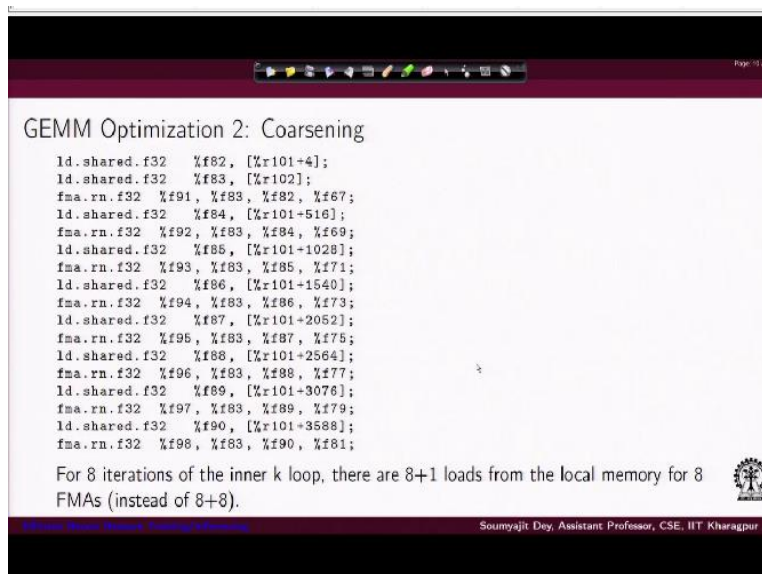


GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-61
Efficient Neural Network Training/Inferencing (Contd.)

Hi, welcome back to our lecture series on GPU architectures and programming.

(Refer Slide Time: 00:28)



The slide displays the following CUDA code for GEMM optimization:

```
ld.shared.f32 %f82, [%r101+4];
ld.shared.f32 %f83, [%r102];
fma.rn.f32 %f91, %f83, %f82, %f67;
ld.shared.f32 %f84, [%r101+516];
fma.rn.f32 %f92, %f83, %f84, %f69;
ld.shared.f32 %f85, [%r101+1028];
fma.rn.f32 %f93, %f83, %f85, %f71;
ld.shared.f32 %f86, [%r101+1540];
fma.rn.f32 %f94, %f83, %f86, %f73;
ld.shared.f32 %f87, [%r101+2052];
fma.rn.f32 %f95, %f83, %f87, %f75;
ld.shared.f32 %f88, [%r101+2564];
fma.rn.f32 %f96, %f83, %f88, %f77;
ld.shared.f32 %f89, [%r101+3076];
fma.rn.f32 %f97, %f83, %f89, %f79;
ld.shared.f32 %f90, [%r101+3588];
fma.rn.f32 %f98, %f83, %f90, %f81;
```

For 8 iterations of the inner k loop, there are 8+1 loads from the local memory for 8 FMAs (instead of 8+8).

The slide footer includes the text: "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur" and a small logo on the right.

So if you remember in the previous lecture, we have been talking about optimizing the GEMM computations. So we discussed one optimization, which was thread coarsening application of thread coarsening over a tile matrix multiply.

(Refer Slide Time: 00:39)

GEMM Optimization 3: Wider loads

- ▶ In the previous implementation we increased the amount of work in the column-dimension of C.
- ▶ The same optimization trick can be done for the row-dimension
- ▶ The additional advantage for optimizing across the row dimension is using wider data-types.
- ▶ Increasing the work per thread (WPT) in the row-dimension of C can be done by considering vector data-types instead of loops over WPT.
- ▶ NVIDIA GPUs do not support vector operations (such as multiply or add) in hardware but possess special wider load and store instructions both for the off-chip and the local memory.

Page 11/10

© IIT Kharagpur

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

Now, let us start with the next optimization. So, if you remember our previous optimization, we increased the amount of work per thread, and that was in the column dimension of the output, right. Well, why do we say it is the column diamonds because when we are giving the examples, essentially we are talking in the rows, but since it is basically in the column major format. So of course, it will be the, I mean, it is whatever is the row major format of the mathematical matrix is actually the column dimensions here, right. Now, we can just apply the same optimization trick for the other dimension right.

(Refer Slide Time: 01:24)

GEMM Optimization 3: Wider Data Types

```

19 for (int k=0; k<TS/WIDTH; k++){
20   vecB = Bsub[col][k];
21   for(int w=0; w<WIDTH; w++) {
22     vecA = Asub[WIDTH*k + w][row];
23     switch (w) {
24       case 0: valB = vecB.s0; break; case 1: valB = vecB.s1; break;
25       case 2: valB = vecB.s2; break; case 3: valB = vecB.s3; break;
26       case 4: valB = vecB.s4; break; case 5: valB = vecB.s5; break;
27       case 6: valB = vecB.s6; break; case 7: valB = vecB.s7; break;
28     }
29     acc.s0 += vecA.s0 * valB; acc.s1 += vecA.s1 * valB;
30     acc.s2 += vecA.s2 * valB; acc.s3 += vecA.s3 * valB;
31     acc.s4 += vecA.s4 * valB; acc.s5 += vecA.s5 * valB;
32     acc.s6 += vecA.s6 * valB; acc.s7 += vecA.s7 * valB;
33   }
34 }
35 __syncthreads()
36 }
37 C[globalCol*(M/WIDTH) + globalRow] = acc;
38 } // Launch parameters: <<<(M/TS, N/TS),(TS/WIDTH, TS)>>>

```

Page 11/10

© IIT Kharagpur

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

But how do you really do it. Well, the way you can do it is by using wider data types. Now, what is a wider data type, we will soon see. But let us understand how this is really going to help. First

of all for these optimizations we will not carry over the previous optimization here, although we understand that using both optimizations together can help. So now let us consider that we are just considering how our wider load instruction can really help me right.

So this is actually done by using vector data types instead of loops over the work parts are variable that we discussed earlier, right. Now, in NVIDIA GPUs, we do not have support for vector operations like vector multiply or vector add, like vector operations in CPUs, and of course, we do not need them because we are having multiple scalar units, which work using the idea of warps in all that.

But so we do not have vector operations. But there is something called wide load and store instructions both for the off chip, as well as the shared local memory. So what does that mean well, I am not going to have multiple add additions or multiplication operations have been on vector data, but I can have vector kind of wide loads, right, I can load multiple data together using a suitable vector data type.

Let us see how that really works. So when I am really using a wide data type one example would be this float8. So essentially, if I declare a floating8 kind of variable, that means, let us say I am now making this A_{sub} and B_{sub} array arrays which are in my shared memory, I am declaring them as of type float8. And that would mean, each element in that array can accumulate, 8 constituting floating point data fine.

So that also means that well, I do not need that much width of a sub that was there earlier right So I decrease the column dimension here for A_{sub} by TS by width, right. There is a Ts tile size, right. So that is how now my A_{sub} and B_{sub}s will get reduced with respect to columns. But on the contrary, what is happening each of the elements are getting wide by 8, right, each entry in the matrices will now be having 8 consecutive floating point data.

Again, where we really trying to do it well, we are trying to increase the total amount of data that is there I mean, values get loaded both from the global memory. So from global memory when I am using one load instruction and making it explicit that you load these many together right and

put it in a wide variable, and similarly consider this as a shared memory and you are trying to load it into the GPU register for doing some operation.

That will also be a wide load from the shared memory. So now, of course, the usefulness of the instructions can be exploited only if we can change our programs with W. So what will be the changes. Now let us understand the implication here.

(Refer Slide Time: 04:54)

```
1 __global__ void GEMMS(const int M, const int N, const int K,
2 const float8* A, const float8* B, float8* C) {
3 //Code for thread identifiers
4 // Modify shared memory initialization
5 __shared__ float8 Asub[TS][TS/WIDTH];
6 __shared__ float8 Bsub[TS][TS/WIDTH];
7
8 float8 acc = { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f };
9 const int numTiles = K/TS;
10 for (int tile=0; tile<numTiles; tile++) {
11
12     const int tiledRow = (TS/WIDTH)*tile + row;
13     const int tiledCol = TS*tile + col;
14     Asub[col][row] ← A[tiledCol*(N/WIDTH) + globalRow];
15     Bsub[col][row] ← B[globalCol*(K/WIDTH) + tiledRow];
16     __syncthreads()
```

So what we are really looking at now is something like this that you have. So let me first draw one tile. So in this tile we are seeing that well. So this one tile inside that entire C matrix, something like that, right. Inside that tile, I am saying that well, the thread should compute, let us say a wide number of, let us say some multiple entries together, right. And again, I will say I am trying now in the column dimension.

Because now they are going to be these are the things that will be accessed in parallel, right. So, now, if you see that how the wide loads are going to happen here. So again, we have the outer loop tiles to number of tiles, which remains tiles, iterating from 0 to number of tiles, which remains same. And here I have the loading part to the shared memory right. If you remember in the previous course and version, we introduced a loop here with that is not going to be there.

Because now I am not applying thread coarsening, I am just playing wide loads right. So, now, when I am defining this operation that will come from the global A and B arrays, you load values to A_{sub} and B_{sub}. So, this since the data type here is this float8. So, I will have the wide vector loads operating right. So, with each load I will get multiple let us say since this float8. So, I will get I mean 8 consecutive float values getting loaded into the tiles right.

But then there is a problem right let us understand. So, I mean, if the mathematical arrangement of the data was like this, and like we discussed that you will have 1 row here to traverse and you have multiple columns to traverse right. So, you multiply like this, these in the mathematical domain, but now when you are trying to work it out for this many data points here. What do you really need in the input tiles that are A_{sub} and B_{sub}.

So, if you have to get this done, then you need data here like this. And you need values like this right. There is a requirement, you want to work on them, right. And they are all located, right. These values, since it is a column major format, they are all located in one entry of the matrix, right. These values again, they are located in one entry of the matrix. These values, they are again located in one entry of the matrix.

But you do not really want to multiply this entry with this entry, right. Because since this, you really want to multiply these entries with these entries. Get this fellow, but using normal mathematical matrix multiplication, I hope this is clear and let me draw this thing again in a better way.

(Refer Slide Time: 08:09)

```

19 for (int k=0; k<TS/WIDTH; k++){
20   vecB = Bsub[col][k];
21   for(int w=0; w<WIDTH; w++) {
22     vecA = asub[WIDTH*k + w][row];
23     switch (w) {
24       case 0: (valB = vecB.s0); break; case 1: valB = vecB.s1; break;
25       case 2: valB = vecB.s2; break; case 3: valB = vecB.s3; break;
26       case 4: valB = vecB.s4; break; case 5: valB = vecB.s5; break;
27       case 6: valB = vecB.s6; break; case 7: valB = vecB.s7; break;
28     }
29     acc.s0 += vecA.s0 * valB; acc.s1 += vecA.s1 * valB;
30     acc.s2 += vecA.s2 * valB; acc.s3 += vecA.s3 * valB;
31     acc.s4 += vecA.s4 * valB; acc.s5 += vecA.s5 * valB;
32     acc.s6 += vecA.s6 * valB; acc.s7 += vecA.s7 * valB;
33   }
34 }
35 __syncthreads()
36 }
37 C[globalCol*(M/WIDTH) + globalRow] = acc;
38 } // Launch parameters: <<<(M/TS, N/TS),(TS/WIDTH, TS)>>>

```

So we will just say I want to compute more elements in the matrix row. So I will employ a wider load instruction. This will help me to compute multiple values together. If I employ the wider load instruction, then what I get here in the Asub and Bsub, so this is Bsub and this is Asub. So then I have data here like this. And due to collaboration I am drying N terms of 4 instead of 8. Just because of course, we need to get the meaning correct.

And the wider load instructions here are giving me values like this, right. The point I am trying to make here is well, to compute this, we understand that we have to go like this. But how is the data there in the variable, there is a problem, the data is there, these consecutive data points are there in one location of Asub, the next set of values are there in the next location of Asub, right there, that would mean, after now look inside the value and segregate individual parts from the value.

And then use them to do component wise multiplication and compute this values right. Let us figure out what does that really mean. So now after the loading is done, which is very simple. Let us go to the compute loop and see how things are going to change. So of course, this is the outer loop for the computation.

(Refer Slide Time: 09:51)

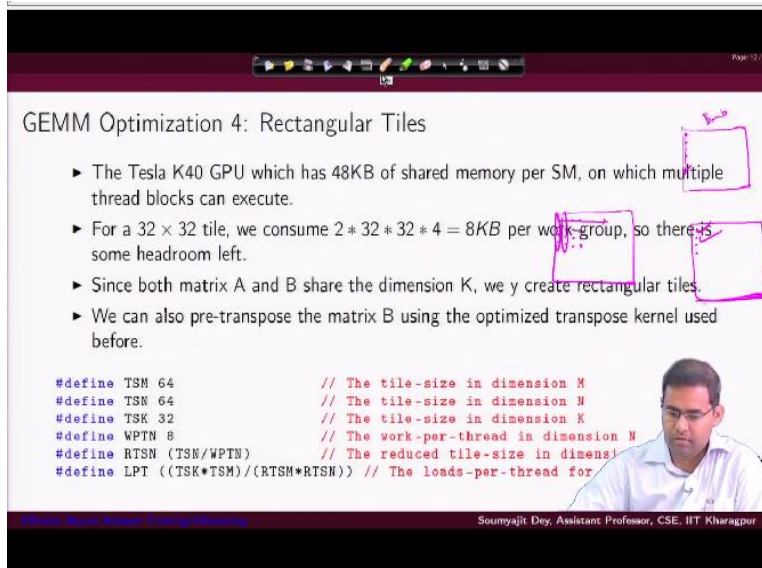
GEMM Optimization 4: Rectangular Tiles

- ▶ The Tesla K40 GPU which has 48KB of shared memory per SM, on which multiple thread blocks can execute.
- ▶ For a 32×32 tile, we consume $2 * 32 * 32 * 4 = 8KB$ per work-group, so there is some headroom left.
- ▶ Since both matrix A and B share the dimension K, we create rectangular tiles.
- ▶ We can also pre-transpose the matrix B using the optimized transpose kernel used before.

```

#define TSM 64          // The tile-size in dimension M
#define TSN 64          // The tile-size in dimension N
#define TSK 32          // The tile-size in dimension K
#define WPTN 8          // The work-per-thread in dimension N
#define RTSN ((TSN/WPTN)) // The reduced tile-size in dimension N
#define LPT ((TSK*TSM)/(RTSN*RTSN)) // The loads-per-thread for

```



© Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

This is the outer loop for the computation, right. But the issue is with the inner loop because as I am saying, that for computing each of these values, I need access to consecutive hellos in Asub, because each of the values here they are of vector type. And I need to look into each of the components. So what do we do. Well, we take a piece of value in some vector B, which is fine, because I need these values which are anywhere together.

But for Asub, I have to have a loop, which will run through the entire width, which is 8 in our implementation, consider it 4 right. In each iteration, I take 1 element of Asub, in the first iteration, I take 1 element of Asub, what am I really supposed to do, I am supposed to look into each of these components individually, multiply them by each of these components, and store them as partial sums here.

In the next iteration of this inner loop I am going to take the next Asub value which is again, accessible as one of this, right. But then I have to look inside it into its components. And I have to multiply these components, again with the piece of values. I hope this is clear. What is the problem because I have them in these columns, right. Since they are in columns, but actually I need to compute like the rows.

So I have to look inside the components. That is what is done by the switch case. So here, as I progress through this inner loop, with respect to W equal to 0 to width, what really am I doing in

each iteration of this loop I pick up one of these A_{sub} values, and then I sample out. Well, for each thread, I am trying to sample out which component of this vector I would be interested in, right because A_{sub} I know that well sorry.

I know that I have picked it up and I have stored it in $vecA$ right. But I have to realize now that for the other vector this one, which are the components with which I am going to multiply them, right because as you know that for $vecA$ this first component, sorry again the first component of $vecA$ needs to be multiplied with the first component of the $vecB$. Of course, let me first define, so $vecB$ is for storing this right.

And $vecA$ is storing this value right now, in the next iteration, $vecA$ will store the next value like that, right. Also, we need to figure out a way to access the components, now for float data types, the ways that you just do a normal structure like computation, like so, for this vector $vecA$, you dot s_0 that gives you the first component dot s_1 is going to give you the second component, so on so forth. So what do I really need to do.

So you take $vecA$ and you need to multiply if a case s_0 , I am speaking for W equal to the 0 case, right. So what do you need to do. You need to take $vecA$'s s_0 component and multiply it with $vecB$'s s_0 component right. So for W equal to 0, I need $vecB$'s s_0 component. So that is why I pick up this right and here what am I doing I am multiplying $vecA$'s s_0 component with this and thus given me the partial sum for this.

But parallelly I have other things also right, for this right now I have with me in $valB$. I have vector B 's s_0 component. Well, do I need it for some other computation. Of course, I would need it here for s_0 . But in future when I pick up the next $vecA$, I would again need it, right. So, let this so here what is happening for when I am accumulating for the s_0 component this is what I am doing. But what about the other components.

So as you can see, right now, for all the other components, also, let us say this component, I would need $vecA$'s component s_2 , let us say, the third component that is s_2 , I would need $vecA$'s s_2 component to get multiplied with s_0 , right, which is already in $valB$. So that is what

takes place here right and similarly for all the components, I hope we are making some sense here. So essentially, what am I really doing.

By using this switch case, we are taking out this variable, the first value and then I am in by all these accumulation statements, we are multiplying this s_0 component of the vecB with all the different components of vecA right. And that is what I require, I required all these values to get multiplied with this as a partial sum for here right, which again I have tabular later on. And this continues right with the loop making progress.

Now, I will take the next value of vecA right and for each component of vecA , I will name again I have to sample out what to do for which component of vecB to pick, which will be the next component, and I will now use this component for all the other ones right. So again, as you can see, there is a lot of saving happening. Like for with one load, for this vecB whatever component I am getting, I am choosing that appropriate component.

And I am doing the multiplication of that component with everything else, for the first component of the vector A, right, and that way this computes right. So just to summarize for this multival values, for each of them, I am making a vector load like this. And I am multiplying these vector's different components with the components that are here, right, I am just choosing which component to multiply.

Because if we can understand for this computing this value, I need this one to get multiplied with this for computing the next value I need the next one that is vecA dot s_1 to get multiplied with the same value. For the next one again, I need the vecA dot s_2 to get multiplied with the same value. That is why I am choosing this value here using the switch case and I am using that as valB in all the accumulations.

Again as I progress, I will choose one component here right, I will choose the second component, right for vecB , this value. By the way, it is important to note that vecB is loaded once and there is getting used entirely in this loop, right vecB is not changing. In one iteration of

the loop as long as I am inside this inner loop, right, I am inside this inner loop, I have this vecB, right. And here, inside this inner loop, I have this vecB.

So once I load it for each of the components of them, I am just figuring out which component to multiply with the different components of vecA. And I am just using it. So, I load vecB once, I load vecA multiple times the number of times that is required to fully do the compute of the final entries. And in each case I select the suitable component from vecB and do a multiplication with all the components of vecA. And that is how it progresses.

So, as you can see that this is just a different idea with respect to thread coarsening, what we are doing is we are for part thread we are doing wider loads, so that I get more amount of data per load, and then that gives me a problem like the data is all in columns to use them in a suitable way I make use of this intelligent loop structure here.

Now, coming to the next optimization, which is rectangular tiles, so, what is this. So, if we see in our previous examples we just use tiles of size I mean we just use I mean earlier for also for matrix multiplication, typically we use tiles of size 32 cross 32. Now that requires how much memory well, 2 of the shared memories, each of size 32 cross 32 right. So 32 cross 32 times 2 multiplied by 4 bytes because it is float.

So 8 KB per work group, right or let us say per block I mean like that I have this per work group 8 KB, right. So that is not much. Why, because even if we consider as Kepler's old GPU like Tesla K40 it has 48 KB of shared memory. So, I have more shared memory there, right. So, what I can do is, I can create bigger tiles right. So, it is a small mistake here.

So, this is called creation of rectangular tiles, right. And also there is something important that I can do that since I am multiplying A and B together. And as you can see, from B I am doing an access and so far A that data since it is a major representation. So, that means for A the columns are A are they are in the memory right. But in for B I mean for B is basically I need the data in the other way around, right.

So, as we know that when you are multiplying matrices and load loading data for one I will get this memory coalescing effect for loads or for the other I will not get it right. But a good way to optimize that is that well, we transpose the other matrix. So, in this case, I can just transpose B right. So, because if you see that for B I have the access which is not memory friendly. So, I will just transpose B.

Because we have already figured out earlier in our discussions with respect to matrix transpose computation, it is really possible to do very efficient transpose computation, where I mean, the amount of time required for doing a transpose is not too high with respect to a normal memory copy, right. So assume that we first do a transpose so that the matrix B is available in the global memory in a transposed way.

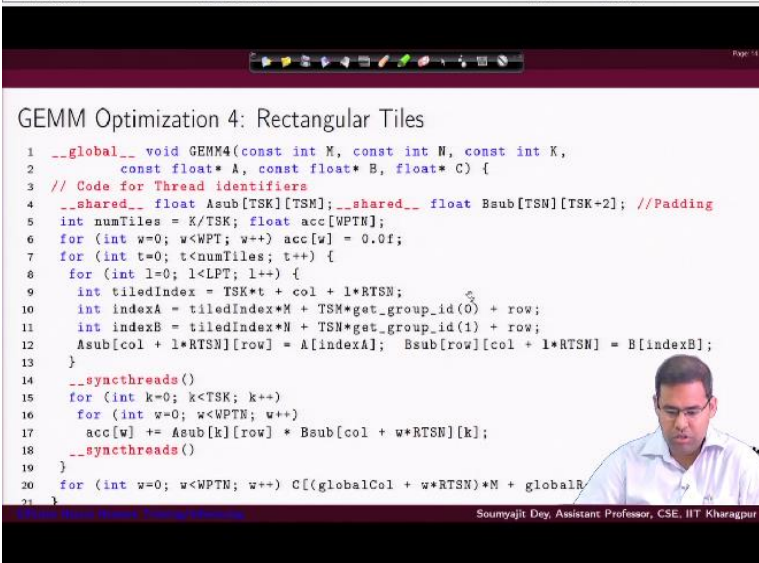
Well, what does that help in. Well, now when I have B in transposed with now when I am bringing data, I get the good effect of memory coalescing happening for both matrix A and B right. So now let us just go through some basic defence what we do is so let us say TSM define the tile size in dimension M TSN for dimension N and the tile size in dimension K is 32 okay, and also, we will increase the work per thread.

So, we will bring in the previous optimizations right. So, we will bring in the work per thread coarsening optimization, let us say 8 and with this what is the reduced tile size in the dimension. So, as you can see that since I have more work per thread, so, the tile size in the dimension N would be used if you just take the actual dimension divided by the work per thread right.

So, in general, what is the total number of loads per thread for a single tile. How many loads you really have to do in a single time. Well, that would be you multiply the tile size in dimension K multiplied by the tile size in dimension M you divided by the reduced tile size right for both the dimensions, right just like RTSN, I can have an RTSM right. So you have the original tile size in the dimensions, you just reduce it.

I mean, you just divided by the reduced tile size because now you are going to do more activity per thread. So that tells you what is the number of loads that you have to do for per for each thread. So let us call it LPT loads per thread right.

(Refer Slide Time: 23:18)



```
1 __global__ void GEMM4(const int M, const int N, const int K,
2   const float* A, const float* B, float* C) {
3   // Code for Thread identifiers
4   __shared__ float Asub[TSK][TSM]; __shared__ float Bsub[TSM][TSK-2]; //Padding
5   int numTiles = K/TSK; float acc[WPTN];
6   for (int w=0; w<WPT; w++) acc[w] = 0.0f;
7   for (int t=0; t<numFiles; t++) {
8     for (int l=0; l<LPT; l++) {
9       int tiledIndex = TSK*t + col + 1*RTSN;
10      int indexA = tiledIndex*M + TSM*get_group_id(0) + row;
11      int indexB = tiledIndex*M + TSM*get_group_id(1) + row;
12      Asub[col + 1*RTSN][row] = A[indexA]; Bsub[row][col + 1*RTSN] = B[indexB];
13    }
14    __syncthreads()
15    for (int k=0; k<TSK; k++)
16      for (int w=0; w<WPTN; w++)
17        acc[w] += Asub[k][row] * Bsub[col + w*RTSN][k];
18    __syncthreads()
19  }
20  for (int w=0; w<WPTN; w++) C[(globalCol + w*RTSN)*M + globalR
21 }
```

Now, once we start using all this, will I mean, again, we will not try to discuss the entire thing in great detail, but we will just try to identify that how things are really going to help us. So, first thing, the first optimization here for rectangular tiles is that well, we have taken B as a already transpose, right. So that would give me some acceleration when I am loading data from the global memory for both A and B, right. But then there is a problem. When I am loading the data, if you look at the previous quotes.

(Refer Slide Time: 24:08)

```

GEMM Optimization 2: Coarsening

1  __global__ void GEMM2(const int M, const int N, const int K,
2  const float* A, const float* B, float* C) {
3  //Code for thread identifiers
4  //Code for initializing Local memory
5  const int numTiles = K/TS; float acc[WPT]; // WPT-> Work per thread
6  for (int w=0; w<WPT; w++) acc[w] = 0.0f;
7  for (int t=0; t<numTiles; t++) {
8  for (int w=0; w<WPT; w++) { //RTS = TS/WPT : Reduced Tile Size
9  const int tiledRow = TS*t + row; const int tiledCol = TS*t + col;
10  Asub[col + w*RTS][row] = A[(tiledCol + w*RTS)*M + globalRow];
11  Bsub[col + w*RTS][row] = B[(globalCol + w*RTS)*K + tiledRow];
12  }
13  __syncthreads()
14  for (int k=0; k<TS; k++)
15  for (int w=0; w<WPT; w++)
16  acc[w] += Asub[k][row] * Bsub[col + w*RTS][k];
17  __syncthreads()
18  }
19  for (int w=0; w<WPT; w++) C[(globalCol + w*RTS)*M + globalRow]
20  } // Launch Parameters: <<<(M/TS,N/TS),(TS,TS/WPT)>>>

```

Pravin Kumar Shrivastava, IIT Kharagpur
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So let us look at the previous code of quotes and version which was optimization 2. So, this is how I was doing the load right. So from B, I was transferring the data here in B, right, which in the shared memory representation of B, right. So, that was all nice when from B I was loading well the load was not coalesced. But the store in the shared memory was really happening in consecutive locations.

So that when I read the data, it is already in a nice way. It is arranged properly right. But now, when I am loading from B transpose, I will have a opposite effect here, right. When the data is getting stored in the shared memory, I actually lose the advantage which I gain for the shared load, by doing the load of the transpose. Well, does that mean that I do not want to do a B transpose from B.

No, I really want to do it, why because global loads are expensive. I really want to minimize the time there. So definitely I will do it. But what I will also do is, when I store here in the shared memory, I will flip the indices. So that is what we do. If you see here, I have row here in the second index. But for rectangular times or rectangular stores we will actually load in the opposite way. Yeah, so this was your original coarsen code for optimization 2.

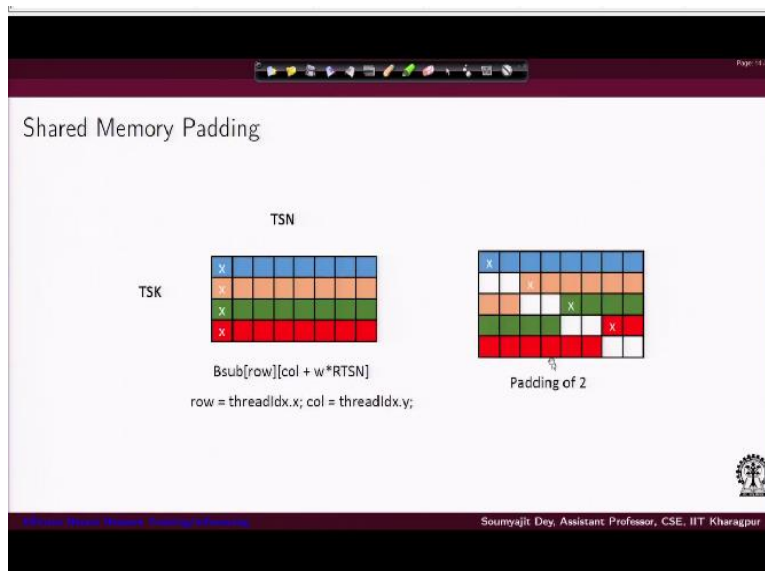
As you can see, I have flipped the indices for Bsub here. So row comes first right. So, that means, well I am loading in a nice coolest manner, but also to preserve the advantage of storage

in the shared memory and not losing out on that what I do is a flipping this as a fine, I hope that clears your. So apart from it, the rest of the things are fine only issue is you may just to check that we have used a bit of open CL semantics here like get group ID.

And all those which of course, is a CUDA code we are trying to see and so that you may just replace it with a suitable CUDA format here fine. So, once that is done, we have got an advantage that now my loads are optimized. And by flipping these parameters here, I have also kept the advantage of having consecutive data in shared memory in a proper way. And then as you can see, you have a normal sync thread.

And then a computation of the data just like previous things, but there will be one problem here, which you have to understand.

(Refer Slide Time: 27:23)



That here again, when you are going to load the data from the shared memory, that is these operations when you are going to load the data from the shared memory like this. So, this is where you are doing the accumulation right. Now, just to remember that, what really we did. We did a qualis load we enable qualis load by doing a transpose previously, but then to get the advantage of having a nice in nice consecutive operations in the shared memory.

I have flip the storage here, but that also creates a problem because when after I flip the storage, what I get is the data gets stored in the shared memory like this. So then that would mean that I will have lot of bank conflict. When I am now loading the data from the shared memory to the accumulation loop. Again, this is a bit tricky so let me just repeat, I just, I just made nice qualis loads from the global memory.

But since it is all transposed data, so when I am putting it in the shared memory, it is coming for the mathematical matrices, columns, right. So and that that is not the exact order in which you would like to access them, right. So in order to alleviate that problem with respect to access of the shared memory, you have switched columns here. Now once you have switched columns here, the problem that comes with respect to the warps which are going to access this shared memory, just look at the way the warps are going to access the shared memory.

So here, when you are accessing this shared memory B_{sub} , you are moving by columns, right. You are operating on columns, well because this is just for your elevating your issue with the storage. But now when you are going to operate the data from the shared memory, you are accessing first by columns, and then your warp moves over to the next K , right. Now, this would give you a problem because now, you will have lot of bank conflicts.

So, we will do a simple optimization that we have studied earlier, which was just putting in extra locations which are just padding locations. So that earlier without padding, that is having extra locations in your shared memory in your array in the shared memory, your data would have come in like this, but now your data would be coming in like this because there is a first data, right and then after all the padding's in the next data point is here, next one is here, next one is here.

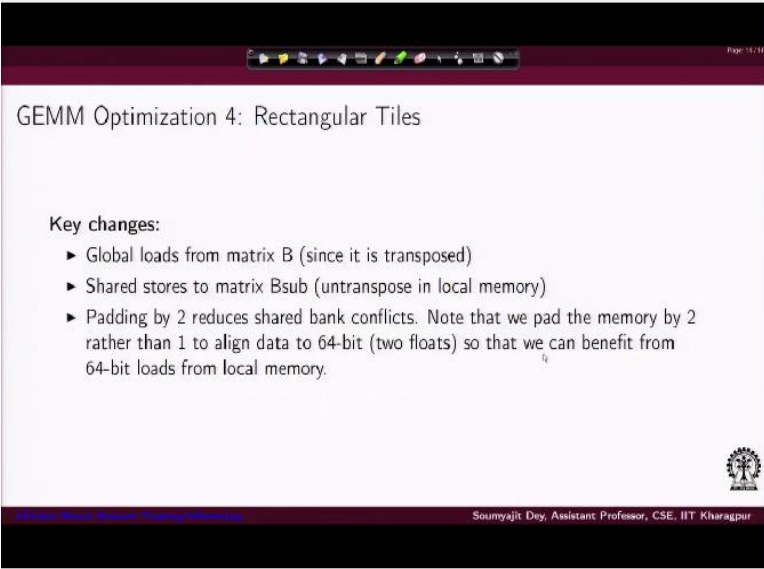
So, these are the data points which the warp will actually access right. I hope this is clear, but still I will just repeat. So, we have already understood why transpose is necessary. But as you understand the difficulty with transpose will be that when you are next I mean, of course, in the shared memory you are not having the regularity of the data that you are having earlier because

you have optimized the loads, but now your stores have actually disturbed the constitutive nature of the data.

In order to preserve that you have switched the row and column indices here. But then the problem due to this would be that when consecutive thread Id's on the warp will be accessing the shared memory, we would get their consecutive equal data points in the same bank. For alleviating that we use the padding trick that we learned earlier, which is just to add extra locations here.

All you do is you will just add extra locations put in more than that is required. And that would actually again offset the locations here. So that when the warps are going to load the data, you avoid the bank conflicts fine.

(Refer Slide Time: 31:15)



The slide is titled "GEMM Optimization 4: Rectangular Tiles". It lists three key changes:

- ▶ Global loads from matrix B (since it is transposed)
- ▶ Shared stores to matrix Bsub (untranspose in local memory)
- ▶ Padding by 2 reduces shared bank conflicts. Note that we pad the memory by 2 rather than 1 to align data to 64-bit (two floats) so that we can benefit from 64-bit loads from local memory.

The slide also features a small logo in the bottom right corner and a footer with the text "Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur".

So, that is the good thing about rectangular tiles, and so helps you in optimizing global loads. And so the second important thing is you made the changes to optimize the shared stores. And then you padded the extra locations to avoid bank conflict. Now, why 2 locations, why not 1 location that is also an important thing, right. The reason is in your shared memory now, for Kepler and other next generation architectures, the banks are 64 bit wide, right So that means you have 2 consecutive data points sitting in the same bank anyway, right. So if your warp is accessing 2 consecutive data points, this is going to have a bank conflict.

So, if you are padding of 2, then you are forcing that well this is the memory location as I am showing right. So, this is one bank and this is the next bank right. So, in that way you are forcing that I mean in 2 consecutive threads in the warp access different banks and there is no conflict, right. So, this is the important thing that why the padding is by a factor of 2. Because the warps are going to do 64 bit loads from the local memory. And the loads are 64 bit per thread since the shared memory pad which is also 64 bit fine.

(Refer Slide Time: 32:42)


Page 14/14

GEMM Optimization 5: 2D Register Blocking

Key changes:

- ▶ Increase the work per thread in both row and column dimensions.
- ▶ 2D register blocking is very similar to for 2D tiling, but at a different memory level
- ▶ Key optimization is to reduce shared memory traffic than optimizing from global memory off-chip traffic.

14



© 2019, IIT Kharagpur. All rights reserved. Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, with this we will end the current lecture and then the next lecture we will look for some other optimizations. Thank you for your attention.