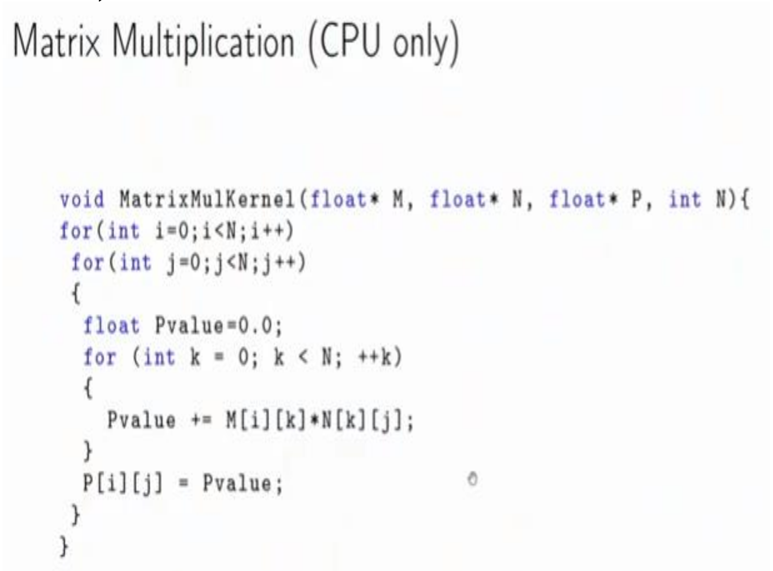
GPU Architecture and Programming

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

Module No # 03
Lecture No # 11
Intro to CUDA programming (Contd.)

(Refer Slide Time: 00:36)



Matrix Multiplication (CPU only)

```
void MatrixMulKernel(float* M, float* N, float* P, int N){
for(int i=0;i<N;i++)
 for(int j=0;j<N;j++)
 {
  float Pvalue=0.0;
  for (int k = 0; k < N; ++k)
  {
    Pvalue += M[i][k]*N[k][j];
  }
  P[i][j] = Pvalue;
 }
}
```

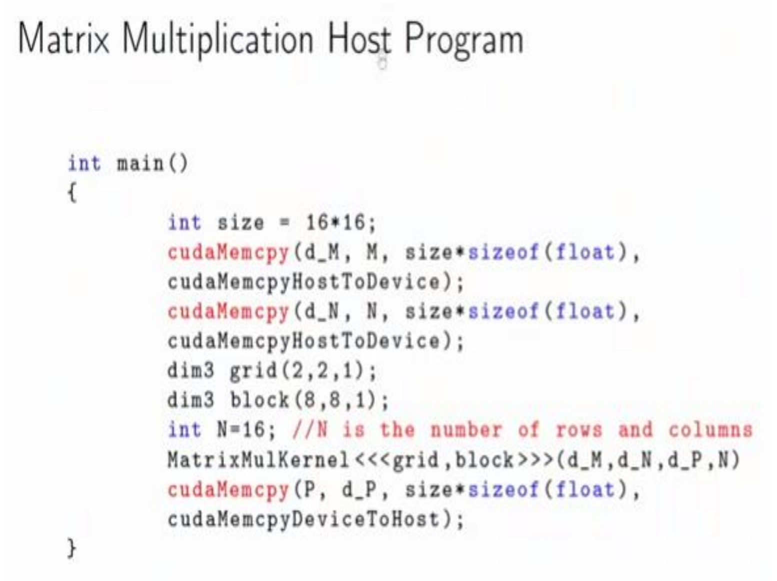
Hi so after the last lecture today I mean what we will be doing is we will reviewing a few more examples of CUDA program writing in terms of I mean we will start with simple CPU only programs and also show that how the corresponding CUDA variant of the program which will run on a GPU can be written. Now we will have some very simple examples for example we will we start with a basic matrix multiplication program I mean.

So first we consider only the CPU only code so this is the basic matrix multiplication kernel which is designed to run in a CPU. Let us see how it works as we know that any standard matrix multiplication should have 3 level nesting of loops so we have a outer loop which is essentially iterating over this i 0 to N and we have this inner loop j 0 to N. So basically I am using this i and j to select the ith pro and the of a matrix M and I am also using the inner iterated j to select a specific column and in that way once in every for every possible choice of i and j the inner lop carries out the computation of the ijth value of the result matrix.

There is the essence of any matrix multiplication program that we know a standard for any CPU only case. So essentially we have this outed 2 loops selecting the i and j and inside this inner loop we carry on this computation of the ijth entry of the output matrix as we can see that inside this loop what is going on. We are kind of traversing the ith row of the M matrix and we are traversing the jth column of the N matrix we are performing point wise multiplication followed by addition and they are result gets accumulated in the variable P value which is written back to the Pij matrix which is showing the result.

So that is how the standard CPU only code works and we like to see that what will be a CUDA variant of this course.

(Refer Slide Time: 03:04)



Matrix Multiplication Host Program

```
int main()
{
        int size = 16*16;
        cudaMemcpy(d_M, M, size*sizeof(float),
        cudaMemcpyHostToDevice);
        cudaMemcpy(d_N, N, size*sizeof(float),
        cudaMemcpyHostToDevice);
        dim3 grid(2,2,1);
        dim3 block(8,8,1);
        int N=16; //N is the number of rows and columns
        MatrixMulKernel<<<grid,block>>>(d_M,d_N,d_P,N)
        cudaMemcpy(P, d_P, size*sizeof(float),
        cudaMemcpyDeviceToHost);
}
```

So for CUDA variant as we know that they are as to be a host program and a device program so this is how the main will look like. So you will have a CUDA mem copy because of course we need to remember that we need to send the input matrixes M and N to the GPU side for doing multiplication right. So we execute this comments CUDA mem copy and you use them to send the matrix M as well as the matrix N to the GPU.

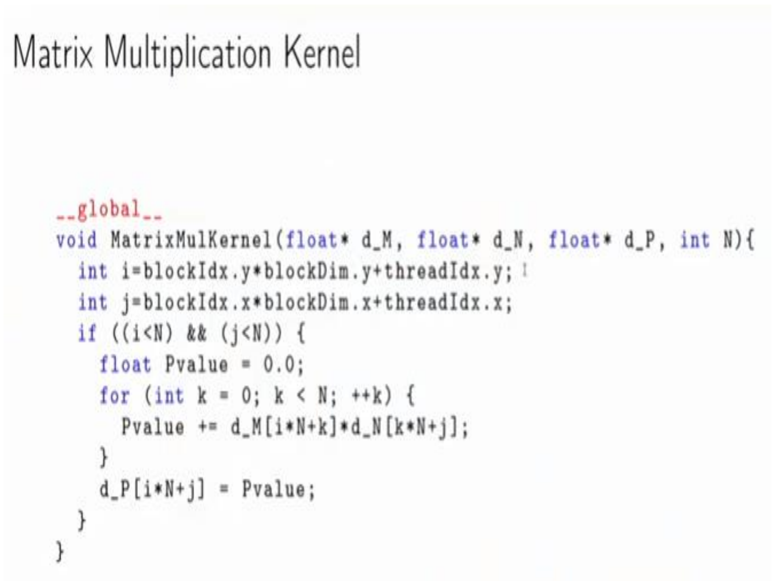
Once this is done we define suitable grids and blocks here as you can see I mean they are given some specific numbers here. So essentially we are defining a 2 dimensional grid and this is denoted by this 2, 2, 1 so we have a 2 dimensional grid in each dimension I have the dimensionality value of 2. So in that way I am having 4 thread blocks that are getting defined and then inside each thread block the block the I have 64 threads that is getting defined by this definition of block 8, 8, 1.

So inside each thread block I have 64 threads and they are also arranged in a 2 dimensional pattern right. So with the configuration of grids and blocks that defining the arrangement of threads for the program we are hiring the matrix multiplication kernel. So this is the code that is going to execute on the GPU and it is assuming that it has got the matrixes M and N available to it from the inside GPU dram and that is already available due to this CUDA mem copy that are already executed.

Once this kernel executes I have expecting that the results should be available in another part of the GPU device memory that is dP and I will be using this CUDA mem copy command again o copy back from device to host this values from the location dP to the original target CUDA location P in the CPU corresponding dram. Let us just have a small look at the basic matrix multiplication kernel essentially it is very simple code so it is going to execute this part of the code sequentially in a part thread basis.

The kernel has been launched with this many number of threads the threads are arranged in the grid comma block has been discussed. So if I ask what is the total number of threads that has been launched that would be let just work it out. So it would simply be this 2 cross 2 cross 8 cross 8 so that would give me 6 cross 4 that is the number of threads that are getting launched here.

(Refer Slide Time: 06:12)



Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int N){
  int i=blockIdx.y*blockDim.y+threadIdx.y;
  int j=blockIdx.x*blockDim.x+threadIdx.x;
  if ((i<N) && (j<N)) {
    float Pvalue = 0.0;
    for (int k = 0; k < N; ++k) {
      Pvalue += d_M[i*N+k]*d_N[k*N+j];
    }
    d_P[i*N+j] = Pvalue;
  }
}
```

So for each of this threads the thread id and block id variables will get their own specific values here. As we have learned earlier also so for example the first thread should have a block id 0 and thread id 0 here and so in that way I can use those values to compute an i and a j so this essentially is giving me what is the index for which I am going to do the computation. Now let us try to map this properly so what do I really have for I have got 1 thread that thread has a specific block id and a specific thread id right.

So if I do a block id x dot y times block dimension dot y + thread id x dot y then I get an i value and this i value is denoting a row of the matrix and opponent matrix 1 that is M or here that is revise side its dM and similarly. If I do a computation of this j what do I really get here. So have a block id x value and I am multiplying it with the corresponding dimension and I am adding it up with the thread id x value and this gives me a corresponding column position in the other matrix on the device memory and that would be dN.

(Refer Slide Time: 07:50)

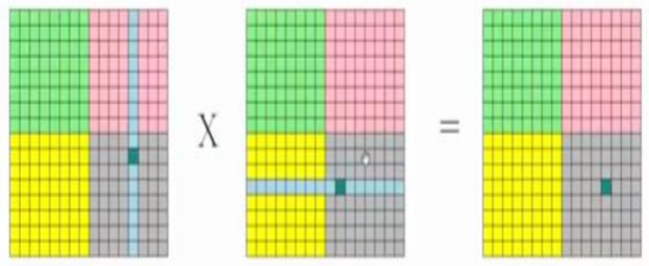


Figure: Matrix Multiplication

$$d\_P[i*N+j] = \sum_{k=0}^{N} d\_M[i*N+k]*d\_N[k*N+j]$$

So if we take this pictorial example and this gives me the more holistic view of the system so now have a look at the original code here. So you have defined a grid containing 4 thread blocks right and you have for each thread blocks you have got a block of 8 cross 8 size that means you have got 64 threads executive. So this is how it is going to be arranged so you have this many number of threads that have been launched each thread is trying to compute one of the elements of the target matrix right.

Question is how will each thread know for which value it is going to do the compute this is the main problem here. Now as you can see that we have 4 blocks due to this grid definition of 2, 2, 1 each of the blocks are getting marked here in different colors in 4 different colors I have and for inside each of the blocks I am going to have this 64 threads as per the block definitions we have discussed earlier.

Now suppose I pick up any one of this threads let us say the thread has got a this point I am talking about the thread at this point which is highlighted here. So if I ask what is the block id of this thread so it is easy to see that the blocks since the blocks at the dimensions from 0 so I have block id’s of this 4 blocks are 00, 01, 10, 11 right I would have for this thread the block id x dot x as well as block id x dot y value both as 1 right.

Now suppose I am trying to get back from this block id x and thread id x values what is the corresponding row and column position here in this target matrix for which the thread is going to do the computation. Now this is what is getting done here right so let us break it and observe it here so I have already understood that how I can get the block id x dot x and block id x dot y values.

Now whatever the values of the thread id x dot x and thread id x dot y values for this specific thread as we can see that here we are looking at the forth row. So 0, 1, 2, 3 so thread id x dot x is I say what is the thread id x dot y so that should be 3 and what is the thread id x dot x so that is this much right. So that is 0, 1, 2, 3, 4 so that should be 4 right so in that way looking back for that thread id I can compute what is the thread id x dot y value and thread id x dot x value.

I mean this at the value back the thread as already got and it has got to use this values to compute the i and j. So let us so that is like the motivation of why this equation is there but the issue is now to compute the value right. So the value of the row is i and the value of the column is j so i would essentially mean this height there is a row and j would essentially mean this shift there is a column right.

The height I would be now getting by following this formula so we have to look at what is the block id. If I am talking about i so essentially I am thinking in terms of what is the block id y that block id y needs to be multiplied by the dimension of the blocks so since this block id 1 and there was already a block here so you multiplied by the block dimension so you shift this much that is why you have block id x dot y times block dimension dot y right and then you are trying to figure out what is the shift in the y axis further inside this block.

Now that shift is as we know 0, 1, 2, 3 so that is the value of the thread id x dot y variable so that is why for computing the i we will have to write this block id x dot y times the block dimension dot y all in the y dimension and we have to add the shift that is the thread id x x dot y value. So this gives me the value of the row the target row for which I am going to the computation. Now similarly I can do a computation of the value of the x right sorry the value of the j for i have j.

Now j represents I mean what is the column index for which I am going to do the computation but how do I get the j in terms of xy axis as I can see that j should correspond to the x dimension of the block as well as the threads since the block id x dot x is 1 for this block so that means I would multiply it with the block dimension so that is what I do right. So block id x dot x times block dimension dot x and I would like to act with the thread id x dot x right so that is what I have plus the thread id x dot x.

So then in that way using this values of block id x dot x block id x dot y thread id x dot x and thread id x dot y I have been able to come to the i and j and that tells me what is the target location for which I am going to do the matrix multiplication operation after been able to recover this i and j value from the values of the system variables in terms of threads and blocks the kernel will proceed into this loop it has a loop which is iterating from k = 0 to N inside this essentially this thread would be traversing this row for matrix dM in the device memory and it will be traversing this column.

So I mean we are sorry so this picture should be actually I mean reverse like this a row should come on this side and this column should come on this side. So essentially will be multiplying them to get the value here right so just consider this one on this side and the column matrix on the opposite side of this. So anyway so looking at that the question is what should the expression here as you can see that the expression would nicely match.

So for every value of k all you are doing is in for a fixed row that is the ith row you have got i times N so you have already covered all the other elements visiting to this row and then you shift by k right so when I am talking about this entity I am referring to this matrix I think this is on the left hand side right. So I would do a i times N is the x as well as y dimension of the matrix and you shift by k position to get to any of the k positions here and your multiplying it with the this columns corresponding value.

So that is why you are also thinking that okay essentially we are doing computation of the ikth element times the kgth element. So the ikth element from this matrix is getting multiplied by the kjth element of this matrix right. So in that way you multiply the contents of this row point wise with the contents of this column again point wise and you are accumulating that sum and that sum goes to this position.

Now of course why do you do this multiply by N because as we know following c and other programming languages that I mean this is the way we always do for the simple reason that the arrays would be stored in the memory sequential. So coming back to the code for every thread the first thing that the kernel as to do is find out what is the location in the memory for which this kernel is suppose to do the matrix multiplication computation.

That means you are given a kernel when you are launch the kernel you launch that many threads you have launch the number of threads as is the size of the 2D matrix right. So 1 thread is responsible 1 thread is going to be responsible for doing a computation of each location in the matrix but it has to figure out what is the location. For that what it does is if it recovers the i and j value the way it does is it as a loop into the block id x thread id x values corresponding to the thread and it tries to figure out what should be the i and what should be the j using this i and j values it is going to find out.

Okay in that case I am picking up this row from the matrix M and this row this column from the matrix N and does the point wise multiplication and accumulate once this is done the value is stored in this variable P value and it is read back. So the question is all that is going on is for the GPU case the computation is done in a part thread basis the possibly problematic scenario here you have to identify that how to map the computation in terms of thread id's and block id's.

So like you have you are firing the kernel with a suitable launch parameter set that is specifying how the threads are going to be arranged hierarchically and then you are using this launch parameters to identify your working set part thread in this case the working set for this thread is this location ij. And we have explained how to compute the value of i and j block id's and thread id's and you are going to use them for doing the computation.

So in that way you have the matrix multiplication operation done now one thing as to be remember that this is not an optimize GPU program why I say that this is not an optimize GPU program we learnt later in the course where we show that how this matrix multiplication computation can be accelerated by using several primitives that GPU additionally offered to you. Here all we are trying to do is we are trying to parallelize the program.

We are simply saying that the operation of matrix multiplication have got lots of parallelism because I can compute each of the ijth element of the product matrix come independently there are no conflicting rights that are going to happen in the product matrix space. Due to this region I can launch a GPU kernel having as many threads as is the size of the matrix so that each thread is responsible for computing 1 element of the product matrix and the way it does is as we have already discussed.

That it will recover the working set and then it will decide how to do the computation and so every thread is executing this sequence of code completely sequentially and when this kernel is completed we would have got the result available in the GPU memory and from the GPU memory the data has to be copy back to the CPU memory by using the CUDA main copy command.

So that would be our way of doing the matrix multiplication computation so with this we will complete this simple example of matrix multiplication and thank you for your attention and in next we will go through another a bit more example and this will be part of our coverage of basic CUDA program without much of optimization thank you.