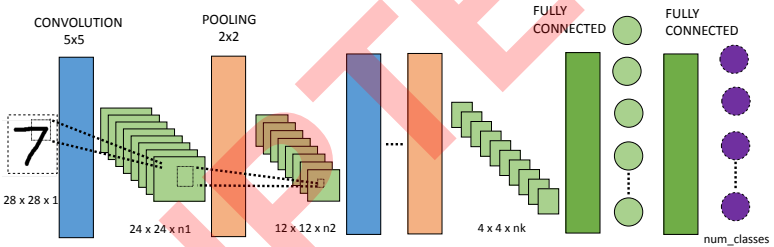
End to End CNN



Recap: Backward Propagation

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

Recap: Backward Propagation

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

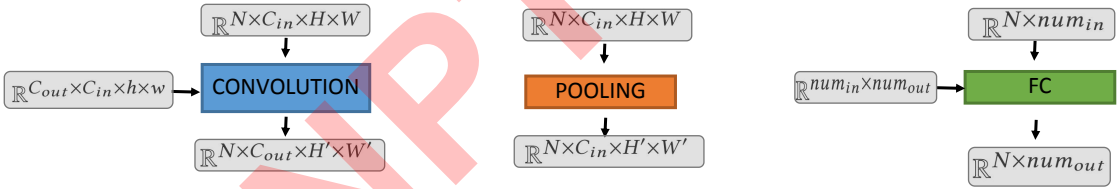
Recap: Backward Propagation

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

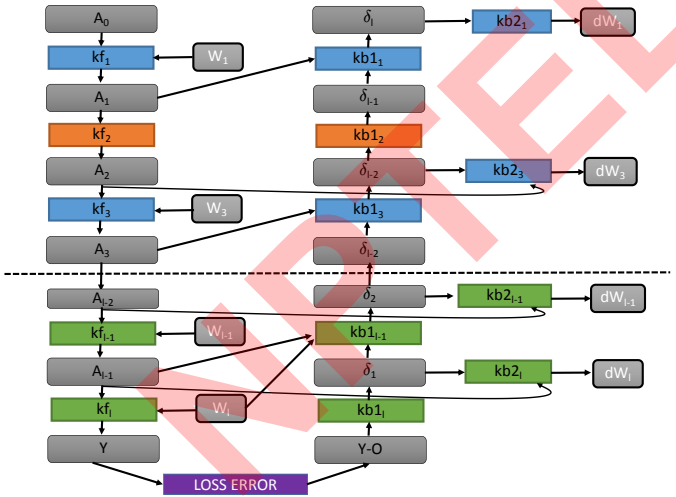
Let us consider a general scenario with l layers with each layer i having weight matrix W_i.

CNN Layers

N: Number of images/input examples



CNN Training (Layer View)



A_i : Layer i's output

W_i : Layer i's weight

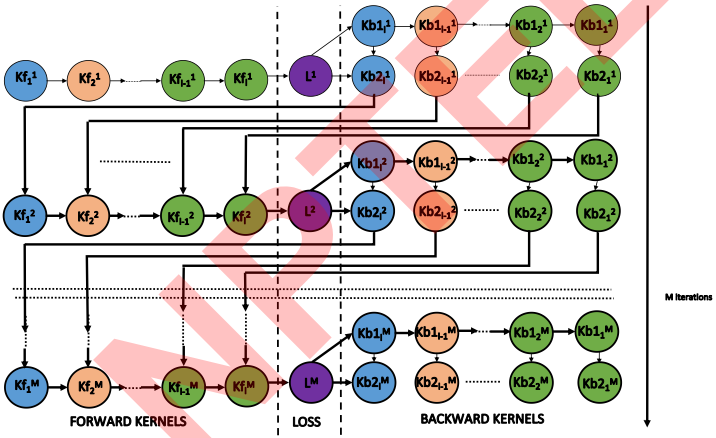
dW_i : $\frac{\partial J}{\partial W_i}$

kf_i computes A_i

$kb1_i$ computes δ_{i-i-1}

$kb2_i$ computes dW_i

CNN Training (Task Graph View)



GEMM

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

GEMM

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

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

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

GEMM Optimization 1: Tiling

```
1  __global__ void GEMM1(const int M, const int N, const int K,
2  const  float* A, const  float* B, float* C) {
3          const int row = threadIdx.x; // Local row ID (max: TS)
4          const int col = threadIdx.y; // Local col ID (max: TS)
5          const int globalRow = TS*blockIdx.x + row; // Row ID of C (0..M)
6          const int globalCol = TS*blockIdx.y + col; // Col ID of C (0..N)
7          __shared__ float Asub[TS][TS];__shared__ float Bsub[TS][TS];
8          float acc = 0.0f; const int numTiles = K/TS;
9          for (int t=0; t<numTiles; t++) {
10          const int tiledRow = TS*t + row;  const int tiledCol = TS*t + col;
11          Asub[col][row] = A[tiledCol*M + globalRow];
12          Bsub[col][row] = B[globalCol*K + tiledRow];
13          __syncthreads()
14          for (int k=0; k<TS; k++)
15           acc += Asub[k][row] * Bsub[col][k];
16          __syncthreads()
17         }
18         C[globalCol*M + globalRow] = acc; // Store the final result in C
19 } // Launch Parameters: <<<(M/TS,N/TS),(TS,TS)>>>
```


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

GEMM Optimization 2: Coarsening

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


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).

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.

GEMM Optimization 3: Wider Data Types

```
1  __global__ void GEMM3(const int M, const int N, const int K,
2  const float8* A, const float8* B, float8* C) {
3  //Code for thread identifiers
4  // Modify shared memory initialization
5  __shared__ float8 Asub[TS][TS/WIDTH];
6  __shared__ float8 Bsub[TS][TS/WIDTH];
7
8  float8 acc = { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f };
9  const int numTiles = K/TS;
10 for (int tile=0; tile<numTiles; tile++) {
11
12   const int tiledRow = (TS/WIDTH)*tile + row;
13   const int tiledCol = TS*tile + col;
14   Asub[col][row] = A[tiledCol*(M/WIDTH) + globalRow];
15   Bsub[col][row] = B[globalCol*(K/WIDTH) + tiledRow];
16   __syncthreads()
```


GEMM Optimization 3: Wider Data Types

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


GEMM Optimization 4: Rectangular Tiles

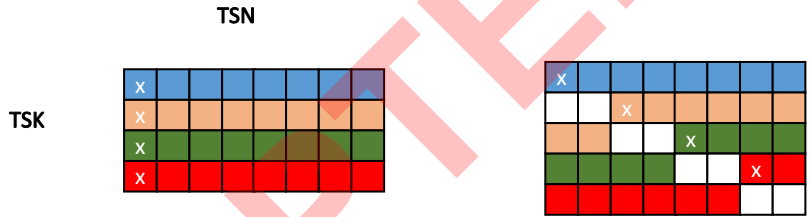
- The Tesla K40 GPU which has 48KB of shared memory per SM, on which multiple thread blocks can execute.
- For a 32×32 tile, we consume $2 * 32 * 32 * 4 = 8KB$ per work-group, so there is some headroom left.
- Since both matrix A and B share the dimension K, we y create rectangular tiles.
- We can also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


GEMM Optimization 4: Rectangular Tiles

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


Shared Memory Padding



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.

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.

GEMM Optimization 5: 2D Register Blocking

```
1  #define TSM 128                  // The tile-size in dimension M
2  #define TSN 128                  // The tile-size in dimension N
3  #define TSK 16                   // The tile-size in dimension K
4  #define WPTM 8                   // The work-per-thread in dimension M
5  #define WPTN 8                   // The work-per-thread in dimension N
6  #define RTSM (TSM/WPTM)          // The reduced tile-size in dimension M
7  #define RTSN (TSN/WPTN)          // The reduced tile-size in dimension N
8  #define LPTA ((TSK*TSM)/(RTSM*RTSN)) // Loads-per-thread for A
9  #define LPTB ((TSK*TSN)/(RTSM*RTSN)) // Loads-per-thread for B
10 //Since TSM and TSN are considered to be equal, load-per-thread for A (LPTA)
       is equal to loads per thread for B (LPTB)
11
12 dim3 blocks(M/TSM, N/TSN);
13 dim3 threads(TSM/WPTM, TSN/WPTN);
```


GEMM Optimization 5: 2D Register Blocking

```
1  // Use 2D register blocking (further increase in work per thread)
2  __global__void myGEMM6(const int M, const int N, const int K, const float* A,
3  const float* B, float* C) {
4
5  // Thread identifiers
6   const int tidm = threadIdx.x; // Local row ID (max: TSM/WPTM)
7   const int tidn = threadIdx.y; // Local col ID (max: TSN/WPTN)
8   const int offsetM = TSM*blockIdx.x; // Work-group offset
9   const int offsetN = TSN*blockIdx.y; // Work-group offset
10 // Local memory to fit a tile of A and B
11  __shared__ float Asub[TSK][TSM];
12  __shared__ float Bsub[TSN][TSK+2];
13 // Allocate register space
14  float Areg;
15  float Breg[WPTN];
16  float acc[WPTM][WPTN];
17 // Initialise the accumulation registers
18  for (int wm=0; wm<WPTM; wm++)
19   for (int wn=0; wn<WPTN; wn++)
20    acc[wm][wn] = 0.0f;
21
```


GEMM Optimization 5: 2D Register Blocking

```
1  // Loop over all tiles
2  int numTiles = K/TSK;
3  for (int t=0; t<numTiles; t++) {
4  // Step1 : Load one tile of A and B into shared memory
5  // Step2: Loop over the values of a single tile and perform the computation
6  }
7  // Step3: Store the final results in C
```


Step 1: Loading

```
1  for (int la=0; la<LPTA; la++) {
2   int tid = tidn*RTSM + tidm; int id = la*RTSN*RTSM + tid;
3   int row = id % TSM; int col = id / TSM;
4   int tiledIndex = TSK*t + col;
5   Asub[col][row] = A[tiledIndex*M + offsetM + row];
6   Bsub[row][col] = B[tiledIndex*N + offsetN + row];
7  }
8  __syncthreads()
9  // 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.
```


Step 2: Performing the computation

```
1  // Loop over the values of a single tile
2  for (int k=0; k<TSK; k++) {
3  // Cache the values of Bsub in registers
4    for (int wn=0; wn<WPTN; wn++) {
5      int col = tidn + wn*RTSN;
6      Breg[wn] = Bsub[col][k];
7    }
8
9  // Perform the computation
10   for (int wm=0; wm<WPTM; wm++) {
11     int row = tidm + wm*RTSM;
12     Areg = Asub[k][row]; // Cache a single value of Asub into a register
13     for (int wn=0; wn<WPTN; wn++)
14       acc[wm][wn] += Areg * Breg[wn];
15
16   }
17 }
18 // Synchronise before loading the next tile
19 __syncthreads();
```


Step 3: Storing in C

```
1  // This is outside the loop with induction variable t iterating over different
       tiles.
2  for (int wm=0; wm<WPTM; wm++) {
3          int globalRow = offsetM + tidm + wm*RTSM;
4          for (int wn=0; wn<WPTN; wn++) {
5                  int globalCol = offsetN + tidn + wn*RTSN;
6                  C[globalCol*M + globalRow] = acc[wm][wn];
7          }
8  }
```


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

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.

Sparse **M**atrix **V**ector 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.

Sparse **M**atrix **V**ector Multiplication: **SpMV**

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

CSR Format

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 uptil 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] + # non-zero elements in the $(i-1)^{th}$ row.

CSR Format Example

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

ptr = [0 2 4 7 8]
cols=[0 1 1 3 2 3 4 5]
vals=[10 20 30 40 50 60 70 80]

SpMV: CPU Implementation

```
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 }
```


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

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>>>
```


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

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();
```


SpMV CUDA Vector Implementation

```
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 }
```


Teaching Assistants

Anirban Ghose
PhD scholar in Department of Computer Science and Engineering
IIT Kharagpur
Intelligent Scheduling and Compiler Optimization Techniques for Heterogeneous Architectures

Research interests:

Srijeeta Maity
PhD scholar in Department of Computer Science and Engineering
IIT Kharagpur
Real time scheduling on heterogeneous embedded architectures

Research interests:

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

SpMV Adaptive CSR Implementation

- The implementation has a separate CPU function that divides the matrix into row-blocks.
- Depending on the number of non-zero elements in each row and a fixed block size, multiple rows constitute a row-block.

Reference: Greathouse J.L., Daga M. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format

Row Block Partitioning

```
1  int spmv_csr_adaptive_rowblocks(int *ptr,int totalRows,int *rowBlocks)
       {
2  rowBlocks[0] = 0; int sum = 0;  int last_i = 0; int ctr = 1;
3  for (int i = 1; i < totalRows; i++) {
4  sum += ptr[i] - ptr[i-1]; // Count non-zeroes in this row
5  if (sum == BlockDim){    // This row fills up LOCAL_SIZE
6  last_i = i; rowBlocks[ctr++] = i; sum = 0;
7  }
8  else if (sum > BlockDim){
9  if (i - last_i > 1) { // This extra row will not fit
10 rowBlocks[ctr++] = i - 1; i--;
11 }
12 else if (i - last_i == 1) // This one row is too large
13 rowBlocks[ctr++] = i; last_i = i; sum = 0;
14 }
15 }
16 rowBlocks[ctr++] = totalRows;
17 return ctr;
18 }
19 int countRowBlocks = spmv_csr_adaptive_rowblocks(ptr,M,rowBlocks);
20 spmv_csr_adaptive_kernel<T><<<(countRowBlocks-1), BlockDim>>>
```


SpMV Adaptive CSR Implementation

- The number of blocks launched is equal to the number of row-blocks.
- The size of shared memory used is equal to the block dimension used.
- Each thread in thread block loads, multiplies each element of the input sparse matrix and input dense vector and stores in shared memory.
- The products for each row are summed up by one thread per row.

SpMV

```
1   template <typename T>
2   __global__ void spmv_csr_adaptive_kernel(T * d_val,T * d_vector,int *
        d_cols,int * d_ptr,int N, int * d_rowBlocks, T * d_out) {
3   int startRow = d_rowBlocks[blockIdx.x];
4   int nextStartRow = d_rowBlocks[blockIdx.x + 1];
5   int num_rows = nextStartRow - startRow; int i = threadIdx.x;
6   __shared__ volatile T LDS[BlockDim];
7   if (num_rows > 1) {
8   int nnz = d_ptr[nextStartRow] - d_ptr[startRow]; int first_col = d_ptr
        [startRow];
9   if (i < nnz)
10  LDS[i] = d_val[first_col + i] * d_vector[d_cols[first_col + i]];
11  __syncthreads();
12  // Threads that fall within a range sum up the partial results
13  for (int k = startRow + i; k < nextStartRow; k += blockDim.x) {
14  T temp = 0;
15  for (int j= (d_ptr[k] - first_col); j < (d_ptr[k + 1] - first_col); j
        ++)
16  temp = temp + LDS[j];
17  d_out[k] = temp;
18  }
```


SpMV: Adaptive CSR Implementation

```
1           // If the block consists of only one row then run CSR Vector
2           else {
3           // Thread ID in warp
4           int rowStart = d_ptr[startRow]; int rowEnd = d_ptr[nextStartRow];
5           T sum = 0;
6           // Use all threads in a warp to accumulate multiplied elements
7           for (int j = rowStart + i; j < rowEnd; j += BlockDim) {
8           int col = d_cols[j];
9           sum += d_val[j] * d_vector[col];
10          }
11          LDS[i] = sum;
12          __syncthreads();
13          // Reduce partial sums
14          for (int stride = blockDim.x >> 1; stride > 0; stride >>= 1) {
15          __syncthreads();
16          if (i < stride)
17          LDS[i] += LDS[i + stride];
18          }
19          if (i == 0)  d_out[startRow] = LDS[i];
20          }
21          }
```