Vectorization Additions to SMaT CPU Code and use Case Comparison of SMaT and dgl-SpMM

Jason Shin
jshin60@u.rochester.edu

***Abstract*— Recent frameworks such as SpTransX [7] employ the utilization of Sparse Dense Matrix Multiplication (SpMM) to show empirical performance gains on knowledge graphs. For its implementation, SpTransX utilizes dgl g-SPMM [16] for GPU based SpMM operations. Highly optimized SpMM libraries such as SMaT [11] can outperform dgl g-SPMM and other cuSparse based SpMM libraries by $8.60 - 16.32\times$ in most cases. As such, a comparison of dgl g-SpMM and SMaT can determine if employing SMaT into SpTransX can improve the SpMM operation performance of SpTransX. This project is separated in two parts. The first vectorizes portions of the csr to bcsr code of SMaT showing a $3.85\times$ reduction in average cycles, $3.41\times$ reduction in average instruction count, and a $1.69\times$ reduction in average execution time. The second compares the adj and adjneg SpMM operation times and throughput of transe, transh, and tansr on dgl g-SPMM and SMaT finding that SMaT under performs compared to dgl g-SPMM due to the highly sparse nature (99.99% sparsity) of the SpMM operations performed.**

I. Introduction

Recent frameworks such as SpTransX [7] employ the utilization of Sparse Dense Matrix Multiplication (SpMM) to show empirical performance gains on knowledge graphs embedding (KGE) methods such as transe [8] which were designed for tasks such as social network analysis and recommender systems. SpTransX utilizes dgl g-SpMM [16] which is based on cuSparse for its implementation. If dgl g-SpMM was replaced with a highly optimized SpMM library, it could potentially allow for further optimization of SpTransX's training pipeline. Highly optimized SpMM libraries such as SMaT [11] report higher empirical SpMM performance compared to cuSparse potentially introducing performance gains to SpTransX allowing for larger Knowledge Graphs (KG) to be trained in a reasonable amount of time. When examining the SMaT source code, some of the commonly utilized CPU code (i.e. csr to bcsr in matrix.h) does not employ vectorization. This project does the following:

1) Implement loop vectorization, loop unrolling, and software pipelining to the csr to bcsr conversion of SMaT [6] as Intel AVX intrinsics [3] to reduce the number of instructions used by the SMaT pipeline.
2) Compare SMaT to dgl-SpMM on SpMM operations performed by the SpTransX reproducibility library [1] observing the conditions in which SMaT improves upon dgl-SpMM.

II. Background

A. SpTransX

SpTransX [7] is a recently introduced framework for computing knowledge graph (KG) embeddings as Sparse-Dense Matrix Multiplications (SpMM) making improvements to four translational models by representing knowledge graph embedding (KGE) computations as a series of SpMM operations to reduce training time as well as memory usage. This improves upon previous KGE calculation methods which utilize dense matrices and suffer from high memory demands and expensive back propogation calculations. SpTransX utilizes dgl g-SPMM [16] for GPU SpMM calculations taking an input of COO matrices for GPU SpMM operations and CSR matricies for CPU SpMM operations.

B. SMaT

SMaT [11], a recently introduced SpMM library utilizing a method with Tensor Cores for additional performance in SpMM operations shows higher emperical performance than other SpMM and SpVM libraries such as cuSparse and DASP being $\times 16.32$ and $\times 10.78$ faster on the suitesparse dataset. Within synthetic matrix tests where $C = A \times B$ and A has the shape $N \times N$, SMaT performs well when N is large being $\times 8.60$ faster than cuSparse when $N = 1000$.

C. CSR, COO, and BCSR

Recent research comparing CSR/COO matrices to BCSR matrices [12] has shown that the usage of BCSR on moderately sparse to highly sparse graphs could have a similar runtime to CSR and COO matrices with reduced GPU memory consumption in tasks such as Breadth First Search.

III. Method

The main work done was on improving SMaT's csr to bcsr conversion code primarily focusing on vectorizing the blockindex calculations. This is due to the remaining code containing a large sum of potentially non contiguous matrix accesses which require gathers and scatters that are inefficient for the CPU to conduct. Such code is kept unvectorized as well as portions of code which cannot be done with AVX SIMD operations (i.e. division, modulo, and the multiplication of two vectors with 64 bit integers). The code optimizations focus on three of the four blocks of loops within the csr to bcsr conversion function. The first block of code from the SMaT source code [6] has the following form:

```
for (size_t row = 0; row < m_row; row++){
```

```
    size_t j = csrRowPtr_host[row]
    for (; j < csrRowPtr_host[row + 1];
    j++){
        size_t col = csrColIdx_host[j];
        size_t rowRegion = row / MMA_M;
        size_t colRegion = col / MMA_K;
        size_t blockIndex = rowRegion *
        numColRegions + colRegion;
        //Remaining unaltered code
    }
}
```

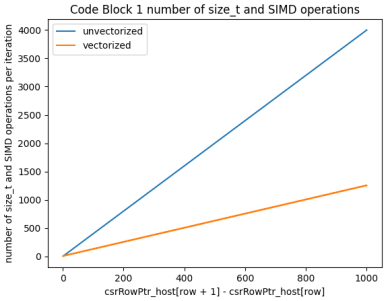
which for the portion of interest will produce $4 * (csrRowPtr\_host[row + 1] - csrRowPtr\_host[row])$ $size\_t$ operations for each iteration of the outer loop.

When applying vectorization, we can assign the vectors for $rowRegion * numColRegions$ outside of the inner loop of the code block. This means that we only need to assign each group of $csrColIdx\_host[j]/MMA\_K$ to a vector and perform a single vector addition within the inner loop. As the code utilizes AVX intrinsics [3] which are 256 bit vectors and $size\_t$ values are 64 bit, each iteration of the loop will include four calculations simultaneously. As such we can produce the following code vectorization:

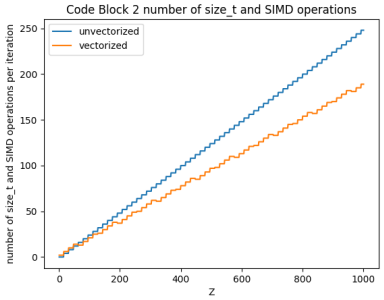
```
size_t* blockindex_nums = (size_t *)
calloc(sizeof(size_t), 4);
for (size_t row = 0; row < m_row; row++){
    size_t rdivm = row / MMA_M;
    __m256i b = _mm256_set1_epi64x(rdivm
    *numColRegions);
    size_t j = csrRowPtr_host[row]
    for (; j+4 < csrRowPtr_host[row + 1];
    j=j+4){
        __m256i colRegion =
        _mm256_set_epi64x(
        csrColIdx_host[j+3]/MMA_K,
        csrColIdx_host[j+2]/MMA_K,
        csrColIdx_host[j+1]/MMA_K,
        csrColIdx_host[j]/MMA_K);
        __m256i blockIndex =
        _mm256_add_epi64(b, colRegion);
        _mm256_store_si256(
        (__m256i*)blockindex_nums,
        blockIndex);
        //Remaining unaltered code
    }
    //unvectorized code block
}
free(blockindex_nums);
```

The vectorized code first places the calculation and vector assignment of $row/MMA\_M * numColRegions$ outside of the inner loop producing two $size\_t$ operations every iteration. Within the loop we have to conduct four $size\_t$ operations (division) to assign the four col/MMA_K values. We can then calculate the four blockindicies with a single SIMD addition. The results are then stored to an array for the non vectorized portion of code in which spatial locality can reduce the memory access time of each of the blockindex calculations. As such, with vectorization the number of $size\_t$ operations and SIMD operations can be reduced down to $2 + 5(\lfloor (csrRowPtr\_host[row + 1] - csrRowPtr\_host[row])/4 \rfloor) + 4((csrRowPtr\_host[row + 1] - csrRowPtr\_host[row])\%4)$ per each iteration of the outer loop. Further loop unrolling and basic software pipelining (although as CPUs can perform independent instructions out of order, it is unlikely any software pipelining will increase the pipeline parallelism of the code) can be applied to potentially allowing for the total number of instructions utilized by the program to be reduced as fewer instructions are spent on the loop itself and more on the instructions within the loop:

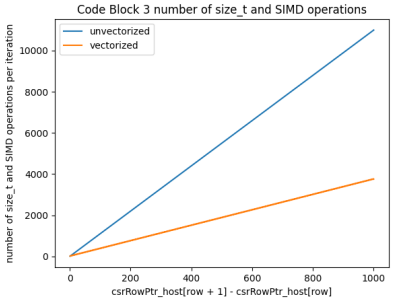
```
size_t* blockindex_nums_16 = (size_t *)
malloc(sizeof(size_t)*8);
size_t* blockindex_nums = (size_t *)
calloc(sizeof(size_t), 4);
for (size_t row = 0; row < m_row; row++){
    size_t rdivm = row / MMA_M;
    __m256i b = _mm256_set1_epi64x(rdivm
    *numColRegions);
    size_t j = csrRowPtr_host[row]
    for (; j+8 < csrRowPtr_host[row + 1];
    j=j+8){
        __m256i colRegion1 =
        _mm256_set_epi64x(
        csrColIdx_host[j+3]/MMA_K,
        csrColIdx_host[j+2]/MMA_K,
        csrColIdx_host[j+1]/MMA_K,
        csrColIdx_host[j]/MMA_K);
        __m256i colRegion2 =
        _mm256_set_epi64x(
        csrColIdx_host[j+7]/MMA_K,
        csrColIdx_host[j+6]/MMA_K,
        csrColIdx_host[j+5]/MMA_K,
        csrColIdx_host[j+4]/MMA_K);
        __m256i blockIndex1 =
        _mm256_add_epi64(b, colRegion1);
        __m256i blockIndex2 =
        _mm256_add_epi64(b, colRegion2);
        _mm256_store_si256(
        _mm256_storeu_si256(
        (__m256i*)blockindex_nums_16,
        blockIndex1);
        _mm256_storeu_si256(
        (__m256i*)&blockindex_nums_16[4],
        blockIndex2);
        //Remaining unaltered code
    }
    //initial vectorized code block
    //unvectorized code block
}
free(blockindex_nums);
free(blockindex_nums_16);
```



(a) Number of $size\_t$ and SIMD operations per iteration of code block 1 as $csrRowPtr\_host[row + 1] - csrRowPtr\_host[row]$ increases.



(b) Number of $size\_t$ and SIMD operations per iteration of code block 2 as $Z = \lfloor m\_col/16 \rfloor$ increases.



(c) Number of $size\_t$ and SIMD operations per iteration of code block 3 as $csrRowPtr\_host[row + 1] - csrRowPtr\_host[row]$ increases.

Fig. 1: The number of $size\_t$ and SIMD operations per iteration as $csrRowPtr\_host[row + 1] - csrRowPtr\_host[row]$ or $Z = \lfloor m\_col/16 \rfloor$ varies.

This ultimately creates a block of code that produces $2 + 10(\lfloor B_1/8 \rfloor) + 5(\lfloor (B_1\&8)/4 \rfloor) + 4((B_1\%8)\%4)$ $size\_t$ and SIMD operations per iteration of the outer loop (where $B_1 = csrRowPtr\_host[row + 1] - csrRowPtr\_host[row]$).

For the second code block in the SMaT source code [6]:

```
for (size_t row = 0; row < m_row;
row += MMA_M) {
    size_t col = 0;
    for (; col < m_col; col += MMA_K) {
        size_t current_block =
        row / MMA_M * numColRegions +
        col / MMA_K;
        //Remaining unaltered code
    }
}
```

we can apply the same strategy as we did for the first code block to reduce the expected number of $size\_t$ and SIMD operations from $4N$ to $2 + 24(\lfloor Z/8 \rfloor) + 11(\lfloor (Z\%8)/4 \rfloor) + 4((Z\%8)\%4)$ $size\_t$ and SIMD operations per iteration of the outer loop (where $Z = \lfloor m\_col/16 \rfloor$).

For the third code block in the SMaT source code [6]:

```
for (size_t row = 0; row < m_row;
row++) {
    size_t j = csrRowPtr_host[row];
    for (; j < csrRowPtr_host[row + 1];
    j++) {
        size_t col = csrColIdx_host[j];
        size_t rowRegion = row / MMA_M;
        size_t colRegion = col / MMA_K;
        size_t blockIndex = rowRegion *
        numColRegions + colRegion;
        half val = csrVal_host[j];
        size_t offset = row % MMA_M *
        MMA_K + col % MMA_K;
        size_t bcsrIndex =
        relativeBlockIndexMapping_host
        [blockIndex]
        * MMA_M * MMA_K + offset;
        blockSize, offset, bcsrIndex);
        bcsrVal_host[bcsrIndex] = val;
    }
}
```

We have two points in which the code will not be vectorized, at the end with $bcsrVal\_host[bcsrIndex] = val$; and in the middle when we call $relativeBlockIndexMapping\_host[blockIndex]$. For the first we can apply similar methods as before (storing the $bcsrIndex$ values into memory) as it is simply assigning values to non contiguous indexes of $bcsrVal\_host$. For the second we have to switch from AVX SIMD intrinsics [3] to non vectorized code as we will multiply three $size\_t$ values together (thus two 64 bit multiplications). As such, we have to store the values of $blockIndex$ into memory and perform the two multiplications. This means that the calculation of the $offset$ must be done either before we temporarily switch to a chunk of non vectorized code or after. It is done before the section of non vectorized code, as $col\%MMA\_K$ must be calculated to get the $offset$. We also have to calculate col/MMA_K prior to the non vectorized block of code and can take advantage of spatial and temporal locality (as we are accessing the same contiguous elements of $csrColIdx\_host$ for both) to reduce the memory access time. If we apply the same vectorization, loop unrolling, and software pipelining as code blocks 1 and 2 (with the consideration of needing to switch to non vectorized code for a portion of the calculation), we can reduce the expected number of $size\_t$ and SIMD operations from $11B_1$ to $5 + 30(\lfloor B_1/8 \rfloor) + 15(\lfloor (B_1\%8)/4 \rfloor) + 11((B_1\%8)\%4)$ for each iteration of the outer loop (where $B_1 = csrRowPtr\_host[row + 1] - csrRowPtr\_host[row]$).

For all three code blocks, we can observe that applying vectorization and loop optimizations can allow for a reduction in the number of expected instructions (in the chunks of altered code). We can see from Figure 1 that in almost all cases (other than extremely small $csrRowPtr\_host[row +$

$1] - csrRowPtr\_host[row]$ or $Z = \lfloor m\_col/16 \rfloor$ values), employing code vectorization can reduce the number of $size\_t$ and SIMD operations we conduct per iteration and thus the total number of $size\_t$ and SIMD operations for these blocks of code. The largest benefit is for code blocks 1 and 3 in which the difference number of expected $size\_t$ and SIMD operations grows significantly as the space between row pointer values grow. In the context of the matrices that will run within the SpTransX reproducibility code [1], for the largest sparse matrix (the sparse matrices associated with transe SpMM operations) $\lfloor m\_col/16 \rfloor = 1018$ and the average $csrRowPtr\_host[row+1] - csrRowPtr\_host[row]$ value should be $\tilde{3}$. This means that for code blocks 1 and 3, we do not gain any benefit to vectorization and for code block 2 we reduce the expected $size\_t$ and SIMD operations from 4,072 to 3,054 per iteration of the inner loop. As we have 393,216 rows or 24,576 iterations of the outer loop for code block 2, we should at most save 25,018,368 $size\_t$ and SIMD operations for any of the sparse matrices used in by SpTransX. To take advantage of the instruction reductions for code blocks 1 and 3 by vectorization we need on average more elements per row (thus a more dense matrix).

Beyond the implementation of AVX SIMD intrinsics [3] into the csr to bcsr function of SMaT, minor changes (which reference some online C tutorials on file reading and writing [9], [10], [13], [14] for its implementation) were made to the tester.h source code of the reproducibility library [6]. This allowed for comparisons of SMaT and dgl-SpMM to be made for SpMM operations from transe, transh, and transr. On top of the .mtx file the code takes for the A sparse matrix, it will also read a num_data.txt file which contains the N and K dimension of the B dense matrix (each on its own line) and a float_data.txt file which contains the float values of the B dense matrix in row order form. The output is then written to a C_vals.txt for checking purposes. Although reading from these files introduces overheads, the evaluation for vectorizing csr to bcsr does not consider these overheads as it is testing the band_mtx_16384_16384.mtx synthetic test case and the SMaT and dgl-SpMM gathers the Mma-CBT-Kernel data for SMaT SpMM computations which also does not consider the overhead.

IV. Evaluation

An empirical evaluation of both the difference in performance of the vectorized and unvectorized csr to bcsr conversion function as well as the performance difference between SMaT and dgl-SpMM were conducted. For the csr to bcsr conversion test the built in band_mtx_16384_16384.mtx synthetic test provided by the reproducibility library [6] was conducted with the following command line arguments:

```
-M=512 -N=512 -K=512 -enable_wmma=true
-enable_mma=true -warmup_iterations=1
-profiling_iterations=10
-sleep_duration=100
-enable_check=false -n_mult=1
-filename=<path to file>
```

For the SMaT and dgl-SpMM test, dgl-SpMM data was gathered from profiling the reproducibility code of SpTransX [1] (specifically the transe-fastkg.py, transh-fastkg.py, and transr-fastkg.py scripts) and SMaT data was gathered utilizing the following command line arguments:

```
-M=512 -N=512 -K=512 -enable_wmma=true
-enable_mma=true -warmup_iterations=0
-profiling_iterations=1
-sleep_duration=100
-enable_check=false -n_mult=1
-filename=<path to file>
```

For the SMaT SpTransX tests .mtx files were generated utilizing scipy csr arrays [2] and the mmwrite function [4]. The float values of the dense B matrix were gathered utilizing numpy savetxt [5] (a tutorial [15] was also briefly referenced for writing files in python).

A. Vectorized CSR to BCSR

For the band_mtx_16384_16384.mtx synthetic test, the SMaT baseline had on average 261,476,275,200 cycles and 202,674,662,400 instructions. The average runtime of the test was 101.817 seconds with an average IPC of 0.775 for the CPU portion. With the vectorization and loop unrolling of index calculations in the three blocks the average number of cycles was 67,936,960,000 and average total number of instructions 59,477,376,000. The average time was 60.18 seconds with an average IPC of 0.875. Employing loop optimization and vectorization techniques reduced the average runtime of the test by $1.69\times$, the average number of cycles by $3.85\times$, the average number of instructions by $3.41\times$, and increased the average IPC by $1.13\times$. As such we can observe that vectorizing the code blocks as well as applying loop unrolling and software pipelining heavily reduces the number of cycles and instructions. This is because with vectorization four additions can be done in one SIMD operation and loop unrolling can reduce the number of instructions spent on conditional jumps for loops. Furthermore, reordering some of the calculation to improve temporal locality of the code could also reduce the number of instructions and cycles as the result of the operations could be stored to a single point in memory or a register rather than being recomputed at every iteration if no code optimizations were applied. The lower reduction in average time is likely due to the additional overhead of switching from SIMD intrinsics to unvectorized code which requires writing (and for code block 3 reading) from memory. This can also be observed in the collected results as the average DRAM bound for the base case was 2.94% and 8.57% for the vectorized code.

B. Performance improvements from vectorization vs. unrolling and software pipelining

Basic tests on the effectiveness of just vectorization and vectorization with loop unrolling and software pipelining were conducted. What was found was that the majority of the improvements came from the vectorization of code with minimal improvements coming from loop unrolling

and software pipelining. On average the time was 56.37 seconds, the cycles were 88,082,624,000, and the instructions were 82,601,792,000 which is marginally faster than with loop unrolling and software pipelining with an additional $\tilde{2}$0,000,000,000 instructions and cycles on average. This is because the CPU does out of order execution as well as independent instruction execution. This means explicitly unrolling the loop doesn't matter as the CPU could process multiple iterations of the loop anyways due to the nature of how the CPU processes instructions. A higher average dram bound percentage of 17.98% was observed which is likely due to the fact that when switching from vectorized to unverctorized code, allowing more values to take advantage of spacial locality can reduce the number of times memory needs to be read from ram.

C. SMaT and dgl-SpMM

TABLE I

SMaT vs dgl-SpMM (Averages)

Type	Model	type	Time (ms)	TFLOP/s
dgl-SpMM	Transe	adj	13.37	0.0414
SMaT	Transe	adj	186.392	0.003
dgl-SpMM	Transe	adjneg	36.13	0.0153
SMaT	Transe	adjneg	191.742	0.003
dgl-SpMM	Transh	adj	1.324	0.0248
SMaT	Transh	adj	4.1214	0.0084
dgl-SpMM	Transh	adjneg	0.617	0.0532
SMaT	Transh	adjneg	3.9376	0.0086
dgl-SpMM	Transr	adj	680.34	0.000091
SMaT	Transr	adj	5.8774	0.0114
dgl-SpMM	Transr	adjneg	457.8	0.000135
SMaT	Transr	adjneg	5.581	0.012

SpTransX for each forward pass does two SpMM calculations. One for the adj sparse matrix and another for the adjneg sparse matrix on the embeddings. In both the transe and transh case we can observe in Table I that dgl-SpMM outperforms SMaT even though in it's papers evaluation section SMaT outperformed cuSparse by $8-16\times$ [11]. This even extends to the amount of vram utilized as SMaT for transe utilizes 2,44 GB of vram while dgl-SpMM utilizes only 1.83 GB of vram and for transh SMaT utilizes 224 MB while dgl SpMM utilizes only 41.87 MB of vram. This is likely because the adj and adjneg SpMM operations performed are extremely sparse (with all the adj and adjneg SpMM matrices for transe, transh, and transr being 99.99% being sparse) which is known to cause SMaT to under perform compared to cuSparse (and by extension dgl-SpMM which is based upon it) in some cases. However, for the adj transr SpMM operations, dgl-SpMM took 680.34 ms utilizing 152.6 MB of vram with a throughput of 0.000091 TFLOP/s and SMaT took 5.8774 ms utilizing 224 MB of vram with a throughput of 0.0114 TFLOP/s. Similarly for the adjneg SpMM operation, dgl-SpMM took 457.8 ms with a throughput of 0.000135 TFLOP/s and SMaT took 5.581 ms with a throughput of 0.012 TFLOP/s. In these cases despite the sparse matrices being extremely sparse SMaT heavily outperforms dgl-SpMM. This is potentially because the adj and adjneg matricies for dgl-SpMM do not have high row imbalance or other conditions which can cause SMaT to show low performance. However, it could also be from other factors such as potential overheads produced when gathering the dgl-SpMM data that were not present when gathering SMaT data which was done in a vacuum without the overlaying SpTransX pipeline. In general, we can argue that SMaT is not well suited for optimizing SpTransX due to its low performance with very sparse matrices and higher vram usage when running.

V. Conclusion

This report employs code vectorization to the SMaT csr to bcsr conversion of the SMaT pipeline and examines the improvements it gains upon the initial unvectorized code as well as examines SMaT's performance compared to dgl-SpMM on SpTransX SpMM operations. For the code vectorization, the per iteration operation reductions to each of code blocks was substantial enough that the additional overhead of loads and stores to switch from unvectorized to vectorized code was smaller than the improvements from these optimizations. However, it is worth noting that this project did not explore if the same benefits can be observed for very small matrices where the operation reduction is smaller. Future work can explore either fully vectorizing the csr to bcsr code observing how the usage of gathers and scatters affects the performance. Employing a GPU based csr to bcsr is also another potential direction. In the context of SpTransX, SMaT performed worse compared to dgl-SpMM due to the sparse nature of the SpMM operations conducted. Future work can look at how sparsity affects the performance of different SpMM libraries and selects certain libraries based on the sparsity information known of the given sparse matrix. Although SMaT did not outperform dgl-SpMM, the code vectorization of csr to bcsr shows that performance improvements can be gained even from partially vectorizing loop heavy code if the operation reduction is larger than the overhead introduced from switching between vectorized and unvectorized code. Furthermore, one thing not considered, but mentioned during the presentation was that division, modulo, and multiplication can be conducted utilizing shift operations when they are powers of two (which are the majority of the division, multiplication, and modulo operations in the code) meaning additional vectorization improvements to the csr to bcsr code can still be applied.

References

[1] Artifact evaluation reproduction for "sparsetransx: Efficient training of translation-based knowledge graph embeddings using sparse matrix operations", mlsys 2025. URL: https://github.com/OnixHoque/sptransx-mlsys2025-reproduce/.

[2] csr_array - scipy v1.16.1 manual. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_array.html#scipy.sparse.csr_array.

[3] Intel® intrinsics guide. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX_ALL.

[4] mmwrite - scipy v1.16.1 manual. URL: https://docs.scipy.org/doc/scipy-1.16.1/reference/generated/scipy.io.mmwrite.html.

[5] numpy.savetxt - numpy v2.3 manual. URL: https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html.

[6] Smat: (s)parse (ma)trix matrix (t)ensor core-accelerated library. URL: https://github.com/spcl/smat.

[7] Md Saidul Hoque Anik and Ariful Azad. Sparsetransx: Efficient training of translation-based knowledge graph embeddings using sparse matrix operations. In *Eighth Conference on Machine Learning and Systems*, 2025. URL: https://openreview.net/forum?id=73tG7ZDSBT.

[8] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 2787–2795, Red Hook, NY, USA, 2013. Curran Associates Inc.

[9] GeeksforGeeks. C program to read content of a file, Jul 2025. URL: https://www.geeksforgeeks.org/c/c-program-to-read-contents-of-whole-file/.

[10] GeeksforGeeks. Removing trailing newline character from fgets() input, Aug 2025. URL: https://www.geeksforgeeks.org/dsa/removing-trailing-newline-character-from-fgets-input/.

[11] Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. High performance unstructured spmm computation using tensor cores, 2024. URL: https://arxiv.org/abs/2408.11551, arXiv:2408.11551.

[12] Naw Safrin Sattar, Hao Lu, and Feiyi Wang. Bcsr on gpu: A way forward extreme-scale graph processing on accelerator-enabled frontier supercomputer. In *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '24, page 280–289. IEEE Press, 2025. doi:10.1109/SCW63240.2024.00044.

[13] W3Schools. C read files. URL: https://www.w3schools.com/c/c_files_read.php.

[14] W3Schools. C write to files. URL: https://www.w3schools.com/c/c_files_write.php.

[15] W3Schools. Python file write. URL: https://www.w3schools.com/python/python_file_write.asp.

[16] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020. URL: https://arxiv.org/abs/1909.01315, arXiv:1909.01315.

APPENDIX

Code Repository

The repository with all the code I wrote within the SMaT reproducibility code [6] and SpTransX reproducibility code [1] (as well as time data for SpMM operations for transe, transh, and transr) can be found at https://github.com/jshin60/CSC290FPCode

CSR to BCSR data

Original SMaT Vtune Data

Run	Time	Cycles	Instructions	DRAM Bound
1	105.656s	278,982,144,000	202,153,728,000	2.7%
2	106.206s	299,548,032,000	201,387,648,000	3.2%
3	101.135s	556,372,992,000	478,942,464,000	3.7%
4	99.444s	169,572,480,000	129,526,656,000	1.8%
5	96.645s	2,905,728,000	1,362,816,000	3.3%

Vectorized SMaT Vtune Data

Run	Time	Cycles	Instructions	DRAM Bound
1	71.760s	169,593,984,000	136,273,536,000	4.7%
2	64.484s	844,032,000	551,040,000	12.9%
3	60.610s	230,684,160,000	211,817,088,000	18.5%
4	56.204s	3,249,792,000	3,448,704,000	9.0%
5	56.204s	3,249,792,000	3,448,704,000	2.3%
6	51.793s	2,593,920,000	1,325,184,000	9.7%

Vectorized SMaT Vtune Data Without Loop Unrolling and Software Pipelining

Run	Time	Cycles	Instructions	DRAM Bound
1	62.481s	13,047,552,000	8,781,696,000	25.1%
2	67.384s	272,426,112,000	265,730,304,000	16.4%
3	53.005s	61,541,760,000	47,343,744,000	32.5%
4	53.034s	170,083,200,000	163,511,040,000	16.4%
5	56.204s	3,249,792,000	3,448,704,000	2.3%
6	46.119s	8,147,328,000	6,795,264,000	15.2%

SMaT SpTransX SpMM Data

MMA-CBT Kernel Data

Run	Model	type	Time (ms)	TFLOP/s
1	Transe	adj	184.327	0.003
2	Transe	adj	196.716	0.003
3	Transe	adj	183.917	0.003
4	Transe	adj	181.821	0.003
5	Transe	adj	185.179	0.003
1	Transe	adjneg	210.492	0.003
2	Transe	adjneg	187.642	0.003
3	Transe	adjneg	186.839	0.003
4	Transe	adjneg	186.572	0.003
5	Transe	adjneg	187.165	0.003
1	Transh	adj	5.920	0.006
2	Transh	adj	3.284	0.010
3	Transh	adj	3.293	0.010
4	Transh	adj	4.356	0.007
5	Transh	adj	3.754	0.009
1	Transh	adjneg	4.872	0.007
2	Transh	adjneg	4.516	0.007
3	Transh	adjneg	3.741	0.009
4	Transh	adjneg	3.579	0.009
5	Transh	adjneg	2.980	0.011
1	Transr	adj	6.891	0.009
2	Transr	adj	6.074	0.011
3	Transr	adj	5.079	0.013
4	Transr	adj	5.206	0.013
5	Transr	adj	6.137	0.011
1	Transr	adjneg	6.927	0.010
2	Transr	adjneg	5.699	0.012
3	Transr	adjneg	5.443	0.012
4	Transr	adjneg	4.596	0.014
5	Transr	adjneg	5.240	0.013