Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference

Course Project of Group 6: Fred Morsy, Jingyi Yang, Zhongrun Xiang

Last Time: Pruning Deep Learning Models

-Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference

-Paper discussed computations that arise in neural network pruning and attempts to handle these computations on GPUs

-In a DNN pruned via unstructured pruning, the core operation at inference time is SpMM

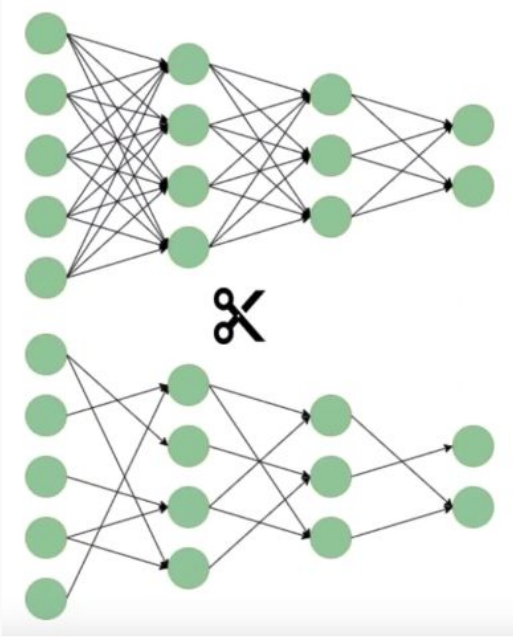
Pruning Deep Learning Models

-Downside of Deep Learning models is overfitting data

-Solution: Prune model by only including useful weights and preserve classification accuracy

-Setting useless weights to zero and only including useful weights

-After this we are left with a sparse weight matrix



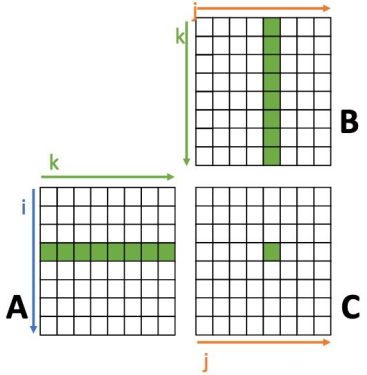
Pruning via SpMM

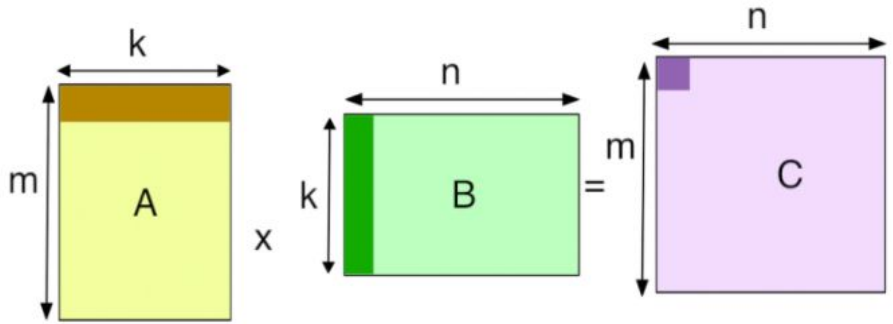
-Main operation in many pruning methods for Deep Neural Networks is Sparse Matrix Matrix Multiplication

-SpMM: Operation in which a sparse matrix in multiplied with a dense matrix

- Matrix multiplication

```
float A[N][N], B[N][N], C[N][N];

for (int i=0; i<N; i++)
   for (int j=0; j<N; j++)
      for (int k=0; k<N; k++)
         C[i][j] += A[i][k] * B[k][j];
```





SpMM Issues

-Current SpMM solutions are catered to scientific computing applications

-This is where the dimensions of the sparse matrix are massive compared to the dense matrix dimension

-Only achieve significant speed up when the sparsity ratio is very high (99.0%-99.9%)

SpMM Issues

-However the dimensions of the dense matrix may be larger than the sparse matrix dimension for SpMM problems in deep learning

-Existing implementations often fail to bring speedups in these cases

Goal: Optimize SpMM for Deep Learning

-Want to improve SpMM implementation for Deep Learning using optimization techniques mentioned in class

-In particular we would like to think about: Modeling and improving the movement of data

-Implement algorithms and try to improve performance

Potential Solutions (Methods)

-Blocking/tiling: Split Matrix into blocks and test various tile sizes

-Loop interchange: Test all valid loop permutations and pick the most efficient one

-Classical Usage: Both techniques are used to reduce memory access latency or the cache bandwidth necessary due to cache reuse for some common linear algebra algorithms

-Take advantage of data locality

Potential Solutions (Methods)

-Since most entries in Sparse matrix are 0s we are able to reduce computation and storage

-Data Formats: Try various data formats such as CSR, COO, etc. in order to reduce memory overhead

-Multithreading

-Load Balancing

-Shared Memory on GPU

Potential Solutions (Platforms)

SpMM on multiple platforms:

-Single threaded CPU (sequential implementation)

-Multi-threaded CPU (parallel implementation using OpenMP or pthread API)

-GPU Implementation (CUDA Programming)

Performance Analysis

-Baseline will be normal Matrix-Matrix multiplication in C++

-Will use the real data and apply techniques learned in class to achieve speedups

-Performance Metric: Improve running time

SpMM Assumptions for Project

-Dimensions of Dense Matrix are much larger than Sparse Matrix

-SpMM here is: C = AxB where matrix A has dimensions MxK, matrix B has dimensions KxN and matrix C has dimensions MxN

-Sparse matrices are unstructured

Datasets

-The sparse Transformer layers in real deep learning projects from paper: “Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science”

-We can load the data for the matrices A and B for SpMM from the Transformer model layers

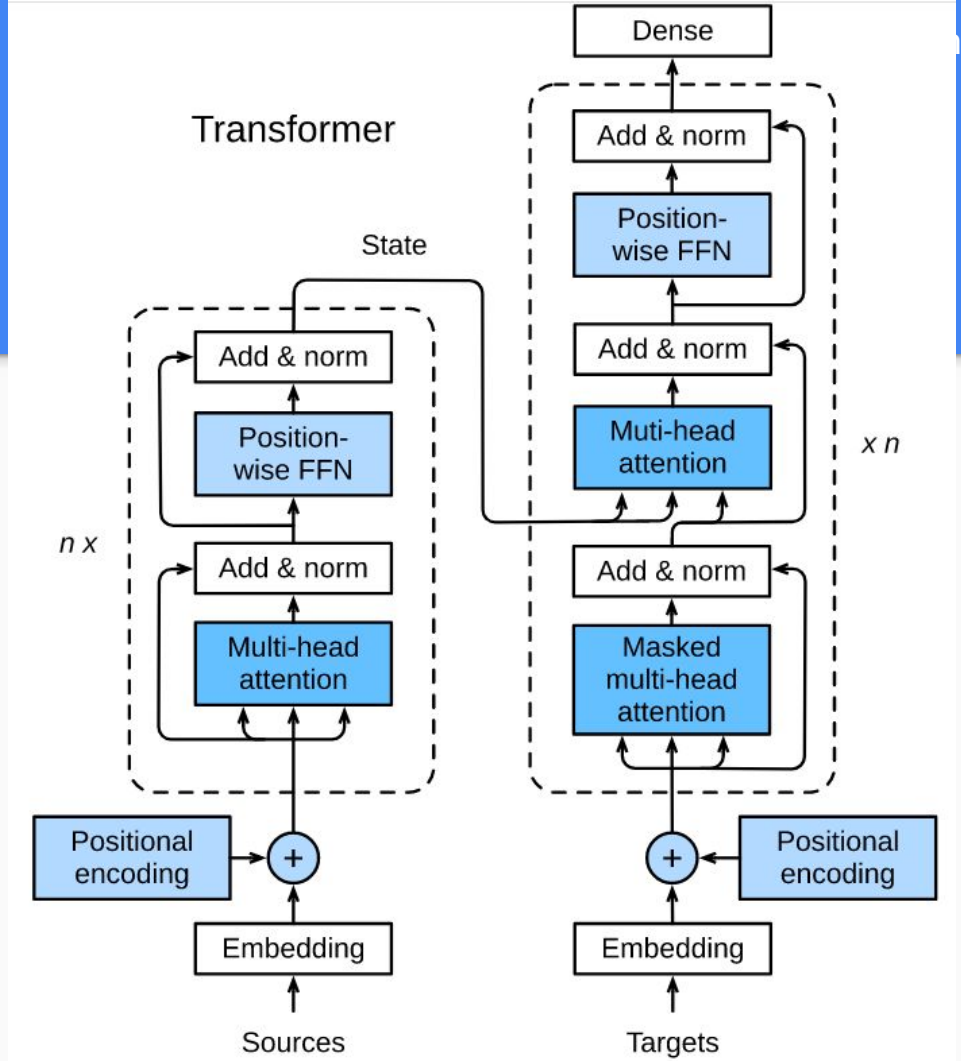
-We can load matrices with different sparsity levels

Transformer Model

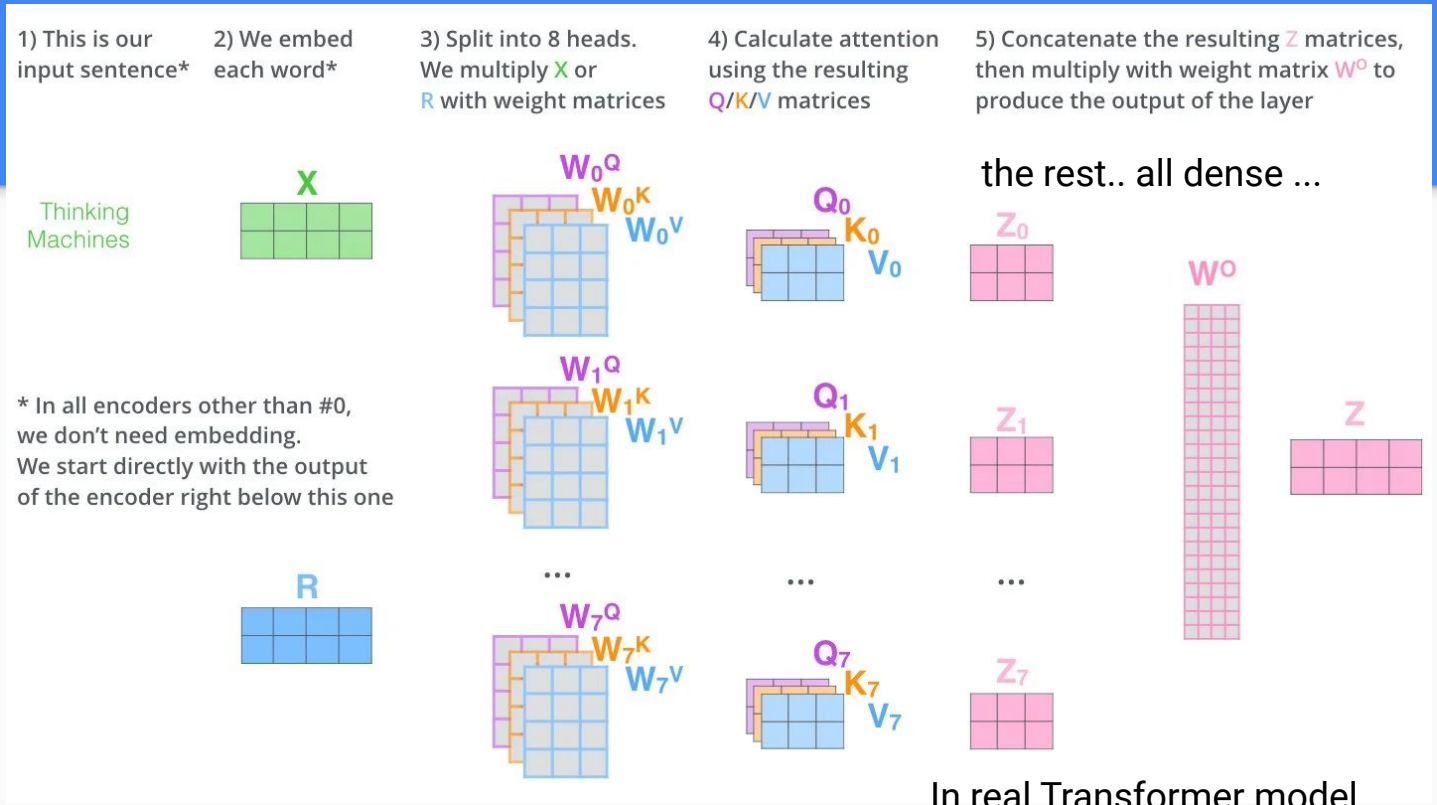
-Transformer is the most popular NLP model.

-The Multi-head attention calculation is the main part. It takes NxN attention matrix for every attention head, which can consume large amounts of memory.

https://github.com/jgcbrouns/Introducing-Sparsity-in-the-Transformer



Multi-head Attention



Dense A n*dim x Dense/Sparse B dim*dim = Dense C n*dim

In real Transformer model, dim=512,
n=maximum number of words in sentences

Dataset

-WMT 2016 English-to-German dataset (en2de), 29,000 samples in training set, 1014 samples in test set

-The longest sentence in the test set contains 33 words. Plus the ending code, the length is 34. After the word and position embedding, shape is 34*512.

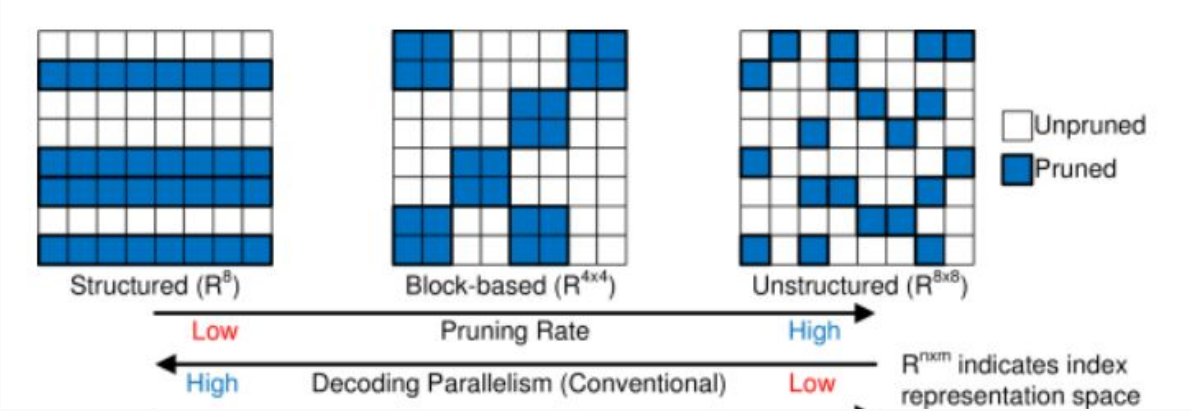
-The Sparse Transformer make the weight matrix (B) into a certain level of sparsity in both training and inference.

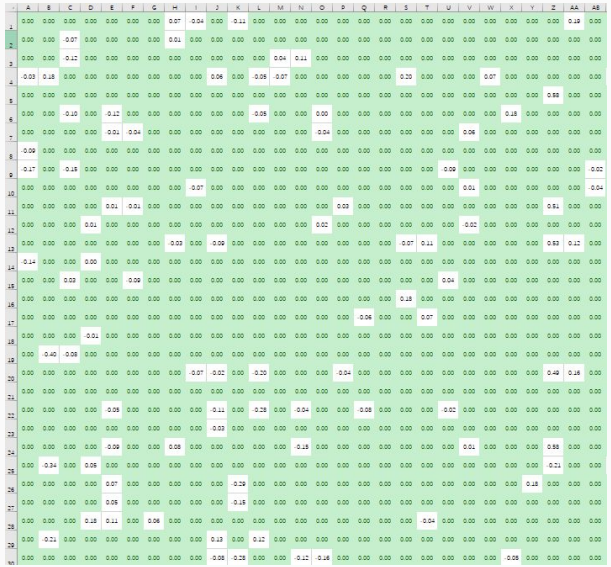
-In our projet, we tested our accelerating algorithms on 1014 samples in test set.

Dense A		Sparse B		Dense C
(1014 *34)*512	x	512*512	=	1024*34*512

Sparse Matrix B

-The shape of Matrix B is unstructured sparse matrix. We can use all the approaches learned in class and papers.





Dense Matrix A

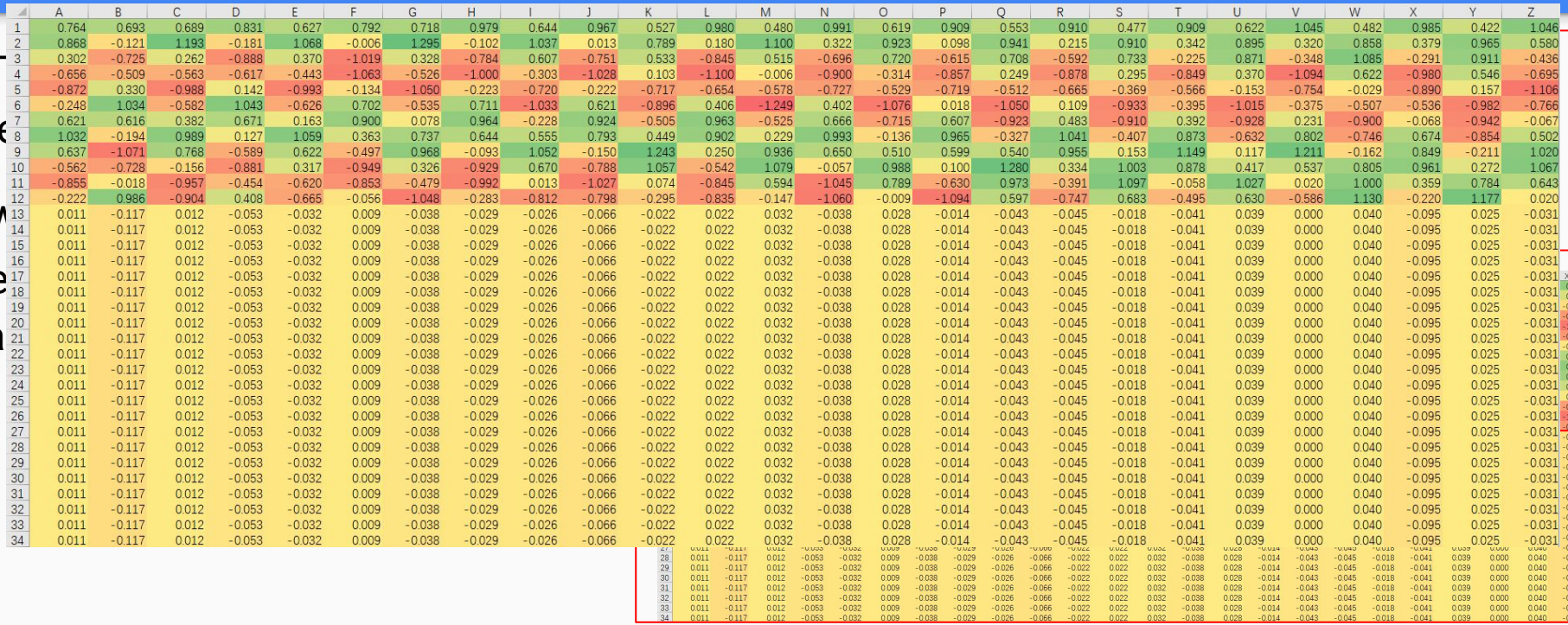
-T

de

-W

re

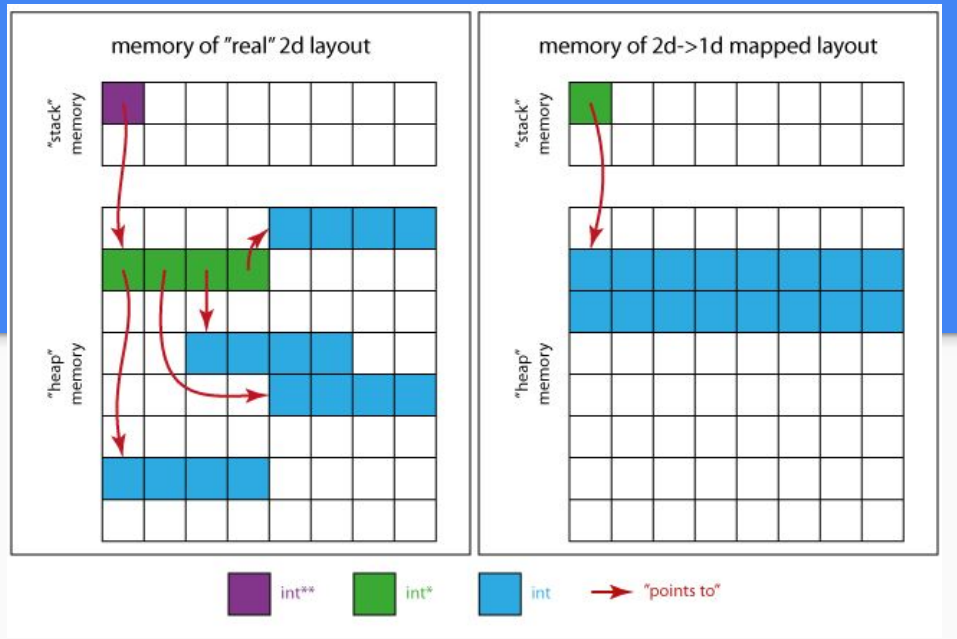
la



Baseline

- 2D array for A, B, C
- 1D array for A, B, C

```
for(i = 0; i < M_A; ++i)
{
    for(j = 0; j < N_B; ++j)
    {
        for(k = 0; k < K_A1; ++k)
        {
            V_C[i][j] += V_A[i][k] * V_B[k][j];
        }
    }
}
```



Loop Interchange

	A	B	C
i	N	N	N
j	N	N	N/B
k	N/B	N	1
	N^3/B	N^3	N^2/B

```
for(i = 0; i < M_A; ++i)
{
    for(j = 0; j < N_B; ++j)
    {
        for(k = 0; k < K_A1; ++k)
        {
            V_C[i][j] += V_A[i][k] * V_B[k][j];
        }
    }
}
```

Details on Data Reformat

- CSR : A (Sparse)
- Row pointers : get rid of all zeros in the matrix
- 2D array/ 1D array: B and C

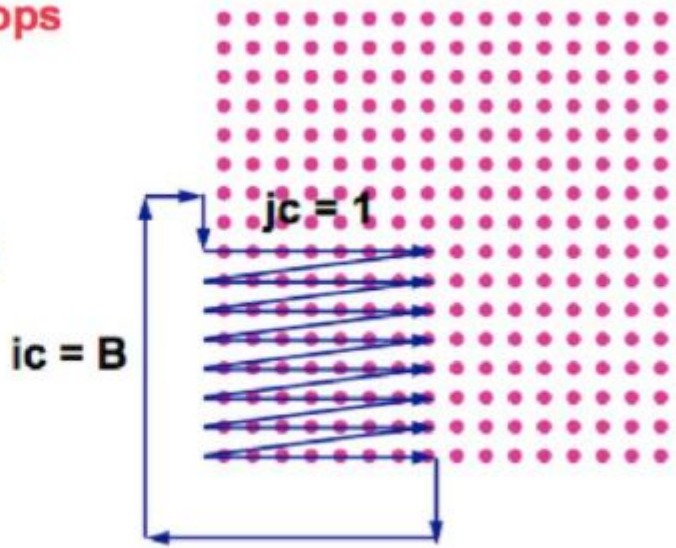
```
int* V_A = new int[L_A];
int* ROW_INDEX_A = new int[L_A];
int* COL_INDEX_A = new int[L_A];
for (int i = 0; i < L_A; i++) {
        int row_A, col_A, value_A;
        fin >> row_A >> col_A >> value_A;
        ROW_INDEX_A[i] = row_A;
        COL_INDEX_A[i] = col_A;
        V_A[i] = value_A;
}
```

```
for (int i = 0; i < M_A; i++) {
        for (int k = rowptr_A[i]; k < rowptr_A[i + 1]; k++) {
                int column_A = COL_INDEX_A[k];
                int val_A = V_A[k];
                V_C[i][column_A] += val_A * V_B[column_A][i];

        }
}
```

Details about Tiling / Blocks

```
do ic = 1, n, B
 do jc = 1, n, B
  do t = 1,T
   do i = ic, min(n,ic+B-1), 1
    do j = jc, min(n, jc+B-1), 1
     … a(i,j) …
    end do
   end do
  end do
 end do
end do
```

control loops



B is the block size.

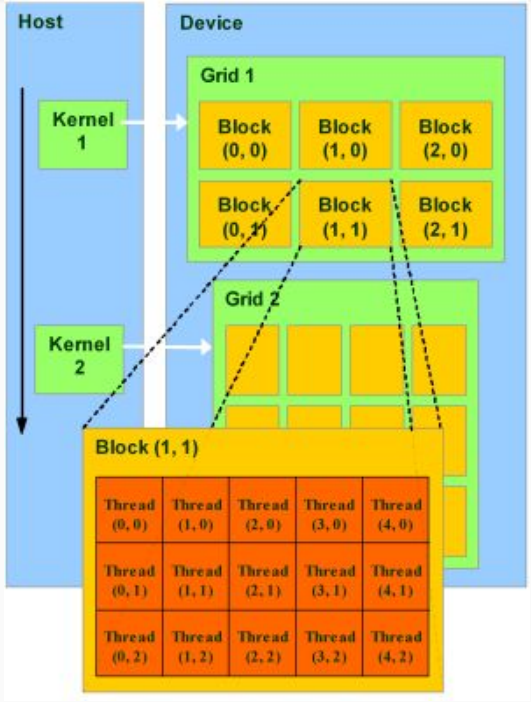
B ∈ [16, 32, 64, 128]

--parameter tuning

Multithread on CUDA

-CUDA creates multiple blocks and each block has multiple threads.

-Each thread calls the same kernel to process a section of the data.



Detail about Shared Memory

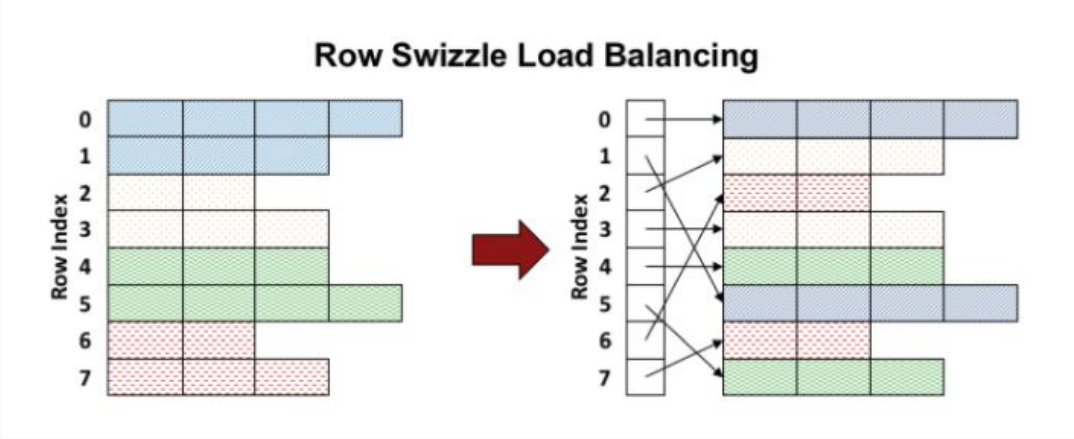
```
__global__ void matrixMultiplyShared(int* a, int *b, int *c)
{
    int x = blockIdx.x;
    int y = blockIdx.y;
    int k = threadIdx.x;
    __shared__ int p[col1];

    c[col2*y+x] = 0;
    p[k] = a[col1*y+k]*b[col2*k+x];

    __syncthreads();

    for(int i = 0; i < col1; i++)
    {
        c[col2*y+x] = c[col2*y+x]+p[i];
    }
}
```

```
__shared__ float smem_c[64][64];
__shared__ float smem_a[64][8];
__shared__ float smem_b[8][64];
int c = blockIdx.x * 64;
int r = blockIdx.y * 64;
for (int kk=0; kk<N; kk+=T) {
  for (int i=threadIdx.x+blockDim.x*threadIdx.y;
i<64*8; i+=blockDim.x*blockDim.y) {
    int k = kk + i / 64;
    int rt = r + i % 64;
    int ct = c + i % 64;
    smem_a[i%64][i/64] = A[rt*N+k];
    smem_b[i/64][i%64] = B[k*N+ct];
}
__syncthreads();
```

Load Balancing

-Load balancing helps to balance the sparse matrix.

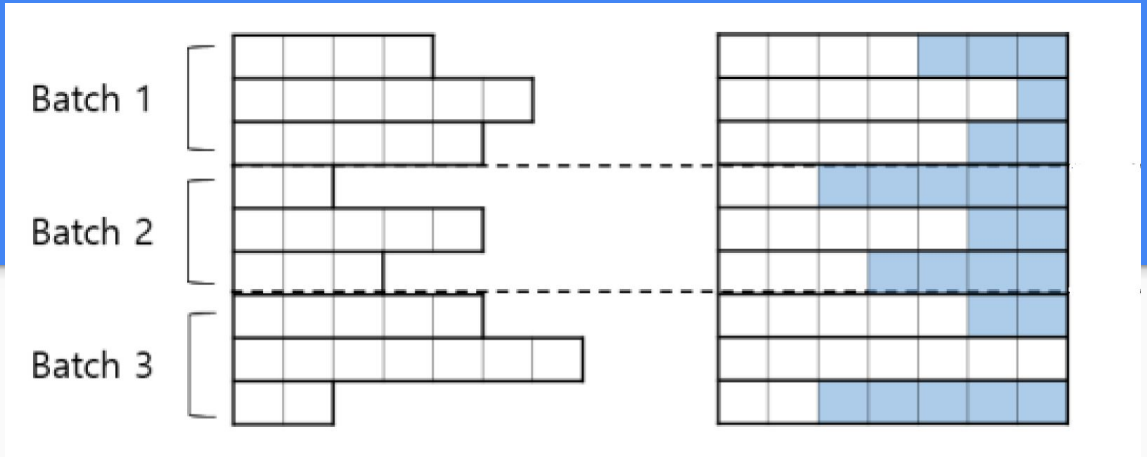
-For an overall 90% sparsity in our dataset, we have a sparsity level of 86.1% to 93.6% at each row.



Gale, T., Zaharia, M., Young, C., & Elsen, E. (2020). Sparse gpu kernels for deep learning. *arXiv preprint arXiv:2006.10901*.

Data Reuse

-Each sentence in the dataset has different words, and Transformer make it a matrix as shown in figure above.



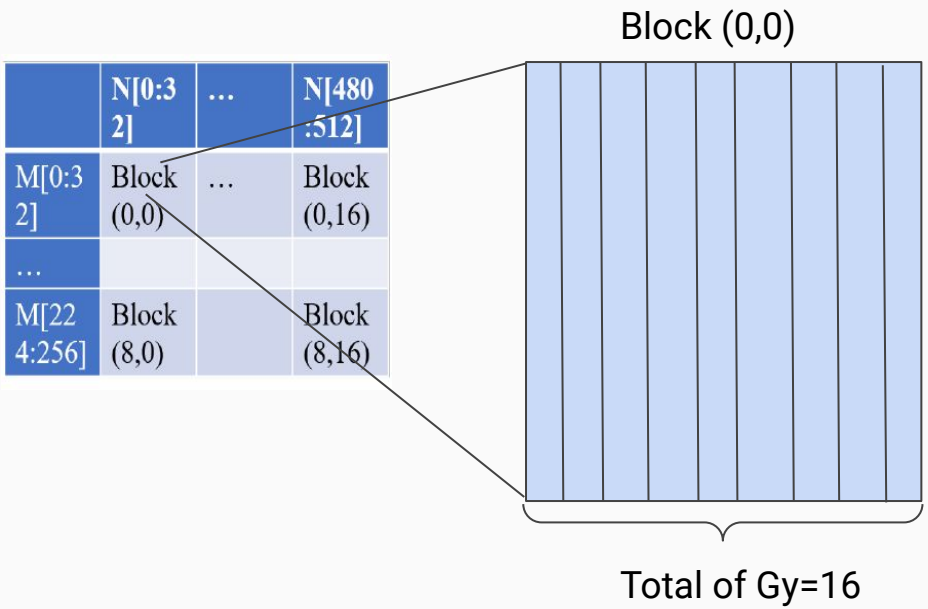
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	0.764	0.693	0.689	0.831	0.627	0.792	0.718	0.979	0.644	0.967	0.527	0.980	0.480	0.991	0.619	0.909	0.553	0.910	0.477	0.909	0.622	1.045	0.482	0.985	0.422	1.046
2	0.868	-0.121	1.193	-0.181	1.068	-0.006	1.295	-0.102	1.037	0.013	0.789	0.180	1.100	0.322	0.923	0.098	0.941	0.215	0.910	0.342	0.895	0.320	0.858	0.379	0.965	0.580
3	0.302	-0.725	0.262	-0.888	0.370	-1.019	0.328	-0.784	0.607	-0.751	0.533	-0.845	0.515	-0.696	0.720	-0.615	0.708	-0.592	0.733	-0.225	0.871	-0.349	1.085	-0.291	0.911	-0.436
4	-0.656	-0.509	-0.563	-0.617	-0.443	-1.063	-0.526	-1.000	-0.303	-1.028	0.103	-1.100	-0.006	-0.900	-0.314	-0.857	0.249	-0.878	0.295	-0.849	0.370	-1.094	0.622	-0.980	0.546	-0.695
5	-0.872	0.330	-0.988	0.142	-0.993	-0.134	-1.050	-0.223	-0.720	-0.222	-0.717	-0.654	-0.578	-0.727	-0.529	-0.719	-0.512	-0.665	-0.369	-0.566	-0.153	-0.754	-0.029	-0.890	0.157	-1.106
6	-0.248	1.034	-0.582	1.043	-0.626	0.702	-0.535	0.711	-1.033	0.621	-0.896	0.406	-1.249	0.402	-1.076	0.018	-1.050	0.109	-0.933	-0.395	-1.015	-0.375	-0.507	-0.536	-0.982	-0.766
7	0.621	0.616	0.382	0.671	0.163	0.900	0.078	0.964	-0.228	0.924	-0.505	0.963	-0.525	0.666	-0.715	0.607	-0.923	0.483	-0.910	0.392	-0.928	0.231	-0.900	-0.068	-0.942	-0.067
8	1.032	-0.194	0.989	0.127	1.059	0.363	0.737	0.644	0.555	0.793	0.449	0.902	0.229	0.993	-0.136	0.965	-0.327	1.041	-0.407	0.873	-0.632	0.802	-0.746	0.674	-0.854	0.502
9	0.637	-1.071	0.768	-0.589	0.622	-0.497	0.968	-0.093	1.082	-0.150	1.243	0.250	0.926	0.650	0.510	0.599	0.540	0.955	0.153	1.149	0.117	1.211	-0.162	0.849	-0.211	1.020
10	-0.562	-0.728	-0.156	-0.881	0.317	-0.949	0.326	-0.929	0.670	-0.788	1.057	-0.542	1.079	-0.057	0.988	0.100	1.280	0.334	1.003	0.878	0.417	0.537	0.805	0.961	0.272	1.067
11	-0.855	-0.018	-0.957	-0.454	-0.620	-0.853	-0.479	-0.992	0.013	-1.027	0.074	-0.845	0.594	-1.045	0.789	-0.630	0.973	-0.391	1.097	-0.058	1.027	0.020	1.000	0.359	0.784	0.643
12	-0.222	0.986	-0.904	0.408	-0.665	-0.056	-1.048	-0.283	-0.812	-0.798	-0.295	-0.835	-0.147	-1.060	-0.009	-1.094	0.597	-0.747	0.683	-0.495	0.630	-0.586	1.130	-0.220	1.177	0.020
13	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
14	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
15	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
16	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
17	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
18	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
19	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
20	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
21	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
22	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
23	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
24	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
25	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
26	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
27	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
28	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
29	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
30	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
31	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
32	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
33	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031
34	0.011	-0.117	0.012	-0.053	-0.032	0.009	-0.038	-0.029	-0.026	-0.066	-0.022	0.022	0.032	-0.038	0.028	-0.014	-0.043	-0.045	-0.018	-0.041	0.039	0.000	0.040	-0.095	0.025	-0.031

Evaluations

- Time Complexity (Running Time)

Thanks! Question?

Densify and Tile (Continue)



- Block is divided into Gy thread groups of size Gsy
  - Assume Gy=16
  - then Gsy = K/Gy = 3056/16 = 191
- Thread group 0 of thread block (0,0), would process

$$C[i,k] = \sum_{j=0}^{190} A[i,16j] \times B[16j,k],$$

,where i = [0,1,…,31], k=[0,1,…,31]

Load Balancing among blocks

K0	M[0:32]	M[32:64]	…	M[480:512]
K1	M[0:32]	M[32:64]	…	M[480:512]
…	…	…	…	…
K3055	M[0:32]	M[32:64]	…	M[480:512]

First way: Different thread blocks process different number of K elements, while keeping the number of M elements processed by each thread block the same.

Benefit: good at load balancing

Weakness: thread block process more than one output tile =>complicated logic

Densify and Tile (Continue)

Algorithm 1 GeMM at thread group level: example

Input: matrix pointers A, B, C
$ACC[0:32, 0:32] = \{0.0\}$
for b in $0, 16, 32 \ldots 3040$ **do**
  Cache $A[0:32, b]$ in shared memory
  Cache $B[b, 0:32]$ in registers
  for a in $0, 1 \ldots 32$ **do**
    $ACC[a, 0:32] += A[a, b] \times B[b, 0:32]$
  end for
  Add ACC to C atomically using shared memory
end for

Block 0 Thread Group 0	
Step 0	Load B[0:0:32]
Step 1 (K0 M0)	C[0,0:32] += A[0,0] × B[0,0:32]
Step 2 (K0 M1)	C[1,0:32] += A[1,0] × B[0,0:32]
…	…
Step 32 (K0 M31)	C[31,0:32] += A[31,0] × B[0,0:32]

Densify and Tile (Continue)

Algorithm 2 GeMM at thread group level: general case

Input: matrix pointers A, B, C
K_{list} = list of K elements to process for thread group
M_{list} = list of M elements to process for thread group
N_{list} = list of N elements to process for thread group
$ACC[M_{list}, N_{list}] = \{0.0\}$
for b in K_{list} **do**
    Cache $A[M_{list}, b]$ in shared memory
    Cache $B[b, N_{list}]$ in registers
    for a in M_{list} **do**
        $ACC[a, N_{list}] += A[a, b] \times B[b, N_{list}]$
    end for
    Add ACC to C atomically using shared memory
end for

Load Balancing among blocks (Continue)

K0	M[0:36]	M[36:63]	…	M[485:512]
K1	M[0:36]	M[36:63]	…	M[485:512]
…	…	…	…	…
K3055	M[0:36]	M[36:63]	…	M[485:512]

Second Way:assign different number of elements in M to different thread blocks to balance the number of nonzero values in the thread blocks.

Benefit: thread block only processes one output tile

Weakness:

Tile in different sizes => CUDA programming model not support load balancing is worse

Load Balancing among groups

Thread Group 0	Thread Group 1	…	Thread Group 15
K0 M0	K1 M0	…	K15 M0
…	…	…	…
K0 M31	K1 M 31	…	K15 M31
…	…	…	…
K3040 M0	K3041 M0	…	K3055 M0
…	…	…	…
K3040 M31	K3041 M31	…	K3055 M31

Thread Group 0	Thread Group 1	…	Thread Group 15
K0 M0	K1 M0	…	K15 M0
K0 M15	…	…	…
…	…	…	K3054 M11
K200 M25	K380 M27	…	
K200 M31	K380 M29	…	

Code Generation

Algorithm 2 GeMM at thread group level: general case

Input: matrix pointers A, B, C
K_{list} = list of K elements to process for thread group
M_{list} = list of M elements to process for thread group
N_{list} = list of N elements to process for thread group
$ACC[M_{list}, N_{list}] = \{0.0\}$
for b in K_{list} **do**
    Cache $A[M_{list}, b]$ in shared memory
    Cache $B[b, N_{list}]$ in registers
    for a in M_{list} **do**
        $ACC[a, N_{list}] += A[a, b] \times B[b, N_{list}]$
    end for
    Add ACC to C atomically using shared memory
end for

Algorithm 3 SpMM for thread group

Input: matrix pointers A, B, C
K_{list} = list of B elements to process for thread group
M_{list} = list of A elements to process for thread group
N_{list} = list of C elements to process for thread group
$ACC[M_{list}, N_{list}] = 0.0$ (initialize accumulation buffer)
for b in K_{list} **do**
    M_{nnz} = M indices of nonzero elements in $A[M_{list}, b]$
    Cache $B[b, N_{list}]$ in registers
    for a in M_{nnz} **do**
        $ACC[a, N_{list}] += A[a, b] \times B[b, N_{list}]$
    end for
end for

Supporting Convolutions

- SpMM is good for 1x1 convolutions but not 3x3 convolutions => im2col
- Im2col : the convolution is basically represented as a matrix multiplication by replicating the input activations.
- In SparseRT, B can be treated as a “virtual” dense matrix
  - Bounds check required for padding can be simplified

Autotuning

Recall: Several parameters [M_blocks, N_blocks and Gy]

Autotune with exhaustive grid search.

- The split factors should be divisible by their respective axes.
- the block size and the shared memory usage of the kernel is supported by the GPU architecture.

Section 4: Evaluation

Tasks: **SpMM**

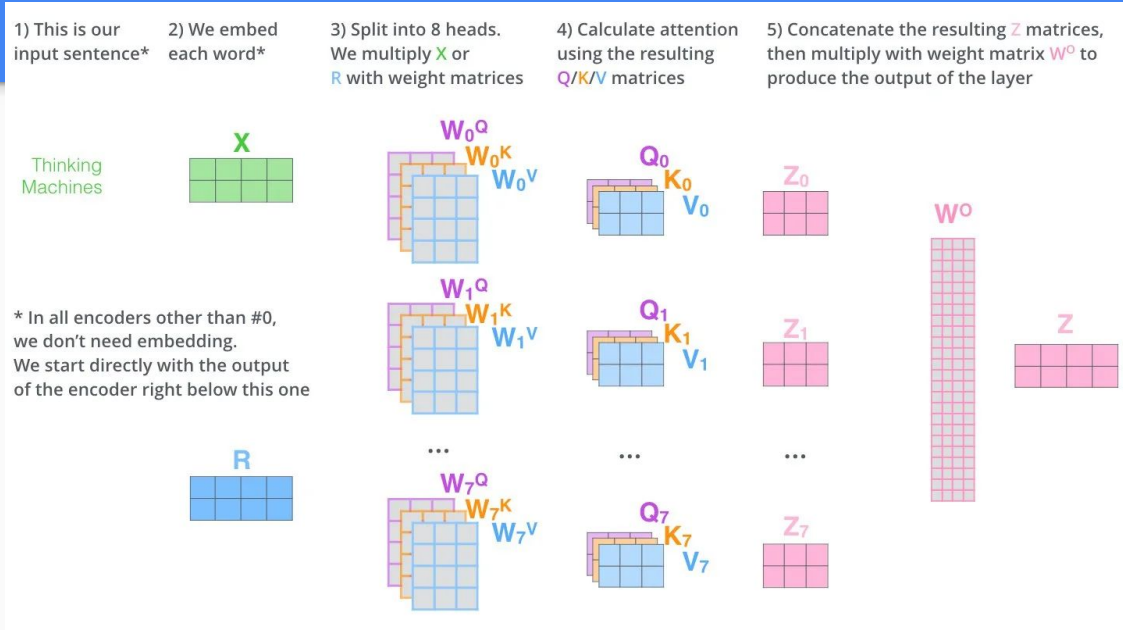
Benchmarks: **cuDNN**, **cuBLAS** and **cuSPARSE?**

Hardware

SpMM Tasks

We use pruned weights from real neural networks.

Natural Language Processes: **Transformer (90% and 95% sparsity).** WMT English-to-German 2014 dataset.



SpMM Tasks (continue)

Top-1 model accuracy of the original and pruned weight performance at different sparse levels. This indicates the 90% sparsity level is more meaningful.

Network	Sparsity Level	Original Performance	Pruned Performance
ResNet-50	90%	76.7%	75.2%
ResNet-50	95%	76.7%	72.7%
Transformer	90%	27.3	23.3
Transformer	95%	27.3	20.7
MobileNet V1	90%	70.9%	68.4%

SpMM Tasks (continue)

This is then result in 20 problems based on different M, K, N settings.

Count represents the times of SpMM problems appear in the network, which contributes to totally 257 instances.

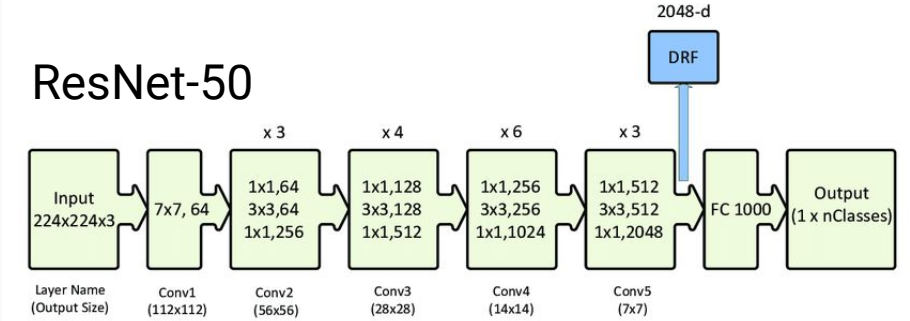


Table 1: Results for SpMM problems in deep learning. Problems are taken from publicly available state-of-the-art pruned neural networks [11, 14] (RN50 = ResNet-50, NLP = Transformer, MbNetV1 = MobileNet V1). Count represents the number of times this SpMM problem dimension appear in the network. Benchmarked on the Tesla T4.

Problem Number	*M*	*K*	*N*	Use Case	Count	% nnz	Speedup wrt cuBLAS
1	64	256	3136	RN50	2	0.90	4.3
2	256	64	3136	RN50	2	0.90	2.5
3	128	512	784	RN50	3	0.90	4.0
4	512	128	784	RN50	3	0.90	3.7
5	256	1024	196	RN50	5	0.90	3.0
6	1024	256	196	RN50	5	0.90	3.1
7	512	2048	49	RN50	3	0.90	2.7
8	2048	512	49	RN50	3	0.90	4.8
1	64	256	3136	RN50	2	0.95	5.7
2	256	64	3136	RN50	2	0.95	3.0
3	128	512	784	RN50	3	0.95	5.9
4	512	128	784	RN50	3	0.95	4.5
5	256	1024	196	RN50	5	0.95	4.4
6	1024	256	196	RN50	5	0.95	4.2
7	512	2048	49	RN50	3	0.95	4.5
8	2048	512	49	RN50	3	0.95	6.6
9	2048	512	256	NLP	12	0.90	4.2
10	512	2048	256	NLP	12	0.90	4.2
11	512	512	256	NLP	72	0.90	6.0
9	2048	512	256	NLP	12	0.95	6.8
10	512	2048	256	NLP	12	0.95	7.6
11	512	512	256	NLP	72	0.95	9.0
12	64	32	12544	MbNetV1	1	0.90	1.4
13	128	64	3136	MbNetV1	1	0.90	3.4
14	128	128	3136	MbNetV1	1	0.90	4.0
15	256	128	784	MbNetV1	1	0.90	4.2
16	256	256	784	MbNetV1	1	0.90	3.5
17	512	256	196	MbNetV1	1	0.90	2.3
18	512	512	196	MbNetV1	5	0.90	3.2
19	1024	512	49	MbNetV1	1	0.90	2.6
20	1024	1024	49	MbNetV1	1	0.90	4.5

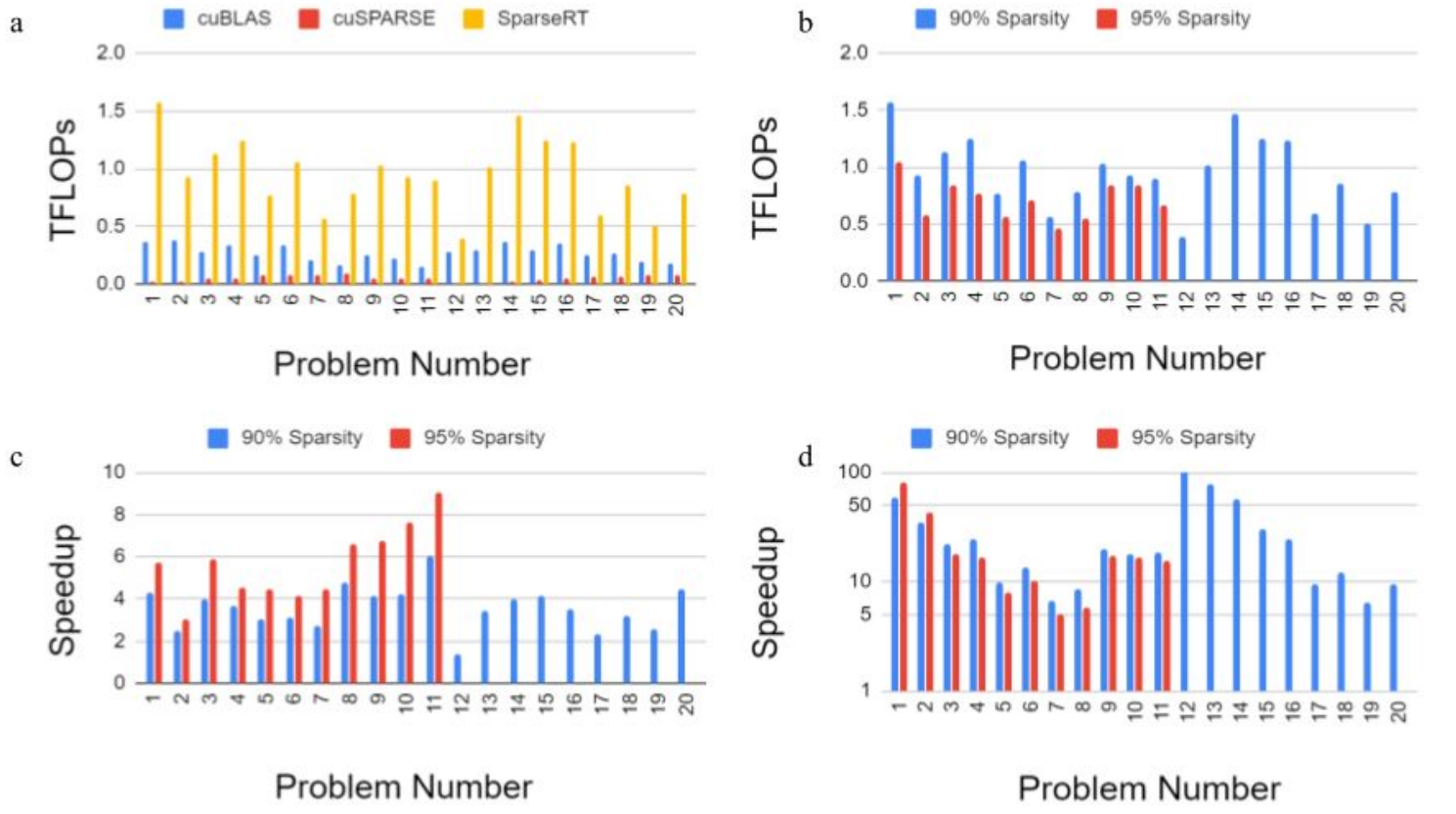
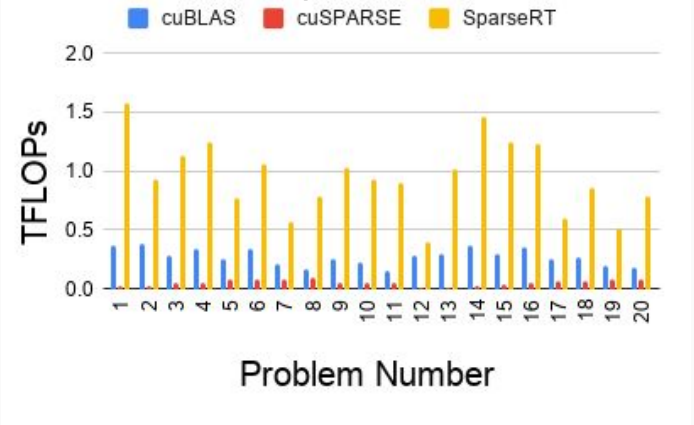


Figure 4: Comparison of our kernels against Nvidia vendor libraries. a) Teraflops per second (TFLOPs) achieved by cuBLAS vs cuSPARSE and SparseRT. cuBLAS performs the sparse matrix multiplication as a dense matrix multiplication, thus requiring 10 times more total FLOPs. We evaluate at 90% sparsity. b) TFLOPs achieved by SparseRT at different sparsity levels. c) Fold speedup of SparseRT over equivalent dense computation performed by cuBLAS. d) Fold speedup of SparseRT over cuSPARSE.

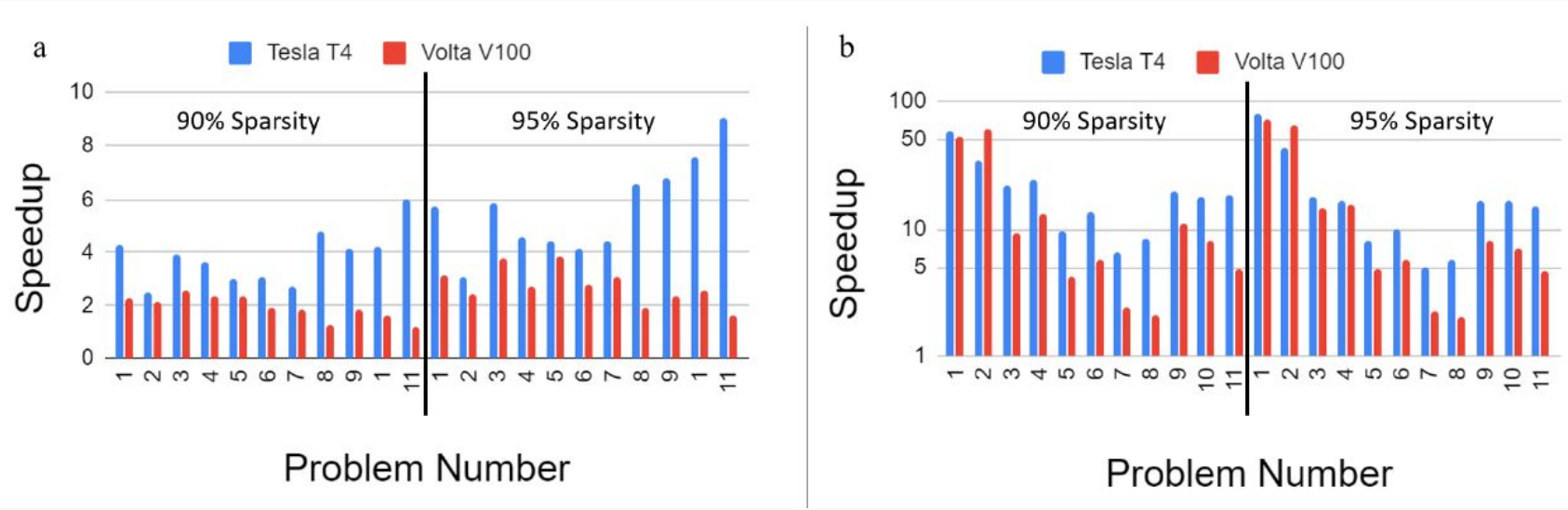
SpMM Tasks (continue)

- SparseRT's performance slightly degrades as N decreases and M and K increase, whereas cuSPARSE's performance increases. This consistent with the current SpMM engineering efforts on the GPU that cuSPARSE is focusing on cases where the dense matrix is very slim.

- And, when N decreases or is not a multiple of a power of 2, such as 49 in problem number 7 and 8, the tiling strategies for N is severely limited in SparseRT.



SpMM Tasks (continue)



Speedup comparing to cuBLAS

Speedup comparing to cuSPARSE

SpMM Tasks (continue)

Sputnik uses a combination of optimization techniques such as row reordering, memory alignment and subwarp tiling.

Sputnik offers a huge improvement over state-of-theart baselines such as ASpT and cuSPARSE.

On MobileNet V1, SparseRT achieved faster speed at sparsity of 90%. On sparsity of 95%, we achieved comparable performance.

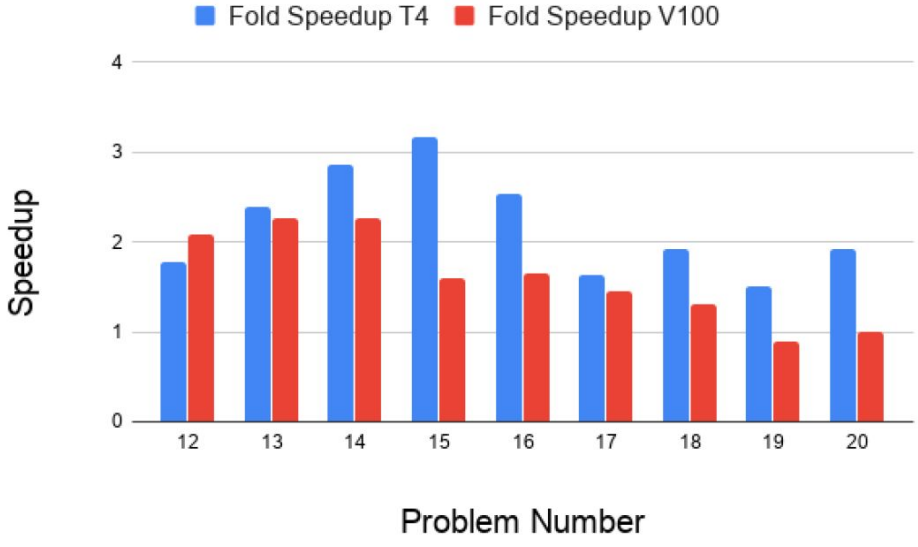


Figure 6: Performance of SparseRT vs Sputnik on the SpMM problems in MobileNet V1.

Sparse Convolution

3x3 Conv tasks in ResNet-50 on T4. Assume batch size and padding of 1.

Same settings as 1x1 conv task before.

4 Different M, N, K settings in 2 sparsity levels.

Table 3: Sparse convolution problems in deep learning. All filter sizes are 3x3 with a padding of 1. Problems are taken from pruned ResNet-50 [14].

Problem Number	Image Dimension	Input Channels	Output Channels	Sparsity	Speedup
1	56	64	64	0.9	3.7
2	28	128	128	0.9	2.0
3	14	256	256	0.9	1.4
4	7	512	512	0.9	2.4
1	56	64	64	0.95	5.3
2	28	128	128	0.95	3.5
3	14	256	256	0.95	2.5
4	7	512	512	0.95	8.5

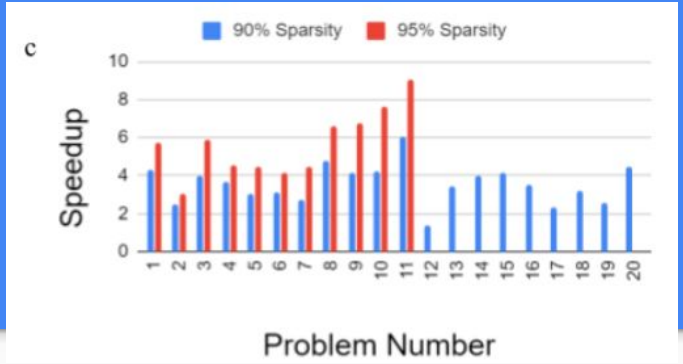
Sparse Convolution (continue)

Figure 7 is missing...

As expected, the cuDNN baseline use the Winograd algorithm, which is 2x faster than im2col + GeMM.

This means the speedup of SparseRT's im2col + SpMM convolution over the comparable im2col + GeMM convolution in cuDNN is even larger than the numbers presented in Table 3, confirming the efficiency of our SpMM implementation.

Analysis



Problem 1 and 3 have smaller sparse matrix than dense matrix, problem 6 and 7 have larger sparse matrix than dense matrix. Why SparseRT cannot offer as much speedup over cuBLAS when sparse matrix is too large?

In dense matrix multiplication (GeMM), the asymptotic runtime is proportional to MNK. This is because optimized GeMM routines in cuBLAS are typically compute-bound, dense GeMM has O(MNK) multiplications.
However, Sparse kernels, are typically memory-bound instead of compute-bound since SparseRT employs aggressive code unrolling approaches. When M and K increase, the 64Kb constant cache is depleted, and leads to less efficiency. SparseRT kernel runtimes are roughly proportional to MK, instead of MNK.

Analysis (continue)

The input size is 224x224 in this study, which may smaller than real cases.
If the size is larger in real cases, it increased N, rather than M or K, which means SparseRT will be better in real cases.

Comparing to Sputnik (SpMM)

Same: parallelizing the dense matrix columns across thread blocks

Difference:
Sputnik stores the sparse matrix data **in shared memory**, and is **bottlenecked** by shared memory loads. Thus, its novel optimizations are focusing on the are targeted **towards mitigating the load bandwidth bottleneck** by enabling vector memory loads (i.e., subwarp tiling and reverse offset memory alignment)
SparseRT, the sparse matrix data is **stored exclusively** in the instruction cache baked into instruction operands, so these optimizations are not necessary. But SparseRT suffer from instruction fetch bottleneck for larger matrices.

Similar: Sputnik applied row-swizzling strategy on the sparse matrix to improve load balancing as well, but it is a preprocessing strategy in Sputnik. We are evaluating if it benefits SparseRT.

Comparing to ASpT and MergeSpMM (SpMM)

Not compared since ASpT and MergeSpMM are already compared with Sputnik and shown to be inferior on almost all deep learning SpMM problems considered in other papers.

Comparing to SkimCaffe and Tiramisu on CPUs and Escort on GPU (3x3 Convolution)

Not compared since this study is working on GPU, and Escort is 2-3 times faster than im2col + cuBLAS GeMM, but the latest cuDNN library is several times faster than im2col + cuBLAS GeMM. So this paper compare with cuDNN only.

Escort aims to alleviate memory bottlenecks through putting the sparse matrix and the dense matrix into different forms of on-chip data caches.
SparseRT puts the sparse matrix into the instruction cache and constant cache, avoiding the on-chip data caches altogether. This “virtual” dense matrix technique is similar to SkimCaffe and Tiramisu.

Discussion

Limitation:

- SparseRT is based on the pruning technology. However, the performance of modern DNNs is often bottlenecked by other operations, such as softmax and self-attention. For example, in the self-attention in Transformer, even though the weight matrix used to generate the query and key vectors can be pruned, the query and key vectors themselves are still dense, making the self-attention a dense operation.
- SparseRT currently does not support sparse gradient computations, which are typically sampled dense-dense matrix multiplications.
- Only focus on single layer. We refrain from presenting end-to-end neural network inference timing results here because there could be a variety of confounding factors due to inter-layer optimizations employed by state-of-the art inference engines such as TensorRT.

Potential applications: SparseRT is designed for sparse inference, SparseRT can also potentially be applied to sparse training.

Conclusion

- We present significant speedups on a test suite of hundreds of SpMM and sparse convolution problems in deep learning.
- Future work include supporting other sparse operations such as sampled dense matrix matrix multiplication, lower precision data formats such as int8, and porting to other architectures such as multicore CPUs with vector instructions.
- We hope to inspire further research on unstructured sparse pruning methods.

The End

-Thanks for Listening!

-Questions?