Memory-Efficient Voxel Processing Library

Submitted to:
Professor Kenneth Fletcher
Department of Computer Science
University of Massachusetts Boston

By

Radhikabahen D Patel
Jithin Jacob
Quan Tran
Senthil Palanivelu

Table of Contents

1 Introduction

Voxels are one of the ways to represent 3D objects in additive manufacturing (aka 3D printing). The ability to use voxel-based geometry for non-complex and robust morphological processing is an important element of 3D simulation. Without a voxel data structure, we cannot represent an object or structure in three-dimensional space.[1] However, the memory footprint increases with the cell size of the volume being described, leading to large voxel data structure, which requires very large matrices (data structure) to represent the object.

Very large matrices obviously require a lot of memory, and this is often a space complexity bottleneck in large computed-imaging problems. Yet, most of these large matrices that we wish to work with are sparse, i.e., they are matrices that have many more zero values than non-zero values. For this reason, storing large sparse matrices in their original (matrix) form is clearly a waste of memory resources as those zero values do not contain any information.

The client of this project, Product Innovation and Engineering, LLC., wants to perform morphological and basic arithmetic operations on voxel images whose size is larger than the RAM of the system that they use. So, there was a need for a compressed data structure so that morphological and arithmetic operations could be carried out without running out of memory while processing large-sized files. An additional requirement was that the data structure had to be castable to and from a NumPy array. To address this issue, existing technologies were researched and a voxel-processing module for Python was developed using NumPy and SciPy modules, which provides morphological and arithmetic operations using divide and conquer–based algorithm.

2 Literature Survey

A few existing data structures and libraries were considered to help solve the problem presented. The explored approaches are explained in detail in the following sections.

2.1 Octree

An octree is a data structure in which each node has exactly eight children. It is used to partition three-dimensional space by recursive division of the space it represents into eight octants, as shown in Fig 2.1. This data structure helps obtain high speeds during morphological operations. The octree also provides an extremely rigorous compression as compared with other 3D spatial subdivision approaches. This means that a relatively enormous space containing a very small amount of voxel solids can be efficiently stored using an octree.

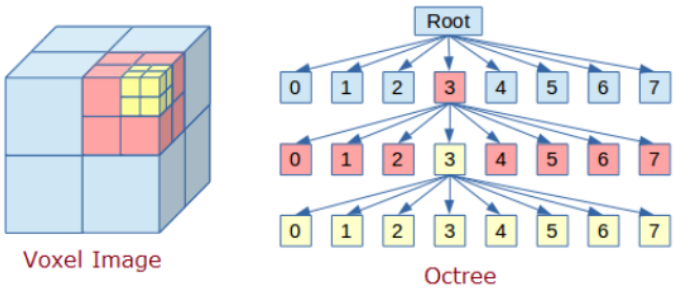

Figure 2.1 Octree Data Structure

However, the drawback of the octree is that it does not help solve the memory problem. As the size of the matrix increases, so does the depth of the tree representing the matrix. As a result, the memory grows exponentially with respect to the depth of the tree. In addition, since working on a massive contiguous array is not possible, it is more feasible to process chunks of data at a time. This means retrieving block data from the disk, dynamically allocating memory, and performing morphological operations. Such operations will be complicated as it is not certain whether the desired output will be obtained.

2.2 OpenVDB

The OpenVDB library is an open-source C++ library that uses a novel hierarchical data structure and has its own suite of tools that can be used for the efficient storage and manipulation of sparse volumetric data on three-dimensional grids. The drawback of this library is the fact that it runs on Python 2.7, whereas the client uses Python 3.0.

2.3 Memory Mapping

Memory mapping is a technique that deals with files by mapping them into the address space of the processor. Memory mapping a file uses the virtual memory system of the operating system to access the data on the file system directly, instead of using normal input/output functions. Memory mapping typically improves input/output performance because it does not involve a separate system call for each access and it does not require copying of data between buffers—the memory is accessed directly.

2.4 Compression of Sparse Matrices

As mentioned before, representing a sparse matrix using a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes along with non-zero elements, we only store non-zero elements. There are many methods for the storage of sparse data, of which Compressed Row Storage (CRS) was used.

In CRS, the subsequent non-zeros of the matrix rows are put in contiguous memory locations, as shown in Fig 2.2. Assuming a non-symmetric sparse matrix, three vectors are created: one for floating point numbers (val) and two for integers (col_ind and row_ptr). The ‘val’ vector stores the values of the non-zero elements of the matrix, as when traversed in a row-wise fashion. The ‘col_ind’ vector stores the column indexes of the elements in the ‘val’ vector. The ‘row_ptr’ vector stores information of non-

zero elements of each row. The number of elements in the *i*th row can be obtained by subtracting the value of row_ptr[i] from row_ptr[i+1].

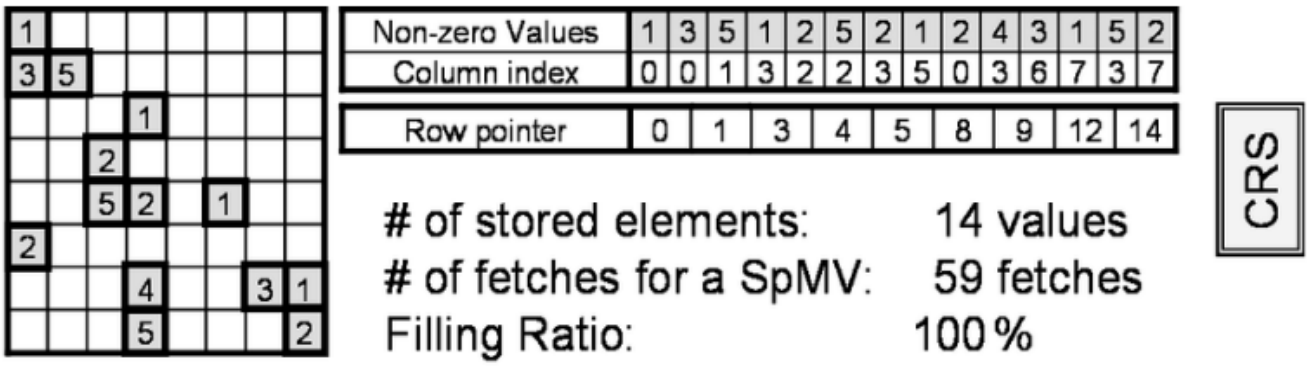

Figure 2.2 CRS Compression

2.5 Morphological Operations

Morphology is a broad set of image processing operations that process images based on shapes. In a morphological operation, each pixel in the image is adjusted based on the value of other pixels in its neighborhood. By choosing the size and shape of the neighborhood, you can construct a morphological operation that is sensitive to specific shapes in the input image.
The most basic morphological operations are dilation and erosion. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries. Dilation and erosion are often used in combination for specific image preprocessing applications, such as filling holes or removing small objects.

3 Algorithm

The first step is to check whether the input array is sparse. If 50 percent of the elements or less are non-zero elements, it is treated as a sparse array and the CRS algorithm is applied. If the 3D array is determined to be sparse, the array is converted into a 2D array for compression. For example, a 3D array of dimensions x × y × z would be converted into a 2D array of dimensions (x × y) × z. If the array is not sparse, then compression is not applied.

Then, the required morphological operation is carried out using block operation, as shown in Fig 3.1. In block operation, an image is processed in blocks rather than all at once. The blocks have the same size across the image, and the same operation is applied to all the blocks, one at a time. The blocks are then reassembled to form an output image. The portions of blocks can be easily reconstructed from CRS using the array index notation. Using this approach, the whole array does not need to be stored in the RAM; instead, blocks of submatrices are passed. The benefit of block-wise computation can be extended to distributed work on different streams.

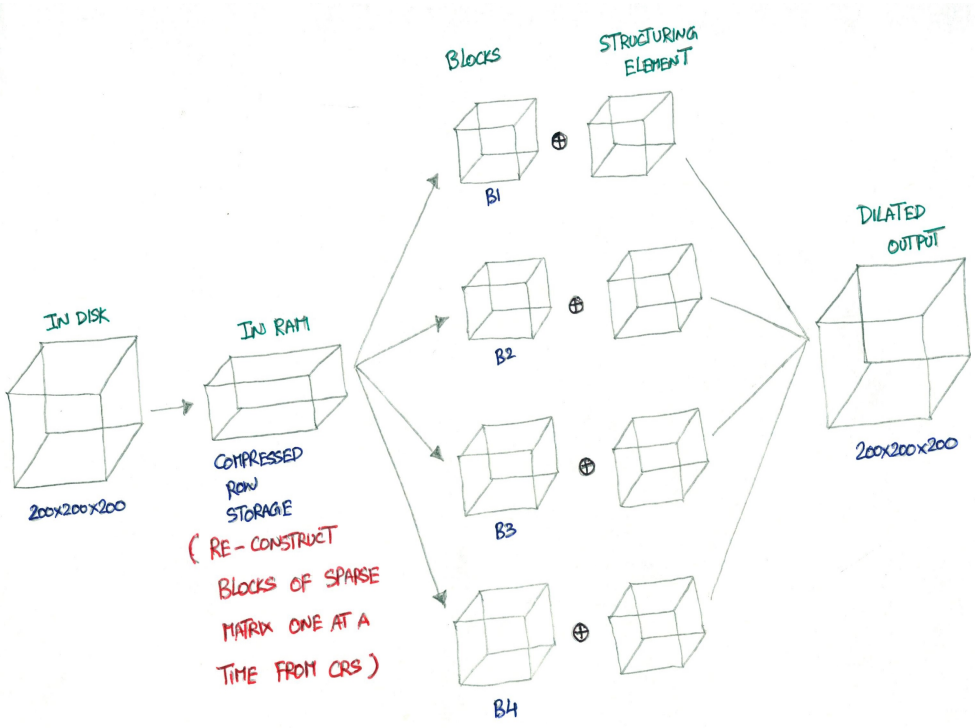

Figure 3.1 Basic Architecture

For algorithms that expect multiple samples of one or more steps and one or more features, there is a need to reshape two-dimensional data, where each row represents a sequence, into a three-dimensional array. Morphological operations are such algorithms. CRS compression was carried out on a 2D array, and therefore to perform morphological operations, the 2D array is converted into a 3D array. For example, if the reconstructed 2D blocks are of dimensions (x × y) × z, then it gets converted into 3D blocks of dimensions x × y × z (Fig. 3.2).

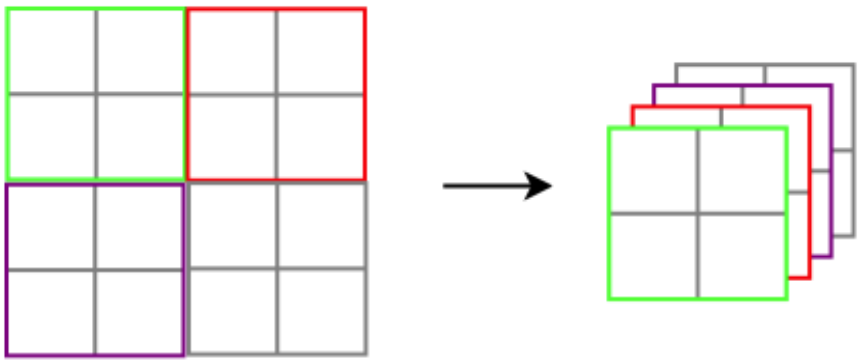

Figure 3.2 2D to 3D transformation

Then, the required morphological operations are carried out block by block. However, a common problem with this approach is the calculation of values at the borders between blocks, since these require values from one or more neighboring blocks. The solution is to allocate additional space for a series of ghost cells around the edges of each block. For every iteration, have each pair of neighbors exchange their borders and place the received borders in the ghost-cell regions (Fig. 3.3). Each block receives a vector of ghost cells from its neighboring block.

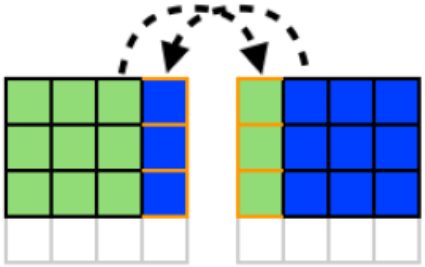

Figure 3.3 Ghost Cell Exchange

After the morphological operations have been carried out, the merging operation merges multiple same-sized blocks. The blocks to be merged are stored contiguously in disk memory. Merging combines continuous blocks on the same shard into a single block.

Fig. 3.4 shows the complete process for a single block when dilation is applied to a 200 × 200 × 200 matrix.

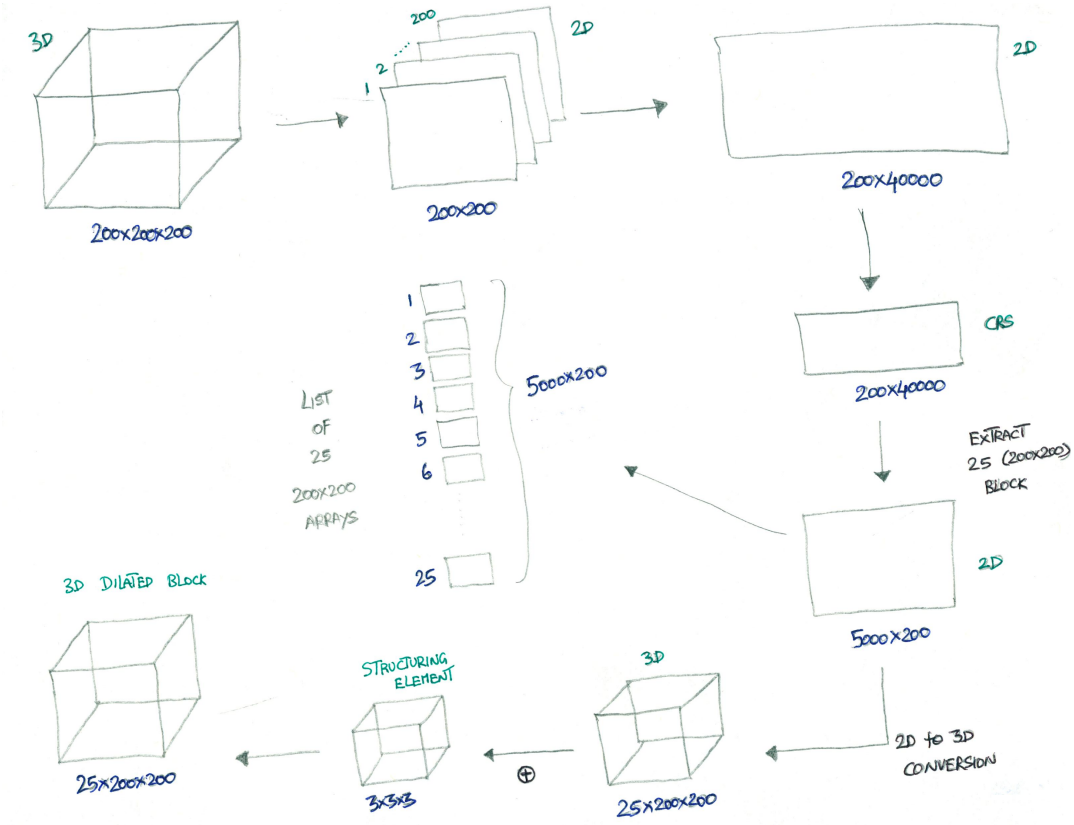

Figure 3.4 Architecture of a Single Block Operation

3.1 Software and Module Requirements

This novel approach is compatible with Python 3.0 and later versions. It also makes use of the NumPy and SciPy Python libraries.

3.2 Implementation

For ease of use, functional coding was used in the implementation of the approach. Only the created library, voxel.py, needs to be imported by the client. voxel.py inherently imports the voxelProcessing library and the prerequisite libraries, which are SciPy and NumPy.

voxel.py has 17 functions, 15 of which perform the same operations as the functions in the Morphology section of the SciPy library and 2 are custom functions. The created morphological operations have the same names as the functions in the SciPy library for ease of use. The functions are as follows:

1. Binary Morphology
    a. binary_erosion(…)
    b. binary_dilation(…)
    c. binary_opening(…)
    d. binary_closing(…)
    e. binary_fill_holes(…)
    f. binary_hit_or_miss(…)
    g. binary_propagation(…)

2. Grey-scale Morphology
    a. grey_erosion(…)
    b. grey_dilation(…)
    c. grey_opening(…)
    d. grey_closing(…)
    e. morphological_gradient(…)
    f. morphological_laplace(…)
    g. white_tophat(…)
    h. black_tophat(…)

3. Custom Functions
    a. multiply(…)
    Integer/float value multiplication of each element with given scalar value.
    b. nothing(…)
    It does not perform any morphological or arithmetic operations. It just carries out blocking and merging, to verify that the algorithm works correctly in case the client makes any changes to main() in voxelProcessing.py

The parameters are as follows:

1. input_var
   Type: 3D NumPy array
   Description: The input array
2. no_of_blocks
   Type: int
   Description: The number of blocks (with respect to x-axis) into which to divide the original image. Default: 4
3. fakeghost
   Type: int
   Description: The number of extra rows or ghost cells around each block required for morphological operations, generally proportional to the structuring element. Default: 2. But it needs to be greater than 1.
4. make_float32
   Type: boolean
   Description: Whether to type-cast input array to float32. Default: True
5. structure
   Type: NumPy 3D array
   Description: The structuring element to be used for the morphological operation.

The voxelProcessing class has only one public method: main(). The main() method creates blocks of the NumPy array input and performs the specified operation on each block. After operation on each block, it will store the output of each block in a file. After processing all the blocks, it then reads data from the created file and returns the whole array as the output.

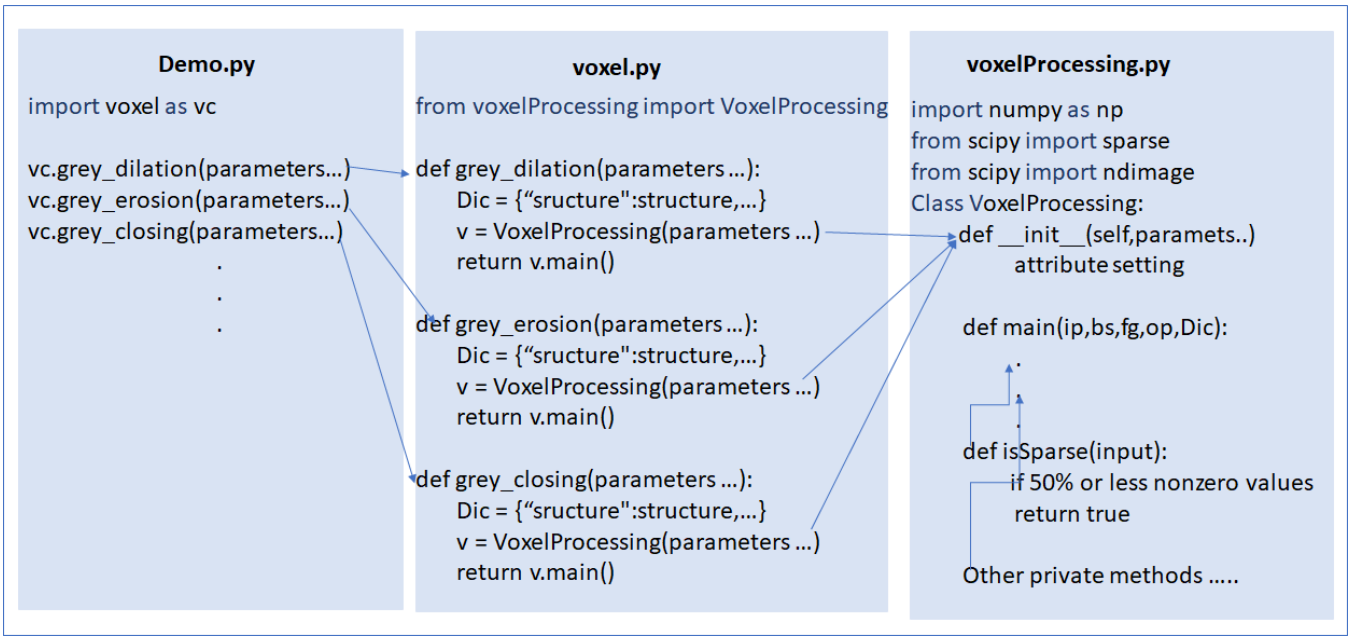

Figure 3.5 Module API

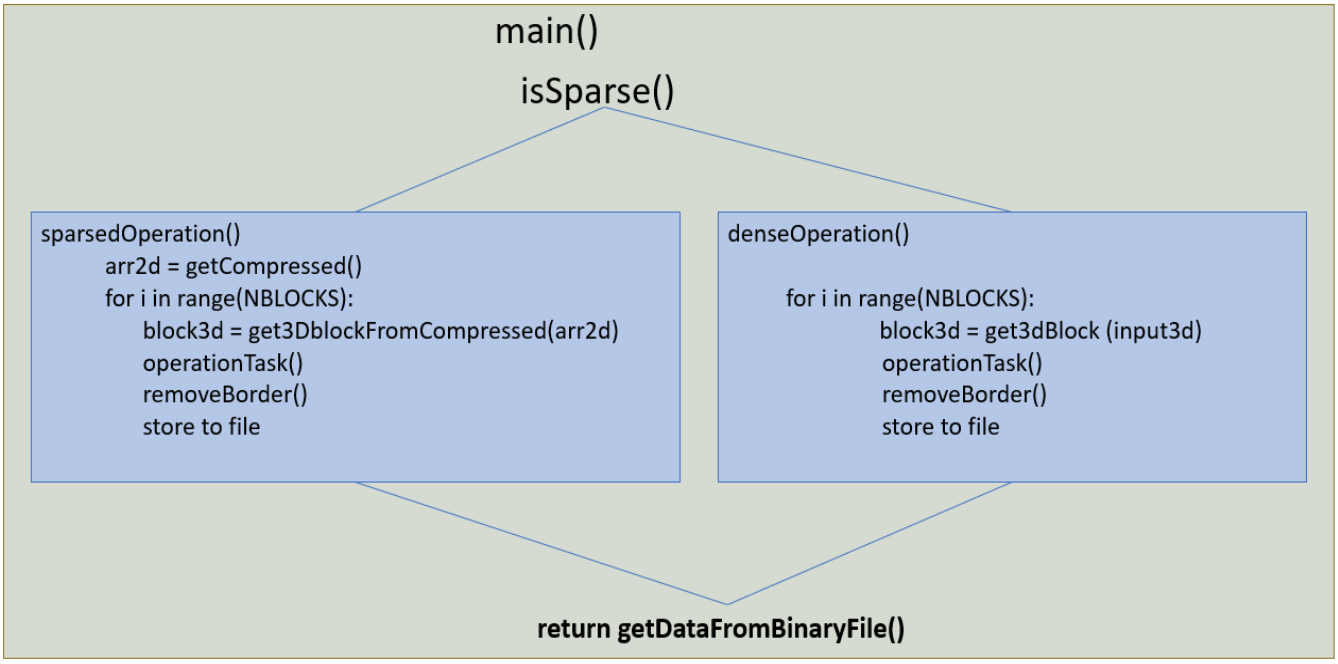

Figure 3.6 Approach Implementation

3.3 Directory Architecture

Memory-Efficient-Voxel-Processing-Library
|
\---src
| | demo.py — demo of voxel library functions
| | voxel.py — contains functions that perform operations using voxelProcessing.py
| | voxelProcessing.py — Implements the divide and conquer–based algorithm
|
| +---unittest — unit tests for public methods of voxel.py and voxelProcessing.py
| |
| | run_all_tests.py — executes all unit testing
| | test.py — voxel module unit test with multiple parameter values.
| | test_binary_closing.py
| | test_binary_dilation.py
| | test_binary_erosion.py
| | test_binary_fill_holes.py
| | test_binary_hit_or_miss.py
| | test_binary_opening.py
| | test_binary_propagation.py
| | test_black_tophat.py
| | test_grey_closing.py
| | test_grey_dilation.py
| | test_grey_erosion.py
| | test_grey_opening.py

| | test_morphological_gradient.py
| | test_morphological_laplace.py
| | test_multiply.py
| | test_nothing.py
| | test_voxelProcessing.py
| | test_white_tophat.py

4 Tests and Analysis

For testing, grey dilation was carried out on four input arrays of different matrix dimensions and densities to determine runtime and average memory usage.

4.1 Test 1

- Input matrix dimensions = 2000 × 1500 × 1500
- Matrix density = 10%
- Matrix size = 17.0 GB
- Compressed object size = 3.6 GB

When grey dilation was carried out using the NDImage library of SciPy, the system ran out of memory and displayed the following error message:

"OSError: [Errno 12] Cannot allocate memory. The total system memory is 32gb."

When done with voxelProcessing.py, the average memory usage dropped with respect to the number of blocks. The minimum average memory usage was 3429 MB using 500 blocks of operation (Fig. 4.1). However, the runtime increased with respect to the number of blocks. For the minimum average memory usage of 3429 MB using 500 blocks of operation, the runtime was 1849 seconds.

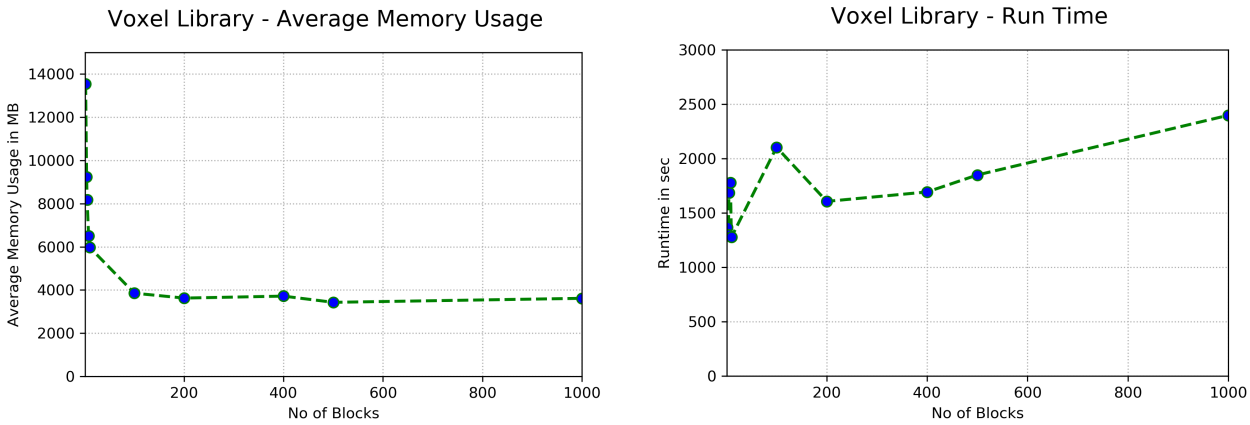

Figure 4.1 (a) Average memory usage and (b) run time of testing on a matrix density of 10%

No of Blocks	Average Memory Usage in MB
2	13545
4	9237
5	8179
8	6514
10	5979
100	3852
200	3626
400	3717
500	3429
1000	3615

Figure 4.2 Table showing average memory usage with increasing number of blocks

4.2 Test 2

- Input matrix dimensions = 1000 × 1200 × 1200
- Matrix density = 20%
- Matrix size = 5.4 GB
- Compressed object size = 2.3 GB

When grey dilation was carried out using the NDImage library of SciPy, the memory usage was 8159 MB with a runtime of 64 seconds using 1 block of operation.
When done with voxelProcessing.py, the minimum average memory usage was 2165 MB using 500 blocks of operation, which was a 73.46% decrease in memory usage. However, the runtime was 717 seconds, as a result of using 500 blocks.

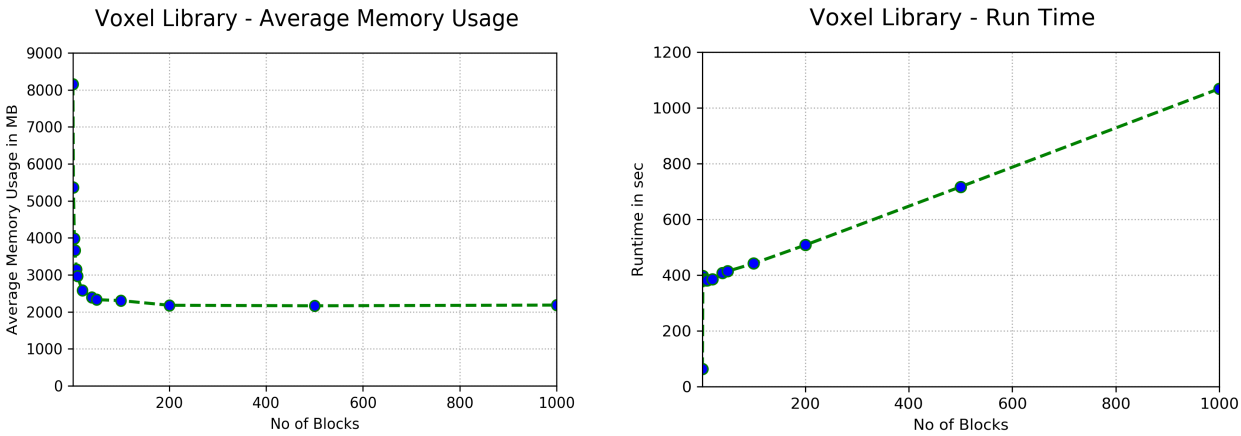

Figure 4.3 (a) Average memory usage and (b) run time of testing on a matrix density of 20%

4.3 Test 3

- Input matrix dimensions = 600 x 700 x 800
- Matrix density = 40%
- Matrix size = 2.6 GB
- Compressed object size = 1.07 GB

When grey dilation was carried out using the NDImage library of SciPy, the memory usage was 3854 MB using 1 block of operation.
When done with voxelProcessing.py, the minimum average memory usage was 1018 MB using 200 blocks of operation, which was a 73.58% decrease in memory usage. The runtime was 192 seconds

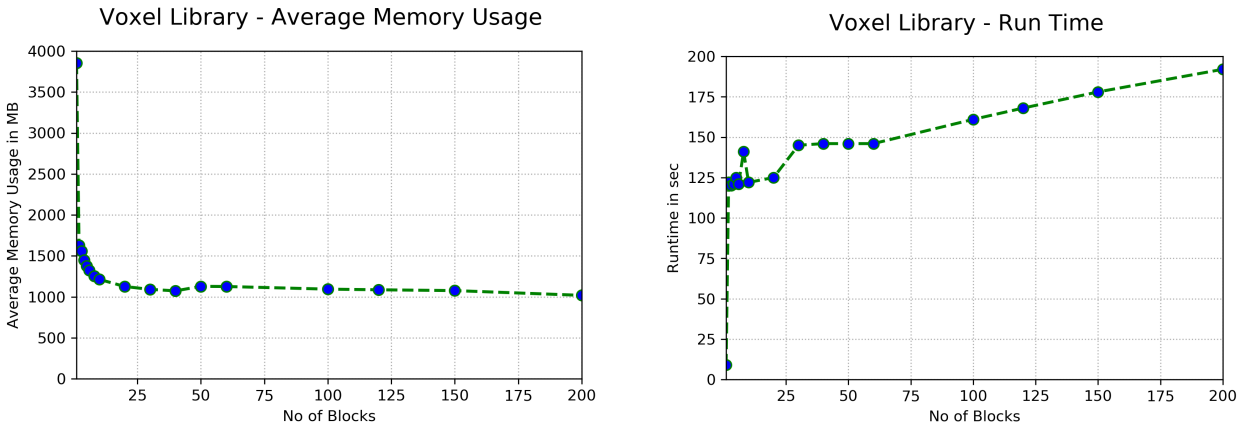

Figure 4.4 (a) Average memory usage and (b) run time of testing on a matrix density of 40%

4.4 Test 4

- Input matrix dimensions = 25200 x 100 x 200
- Matrix densities = 10%, 20%, 30%, 40%

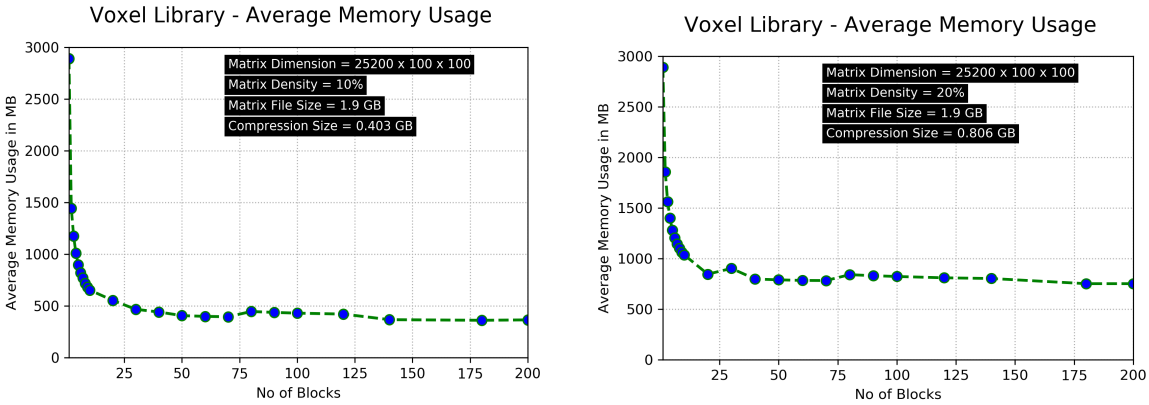

Figure 4.5 Average memory usage of testing on a matrix density of (a) 10% and (b) 20%

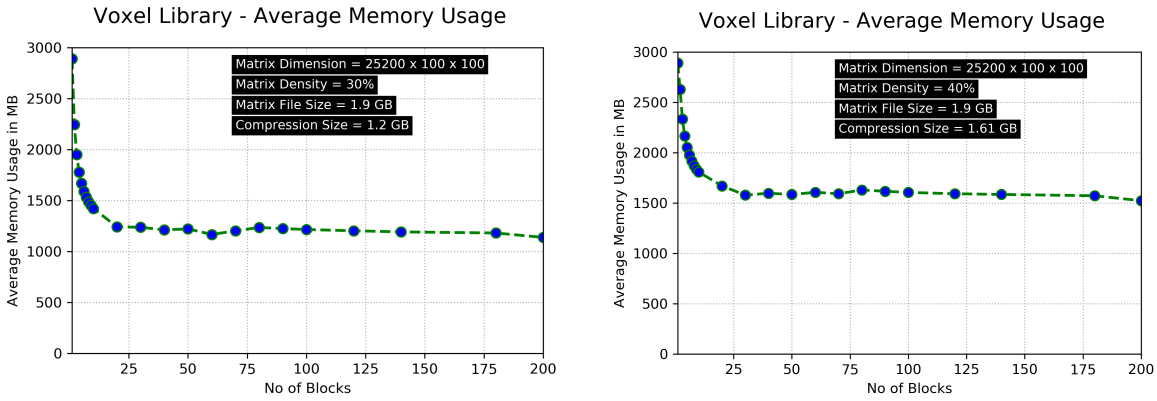

Figure 4.6 Average Memory Usage of testing on a matrix density of (a) 30% and (b) 40%

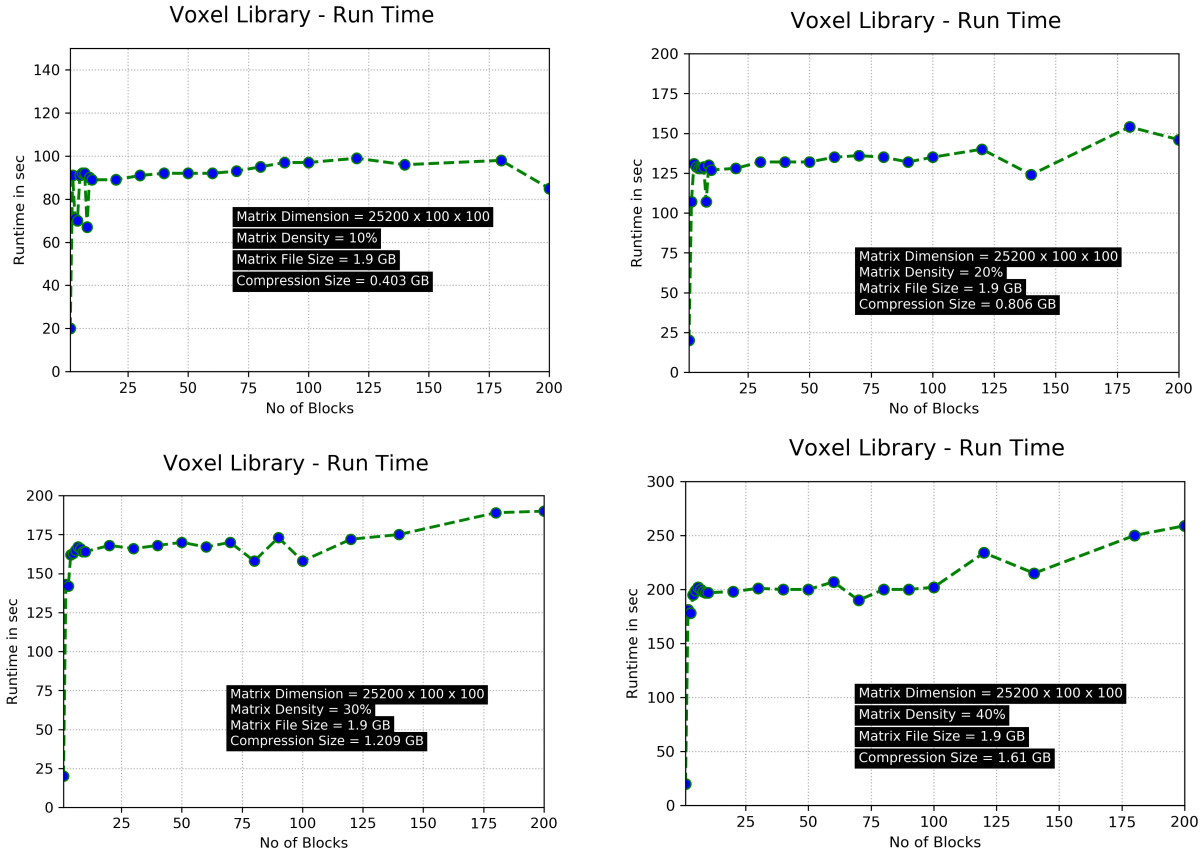

Figure 4.7 Run time of testing on matrix density of (a) 10%, (b) 20%, (c) 30%, and (d) 40%

4.5 Analysis

The threshold of the number of blocks and minimum average memory usage depend on the density of the matrix. For high-density matrices, the library achieves lower memory consumption using fewer number of blocks. On the other hand, for low-density matrices, the library achieves lower memory consumption using higher number of blocks. However, the runtime is directly proportional to the number of blocks.

4.6 Multicore Processing

To decrease the runtime, utilization of multicore processing was considered. But in the current approach, the only part where Single Instruction, Multiple Data (SIMD) operations are performed is during morphological block operations. During those operations, the SciPy library already takes advantage of multicore processing. Therefore, utilizing multicore processing libraries and performing multiple block operations in parallel will be not helpful. Also, it may cause the system to run out of memory. In order to improve run time, use the least number of blocks required to solve the memory issue so that the runtime can be kept to a minimum. The number of blocks depends on the available RAM, density of the matrix, i.e. how sparse the matrix is, and the dimensions of the matrix size. The selection of the number of blocks should be such that it utilizes the maximum possible RAM during operation on each block.

Fig. 4.8 shows that the system utilizes multicore processing. Using 80 blocks during morphological operations, the CPU multicore utilization varies significantly. In Fig. 4.9, it can be seen that by using 5 blocks CPU utilization is more in a multicore environment.

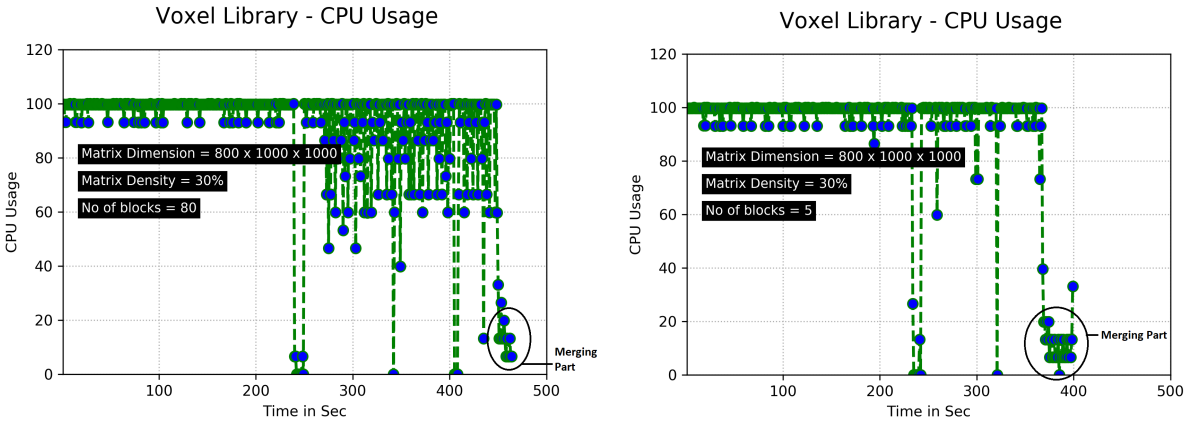

Fig. 4.8 CPU usage of model voxel library using (a) 80 blocks and (b) 5 blocks

4.7 Distributed Computing

Implementation of parallel block operations in a distributed environment can be used to improve runtime efficiency and performance. The main reason for using parallel processing is to reduce the computation time required for what would otherwise be very long-running programs.

5 References

1. Voxel-based Solid Models: Representation, Display and Geometric Analysis: https://pdfs.semanticscholar.org/b2ec/9d0c9fadba4174e1ae5d29fccb7b344dc365.pdf
2. RAM vs. CPU: https://www.quora.com
3. Data processing in computers: peda.net
4. How to slice and cut STL files for 3D printing: https://www.youtube.com
5. Splitting 1 STL into 4 STLs: https://www.youtube.com
6. Efficient data-compression methods multidimensional sparse array: https://ieeexplore.ieee.org/
7. Working with big data in Python and NumPy: https://stackoverflow.com/questions/16149803
8. Image manipulation and processing using NumPy and SciPy: scipy-lectures.org
9. SciPy.org API: https://docs.scipy.org/doc
10. Split and merge algorithms: https://hal.archives-ouvertes.fr
11. Split a 3D NumPy Array into 3D blocks: split-a-3d-numpy-array-into-3d-blocks
12. Numerical Computing with NumPy: https://www.youtube.com
13. Merging two or three 3D array blocks: https://stackoverflow.com
14. Combining 3 arrays in one 3D Array in NumPy: combining-3-arrays-in-one-3d-array-in-numpy
15. K. Nishio, K. Kobori, T. Kutsuwa, and Y. Nishikawa. Three-dimensional morphological filter using spatial partitioning representations. Dec. 12, 1999. Retrieved Sept. 24, 2018: https://onlinelibrary.wiley.com/doi/pdf
16. Octree in python. Retrieved Sept. 25, 2018: http://www.anderswallin.net
17. Sparse voxel octrees. Retrieved Sept. 25, 2018: sparse-voxel-octrees
18. Check current CPU and RAM usage in Python: how-to-get-current-cpu-and-ram-usage-in-python
19. Neighborhood and block Operations: http://radio.feld.cvut.cz
20. Memory management: https://docs.python.org/3/c-api/memory.htmlg
21. DASK API: http://docs.dask.org/en/latest/
22. DASK Tutorial: https://github.com/dask/dask-tutorial, https://dask.org/
23. DASK Parallel: introducing-dask-parallel-programming.html
24. Out of core data frames: out-of-core-dataframes-in-python/
25. OpenVDB: http://www.openvdb.org/
26. Scipy NDImage binary morphology: scipy.ndimage.morphology.binary_erosion.html
27. NDImage morphology: ndimage.html#morphology
28. Image measure marching cubes: http://scikit-image.org
29. Marching cubes: https://en.m.wikipedia.org/wiki/Marching_cubes
30. Memory mapping: http://www.roboticsproceedings.org
31. OpenVDB - Python: http://www.openvdb.org
32. Voxel data structure suggestions: https://gamedev.stackexchange.com
33. Octree storage vs. array storage: https://pdfs.semanticscholar.org
34. Plot 3D images: https://terbium.io/2017/12/matplotlib-3d/
35. NumPy matrix memory size: numpy-matrix-memory-size-low-compared-to-numpy-array