Element-Wise Multiplication Kernel

By Morgan Cupp
12/30/2020

Table of Contents

1. Relevant Background Information

This project was completed using the CUDA-lite programming model and HammerBlade Manycore architecture. Sufficient details will be provided to understand the project and code.

1.1 CUDA-lite Review

CUDA-lite is a programming model designed to simplify programming many-core GPUs. Specific core IDs can be identified in software to distribute parallel computations across multiple cores. CUDA-lite requires separate code for a host and kernel. The host code executes once on a single core and prepares the kernel code for execution. The kernel code executes on several cores in parallel. The cores execute the same kernel code but operate on different subsets of a dataset.

1.2 HammerBlade Review

The HammerBlade Manycore architecture contains cores which execute the kernel code. It is a 2-dimensional network of tiles. Each tile contains a RISC-V core, instruction cache, and data memory. Tiles are able to communicate with each other and the host. A 2-dimensional collection of tiles that execute the same program is called a tile group. By varying tile group dimensions, users can select how many tiles execute in parallel. Tile grids consist of multiple tile groups. For this project, a single tile group is used.

1.3 Element-Wise Multiplication Review and Applications

This kernel performs element-wise multiplication. An example is in **Figure 1**. The inputs are two matrices with equal dimensions. Elements with matching indices are multiplied and stored at the same indices in an output matrix.

0	0	**1**
3	**8**	0
0	5	0

X

0	0	**7**
0	**2**	0
1	0	0

=

0	0	**7**
0	**16**	0
0	0	0

Figure 1. Element-wise multiplication example

Element-wise multiplication has many real-world applications. It is built into programming languages such as MATLAB, Python, C++, Wolfram Language, and more. It is used in lossy compression algorithms like JPEG, neural network architecture design, signal processing, and cryptography. Because element-wise multiplication is so prevalent, making it faster is valuable.

2. Code Walkthrough

The code parallelizes element-wise multiplication across multiple cores. Specifically, it multiplies matrices stored in sparse COO format. Sparse matrices contain mostly zeros, and exploiting this can improve performance. Key differences from previous kernels will be the primary focus. The full code and its detailed comments can be found on the "mmc274" branch of the "cornell-zhang/hb-sparse-kernels" GitHub repository.

2.1 Host Code: Overview

At a high-level, the host code is similar to previous kernels. Lines 4-7 define properties of the sparse input matrices. Lines 12-48 define the sparse_mat_t struct and helper functions to retrieve its data. Lines 54-221 define additional helper functions for creating and freeing sparse_mat_t structs, copying matrices to and from the device, calculating reference outputs, and verifying kernel outputs.

When a test runs, lines 233-326 create two sparse input matrices in COO format. Line 329 creates a reference output matrix on the host that is known to be correct. Lines 331-336 create an empty output matrix of the appropriate size that will be copied to the device. Lines 361-387 initialize the device and copy the input and blank output matrices to it. Lines 389-416 specify the grid and tile group dimensions. Lines 421-436 enqueue and execute the kernel code, causing it to run on the desired tile group. Lines 441-451 retrieve the output matrix from the device and verify its correctness. Finally, lines 456-469 freeze the tiles and free all matrix structs.

2.2 Host Code: Key Changes

The first key change is on lines 14-32. Previous kernels stored sparse matrices in CSR format. The element-wise kernels store them in coordinate format, or "COO". Matrices in COO format are stored as three arrays: nonzero data, nonzero rows, and nonzero columns. The nonzero data array contains every nonzero value in the sparse matrix. The nonzero row and column arrays contain the indices of each nonzero value at the corresponding index in the nonzero data array. **Figure 2** shows an example.

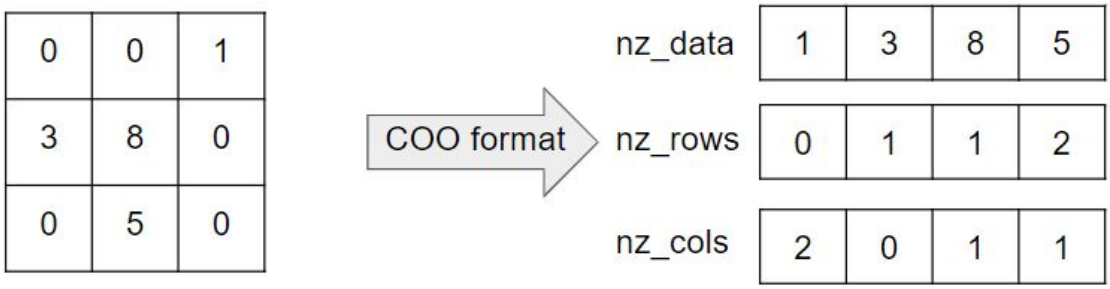


Figure 2. A sparse matrix's equivalent COO representation

The reference function on lines 156-188 was modified to take COO inputs and generate a COO output. Consider an *m x n* input matrix. Unlike with dense matrix format, the sparse_mat_t

arrays do not have size *m* * *n*. Instead, the arrays have size equal to the number of nonzero values in the output matrix. For example, the matrix in **Figure 2** would have arrays of size nine in dense format. In COO, they have size four. This poses a challenge since the reference function does not initially know what size to make the output arrays. Thus, it first creates temporary, worst-case *m x n* arrays to fill while the multiplications occur. Once complete, the number of nonzero values is known, and the values in the temporary arrays are copied into output arrays with the appropriate size. This solution does not work in the kernel and is discussed in section 2.4.

The check function on lines 191-221 also had to be modified to take COO inputs. It iterates through all possible indices in the *m x n* matrices being compared. For every coordinate pair, the two input matrices are searched. If a matrix contains indices matching a given coordinate pair, then the matrix contains a nonzero value at those indices. If there is no match, then the matrix contains a zero at those indices.

To run tests, sparse inputs in COO format are needed. The code on lines 233-326 randomly generates two sparse matrices in COO format with dimensions and density specified by the user. Lines 241-249 initialize a random number generator. The number of nonzero data values in each matrix is calculated using the dimensions and density. Lines 251-269 populate the nonzero data array for each matrix with random float values. Lines 272-322 generate random integer indices for each nonzero data value. To avoid duplicate indices, arrays tracking which indices have been used are maintained. First, random row and column indices are generated. If they are valid, then they are used. If they are invalid, then indices starting at (0,0) are searched until valid indices are found. Lines 325-326 then create two sparse_mat_t structs.

2.3 Kernel Code: Overview

The high-level function of the kernel code is straightforward. It does an element-wise multiplication of two sparse_mat_t structs in COO format and outputs a COO matrix of the appropriate size. The multiplication is parallelized across all cores in the specified tile group.

2.4 Kernel Code: Key Differences and Challenges

The primary difference between this kernel and previous kernels is both the inputs and output are in COO format. This posed two main challenges whose solutions are in section 3.1.

First, the parallelization of work had to be done in a slightly different way. Multiplications only occur when both inputs have a nonzero value at the same indices; otherwise, the output will have a zero at those indices. This means the sparse_mat_t struct will not contain those indices at all. Thus, a core only does a multiplication when it finds matching indices.

The second challenge is more difficult to address. Consider the example in **Figure 3**. In previous kernels, the outputs were stored in dense format. This means an *m x n* output matrix would be stored in an array of size *m*n*, and every value in the output has a unique location based on its indices. Meanwhile, order does not matter in the COO representation. Any nonzero value can be stored in any index of the arrays as long as all nonzero values are present. Thus, every time a core computes a nonzero value, it needs a way to determine where to write this value. This is further complicated by the fact that in COO format, the array size is equal to the number of nonzero values. Thus, not a single spot in the output arrays can be wasted.

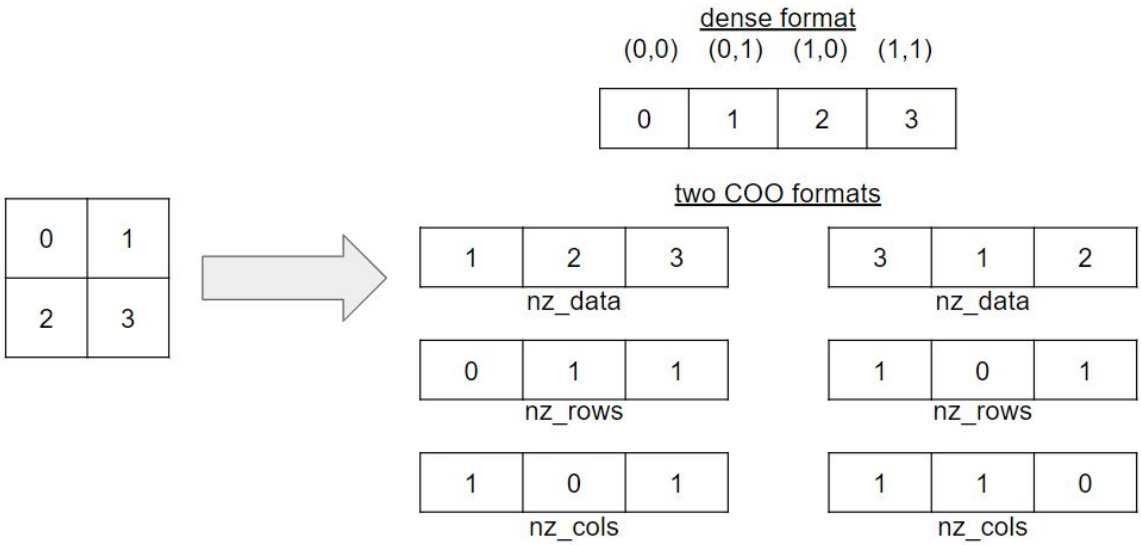


Figure 3. Representations for the same matrix in dense and COO format

3. Aside: First Kernel Attempt

The first kernel tried to resolve the second challenge mentioned above using temporary arrays. Each core initialized temporary data, row, and column arrays on its scratchpad. As each core did multiplications, it stored the values in the temporary arrays. Once the tiles were done with their computations, they could write the output arrays. This would now be easier since each tile knows how many nonzero values it must write to the output.

This worked for small or very sparse inputs, because very few values needed to be stored in the scratchpad. However, inputs can be arbitrarily large or dense. In this case, scratchpad overflow occurs. This exact approach was abandoned, but it was a step in the right direction.

4. Unoptimized Kernel Summary

The first working kernel performed element-wise multiplication successfully on inputs of any density and dimension. It solved the aforementioned challenges and provided a reliable foundation to work with. However, it was very inefficient and left a lot of room for improvement. Note that the code on GitHub is not for this kernel version.

4.1 Implementation Details

This kernel will be explained using the example in **Figure 4**. Suppose a 2x2 tile group is used. This means there will be four tiles.

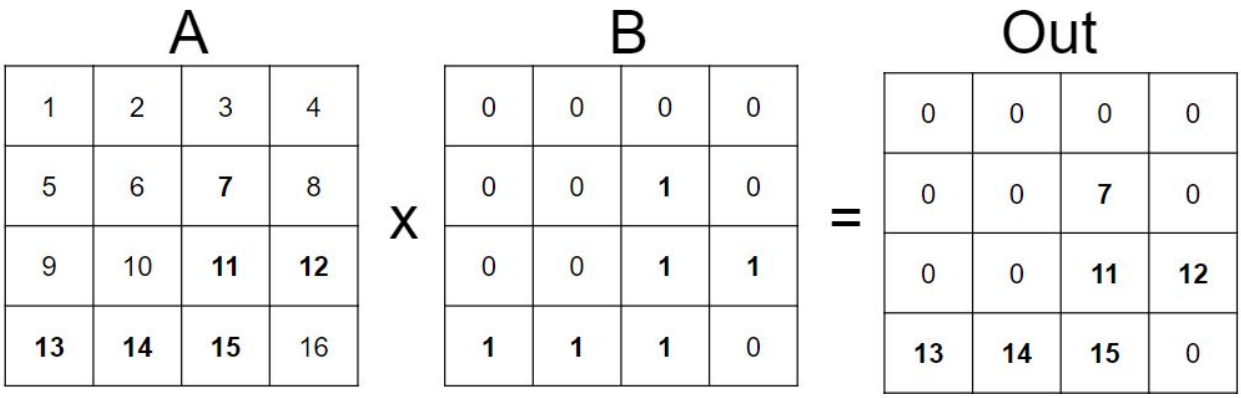


Figure 4. Element-wise multiplication example

The kernel code assigns a subset of input A to each tile. **Figure 5** shows which values get assigned to which tile. Note that the partitioning is made as equal as possible.

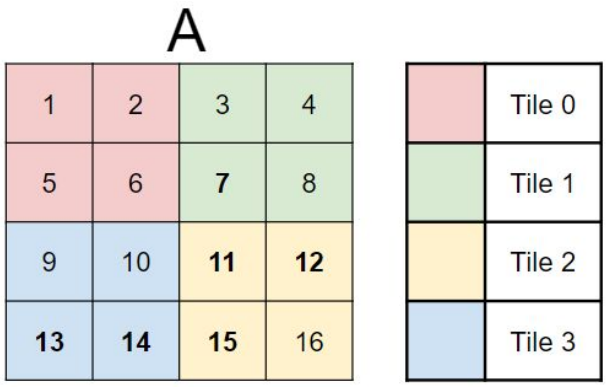


Figure 5. Partitioning of matrix A among different cores

Each core only looks at its assigned subset. For each nonzero value in A, the row and column indices of that value are retrieved. Next, the indices of all nonzero values in B are checked for a match with the A value. If matching indices are found, then these values will eventually be multiplied and stored in the output. Each tile uses this strategy to count how many nonzero values it will have to write to the output arrays. **Figure 6** shows matching indices found by each tile. Tile 0 will calculate zero nonzero values, tile 1 will calculate one, tile 2 will calculate three, and tile 4 will calculate two.

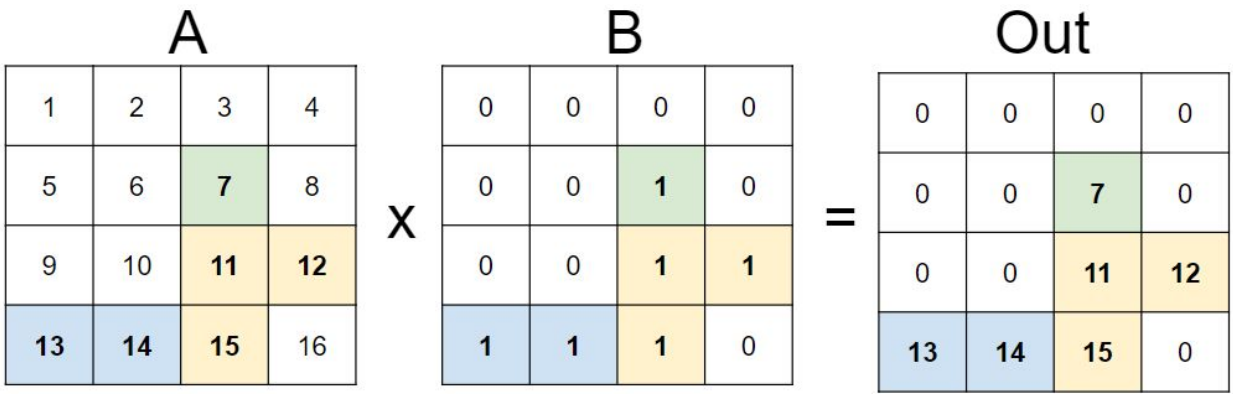


Figure 6. Matching indices found by each tile

Once every tile knows how many nonzero values it will compute, it must determine where to write the output arrays. The output arrays will each have size six, because there are six nonzero values in the output. Each tile computes the first index it will write to in the output arrays by summing the number of nonzero values computed by all tiles with lesser tile IDs. This is shown in **Figure 7**. The calculations are as follows:

- tile 0 starting index = n/a because it computes zero nonzero values
- tile 1 starting index = tile 0 # nonzeros = 0
- tile 2 starting index = (tile 0 # nonzeros) + (tile 1 # nonzeros) = 0 + 1 = 1
- tile 3 starting index = (tile 0 # nonzeros) + (tile 1 # nonzeros) + (tile 2 # nonzeros) = 0 + 1 + 3 = 4

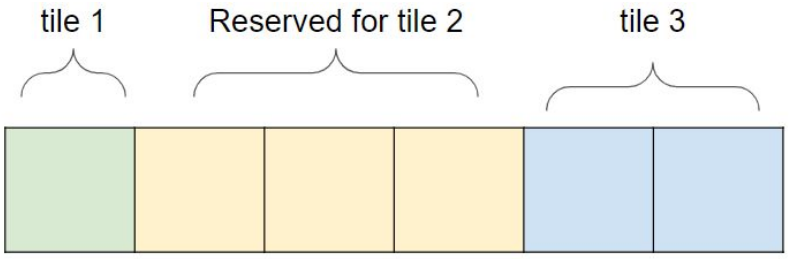


Figure 7. Locations that each tile will write to the output arrays

Now each tile knows exactly where to write in the output arrays. Each tile does the element-wise multiplications by again selecting a nonzero element in A, searching for a nonzero element in B with matching indices, and multiplying them if there is a match. Values and indices are written to the corresponding output arrays in each tile's reserved section.

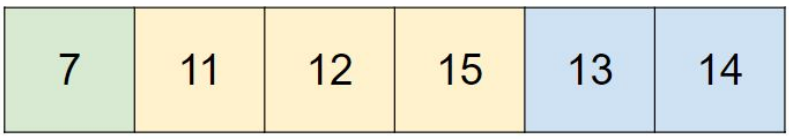


Figure 8. Final state of the output array containing nonzero data values

4.2 Data Summary

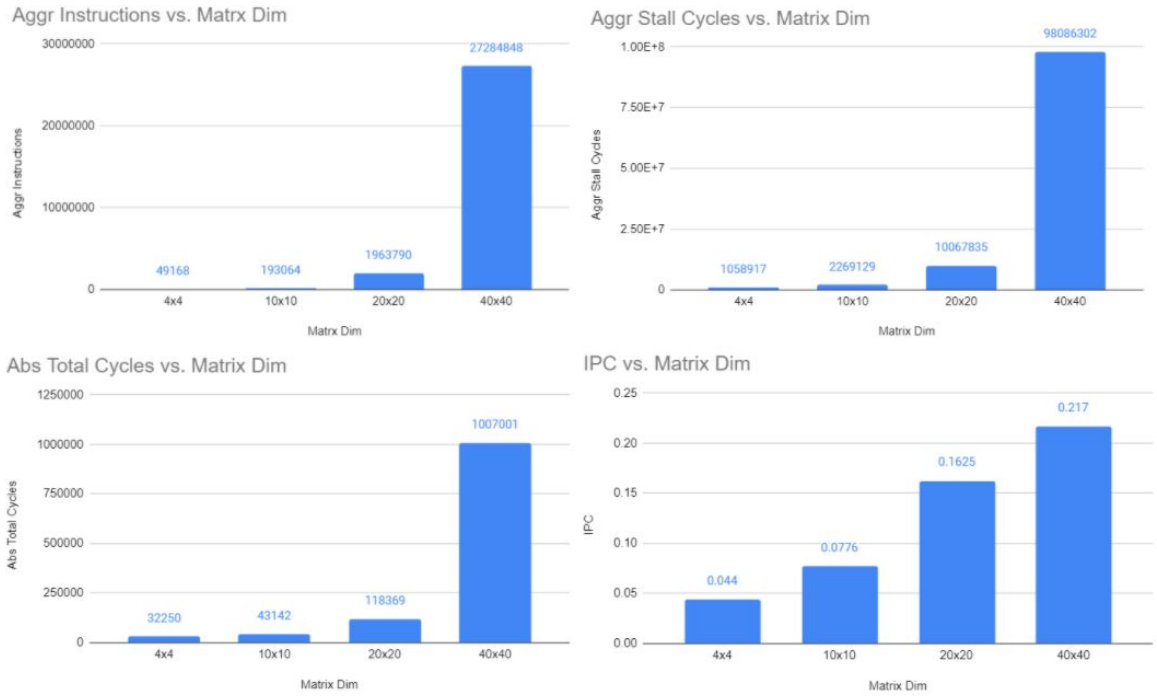


Figure 9. Unoptimized kernel data for a 16x8 tile group multiplying input matrices with 75% density

The results from the first tests are shown above. They were run on a 16x8 tile group. The input matrices had 75% density with varying dimensions. The rapid increase in aggregate instructions and absolute total cycles suggests the code becomes inefficient as inputs contain more nonzero values. The increase in stall cycles suggests tiles spend a lot of time waiting for each other and may have an uneven work distribution.

4.3 Room for Improvement

This first kernel had several areas for improvement. Even in the ideal case, each multiplication requires nine data accesses: six to read input arrays of each input matrix and three to write the output arrays. This means the best-case operational intensity is one FLOP per nine data accesses, or 0.11. This is already quite low, so optimization is important.

The greatest issue is that for every nonzero value in A, B must be searched for matching indices. When matching indices are not found, this searching is wasteful. Furthermore, the unoptimized kernel sequentially searches through all values in B from start to finish. This is because in COO format, values are stored in an arbitrary order. Thus, a value with matching indices in B may be located at the very last index in B's arrays. Because each nonzero value in A requires a separate search through all values in B, redundant data accesses also occur. Clearly, a better way to search for matching indices would drastically reduce wasted cycles and memory accesses.

5. Optimized Kernel Summary

The focus of the optimized kernel was to speed up the search for matching indices. This searching process occurs twice: once to count the number of nonzero values that will be computed, and once to actually do the multiplications. By sorting the inputs in the host code and using binary search in the kernel code, matching indices are found much faster. These changes significantly improved performance compared to the unoptimized kernel. The code in GitHub is for this kernel version.

5.1 Implementation Details

Consider an *m x n* matrix. For a nonzero value *v* with indices (*row*, *col*), define its absolute index as *row*n + col*. Lines 341-358 of the host code sort the nonzero values in B from lowest to highest absolute index using the insertion sort algorithm. **Figure 10** shows a 4x4, 100% density matrix in the host code before and after sorting. Note that data, row, and column values at the same index must be at the same index after sorting to maintain correct COO representation.

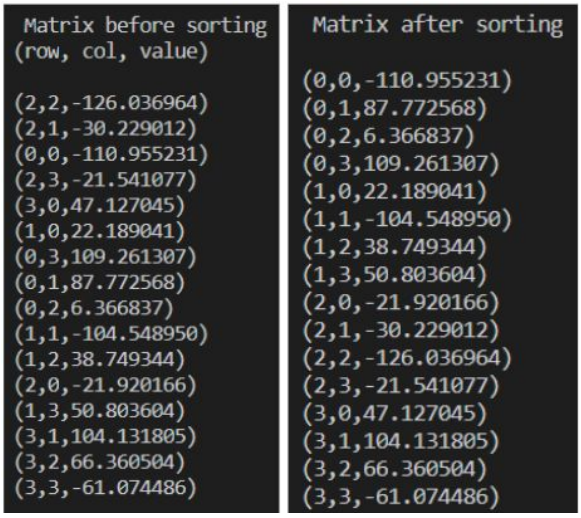


Figure 10. Randomly generated matrix before (left) and after (right) insertion sort

Now that matrix B is sorted, searching for matching indices is much faster in the kernel since binary search can be used. **Figure 11** shows the binary search algorithm. Binary search is advantageous because it does not check every B value's indices. Instead, it quickly narrows down where the value with matching indices either is or would be, and terminates the search at this point. As a result, far fewer memory accesses occur. Furthermore, the time to find the matching indices is approximately equal regardless of what the indices are. This creates a more equal work distribution.

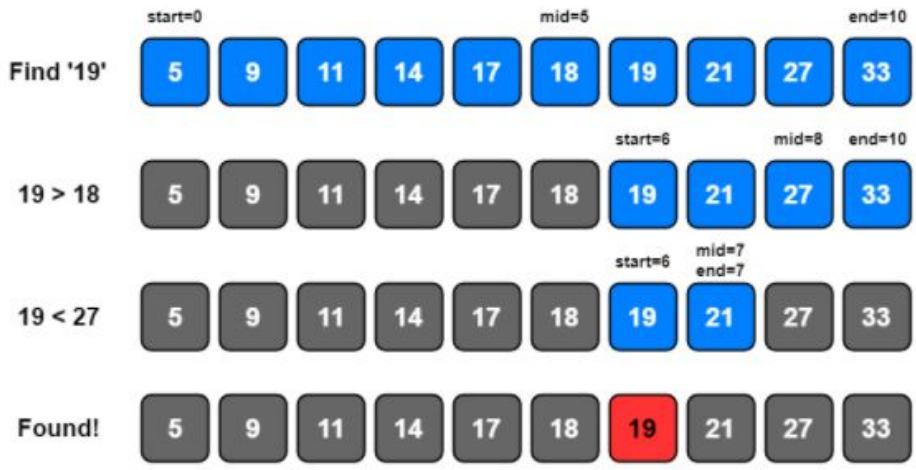


Figure 11. Binary search algorithm
Source: https://emre.me/coding-patterns/modified-binary-search/

The code on GitHub is for this optimized version. The key steps of the kernel may be summarized as follows:

1. line 92: assign each tile to a subset of A's nonzero values
2. lines 93-102: for each nonzero value in A, search for a value in B with matching indices using binary search
3. lines 103-104: if matching indices are found, increment the number of nonzero values this tile will compute
4. line 108: wait for all other tiles to finish computing their number of nonzero values
5. lines 111-118: determine where each tile will write to the output arrays
6. lines 121-141: following similar algorithm to counting nonzero values, perform the element-wise multiplications and write the results to the output arrays
7. line 143: wait for all tiles to finish computing

5.2 Data Summary

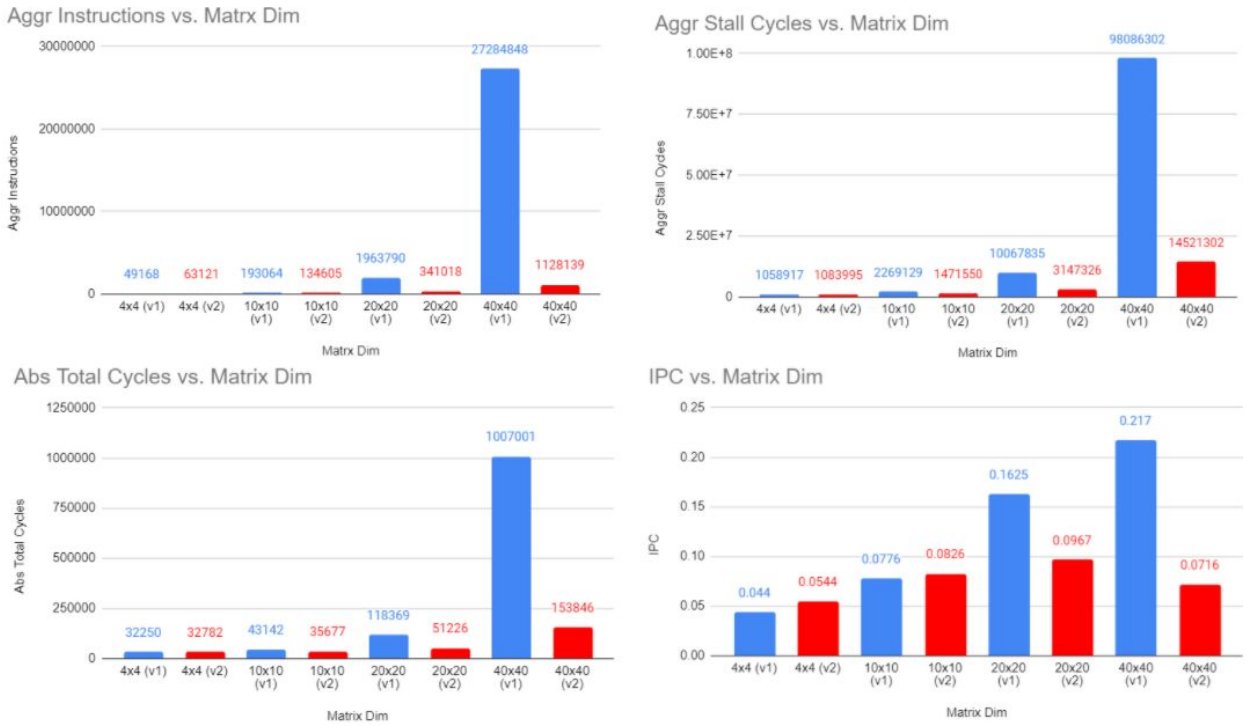


Figure 12. Unoptimized (v1) vs. optimized (v2) kernel data for a 16x8 tile group multiplying input matrices with 75% density

As **Figure 12** shows, the new searching strategy drastically improved performance. For the 40x40 inputs, aggregate instructions, aggregate stall cycles, and absolute total cycles improved by 96%, 85%, and 85%, respectively. The performance improvement increases as the inputs contain more nonzero values. This makes sense, because there are more binary searches being done in place of the unoptimized search algorithm. Less searching reduces the instruction count and cycles necessary to finish the multiplications. Stalling decreases significantly since each tile takes less time to finish, meaning other tiles do not have to wait as long. Moreover, binary search helps evenly distribute computations as mentioned previously, reducing stalling.

Figure 13 shows the statistics for various tile group dimensions on 1000x100 inputs with 1% density. This is a more realistic test that takes advantage of the sparse COO representation. The performance generally improves as tile group dimensions increase, because there are more tiles available to parallelize the computations.

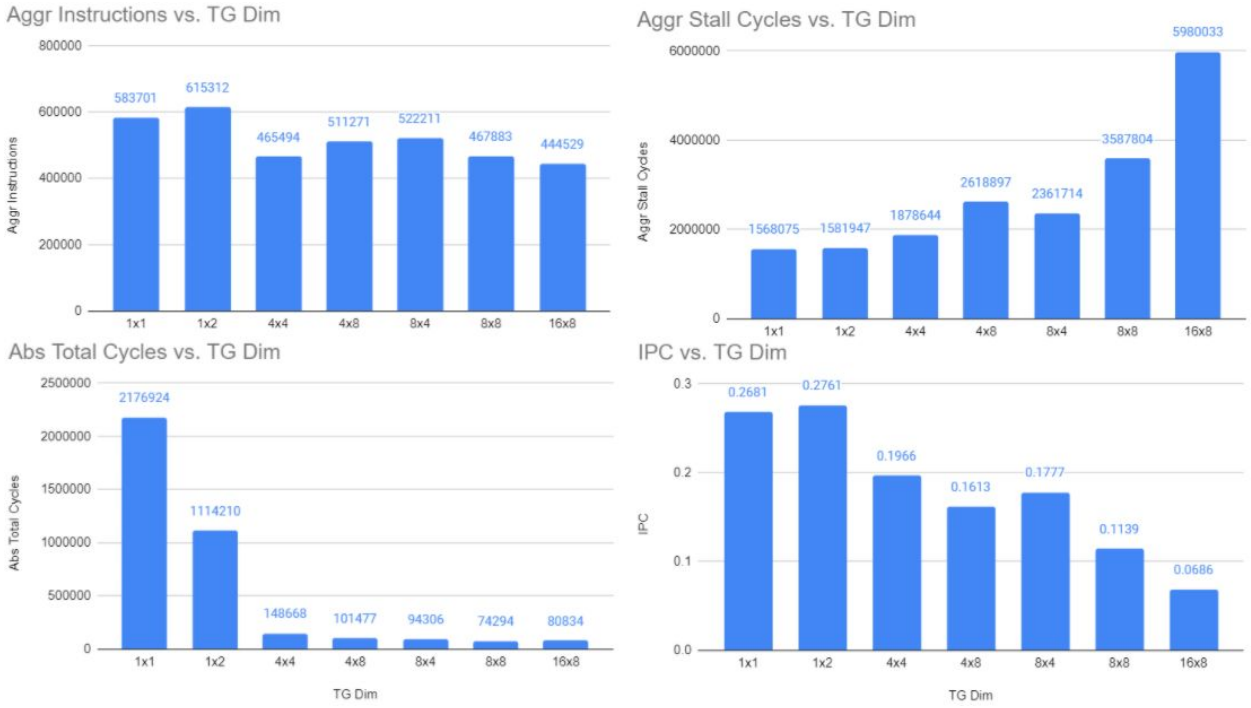


Figure 13. Optimized kernel data for various tile groups multiplying 1000x100 input matrices with 1% density

5.3 Room for Improvement

Although this optimized significantly improved performance, further optimizations could still be done. Searching for matching indices still consumes many cycles and could potentially be done even better. Counting the number of nonzero outputs prior to doing the actual multiplications is also a bit redundant, because the searching occurs twice. Perhaps the array indexes of values with matching indices could be stored while counting nonzero outputs so that searching only occurs once. Perhaps this method could be replaced altogether with a new way to determine where each tile should write to the output matrix. Lastly, some tiles do many more multiplications than others depending on how many matches each tile finds. Distributing the actual multiplications more evenly across all tiles would be beneficial.

6. Stats File Bug Description

When testing the optimized kernel, the stats in **Figure 14** were generated in the manycore_stats.log. Every test used the same inputs. This seemed like it might be incorrect, because there is no reason for the aggregate instruction count to decrease that much for the 16x8 tile group.

Figure 14. Bar graph showing stats irregularity for the 16x8 tile group

When investigating this issue, it was determined that the stats file said some tiles executed zero instructions. To ensure these tiles were in fact executing, a print statement was added to the kernel. After adding the print statement, the stats log said all of the tiles were executing instructions as in **Figure 15**. The data also seemed reasonable. Thus, the issue seemed to be with the stats tracking, not the kernel code.

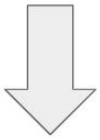
```
Per-Tile Timing Stats
Relative Tile Coordinate (Y,X)            Instructions
============================================================
------------------------------------------------------------
0 , 0                                             2581
0 , 1                                             2429
0 , 2                                             2433
0 , 3                                             2422
```

Figure 15. Stats file section containing per-tile instruction count after adding a print statement

6.1 Temporary Solution

```
int num_threads = num_threads_x * num_threads_y;
int tid = tid_y * num_threads_x + tid_x;

bsg_cuda_print_stat_kernel_start();
```



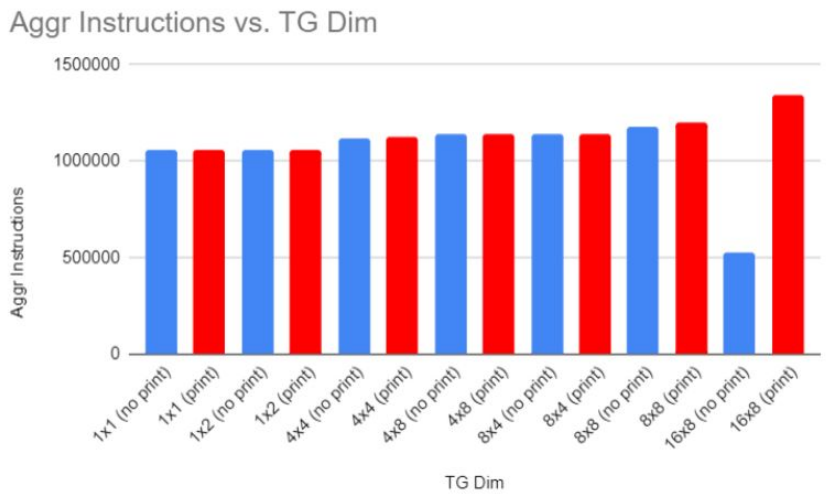
```
int num_threads = num_threads_x * num_threads_y;
int tid = tid_y * num_threads_x + tid_x;
bsg_printf("debug");
bsg_cuda_print_stat_kernel_start();
```

Figure 16. Temporary code change needed for correct stats

Although it has not been determined why, including the print statement in **Figure 16** seems to make the stats file accurate. The print statement can contain anything; it just needs to be present in the kernel. It should ideally be added before the bsg_cuda_print_stat_kernel_start() so that it does not get included in the actual stats collection.

6.2 Data Demonstrating Bug

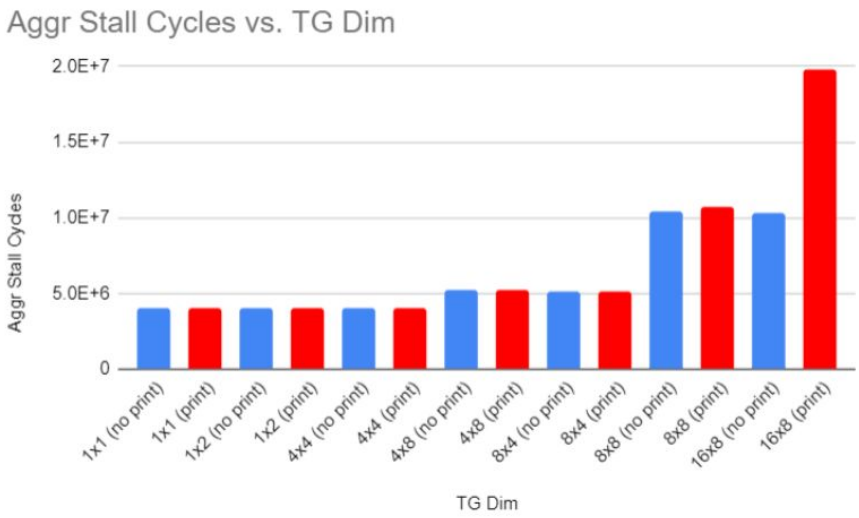
Several tests were run to see how much the print statement impacts the data. For all of these tests, the same 128x128 input matrices with 10% density were used. The output contained 232 nonzero values, so 232 fmul operations were expected. Only the tile group dimensions and presence of a print statement changed for each test.



TG Dim	Aggr Instructions
1x1 (no print)	1055234
1x1 (print)	1057331
1x2 (no print)	1055597
1x2 (print)	1057691
4x4 (no print)	1121503
4x4 (print)	1123952
4x8 (no print)	1141740
4x8 (print)	1144050
8x4 (no print)	1139194
8x4 (print)	1139798
8x8 (no print)	1175193
8x8 (print)	1200026
16x8 (no print)	525152
16x8 (print)	1340483

16x8 percent error = 61%

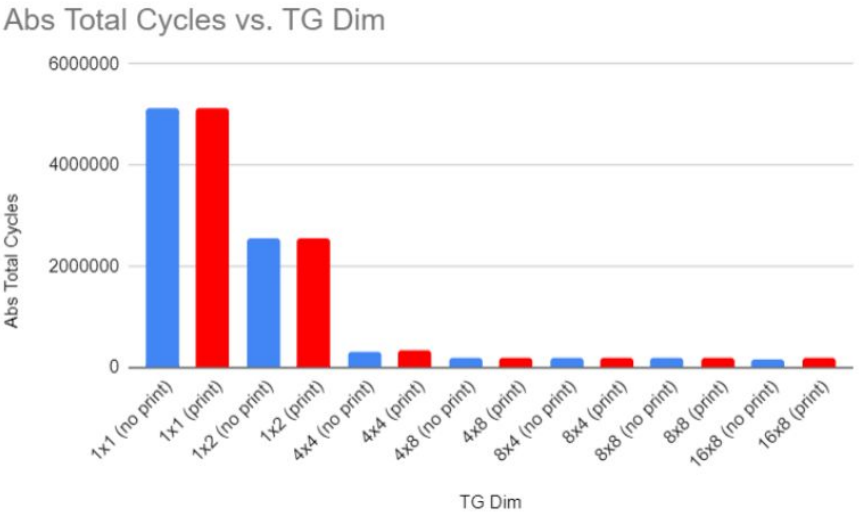
Figure 17. Effect of print statement on aggregate instructions



TG Dim	Aggr Stall Cycles
1x1 (no print)	4035648
1x1 (print)	4028126
1x2 (no print)	4038747
1x2 (print)	4032068
4x4 (no print)	4075781
4x4 (print)	4087183
4x8 (no print)	5194272
4x8 (print)	5216185
8x4 (no print)	5147255
8x4 (print)	5181960
8x8 (no print)	10436023
8x8 (print)	10680761
16x8 (no print)	10323030
16x8 (print)	19828259

16x8 percent error = 48%

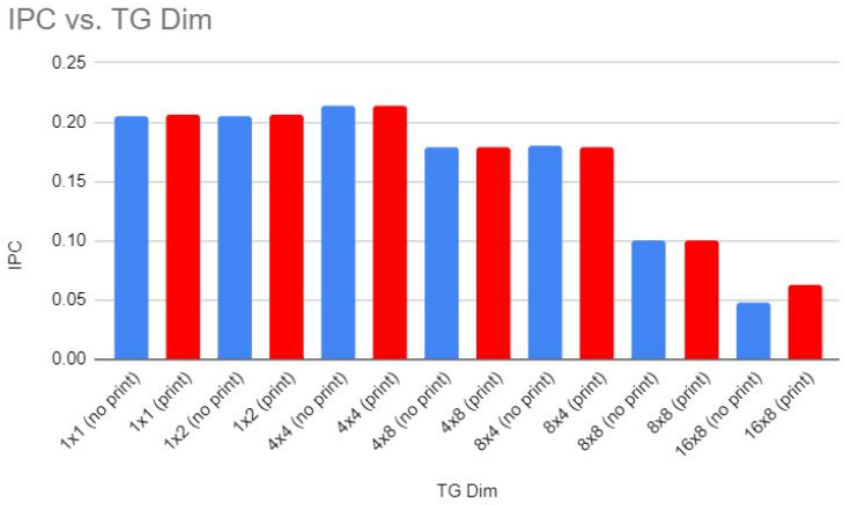
Figure 18. Effect of print statement on aggregate stall cycles



TG Dim	Abs Total Cycles
1x1 (no print)	5138188
1x1 (print)	5132913
1x2 (no print)	2570885
1x2 (print)	2568773
4x4 (no print)	328437
4x4 (print)	329521
4x8 (no print)	200952
4x8 (print)	202873
8x4 (no print)	199456
8x4 (print)	201637
8x8 (no print)	188601
8x8 (print)	194438
16x8 (no print)	177419
16x8 (print)	190081

16x8 percent error = 7%

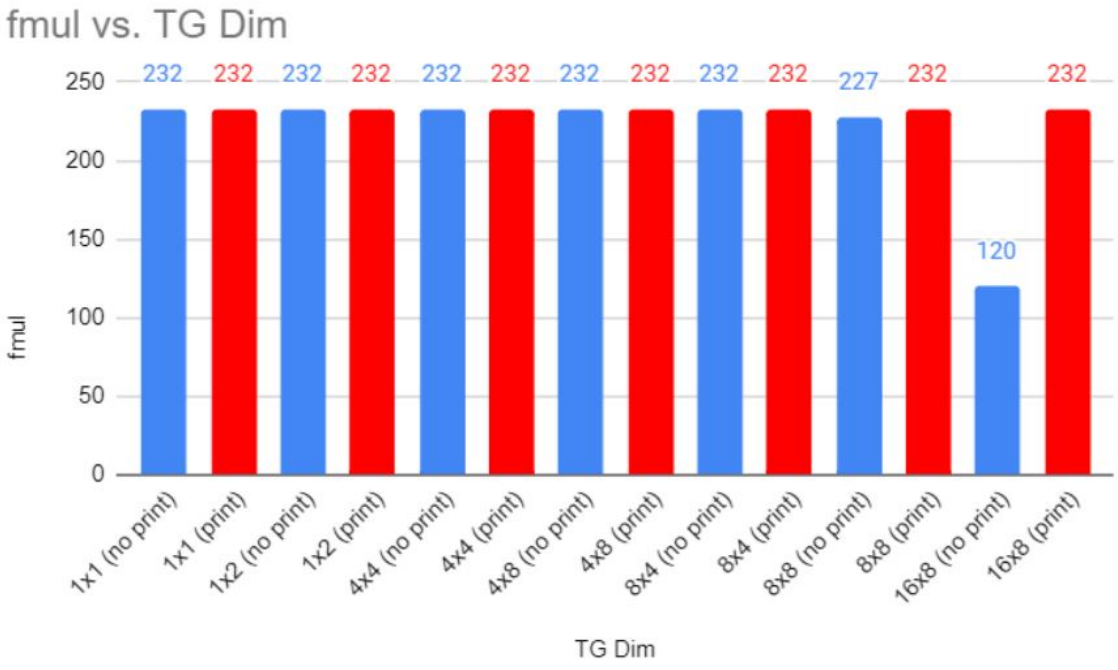
Figure 19. Effect of print statement on absolute total cycles



TG Dim	IPC
1x1 (no print)	0.2054
1x1 (print)	0.206
1x2 (no print)	0.2053
1x2 (print)	0.2059
4x4 (no print)	0.2138
4x4 (print)	0.2136
4x8 (no print)	0.1788
4x8 (print)	0.1784
8x4 (no print)	0.1798
8x4 (print)	0.1788
8x8 (no print)	0.1008
8x8 (print)	0.1005
16x8 (no print)	0.0483
16x8 (print)	0.0631

16x8 percent error = 23%

Figure 20. Effect of print statement on IPC



16x8 percent error = 48%

Figure 21. Effect of print statement on fmul count

Figures 17-21 show that the error only becomes significant for the 16x8 tile group. For every statistic, not having a print statement makes the values in the stats file lower than expected. The print statement data seems accurate since the red bars better match the expected trends in

each graph. The fmul data is especially revealing since the fmul count should be 232 every time. The 16x8 tile group was very inaccurate without the print statement, counting only 120 fmuls.