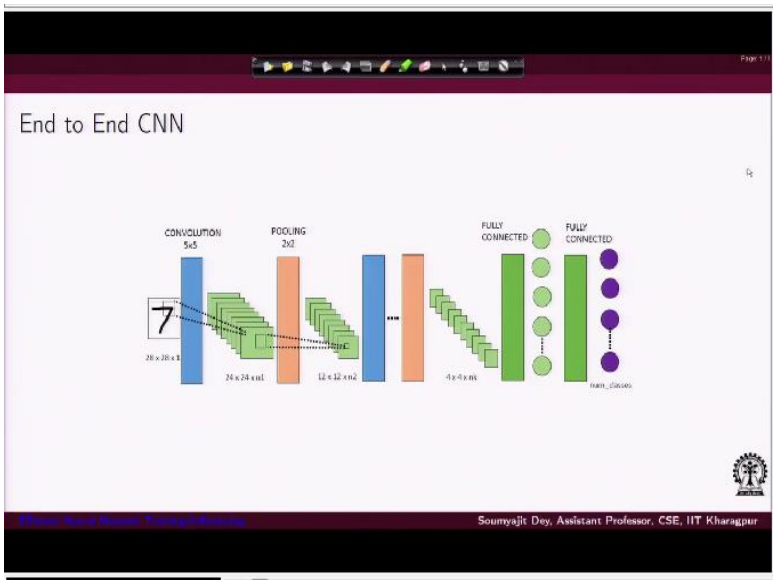
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-59
Efficient Neural Network Training/Inferencing

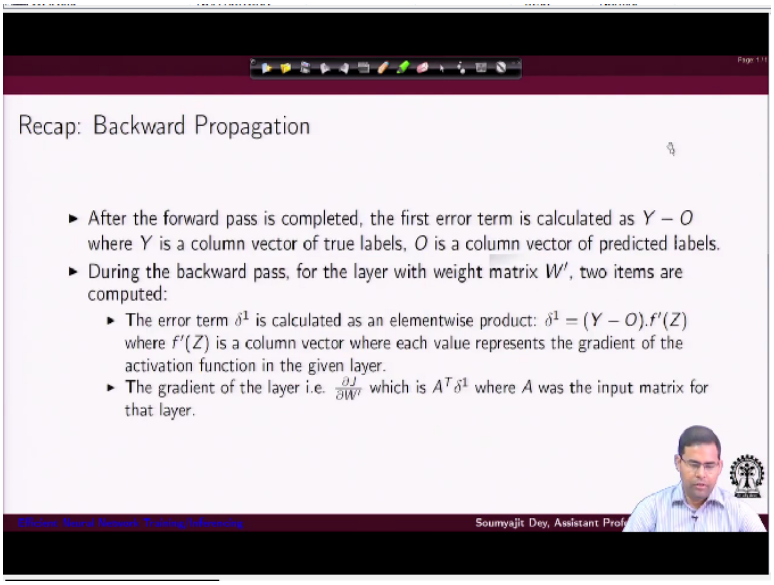
Hi, welcome back to the lecture series on GPU architectures and programming.

(Refer Slide Time: 00:27)



So, if we remember in the last lecture, we have been discussing how CNN pipelines work and what are the underlying computations that need to be done for such systems.

(Refer Slide Time: 00:41)



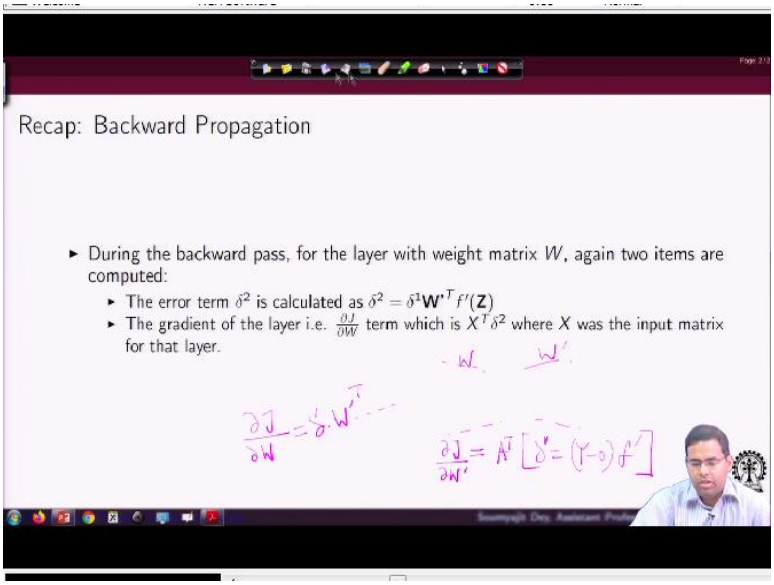
And based on that we will just try to now figure out what is the nature task graph model for CNNs, just like we have discussed the task graph models for I mean, are more specifically the computational graphs that can be created for generic neural networks. So before going to that, we will just do a quick recap of our discussions on back propagation. So, what really happens is that if you remember, I mean, we are just doing this up to.

So that we are again back to the notations that we have discussed earlier. So, while completing the forward pass over a neural network, we were denoting the error term as this Y - O, right, where Y is the column vector of true levels and O is the column vector of the predicted levels like we discussed earlier. And then during the backward pass, which started with a layer on with the weight matrix W prime, we are computing the following quantities.

For example, we first computed delta 1, right, which was product term that we figured out essentially delta 1 was like this Y – O multiple by the derivative of the activation function right, which again is a column vector and each of the values represent the derivative of the function in that given layer right. And once the delta 1 was computed, that actually helped us to figure out what is delta J delta W prime and we have already derived that for this the expression is this A transpose delta 1 where A is the input matrix for that layer.

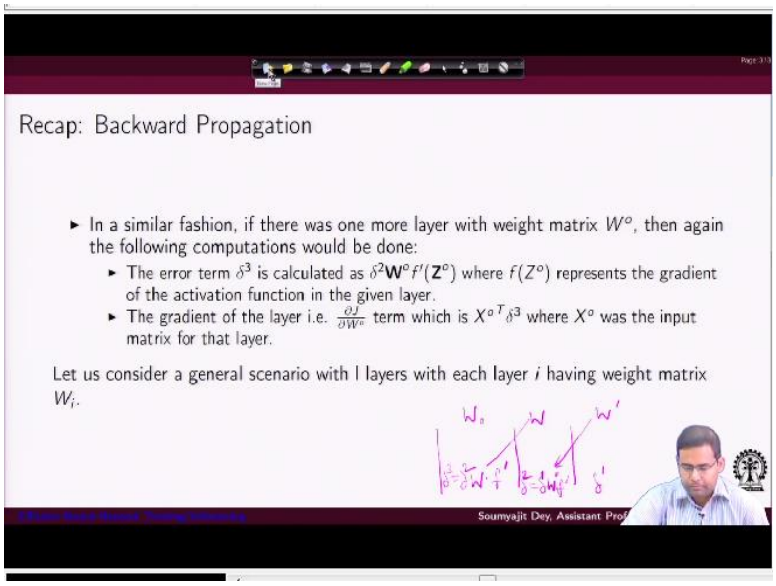
I mean it has been found by taking the input and doing a product with W and all that like we discussed earlier.

(Refer Slide Time: 02:34)



So, if you remember, **we** now we have the availability of delta 1 with us. Now, using delta 1 we can also calculate delta 2 like we discussed earlier. So, delta 2 is nothing but delta 1 times W prime transpose multiplied by again the derivative of the activations right and using delta 2, we can compute the derivative of J with respect to W. And this is given by X transpose multiplied by delta 2 where x is the input matrix for that layer right. So, these are the different terms that need to be derived while doing the backpropagation.

(Refer Slide Time: 03:19)



Again, we will just recall like, so, the one of the derivatives of the loss function is A transpose delta 1, where delta 1 is the error multiplied by f prime of the I mean f prime Z which is the gradient of the activation and the other that is delta J delta W, this is computable as X transpose delta 2, where X is the input matrix and delta 2 is as is given using delta 1 and the other expressions here right.

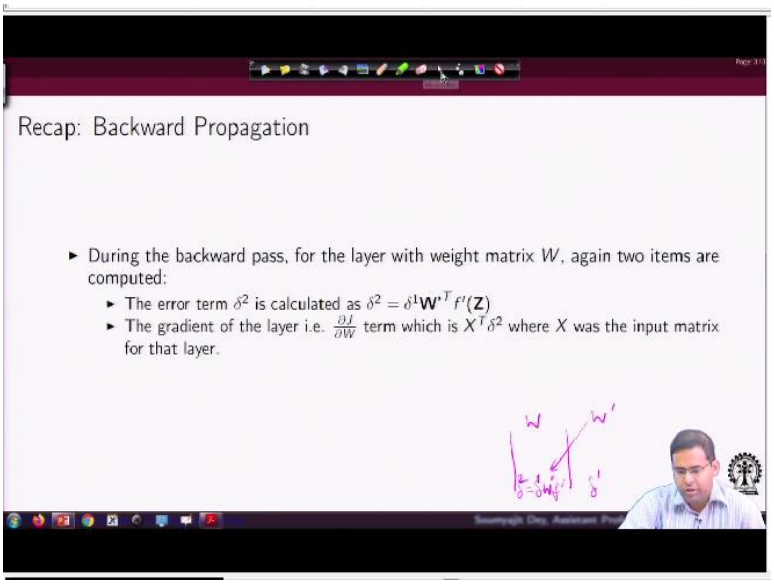
So if we are just using this values in a similar fashion. If there are more layers, then what would really happen. So, let us recall the previous things like so in our first neural network example, we just considered that there is a layer with the weights W and then there was this layer with weights W prime and we derive delta 1 here, right. So, if we are just doing a pictorial description.

Then delta 1 was this value and f prime and then we will just multiply this with the A transpose, right. And that essentially gave me this derivative that del J del W prime right. So, that is essentially the computation we require for the last layer for W prime and then we had this previous layer corresponding to W and for that we next computed this delta 2 right and for delta 2 we actually use delta 1 and then we got del J del W we utilize data 1.

And then we multiply it by the previous layers, weights transpose and of course, we have the f prime and others right. So, essentially as we can see that there is a dependency of the previous layers loss derivative on the last layer right. Now, if we just take this solution into a cascade then what is going to happen. So, suppose there are more such layers let us for example, add one layer del W 0 right then we will have in if we just continue like this first we have calculated delta 1.

We use delta 1 to calculate delta 2, right. We just continue like this. So one small problem we have, I believe this should be del W is not going to be W prime. **(Video Starts: 06:18)** So just a minute yeah. So when are doing this, yeah this is W prime. So when we are doing the del J del W is using W prime which is a last layer and if we are going to continue the same. **(Video Ends: 06:40)**

(Refer Slide Time: 06:41)

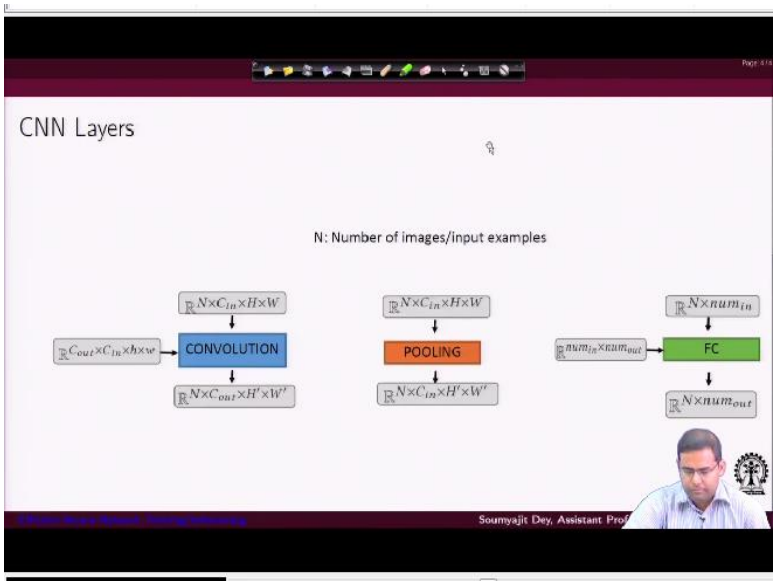


Then just like I am computing delta 2, and I am using delta 2 as delta 1 multiplied by W prime transpose. So essentially, W prime is the weight matrix for the layer in which I have computed delta 1 right. So if we just try and segregate, here I have W. Here I have W prime the next layer. In this layer, I have computed delta 1. In this layer, I am computing delta 2, and I am using delta 1. And then W prime, which is basically the weight of the next layer, right. And then I have this f prime, right. So what we are really trying to say is that this is going to continue, right.

So now, if we bring in a new layer here, let us call it W 0. So then we will have delta 3 here. And delta 3, we will just see that the same thing is going to continue. So we will now have delta 2 and we will have W here and then we will have f prime again for Z 0, that this represents the gradient of the activation function in the given layer. So when I am computing data 3 is data 2 W0, then f prime Z 0, where this represents a gradient of the activation function of this layer.

Whereas for the previous it was just Z, where it was the gradient of the activation function of that layer like that right. Now, so the gradient of the layer delta J delta W0, that I can again figure out using X 0 transpose delta 3, considering that X 0 is the input matrix for this layer, right.

(Refer Slide Time: 08:49)

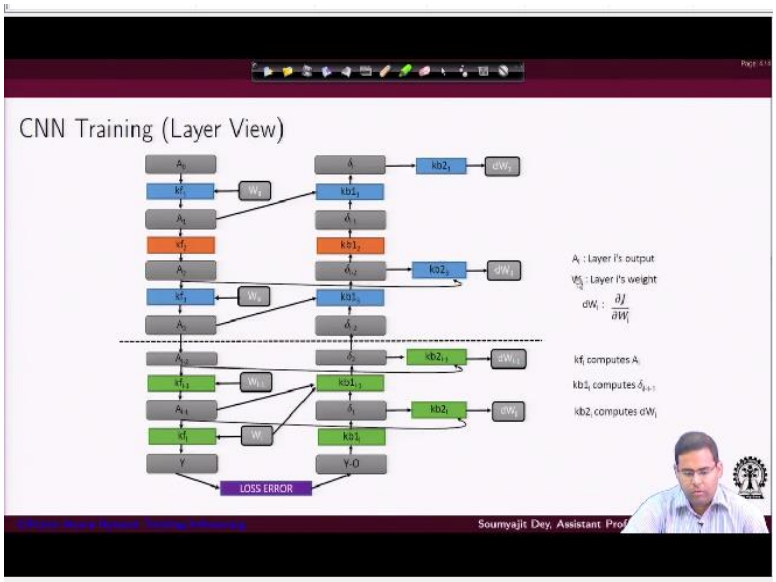


So the point is very simple for every layer, I can keep on computing these delta values. And then I am going to do a matrix multiply with the weight matrix, and then also a further matrix multiply with the activation functions output, right. So that is how things are going to progress right. Now, if I consider a very general scenario, where I have these l layers, and each layer i has a weight matrix W i.

So, in general, as we have already discussed in CNNs, I have this different kinds of layers. For example, I have the convolution layer. So in convolution layer, I have these 2 inputs, right. So I have the image input and I have the mask to be confirmed with the image and I gave the converted output, right. In pooling I have a transformed image input, and I am going to down sample it, right.

So the H and Ws are going to change to H primes and W prime's right whereas the other things remain same right. And then I have the fully connected layer. So, in the fully connected layer, I am having the weights and I am having the input transformed image. And finally, it gets reduced to a linear array full of different class members, right.

(Refer Slide Time: 10:24)



So, if we are now trying to look into how the CNN computation goes on, so, this would be more like a task graph representation for that. Okay, so, let us see how things are going on here. So let us say this is my starting layer is 0. So that goes as an input to the first kernel kf 1, let us call it. So here the blue ones are essentially convolution layers and they are providing the and so essentially, you can think that I have a convolution with that activation.

And now I am getting the input for the next layer, right. So I get from A 0, I get it transformed to A 1 from A 1 I am again. So let us say the green ones, the orange ones are trying to represent pooling layers again. So then again, I pull it down, and the down sample, I get A 2. Again, there is a convolution layer. Now, of course, for every convolution layer, I not only need the image, but also the mask.

So I have a W 1 here as an input. Let us say I have a W 3 here as an input, and so on, so forth, right. So I have a convolution followed by pooling again, convolution followed by pooling like that. So here, I can think that these dots represent that there are more such different layers here, right. At the end, I am going to have the fully connected layers, right. So in the fully connected layers, let us say the green ones, represent the fully connected layers operations, right.

So at the end of the fully connected layer, I will get the overall output of the network, right from here, I can compute the loss. And then I can use this loss error, that is I can get Y, I can compute

the error that is Y minus O, and then I am going to use Y minus O to compute the each of the previous layers delta values right, because we have already discussed here, like how I can in general compute the delta values of the some layer based on the delta value of the previous layer.

And also use that layers weight matrix, and the activations gradient and all that, right. So with this here we are getting the error, and I am using let us say this matrix Kb 1 and this kernel is actually representing the computation of delta 1 making use of , so if you remember what was my delta 1. So delta 1 just requires Y minus O and f prime, right. So there is no requirement for any input of the weight matrices, right.

So I get delta 1 here. But then when I use delta 1, to get delta 2, if you remember the expression of delta 2, which is I mean something with respect to the previous one, in general, is that you need a weight matrix is transpose right unit S transpose. That is why this is A will be transposed here, right. So there are dependencies from A to here. So this kernel is going to make use of A, is going to make use of the W of that layer in order to compute delta 2.

Now let us again review why this is necessary. So as you can see, this delta 2 is nothing but delta 1, the weights and the value of f prime right and then we are multiplying it with the input matrix in the backward propagation. But what happens when you are just computing the first delta, the first delta is Y minus O into f prime, and then you are multiplying this with A transpose, which is the input matrix to get original loss function right.

So coming here, when I have to go from this delta 1 to delta 2, what is really happening. So I am using this delta 1, and I am using this W l, this weight matrix. And I am going to use this A to get delta 2 then I am again using delta 2 to get the previous one delta 3, and like that, this will make progress, right. And then I come back to the different pooling layers, right, where again, I will be computing this in the backward pass and I am keep on computing kb 1 kb 3 kb 1 2 like that to get the all the delta values here right.

So, we are trying to show our task graph view of these different kernels that are going to perform these data transformations of what the convolution layer for the pooling layer and for the fully

connected layer like that right. Now also we need to see that computing the delta 2s are the delta 3s or tiles in general that is not all right, because what is my overall requirement. My overall requirement is to compute in general the loss with respect to the different weights right.

So, let us represent this del J del W i by this text dW i right. So, in general, I can say that this kf i they compute this are the kernels which compute the output given matrices A i's, right. Then let us say kb 1 i, this represents a kernels, which are actually responsible for computing the deltas in the backward pass right. And similarly, let us say kb 2 i, this is a different class of kernel, which uses kb 1's outputs that is the delta l minus i's.

And also these is to compute the overall loss because if you remember, the final loss is basically this image matrix transpose multiplied by the deltas, right. So that is a final loss. So that is why I need this additional kernels with dependencies which are coming from the A's right. So I will just review what we exactly did. So if you see, the computation of every delta depends on the previously computed delta times.

So every delta's computation depends on the previously computed deltas value, let us say I am taking delta 2 and delta 1 and the previous layers, the previous layers weight matrix, right. As you can see here. So delta 2 is going to use this W here . I think this should be, yeah, delta 2 is going to use the weight matrix and the activation functions gradient right. So both of those. So that is why I have this dependency here, right.

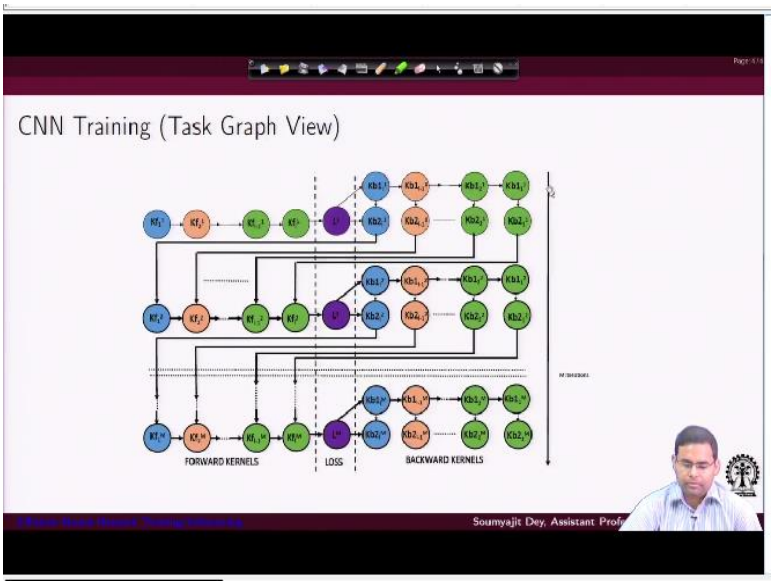
And we are just trying to show that this activation function is available and the gradients outputs are also available here. Now, when are the gradients outputs available. Well they get available immediately after you compute the A's right because finally, from the A's you are passing them through the activation functions only to get the actual matrices, right. So here, you are using these W's and A's.

And the previous deltas for this kb 1 kernels kb 1 type kernels to get the next layer deltas, right. So they form one class, then you use these kb 2 class of kernels for, and they get as input the deltas as well as the A matrices for different layers to give you the losses with respect to the

weights in that layer, right. And that is how it will progress right. So, overall, if I see task graph view of the CNNs forward pass, followed by the backward pass.

I start with these values of W 1 W 2 W 3 like this for all the layers and at the end of the backward pass I am successful in computing the derivatives of the all these layers weights. And then of course, we will do the obvious thing, which is we will just update this W 1's with W 1 plus dW 1. Similarly, like that for the other W's right and then we will do again the forward backward traversal like that right. This is just like a dependency oriented view of the CNN computation.

(Refer Slide Time: 19:09)



So, if we now try to have a look like how these different kernels or these different data parallel kernels are going to work, what are their dependencies. So, as you can see, you have the convolution layer kernel which is doing the operation of computing the image transform followed by the activation function operation right. Then you have the pooling layer kernels here, right, so convolution then pooling like that.

So, that sequence will continue and then you will have the kernels for the fully connected layer, these entirely a forward pass, then you have functions which compute the loss values. So these are kernels for doing the loss value computation. And then you have these 2 set of kernels, one

set of kernels which are sitting in the casket and they are just going computing the delta values for each of these different layers.

And we have the other set of kernels, which use those Delta values. So these are the kb 2 + kernels, I mean is, of course, our own labeling scheme. Here, we are using them to compute the differentiated weights, right, the gradient gradients of the weights, right. So that is what we get as output from this kernels. And these are the outputs that we are going to use. Now, of course, we will be using them to update the W values for the next forward pass.

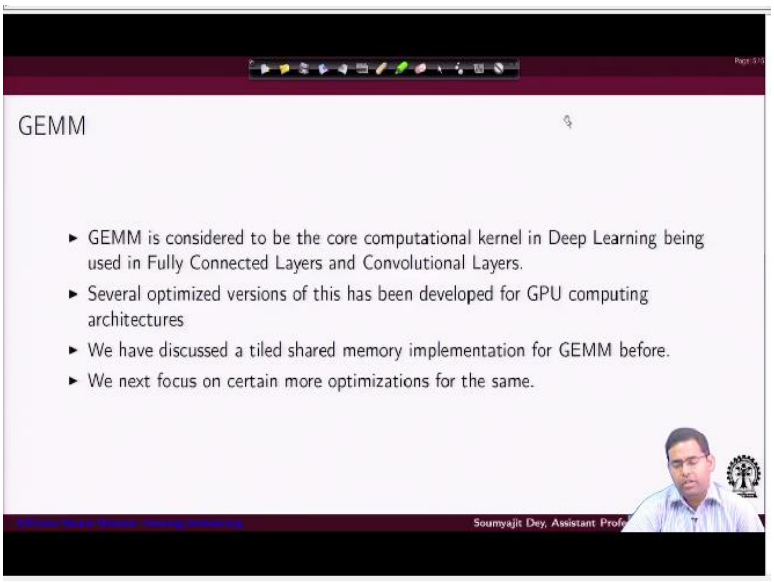
So that is why their outputs go again, to the second instance, of the same kernels in the forward pass. So as you can see that forward pass kernels kf 1 kf 2, all of them had a superscript 1. Now we are just changing it to a superscript 2, right, just to highlight that this is a second iteration over the forward pass. So I have the kf 1 2 kf 2 2 like that convolution followed by pooling against the sequences and followed by the kernels in the fully connected layer.

Then again I go to the loss function kernel for a second time. So is the same kernel right is just operating on a second iteration right. And this loop again leads us to those sequence of kernel pipelines for computing the delta values and also the dW values, which are this, I mean, which I get just by using the deltas and the weight matrices. And then again, this continues, right. So, all the outputs of this set of kernels in the second iteration will be again passed to the forward pass kernels for their third iteration.

And that is the view, that is just like an unrolled viewed overlay dependency graph right. So essentially, I can say that in my system, I have got this set of parallel kernels which are working right, here I have the forward pass, the loss computation, the backward pass, and it keeps on happening for multiple iterations inside the loop with the updates going on right. So this is like a task graph view for the convolution.

And network training part, right through which we can, once it converges, we get the updated weight values.

(Refer Slide Time: 22:10)



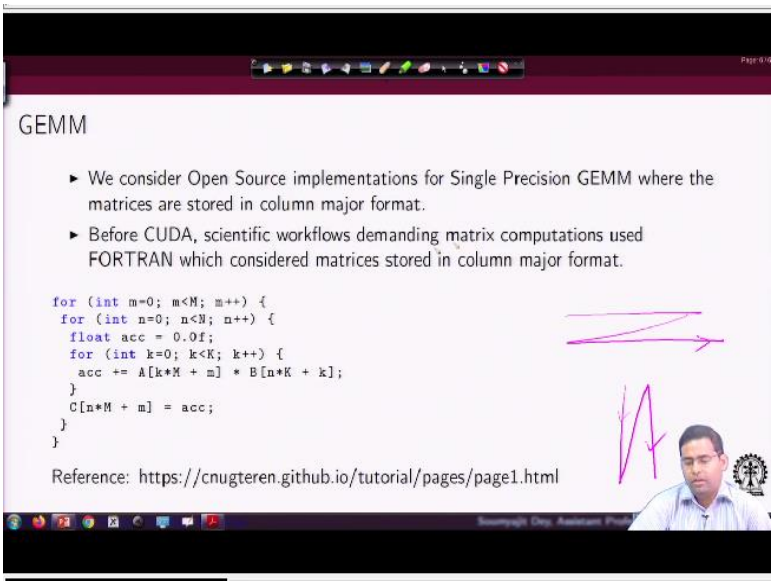
But then the important part that we want to discuss, like, well, we have the task graph view, but we have also understood that all the operations that are really going on, which is basically the delta computation, right. It is all general, matrix multiplications, right. So as you can see the loss matrix multiplied by the gradients of the activation, and then again, using each of the layers deltas with multiplying them with the layer weights right to get the previous layers deltas.

And again, you I mean, continuing like that, and also in each layer, using the value of the delta and the value of the corresponding image in case input current input layer, then is the input image. If it is an intermediate layer, then the transformed image available for that layer, so on and so forth. So finally all of them are general matrix multiplications right. So fundamentally, that is where we have to optimize, right.

So that is why this generalized matrix multiplication is considered to be the core kernel in deep learning, and is being used in both the fully connected as well as the convolution layers, right, because we have already seen earlier how that convolution operation is simply converted to the GEMM by doing a simple image to column transformation. And for doing this, there exist several optimized an implementations for GEMM.

And previously, we have discussed some tile shared memory implementation. And we will just focus a bit more on more optimized versions of the same, right.

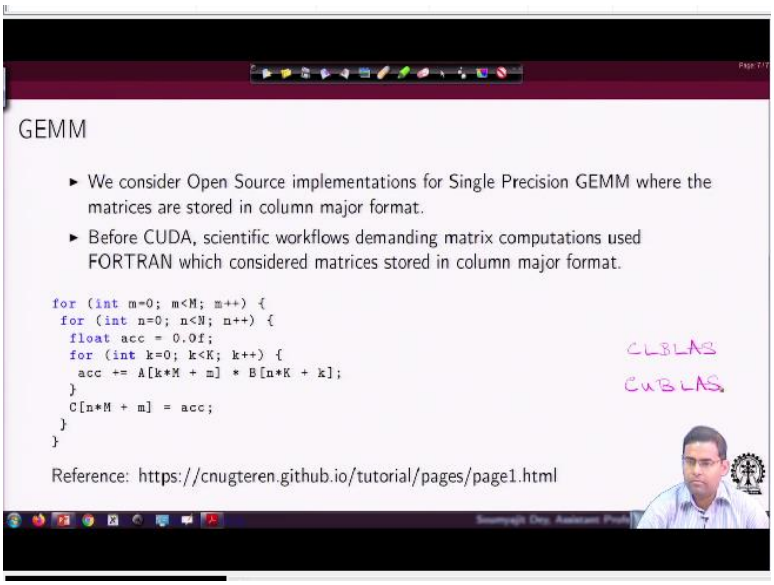
(Refer Slide Time: 23:53)



But there is something interesting we would like to talk about here, which is like of course, this idea of multiplying large matrices has been there for quite a while, right people will have been working with such large matrices and multiplications and if you see, I mean, there are popular routines for scientific computation, like BLAS, which implement underlying FORTRAN libraries, which have been doing this matrix manipulation operations very efficiently.

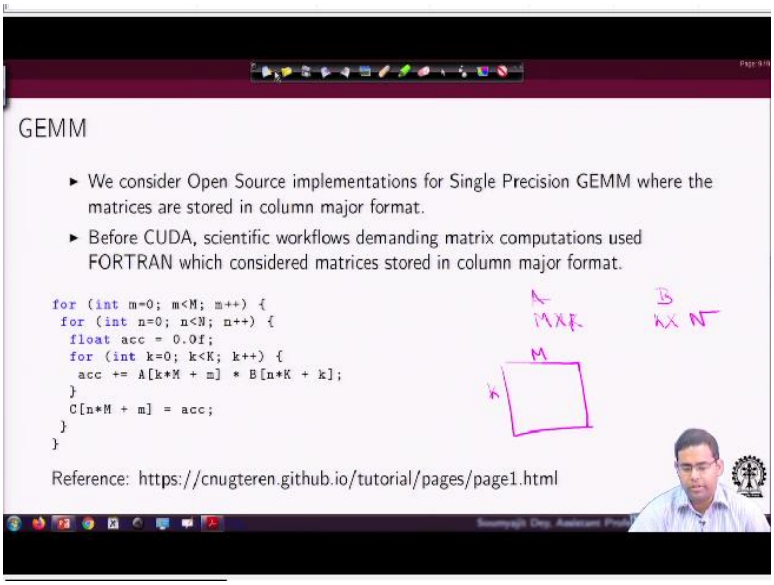
But one issue is these libraries assume a column major format. So that means the data is stored in the matrix not in the I mean, when we are storing the data in the matrix, we are not assuming that the storage is row wise. But rather it is like this right. So it is a column major format. And the issue is the so in the memory when I am storing and traversing the matrix in a columnar major fashion, right. Now when I am going to do column major traversal, of course, the access expressions for the matrix are going to change.

(Refer Slide Time: 25:19)



And the important thing for us is this idea of column major format of metric storage has got carried forward into modern scientific workloads for were in popular libraries, like this CLBLAS, which is the open CL version of BLAS, and also CuBLASS, which is the CUDA version of BLAS right.

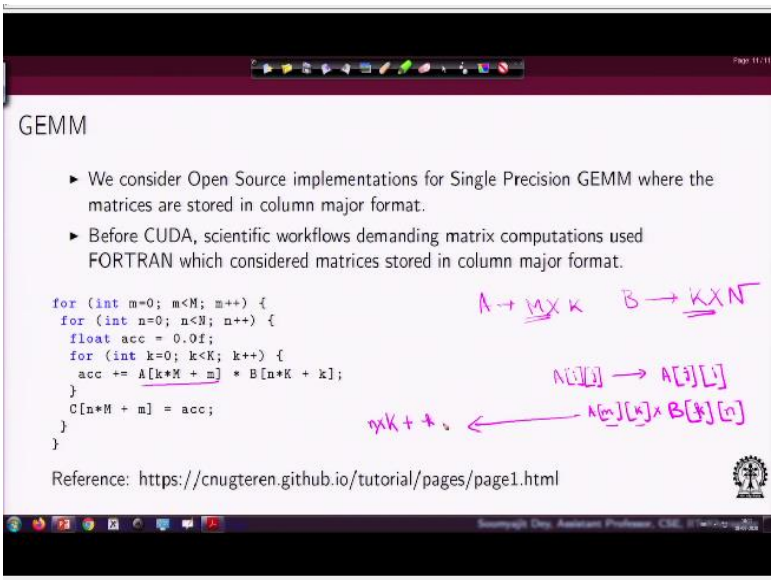
(Refer Slide Time: 25:43)



For example, if you just have a loop, here, we are just providing a simple example of a 3 level for loop for matrix multiplication, assuming a column major representation of the data. So what is really going to change here. So if you consider a column major representation of the data then when in terms of math, you are trying to multiply matrix A and B of dimensions M cross K and K cross N in reality, you will have the matrices arranged in the other way around, right.

So, that would be like K and M and similarly here, right. So, there is a transformation right is let us take a transpose, but that is it.

(Refer Slide Time: 26:39)



I mean we have to live with you. That also means when you are trying to access data points, so whatever was some A i J in my original problem That is going to change to some A j i right. So that is how sorry, if I just write it more clearly. So it is different, right. But the problem is, that also means that your programs are going to not change, right. So, when you are really writing a row major code, your matrix multiplication loop internally you would have been multiplying like this right.

So, you may have been multiplying entities like A i k multiplied by B, k, j, like that right where you are iterating over k, but now that is really going to change, well mathematically if it is, so then in terms of the code when you are trying to see what is the memory access expression, if this is the mathematical point that you are trying to use that is you are going to multiply A i k and B k j, but here in terms of memory access, they will be at different locations right.

Because since you are accessing in a column major way, so, you have to think it the other way around. And essentially you are going to use something different. So here, let us say for example, in my code the iterators are M N and K, right. So, let me just consider it here. That it is all M's

and N’s, right like this right and then since we are taking it column major, so, really we should have to think of the A k m entry right.

So, that means basically, what is the mathematical location is A m k, but in our memory for where the storage using column major fashion the elementary access must be the other way around, that means considering in a row major way if it is A k small m. So, you multiply k by capital M right number of rows and add smaller m and similarly for the other one the mathematical entities B k n.

But when you are accessing the real entity in memory, since the storage is in columnar major format, so we need to we just take it the other way around. So, we will have n multiplied by the iterator here, considering that we are multiplying matrices of this size, M cross K, so A is of this dimension and B is of dimension K cross capital N. That is why, for the first one, the entry I was looking for was A small m small k you write it the other way around.

That is you take the small k and multiply with M and add small m here, right because it is column major, and then for B's small k, small n, right, you take it the other way around. You take small n and multiply it with a capital K and add small k, right so there is a shift right. So, this is a change that we have to keep in mind when we are doing the matrix multiplication operation assuming that the data is stored in column format instead of row measured that we normally do for C and similar languages right.

So, this is the knife C I mean code for C based I mean C program like implementations of matrix multiplication. In the next lecture, we will touch upon the parallel versions assuming column major data storage and we will see how optimizations can be done for that. With this we will end here, Thank you.