

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


**Lecture-63**  
**Efficient Neural Network Training/Inferencing (Contd.)**

Welcome back to the lecture series on GPU architectures and programming. So, if you remember in the last lecture, we have covered these different this GEMM optimizations, the different ways the GEMM kernel can be optimized.

**(Refer Slide Time: 00:37)**

GEMM : The core computational kernel for deep learning

- ▶ GEMM is the most computational heavy kernel used in fully connected layers and convolution layers for a neural network.<sup>4</sup>
- ▶ The optimized implementations discussed so far focuses during the training phase.
- ▶ DNN inference refers to only a forward pass over a trained neural network.
- ▶ Training optimizations are not optimized for inferencing on embedded mobile GPUs



Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

**(Refer Slide Time: 00:39)**

DNN Pruning and Sparse Matrix Operations

- ▶ Several works have been proposed over the years that focus on pruning the weights of a neural network.
- ▶ This essentially implies that the weight matrices are now sparse in nature.
- ▶ Implementations for sparse matrix operations would be beneficial in this context.

Efficient Neural Network Training/Inference Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So in the current lecture, we will like to focus on this other ideas like how these sparse matrices are operated one, like now why is that important thing. Because people have often found that these neural networks pruning them is actually useful, right. Because if you I mean, DNN can be pruned and the resulting matrices can be sparse, then that that can actually lead to not many nice sparse matrix operations, which are often found useful, right.

So several works have been proposed over these years that focus on pruning the weights of a neural network. Because it I mean, if you prune the weights, then that leads you to a lighter representation of the network. And that also helps you to figure out well, there are many nice sparse matrix algorithms available how to make use of, make good use of such algorithms, right. So, if we consider such modern prune DNN kind of networks right, where the weights have been pruned.

And you have a neural network, which is represented by a sparse matrix, then such algorithms become very relevant, and some of them are also very useful for GPU style implementations, right. So that way implementations for sparse matrix operations are very beneficial in this context.

**(Refer Slide Time: 02:09)**

**Sparse Matrix Vector Multiplication: SpMV**

- ▶ There exists different formats for storing sparse matrices.
  - ▶ Diagonal format
  - ▶ ELLPACK
  - ▶ Coordinate Format (COO)
  - ▶ Compressed Sparse Row Format (CSR)

Soumyajit Dey, Assistant

And the particular algorithm we will be talking about here is sparse matrix vector multiplication. So, you have a sparse matrix and you are multiplying it with a vector. Now, that is also an important operation that you will have to do inside the neural network right. So, in general, these sparse matrix vector multiplication in short form is known as is SpMV. Now, of course, we have to understand that the very reason we want to exploit the sparsity of a matrix that is the absence of I mean non trivial elements.

The significant absence of non trivial elements in a matrix is that in general, if you are storing a matrix in the memory, you are actually consuming order of  $n$  square amount of space, but if you are considering a sparse matrix where most of the entries are, in general, trivial or 0 , then we like to use an alternate storage format, where I do not encourage this kind of order of  $n$  squared amount of storage requirement right.

So specifically for sparse matrices, there are several such well known formats, like diagonal format ELLPACK format, coordinate formats COO, and also compressed sparse row format, CSR.

**(Refer Slide Time: 03:32)**

The compressed sparse row format stores a sparse  $M \times N$  matrix in row form using three 1-D arrays (**val**, **col**, **ptr**).

- ▶ The arrays **val** and **col** are of length  $(nnz)$  which represents the number of non-zero values in the matrix.
- ▶ **col** stores the column index and **val** stores the non-zero value
- ▶ The array **ptr** stores the cumulative number of non-zero elements i.e.  $ptr[i]$  represents the total number of non-zero elements observed upto the  $i$ -th row. The values of the array are derived from the following recursive equation.
  1.  $ptr[0] = 0$
  2.  $ptr[i] = ptr[i-1] + \# \text{ non-zero elements in the } (i-1)^{\text{th}} \text{ row.}$

Page 3/10

Sourmyajit Dey, Assistant P

So what we will do is, in this lecture, we will talk specifically for the CSR format, how multiplication operation on matrices are performed using the CSR format, and how that can be optimized considering a CUDA based parallel implementation. So this is what we will focus on. So I will just recap.

**(Refer Slide Time: 03:57)**

**Sparse Matrix Vector Multiplication: SpMV**

- ▶ Several works have been proposed over the years that focus on pruning the weights of a neural network.
- ▶ This essentially implies that the weight matrices are now sparse in nature.
- ▶ Implementations for sparse matrix operations would be beneficial in this context.

Page 3/10

Sourmyajit Dey, Assistant P

Considering the way I mean in large neural networks which we are going to really face people perform this kind of weight pruning. Now, weight pruning will lead to a sparse matrix and that is the reason you want to do the weight pruning. Because with a sparse matrix you can have nice algorithms to work with and nice storage methods. So one of them is the CSR storage method, which we like to introduce here.

And the next thing is we like to use it for our purpose, like optimizing the matrix multiplication further. So the compressed sparse row format, it stores us parts in cross and matrix in a row form where it uses three 1-D array. So let us call the arrays as val, col, and ptr. Now the arrays are will refer to them as value, column and PTR, right the arrays val and col are of length nnz, which represents the number of nonzero values in the matrix.

So let us say inside  $M$  cross  $N$ , I have got only nnz number of non-zero values and everything else is set to zero right. So, essentially we are just interested in storing these nnz number of entries. And also we are interested in figuring out what are the locations in which these non-zero values are stored, because as we can safely say that all the other locations are just having zeros right. Now, the col or column stores the column index and the val stores the non-zero value.

So, we have one 1-D array, which is storing all these nnz number of non-zero values, we have another 1-D array called col, which is storing the column indices for each of these nonzero values that well this nonzero value is actually positioned in this column of the sparse matrix like that right. And then, I have the other array called ptr, which is storing the cumulative number of nonzero elements. So essentially ptr<sub>i</sub> will represent the total number of nonzero elements observed up till in the i<sup>th</sup> row.

We will see what it means, the value of the array derived from a recursive equation will let us not get into that. Rather, maybe we will have a better idea by looking into an example.

**(Refer Slide Time: 06:20)**

CSR Format Example

Matrix:  $\begin{bmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{bmatrix}$

Handwritten notes: *Matrix*, *M x N matrix Symbolic Representation 4x6=24*

CSR arrays:  $\text{ptr} = [0\ 2\ 4\ 7\ 8]$ ,  $\text{cols} = [0\ 1\ 3\ 2\ 3\ 4\ 5]$ ,  $\text{vals} = [10\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$

Handwritten note: *CSR*

So let us take the simple example. So on the left side, we have our CSR we have a normal M cross N matrix. So in this case, as you can see the amount of storage I require is 4 cross 6. So that is 24 and here on the right hand side, I am having the storage in CSR format and we are just using three 1-D arrays right. So first of all, what are the number of values 1 2 3 4 5 6 7 8, right. So, these are all stored in the val array right now.

**(Refer Slide Time: 07:43)**

CSR Format Example

Handwritten note:  $\text{row}[i] \rightarrow$  first non-zero entry =  $\text{vals}[\text{ptr}[i]]$  in column  $\text{cols}[\text{ptr}[i]]$

Matrix:  $\begin{bmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{bmatrix}$

Handwritten note: *Matrix*

CSR arrays:  $\text{ptr} = [0\ 2\ 4\ 7\ 8]$ ,  $\text{cols} = [0\ 1\ 3\ 2\ 3\ 4\ 5]$ ,  $\text{vals} = [10\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$

Handwritten notes:  $\text{vals}[\text{ptr}[0]]$ ,  $\text{ptr}[0]$ ,  $\text{vals}[\text{ptr}[1]]$ ,  $\text{ptr}[1]$ ,  $\text{vals}[\text{ptr}[2]]$ ,  $\text{ptr}[2]$

So, let us try to figure out how this thing can be constructed back okay. So, look at the so we will start by looking at this ptr array. So this essentially is containing information related to the rows okay. Now, so I am now interested in constructing the 0th row okay. So what we will do is where

we will see that the entry for the 0th row here is 0. Now this 0 means well you start looking for the entry in val 0 okay.

This means you start looking for the entry in vals 2 and so on so forth. Then 4 means you start looking for the entry in vals 4 like this, right. So if we just now try and construct this thing, so for the 0th row, we get that vals 0 for ptrs 0th position I have 0. So let us see what is this 0, this is nothing but this is the content in ptrs 0th position, right. This is the content in ptrs first position. This is the content in ptrs second position right there.

So essentially from this for each row, I am looking into ptr. And I am getting the corresponding index in val, right. So this is essentially telling me what is the location of the first non trivial element in that row. So for let us take an example for the first row, I get an index 2. Now, I look into so I follow this index 2, and they look into vals 2 right. So essentially, the way what I am doing is first row I am looking into ptr 1, and then I am looking into vals of ptr 1.

That is what I am doing right row 1. So if I just write it like this row 1, first non trivial entry equal to vals ptr 1 row 1's first non trivial entry is located vals ptr 1 in column cols ptr 1. So for 1, you go to ptr 1. So, that is true, right. So, you know just look into the location for it, right that is how it is getting done right. So it is here that is the column is 30. So, I hope that is clear. So that tells me that for row 1 so just remember this.

So for ptr's, position 1 gives me some information about some element in row 1, right. So, row 0 row 1, fine. Now then in row 1, what do we have. So, you use ptr 1's position. And since it contains 2, it tells me that the value the first non trivial entry in row 1 is located at this position, which is val sort of ptr 1. So, whatever is the content of ptr 1, you would now you know look into that location of vals right.

So, that contents 30. So, that means N the corresponding column position is what. So, that simply cols of ptr 1, right 0 and then 1, since, sorry ptr 1 is 2 so 0, so columns ptr 1 that is containing 1. So that is column 2 that is containing 1, right. So now I have gone overall

information, the overall information is the first non trivial entry of row 1 is of value 30. And it is located in column 1. And I already know the row, right.

So this is going to be 0. And this is 30 right. Well, in that way, I mean, I can say that val, I will be able to figure it out all the non trivial entries in the first non trivial entries in each row, right. But what about other entries. Well, let us first use this idea to figure out all the first non trivial entries in each row. For example, 0, so let us go here. So that is 10, 0, right. So that means it is right here, right.

Similarly for 4, you just follow it right. So location 4 here 0 1 2 3 4 50 and for 50 what is the column 0 1 2 3 4 2 right. So that is a 50 right. And the last one is 8, right. So after 4 what do you really have 7, right. So go to 7 here 0 1 2 3 4 5 6 7. It is 80 right. So where is 80 lok at it, as it is saying it is in the fifth position, right. And there will be 2 more positions here right, and 80 will come here in the fifth position 0 1 2 3 4 5 right.

So in that way, so this clear because this is the last cohesion. So the all the previous entries must be trivial, right. So these are all trivial entries. But we know up to this, but what about the rest. Well, the rest is as easy to form. As you can see here, it is 0 and the next is 2 that means before this in the first row, I have got 2 values. Well, that means the value next to 10 I come here to the value 10 right, the next location is telling that val you look into vals 2.

So that means the value previous to vals 2 is also in the first row and the location of that the column wise location I can just get from here, right. So that tells me 20 is there and then immediately after 2 what do I have if 4 and that takes me right here to 50 right and for 50 I know where to go, right. So essentially what we are following is, if we just subtract them, you know that in each row, how many you do you really have right.

So, essentially, this 2 minus 0 is telling you how many are there in the first row, 4 minus this. So that is 2. So you have 2 elements in the second row, or the sorry, the 0th row has 2 elements. The first row has 2 elements. Since the first row has 2 elements, how do you get the second element.



Well, if you follow 4, you get to 50. Before that you have 40. If you have followed 2, you got to 30. So whatever is next to 30, that is 40.

And is before this one, where do you go from 4. So this is also in the same row. That is the information. Now of course, the location you can get right from here, this is third right, so here it must be a 0 and here it must be 40. And since there only 2 elements here. So, all the other locations has to be 0 right. So, I hope now, it is clear that if we just keep on following this process, I can just simply keep on building the CSR.

**(Refer Slide Time: 17:14)**

```
1 template <typename T>
2 void spmv_cpu(T *val, T *vec, int *cols, int *ptr, int N, T *out)
3 {
4     for (int i = 0; i < N; i++){
5         T t = 0;
6         for (int j = ptr[i]; j < ptr[i + 1]; j++){
7             int col = cols[j];
8             t += val[j] * vec[col];
9         }
10        out[i] = t;
11    }
12 }
```

Well, the next thing is we will just shortly touch upon that how really this is going to help you to do I mean, assuming that you have the storage in the CSR format, you are saving on the storage. Well, how do you really do the multiplication. Well, is easy, right. All you need to do is you already have the vector, all you need to do is you have to figure out the elements in the corresponding column, right. You have just required to find out the elements located in the corresponding column because of vector is already there.

So how do you just do it. So you run this outer loop, you just running this outer loop. Using this outer loop you are scanning the ptr elements right. And then so for every i right, so essentially this outer loop is telling you which row to access. Now for the regular matrix. Now that row is for that you have to go here. So suppose for any ith row, you have to start from ptri, and how

much really is this inner loop going to iterate. Well, that is easy to get, just observe our previous definition.

**(Refer Slide Time: 18:25)**

SpMV: CUDA Scalar Implementation

- ▶ Assign each thread, the task of multiplying one row of the input sparse matrix with the dense vector.
- ▶ The number of blocks launched by the kernel is therefore equal to  $M/BlockSize$  where  $M$  represents the number of rows of the input matrix and  $BlockSize$  represents the block dimensions used while launching.

Reference: N. Bell and M. Garland, Implementing Sparse Matrix vector Multiplication on Throughput-oriented Processors

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So all that we said was that this location 0 and this ptr 2, so what is the difference. That is the number of elements in the 0th row from 2 to 4 the difference is there is a number of elements in row 1, 4 to 7, there is a difference is the number of elements in row 2 like that, right. So that is why you start from ptr<sub>i</sub> and you have up till ptr<sub>i</sub> plus 1, how do you get the value, you just get the value in this way that since you have set ptr<sub>i</sub> to j, you will just look into Ptr val j right.

Because that is the val, and the column you will just get because you have already got ptr<sub>i</sub>'s value in j, just sample from the column matrix cols j that is giving you the column, right. And once you have got the column, you know I mean how to get the vector values, right. So, that is how you are going to do the matrix vector multiplication by sampling these vectors corresponding locations value. Because this column whatever you are getting, that is also telling you which value to sample from the vector right.

**(Refer Slide Time: 19:31)**

I hope that is clear. So with each new iteration, you are going to change the  $j$  that means you are simply incrementing  $j$ , right. That means you are just proceeding through the row, right. So you are just proceeding through the value matrix you are going to the next element in the value matrix, because  $ptr$  has told you in value matrix at which point to start, right. So that is  $val_j$ , and then you are just incrementing following  $val_j$  right, I hope this is clear.

So if we just go back in this example, let us say you have started here from 0 you are here, then you are just following the value metrics row wise right. So, all that you need is the start position where to start, for example, 50 60 70, there in a row, you get the idea that have to start here, and then you are just following the elements in the  $val$  matrix. And while you follow them, you are also hopping over the corresponding columns in the row matrix.

I hope this is clear. So, again to just give an example, consider that this is your vector. Now, when you are hopping through this row, instead of our normal matrix multiplication, where you start from here and scan through up to here. You are doing it in an optimized way from  $ptr$  you get to  $val$ , you get 50, you get the position of 50 which would be 0 1 2 3 4 0 1 2 3 4. That is 2, so you know that you have to start from here.

And you know that you have to go up to this because the position next just up to the next  $ptr$  position. So you just scan through this, while you are scanning to the whatever you were doing is

for each of these values, you also get to know what are their column indices, and then you just scan through those column indices only in the vector because the others are not required because they will be getting multiplied by the 0, right.

So that is how it is getting optimized. So that I hope now is clear from the  $\text{ptr}_i$  to  $\text{ptr}_i + 1$  this route is going to run you get the starting point  $j$  in your val you do  $j + \text{plus}$  to skip through the val, for each of the  $j$  locations you also know that what is the corresponding column index, and that gives you the which vector component to multiply. And that is how this CPU implementation will work.

Well, how will the CUDA implementation work. Let us do a simple thing first. We will assign each thread the task of multiplying one row of the input sparse matrix with the dense vector. So unlike the CPU implementation, where we had this outer loop now, we will not have this outer loop right because every thread will know which row to work with. And accordingly it will just do the job. So, every thread is in put in charge of multiplying one row of the input matrix with the vector,

The number of blocks that will be launched will therefore be equal to  $M$  divided by the block size. So let us say  $M$  represents the number of rows and rows of the matrix and block size represents the block dimension that I want to use while lunched right. So every block has got that many threads right. So, the number of blocks will simply be  $M$  by block size and  $M$  block size, they have got that many threads.

So overall, I have got this as many number of threads as there are going to be rows right. And that is as a naive implementation for the corresponding CUDA kernel.

**(Refer Slide Time: 22:54)**

```
SpMV: CUDA Scalar Implementation

1 template <typename T>
2 __global__ void spmv_csr_scalar_kernel(T * d_val, T * d_vector, int * d_cols, int
   * d_ptr, int N, T * d_out)
3 {
4     int tid = blockIdx.x * blockDim.x + threadIdx.x;
5     for (int i = tid; i < N; i += blockDim.x * gridDim.x)
6     {
7         T t = 0;
8         int start = d_ptr[i]; int end = d_ptr[i+1];
9         // One thread handles all elements of the row assigned to it
10        for (int j = start; j < end; j++)
11        {
12            int col = d_cols[j];
13            t += d_val[j] * d_vector[col];
14        }
15        d_out[i] = t;
16    }
17 } //Kernel Launch parameters: <<<M/BlockSize>>,BlockSize>>>
```

So have a look. So this would be my corresponding CUDA kernel. As you can see the I am writing a templated code because we are not trying to specify what kind of data type to be will be there for the matrix. So this will work for any of them right. Can we float can be int and all that. So this is my kernel. So, since I am charging each thread to multiply one row of the matrix with a column with the vector, the dense vector, so we will just find out the tid, the global ID of the thread.

And then inside this loop, we will start from tid. And we will go up to the end actually, like or whatever is the number of threads, I am going to have, right the total number of rows, right. And then inside I have this thread that is handling all the elements of the row that have been assigned to it. So as you can see, I am running this thread this outer loop from tid and M hopping over with block dimension multiplied by grid dimension, any idea why we are really doing it.

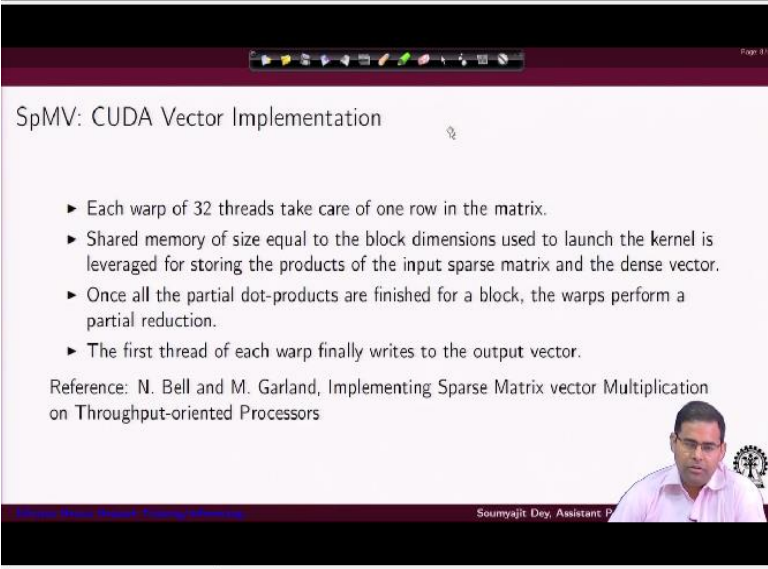
So let us just chop it out. So I want to row wise multiplication. But again, let us remember this is the column major format right. So the values I am really interested in, they would be like this right. So the thread in each of its iterations has to get values like this, I mean, from which location of the matrix that i index will be found by incrementing i by block dimension multiplied by the good dimension.

As you can understand that this multiplication gives me the number of rows, right, this multiplication is going to be the number of rows of the mathematical matrix. So I will just hop over to get to the next  $i$  like this and then with those  $i$ 's, what am I doing is once am I am inside the row, I have figured out which value to get, then that thread handles all the elements of the row that have been assigned to it.

So, I mean, the rest of it is very simple. I mean it is similar to the CPU on implementation. So you start from the  $d$  ptr so that is the ptr matrix position and you are going to end of course at the  $d$  ptr plus 1 and for each of these locations, you are sampling the column and you are getting to what location from the vector to get and also you have you are accumulating here the value and in  $t$  and you are multiplying this vector with the val matrix corresponding location that you have in  $j$ , right.

So that is very simple similar to the CPU implementation right. So, in that way you are going to continue and all you are doing is you have to figure out what is the  $i$  value from where to start in the ptr right. So, to get to the different  $i$  values in the ptr since you have different representation here, you are having this outer loop. So, I hope that would be really okay with the scalar implementation. Now, what about the vector implementation.

**(Refer Slide Time: 26:39)**



SpMV: CUDA Vector Implementation

- ▶ Each warp of 32 threads take care of one row in the matrix.
- ▶ Shared memory of size equal to the block dimensions used to launch the kernel is leveraged for storing the products of the input sparse matrix and the dense vector.
- ▶ Once all the partial dot-products are finished for a block, the warps perform a partial reduction.
- ▶ The first thread of each warp finally writes to the output vector.

Reference: N. Bell and M. Garland, Implementing Sparse Matrix vector Multiplication on Throughput-oriented Processors

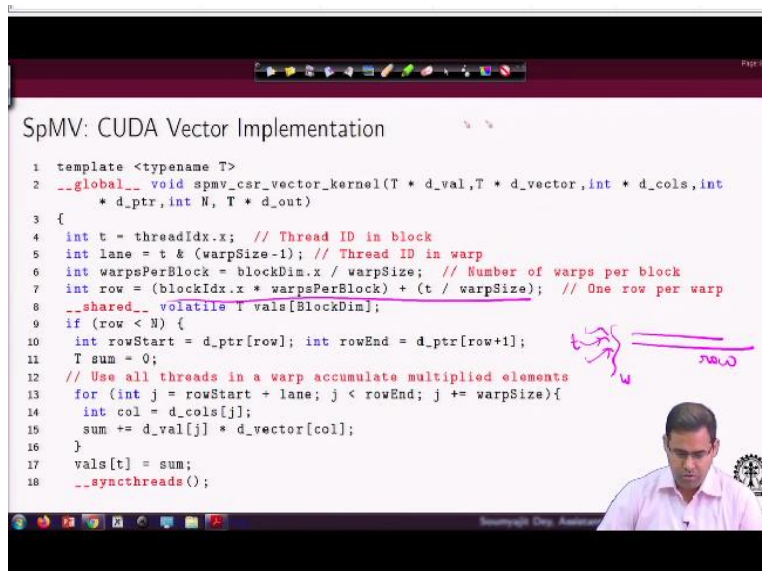
Soumyajit Dey, Assistant P

So, it is simple. Let us just optimize like this we bring back our previous ideas of reduction. So for each warp of 32 threads let warp together take care of one row in the matrix right. So that means we will apply our previous reduction ideas here. So a shared memory whose size is equal to the block dimension that is used to launch the kernel and this shared memories used to leveraged for storing the products of the input sparse matrix and the dense vector.

So now in some way we will use that. I mean, first of all, where are we really doing this. Again, we are considering very large matrices, lot of rows, right. But again, we also want to optimize. So for that, we want the threads to cooperate. That is why we want each warp to take care of one row of the matrix right. And also, while each wrap is taking care of one row of the matrix, I want to have shared memory based partial loads, right.

Where the shared memory size is equal to block dimension, so that many threads will be loading the data to the shared memory in a cooperative way. Then they will be performed in partial dot products, once the partial dot products are performed, the warp will be 32 threads will perform a row wise partial reduction. Then again, I will bring some more data into the shared memory base tile for the next set of points. Again, the worst will perform partial products and partial reduction like that, right.

**(Refer Slide Time: 28:26)**



```
SpMV: CUDA Vector Implementation

1 template <typename T>
2 __global__ void spmv_csr_vector_kernel(T * d_val, T * d_vector, int * d_cols, int
   * d_ptr, int N, T * d_out)
3 {
4     int t = threadIdx.x; // Thread ID in block
5     int lane = t & (warpSize-1); // Thread ID in warp
6     int warpsPerBlock = blockDim.x / warpSize; // Number of warps per block
7     int row = (blockIdx.x * warpsPerBlock) + (t / warpSize); // One row per warp
8     __shared__ volatile T vals[BlockDim];
9     if (row < N) {
10        int rowStart = d_ptr[row]; int rowEnd = d_ptr[row+1];
11        T sum = 0;
12        // Use all threads in a warp accumulate multiplied elements
13        for (int j = rowStart + lane; j < rowEnd; j += warpSize){
14            int col = d_cols[j];
15            sum += d_val[j] * d_vector[col];
16        }
17        vals[t] = sum;
18        __syncthreads();

```

So let us see how it works. So in our CUDA vector style implementation, we get the thread idea of each block. But then I need to figure out what is the location of this thread inside the warp. So what that is, let us say that my warp size is 32,  $32 - 1$  gives me an array of all 1's, do a bitwise  $\&$ . So that gives me the offset that is the thread ID in the warp. So essentially, I am getting the thread ID.

I mean, what is his location inside the warp. So it is basically a modular 32 operation, I would say, right. Now, the next thing I figured out is, how many warps do I have inside a block. So I know my block dimension, I am considering, I mean one dimensional blocks, right. And then I just divide the block dimension by a warp size. So that gives me that inside one thread block how many warps do I have. Why am I really computing this.

Because I want the warps to each of the warps to do the reduction for one data points, right. So for each block, I figure out how many warps are there per block, and I am really going to use this later on we will see, first we have computed this warps per block. But the issue is, I want to figure out the threads inside the warps. They are going to do compute or reduction for which row, that is being taken care of by the next line. Well, I mean, I just like to say that these are a bit advanced, but I hope this will be clear.

So you see figured out the position of the thread inside the warps, then I found that well, this is the total number of threads launched per block. And I want to figure it out by figuring that what is the total block dimension. And by divided by the warp size that gives me that for each block of data, how many warps do I really have right.

Now, since I have figured out how many warps I do really have, and I multiply it by the block size, that gives me how many blocks are already, I mean, how many warps are already done. I mean, I am only considering the current block, right. So this multiplication gives me that, well, I mean, how many warps I have already got covered by the previous blocks, and then I multiply and sorry, then I add with this offset of  $t$  divided by the warp size.



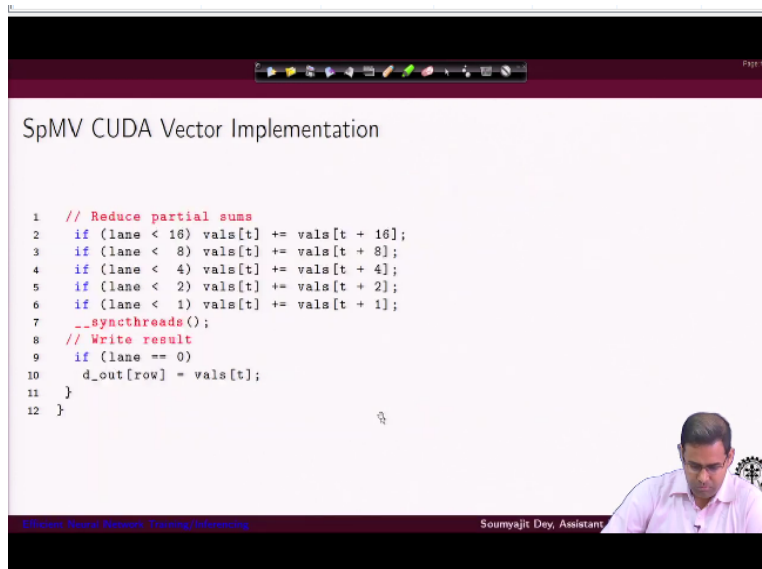
So essentially, what does this tell me. First of all, I have figured out, well, what is my thread. And then let us understand that since a warp of threads are going to work for a row. So I have essentially got multiple threads working for the same row, right. So they all are elements of the same warp, right. So by using this formula, I am trying to figure out this in general, any thread  $t$ , it is going to work for which row.

That is what I am trying to figure out. So, what do we really do is we have already figured out what is wraps per block, you multiply the block ID. So that gives me that how many warps are already done. And then you just figure out well, you just edit with the offset. And that tells you that this current thread is going to work on this row, right. And of course, you have for each block you have got this start time  $t$  which will be there for this block dimension.

The next point that comes is well, I am going to load the data from the row, right. So for that every thread will load the corresponding data as long as this is less than  $n$ . So you what you get is with this row value, you figure out what is the position to start from  $dptr$  row and then from  $dptr$  row plus one, you know, what is the position to  $n$  right. I hope this clear this transformation is required.

Because every thread is not working for each row, a set of threads inside the warp are going to work on a row. So for every thread by doing this calculation and figuring out for which rows is going to work. And then I am going to figure out, I am going to load data from which point to each point right. Well, once that is done, use all the threads in our warp to accumulate the multiplied elements. That is what I am going to do right. So first, you see I have already figured out from where to start and where to end right.

**(Refer Slide Time: 32:59)**



```
1 // Reduce partial sums
2 if (lane < 16) vals[t] += vals[t + 16];
3 if (lane < 8) vals[t] += vals[t + 8];
4 if (lane < 4) vals[t] += vals[t + 4];
5 if (lane < 2) vals[t] += vals[t + 2];
6 if (lane < 1) vals[t] += vals[t + 1];
7 __syncthreads();
8 // Write result
9 if (lane == 0)
10     d_out[row] = vals[t];
11 }
12 }
```

And once that is done I mean well as you can see that some part we are skipping here for brevity which is basically how the loads will be done into the vals and all that I hope you can figure it out. But we are just talking about how the data from these input matrixes d vals and they are actually going to be accumulated here. Then what you are going to do is this part, which is the important part.

That is once I have these partial sums calculated, as you can see, now, I activate the lanes. So this is very much like the reduction optimization we taught in our reduction class where we did reduction inside a warp, right. So, that is what you can see that we are assigning for each thread will which parts of the val is going to add right. So, once the partial multiplied values are calculated, after that all of them are written back to this shared memory right.



Now, this is where the importance of the shared memory is coming because now I want the wrap to do cooperative reduction. And of course for that, I will need the data to be in the shared memory. So once every thread has figured out, which row it is belonging to, and this is non trivial, because as we discussed earlier a set of threads inside a warp are going to work for the same row. Once that is figured out, I just figured out what is my row start and row end.

And I use them for the normal loop for doing the multiplication. After the multiplication for where the accumulation is done, I just told back the value in the shared memory, where I will do

reduction using my warps, right. And now, using these consecutive use statements, we are not getting into these details, this is something we have already covered, we are able to do the final reduction. And that gives me the final result, which is of course, written back by only the thread which has the lane value equal to 0's, right. Because we will have one thread per warp with this value equal to 0 and that gives me the final output.

**(Refer Slide Time: 36:00)**

Teaching Assistants

	<b>Anirban Ghose</b> PhD scholar in Department of Computer Science and Engineering IIT Kharagpur <b>Research interests:</b> Intelligent Scheduling and Compiler Optimization Techniques for Heterogeneous Architectures
	<b>Srijeeta Maity</b> PhD scholar in Department of Computer Science and Engineering IIT Kharagpur <b>Research interests:</b> Real time scheduling on heterogeneous embedded architectures

Soumyajit Dey, Assistant Professor

So with this, we will also like to say that this is our kind of end to the coverage for the course, I hope you had a nice exposure to GPU architectures and programming. And just I would also like to thank 2 of my Ts who are Anirban Ghose and Srijeeta Maity, they are doing PhD in under my supervision in the Department of Computer Science, and without their cooperation and help with respect to assignments and this slide creation, this course would really not have been possible.

**(Refer Slide Time: 36:35)**

Page 1/10

## References

1. Perceptron: <https://datasciencelab.wordpress.com/2014/01/10/machine-learning-classics-the-perceptron/>
2. Neural Networks
  - ▶ Welch Labs Youtube Videos
  - ▶ moDNN: Memory Optimal Deep Neural Network Training on Graphics Processing Units by Chen et al published in TPDS 2019.
  - ▶ CS231n: Convolutional Neural Networks for Visual Recognition
3. GEMM: <https://cnugteren.github.io/tutorial/pages/page1.html>
4. SPMV: <https://github.com/poojahira/spmv-cuda/tree/master/code/src>

Efficient Neural Network Training/Inference

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And finally, these are the important references that are there for the last set of slides. Of course, for perceptron part we refer to this link, for the neural networks part and of course, there were this nice Welch labs videos. And you can look into this a recent paper in called moDNN, where they discuss about optimizing the memory consumption while doing the neural network training.

Now this is really something that is useful. I mean, that is something we could not cover, but it is really useful. And there were several other sources from which we borrowed material for the convolution neural networks, the GEMM and the SPMV. So, I hope in general, this was a nice exposure for you. And that would be all from our side, I thank you for attending the course.