Storing and Computing Sparse Matrices with Dense Submatrices

JIN XIN EETU KAHELIN
xiji | kahelin @kth.se

October 2, 2023

Abstract

The use of heterogeneous computing (CPU-GPU) for sparse matrix operations is crucial. In feature maps produced by machine learning calculations, such as feature maps following the Fourier and wavelet transform, brain Computerized Tomography Medical Image (CT) scans, sparse matrix operations with dense submatrices are frequently utilized. In this research, we wish to provide a dynamic block storage approach for a sparse matrix with a dense submatrix to improve the operational continuity and the space requirements of a sparse matrix multiplication. It is a Nondeterministic Polynomial (NP) problem to find a dense submatrix in a sparse matrix and divide it into blocks. We anticipate using a heuristic approach to locate the dense submatrix in a sparse matrix, which most likely will not lead to global optimality. Finding and splitting suitable dense submatrix partitions for sparse matrices with clear dense submatrix features can reduce the number of memory visits to global memory. The idea of a dynamic Block Compressed Row Format (BSR) is to reduce access to row-continuous or column-continuous operations, which results in numerous accesses to shared memory, and to assign the length of shared memory equitably. Different allocations of shared memory are made dynamically based on the block matrix's size. We are using the Compressed Sparse Row (CSR) storage method to store the non-zero element distribution of the unprocessed non-dense distribution following sparse matrix partitioning.

Contents

1 Introduction **3**
1.1 Theoretical framework . . . 3
1.1.1 Sparse matrix . . . 4
1.1.2 Sparse matrix storage format . . . 5
1.1.3 GPU parallel computation . . . 5
1.2 Research questions . . . 5

2 Methods **6**
2.1 Submatrix block multiplication . . . 6
2.2 Dynamic block storage partition algorithm flow (dynamic BSR) . . . 6
2.3 Dataset Preprocessing . . . 6
2.4 Environment . . . 8

3 Results and Analysis **8**
3.1 Exploratory data analysis . . . 8
3.2 Experiment Comparison . . . 10

4 Conclusion **12**

5 Future work **12**

A Examples of CT scan images used in research **14**

B Dataset generate code **15**

List of Acronyms and Abbreviations

BSR Block Compressed Row Format

CSR Compressed Sparse Row

CSR5 Compressed Sparse Row 5

CT Computerized Tomography Medical Image

DIAG Diagonal Storage Method

ELL ELLAPCK

GEMM General Matrix-Matrix Multiplication

HYB Hybrid ELL/COO

NP Nondeterministic Polynomial

spECK "SpGEMM achieving Efficient Computation for all Kinds of matrices"

SpGEMM Sparse General Matrix-Matrix Multiplication

SpMV Sparse Matrix Multiplication

1 Introduction

Every image or picture can be converted to a matrix. When such an image contains a large number of pixels with the same colour, thus the matrix will contain a high number of the same entries, for example, the number zero, the converted matrix can be called a sparse matrix. Sparse matrices are used in a broad range of fields, such as Internet maps, 3D graphics, cryptography, statistics, and others.

The purpose of this research is to find an effective matrix storage method and the corresponding computation algorithm suitable for the specific type of sparse matrix; sparse matrix with dense submatrix. These matrices are commonly used in medical image processing, such as Computerized Tomography Medical Images (CT) and magnetic resonance imaging scans. We propose to use the dynamic Block Compressed Row format (BSR) sparse matrix storage method to store the dense submatrix.

Sparse matrices can be classified into various categories depending on the features these matrices have. Depending on the features, the sparse matrices can be categorized as banded sparse matrix, diagonal sparse matrix, block sparse matrix, and some sparse matrices do not have obvious features. As the features may vary, it is unrealistic to find a uniform sparse matrix storage method, to cover different types of matrices and improve the multiplication time, for all scenarios. This research will focus on a specific type of sparse matrix optimization, corresponding to a specific scenario data set, namely a medical image data set; CT scan.

1.1 Theoretical framework

Duff et al. (2017) [1] analysed and discussed classic algorithms for sparse matrices and compared the similarities and differences of how the same algorithm applies to dense matrices and sparse matrices, such as Gaussian elimination. In practical problems, dense and sparse matrices often appear at the same time in the same operation process, which is beneficial to help workers select more appropriate algorithms for engineering problems.

AlAhmadi S. et al. (2020) [2] found that there is not enough detailed research on the optimization of Sparse Matrix Multiplication (SpMV) on GPU, due to the sparse matrix features generated in different application scenarios such as numerical solutions of partial differential equations, machine learning and engineering questions, are not the same. As a result, there is no single unified sparse matrix storage or computation scheme that provides consistent maximum performance for all sparse matrix multiplication operations. Four sparse matrix storage schemes are compared in the article: CSR, ELLAPCK (ELL), Hybrid ELL/COO (HYB) and Compressed Sparse Row 5 (CSR5), and performed on 17 sparse matrices from 10 application areas tests. Based on the deep understanding gained from performance analysis, the heterogeneous CPU–GPU hybrid technique is proposed. This article is the first work to provide an exhaustive analysis of the performance of SpMV on GPUs.

The widely used library of sparse matrix methods that is now in use is not sufficiently specialized, and efficiency has not been maximized. For instance, the often-used *cudnn* package for deep learning is primarily intended for neural network convolution, pooling, normalization, and activation layers and does not support the processing of sparse matrix multiplication. A C++ library for matrix operations called *eigen*, has a decent support for sparse matrix multiplication, but is not specifically designed to speed up operations on sparse matrices. Deep learning has been widely applied in numerous fields since convolution was first introduced. General Matrix-Matrix Multiplications (GEMMs) serve as deep learning's primary computing foundation. However, the result is not perfect or uses a lot of computational effort when irregular and sparse matrix multiplication becomes more prevalent in deep learning application settings. A flexible and scalable architecture called SIGMA was presented by Eric Qin et al. in 2020 [3]. It is capable of adapting to various numerical distribution characteristics and matrix sparsity.

There are two keys to the acceleration of sparse matrix multiplication: sparse matrix storage method and sparse matrix optimization algorithm.

1. When storing the sparse matrices, they are recorded in a more complex and special sparse storage method. Compared to direct operations on the data, reading matrices in sparse storage format will

experience more traversal, resulting in more irregular and indirect access. When the matrix is too large, the shared memory cannot accommodate it, and block reads are also required. Sparse matrix multiplication operation time is more limited by data exchange and memory size. Therefore, data movement should be minimized during the algorithm process and more attention should be paid to the sparse storage format, rather than arithmetic computations that have been greatly reduced by sparse storage formats.

2. On the other hand, sparse matrix multiplication has different sparse characteristics in different application scenarios, and there is a situation where a single algorithm is no longer applicable due to the large changes in the characteristics of the data during the operation. Thus making it difficult to find a unified sparse matrix. An algorithm that applies to each type of sparse matrix multiplication does not exist, so specialized algorithms should be used for different types of sparse matrix multiplication to improve efficiency.

Gustavson (1978) [4] described a fast algorithm for two sparse matrices, one for sparse-dense multiplication and the other for sparse-sparse multiplication, which is still applicable today. Gustavson's algorithm provides ideas for the optimization of sparse matrix multiplication.

Considering that an optimization algorithm suitable for one class of sparse matrices may perform poorly on another class of sparse matrices with different sparse characteristics, it is not practical to speed up all types of matrices in the same way. Parger et al. (2020) [5] proposed a dynamic optimization model, which combined multiple optimization strategies with dynamic parameter selection, dynamically adjusted the best-fitting algorithm and parameters for each row, and obtained excellent results. The optimization model is called "SpGEMM achieving Efficient Computation for all Kinds of matrices" (spECK). Unlike previous sparse matrix algorithms, spECK aims to achieve good performance on more types of sparse matrix multiplications.

Tang Yang et al. (2020) [6] pointed out that, although GPU parallel computing was used to directly accelerate Sparse General Matrix-Matrix Multiplication (SpGEMM) in previous research, the code logic was not fully optimized. For example, in the algorithm, a fixed value of task length is allocated to each thread across the board, and the block is not suitable for the heavy global memory access caused by the application scenario, which makes the operation efficiency low. Tang Yang et al. (2020) suggested performing a pre-analysis before the calculation, adopting a reasonable dynamic block by row for matrix operation tasks, and using the hash table in shared memory to process the values produced by the intermediate calculation. These recommendations were based on the Coordinate Format (COO) sparse matrix storage method. It can be seen that reasonable dynamic partitioning before GPU operation is very important for the continuity of matrix operations and for reducing the number of memory accesses.

1.1.1 Sparse matrix

A sparse matrix [7] is a matrix with a small number of non-zero elements and a large number of zero elements. Whether in traditional engineering problems such as fluid mechanics, numerical solutions of partial differential equations, or in the recent increasingly popular application scenarios of machine learning image processing, sparse matrices are often computational objects. For example, in the application scenario of image recognition or image super-resolution reconstruction, an image matrix with rich colour pixel information is input, and only a small amount of key information is included in the feature map during the operation. L1 regularization [8] is commonly used in deep learning to obtain sparse matrices, and sparse matrices are significant computational objects that frequently appear in machine learning and engineering problems. In natural language processing, when the feature dimension is high and vectorized coding is used, high-dimensional sparse matrices appear more frequently. One of the main tasks of natural language processing is to reduce the sparsity of features.

A sparse matrix is a special form of data. On the one hand, there is currently no ideal data structure that can use limited space to represent sparse matrices in a simple process. On the other hand, sparse

matrices have different characteristics in different application scenarios. Therefore, there are no uniform data structure algorithms that can be applied efficiently to all sparse matrices.

1.1.2 Sparse matrix storage format

Using a sparse matrix storage format to record sparse matrices and reduce information redundancy is one of the current mainstream sparse matrix acceleration methods. A specific sparse matrix storage method is used to represent sparse matrices with unique sparse characteristics. For example, CSR is more suitable for sparse matrices in which the number of non-zero elements in each row is not much different, and there is no focal point in the horizontal dimension. For example, the Diagonal Storage Method (DIAG) is more suitable for storing a diagonal strip sparse matrix, and then operations are performed based on this type of storage structure. When the sparsity is large enough and the data size is small enough, after the sparse storage format is represented, it is more time-saving to perform operations directly on the CPU. When the amount of computation is small enough, the time it takes for the CPU to calculate a small amount of data is much shorter than the time it takes for the CPU to transfer the data to the GPU and call the process to read the shared memory. Therefore, when the sparseness of the sparse matrix is large enough, we can consider how to transfer the workload of format conversion to the GPU through the logic control of the CPU without the GPU performing the conditional logic judgment.

1.1.3 GPU parallel computation

As with ordinary matrix multiplication, the GPU is directly used for parallel computing, which is also a commonly used operation acceleration method in natural language processing and deep learning image convolution [9]. For a dense image matrix with rich content that is not sparse, this processing method is effective because using a particular sparse storage method cannot reduce the space occupation and calculation amount of zero elements, but increases the process of matrix format conversion. However, there are also grey-scale matrices that are not rich in changes in image processing, such as medical CT and remote sensing images, where medical CT images are usually sparse matrices containing density submatrices. For example, a sparse matrix of an ordinary pixel matrix after compression mapping. Although direct computing means that numerous unprocessed zero elements occupy and interfere with computing resources, direct parallel computing is still the mainstream computing method. Although there is a special sparse tensor representation in Python and there is also a lot of research on sparse deep learning in the recent literature, in practice, sparse matrix operations are rarely distinguished from dense matrix operations. Such a processing method of directly using the GPU for parallel computing without distinction affects the long-term development of deep learning to improve efficiency.

1.2 Research questions

The main problem is how to design a low-cost path to find the dense submatrix in an arbitrary sparse matrix. The discrete non-zero elements distributed outside the dense submatrix in a sparse matrix do not always make sense in the final image processing result. Therefore, it is necessary to store such discrete non-zero elements using the CSR sparse matrix storage method, or just to omit the discrete non-zero elements.

The matrix multiplications in our chosen scenarios are a specific type of matrices that we investigate in the project; sparse matrix with dense submatrix/submatrices. The algorithm is designed for a specific matrix multiplication scenario and is assumed to work on all matrices under practical conditions. The specific scenario would be the medical imaging dataset that will be described later. In the data set, there might be a bar sparse matrix or a sparse matrix without specific features in practical scenarios. We will not set additional “if” operations in the code to distinguish if the matrix used is a specific sparse matrix with a dense submatrix or not. Using additional “if” operations will waste a lot of computation time, especially in GPU, as it only has a few control units. Thus, the computation has to return to the CPU to implement the “if” operation, making the previous improvements meaningless. We hypothesize that it is possible to improve the algorithm, which means that the computation is faster for a specific matrix scenario without adding additional “if” operations.

2 Methods

For our research, we use publicly available medical datasets. These datasets contain CT-scan images of anonymized patients. Anonymization has been performed by either the third party or the party that offers the medical dataset. All in all, the datasets contain more than 45,000 images, but we sliced the datasets to a more reasonable size. The size is on par with the time constraint given to this project. We used dataset slices from the Radiology Data from The Cancer Genome Atlas Lung Adenocarcinoma (TCGA-LUAD) (2016) [10] dataset and from the melanoma ISIC dataset [11]. The data was downloaded from the source, and it was stored locally where the computations were done.

Permission to distribute the scan images included in the datasets has been given by each patient. The data has been anonymized, each participant has received an identification code. Only by the identification code, it is not possible to determine the personality of the patient. Additionally, researchers agreed not to attempt to determine the personality of the participants and has not included any identifiable information in the report.

The analysis was carried out by creating the proposed algorithm with the C++ language and validating it using the *cuSparse* API by measuring results such as multiplication time.

2.1 Submatrix block multiplication

Multiplication between sparse matrices with dense submatrix features requires more access to shared memory and movement of data than BSR storage if using CSR. A sparse matrix with dense sub-matrix characteristics may also have scattered elements. When the dynamic storage of the dense sub-matrix is complete, we use the CSR to store the remaining scattered non-zero elements. For matrix multiplication, it has the following properties; if the two matrices contain only one dense sub-matrix, and the remaining elements are zero, then: let A and B be the non-zero elements block in the first matrix, and let C_1 and C_2 be the non-zero elements of the second sparse matrix as shown in Figure 1. The element block C_1 is all regular rectangles or squares in shape, which is the part that C_1C_2 coincides with the AB sum after the second matrix is transposed.

$$\begin{bmatrix} 0 & \dots & \dots & \dots \\ \dots & A & B & 0 \\ \dots & \dots & \dots & 0 \\ 0 & 0 & \dots & 0 \end{bmatrix} \bullet \begin{bmatrix} 0 & \dots & \dots & \dots \\ \dots & C_1 & \dots & \dots \\ \dots & C_2 & 0 & \dots \\ 0 & \dots & \dots & \dots \end{bmatrix} = \begin{bmatrix} 0 & \dots & 0 & 0 \\ \dots & A \bullet C_1 & 0 & \dots \\ \dots & 0 & 0 & \dots \\ 0 & \dots & \dots & 0 \end{bmatrix}$$

Figure 1: Schematic diagram of submatrix block multiplication for sparse matrices

2.2 Dynamic block storage partition algorithm flow (dynamic BSR)

Our dynamical BSR methods are based on original BSR blocks with stubborn size. For example, in Figure 2, the BSR's every dense block is (2,2) size, and we find the suited block record size by connecting adjacent matrices. We apply a simple and quickly path to decide and find adjacent matrices: If the distance between the first element of two blocks is less than (block_size+2), then the two dense block will be grouped in a bigger block.

2.3 Dataset Preprocessing

To preprocess the dataset, the matrices were cut from a block size of (1024, 1024) to a matrix size of (256, 256). Original pixel values were converted to reciprocal values, in order to remain and transfer the dark parts in the matrices to sub-dense matrices, and then normalize the pixels to [0,1]. To observe the influences

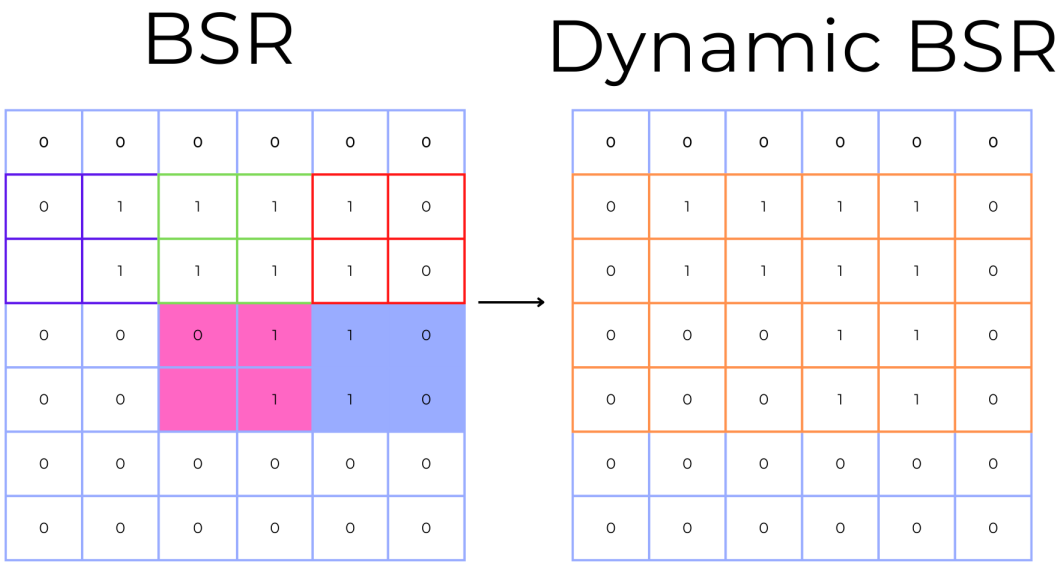

Figure 2: Dynamic BSR

caused by other storage methods, random matrices with different sparsity conditions were generated. The code structure is based on code by T. Nesrovnal [12]. We mainly added a group pointer to record the different sizes of sub-dense matrices in the sparse matrix, which is marked red in Figure 3. All the matrices are stored in the COO format at the beginning.

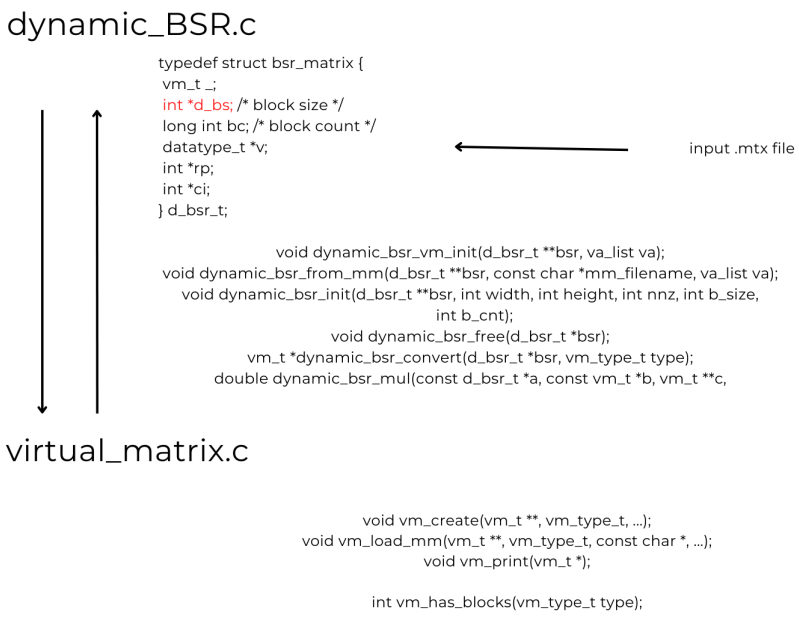

Figure 3: Code Structure

2.4 Environment

The computer environment for the project is:

- Ubuntu 18.04
- gcc 7.5.0
- python 3.6
- Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz.

3 Results and Analysis

Raw CT-scan images, such as those presented in Figure 12 of the Appendix A and in Figure 4, were converted to binary matrices and then analysed if they match the “dense submatrix in sparse matrix” criteria. We generated bar charts, examples presented in Figures 5 and 6, from all raw CT-scan images, confirming that there are indeed dense submatrices in our CT-scan matrices.

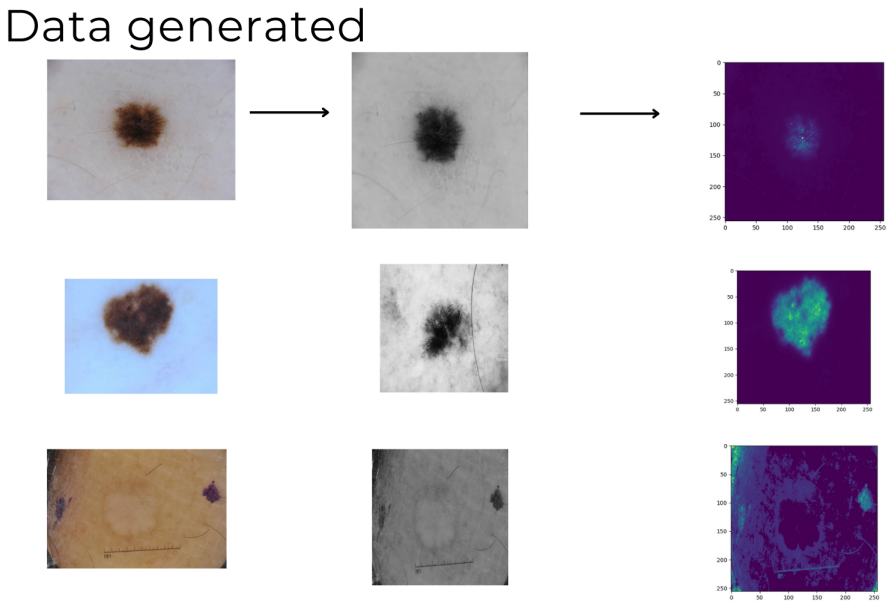

Figure 4: Data Preprocessing

3.1 Exploratory data analysis

When doing the exploratory data analysis, we investigated whether the images and generated sparse matrices contain dense sub-matrices. This was done with Python and if CT-scan contains a dense submatrix, the program will simply print ’True’ to stdout. The check was done by first creating a new matrix of the sum of every neighbouring (3 elements in every direction) element, and this “neighbour sum matrix” was then transformed into an array by summing every element in each row. Finally, it was checked that in this array there is at least one row with only zeroes (the sum is 0) and at least one row with a sum value higher than 512 (this was chosen by comparing the results).

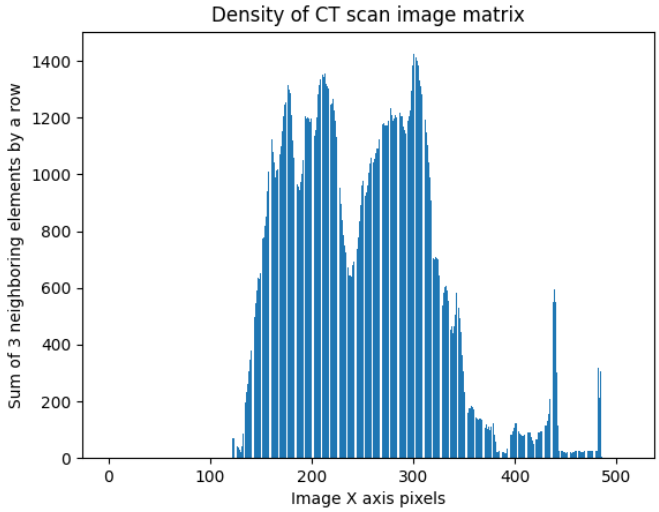

Figure 5: Validating the CT scan matrix of the CT scan image presented in Figure 12 of Appendix A.

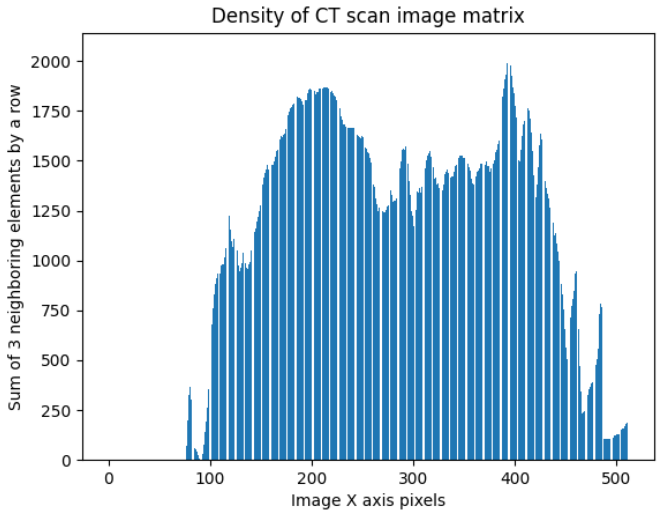

Figure 6: Validating that the CT scan image, presented in Figure 13 of Appendix A, satisfies “dense submatrix in sparse matrix” criteria.

3.2 Experiment Comparison

The C-language implementation of the experiment analyses the performances of different matrix storage algorithms, and how the dynamic BSR affects the melanoma dataset. It is obvious, that the dense matrix multiplication computation time is considerably higher than with other matrix storage methods, as seen in Figure 7. Additionally, as Figure 7 presents, storing the sparse matrix with an accurate method can save a significant amount of time and computation resources for image processing, and also prove the efficiency of applying sparse matrix storage in image machine learning applications. In current experiments, the COO sparse matrix storage method is more efficient than other sparse storage methods, but such a conclusion is not viable because our test matrix is stored in COO format at the beginning. For BSR storage method with different block sizes, it can be seen that different sparse matrices have different suitable block sizes to meet the best matrix multiplication efficiency. By analysing the Table 1, we can conclude that the dynamical BSR storage method is more efficient than other methods in the particular case.

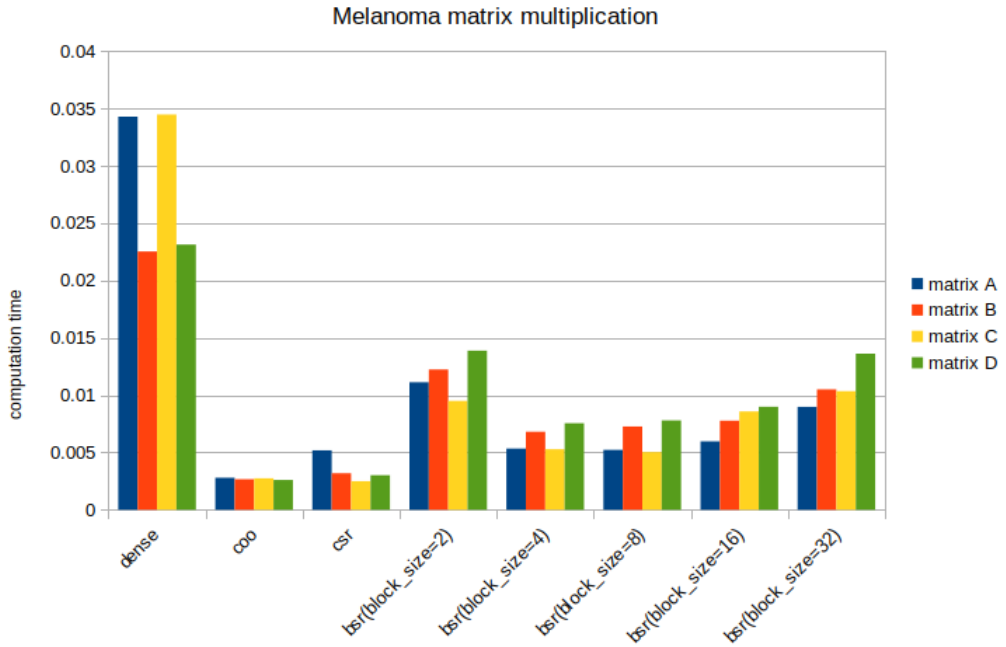

Figure 7: Melanoma matrix multiplication time

Table 1: Matrix storage method comparison on the Melanoma dataset

Melanoma(256 X 256, 55.6% sparse ratio)	matrix A	matrix B	matrix C	matrix D
dense	0.034294	0.022537	0.034482	0.023131
coo	0.002797	0.002665	0.002744	0.002602
csr	0.005182	0.003184	0.002496	0.003013
bsr(block_size=2)	0.011117	0.012236	0.009499	0.013874
bsr(block_size=4)	0.005338	0.006803	0.005279	0.007544
bsr(block_size=8)	0.005228	0.007262	0.005027	0.007792
bsr(block_size=16)	0.005966	0.007766	0.00858	0.008983
bsr(block_size=32)	0.008973	0.010501	0.010343	0.013618

In the randomly generated matrices, dynamic BSR storage matrix multiplication, in some situations, takes less time to compute than CSR or COO, as it is demonstrated in Figures 8, 9, 10 and 11. This happens particularly when dense matrix distributions in a sparse matrix are located relatively close to a block. The BSR storage works better when a matrix is closer to “a sparse matrix with multiple sub-dense matrices”.

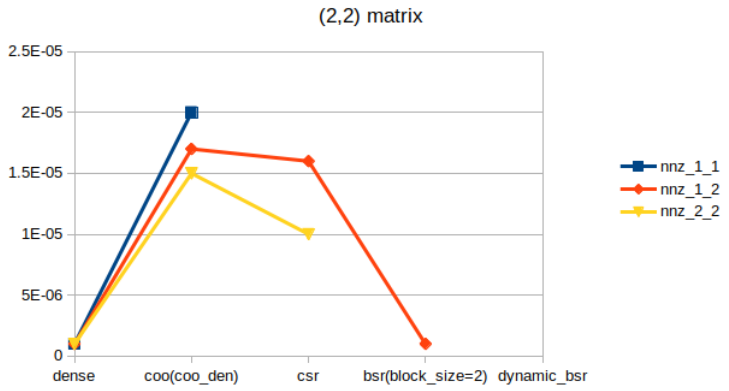

Figure 8: (2,2) matrix multiplication time

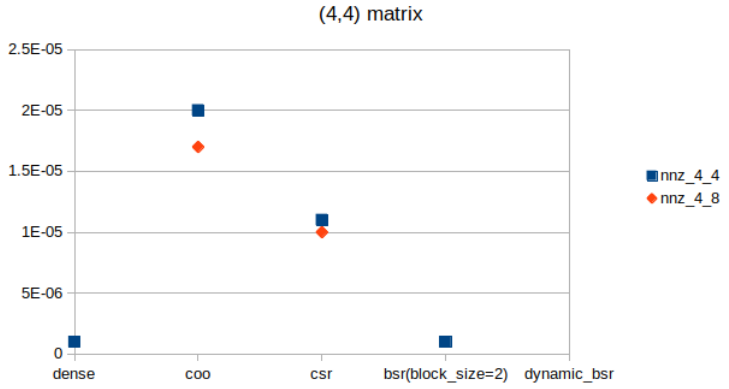

Figure 9: (4,4) matrix multiplication time

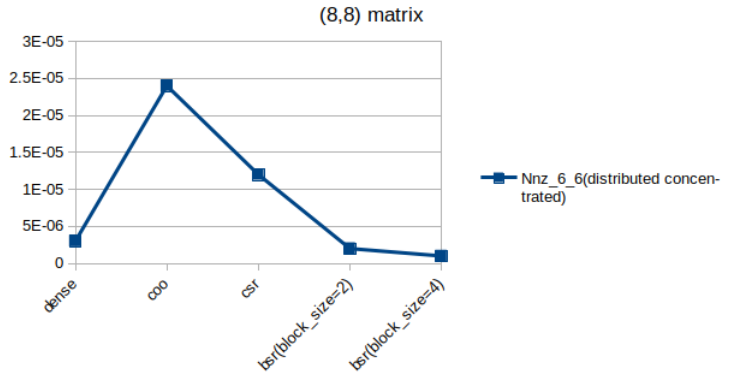

Figure 10: (8,8) matrix multiplication time

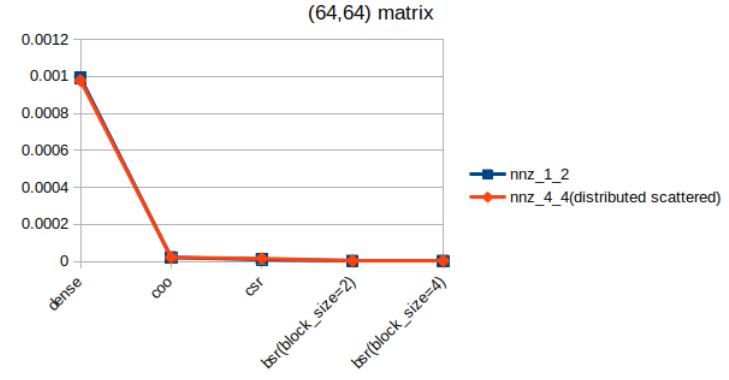

Figure 11: (64,64) matrix multiplication time

4 Conclusion

The results of the data analysis indicated that our project topics are meaningful and feasible. The problem was to design a low-cost algorithm that finds a dense submatrix in an arbitrary sparse matrix. The designed algorithm, dynamic BSR storage method, is implemented in C-language, and could be transferred to Python, thus be combined with practical sparse matrix multiplication applications. Most sparse matrix storage applications are used for scientific computing scenarios such as linear algebra operations. Image machine learning always used the original dense matrix, which can waste a lot of computer space, time, and resources. Our results indicate that sparse matrix storage methods are very effective in matrix multiplication applications when the matrix objects are images. BSR storage with dynamical block sizes is more efficient than BSR storage with fixed block size, so we can save matrix multiplication time by modifying the suitable block for the BSR method.

5 Future work

The algorithm could be improved by optimizing the GPU usage. The GPU should show improvements in matrix multiplication efficiency with the appropriate block size. BSR storage could allow the GPU to read a matrix row constantly and start with a variety of reasonable memory sizes. A row reading for matrix multiplication wouldn't be interrupted by BSR storing if the block size was appropriate, lowering the frequency of GPU-CPU data interaction. The improvements to the dynamical BSR so far are small on CPU. The dynamic BSR matrix's main goal is to increase GPU parallel computing by ensuring that each row of the matrix is continuously read from global memory by the GPU. All the tests conducted so far have used a CPU. In our upcoming work, we will use GPU multiplication to build the dynamical BSR storing method, which should significantly speed up matrix multiplication.

References

[1] I. Duff, A. Erisman, and J. Reid, *Direct Methods for Sparse Matrices*, ser. Numerical Mathematics and Scientific Computation. OUP Oxford, 2017. ISBN 9780192507501

[2] S. AlAhmadi, T. Mohammed, A. Albeshri, I. Katib, and R. Mehmood, "Performance analysis of sparse matrix-vector multiplication (spmv) on graphics processing units (gpus)," *Electronics*, vol. 9, no. 10, p. 1675, Oct 2020. doi: 10.3390/electronics9101675. [Online]. Available: http://dx.doi.org/10.3390/electronics9101675

[3] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020. doi: 10.1109/HPCA47549.2020.00015 pp. 58–70.

[4] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, p. 250–269, sep 1978. doi: 10.1145/355791.355796. [Online]. Available: https://doi.org/10.1145/355791.355796

[5] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, "Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3332466.3374521. ISBN 9781450368186 p. 362–375. [Online]. Available: https://doi.org/10.1145/3332466.3374521

[6] Y. TANG, D.-f. ZHAO, Z.-b. HUANG, and Z.-t. DAI, "High performance row-based hashing gpu spgemm," *Journal of Beijing University of Posts and Telecommunications*, vol. 42, no. 3, pp. 106–113, 2019. [Online]. Available: https://journal.bupt.edu.cn/EN/10.13190/j.jbupt.2018-252

[7] I. S. Duff, "A survey of sparse matrix research," *Proceedings of the IEEE*, vol. 65, no. 4, pp. 500–535, 1977.

[8] R. Ma, J. Miao, L. Niu, and P. Zhang, "Transformed e1 regularization for learning sparse deep neural networks," *Neural Networks*, vol. 119, pp. 286–298, 2019.

[9] Z. Gu, "Optimizing block-sparse matrix multiplications on cuda with tvm," 2020. [Online]. Available: https://arxiv.org/abs/2007.13055

[10] B. Albertina, M. Watson, C. Holback, R. Jarosz, S. Kirk, Y. Lee, and J. Lemmerman, "Radiology data from the cancer genome atlas lung adenocarcinoma [tcga-luad] collection. the cancer imaging archive," 2016. [Online]. Available: http://doi.org/10.7937/K9/TCIA.2016.JGNIHEP5

[11] B. Cassidy, C. Kendrick, A. Brodzicki, J. Jaworek-Korjakowska, and M. H. Yap, "Analysis of the isic image datasets: usage, benchmarks and recommendations," *Medical Image Analysis*, vol. 75, p. 102305, 2022.

[12] T. Nesrovnal, "Vliv formátu uložení řídké matice na vỳkonnost násobení řídkỳch matic," 2014, http://hdl.handle.net/10467/24976 and https://github.com/nesro/sparse-matrices.

A Examples of CT scan images used in research

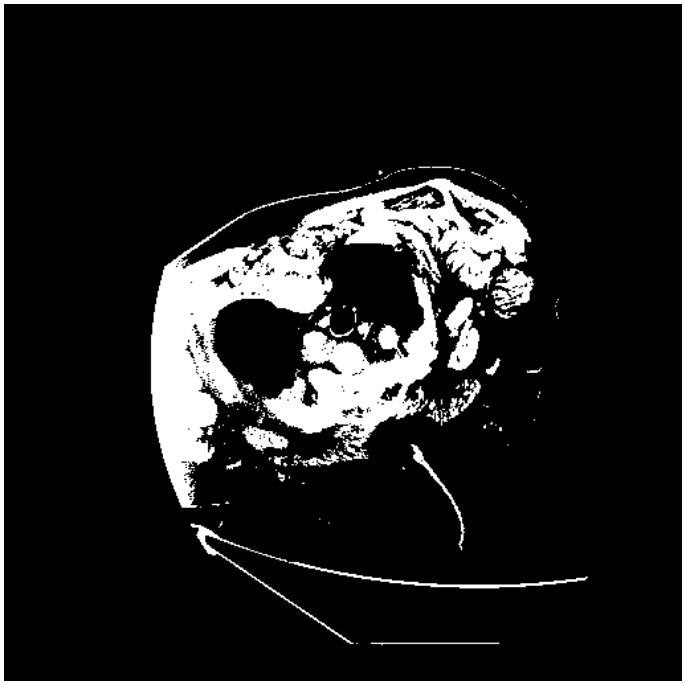

Figure 12: Another example of a CT scan image

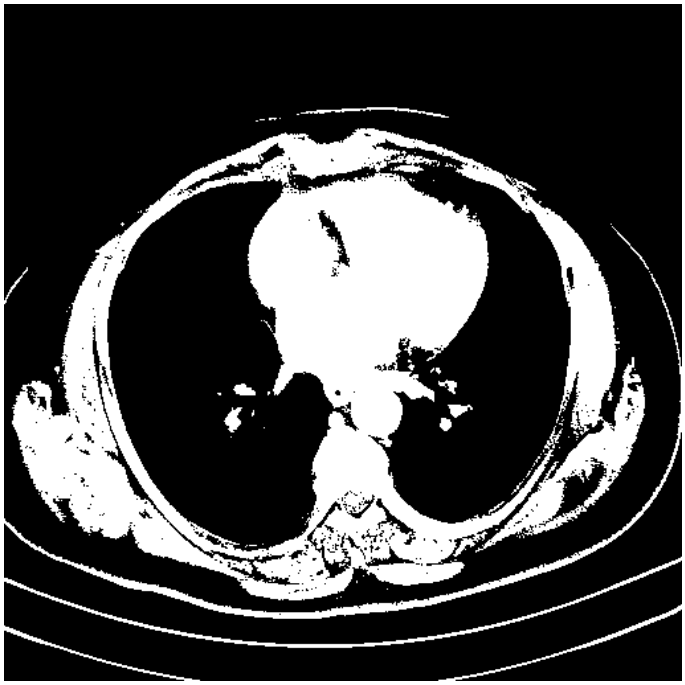

Figure 13: Example of a CT scan image

B Dataset generate code

```
from numpy import *
from scipy.sparse import coo_matrix
from scipy import sparse
import matplotlib.image as mab
import numpy as np
from skimage import io, transform, img_as_float
import matplotlib.pyplot as plt
import os
import cv2
from PIL import Image
data_dir='/home/dell/Desktop/Project/ISIC_2019_Training_Input/'
mtx_save_dir='./save_dir/'
def dataset_generate(ori_mtx_dir, file, mtx_save_dir):
  im = mab.imread(ori_mtx_dir);
  #read color image
  color_img = cv2.imread(ori_mtx_dir)
  reImg = cv2.resize(color_img, (256, 256), interpolation=cv2.INTER_CUBIC)


  gray_img=cv2.cvtColor(reImg,cv2.COLOR_RGB2GRAY)
  #gray_img is a two-dimensional matrix representation, to achieve conversion from
    array to image
  gray=Image.fromarray(gray_img)
  #Save the picture to the current path, the parameter is the saved file name
  #gray.save('gray4.jpg')

  gray = img_as_float(gray)  #becomes a float [0-1].
  #gray= np.around(gray, 2)
  #np.set_printoptions(precision=2)
  gray = (gray - gray.min()) * (1 / (reImg.max() - reImg.min()))

  print('The gray matrix is:{}'.format(gray))

  gray_median=np.median(gray)

  width=gray.shape[0]
  height=gray.shape[1]
  #gray_median=0.0035
  print('The width is:{}'.format(width))
  print('The height is:{}'.format(height))


  for i in range(width):
    for j in range(height):
      if(gray[i,j]<=gray_median):
        gray[i,j]=0
      if(gray[i,j]==nan):
        gray[i,j]=1
      print(gray[i,j])
  #plt.imshow(gray)
  #plt.show()
  gray= np.around(gray, 6)# reserve 6 digits
  nnz=np.count_nonzero(gray)

  for i in range(width):
    for j in range(height):
      print(gray[i,j])
  print('The sparse ratio is:{}'.format(1-(nnz/(width*height))))
  save_path=mtx_save_dir+file.split('.')[0]+'.mtx'

```

```
  with open(save_path,'w') as f:
    f.truncate(0)
    f.write("%%MatrixMarket matrix coordinate real general\n")
    f.write('{} {} {}\n'.format(width,height,nnz))
    for i in range(width):
      for j in range(height):
        if(gray[i,j]!=0):
          f.write('{} {} {}\n'.format(i+1,j+1,gray[i,j]))


#dataset_generate(dir,'hhh')
files = os.listdir(data_dir)
for file in files:
  path=data_dir+file
  print(path)
  dataset_generate(path,file,mtx_save_dir)
```