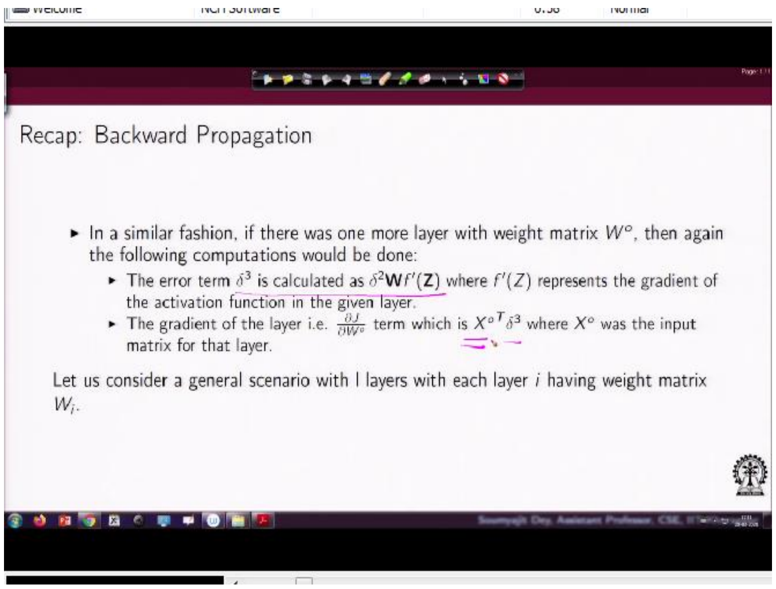
GPU Architectures and Programming

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

Lecture-60
Efficient Neural Network Training/Inferencing (Contd.)

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

(Refer Slide Time: 00:27)



Recap: Backward Propagation

- In a similar fashion, if there was one more layer with weight matrix W^o, then again the following computations would be done:
  - The error term δ^3 is calculated as $\delta^2 \mathbf{W} f'(\mathbf{Z})$ where $f'(Z)$ represents the gradient of the activation function in the given layer.
  - The gradient of the layer i.e. $\frac{\partial J}{\partial W^o}$ term which is ${X^o}^T \delta^3$ where X^o was the input matrix for that layer.

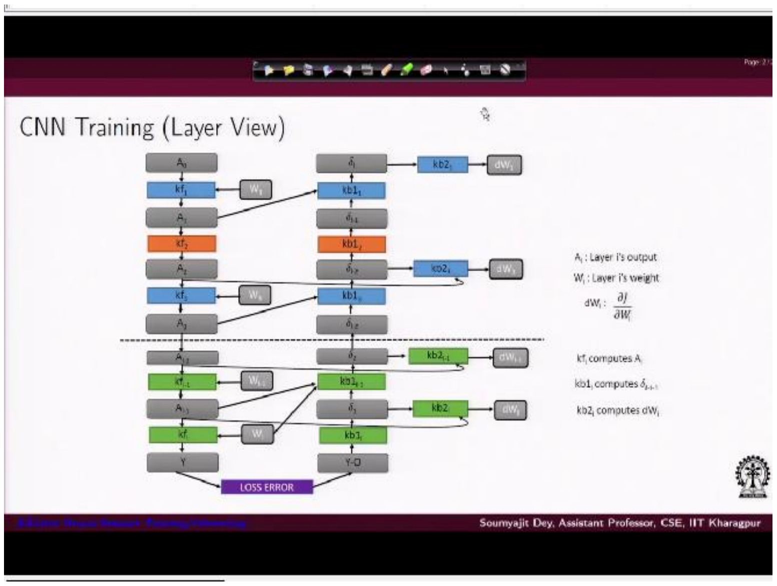
Let us consider a general scenario with l layers with each layer i having weight matrix W_i.

So we will just revise once again our previous coverage. So if you remember we have been talking about how to build the dependency graph for the CNN style computation. And in general, we also visited how the math for backpropagation works. And what we could see is that in general, if we have a cascade of layers, and we are trying to compute some error term for a specific layer.

So in general, the way we compute it is that is basically to multi the error term that I figure out the delta 2 for the previous layer, multiplied by the weight of the previous layer, and the gradient of the activation function in the given layer, right. And so, and this in general would continue and this gives me the error term, and you multiply that error term with the previous layers, or whatever is the input with respect to that current layer.

For example, if it is the input layer, then you multiply the currently computed error term with the input matrix. In general, if it is any intermediate layer, then the error term gets multiplied with the previous layers, output activation, right.

(Refer Slide Time: 01:49)



CNN Training (Layer View)

A_i : Layer i's output

W_i : Layer i's weight

dW_i : $\frac{\partial J}{\partial W_i}$

kf_i computes A_i

$kb1_i$ computes δ_{L-i-1}

$kb2_i$ computes dW_i

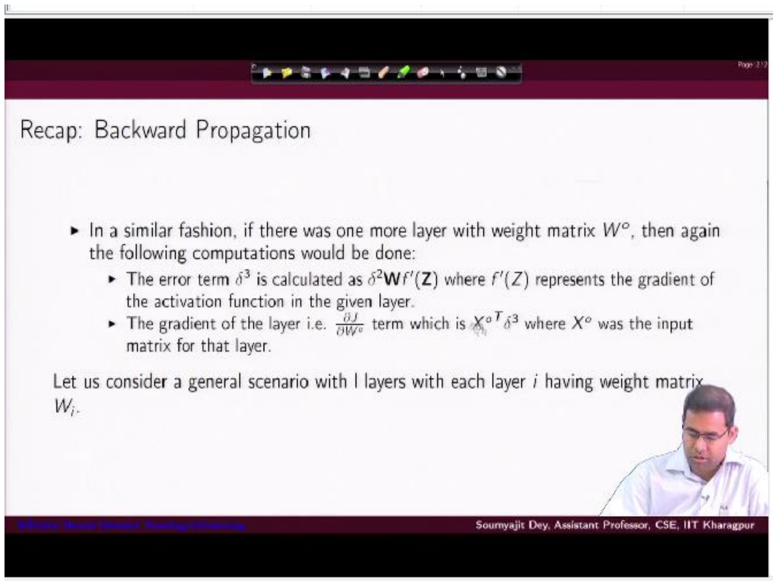
Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So with that, what we figured out was that if we look forward, that how the pipeline computes the different indices that we discussed earlier, and we thought that let there be the following kernels in the kernel k f i, which computes the forward the activation output A i of some ith layer, some kernel kb 1 i, which computes the error term right. Now, one important thing if you notice in this picture for the error term we are writing it delta L minus i minus 1.

The reason is we are considering L number of layers and if you remember the error terms are computed backward right. So, the last one would be for you put i equal to L here, so, that would give you delta 1, there is the error term for the last layer that you get. So, that is why for every kernel in some higher layer kb 1 i it computes delta L minus i minus 1 as per our indexing for this picture right.

And we use this to compute the overall loss that is del J del W i right and the that is computed by so that something that we are representing by dW i here.

(Refer Slide Time: 03:12)



Recap: Backward Propagation

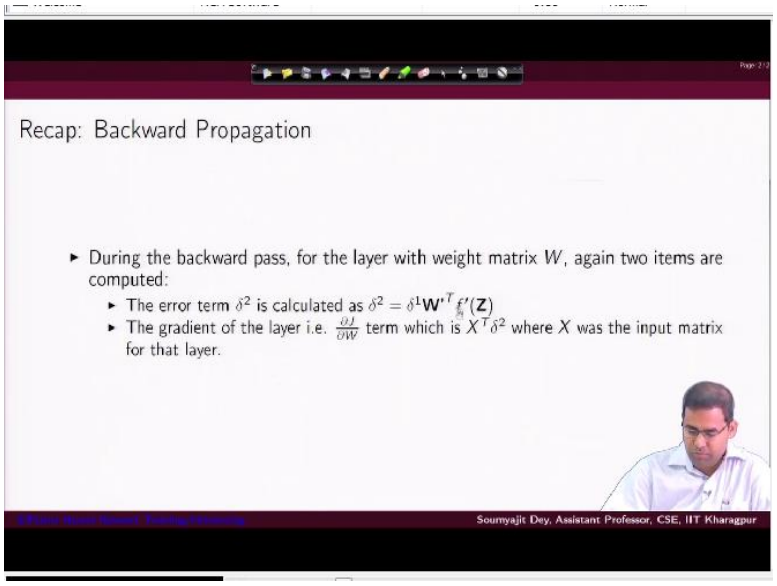
- In a similar fashion, if there was one more layer with weight matrix W^o, then again the following computations would be done:
  - The error term δ^3 is calculated as $\delta^2 \mathbf{W} f'(\mathbf{Z})$ where $f'(Z)$ represents the gradient of the activation function in the given layer.
  - The gradient of the layer i.e. $\frac{\partial J}{\partial W^o}$ term which is $X^{o\,T}\delta^3$ where X^o was the input matrix for that layer.

Let us consider a general scenario with l layers with each layer i having weight matrix W_i.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And that, of course, is basically the computation of the error term delta, delta multiplied by the previous output a previous activation input, which would be this.

(Refer Slide Time: 03:22)

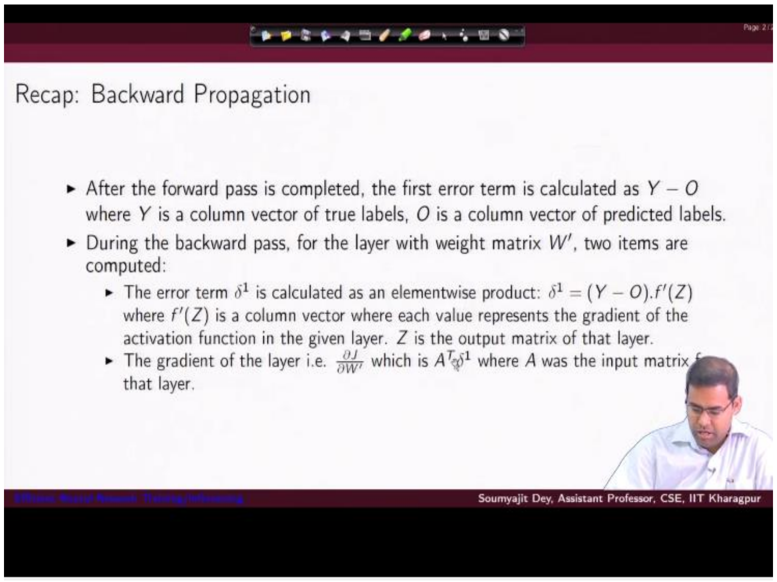


Recap: Backward Propagation

- During the backward pass, for the layer with weight matrix W, again two items are computed:
  - The error term δ^2 is calculated as $\delta^2 = \delta^1 \mathbf{W}'^T f'(\mathbf{Z})$
  - The gradient of the layer i.e. $\frac{\partial J}{\partial W}$ term which is $X^T\delta^2$ where X was the input matrix for that layer.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

(Refer Slide Time: 03:24)



Recap: Backward Propagation

- After the forward pass is completed, the first error term is calculated as $Y - O$ where Y is a column vector of true labels, O is a column vector of predicted labels.
- During the backward pass, for the layer with weight matrix W', two items are computed:
  - The error term δ^1 is calculated as an elementwise product: $\delta^1 = (Y - O).f'(Z)$ where $f'(Z)$ is a column vector where each value represents the gradient of the activation function in the given layer. Z is the output matrix of that layer.
  - The gradient of the layer i.e. $\frac{\partial J}{\partial W'}$ which is $A^T\delta^1$ where A was the input matrix f that layer.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

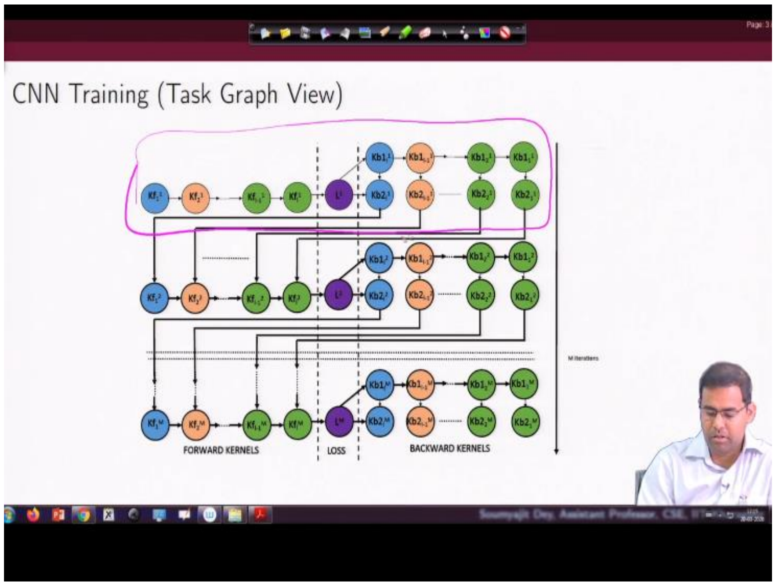
In general, this A transpose times delta 1, this kind of a computation. So, that is what is done by this kb 2 type kernel here. So, if you look into this picture in the forward pass, so, we have I am just reintroducing this picture again after our last lecture, because there is some linkage here, which we need to mention that as you can see for every layer layers, I am introducing this index, right.

I mean, although the kernel is k, which is always computing the activations A i’s right for the ith layer, I call it kf i for the next layer it is kf I mean kf i plus 1, and so on and so forth right. Basically is the same kernel, which is getting called multiple times, right. For computing output of different layers, right. And in the backward computation, we have 2 types of kernels here, one type of kernel for computing the error term right, which is always outputting, the delta terms as you can see.

So I am outputting the delta terms for every kb 1 type Kernel, I am outputting the delta terms. And then for these delta terms are used by these kb 2 type kernels to give me the differential loss, right, the differential weight. So essentially, though, is basically multiplying these deltas with the A transposes right. So that is why the dependency is from the delta. There is a dependency input dependency and they also a dependency from this A L minus 1’s right.

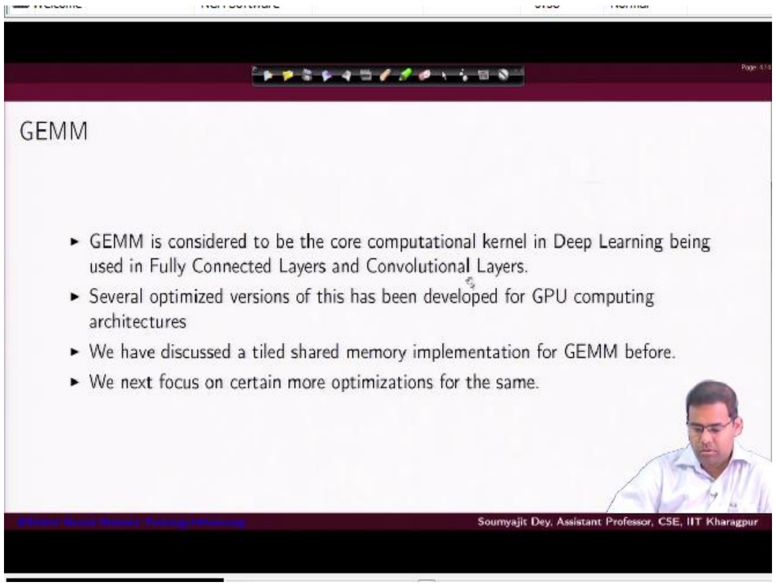
So, they are giving me this I mean the dW i terms which I will add to these W's and update them and again run the forward propagation and that will be again followed by the backpropagation right.

(Refer Slide Time: 05:25)



So, pretty simple, if we now progress with this, this was our task graph view. So essentially, as we discussed that this part is our set of kernels, which are doing the entire computation. But it keeps on repeating, right, the computation is a sequence, forward pass, then last computation, then backward pass, right, and then again, it repeats. Again, the backward pass outputs are used to update weights, and then you are going to run the forward pass and then the backward pass one, so forth, right.

(Refer Slide Time: 05:56)



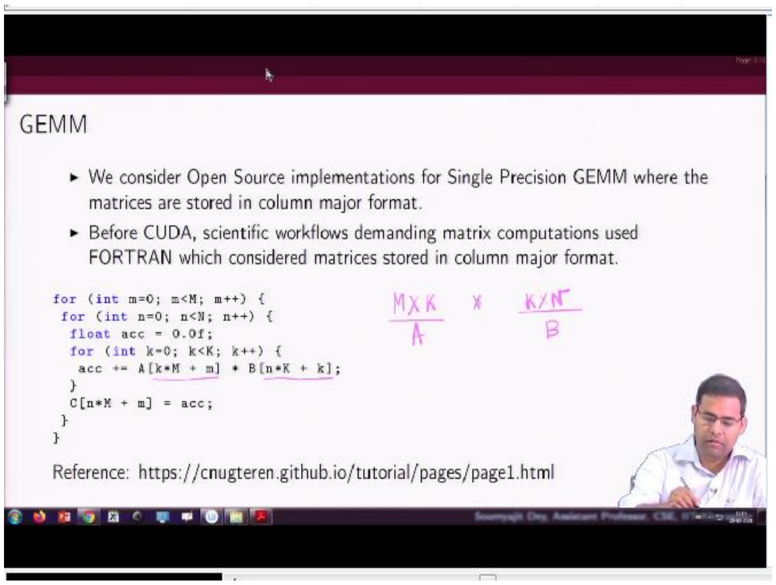
GEMM

- GEMM is considered to be the core computational kernel in Deep Learning being used in Fully Connected Layers and Convolutional Layers.
- Several optimized versions of this has been developed for GPU computing architectures
- We have discussed a tiled shared memory implementation for GEMM before.
- We next focus on certain more optimizations for the same.

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So with this, what we figured is the most important computation we will be requiring is the GEMM computations for this kind of neural network trainings.

(Refer Slide Time: 06:06)



GEMM

- We consider Open Source implementations for Single Precision GEMM where the matrices are stored in column major format.
- Before CUDA, scientific workflows demanding matrix computations used FORTRAN which considered matrices stored in column major format.

```
for (int m=0; m<M; m++) {
 for (int n=0; n<N; n++) {
  float acc = 0.0f;
  for (int k=0; k<K; k++) {
   acc += A[k*M + m] * B[n*K + k];
  }
  C[n*M + m] = acc;
 }
}
```

$\frac{M \times K}{A} \times \frac{K \times N}{B}$

Reference: https://cnugteren.github.io/tutorial/pages/page1.html

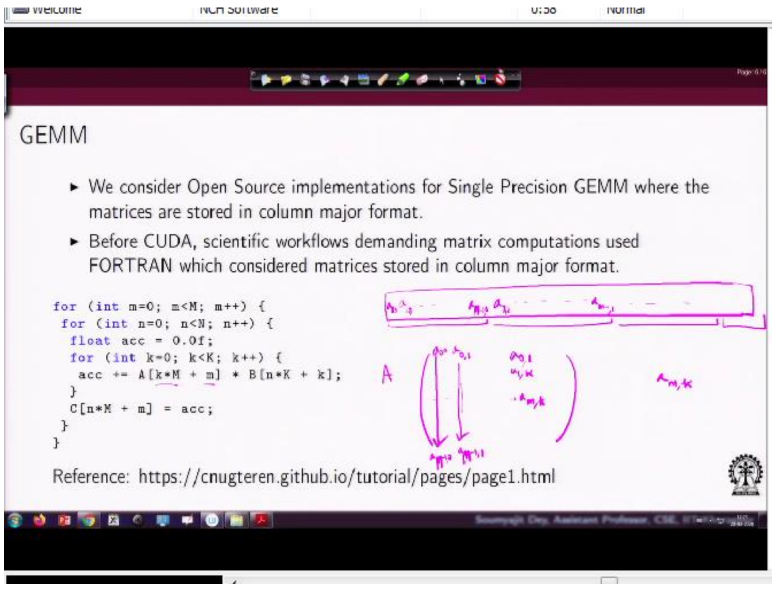
So, overall our observation had been that for doing this GEMM computation will be recurring to use our parallel programs and the major scientific libraries that are available for example plus and the modern times CU plus they actually use these Fortran based ideas that for doing the matrix computation, you assume that data is in a column major format. Now, what would that really mean. Let us understand.

So, like we discussed that whenever we are talking about column major format, one of obvious thing is that considered the matrix in every IJth term in your mathematical matrix becomes a in I mean can be accessed with a JI index. Consider that this data is stored in the column major format right and as you can see, that is what is happening. So, you are multiplying 2 matrices of dimensions M cross K with the mean.

So, this is for A and for B the dimension is K cross N you are multiplying them. So, you see the access expressions are a bit different right. So, we are more habituated to writing that will I will have a loop where I have this K iterating right for a fixed M, right. And so, essentially the way we do it is I will have M times some small m which is the dimension right. So, if I am so M terms small m plus K, multiplied by B.

And the index for B would be, of course, the column index. So K times some capital K plus N, something like that. But as you can see that it is opposite here. The reason is this column major format, like we already discussed. So one simple way to think would be that we will just consider the data entries that whatever is your actual IJth mathematical data in your computation, it can be accessed with the once you flip the indices with J and I, right.

(Refer Slide Time: 08:38)



GEMM

- We consider Open Source implementations for Single Precision GEMM where the matrices are stored in column major format.
- Before CUDA, scientific workflows demanding matrix computations used FORTRAN which considered matrices stored in column major format.

```
for (int m=0; m<M; m++) {
 for (int n=0; n<N; n++) {
  float acc = 0.0f;
  for (int k=0; k<K; k++) {
   acc += A[k*M + m] * B[n*K + k];
  }
  C[n*M + m] = acc;
 }
}
```

Reference: https://cnugteren.github.io/tutorial/pages/page1.html

But just to have a more clear picture, considered the real arrangement of the data, like as we know that we are going to now have that in memory, you have everything stored in a consecutive block. And for our usual C or similar style languages, we have row measure format, that means you have a row followed by the second row right in that way, but now, since we are considering column major format, what does that really mean.

So considered some matrix A right. So, you have the usual these are normal mathematical matrix we are trying to write right and you have the data of the row 1 row 2and all that, but when you are storing it you are not storing it in a C style arrangement in the memory, but rather you are storing it into column major format. So, if I just write the data arrangement here, so, you are going to store the first column here in the first row right.

So, that means you are essentially storing A let me use the 0 indexes here, because finally I will be writing a C program to access them a 00. Then the second element in the column right, so, A 10 like that and finally, you will be having A N minus 1 is capital M the dimension right, M cross K we are considering. So, M minus 1 for the 0th column, right, we move on to the next. So, A for the second column right.

So, that was a 00. So, now, so if I just write it you have a 00 here, I have a M minus 10 here, the next would be so, a 01 and in this column what do you all have we have a M minus 1 comma 1, right. So a 01 and it all ends with a M minus 11, right. So that is how it is. So now, when you are really trying to access the data, as you can see that you are going here for what element, you are going for some MKth element, right.

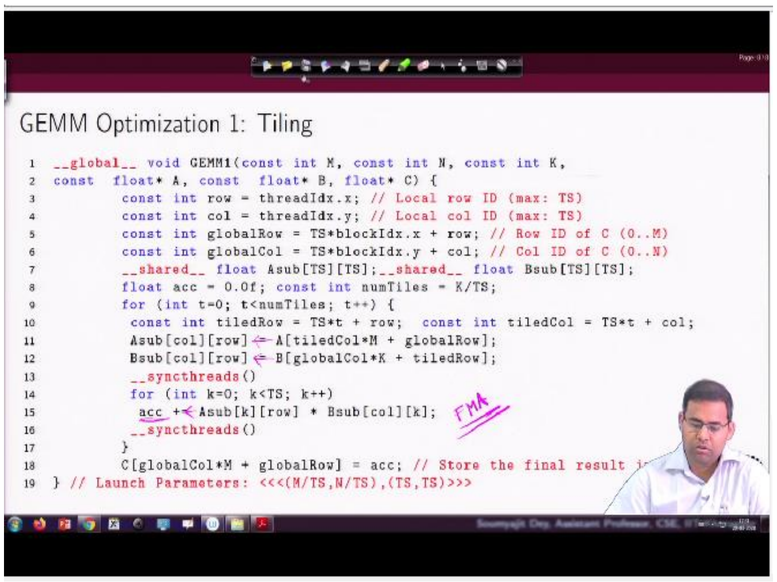
So what would be the location for the MKth element. That is the problem, right. So as you can see, for the MKth element for getting into the MKth element here, what do I really need to do. Well, I have to make capital N number of skips over this data, right. So I want some access here. MK right. This is sorry, this is capital M right. So I have to shift to the Kth column. That means I have to skip through this K times M, number of elements here, right.

And then I get to the column where I have the MKth element, right. So for that, as you can see, I have to multiply k with capital M, that takes me to that position where I have it right in that column. Now the elements of that column are located like this, right. So I have a 0 kth element, a

1 kth element like that up to a M kth element. So that is why the shift by M, right. So I am just trying to make it clear that well, it is very simple that you just flip by a J.

The indexes, but physical in terms of the memory storage, this is what it would mean right. You have to multiply k with capital M since you are shifting over K half of columns here, right.

(Refer Slide Time: 13:06)



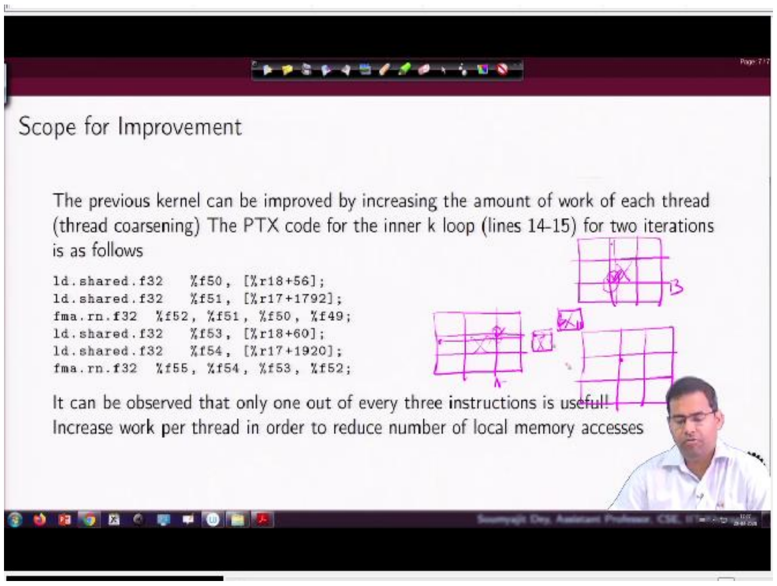
GEMM Optimization 1: Tiling

```
__global__ void GEMM1(const int M, const int N, const int K,
const  float* A, const  float* B, float* C) {
        const int row = threadIdx.x; // Local row ID (max: TS)
        const int col = threadIdx.y; // Local col ID (max: TS)
        const int globalRow = TS*blockIdx.x + row; // Row ID of C (0..M)
        const int globalCol = TS*blockIdx.y + col; // Col ID of C (0..N)
        __shared__ float Asub[TS][TS];__shared__ float Bsub[TS][TS];
        float acc = 0.0f; const int numTiles = K/TS;
        for (int t=0; t<numTiles; t++) {
         const int tiledRow = TS*t + row;  const int tiledCol = TS*t + col;
         Asub[col][row] = A[tiledCol*M + globalRow];
         Bsub[col][row] = B[globalCol*K + tiledRow];
         __syncthreads()
         for (int k=0; k<TS; k++)
          acc += Asub[k][row] * Bsub[col][k];
         __syncthreads()
        }
        C[globalCol*M + globalRow] = acc; // Store the final result i
} // Launch Parameters: <<<(M/TS,N/TS),(TS,TS)>>>
```

So with that, this would be our normal code for the, I mean J multiplication, where I am also assuming that we will do some tiling based optimization. Now, of course, we are familiar with tiling based optimizations for matrix multiplication. So, this is basically that code only. So what do we really do. We use matrix multiplication kernel. We compute the rows and columns which are the, for the thread Idx and thread Idy right.

And we use them to figure out the, what is the global row and column value. So, in general, if you recall, our idea of doing the normal matrix multiplication, so every thread we are going to launch a block of threads, every thread block will try to compute the elements for a tile, right, inside a tile, or in the corresponding view for a block of threads, every thread is trying to compute one element in the tile right, And what was the way that we progress.

(Refer Slide Time: 14:16)



Scope for Improvement

The previous kernel can be improved by increasing the amount of work of each thread (thread coarsening) The PTX code for the inner k loop (lines 14-15) for two iterations is as follows

```
ld.shared.f32   %f50, [%r18+56];
ld.shared.f32   %f51, [%r17+1792];
fma.rn.f32  %f52, %f51, %f50, %f49;
ld.shared.f32   %f53, [%r18+60];
ld.shared.f32   %f54, [%r17+1920];
fma.rn.f32  %f55, %f54, %f53, %f52;
```

It can be observed that only one out of every three instructions is useful!
Increase work per thread in order to reduce number of local memory accesses

So if you just remember, let me just redraw the picture again. So let us say this is your B, this is your A and you are trying to produce an output matrix C here, right. If you break to tiles, the tiles indirectly define the way you are going to launch the thread blocks. So for one block of threads you are corresponding to computation of one tile. So every thread the thread which is going to compute this element will I mean, it will have to do a computation of elements here.

I mean, it has to essentially multiply the elements here with the elements here, right. And how does it really do it. Well, it does it, I mean by coordinating with other threads that means for this, the thread is first responsible for loading this data, it will be using 2 shared memory locations right. It will be using 2 shared memory locations. So, it will use this shared memory location to load these data points.

It will use this inside this shared memory, it will be loading these data points, all the other threads in the block will be loading the other thread data points. Once that is done, this thread is supposed to now if you look back into the code, so this is the position where all those loadings will be done right by all the threads collaboratively. Now once yeah. So, here as you can see, so, we are in the first iteration of this loop.

All the threads have loaded the first data of the first tile right. And then they go to this second loop here in this loop what is going on. Now, this thread is going to compute a multiplication of the data, which has been loaded by all threads here with the data that has been loaded by the every thread here, right. So this multiplied by this, so there gives you the partial sum. In the next iteration of the outer loop.

Again, all the threads in the block, we load the data for the intermediate this block here, and then this block here. And then again, every thread will perform the corresponding activities so that now after bit ends events, the thread, which is here, has performed a partial sum of data up to this much and after this much right here, and it continues like this. So that was our idea of tiling. So just a small recall here, right.

So if you remember, so this is the position where one thread is loading 2 data points into the shared memory is Asub and Bsub, right. And it is waiting here for every other thread in the same tile to do the load. And then is falling into this loop, where it is progressing with all the data in that row for Asub and in that column of Bsub to do the computation of the partial sum, right. And once everything is done, like once the thing is done for all the tiles, which will update into the global memory, right.

So that is how things were going on for normal tiling. No. Again, if you remember that since is the column major formatting. So as you can see the access expression here for Asub and Bsub is just reverse to what we have been seeing earlier right. So that is why, by tradition, k comes first, right. Because it is in the columns now, right. So it is Asub k row instead of Asub row k. And similarly, Bsub column k instead of Bsub k column, right, is just the opposite from earlier.

Now, earlier, if you remember, we were happy with the result of this kind of tiling based operation, right, this kind of matrix multiplication kernel with this kind of tiling. We were happy with that, right. But now we will try to see whether this can be accelerated further reason. Well, of course, we have understood that these the most important operation for neural network training and it is also I mean, for going to be popular with many other kinds of workloads.

So this needs to be optimized, right. So right now all of our efforts would be in terms of optimizing this GEMM kernel. Now just if you I will just point out once again that the difference when we are speaking specifically with respect to the GEMM operations for the Cu plus or similar kinds of libraries for linear algebra.

We will be using this row major representation, right, we will because that, you need to remember that, because that is going to also play a role in our understanding of how things are going to happen here. So, for Asub k comes first followed by a row, for Bsub you have column followed by k here, right instead of the reverse that will be expecting in general to happen. Because let us understand that why is this important.

Well, that would mean that the way that the data is getting access is a bit different, right. So that means here when I am accessing the data, what is the access from, the access of this data is from shared memory, right, the access of these data these are global load to share memory this is a load from the shared memory to the internal registers for doing the fused multiplication and add right.

This is a there is a call to fuse multiply and add unit which is there in the GPU right. Now, this global notes are going to take their own time right. Now, again as we know that one of these loads will be coalesced and the other will be not right because one of them will be row wise and the other would be column wise strength figure out which one is what right. But here as you can see, this is for from the shared memory to the global to the fuse multiply and update unit.

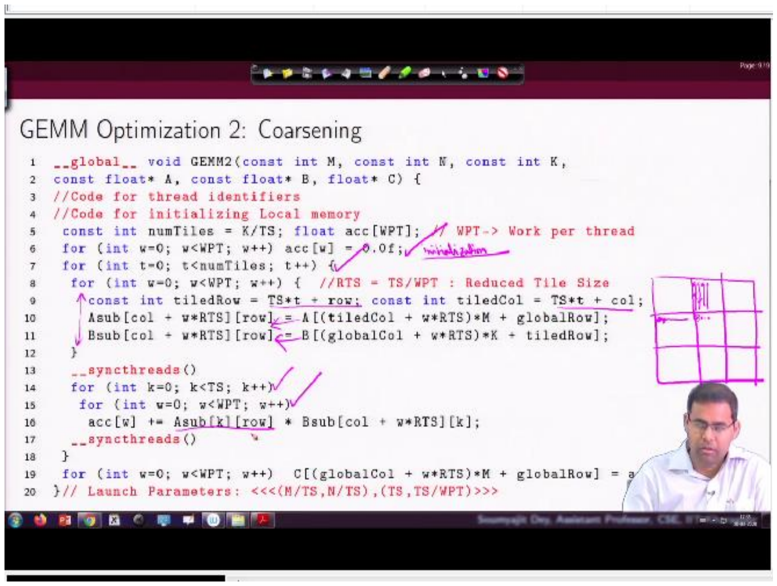
Please multiply and accumulate unit with the output going to a register of type acc and the variable name for the register is here right. Now, if you try to analyze that how this program performs. Let us just have a look into the intermediate code of the PTS code that will be generated for this kernel. So this peak into the PTX code for the inner loop lines 14 to 15. That is it right. So this is the inner loop.

As you can see, we are doing 2 shared loads from the shared memory because of course Asub and Bsub are in the shared memory. And we are then engaging the fuse multiply and add units right. So, in each iteration of the loop, part shared activity is as follows. You are doing 2 shared loads followed by one fuse multiply add operation, again. In the next iteration of the loop, you are doing 2 shared loads and the real operation we are doing is one fuse multiply add.

So as you can see that It is basically like only one out of every 3 instructions is useful for computation. But the other 2 are for memory accesses to the shared memory right. So if I assume that I have a significant number of fuse multiply add units, well, they are not really getting used right. So, fundamental reason is that I have lesser occupancy, because I am not making good use of the number of threads I have launched.

Rather than that, I can make full utilization of the GPU bandwidth by increasing the amount of work that I am delegating part shared, because here as you can see, the third activity I am really getting is one addition of fuse multiply and addition operation for every 2 shared loads, right. We like to increase it, how would that be possible.

(Refer Slide Time: 22:50)



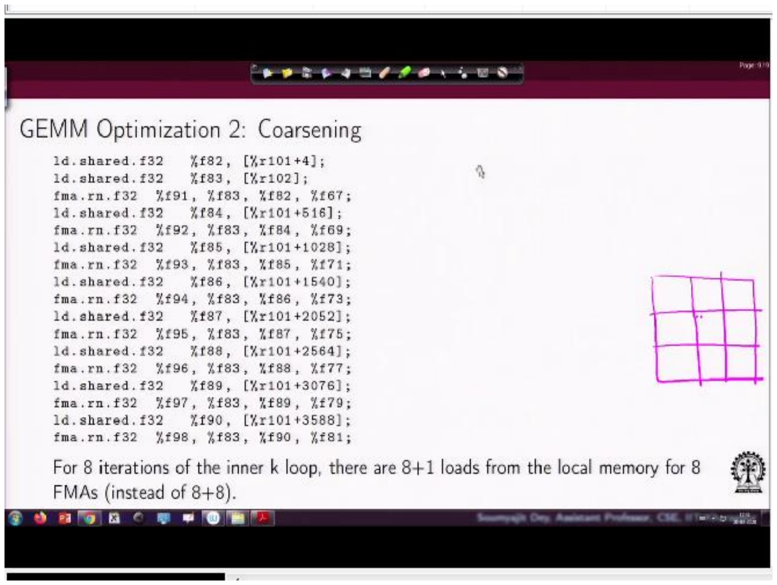
GEMM Optimization 2: Coarsening

```
__global__ void GEMM2(const int M, const int N, const int K,
const float* A, const float* B, float* C) {
//Code for thread identifiers
//Code for initializing Local memory
 const int numTiles = K/TS; float acc[WPT]; // WPT-> Work per thread
 for (int w=0; w<WPT; w++) acc[w] = 0.0f;
 for (int t=0; t<numTiles; t++) {
  for (int w=0; w<WPT; w++) {  //RTS = TS/WPT : Reduced Tile Size
   const int tiledRow = TS*t + row; const int tiledCol = TS*t + col;
   Asub[col + w*RTS][row] = A[(tiledCol + w*RTS)*M + globalRow];
   Bsub[col + w*RTS][row] = B[(globalCol + w*RTS)*K + tiledRow];
  }
  __syncthreads()
  for (int k=0; k<TS; k++)
   for (int w=0; w<WPT; w++)
    acc[w] += Asub[k][row] * Bsub[col + w*RTS][k];
  __syncthreads()
 }
 for (int w=0; w<WPT; w++)  C[(globalCol + w*RTS)*M + globalRow] = a
}// Launch Parameters: <<<(M/TS,N/TS),(TS,TS/WPT)>>>
```

So for this, we will perform we will start to apply our threat coarsening techniques that we have seen earlier, right. So fundamentally what we are really trying to do, now we will try to ask that well, even inside a tile, I did not have a same number of threads in the blog, as is required for computing the number of elements in a tile rather than that in my thread block launch, let me have a smaller number of threads for computing the entry of the entire time.

That would mean now for one thread, I am not just delegating it, the activity for reserved for computing one final output in the matrix. But rather I am asking you that you compute more than one entry in the final matrix, right.

(Refer Slide Time: 23:40)



GEMM Optimization 2: Coarsening

```
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).

So if I just draw a picture, it would be like this that let us say this is my final output. And I am going to launch tiles and all that. So earlier, whatever was the activity for one thread, let us say I am asking one thread to do the computation for 2 elements, or in general, maybe more than 2 elements, right. So that is the amount of depends on what is my amount, of coarsening that I am pushing into the thread, let us say 8 elements, right.

So we will see how it makes progress. So now as you can see, I will introduce a variable WPT work per thread. So of course, one thread will be now accumulating values for more than one output. So instead of keeping acc as a single value, single local variable, which maps to a single local register for part thread, I will make it an array acc WPT, so that the thread is accumulating values for multiple outputs, right.

And then I hope you can understand things are going to be simple. It is like so earlier you had this loop, let us just compare with the previous and the current implementation. So earlier, you had the outer loop where you were doing the loads followed by the account, well you still have the outer loop. This is your. So this is a simple loop just for initializing the acc, right. This, you are just initializing basis here, initialization.

But then this is your normal loop of the previous work. But here earlier you were just doing 2 shared loads, right, this one and this one. Now you are doing those 2 shared loads inside this loop. This loop is running from 0 to WPT minus 1. That means you are performing that many shared loads for both A as well as B, right. And you have to compute the shared load indexes by looking into the values of the offsets and all that which I think you can easily figure out right.

Just to understand that for part shared activity, we have increased the number of loads here, well how does that help because till now, we have not seen how the compute increases per thread because we have increased the number of loads per thread. Well, now, let us look into the compute part. So, after the loading is done, here, you have the compute earlier, your compute was just this loop followed by this multiplication, right.

But now, inside this loop, you have another loop where you are doing this multiplication. So earlier, what was the situation let us say for this entry, you are computing this multiplied by this, right. But now you have 2 entries. So you are going to compute this multiplied by for one thread, the amount of activities, let us say you have 4 entries, you are going to compute by this thread. So you have to load data from here.

And I mean, you have to multiply the data from here along with the data from all these 4 columns, right. So there is the part shared activity you have inside the loop now, right, that is why you have this internal loop. But what is the beauty of this internal loop. The beauty of this internal loop is that of course, here when I am drawing I am using the normal matrix representation, you have to just switch it accordingly.

Because as you can see that the indices are again in column A and all those formats, but if we are trying to understand in as you can see that when this loop is making the computations for these 4 points, it can take one value here, it has to multiply that value with the value here, here, here and here right. So one load of this multiple loads of data from Bsub, and that many number of multiplications, go to the next iteration of the loop, one load from Asub multiple loads from Bsub that many multiplications right.

So there is the good thing right. Because since you are delegating more activity for this thread for that one load is common for multiple loads of data from the other array right. So that is the good thing since this is constant, you have less number of loads at least for one of the inputs and that increases your part thread computation. Now, I hope this is clear. So, the first good thing is you have coarsened the amount of activity per thread.

And the coarsening is more for the compute part rather than the load part. So, if I just compare it with the previous implementation, the amount of loads from shared memory to the registers for doing the fuse multiply and add are drastically getting reduced by a factor which is the threat coarsening factor. If you compare the number of loads here, so this is the PTX, considering that you have coarsened with a value of 8.

So for ith 8 iterations of the inner loop what is going on you load this Asub once you have to load Bsub 8 number of times to achieve 8 number of fuse multiply adds right earlier the number of loads was 8 plus 8, but now it is 8 plus 1 right. So, this is not the inner 8 iterations of inner yeah is basically the innermost loop actually fine. So, we this optimization will be ending this lecture and in the next lecture we will see some more optimizations. Thank you.