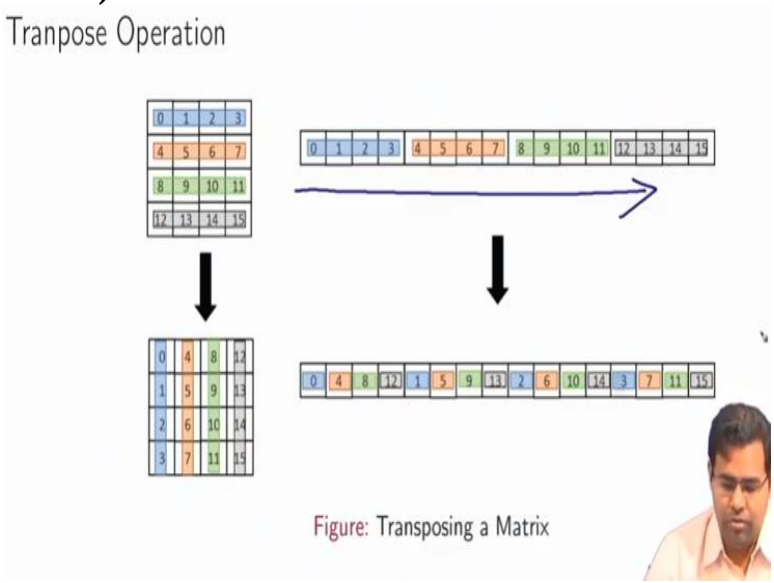
GPU Architecture and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Technology – Kharagpur

Module No # 05
Lecture No # 23
Memory Access Coalescing (Contd.)

(Refer Slide Time: 00:28)



Hi welcome to the classed of GPU architectures and programming in the last lecture we have been discussing about how an operation like matrix multiplication can be accelerated with proper optimization involving the shared memory. And in this lecture we will continue with some more examples may be bit more involved and we will also try to show that how you can able to perform some monitoring based on the optimizations and figure out how will optimizations are working for you using some profiling primitives and Nvidia provides for you.

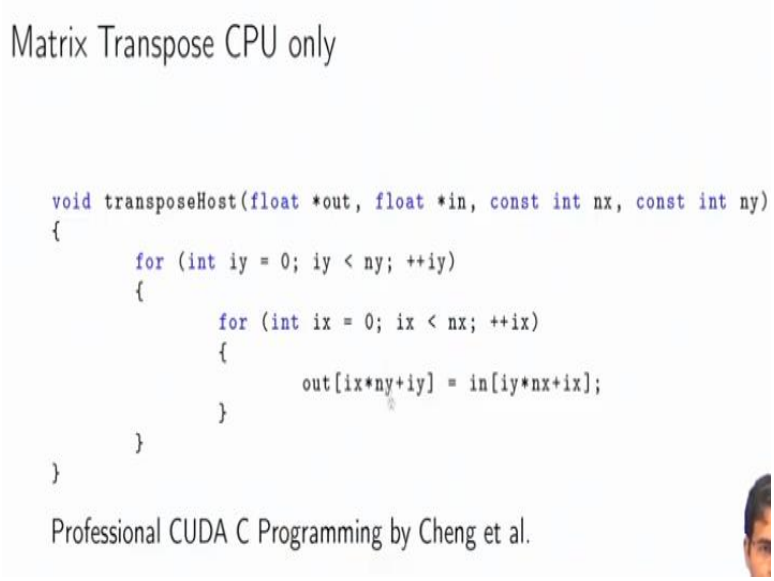
So as an example in the last case we consider the operation of matrix multiplication in this case we will consider this example of matrix transpose. So if you remember from your high school algebra a matrix transpose was a simple operation such that suppose you consider a matrix a is given to you and you want to create a A transpose. So the way you would be defining the entries of transpose would be that whatever was an entry of the ith row jth column aij in the original matrix in the transpose matrix they should be the entry in the it should be the same entry in the jth row and ith column.

So the way it is going to work is that in the transpose matrix if you consider any entity in the I jth column it should be same as the entity in the jith column of the original matrix right. So this is how it would hold at the element level that in the so essentially if we take an example has been provided here so as you can see that we are considering an original matrix which is like this the one that is given on the top left and when I do a transpose essentially all the ijth elements in the original matrix become the jith elements in the transpose matrix right.

So essentially I can just say that the diagonal is remaining the same and the rest of the elements are just flipping over to the other side. So this comes here this goes here and similarly for all the other entities here but how is it going to really happen in the underlying architecture that is the important question here right. So if I look at the arrangement of this matrix with respect to the memory then for all practical purposes this is in derivative of C so we have a row measure implementation right.

So your matrix is finally resident in the GPU memory in a simple consecutive locations for the different elements in the row followed by the next row so on and hence so forth right. So when you are really doing a transpose operation things as you see that which looks very nice in the original 2D representation you may lose that simple property here. Of course in terms of the matrix is fine but here things will be happening in this single 1 dimensional array which you will be accessing using the row and column indices of the original matrix that actually you have to again compute using the block and thread indexes of the original matrix.

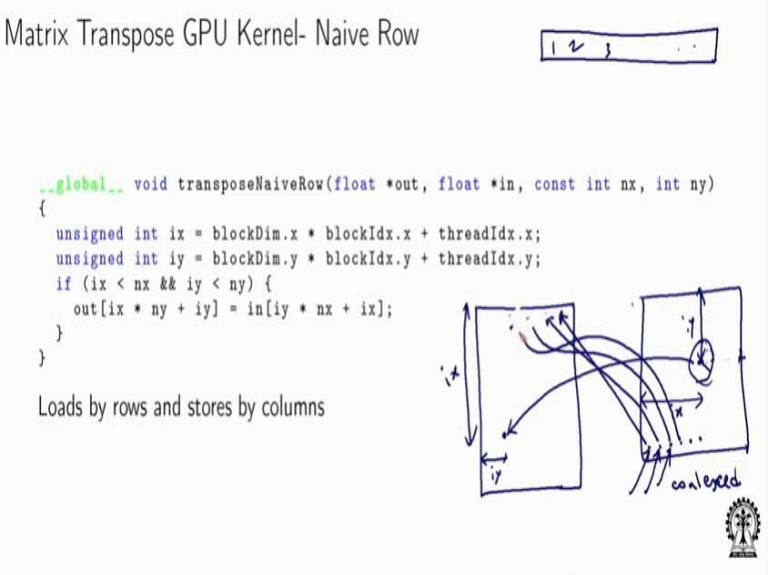
(Refer Slide Time: 04:23)



So if we consider a simple a simple vanilla matrix transpose function in C in CPU only function. So how really you are going to go about it you will just write a cascade of loops right 1 loop is iterating over the rows using y's right so the outer loop is iterating over the rows. So this iy index is iterating over the rows and the inner loop is iterating over the columns per row and inside what you are doing is you have been given an a pointer pointing to the input matrix and you are going to write the data into the output matrix using this out pointer.

So essentially you are just moving you are just copying elements from the y xth location to the x yth location right. So I can just say that this is in terms of the representation of this course whatever is in the y xth location the index in y axis and index in x axis gets flipped to index in x comma index in y location right. So this is the some of the examples taken from this book by professional book on professional CUDA C programming here.

(Refer Slide Time: 06:19)



Now if we consider a GPU version of a code what would happen of course since in the CPU I would have cascaded loop and so it is a sequential loop and it is iterating over the 2D arrangement of data points and using the loop iteration variables and doing the transform notion from the in matrix to the out matrix right by accessing this entities. But when it will be done by the GPU it will be again the work will be part thread basis so you will launch a kernel I mean as is always done as you will launch a kernel and you will be find the partial activity.

So every thread would need to compute it is ix and iy values that is on which location of the matrix is going to work right and then is going to copy the corresponding location from the in matrix to the out matrix. So the every thread which discover which location it is supposed to work 1 and then is going to copy here in the corresponding location where yeah. So whatever was the row it was iy it changes to the column offset here and whatever was the column offset is going to change to the row number here right.

But so again we are calling it as a naïve implementation because I am not making use of any shared memory another thing of course we think that it can be optimized further using such implementation we will get into that. But here observe what is happening so essentially I call it as naïve row implementation because we are loading rows and storing by columns why do I say that values we are loading by rows.

Because think how the warps are getting formed so you have thread id's for consecutive locations in the rows I mean you have a warp which is containing thread id's 1, 2, 3 like that and they are accessing in the input matrix they are accessing consecutive locations right and forming a warp. And they are loading the data from this consecutive locations which would mean the loads are coalesced.

So whenever this thread inside a single warp this thread is loading this data point the next thread is loading immediate next data point and like that. So this is going to get coalesced but what happens when the thread in the same warp are going to write. So here whatever in for this example I am trying to draw the so essentially we are picking up items at the lower level with a large row index right.

So that large row index would change to a large column index here so may be somewhere from here I would be starting to drop the elements right and we are having offsets like this right. So this column offset values would be changing to row offset values right so if I say I have a large row index and a small column offset so that would transfer to a small row index and a large column offset.

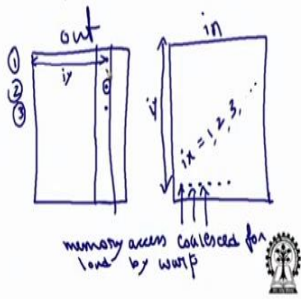
So essentially this thread would be storing the data somewhere here the next thread will be storing somewhere here and the next thread would be storing somewhere here like that. No wait a minute let me draw this with a bit more accuracy sorry.

(Refer Slide Time: 12:14)

Matrix Transpose GPU Kernel- Naive Row

```
__global__ void transposeNaiveRow(float *out, float *in, const int nx, int ny)
{
  unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
  if (ix < nx && iy < ny) {
    out[ix * ny + iy] = in[iy * nx + ix];
  }
}
```

Loads by rows and stores by columns



So consider a fresh picture here we have the original in and outs this is my in and this is my out. So let us assume that I am doing again same example I am doing loads I am accessing the warp is definitely accessing consecutive locations in the row why because inside a warp this y value I mean so since the warps are going to access consecutive locations and the consecutive locations would mean consecutive locations in the row in general.

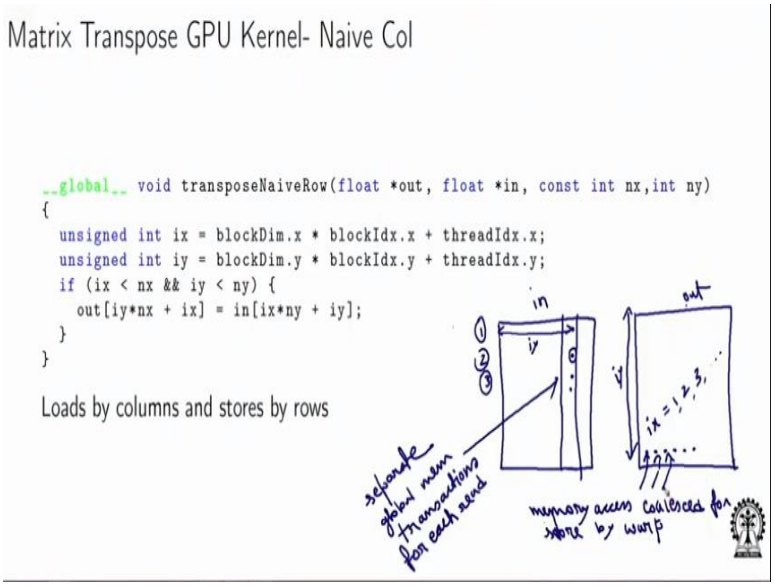
So as long as the i mean the warp is the size dimension of the matrix is divisible by the warp size right so look observe you consider that you are loading data from this point in a single warp right. So these are your loads and they are all coalesced. Now what happens when the threads are going to write so this in index is going to be written like this right. So you have a large offset here with respect to y and then for the same row offset let us give this some name let us say this phenomenon happening for some value of iy and let us say ix values are 1, 2, 3… right.

So now when you think that where are this coalesced load values the values which have been loaded by the warp in a coalesced memory access where are they going to be get stored in the out matrix that would be interesting why. Because now this ix value flips to become the this iy value which was the row index flips and becomes the column index right. So here let us consider the first thread here so it was loading data from a location with iy, 1 right so that would mean it is now going to download that data at in the out matrix at a location 1, iy right.

So that is it so let us say this is the first row so the data would be coming here what about the next one the data was it iy, 2 so it should get in 2 comma iy next 3, iy like that right. So essentially whatever you loaded consecutively are going to get stored in the iyth column from the iyth row. So essentially you are doing a load by row and you are storing the data by column. Now of course if you think from the access patterns all this rights are non-coalesced right.

Technically considering a big matrix all these rights will require separate global memory transaction right. So that is why you will have coalescing the good thing about the goodness of coalescing will help you with respect to the loads but since you are storing by columns you will be having a penalty here.

(Refer Slide Time: 16:45)



Now consider the other scenario where so as you can understand that this was a simple GPU kernel where all we did with respect to the original CPU kernel was we removed that 2 loops and just define a part thread activity. Part thread activity was that okay you just load values and put it in the transpose location in the output matrix. But I have 2 option now I have discussing here the first option is you just load by row and store by column and since this is one option of course I have the option why do not I load by columns and store by rows.

So all that will happen is that you are access expression is just going to change so this is the example where you are going to load the data by column and store the data by row. So I can just to re-use the same picture here may be so since I am loading by column I can say that this is my

in matrix this is my out matrix and I am loading this data points from the column right and what would coalesced would be write right.

So I will have memory access coalesced for write by the same warp so this is now write this is now read. So I have now got separate global memory transactions for each read operation or store operation or load operation right. So now I will have the advantage of coalescing while writing since I am storing by rows and I lose the advantage of coalescing in terms of reads. So as we can see this are the 2 options I can have but in both cases either my reads are coalesced or my writes are coalesced.

So that would give me some parallelization in terms of reducing the global memory loads or stores but I do not get both okay. So we have explained the idea of this shake macro and let us look at how to wrote the driver code including this macro.

(Refer Slide Time: 20:13)

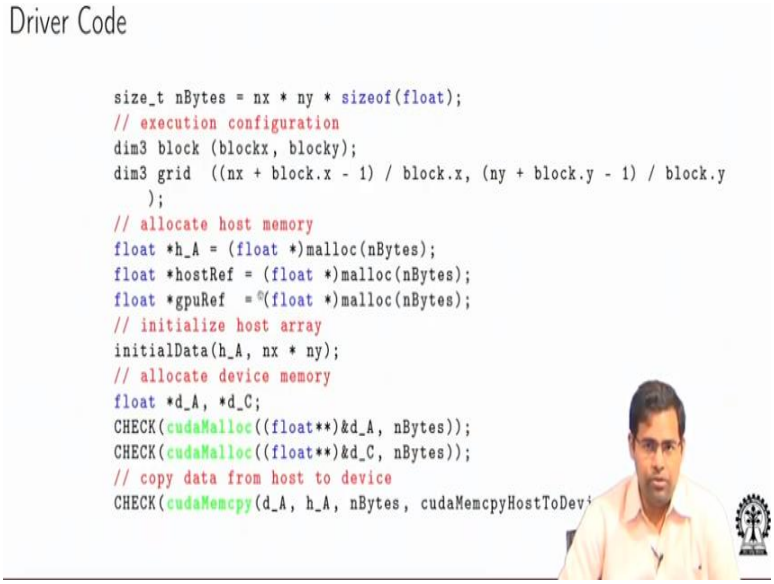
Driver Code

```
int main(int argc, char **argv)
{
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("%s starting transpose at ", argv[0]);
    printf("device %d: %s ", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // set up array size 8192*8192
    int nx = 1 << 13;
    int ny = 1 << 13;

    // select a kernel and block size
    int iKernel = 0;
    int blockx = 32;
    int blocky = 32;

    if (argc > 1) iKernel = atoi(argv[1]);
```

So essentially we are trying to write a simple driver code here which is the host program where we will be using this check macro to nicely capture the device properties of the underlying GPU device and print them up. Next we set up the input errors and we are trying to write a single program through which we can the user can input what you wants as the whether he wants to do a transpose by naïve row or by naïve column and that would be the user preference coming through which will be captured by the main program using the argv parameters.

So essentially I am assuming you are all familiar with handling command line arguments right so the i kernel variable is matching its value with whether I mean if I am using this q2i function to actually process the argv string and this argv string is converted from the string type to a 2i in the at k type and it is stored in the argv i kernel parameter.

(Refer Slide Time: 21:16)



And then so that is actually storing the users preference of what kind of operation you wants and the next thing we do is we just set up the normal grid and block launch parameters for the kernel. I mean what is the block dimension block x block y definitions and grid dimensions in terms of so the way you want the number of blocks to be parameterized like that so this is how we are writing that what should be number of blocks.

I mean essentially you divide the nx by the block x and ny by the block y right. So that gives you this 2D system will give you the total number of blocks and in here you have the definition of the thread block. I mean how the blocks are arranged in terms of block x and block y the next thing you do is you make suitable allocation from the host memory side so your dispose programs will also be available to you and then you make this call assuming that there is a function which is going to allocate the host array right.

This hos array hA with the input with all the input values stored and the dimensions are of course nx cross ny for this array and then you allocate device memory in dA and dC. So you want the host array to be transferred to the device memory using this CUDA mem copy command so in

device you are defining 2 memory locations dA and dC. In da you are copying the host array using the CUDA name copy command and everywhere you are using the check macro to just to check if there is any system error at now and then the check macro will nicely print the corresponding statement for debugging purpose.

(Refer Slide Time: 23:00)

Driver Code

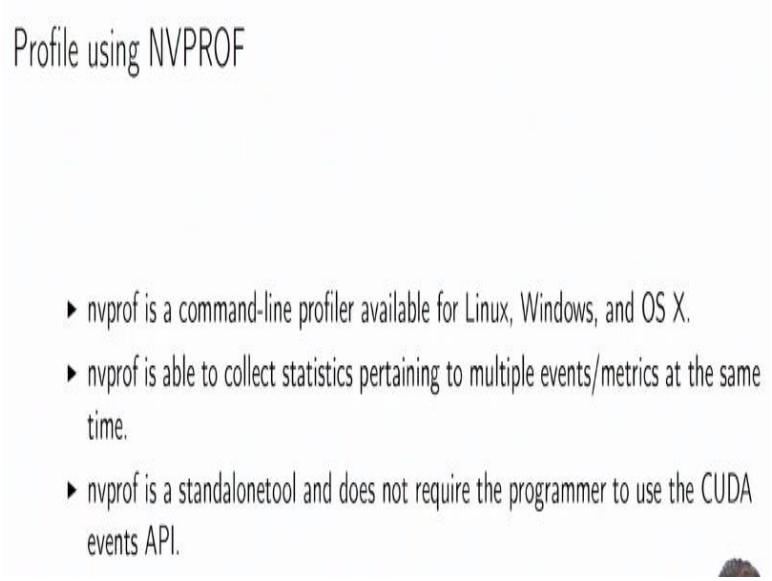
```
// kernel pointer and descriptor
void (*kernel)(float *, float *, int, int);
char *kernelName;
// set up kernel
switch (iKernel)
{
   case 0:
     kernel = &transposeNaiveRow; kernelName = "NaiveRow";  break;
   case 1:
     kernel = &transposeNaiveCol; kernelName = "NaiveCol"; break;
}
   // run kernel

   kernel<<<grid, block>>>(d_C, d_A, nx, ny);
        CHECK(cudaGetLastError());
        CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));
}
```

And here we use the function pointer to actually pass the user's input choice because I have already captured the user's input choice using this i kernel integer right. So argv1 is containing the first option provided is containing what is the functionality that the user want to execute right. So then here through a switch case we are trying to match this value of I kernel whether it is 0 or 1 and accordingly depending on what is the user's parameter we are trying to set this function pointer kernel to the and provide it with the pointer for the device function transpose Naïve row or transpose naïve column right.

So essentially we are using this we are setting this function pointer up here so through this switch structure and the user as already passed what is preference in terms of the in terms of this argv parameter which have been passed to kernel right. So coming here we are finally running the kernel so as you see that now this function pointer kernel as been suitably initialized it has been provided with the address of either transpose naïve row or the transpose naïve column function whichever you want and then this kernel will execute and of course this are all asynchronous statement here like the kernel will be launched in the CPU will be waiting for the last error message if there is any.

And then once then this CUDA name copy command will again be executed to get back the value from the GPU of the transpose get back the transposed kernel value from the GPU side right.

(Refer Slide Time: 24:51)



Now in this example we will demonstrate usage of a profiler from NVidia like how to do a command line profiling using this nvprof tool and how to figure out what is the performance of your program. So we with this initial idea that we can do performance of our programming using this kind of nvprof profiler will end this lecture and in the next lecture we will go into further details of using nvprof for doing CUDA kernel profiling thank you.