesced into a one-d array and that is why row i and column j is written as n * i + j th element. So, there have been i rows and j columns there, and that is multiplied with j th row of the vector.

So, the matrix is put into a one-d array for coalesced access. Why is coalesced access important? If we go back to the fundamental concepts of CUDA programming from reading from the device ram to the GPU cores require a coalesced access and that is a cache friendly access also. So, that is why we try to put the matrix in the coalesced access that gives us better performance.

(Refer Slide Time: 17:05)



Then, when we write this programming GPU instead of doing this through a function, we call a kernel mvm and this kernel takes what is the block size is u, n by n is matrix size the order of the matrix, a is the input matrix, x is the input vector, b is the product each location of b is initialized to 0 and then b is equal to b * (corresponding element of a)*(corresponding element of matrix x).

Well, another thing that we can see here is that we are finding out the global locations of i and j where the thread should point by blockIdx.y* blockDim.y + threadIdx.y and blockIdx.x *

blockDim .x + threadIdx.x. So, we are utilizing the global locations. As we have coalesced the memory access, we are operating for over an index int, this index gives us a coalesce of the memory for the vector.

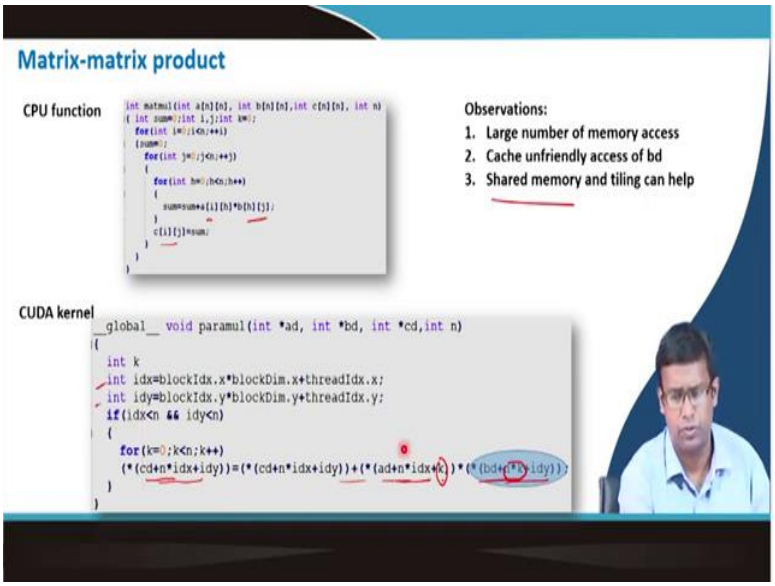
We are accessing the memory in a coalesced manner and this ind *n is the total number of elements which has covered in the previous rows and goes to the particular row, the previous row it covers the previous row and looks into the location and the particular column of that row. So, that is why we are using coalesced memory access here.

A coalesced global memory access pattern is followed, and this gives us better speed up. We can see that for the 1280, 1280 matrix, CPU takes 0.44 seconds and GPU takes 0.38 seconds, and this is not very high speed up why? Because the computing is small, 1280 by 1280 is a small matrix problem, small matrix vector multiplication and we are doing it only once. So, speed up is not substantial in this case.

Now, also you need to copy the entire matrix and vector and from CPU to GPU and then copy it back from GPU to CPU that is the product. So, it takes up a large time. In case, if you have a large matrix, then you get better advantage of the parallelization and we have seen that.

Well, one very important thing is that when we are looking into matrix vector patterns, a coalesced global memory access pattern is followed and this index ind is introduced for that. We are instead of considering the matrices as 2D matrices, we are coalescing them into a single row and doing the operations based on that. This is extremely important to note here.

(Refer Slide Time: 20:16)



In a matrix-matrix product, as we can see when we think of a matrix-matrix product in CPU, we take a row of the matrix left-hand side matrix, a column of the right-hand side matrix and take a dot product between them and make them the corresponding element of the product matrix.

Now, when we look into the CUDA kernel, we would do essentially same thing each thread points to one particular element of the product matrix not on the left and right matrix because in in left in the multiplied and multiplied matrices, it will go over the entire row and entire column and make dot products.

So, it looks into our particular row of the multiplier matrix a the threads x coordinate, the threads y coordinate looks into a particular column of the right multiplier matrix b and the threads itself also looks into i and j or the threadIdx. x and threadIdx. y looks into the particular row column location of the product matrix cd.

So, these matrices cd, ad, bd because they have been copied from the host to device, host matrix a, b, c and these matrixes are ad, bd, cd and this is this addition is done over all the elements, this is dot product over all the elements in a particular row and column well. Now, what we can see we have coalesced the memories instead of 2D matrices, we are considering them as 3D matrices and writing it to the particular location which is given by the global thread ids.

Now, there are few observations. There is a large number of memory accesses. In each row operation, for winding out one element in cd memory of access is done for the entire row of a and entire row of b. So, if n large number say we are talking about a million-by-million matrix. So, for finding one element in the c matrix, I have to go for 1 million elements in the row of a and 1 million elements in the column of b. So, a large number of memory accesses are required, 2 million memory accesses are required and we will get 1 million operations by adding them. CGMA ratio is very wide, but as we have a large number of memory accesses, it will be a slower process.

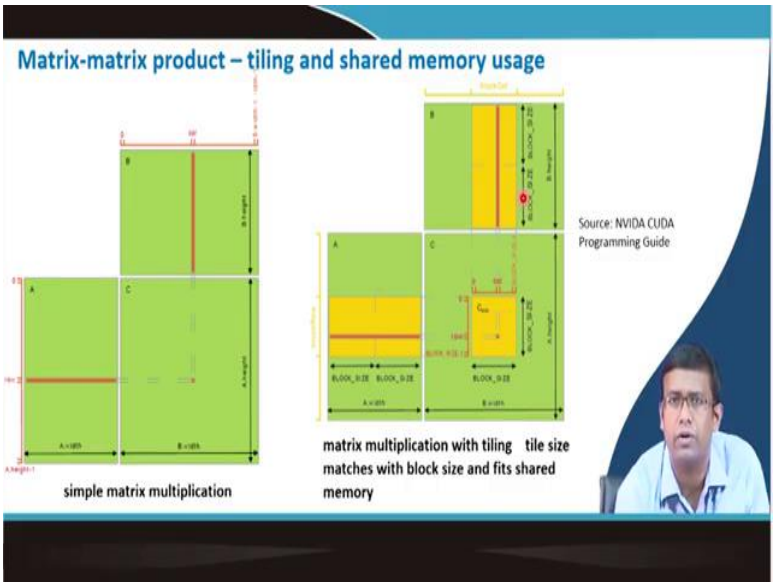
However, for memory access, we avoid reduction of bandwidth by utilizing the cache. What cache does? It prefetches some of the memory elements in the contiguous location, but here what we can see because row a is row is multiplied with b's column, in a from k 0 to n the memories contiguous, but in b, memory is not contiguous because it is multiplied with n into k, it is looking into the column elements, memory is arranged along the row. So, either we have to store it as transpose of b which is possible yes, there will be a problem. So, what we see here that this access, the way we are accessing bd is a cache unfriendly access pattern.

Because we have to read from the global memory to take from the device time global memory and bring it to the CUDA course and many memory readings are required, it will be a cache issue that is affecting, its bandwidth will be low, it will be a slower process.

So, utilizing shared memory might help us and utilizing tiling; we have seen it earlier that if we consider blocks of the matrices and do the multiplications, only within this block, we can use most many of the cached variables and that also can help.

So, using shared memory and tiling can help us in terms of memory access and improve that performance. This particular program will have some performance issues.

(Refer Slide Time: 24:58)



So, we look into NVIDIA CUDA programming guide, the example is given. So, when you do multiplication in between two matrices, you read the entire row of one matrix, you read the entire column of one matrix and only get one product here. Instead, if you take a block of or a tile of number cr and if you take a tile of number cr, you first multiply them, get the products here.

Again, this product is not the final one, so you have to take another block here, another tile here and I will not use a block here, block makes a different sense here. You take another tile here, multiply them and write it back here.

So, this into this will give me some products in this particular operation yellow, this into this will also give me some products here, this into this also give me some products here, this block into this block will also point to this particular row. So, you have to go over loops of tiles and then you can get the product here.

So, what is the advantage? The advantage is that once you read this entire block, all the products can be done here, you do not need to read the memory values; what means that when you are reading, we are reading from the global memory. So, you will not need to read it again and again.

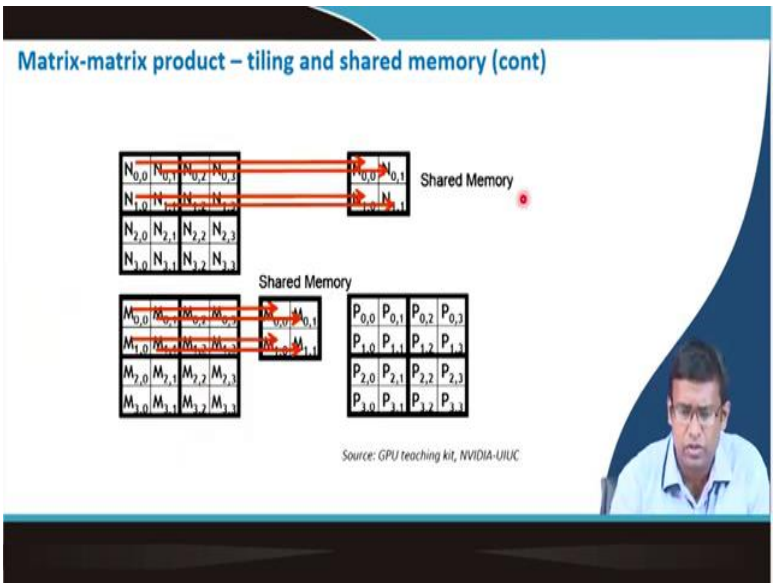
Also, as you are considering a small time in both A and B matrices, the memory of these tiles might be small enough. So, that they can directly feed into the shared memory. So, and once

something is ready in the shared memory, it will remain the shared memory until unless you overwrite it.

So, you one warp can read it, but other warps also can utilize it. So, read one block of one tile of data from matrix A, read one tile of data from matrix B, put them into the shared memory, do all the multiplications required for matrix multiplication for getting the product matrix in between them, write in product matrix C.

Again, write another tile from matrix B, you may utilize this tile which is already in matrix A and do the products and add in the product matrix location. In that way, different tiles can be allocated to different threads and that way you can get better performance.

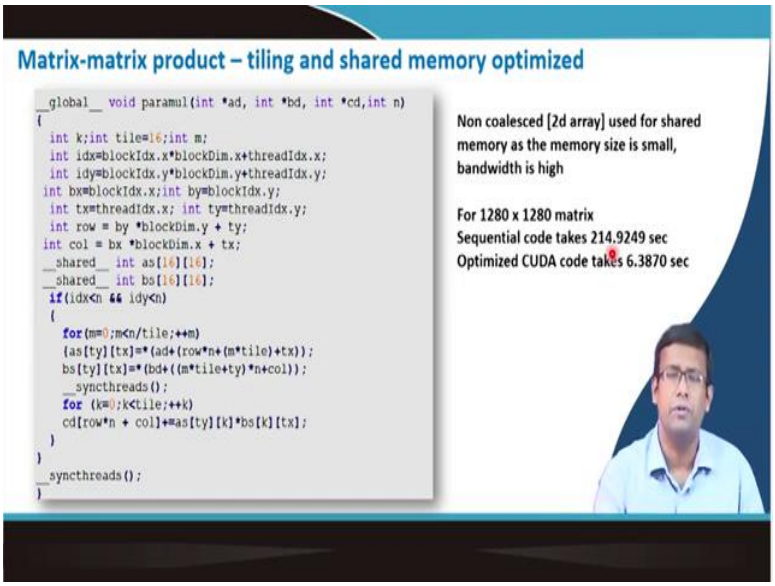
(Refer Slide Time: 27:11)



We will see an example. So, what are we doing? We are taking one tile of matrix A, putting it into shared memory. This is where we are doing M to N is equal to P. One tile of matrix M and putting it into shared memory and doing the multiplication putting them in here.

Then, we are doing multiplication among all the blocks and putting it here. Then, we will take another block here and maybe another block here and some element will again come here. So, we will keep on adding these elements.

(Refer Slide Time: 27:40)



So, the program looks like you identify the global threadId, get the blockId, consider the local thread id also and use a shared memory variable. In the shared memory, you basically look into the local threadId in that particular block and inside the shared memory, now this shared memory is only on a particular block, shared memory has no scope outside the block. So, the shared memory valuables you use using the localId and use this global id for fetching data from the global memory, put it into the shared memory and do multiple tiles still it finishes all the rows in the of the matrix to multiple tiles for that particular block and keep adding on the global memory itself.

So, these memory accesses are local memory or shared memory access. Shared memory is a small memory unit, so entire memory is something like cached, entire memory is directly readable using the shared memory banks etcetera. So, you do not need to think about coalescing the memory.

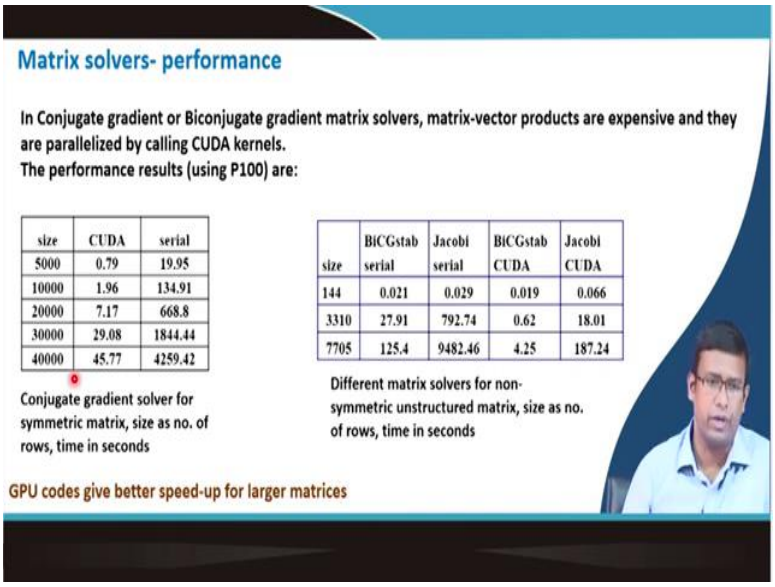
The cache friendly access which might be required for larger matrices, but shared memories here we are discussing only 16 by 16 it is a few kilobyte, 64 by 92 kilobyte it is the maximum space. So, you do not need to coalesce the memory access here, but the device memory or global memory which you are accessing must be coalesced.

So, non-coalesced access is used for shared memory as this size is small and bandwidth is high. For a 1280 by 1280 matrix, sequential code takes 214 seconds, this optimized code takes 6.38

seconds. So, nearly 30 times more than 30 times, speed up has been obtained using the GPU card here .Why is the speedup more than matrix vector product?

Because first is that we are using this optimization based on shared memory and tiling and also the total computation is quite larger. In matrix vector product, total computation is small only getting one row of the product matrix compared to this; there order of n and here the computation is of the order of n^2 .

(Refer Slide Time: 30:04)



We look into matrix solvers. These are very important in scientific computing calculations and we will look into conjugate gradient, biconjugate gradient and Jacobi solvers. If you look into all these solvers, these solvers are the maximum costly part or computable part inside the solver is matrix vector product.

So, a parallelized matrix vector product here, now matrix vector is not as costly as compared to matrix-matrix. The good thing is that these are iterative solvers, so matrix vector products are required again and again.

However, matrices remain the same. So, you do not need to copy the matrix many times from host to device. So, copying the matrix is done only once, but the copied matrix which is laying on those GPU devices is operated again and again for matrix vector product and large number of iterations are done and if you see a Xeon 2650 versus P100 speed, if we keep on increasing

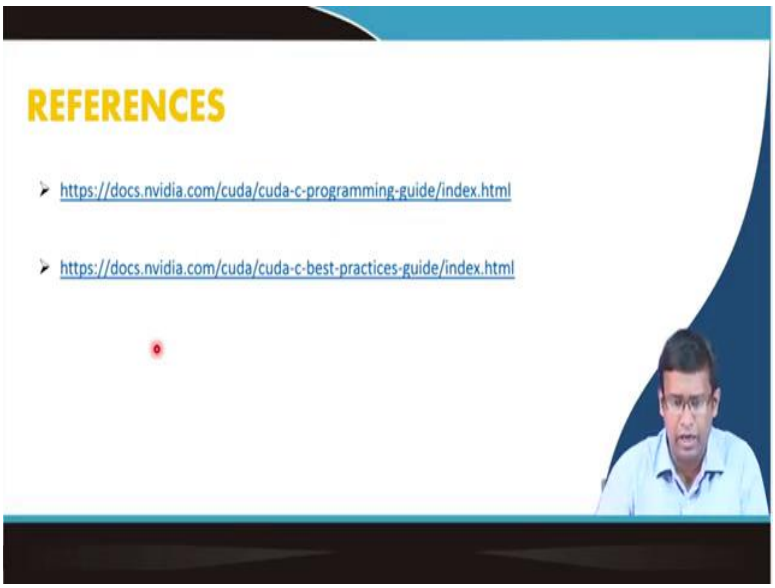
the matrix size say 40000 by 40000 in a conjugate solver symmetric matrix, we get near 90 times speed up.

For BICG also, we get high speed up and as we increase BICG and this is nearly 30 times speed up because the matrix size is small ;as we increase the number of rows of the matrix, the speedup is more. So, these are for unstructured non-symmetric matrices and conjugate gradients for symmetric matrices. As we increase the size, the speed up increases here also. So, that is one observation also this follows the basics of parallel computing performance that as you are increasing the problem, the speedup is better and to get that we in GPU there is a challenge of memory access we have to be very cautious while looking into the memory access.

So, matrix solvers and matrix-vector, matrix-matrix products are very important in scientific computing, but we can see that if we write the GPU codes efficiently, we can get very good speed up and this up is comparable to even more than a large CPU based infrastructure, you need many more racks of CPU, a big room with air conditioning to get something like what we are getting around 90 times speed up, you need maybe 100s of 200s of CPU's there.

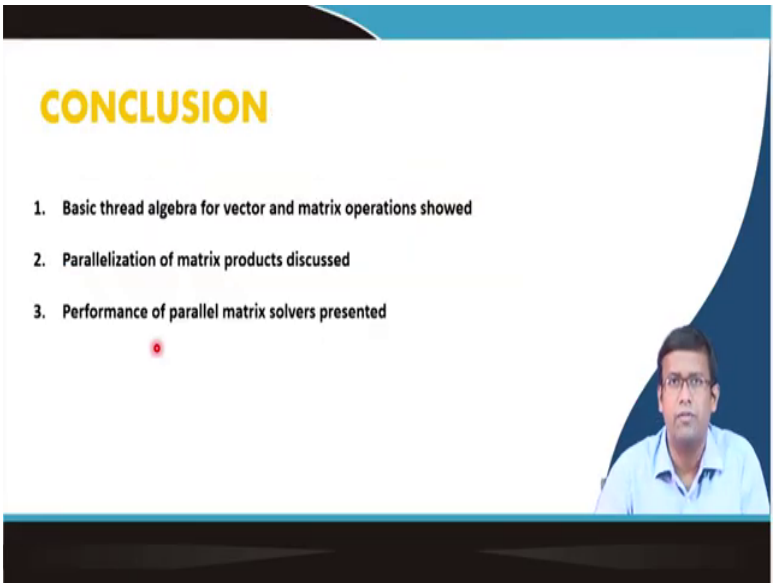
But a simple single GPU card, if the CUDA program is tuned properly can give you similar or better performance.

(Refer Slide Time: 32:44)



Well, mainly you have used CUDA C programming guide and CUDA C best practice guide, there these are very important in case you try to develop your own CUDA program for this lecture.

(Refer Slide Time: 32:58)



We have looked into basic thread algebra for vector and matrix operations, shown parallelization of matrix operation and optimization using shared memory and what is called tiling here and performance of parallel matrix solvers are also presented.