GPU Sparse Matrix Multiplication

Xingyou Song, *Nitin Manivasagan, †Manar Safi ‡

May 8, 2018

1 Introduction

Our code can be found on: `https://github.com/Srizzle/GPU-Sparse-Matrix-Multiply-CS-267`.

GPU-assisted sparse matrix operations have been well-known for a long time. In this report, we discuss the approaches of many of the well-known algorithms and perform experiments to benchmark and compare their performances on both random matrix-matrix multiplication.

To avoid confusion, are separate naming conventions for each case when dealing with matrix-multiplication. These are:

- SpMV - (Sparse Matrix) $\times$ (Dense Vector) - Multiplications of form $M \cdot b$, where M is a sparse matrix, and b is a 1-d vector with any set of non-zero entries.
- SpMM - (Sparse Matrix) $\times$ (Multiple Dense Vectors) - Multiplications of form $M \cdot N$, where M is a sparse matrix, and N is a matrix built by multiple dense vectors.
- SpGEMM (Sparse Matrix) $\times$ (Sparse Matrix) - Multiplications of form $M \cdot N$, where M, N are both sparse matrices.

Note that the libraries of interest, where each library may support some (or all) of these operations are:

- CUSP - Open Source wrapper library for open-source library Thrust: `https://github.com/cusplibrary/cusplibrary/tree/release/v0.6.0-rc` [1]
- CuSPARSE - Production-level library: `https://developer.nvidia.com/cusparse` [2]
- CuBLAS - Production-level library: `https://developer.nvidia.com/cublas`

*xsong@berkeley.edu
†nitinmani@berkeley.edu
‡safi.manar@berkeley.edu

[1] The official release (4.0 and 5.0's do not work with our code)

[2] Production libraries do not present the source code.

In the interest of time (as well as the fact that many open-source libraries are *extremely* error-prone and unstable), we also considered the following libraries, but did not fully implement them due to heavy installation errors: [3]

- bhSPARSE - Recent Modifications to original libraries: `https://github.com/bhSPARSE/bhSPARSE`
- OpenAI Blocksparse - Open Source library for special "block-sparse" matrices: `https://github.com/openai/blocksparse` [4]

To assist in readability in code, we used some wrapper libraries:

- CuBLAS - Cupy: `https://github.com/cupy/cupy`
- CuSPARSE - PyCulib: `https://github.com/numba/pyculib`

The wrapper libraries that we attempted to use, but obsolete or failed are:

- CuSPARSE wrapper - Julia: `https://github.com/JuliaGPU/CUSPARSE.jl`
- CuSPARSE wrapper - SkCUDA: `https://github.com/lebedov/scikit-cuda/tree/master/skcuda`

2 Algorithmic Methods

In this section we explain the common methods for sparse matrix multiplication, and link which methods to the correct libraries.

2.1 SpMV

From [BG08], there is an emphasis on the format representation of sparsity, which in turn is used to consider the appropriate GPU method. The goal is to compute $M \cdot b$. 3 methods are shown:

- Diagonal format (structure specific) - For matrices M whose elements are dominant on diagonals (close to the main diagonal), the matrix can represented by forming a data matrix (with zero-elements replaced as a "*" character), and an offset vector. The elements on the k-th column of the data matrix D are matched with the k-th element of the offset vector, which specifies the diagonal on M. The elements on the data matrix are also row-aligned (i.e. elements of the same row in M will also be in the same row in D).

[3] A large portion of our time was used in setting up different environments, because almost all open-source libraries are poorly documented, or have bugs.

[4] We attempted to install all variants and versions on PSU XSEDE, Amazon AWS with Nvidia Tesla K80 (p2x.large instance), Amazon AWS with Nvidia Tesla P100 (p3x.large instance), Google Cloud Computing API with Nvidia Tesla P100, with no success on any of them

Figure 1: Diagonal format example

$$\texttt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\texttt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \texttt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

For GPU computation, each thread t will be responsible for computing the t-th element on the output (i.e. dot product between row t in M with the input vector). Because of the row alignment, each thread will contiguously access data along the row, incrementing the final output with $+= M[t][i] * B[i]$ for all i.

- ELLPACK format - This format is a general format, under the assumption that each row has a maximum of Z non-zero elements in M. Intuitively, this then compacts M into a data matrix, where each row r of the data matrix is the list of non-zero elements of the row r in M. The r-th row in the index matrix lists the column-coordinates of the elements in data at row r with the elements in M.

Figure 2: ELLPACK

$$\texttt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\texttt{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \texttt{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

For GPU computation, the method is the same as the Diagonal format, where each thread is assigned to a row and computes the dot-product, but instead using the index matrix as a hash for locations (and therefore accesses the index matrix continguously)

The performance of this relies on the fact that overall, most of the rows all have Z elements (i.e. balanced numbers of elements on the rows), which load-balances the work performed on each thread.

- Coordinate format - The most widely used and well known format being the CSR (compressed sparse row), the matrix M is represented by ptr, indices, and data vectors which specify the offset on each row that has elements (with last element being the number of NNZ's), along with the distinct rows, and data elements, respectively. For

Figure 3: CSR example format

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \mathtt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \mathtt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

the SpMV operation, the "vector kernel" is introduced, where an entire thread block may be assigned to each matrix row, whose entries are then partitioned equally among the threads, with a final parallel reduce operation to compute the dot product between the matrix row and vector.

2.2 SpMM

In [DOB15], the algorithm is decomposed into 3 portions, assuming the input is A, B and the output is $C = A \cdot B$. We also assume A, B are expressed in CSR (Compressed Sparse Row) format. The 3 portions are then the "expand, sort, compress" (ESC) methods, similar to a Map-Reduce technique, as explained below.

Figure 4: Example of the 3 techniques, when C is generated from A, B which are in CSR format.

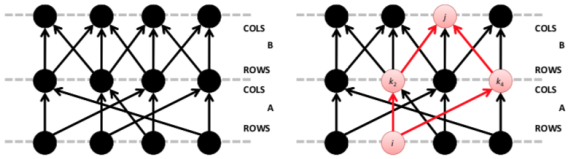
$$\hat{C} = \begin{bmatrix} (0,0,10) \\ (1,3,60) \\ (1,1,40) \\ (1,1,150) \\ (1,0,120) \\ (1,3,280) \\ (1,1,240) \\ (2,3,350) \\ (2,1,300) \\ (3,3,180) \\ (3,1,120) \end{bmatrix} \xrightarrow[\text{sort}]{} \begin{bmatrix} (0,0,10) \\ (1,0,120) \\ (1,1,40) \\ (1,1,150) \\ (1,1,240) \\ (1,3,60) \\ (1,3,280) \\ (2,1,300) \\ (2,3,350) \\ (3,1,120) \\ (3,3,180) \end{bmatrix} \xrightarrow[\text{compress}]{} \begin{bmatrix} (0,0,10) \\ (1,0,120) \\ (1,1,430) \\ (1,3,340) \\ (2,1,300) \\ (2,3,350) \\ (3,1,120) \\ (3,3,180) \end{bmatrix} = C.$$

- Expand: This generates all possible pairings (A_{ik}, B_{kj}) which are used to add to a position $C_{ij} += A_{ik} * B_{kj}$. The generation process can be viewed as forming all-pairs paths on a bipartite graph. More specifically, for a matrix M, the bottom side of the graph consists of vertices, each representing a row, while the top side's vertices

represent the columns, with a connection (a,b) if the element M_{ab} is non-zero. Then conjoining the bipartite graphs A, B makes it convenient to see that the total number of pairings (A_{ik}, B_{kj}) is the number of length-2 paths.

Figure 5: Example figure of expansion bipartite graphs

$$\mathtt{A} = \begin{bmatrix} x & 0 & x & 0 \\ 0 & x & 0 & x \\ 0 & x & x & 0 \\ x & 0 & 0 & x \end{bmatrix} \quad \text{and} \quad \mathtt{B} = \begin{bmatrix} x & x & 0 & 0 \\ x & x & x & 0 \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix}. \tag{6}$$



(a) Layered graph model of $A*B$.

(b) Paths contributing to an entry.

Fig. 7: Schematic of graph-based sparse matrix multiplication.

Thus, using out all length-2 paths corresponds to a "BFS"-type tracking on this path, from which well-known GPU graph traversal methods are used. In the GPU case, the best method would be for each thread to possess a starting node at the bottom side of the bipartite graph corresponding to A. A given core then selects T consecutive bottom vertices, and the total vertices are partitioned by cores. Each thread t then computes the "expansion" (i.e. all possible paths from its vertex v_t including the products associated with a path), using techniques found in [MGG12].

- Sort: This method sorts lexicographically each element (x, y, c) generated from before, where x, y corresponds to location (x, y) in C, and c is to be incremented to C_{xy}. This method is used to assist the compress portion, which is used to remove duplicate/increment (x, y) coordinate entries. Sorting is cited as the most computationally expensive configuration, and the algorithm to invoke derives from the work [MG11]. The baseline algorithm is an inspired GPU-based Radix Sort, found in Thrust's `stable_sort_by_key` algorithm. Here, the number of threads is proportional to the number of elements in the expansion, and the total elements is partitioned equally for each thread, (over same rows) with each thread sorting only its sub partition. All of the sorted sub partitions are then collected within the block shared memory, and then re-sorted within the entire block, which sorts an entire row.

- Compress: This section invokes the work found in [BDO12], and calls the Thrust function `reduce_by_key` algorithm, which for each row index i, the function collects all entries positioned in row i. The algorithm then performs "reduction" (i.e. combine consecutive terms with the same j index) and outputs the compressed format. However, since it is not known at runtime which positions (i,j) have very large duplicate entries, it would be highly inefficient to assign each thread to a position (i,j) on the GPU. Instead, a scan is implemented at first to check which duplicate positions exist, and

then partitions all duplicate entries into consecutive, fixed-length blocks for each thread to reduce, attempting to load balance.

3 Experiments

3.1 Code

Our code and scripts can be found on:
`https://github.com/Srizzle/GPU-Sparse-Matrix-Multiply-CS-267`. The scripts used for simulation are:

- CUSP: cusp.cu - `https://github.com/Srizzle/GPU-Sparse-Matrix-Multiply-CS-267/blob/master/cusp.cu`
- CUPY: cupy.py - `https://github.com/Srizzle/GPU-Sparse-Matrix-Multiply-CS-267/blob/master/cupy_script.py`
- Pyculib: pyculib_script.py - `https://github.com/Srizzle/GPU-Sparse-Matrix-Multiply-CS-267 blob/master/pyculib_script.py`
- CPU: cpu_script.py - `https://github.com/Srizzle/GPU-Sparse-Matrix-Multiply-CS-267/blob/master/cpu_script.py`

3.2 Hardware

We applied our experiments on the following hardware specified on Amazon AWS:

- Nvidia Tesla K80
- 11.75 ECUs, 4 vCPUs, 2.7 GHz, Intel Broadwell E5-2686v4, 61 GiB memory, EBS only, with Intel AVX and Turbo allowed.

Note that for this case of GPU, the memory bandwith is relatively larger than other GPU's of similar costs (480 GB/s vs 288 GB/s on a Tesla K40)

3.3 Erdos Reynii

We at first tested multiplication for $N \times N$ matrices, where each element was non-zero with probability p, where:

- $N \in \{2^4, ..., 2^{14}\}$
- $p \in \{0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5\}$

Note that for floats (where a float32 is a byte), when $N = 2^{14}$, $p = 0.01$, pN^2(float size = 1 byte) $\approx 2GB$, and at $N = 2^{14}, p = 0.5$, $pN^2 \cdot$(float size = 1 byte) $= 134GB$. Since the K80 RAM has 24GB of memory, this is reaches the order-magnitude of the GPU memory (i.e. for $N = 2^{15}$ sizes and higher, memory hardware limits are broken and disallowed at runtime).

3.4 Real World Data

Real world data matrices were also used in computation, found in `https://sparse.tamu.edu/`. We found the matrices that best fit the parameter (e.g. closest to a given sparsity p and size N) The matrices used were:

Figure 6: Real World Graph Names

	4000	8000	40000	80000	160000
0.0001	bcsstm24.mat	bcsstm38.mat	cond-mat-2005.mat	net4-1.mat	para-4.mat
0.001	c-24.mat	c-39.mat	bbmat.mat	consph.mat	pkustk14.mat
0.01	crystk01.mat	msc10848.mat	TSOPF_RS_b2383_c1.mat		
0.1	heart1.mat	human_gene2.mat			

4 Performance and Results

4.1 Erdos Reynii

The results for the Erdos Reynii model were completely out of expectation - for instance, cuBLAS did not increase in time complexity as we varied the size of the matrix - this is due to the fact that purely random matrices do not allow libraries to exploit their structure. Thus, we will not sketch those results in this report. However, we encourage the reader to see the results found in `https://github.com/Srizzle/GPU-Sparse-Matrix-Multiply-CS-267/tree/master/Erdos_Reynii_Experiments`. We saw from these experiments that cuBLAS did not show an increase in runtime regardless of $N = 2^{14}$ or $N = 2^5$, while cuSPARSE ran significantly slower as well.

4.2 Real World Data

We have the following colors over all graphs - (Purple: CUSP, Red: CuSPARSE, Blue: cuBLAS). The graphs can be found in the Appendix 6.

4.3 Analysis

From the data, the most interesting trend was generated by CSR x CSR. We see that cuSPARSE did perform better than CUSP in some particular areas (CSR x CSR at p = 0.001, or CSR x DenseVector early on), but did not outperform cuBLAS in any of the experiments, although was very close in small matrix sizes. Furthermore, we also see that at higher densities, cuSPARSE did indeed increase the most, as expected. cuBLAS had a slow increase in some parts.

For other cases (CSR $\times$ DenseVector or CSR $\times$ Dense Matrix), as expected, cuBLAS remained the most efficient (due to the dense portions)

In terms of runtime, note that for very sparse matrices ($p = 0.0001$), the general trend for all algorithms seems to be sublinear. However, as we increase the density, we can see a convex curvature, as well as suggesting a quadratic runtime. This is due to the fact that increased density eventually forces the NNZ number not to approximate $O(n)$, but towards $O(n^2)$, which is unacceptable and not intended for sparse multiplication.

5 Further Work

There is a variety of further directions that we wished to pursue, but in the interest of time, we did not.

5.1 Measuring Components

It would be interesting to measure the performance of e.g. the separate expansion, sort, and compress algorithms in SpMM, to verify the claimed higher runtime of the sorting section.

5.2 Memory Latency

Because different GPU architectures possess very different memory-load bandwidths, this suggests further testing for different GPUs (K80, P100, etc.) which may affect performance. There have been other variants of GPU matrix algorithms, found in

5.3 Expanding Memory

The size of the GPU RAM restricted our experiment sizes, as any larger dimensions would have overloaded the GPU RAM. Normally, the solutions to this type of problem either feature: 1. Use different GPU's with higher RAM 2. Distribute memory and workload across multiple GPU's.

5.4 Different Matrix Shapes

Note that from well known results in random matrix theory, a Erdos-Reynii matrix can have high probability of being singular [TV06]. Furthermore, the Erdos-Reynii format is generally not realistic, with many matrices in real world applications having generally high block sparsity. While we tested performance on widely used datasets, we did not use the other special sparse matrix formats (e.g. COO, ELLPACK, etc.), which may affect performance as well.

References

[BDO12] Nathan Bell, Steven Dalton, and Luke N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM J. Scientific Computing*, 34(4), 2012.

[BG08] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[BG11] Aydin Buluc and John R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *CoRR*, abs/1109.3739, 2011.

[DOB15] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Trans. Math. Softw.*, 41(4):25:1–25:20, October 2015.

[HGY07] James Demmel Hormozd Gahvari, Mark Hoemmen and Katherine Yelick. Benchmarking sparse matrix-vector multiply in five minutes. *SPEC Benchmark Workshop*, January 2007.

[HKS09] Linh Ha, Jens Krger, and Cludio T. Silva. Fast 4-way parallel radix sorting on gpus, 2009.

[MG11] DUANE MERRILL and ANDREW GRIMSHAW. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[MGG12] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[SGK17] Alec Radford Scott Gray and Diederik P. Kingma. Gpu kernels for block-sparse weights. December 2017.

[TV06] Terence Tao and Van Vu. On random ±1 matrices: Singularity and determinant. *Random Struct. Algorithms*, 28(1):1–23, January 2006.

6 Appendix

Figure 7: CSR × CSR performance (first 4), then CSR × Dense performance (next 2), then CSR × DenseVector performance (last 4)

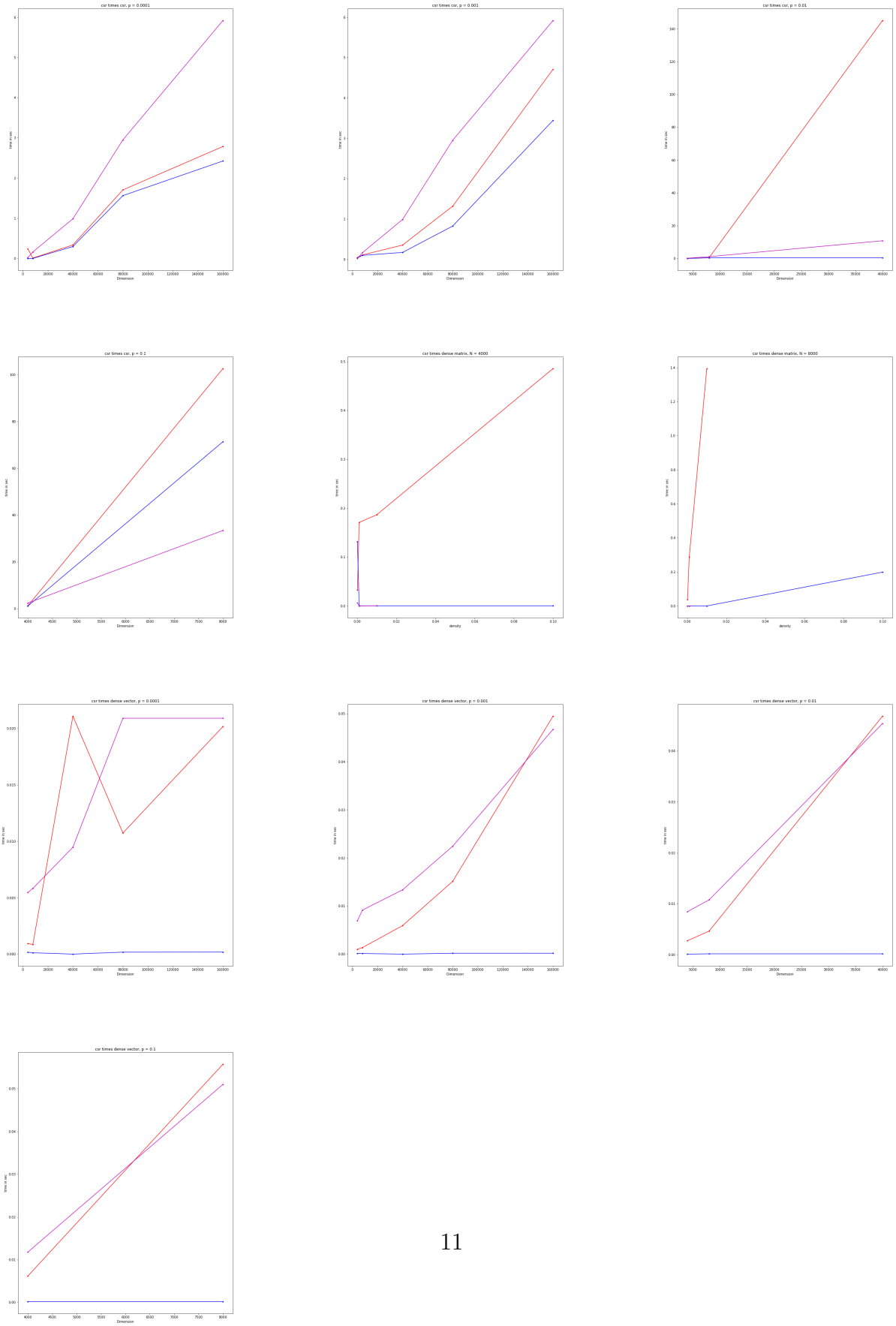


Figure 8: Log graphs of the previous results.