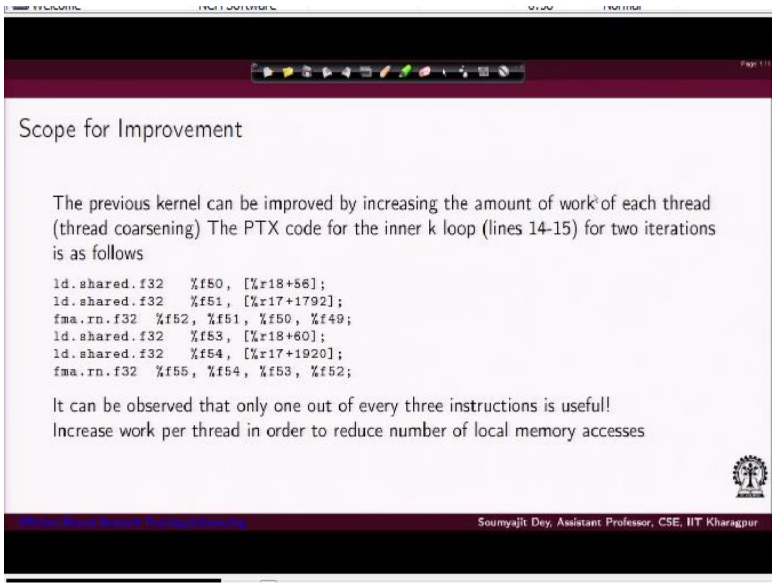
GPU Architectures and Programming
Prof. Soumyajit Dey
Department of Computer Science and Engineering
Indian Institute of Science Education and Research-Kharagpur

Lecture-62
Efficient Neural Network Training/Inferencing (Contd.)

Hi, welcome back to the lecture series on GPU architectures and programming. So, if you recall, we have been discussing different possible GEMM optimizations. For example, we started with the basic GEMM, right. The first optimization was just pushing in our normal tiling concepts, right. So, how the idea of tiling helps in matrix multiplication when done with shared memory based tiles.

(Refer Slide Time: 00:51)



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

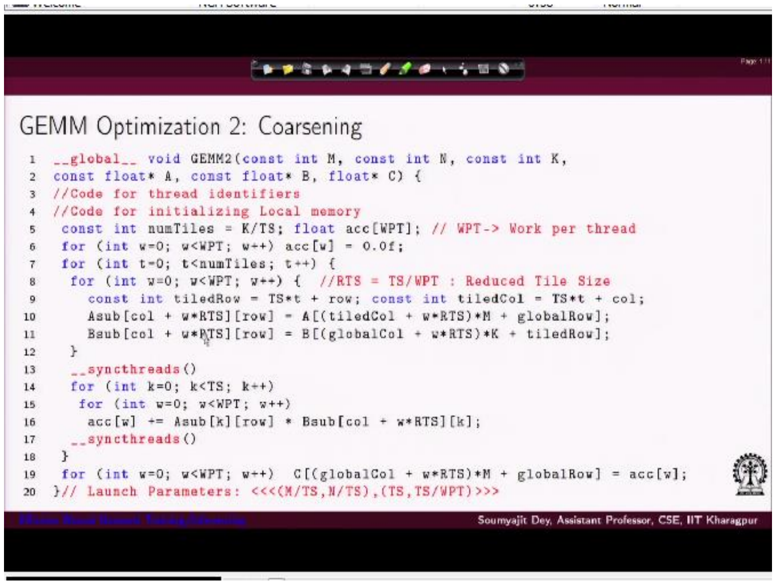
```
ld.shared.f32   %f50, [%r18+56];
ld.shared.f32   %f51, [%r17+1792];
fma.rn.f32  %f52, %f51, %f50, %f49;
ld.shared.f32   %f53, [%r18+60];
ld.shared.f32   %f54, [%r17+1920];
fma.rn.f32  %f55, %f54, %f53, %f52;
```

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

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

But then we also saw that leads to a small amount of compute with respect to the loads and that can be further improved.

(Refer Slide Time: 01:00)



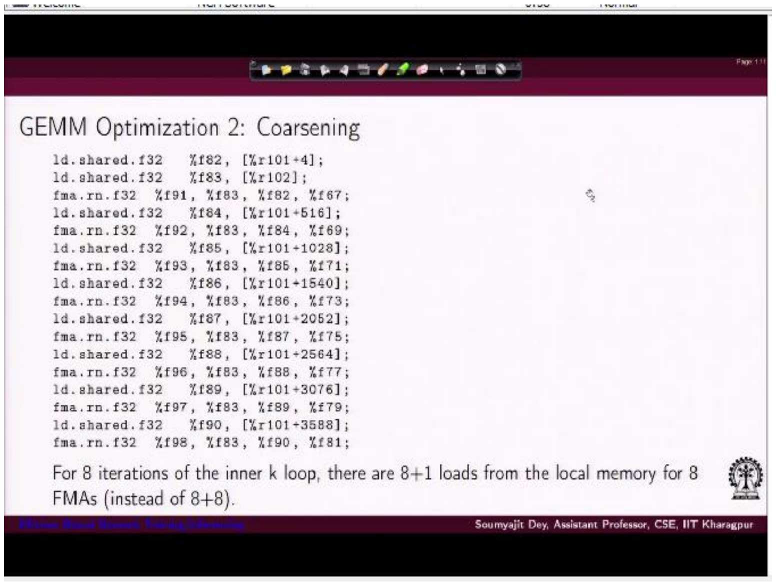
GEMM Optimization 2: Coarsening

```
__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] = acc[w];
}// Launch Parameters: <<<(M/TS,N/TS),(TS,TS/WPT)>>>
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So, all that we did was we coarsened the threats.

(Refer Slide Time: 01:05)



GEMM Optimization 2: Coarsening

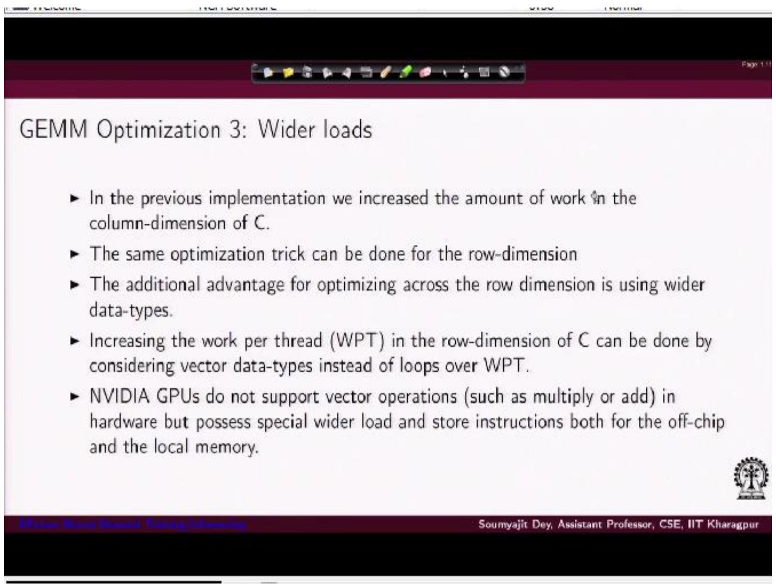
```
ld.shared.f32   %f82, [%r101+4];
ld.shared.f32   %f83, [%r102];
fma.rn.f32  %f91, %f83, %f82, %f67;
ld.shared.f32   %f84, [%r101+516];
fma.rn.f32  %f92, %f83, %f84, %f69;
ld.shared.f32   %f85, [%r101+1028];
fma.rn.f32  %f93, %f83, %f85, %f71;
ld.shared.f32   %f86, [%r101+1540];
fma.rn.f32  %f94, %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).

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And in that way, we could actually reduce the ratio of compute, I mean, we could actually reduce the ratio of load with respect to the computes. And to say in other words, we actually were able to increase the amount of compute per load, right, by doing the coarsening optimization.

(Refer Slide Time: 01:22)



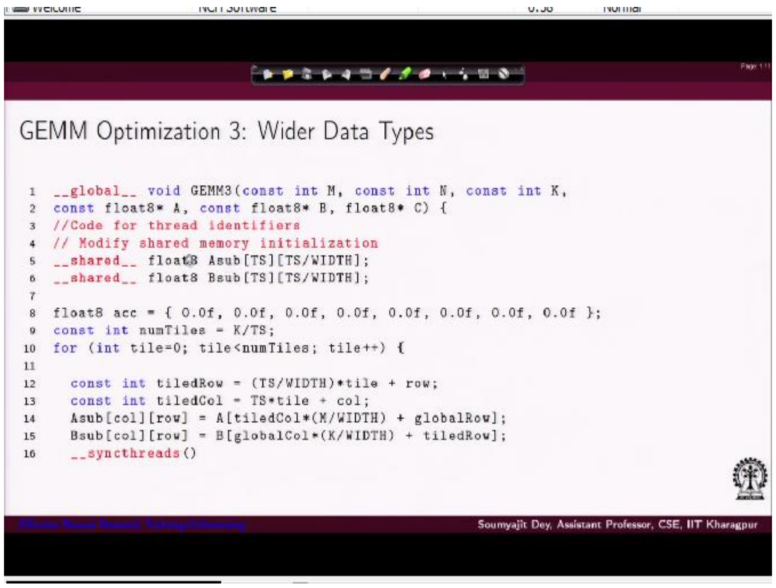
GEMM Optimization 3: Wider loads

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

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

And then we exploited the wider loading GEMMs that are available in NVIDIA GPUs because they do not support vector operations like multiply and add, which is vectorized. But they actually provide you with wider load and store instructions.

(Refer Slide Time: 01:36)



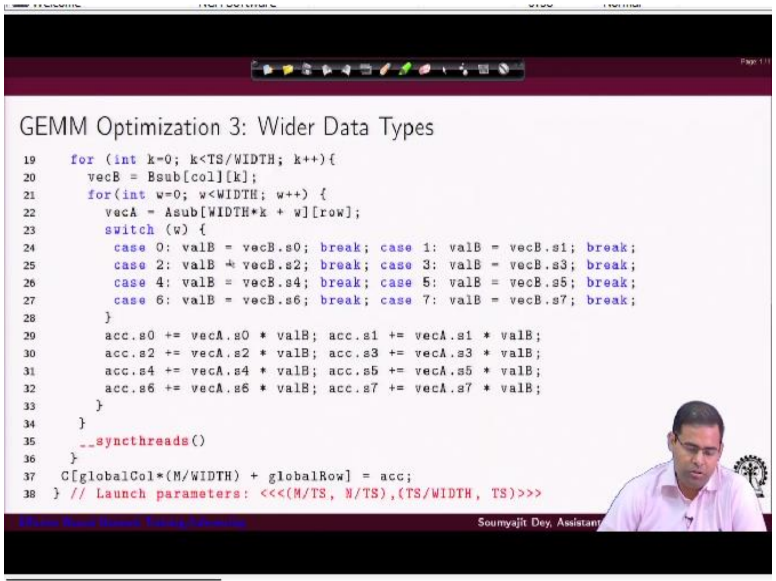
GEMM Optimization 3: Wider Data Types

```
__global__ void GEMM3(const int M, const int N, const int K,
const float8* A, const float8* B, float8* C) {
//Code for thread identifiers
// Modify shared memory initialization
__shared__ float8 Asub[TS][TS/WIDTH];
__shared__ float8 Bsub[TS][TS/WIDTH];

float8 acc = { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f };
const int numTiles = K/TS;
for (int tile=0; tile<numTiles; tile++) {

  const int tiledRow = (TS/WIDTH)*tile + row;
  const int tiledCol = TS*tile + col;
  Asub[col][row] = A[tiledCol*(M/WIDTH) + globalRow];
  Bsub[col][row] = B[globalCol*(K/WIDTH) + tiledRow];
  __syncthreads()
```

Soumyajit Dey, Assistant Professor, CSE, IIT Kharagpur

So we made good use of this float8 kind of data types here for kind of using our tiles with I mean, using the wider load instructions for filling in the tiles. That is how we will put it.

(Refer Slide Time: 01:51)



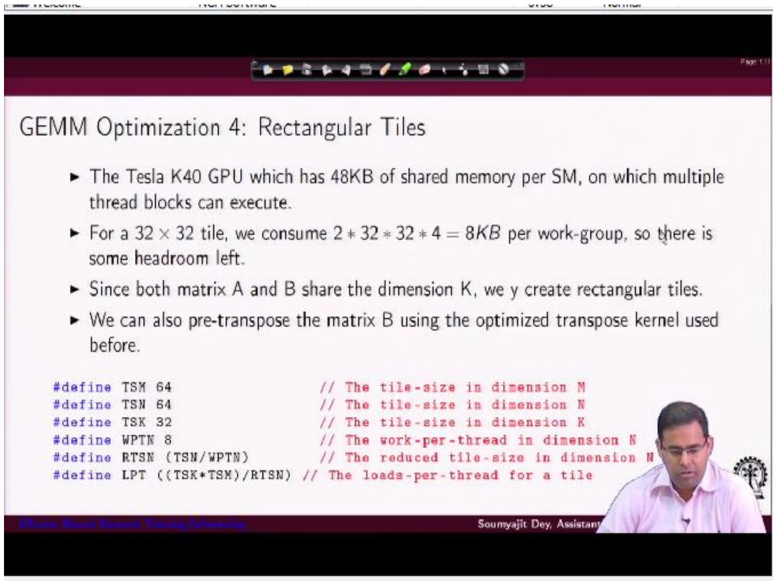
GEMM Optimization 3: Wider Data Types

```
    for (int k=0; k<TS/WIDTH; k++){
      vecB = Bsub[col][k];
      for(int w=0; w<WIDTH; w++) {
        vecA = Asub[WIDTH*k + w][row];
        switch (w) {
         case 0: valB = vecB.s0; break; case 1: valB = vecB.s1; break;
         case 2: valB = vecB.s2; break; case 3: valB = vecB.s3; break;
         case 4: valB = vecB.s4; break; case 5: valB = vecB.s5; break;
         case 6: valB = vecB.s6; break; case 7: valB = vecB.s7; break;
        }
        acc.s0 += vecA.s0 * valB; acc.s1 += vecA.s1 * valB;
        acc.s2 += vecA.s2 * valB; acc.s3 += vecA.s3 * valB;
        acc.s4 += vecA.s4 * valB; acc.s5 += vecA.s5 * valB;
        acc.s6 += vecA.s6 * valB; acc.s7 += vecA.s7 * valB;
       }
     }
     __syncthreads()
    }
   C[globalCol*(M/WIDTH) + globalRow] = acc;
  } // Launch parameters: <<<(M/TS, N/TS),(TS/WIDTH, TS)>>>
```

Soumyajit Dey, Assistant

But that also required us to go through this complex switching of values, which we have already explained in the last lecture right, how to support the wider data types and all that.

(Refer Slide Time: 02:03)



GEMM Optimization 4: Rectangular Tiles

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

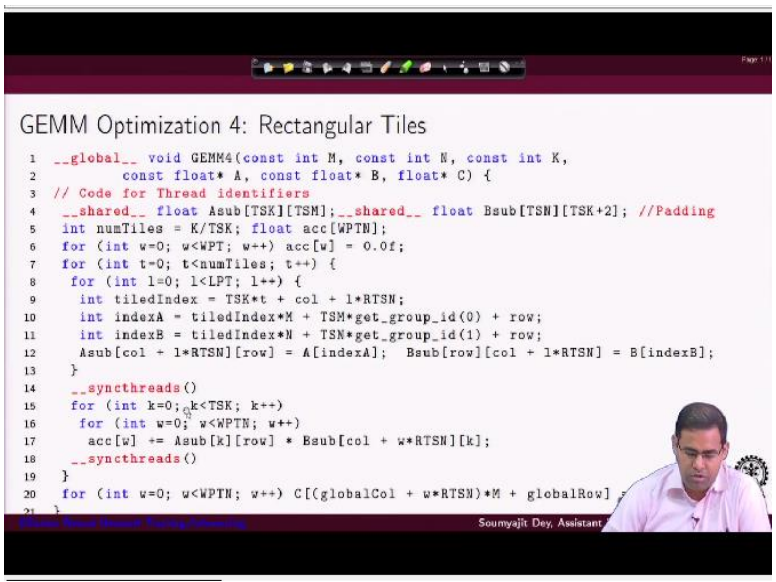
```
#define TSM 64                   // The tile-size in dimension M
#define TSN 64                   // The tile-size in dimension N
#define TSK 32                   // The tile-size in dimension K
#define WPTN 8                   // The work-per-thread in dimension N
#define RTSN (TSN/WPTN)       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

Soumyajit Dey, Assistant

But the next thing that it comes is called what we call as rectangular tiles. So we will just do a brief recap like why that is required. So in the rectangular tiles part if you remember; that K40 GPUs have got this 48KB of shared memory, right. Whereas for these tiles that we have defined, they were 32 cross 32 tiles, 2 of them, each storing 4 bytes of data. So it was overall 8KB. So that is like the basic reason that why you want to have a rectangular tile right.

Now well, what do we do with that. So let us just browse through this idea of rectangular tiles once again. So all that the idea was pointing to 2 is that since we have a large amount of shared memory, so we can increase the tile size and that is what we did right. So we increase the tile size from 32 cross 32 to tiles of size 64 right.

(Refer Slide Time: 03:03)



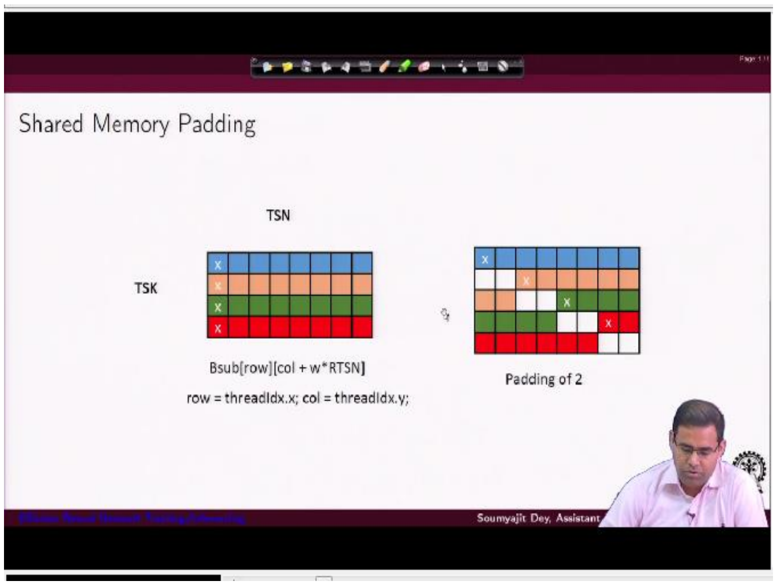
GEMM Optimization 4: Rectangular Tiles

```
__global__ void GEMM4(const int M, const int N, const int K,
        const float* A, const float* B, float* C) {
// Code for Thread identifiers
 __shared__ float Asub[TSK][TSM];__shared__ float Bsub[TSN][TSK+2]; //Padding
 int numTiles = K/TSK; float acc[WPTN];
 for (int w=0; w<WPT; w++) acc[w] = 0.0f;
 for (int t=0; t<numTiles; t++) {
  for (int l=0; l<LPT; l++) {
   int tiledIndex = TSK*t + col + l*RTSN;
   int indexA = tiledIndex*M + TSM*get_group_id(0) + row;
   int indexB = tiledIndex*N + TSN*get_group_id(1) + row;
   Asub[col + l*RTSN][row] = A[indexA];  Bsub[row][col + l*RTSN] = B[indexB];
  }
  __syncthreads()
  for (int k=0; k<TSK; k++)
   for (int w=0; w<WPTN; w++)
    acc[w] += Asub[k][row] * Bsub[col + w*RTSN][k];
  __syncthreads()
 }
 for (int w=0; w<WPTN; w++) C[(globalCol + w*RTSN)*M + globalRow]
}
```

Soumyajit Dey, Assistant

So that is what we discussed that well will increase the rectangular tile size. And but then, there was also this optimization that since we are supporting bigger tiles, why not also store the matrix B in a transpose way, because then you have the advantage of load a memory access coalescing while loading from the global memory. But then we also figured out that well that leads to a problem with the shared stores. For that we needed to flip the indices here. So that the shared memory stores are done in a nice way, so that they can be fetched again properly.

(Refer Slide Time: 03:50)



Shared Memory Padding

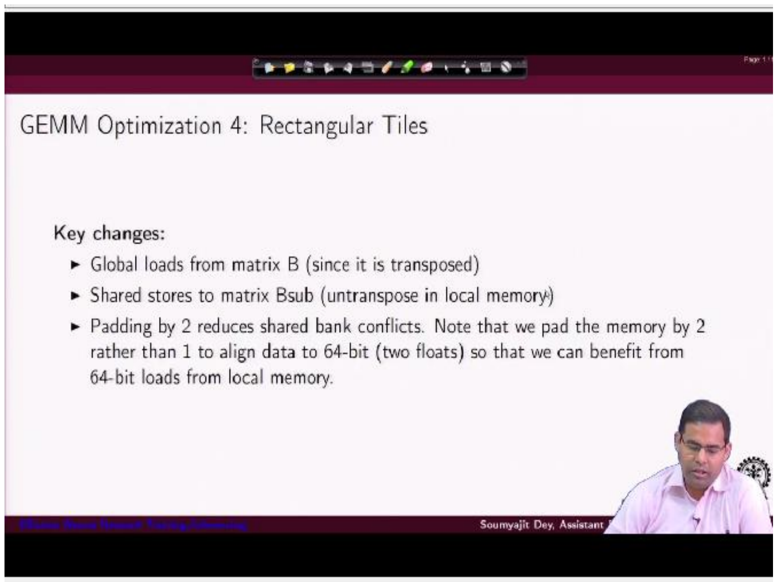
Bsub[row][col + w*RTSN]

row = threadIdx.x; col = threadIdx.y;

Soumyajit Dey, Assistant

But that also led to a problem that when I am now looking for loading from the shared memory, they are all falling in the same band. So, in order to reduce this band conflict, we added the data here, right. So, that is a short summary of how rectangular tiles really helping us, right.

(Refer Slide Time: 04:00)



GEMM Optimization 4: Rectangular Tiles

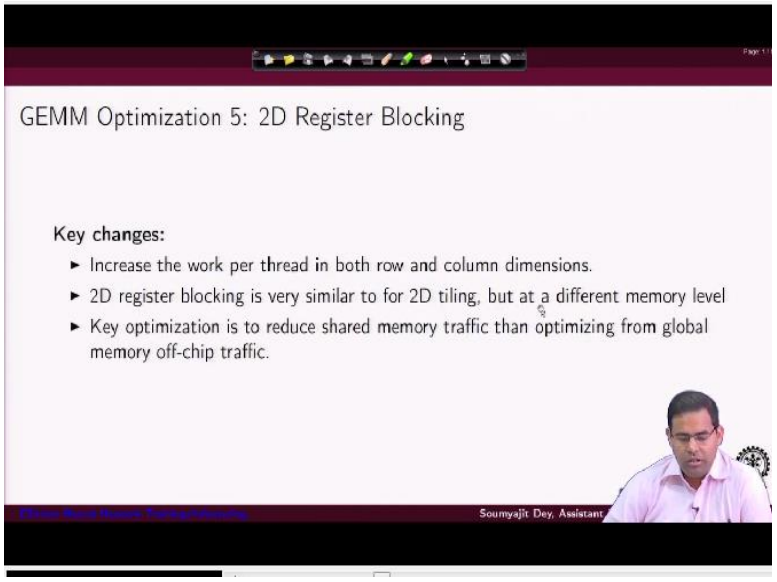
Key changes:

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

Soumyajit Dey, Assistant

So these are the changes like you do global loads from B where B is pre transposed. And then you do shared store through the matrix B sub in such a way that the load is optimized here. And the way you optimize the load is you paired the data types here with 2 elements, because the in each bank you have memory width of 64 bits for the loads. So you paired with 2 positions here. And that gives you a huge advantage over the baseline of implementations.

(Refer Slide Time: 04:38)



GEMM Optimization 5: 2D Register Blocking

Key changes:

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

Now, let us just start with the next optimization, which is what we call as 2D register blocking. So, just to remember that in case of rectangular tiles, we had to do I mean the transpose and then store properly in the shared memory, while doing the untransformed from the local memory right and then we had to paired the data here. So, the other 3 things which we needed to do simultaneously.

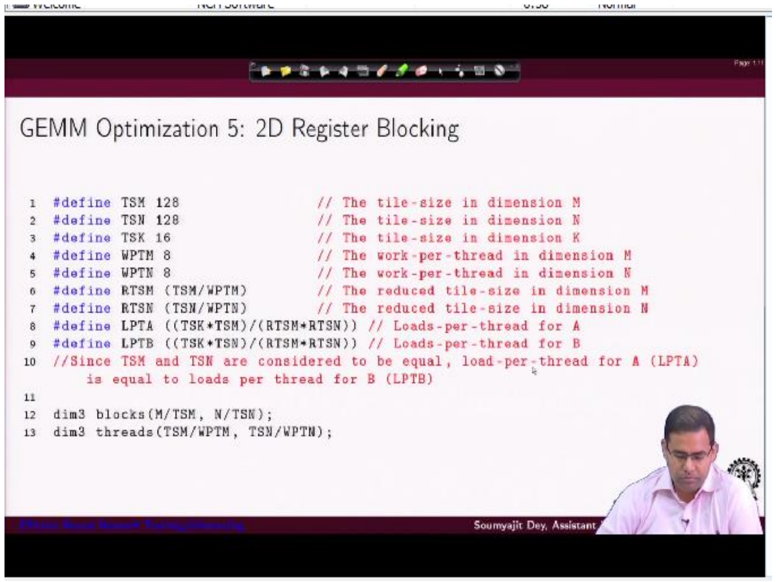
Now again coming to the register blocking part well we have earlier when discussing the optimizations where we are kind of doing the coarsening I would say in 1 dimension right because if you remember while we are doing threat coarsening, we are coarsening we are computing for multiple data points in 1 dimension, when we are supporting wider loads again we are doing a similar course ending in the other dimension, right.

So, the general idea could be that you increase the work per thread in both the row and column dimensions right. So, this is what we call as 2D register blocking. So, essentially we are doing the register loads in such a way that it is essentially similar to 2D tiling that we discussed for the shared memory based optimization for matrix multiplication. But this is done at a different level. So that means just let us just replicate the idea that you are doing a shared 2D I mean share 2D load of data from the global memory to the shared memory.

Let us just replicate the idea and load data in a similar way from the shared memory to the global memory right. So that is what we will be calling as 2D register blocking. So it is similar to 2D tiling, but it is done at a different memory level. In the earlier case, it was from the global memory to the shared memory, in this case is from the shared memory to the registers, right.

So, what really helps you the optimization is that it helps to reduce the shared memory traffic I mean, just like using the earlier optimization of normal standard tiling, what we are really doing where we are really optimizing the global memory off chip traffic? So in this case, you are using the same idea between shared memory and the register file to reduce the shared memory traffic.

(Refer Slide Time: 07:02)



GEMM Optimization 5: 2D Register Blocking

```
#define TSM 128                    // The tile-size in dimension M
#define TSN 128                    // The tile-size in dimension N
#define TSK 16                     // The tile-size in dimension K
#define WPTM 8                     // The work-per-thread in dimension M
#define WPTN 8                     // The work-per-thread in dimension N
#define RTSM (TSM/WPTM)            // The reduced tile-size in dimension M
#define RTSN (TSN/WPTN)            // The reduced tile-size in dimension N
#define LPTA ((TSK*TSM)/(RTSM*RTSN)) // Loads-per-thread for A
#define LPTB ((TSK*TSN)/(RTSM*RTSN)) // Loads-per-thread for B
//Since TSM and TSN are considered to be equal, load-per-thread for A (LPTA)
    is equal to loads per thread for B (LPTB)

dim3 blocks(M/TSM, N/TSN);
dim3 threads(TSM/WPTM, TSN/WPTN);
```

Soumyajit Dey, Assistant

So let us just have a look at how it works. So we will just use our usual definitions, the tile size in dimension M, remember that we are multiplying M cross K and K cross M matrices here, right tile size in dimension N and then tile size in dimension K, this is what we set. And then if you remember earlier, we are defining work per thread while doing coarsening we will just use the same idea in the in terms of work per thread in dimension M and work per thread in dimension N right.

So, if we are trying to coarsen the work in both these dimensions, then we will have a reduced tile size in both the dimensions right. So, these are tiles which will be based on the shared memory for loading data to the registers, right. So then the reduced tile size in dimension N would be just like the tile size in dimension M divided by the amount of work we are now giving to a single thread in dimension M, right.

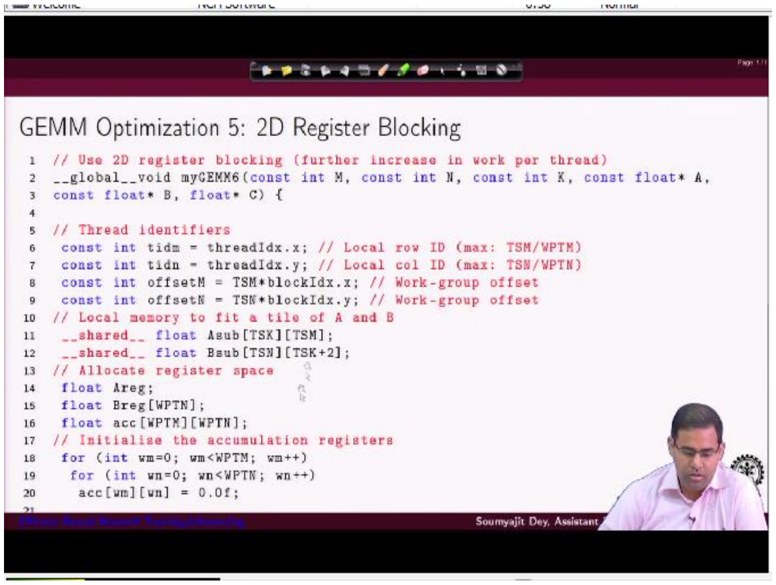
So it is just tile size in dimension M divided by the work per thread that we are now trying to define in the same dimension. So that gives me these RTSM. Similarly, I can have an RTSN, right. So that is essentially nothing but TSN by the work per thread that I have defined in that dimension M right. So, once this is done, then I can define that will what is the amount of data load that each thread has to do for A and B matrices.

In our case, in this simplistic case, we will consider for 2 of them that will be same here. So LPTA is nothing but the overall tile size for the M cross K matrix divided by the reduced tile size in the dimension M multiplied by the reduce tile size in the dimension N right. So that gives you the loads per thread for A. Similarly what happens to loads per thread for B, well you consider the original tile size.

Tile size in dimension K and tile size in dimension N for the matrix B and you divide it by the reduced tile size in the dimensions, right. So, essentially you are dividing the original tile size by the reduced tile size and that is giving you the amount of loads that a thread has to do for A and similarly, the amount of load a thread has to do for B right. So, since this TSM and TSN are considered to be equal in our case.

So, for us the loads per thread for the matrix A LPTA and LPTB the loads per thread for matrix B they are going to be same right. So, with this we can then go and define the blocks and the greed and the blocks right. So, they can be as just like this that so I am defining dividing M by TSM N by TSN. That would be my dim3 definition of blocks and similarly for threads right.

(Refer Slide Time: 09:59)



GEMM Optimization 5: 2D Register Blocking

```
// Use 2D register blocking (further increase in work per thread)
__global__void myGEMM6(const int M, const int N, const int K, const float* A,
const float* B, float* C) {

// Thread identifiers
 const int tidm = threadIdx.x; // Local row ID (max: TSM/WPTM)
 const int tidn = threadIdx.y; // Local col ID (max: TSN/WPTN)
 const int offsetM = TSM*blockIdx.x; // Work-group offset
 const int offsetN = TSN*blockIdx.y; // Work-group offset
// Local memory to fit a tile of A and B
 __shared__ float Asub[TSK][TSM];
 __shared__ float Bsub[TSN][TSK+2];
// Allocate register space
 float Areg;
 float Breg[WPTN];
 float acc[WPTM][WPTN];
// Initialise the accumulation registers
 for (int wm=0; wm<WPTM; wm++)
  for (int wn=0; wn<WPTN; wn++)
   acc[wm][wn] = 0.0f;
```

Soumyajit Dey, Assistant

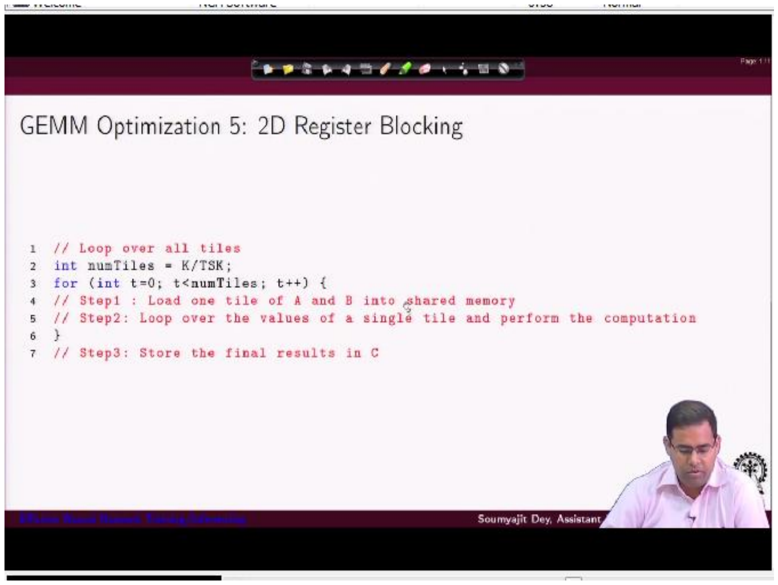
So, what will be my basic steps when I am going to execute this 2D register blocking. So, of course, one thing is to be sure that I am increasing the work per thread further by using this optimization for identifying the threads we use for each thread will initiate this following local variables. So let us say I define this tidm and tidn just for noting down the local threat IDs with respect to the columns and the rows right.

And of course, the maximum value each of them can have will be this TSM divided by WPTM and similarly the TSN divided by the WPTN like we have defined earlier okay. And similarly we can compute the offsets which will be used soon like exactly from which block the thread shall start working to figure that out you multiply this tile size in the dimension M with the block ID for the thread.

And similarly, the offset in the dimension N can be found by the tile size in dimension N multiplied by the block ID in the Y dimension, right. So, the next thing that comes is well, we have to define the memory to fit a tile for A as well as a tile for B, right. So, how do you really do it. Well, we already have this TSKs and TSMs defined. So that gives me the Asub as TSK TSM, and B sub the other memory as TSN, TSK + 2 if you remember our earlier idea of the padding that we have to introduce her, right.

So that is actually getting carried over here. So the next thing that of course, you have to do is well, you have to initialize a 2D accumulation register here. And earlier, it was a 1D array, but now since it is a 2D register blocking so you are initializing these acc in a 2D array with all 0s for the float values, right.

(Refer Slide Time: 12:02)



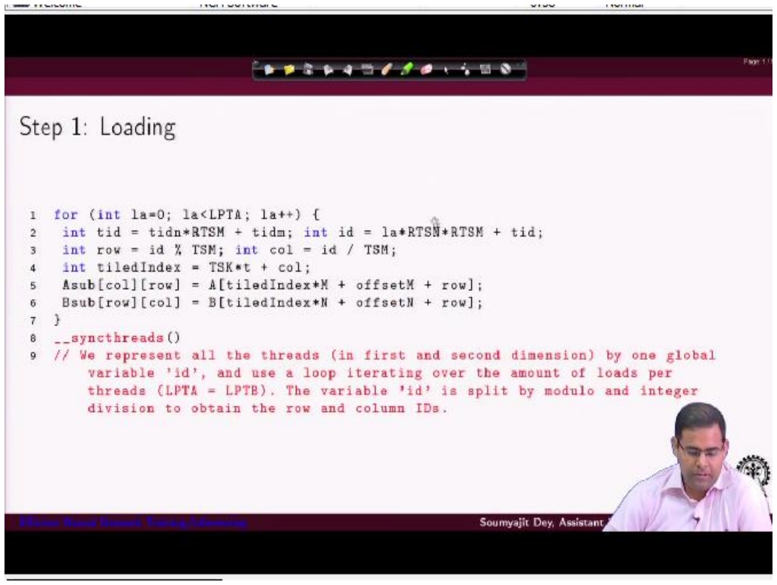
GEMM Optimization 5: 2D Register Blocking

```
// Loop over all tiles
int numTiles = K/TSK;
for (int t=0; t<numTiles; t++) {
// Step1 : Load one tile of A and B into shared memory
// Step2: Loop over the values of a single tile and perform the computation
}
// Step3: Store the final results in C
```

But then comes what is the overall operation that you have to do for per thread for every tile. So, this is your number of tiles right. So, overall the K and you divided by the TSK, that is your tile size in the K dimension. So, that is the total number of tiles you have in that K dimension over which you have to hop right. So, this is the tile loading loop, the outer loop of the multiplications right.

So, this gives you the number of tiles and inside this you have the same older steps to be done for this newer setting that you have to load 1 tile of A and 1 tile of B into the shared memory and then you have to loop over the values of a single tile and perform the computation just like earlier. Only thing is you have more work per thread right now, and at the end of the entire computation when all these load and compute for all the tiles is done you just write back the values result matrix C right.

(Refer Slide Time: 12:58)



Step 1: Loading

```
for (int la=0; la<LPTA; la++) {
 int tid = tidn*RTSM + tidm; int id = la*RTSN*RTSM + tid;
 int row = id % TSM; int col = id / TSM;
 int tiledIndex = TSK*t + col;
 Asub[col][row] = A[tiledIndex*M + offsetM + row];
 Bsub[row][col] = B[tiledIndex*N + offsetN + row];
}
__syncthreads()
// We represent all the threads (in first and second dimension) by one global
    variable 'id', and use a loop iterating over the amount of loads per
    threads (LPTA = LPTB). The variable 'id' is split by modulo and integer
    division to obtain the row and column IDs.
```

Soumyajit Dey, Assistant

So here comes the load part. So how do you really load. So, of course, one may start thinking that well, I have got this 2D data to load, but we will do it in our single loop here. The way we are doing it is, as you can see, for each position, I am just iterating from 0 to LPTA, right, the load per thread value. So this is the amount of data I am supposed to load and since LPTA is equal to LPTB.

So just by iterating, over this 2 I can do the loading for the both the Asub and Bsub matrices. And from where am I supposed to load. Well, we are representing all the threads in the first and second dimension by one global variable ID. So let us understand I mean, what is the problem here. So we have already defined the work per thread. But now, I mean, ideally want to start thinking that well, this would be a 2D loading.

So why do not I have a cascade of 2 loops for doing the loading, instead of that what we are doing is we also know that how many loads one thread is supposed to do, right. So let us just iterate one loop from 0 to that maximum number of loads that one thread is supposed to do, and figure out a global position for each trade, you figure out a global position in the array from where you are supposed to do the load.

So that essentially we are trying to figure out by computing an ID, right. And then, so that position, I can just figure out because I know the M, and what is the offset inside it, and then I can go to the corresponding row, right. So by computing these values, I can just figure out from where to load, right. And then the other thing I will do is, I will figure out the ID of the thread by using this local computation, you can just easily check that how this ID is getting computed.

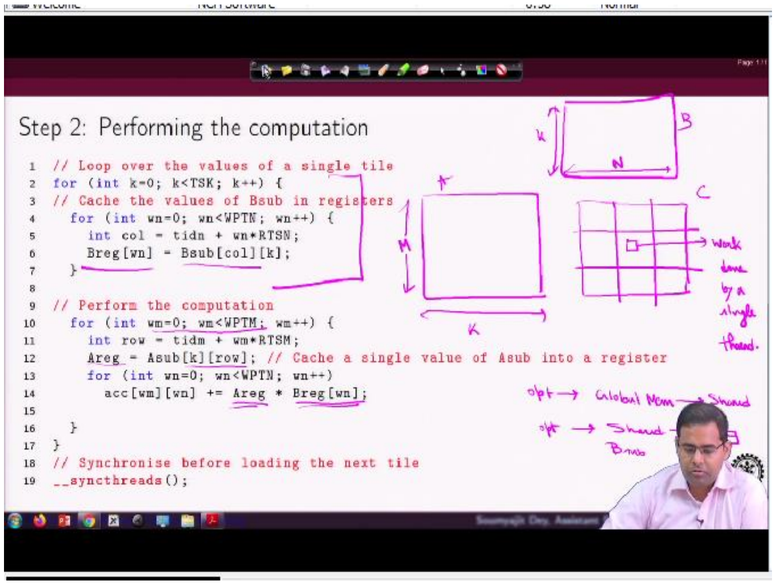
And if you just do a percentile and a divided operation, you can just find out what is row and column index in Asub and Bsub where you are supposed to put the value, right mind that you are Asub and Bsub have the opposite things because for B you are having the transpose matrix, right. So is just like the previous case here, we would say yeah, so, these are always as this rows, same thing.

So, in using this loop, you are just identifying here through this access expression from which location in the matrix A you are doing the load from which location in the matrix B you are doing the load. And by doing this computation of ID with this formula, you can just check it up, because you are just figuring out which number to load and then you are multiplying it by this reduced tile size to go to that corresponding location right.

And then you are just doing an offset with the tid which you are already computed here, right. So with this, you are able to go to the locations of both I mean of the A and B matrix both, right. So, again, we will just repeat that here, we compute the tile index, right. And then you are just multiplying the tile index with the M and N, because these are the corresponding dimensions to look for in the A and B matrix right, mind that B is transposed here.

So, these are the dimensions to look for in the A and B matrix. And then from A and B, you are doing our load to Asub and Bsub just like we have discussed earlier. So, the variable ID that we have which we have computed here, you are just then doing these 2 operations to figure out what is the row and column value where to load right.

(Refer Slide Time: 16:52)



Step 2: Performing the computation

```
// Loop over the values of a single tile
for (int k=0; k<TSK; k++) {
// Cache the values of Bsub in registers
  for (int wn=0; wn<WPTN; wn++) {
    int col = tidn + wn*RTSN;
    Breg[wn] = Bsub[col][k];
  }

// Perform the computation
  for (int wm=0; wm<WPTM; wm++) {
    int row = tidm + wm*RTSM;
    Areg = Asub[k][row]; // Cache a single value of Asub into a register
    for (int wn=0; wn<WPTN; wn++)
      acc[wm][wn] += Areg * Breg[wn];

  }
}
// Synchronise before loading the next tile
__syncthreads();
```

Once the loading is done, you are going to run this computational loop where you are actually going to perform the multiplications and for that, what you do is this is your outer loop, then for inside the outer loop, first notice that the outer loop is definitely going to run for K values right up to TSK, right. That is the tile size in K, right. And then, because again, the next time we load and again, we are going to run these for TSK.

And that is just like how you do computation with tiles as we all know. So then, the thing that you do is you cache the values of b sub in the registers. Now that is an important step. So observe what is going on. So, you have got this value of Bsub now you are going to load them in this array Breg. Now, why is this important. Observe that Bsub is shared, right. So it is in the shared memory, but where is Breg.

Breg is a local array, which means this will be located in the register, right. So you are now going to cache this value of Bsub that you have the consecutive values, right. So, you are just computing these column index right for the same K, you are just computing this column index and getting the value from Bsub and you are loading it into Breg right. So, in that way, with this loop, you are loading all the values of Bsub that you require for the multiplication in Breg.

Well, what is the next step. So, this actually caches all the required values for Bsub and then now you get to the computation. So, now, for the real multiplication computation, you have this for loop where you are running it for this wm 0 to this work per thread for M, right. And here, what you are going to do is well first you compute the row index from using this tidm. And this wm values and that gives you the exact value of from Asub, which is going to be multiplied.

Well, so, as you can see, your essentially caching a single value of Asub into the register and that is the value that is going to be multiplied with all the values of Breg continuously and stored in the accumulator right for the corresponding value. So, this we are not repeating because as you can understand this follows the similar pattern like our earlier optimizations right, you need all the consecutive B values, but you need this single A value right.

That is why you have already cached these B values, right. So, as you can see, this is all the work for one thread, right. So, the real multiplication by the thread is really happening here, right. And since you have 2 loops, so, in that way, you are getting a coarsen set of values done, right. So, if we just repeat inside C so, this is your C you have got these tiles defined for 1 tile. Well, these are the reduced tile sizes I am drawing for one thread, you are now making that thread do a 2D computation.

So, this loop is actually iterated over the number of computations that is to be done. Right work per thread in M. And then in one direction of course, right. Of course you have this A and B. So, this outer loop is going over the work per thread in direction M and the inner loop is going over the work per thread direction N but for a single position of this direction M you load 1 Areg value right and for that you are going to use these Breg values right to sequence of Breg values.

So, in every outer iteration what you are doing is you are caching a single value right and then in the inner iteration, you are going to use this Areg value with this array of Breg values, right. So, what is important to notice that will every thread has got parser activity, for every thread for each tile, you have got some parser activity. So if you see this is pastured activity for a thread any one time, right.

So for that single tile, what you are really doing is you are first figuring out well for this tile, what are the values I am supposed to load and you are. So, this is where the threads loading of values from the shared memory to the registers are getting done, right. So, that is why we have it like cache the values of Bsub in registers right. So, here, inside this compute loop, there is some amount of memory activity going on.

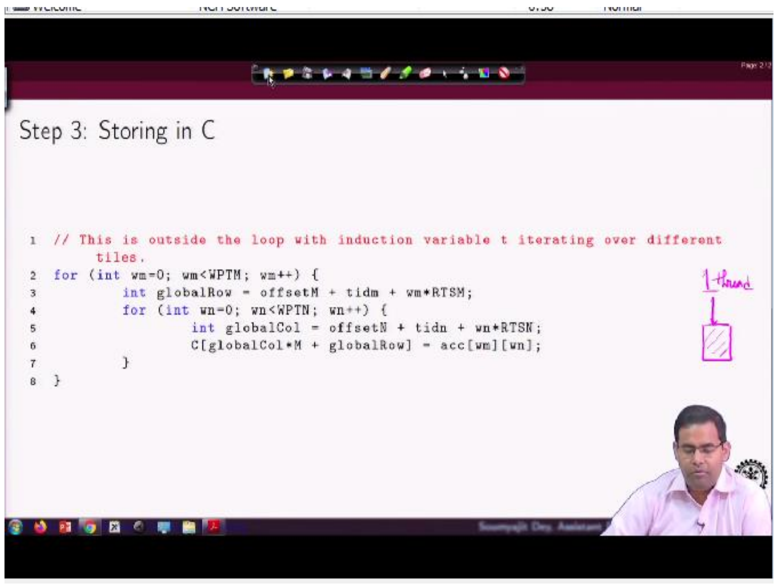
The memory activity to be more specific is for loading values from the shared memory to the registers. And this is what we would say is the cracks of the optimization. Like earlier, we have already seen this optimization right which is like loading from global mem to share. And next, the current optimization is when you load from shared to a 2D register right. So, that is what you do. So, as you can see inside this loop for one value of K you are first making a sequence of loads in from this Bsub right.

So, you are making a sequence of loads from these Bsub to here, right, and then so that is the activity for this thread. And similarly, for the other threads also, they are all making their own corresponding caching of values of Bsub for this loop, right. And for this thread, after it has got these values cached, it goes to perform the real computation, right. So where what it does is inside this for loop, well, first it performs the computation of the row index.

Then it uses the row index to compute well, what value is to be loaded into Areg for the correspondent Asub shared memory, right. For that it has got this index, right, which row and K is getting computed. And then here, it is keeping it constant, and multiplying it continuously with these Breg values, right. So it is multiplying this Areg continuously with Breg values and it is iterated here over this.

Well once this is done again it will load from the outer loop, it will again, load another corresponding value of from Asub to Areg. And then again in the inner loop, it will multiply this Areg value with a sequence of values from Breg and that is how it will work.

(Refer Slide Time: 25:22)



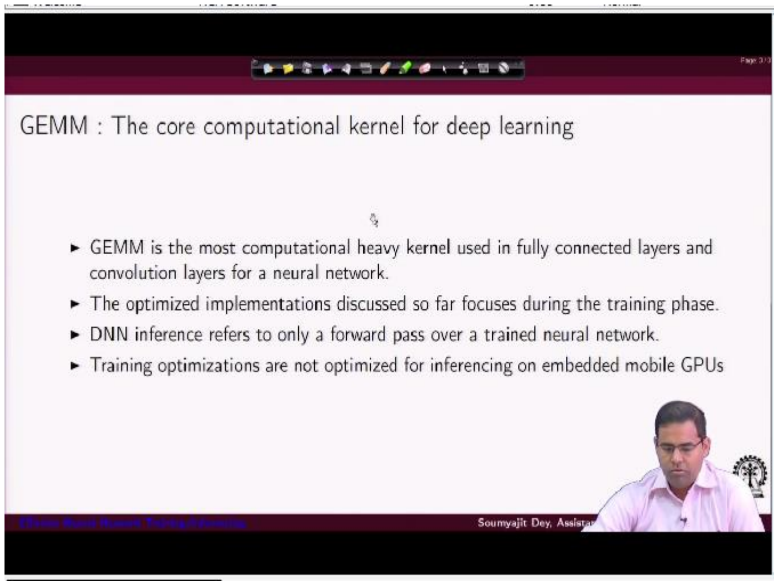
Step 3: Storing in C

```
// This is outside the loop with induction variable t iterating over different
     tiles.
for (int wm=0; wm<WPTM; wm++) {
        int globalRow = offsetM + tidm + wm*RTSM;
        for (int wn=0; wn<WPTN; wn++) {
                int globalCol = offsetN + tidn + wn*RTSN;
                C[globalCol*M + globalRow] = acc[wm][wn];
        }
}
```

So, with this we have the computation done in 2D. And once this is done, the next thing is loading it back to a global memory. So for that, well, first you have to figure out what is a location where it has to be loaded. And again, as you can see, since this will be this is a 2D activity, like this thread is going to again, load back a 2D array of values right for 1 thread. So what it does, again, you have a cascade of 2 loops.

And it figures out that what is the global position where it has to do the load. So that is figured out by fast computing a global row from the outer loop. And then in each iteration of the inner loop, it figures out a global column value. And this combination of global column and global row is used to figure out a position in the C matrix where this value from acc has to be written back, right. So this is how the final values get stored in C.

(Refer Slide Time: 26:31)



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.
- 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

So this is our idea of how these different possible optimizations can be performed for doing the GEMM computation. So just to recall, all we do here is that we are using our earlier optimizations together, but doing it in a 2 dimensional way, by coarsening the threats in both dimensions. Well here we have not used the wider load idea, there is also something you need to understand, right.

But the important thing is for you to figure out this, that how the computes work like as I am giving a 2D amount of computation to be done for each thread. That is why now I have a cascade of loops here for doing all those computations and also the loading of the data. And as I recall that there are 2 levels of loading. Here you have a load going on from the global to the shared memory. And then in your earlier compute loop, you now have an amount of load first done from the share to the registers.

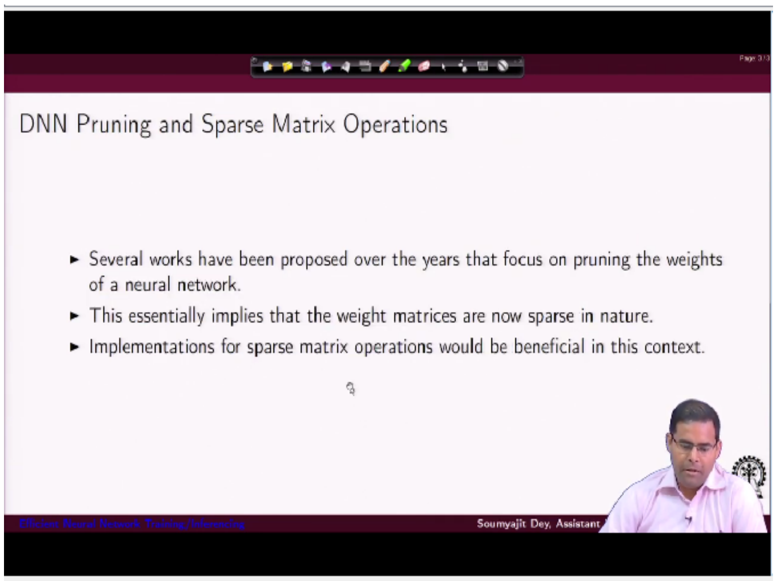
And these 2 steps we are now kind of differentiating here, and then I am using the cached values to multiply with I mean I am caching a single value of Areg and multiplying it with a sequence of values of Breg, and so on so forth through this loop cascade, right. And then again, I am storing it back using another loop cascade. So right now I have more work per thread. And these are good optimization in the sense that in modern GPUs you have big I mean sufficiently big shared memories and so, if you can just choose a proper tile size and a suitable thread coarsening factor in the 2 dimension.

Then this can really help okay, so the summary here would be that GEMM is the most computationally heavy kernel which is used for fully connected layers, and also the convolution layers of neural network and the optimized implementations that we have discussed so far. They focus on that training phase of course, and if we are speaking about the other phase of neural network work like inferencing.

So that refers only to a forward pass over a trained neural network. And so in general, these optimizations we have discussed will likely be more important for HPC kind of process where you because you are primarily going to do large neural network training in such deep as such kind of large, I mean significantly powerful computers, whereas inferencing is the more popular workload for embedded GPUs.

Because in an embedded GPU, you can have an inferencing engine for doing a lot of stuff, like recognizing somebody's image from a camera, or doing some object detection, or maybe live streaming of video and from that doing a face detection and all that those are inferencing kind of works. So they also require different kinds of optimizations, which you can figure out.

(Refer Slide Time: 29:41)



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.

So, with this maybe we will like to close this current lecture. Thank you.