**Memory-Efficient Voxel Processing Library**



***Submitted to***

***Professor Kenneth Fletcher***

***Dept. of Computer Science***

***University of Massachusetts Boston***

**By**

**Radhikabahen D Patel Quan Tran**

**Jithin Jacob Senthil Palanivelu**

Table of Contents

[**Introduction**](#_k7lc28kkt6pc) **2**

[**Literature Survey**](#_8xdgwfau2l2g) **3**

[Octree](#_wc91we2l0ufr) 3

[OpenVDB](#_22ks639sjc7u) 4

[Memory Mapping](#_mhc4xxza0cm8) 4

[Compression of Sparse Matrices](#_kpvgl0mym680) 4

[Morphological Operations](#_lr3kgk7xaohm) 5

[**Algorithm**](#_m691is72222b) **5**

[Software and Module Requirements](#_bl4o2h378kjs) 8

[Implementation](#_vz7np8q739s1) 8

[Directory Architecture](#_bukua9awhvqj) 11

[**Tests and Analysis**](#_nwdys0atdj5y) **11**

[Test 1](#_l2op51ttcme8) 12

[Test 2](#_9vhpgemujqfm) 13

[Test 3](#_ofuif4n6080g) 13

[Test 4](#_3sfzfbsbvgrt) 14

[Analysis](#_2kgs7ff7mine) 15

[Multicore Processing](#_97l9z014lui3) 15

[Distributed Computing](#_yp183gfrztrf) 16

[**Conclusion**](#_ag6eynugyo79) **16**

[**References**](#_lhk73romifd6) **16**

# 

# **Introduction**

In 3D printing, modelling of the object is done with the use of computer-aided design, which is in the form of voxels. 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. However, the memory footprint scales with the cell size of the volume being described, leading to large voxel data structure, leading to very large matrices required to represent the data structure. This tends to occupy considerably more memory than something such as a tessellated surface model.

Very large matrices require a lot of memory, and this is often a computational bottleneck in large computed-imaging problems. Some very large matrices that we wish to work with are sparse, which is what this project will be focussing on. Sparse matrices are matrices that have many more zero values than non-zero values. 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, Product Innovation and Engineering, LLC., wanted to perform morphological and basic arithmetic operations on voxel images whose size is larger than the RAM of the system that was used. 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 developed module voxel using NumPy and SciPy modules, which provide morphological operations using divide and conquer approach.

# **Literature Survey**

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

## Octree

The first data structure under consideration was the octree. An octree is a data structure where 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. This data structure helps obtain high speeds during morphological operations. The octree also provides an extremely rigorous compression as compared with other 3D spatial subdivisions approaches. This means that a relatively enormous space containing a very small amount of voxel solids can be efficiently stored using an octree.

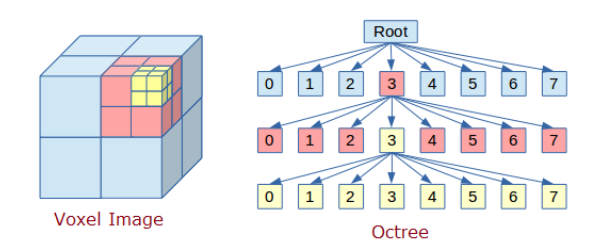


Figure 2.1 Octree Data Structure

However, the drawback of the octree is that it doesn’t 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.

## OpenVDB

The next approach that was considered was using the OpenVDB library. It is an open-source C++ library that uses a novel hierarchical data structure and comes with its own suite of tools which can be used for the efficient storage and manipulation of sparse volumetric data on three-dimensional grids. The drawback of this library was the fact that it runs on Python 2.7, but the version used by the client that is facing the above stated problem used Python 3.

## Memory Mapping

Another technique that was examined was memory mapping. This technique deals with files by mapping them into your processor's address space. Memory mapping a file uses the operating system's virtual memory system to access the data on the filesystem 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 data between buffers—the memory is accessed directly.

## Compression of Sparse Matrices

As mentioned before, representing a sparse matrix by 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 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. Assuming a non-symmetric sparse matrix, three vectors are created: one for floating point numbers (val), and the other two for integers (col\_ind and row\_ptr). The val vector stores the values of the non-zero elements of the matrix, as they are traversed in a row-wise fashion. The col\_ind vector stores the column indexes of the elements in the val vector.

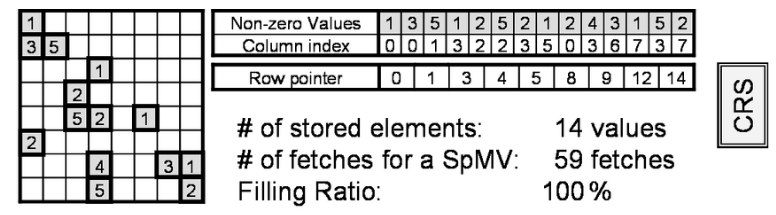


Figure 2.2 CRS Compression Example

## 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.

# 

# Algorithm

The first step is to check whether the input array is sparse. If 50% 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 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.

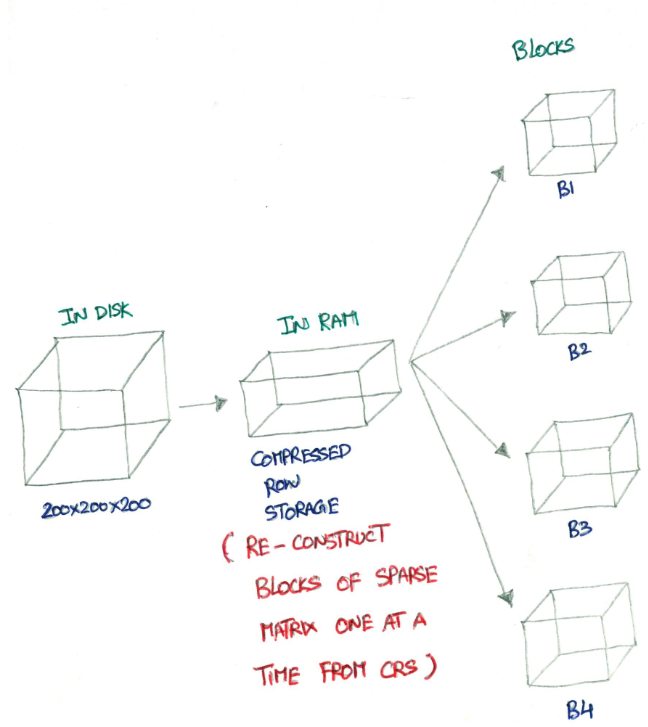


Figure 3.1 Basic Idea

Then, the required morphological operation is carried out using block operation. In block operation, an image is processed in blocks rather than all at once. The blocks have the same size across the image. 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 fed to the RAM instead, blocks of submatrices are fed. The benefit of block-wise computation can be extended to distributed work on different streams. For algorithms that expect multiple samples of one or more time 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.



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 how to calculate the 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. Each block receives a vector of ghost cells from neighboring block.

After the morphological operations have been carried out, the merging operation can merge multiple same-sized blocks. The blocks to be “merged” are stored contiguously in disk memory. Merging allows you to combine continuous blocks on the same shard into a single block.

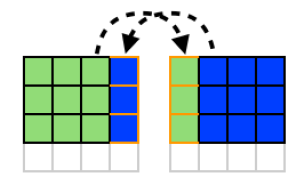


Figure 3.3 overlapping block division

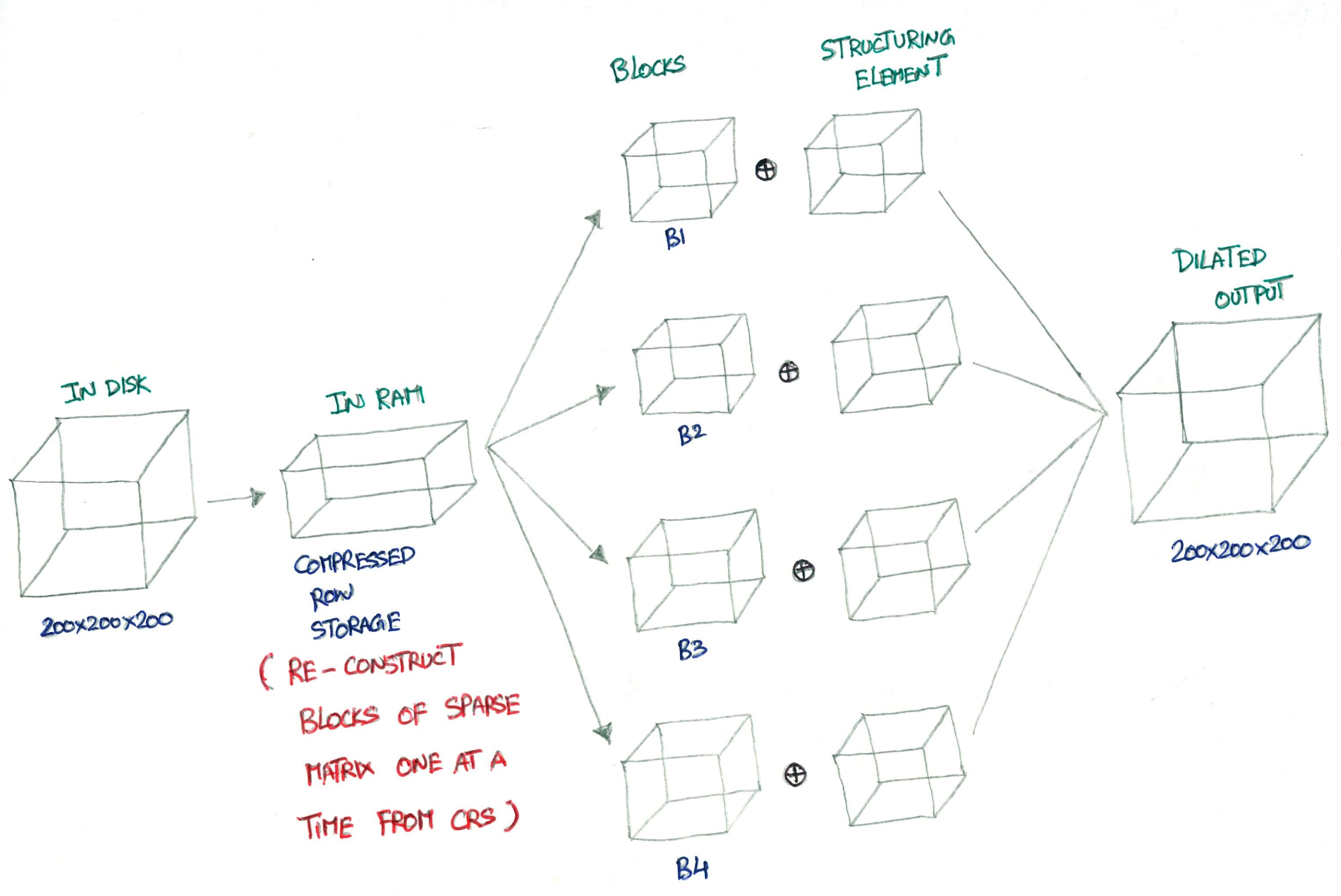


Figure 3.4 Algorithm architecture

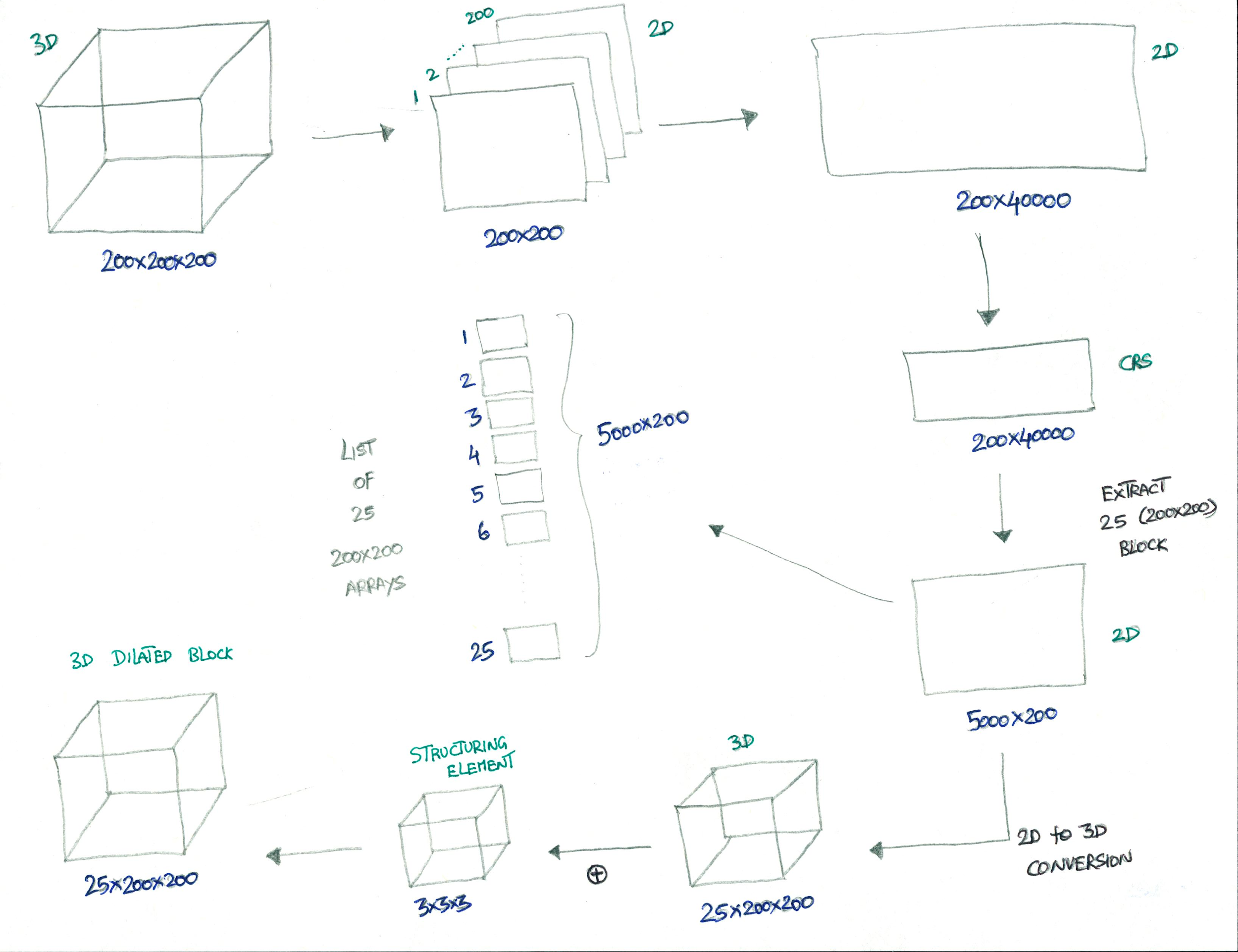


Figure 3.5 Architecture for a single block operation

## 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.

## 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
   1. binary\_erosion(…)
   2. binary\_dilation(…)
   3. binary\_opening(…)
   4. binary\_closing(…)
   5. binary\_fill\_holes(…)
   6. binary\_hit\_or\_miss(…)
   7. binary\_propagation(…)
2. Grey-scale Morphology
   1. grey\_erosion(…)
   2. grey\_dilation(…)
   3. grey\_opening(…)
   4. grey\_closing(…)
   5. morphological\_gradient(…)
   6. morphological\_laplace(…)
   7. white\_tophat(…)
   8. black\_tophat(…)
3. Custom Functions
   1. multiply(…)

Integer/float value multiplication of each element with given scalar value.

* 1. nothing(…)

It does not perform any morphological or arithmetic operations. It just does 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

1. no\_of\_blocks

Type: int

Description: The number of blocks (with respect to x-axis) into which to divide the

original image. Default: 4

1. fakeghost

Type: int

Description: The number of extra rows or ghost cells around around each block required for morphological operations, generally proportional to the structuring element. Default: 2. But It need to be greater than 1.

1. make\_float32

Type: boolean

Description: Whether to type-cast input array to float32. Default: True

1. Other parameters for specific morphological Operations, ex 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 the all the blocks, it then reads data from the created file and returns the whole array as the output.

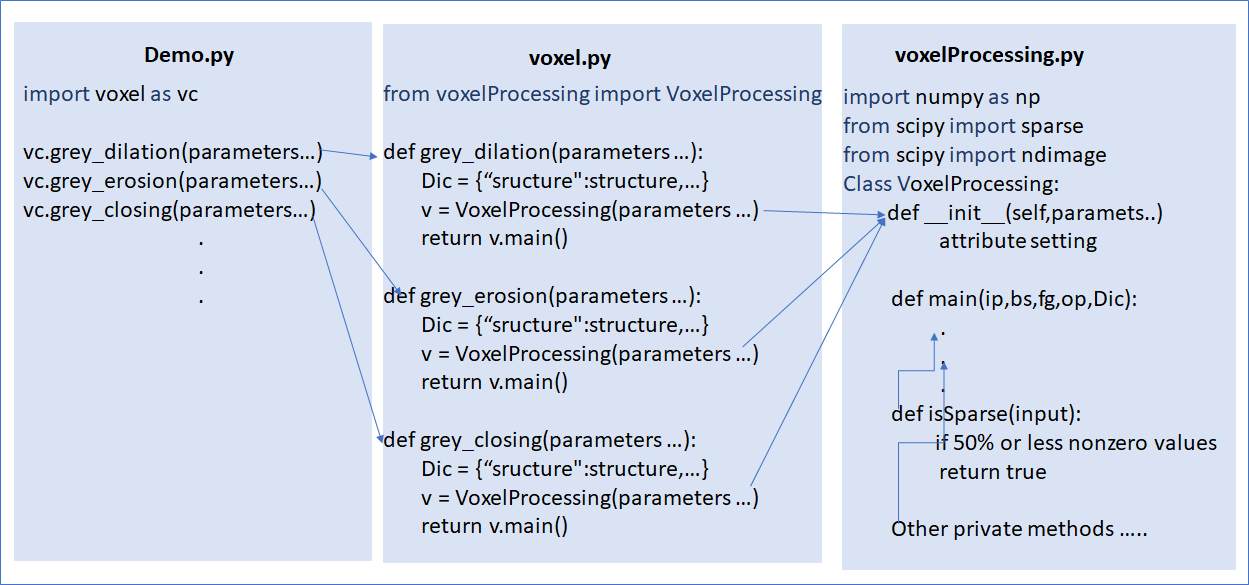


Figure 4.1 Module API

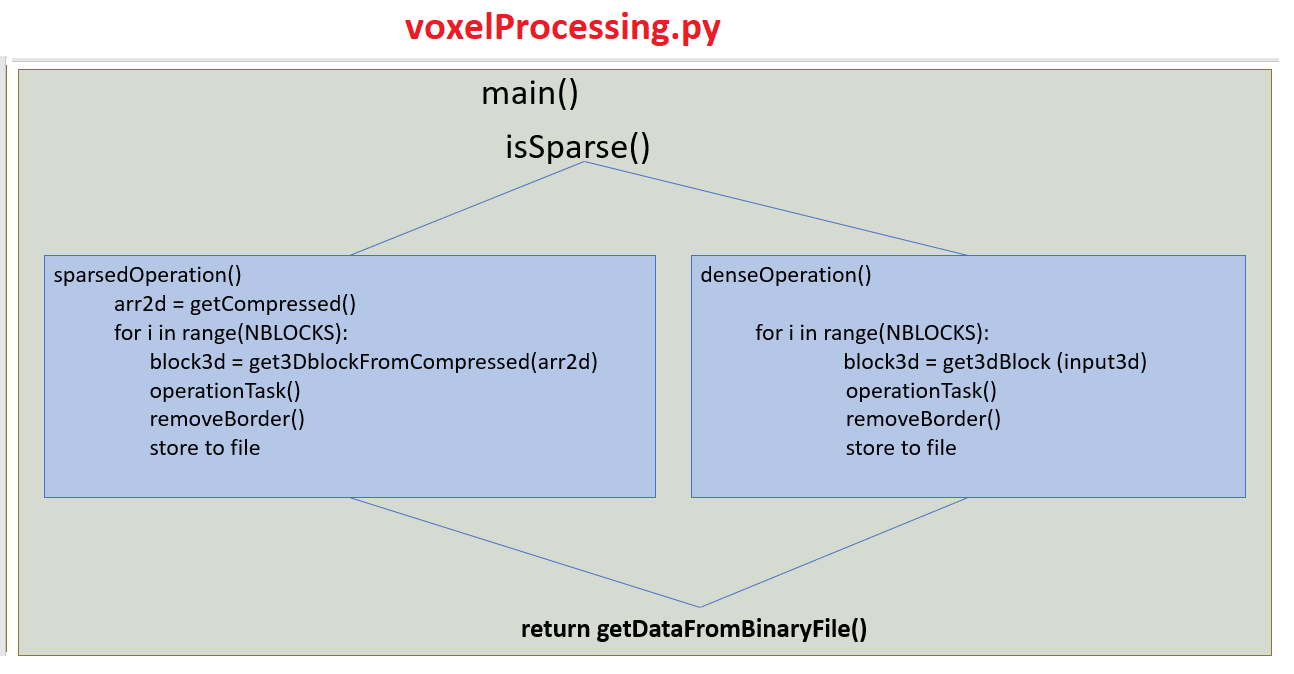


Figure 4.2 Approach Implementation

## 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

# Tests and Analysis

For testing, grey dilation was carried out on 4 input arrays of different matrix dimensions and densities to determine runtime and average memory usage.

## 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 SciPy's ndimage library, 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. 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.

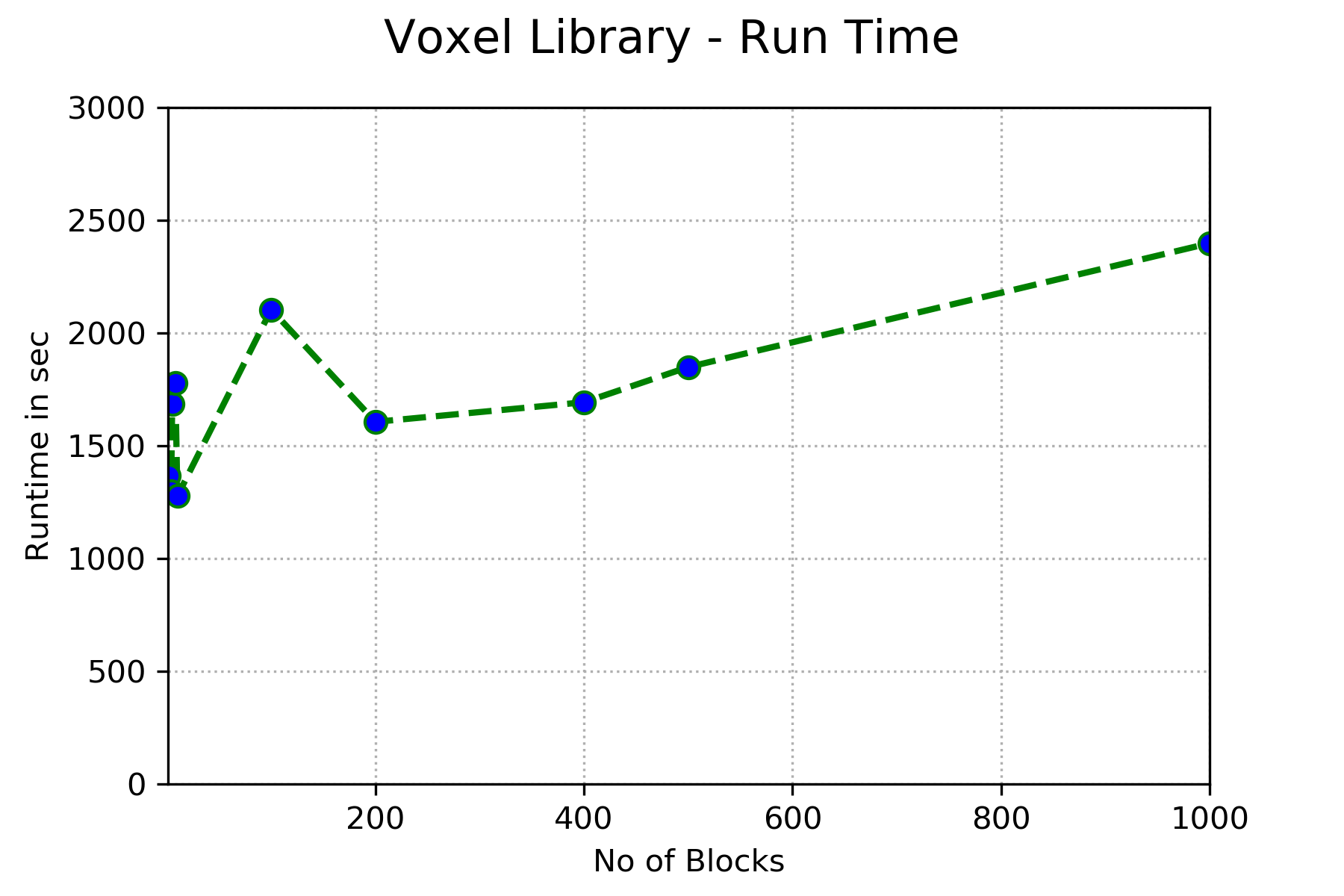
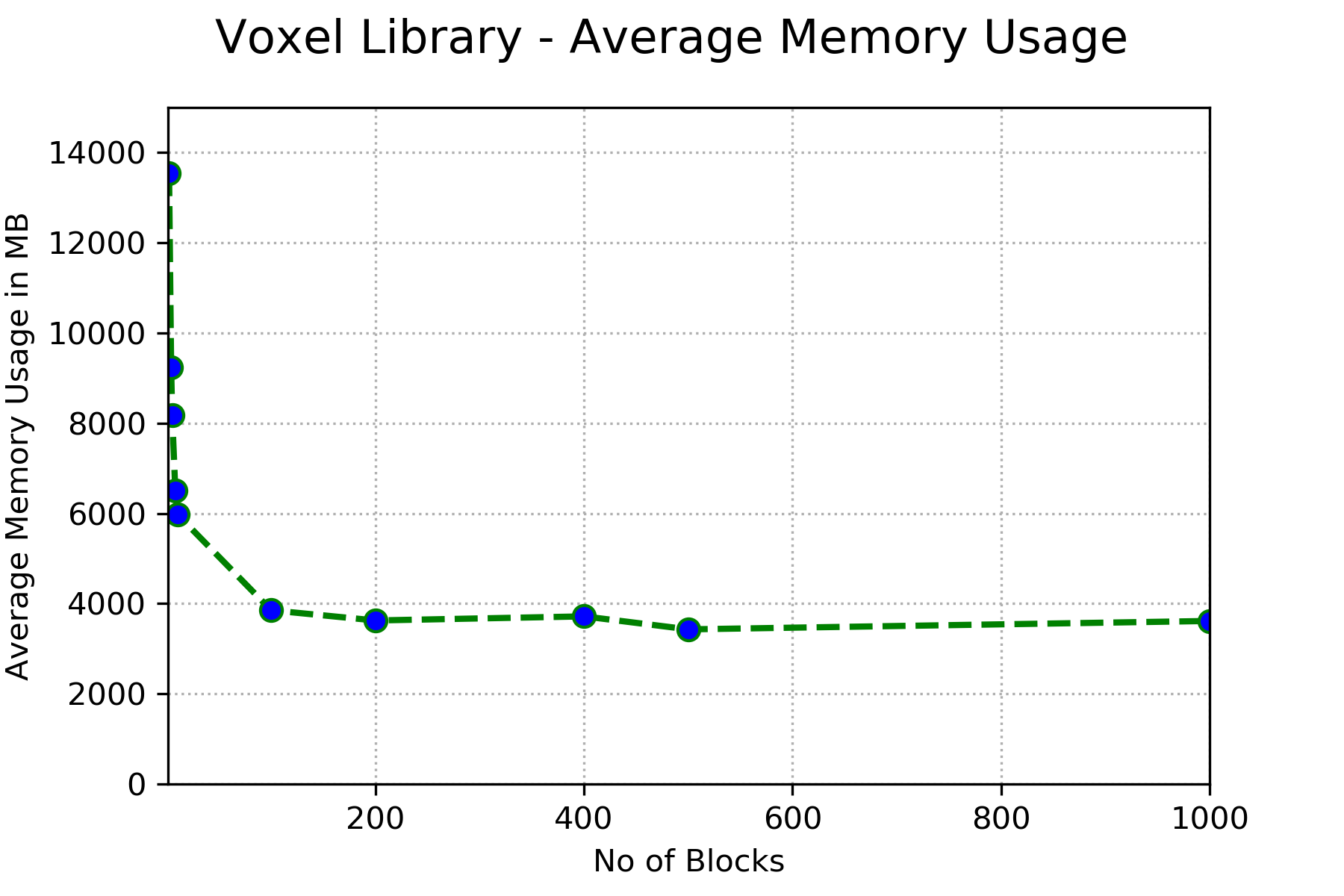


Figure 5.1

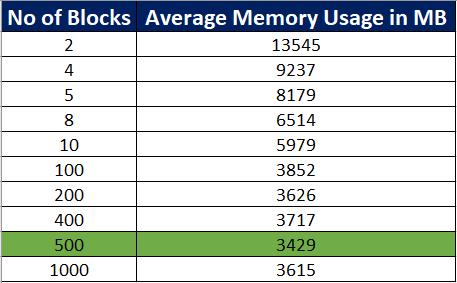


Figure 5.2

## 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 SciPy's ndimage library, 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.

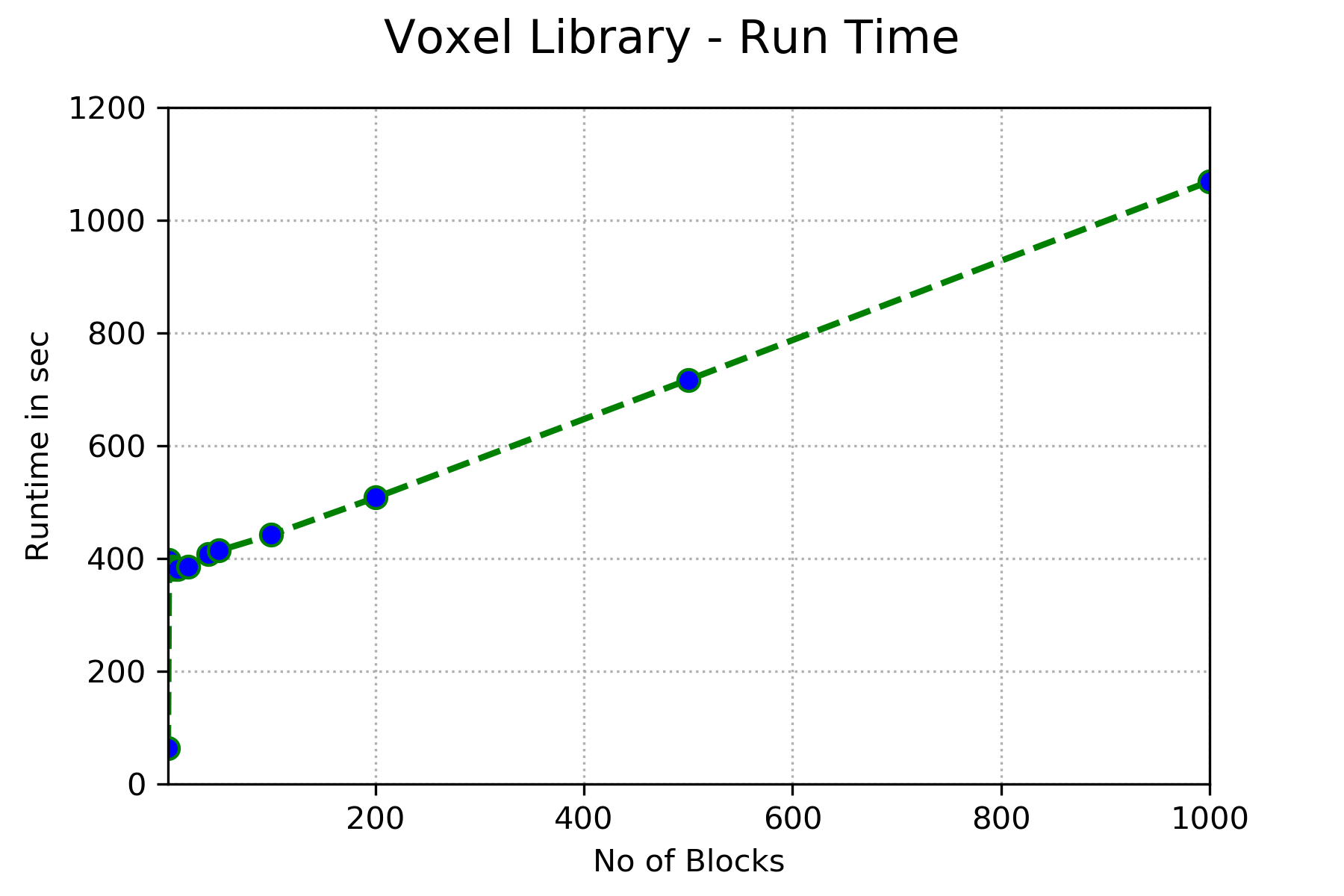
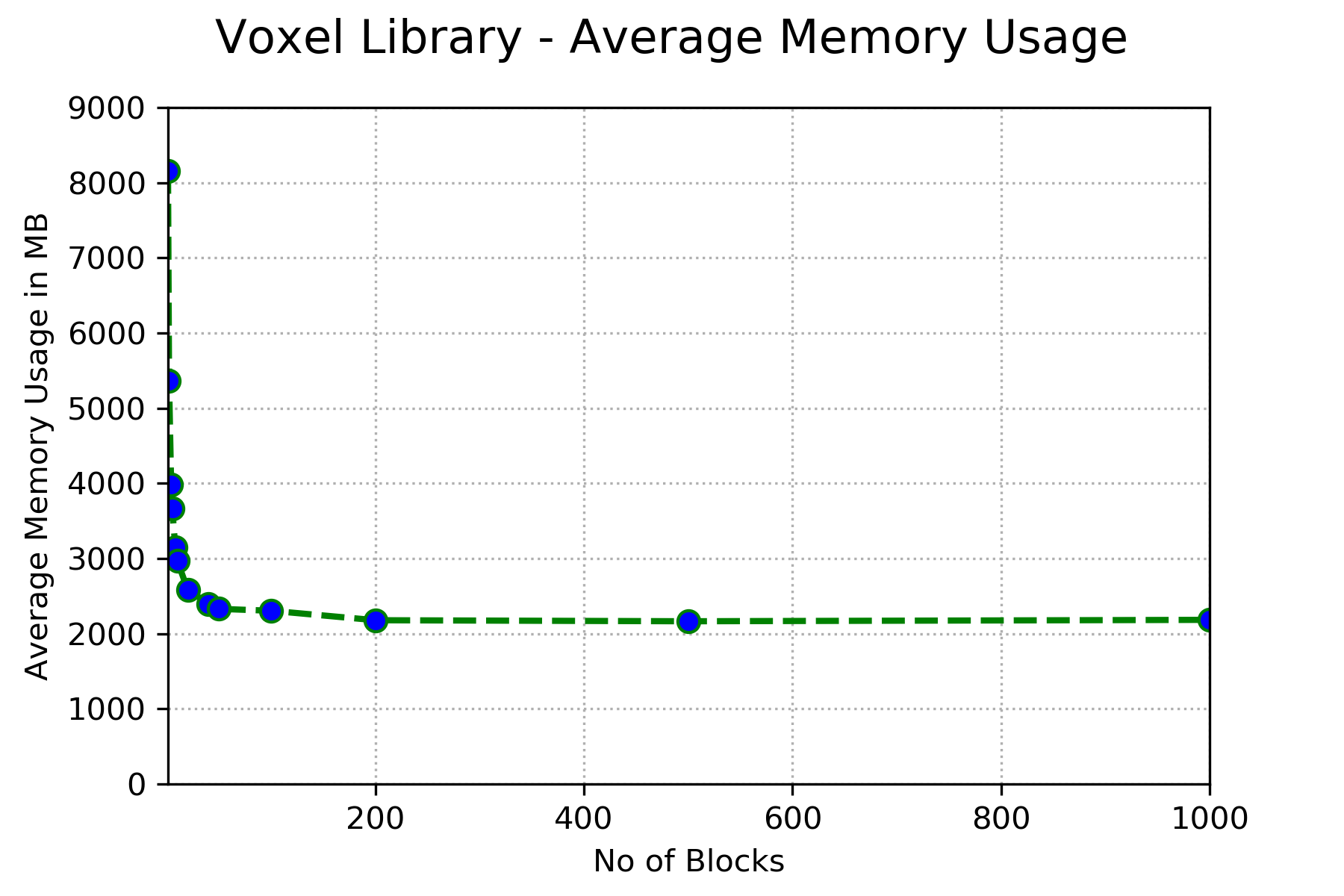


Figure 5.4

## 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 SciPy's ndimage library, 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

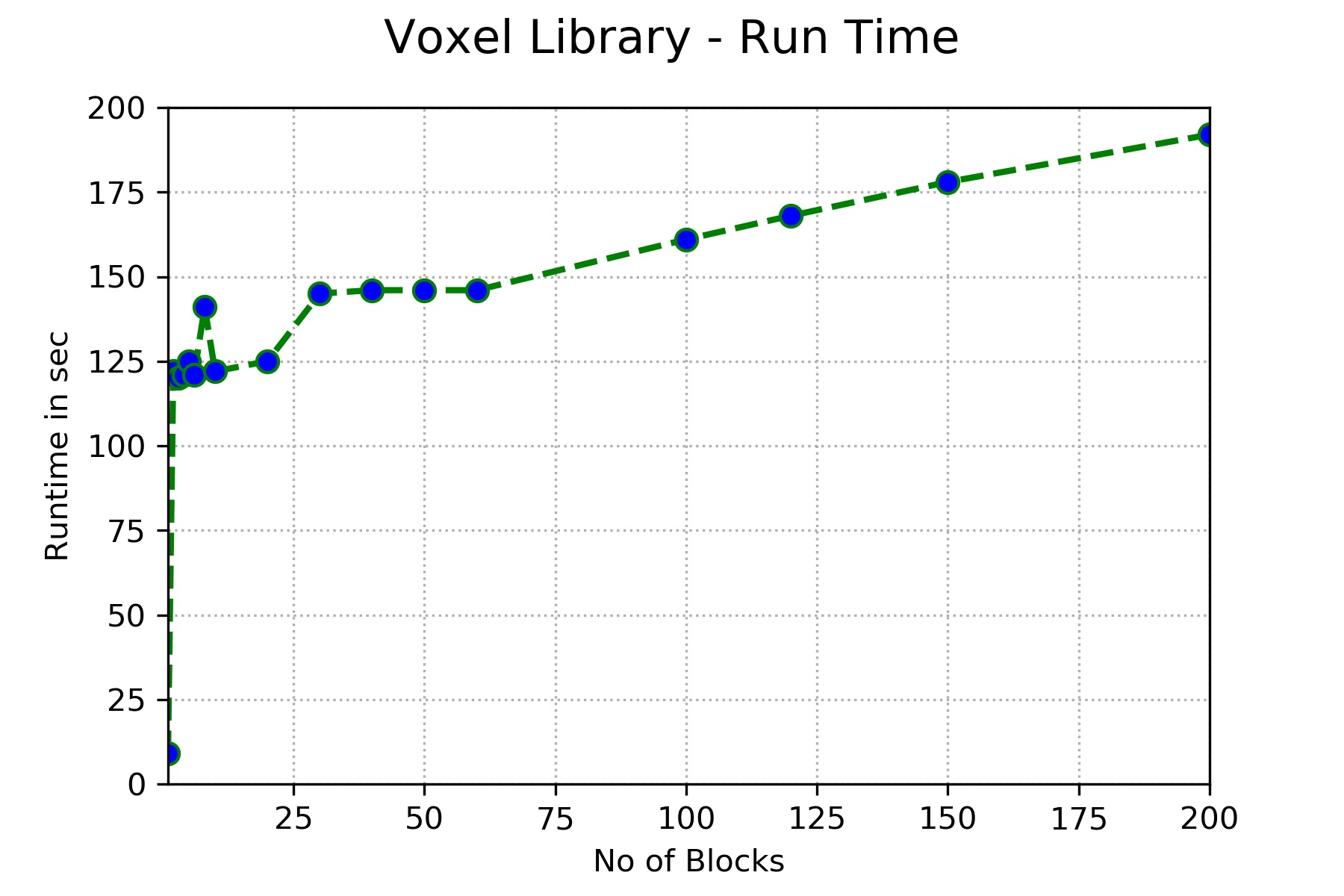
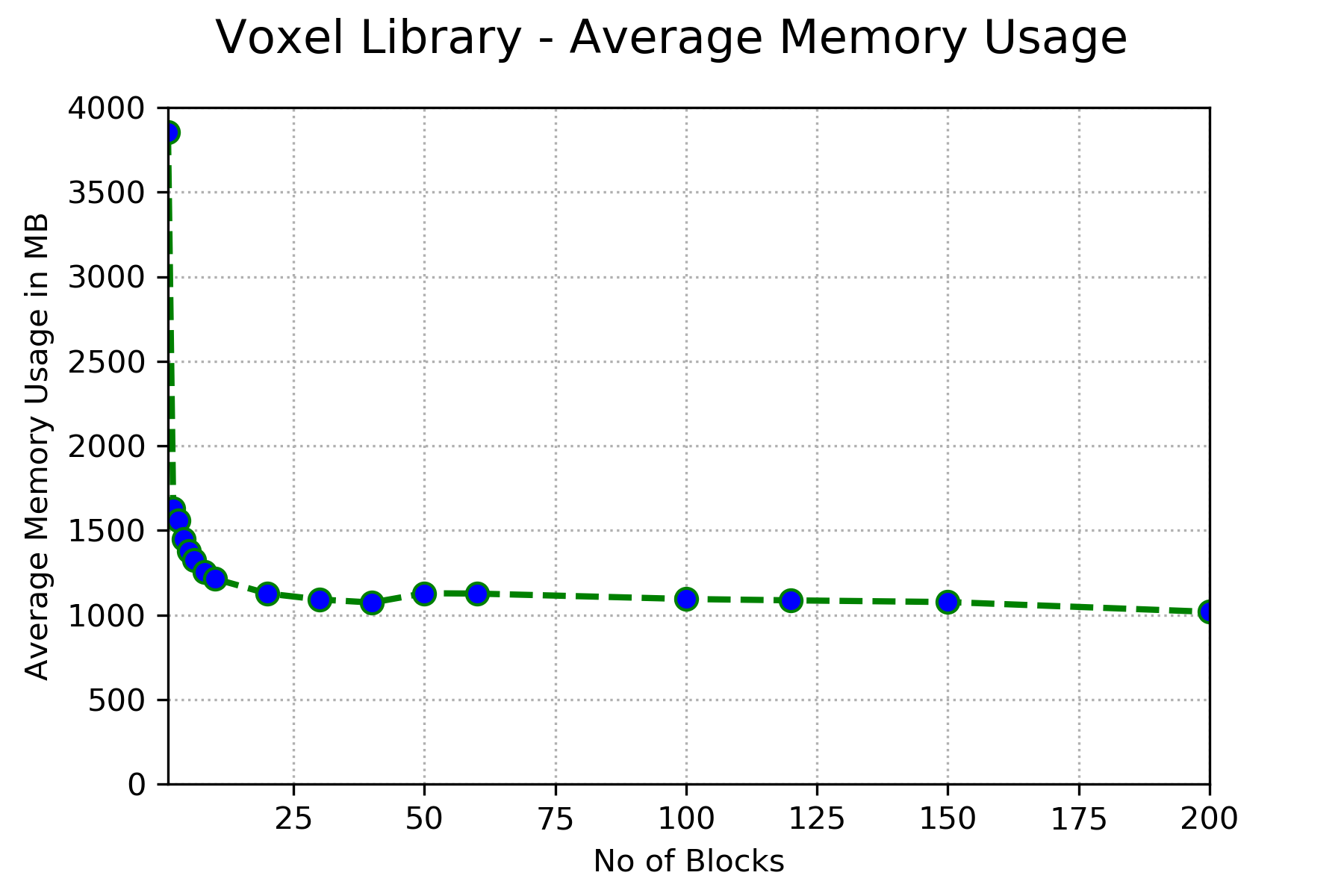
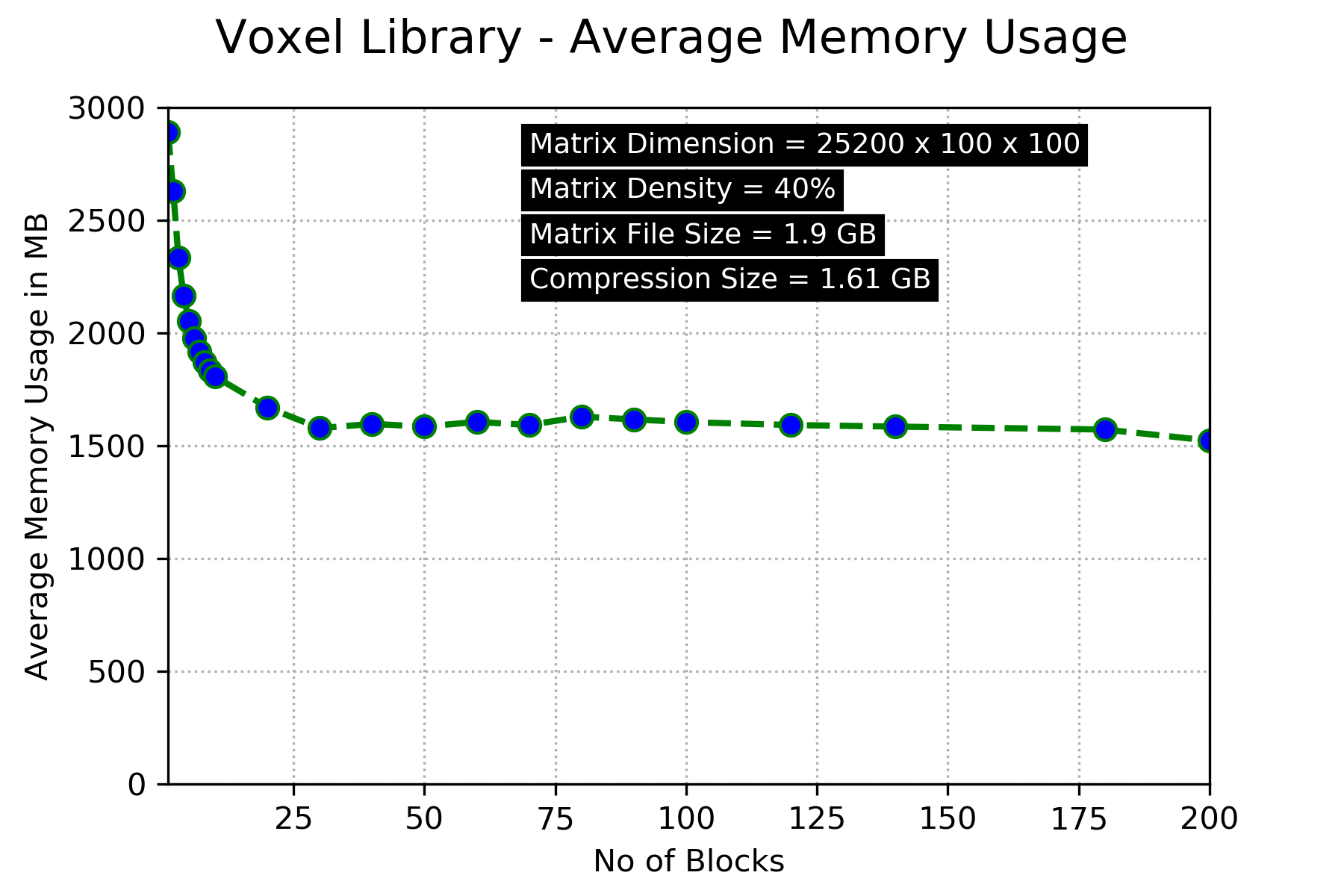
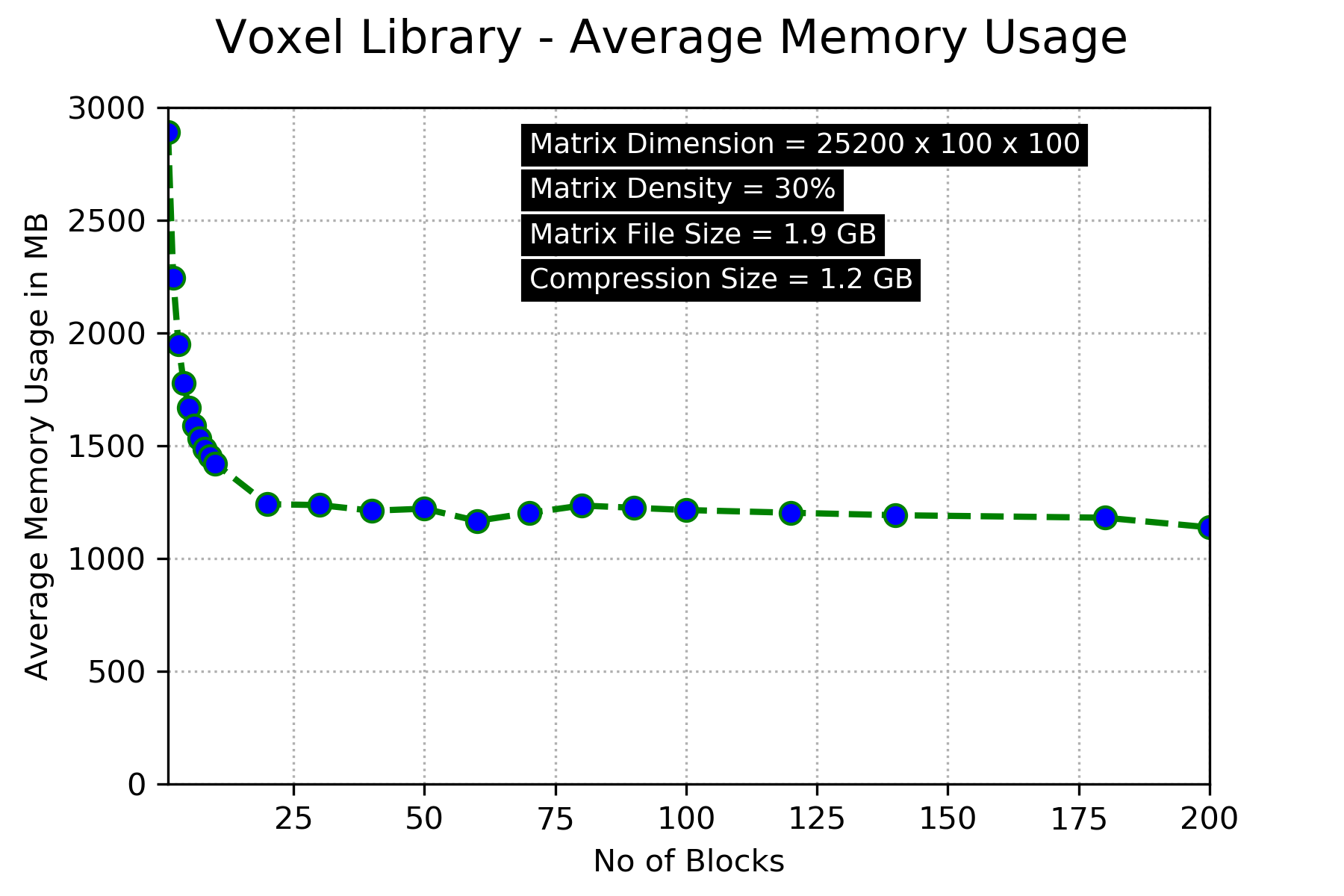
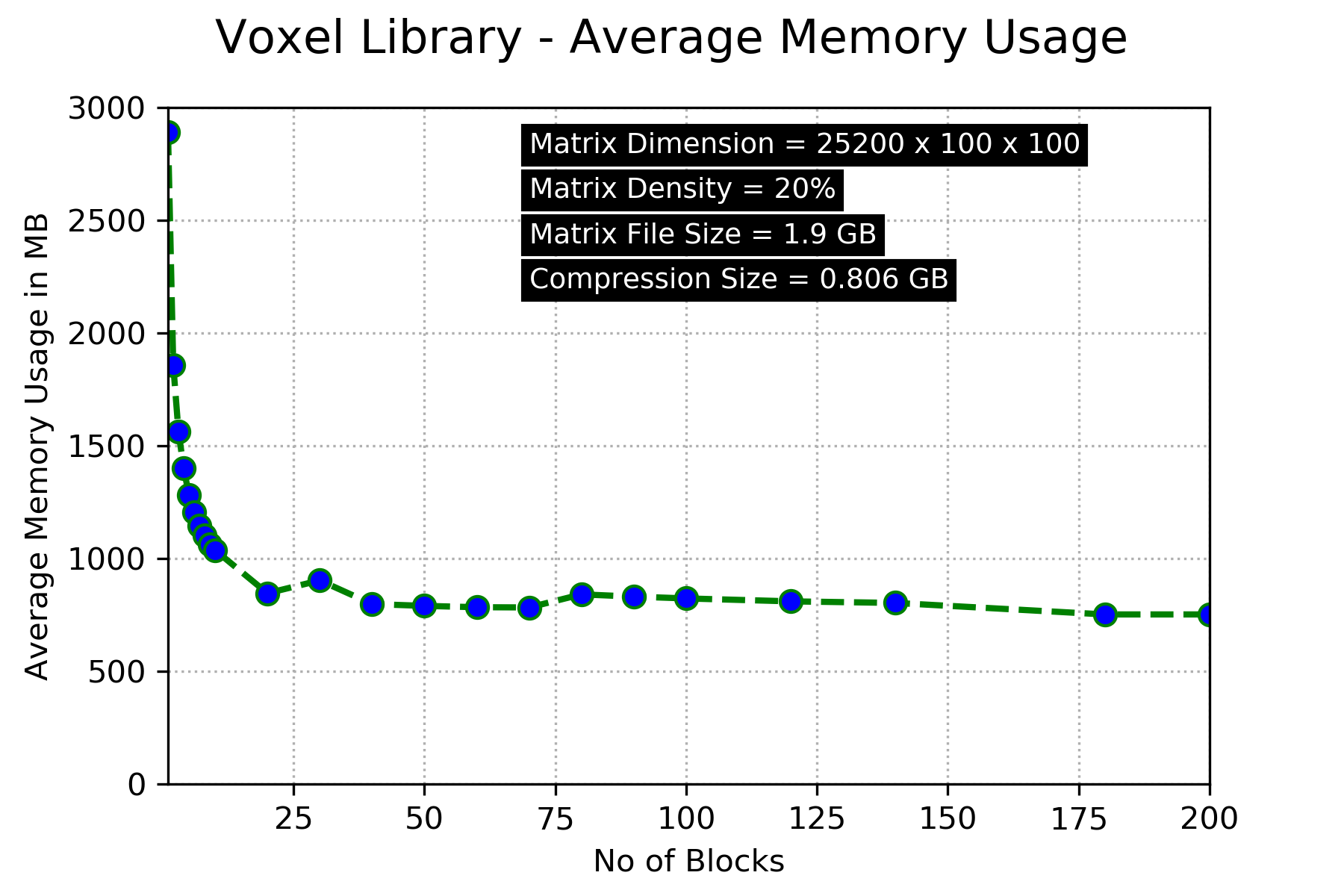
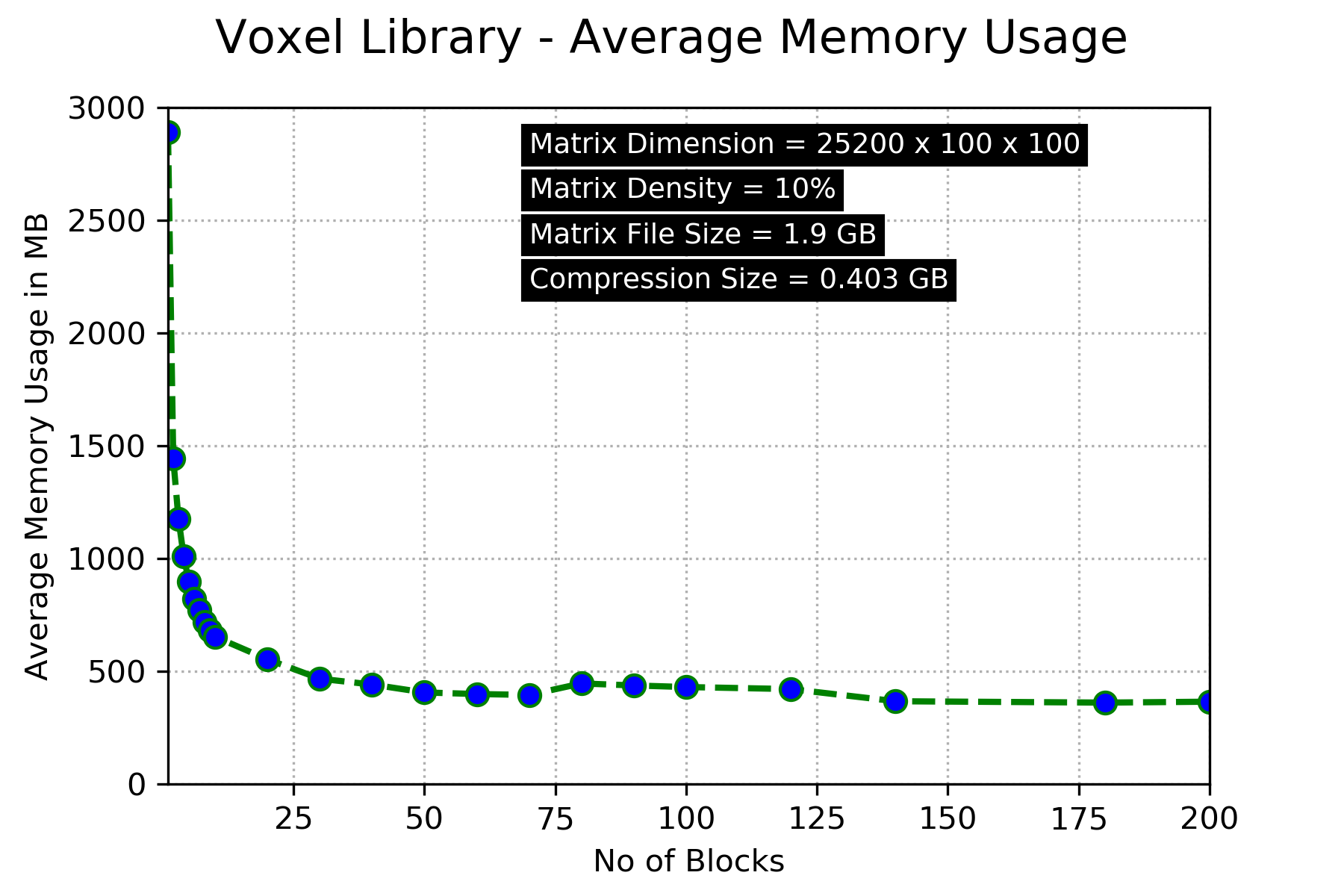
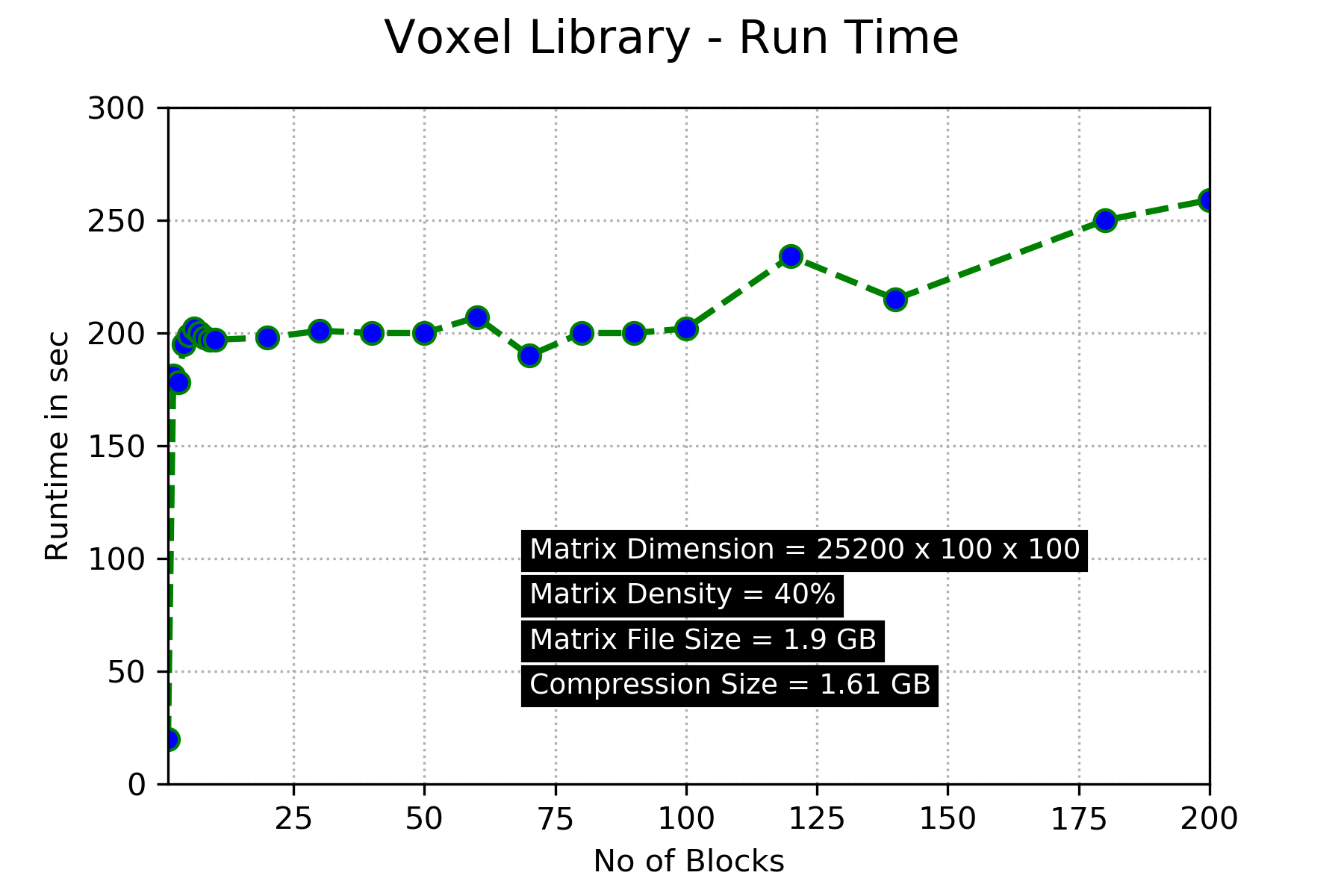
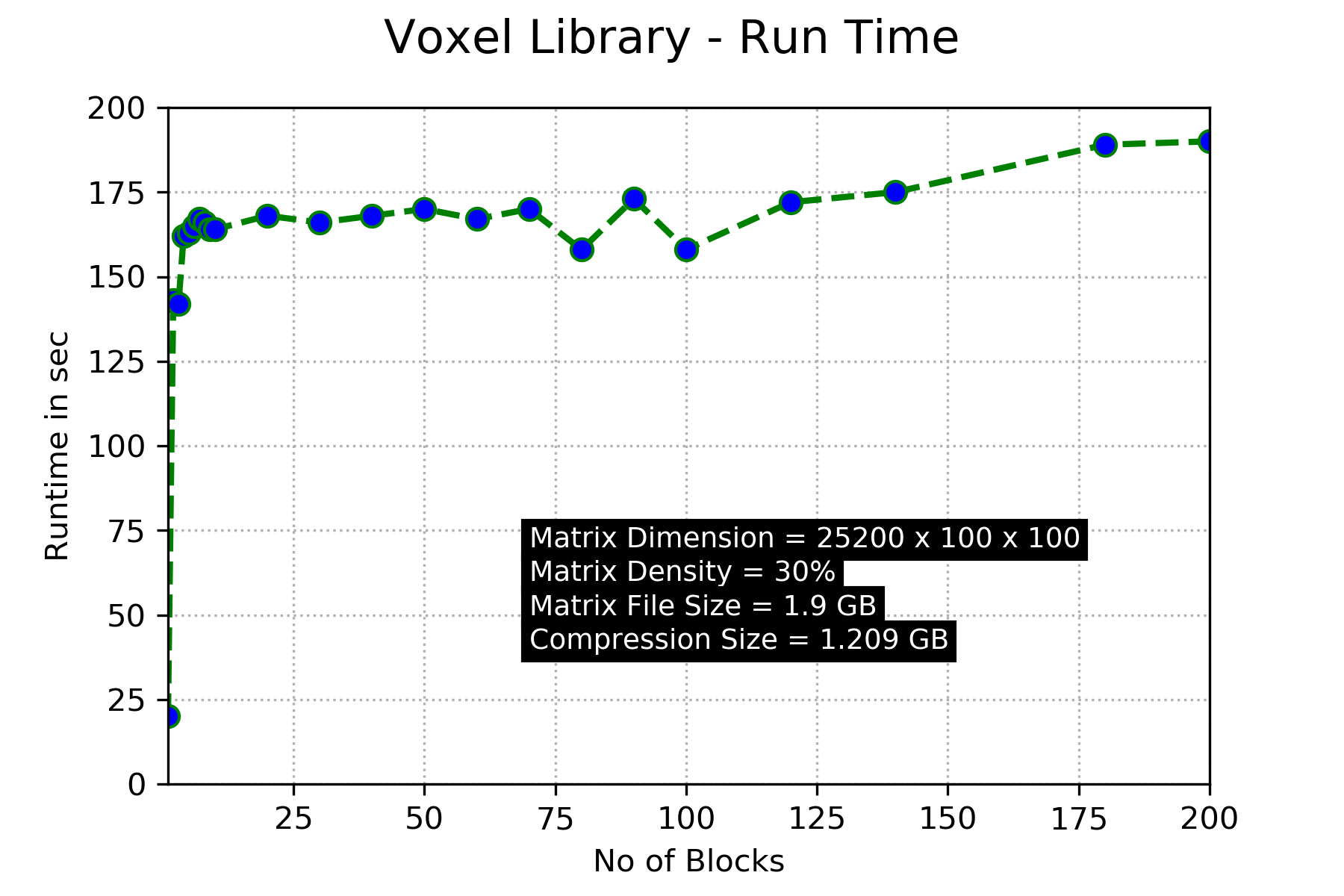
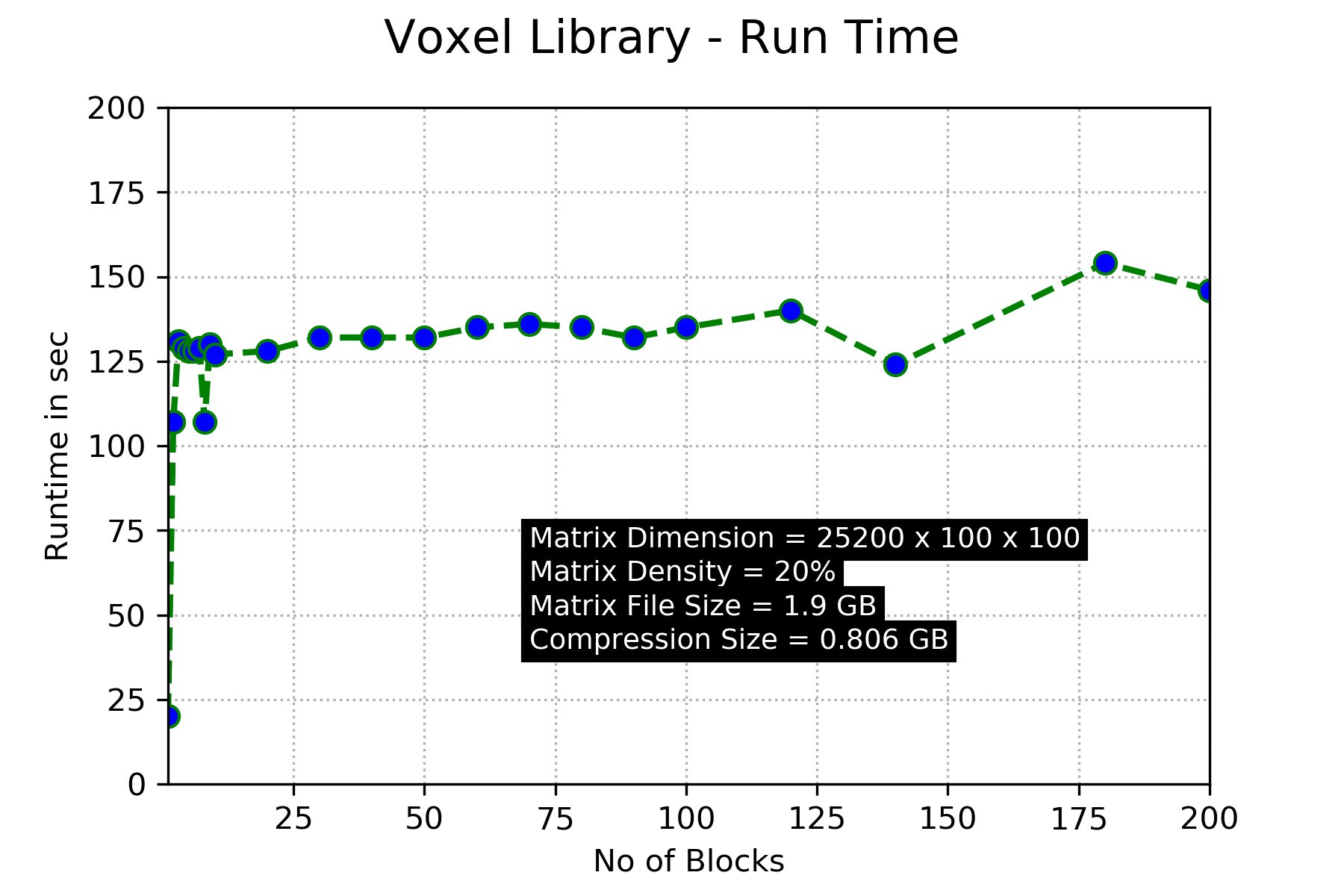
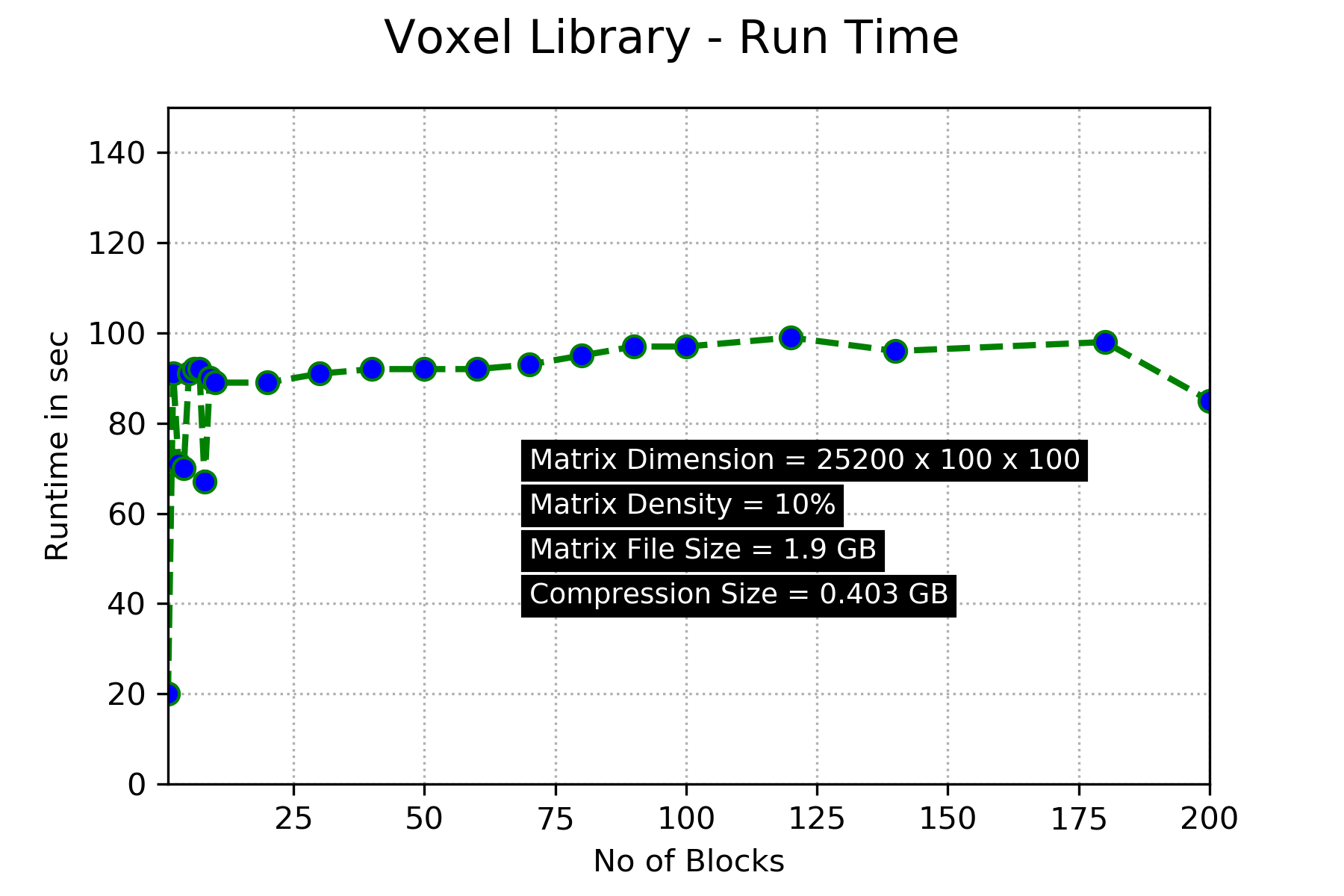


Figure 5.5

## Test 4

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



****

## 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 no of blocks. On the other hand, for low-density matrices, the library achieves lower memory consumption using more no of blocks. However, the runtime is directly proportional to the number of blocks.

## 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. So, to utilizing multicore processing libraries and performing multiple blocks operation in parallel will be not helpful here plus you might end with memory out of RAM issue So to improve run time you should make as possible as less no of block so you can solve your memory issue but you don’t have to pay more run time for it. No of blocks depend on your RAM and density of matrix plus your matrix size (dimensions).

So you will select no of block such that you utilize highest RAM during each block’s operation

Fig. 5.1 shows that the system utilizes multi core processing. Using 80 blocks during morphological operations, the CPU multi core utilization varies significantly. In Fig. 5.2, it can be seen that by using 5 blocks CPU utilization is more in a multicore environment

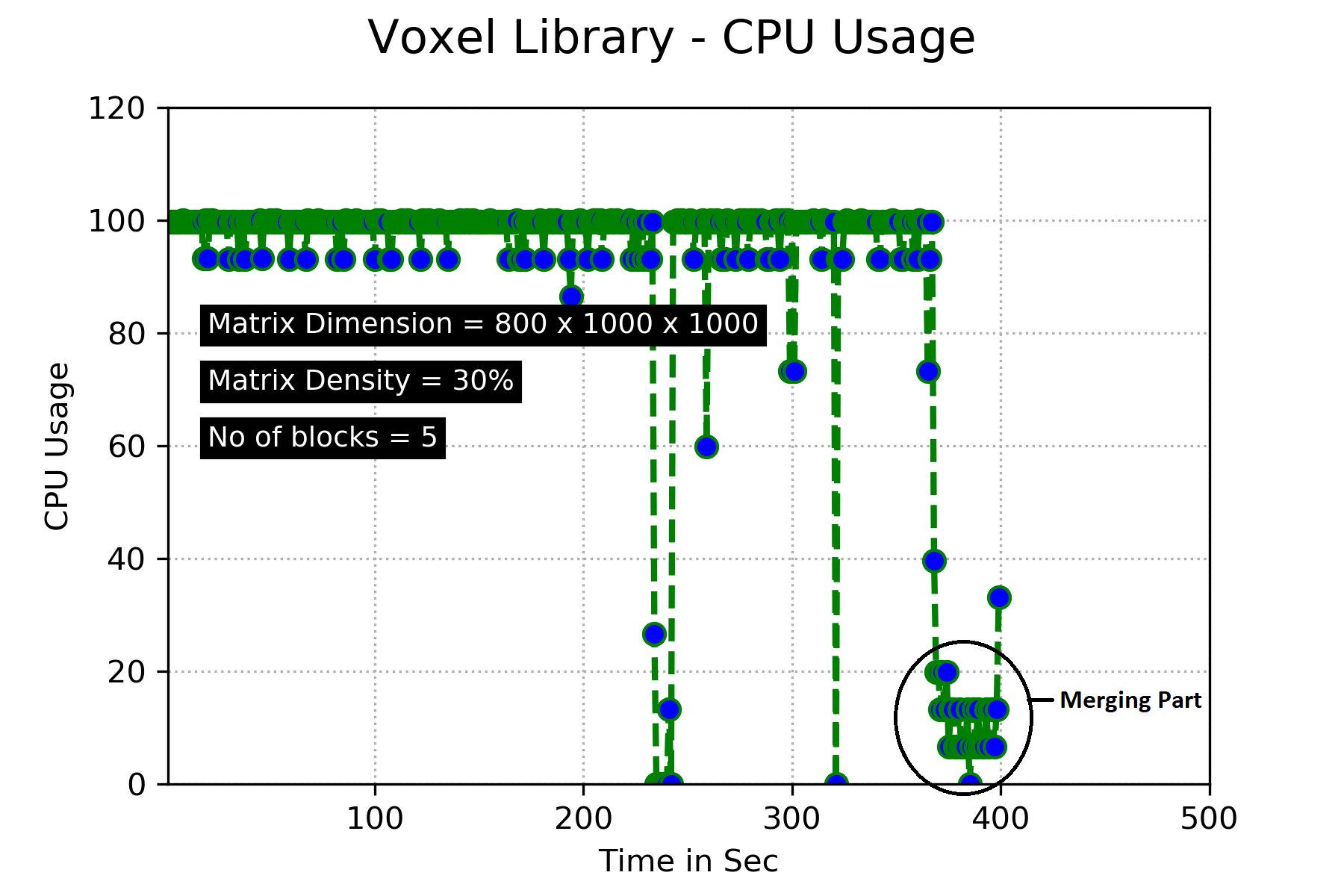
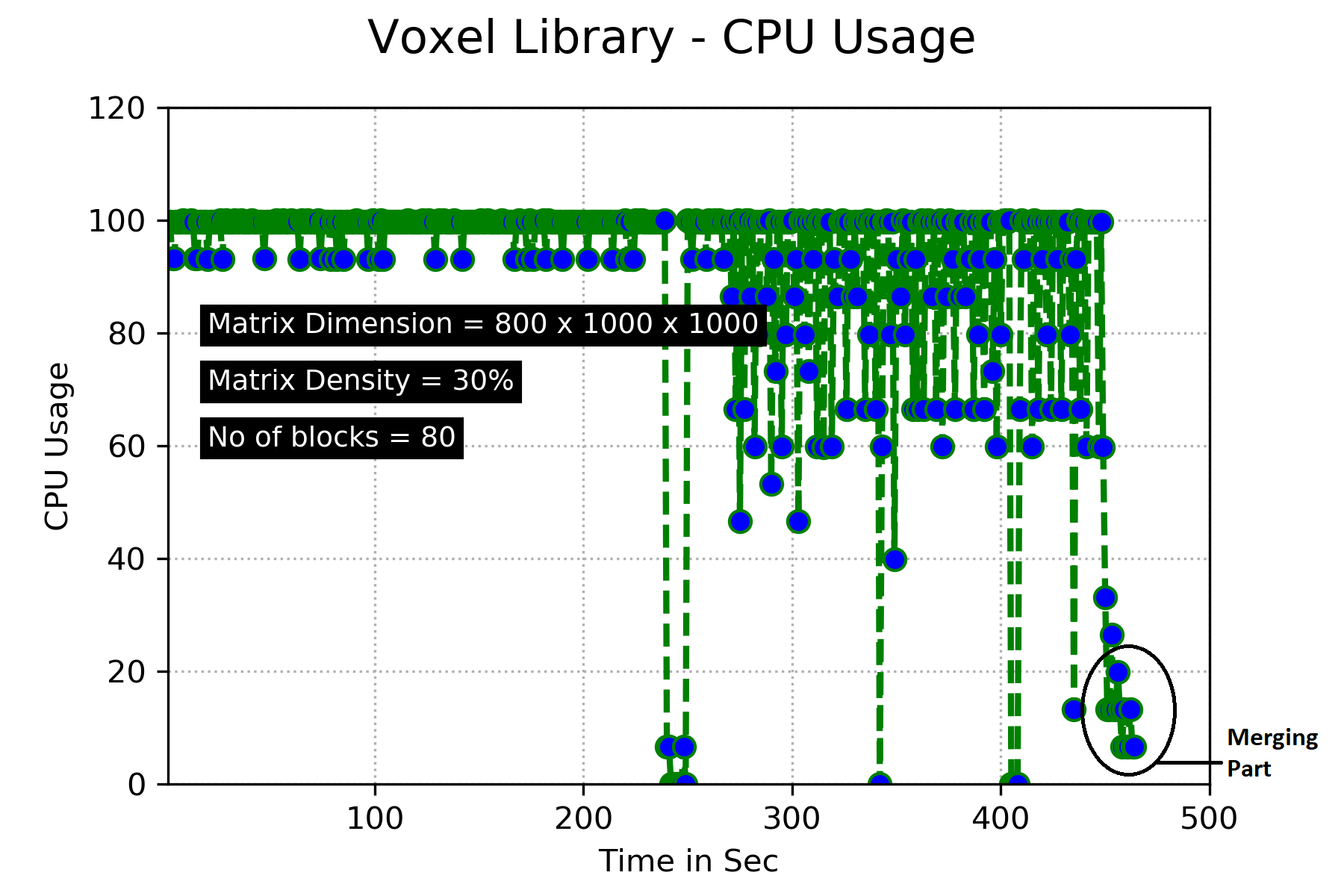


Fig. 5.1 Fig 5.2

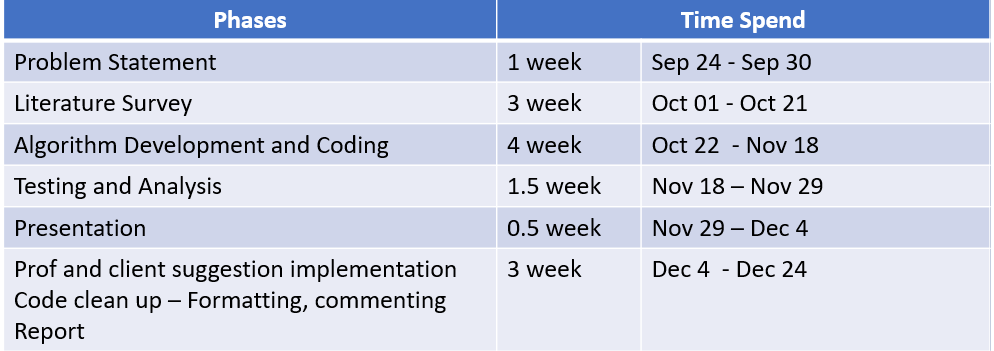
## Distributed Computing

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.

## Challenges and Lesson Learned

* To work in Team - Team member with different schedule.
* Python Modules numpy, scipy, os etc
* Morphological Operation with multidimensional array - 3D
* How to balance between memory usage and time.
* How to design API
* Google python style guide
* Different compression technique and current research related to this problem.
* How to report software development and put things in right words.
* Python is simple but not easy

## Time frame



# 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](https://www.quora.com/What-is-more-important-for-heavy-computation-RAM-or-CPU)
3. Data Processing In Computer, [peda.net](https://peda.net/kenya/ass/subjects2/computer-studies/form-3/data-processing)
4. How to slice and cut stl files for 3D Printing, [https://www.youtube.com](https://www.youtube.com/watch?v=afLolx2OEKE&feature=youtu.be)
5. Splitting 1 STL into 4 STLs, [https://www.youtube.com](https://www.youtube.com/watch?v=xw_ClxnJ1_U&feature=youtu.be)
6. Efficient Data Compression Methods Multidimensional Sparse Array, [https://ieeexplore.ieee.org/](https://ieeexplore.ieee.org/document/1252859)
7. Working with Big Data in Python and Numpy, [https://stackoverflow.com/questions/16149803](https://stackoverflow.com/questions/16149803/working-with-big-data-in-python-and-numpy-not-enough-ram-how-to-save-partial-r)
8. Image manipulation and processing using Numpy and Scipy, [scipy-lectures.org](http://www.scipy-lectures.org/advanced/image_processing/)
9. SciPy.org API, [https://docs.scipy.org/doc](https://docs.scipy.org/doc/scipy/reference/ndimage.html#morphology)
10. Split and Merge Algorithms, [https://hal.archives-ouvertes.fr](https://hal.archives-ouvertes.fr/hal-00211741/document)
11. Split a 3D Numpy Array into 3D Blocks, [split-a-3d-numpy-array-into-3d-blocks](https://stackoverflow.com/questions/39429900/split-a-3d-numpy-array-into-3d-blocks)
12. Numerical Computing with NumPy, [https://www.youtube.com](https://www.youtube.com/watch?v=V0D2mhVt7NE&feature=youtu.be)
13. Merging Two or Three 3D Array Block, [https://stackoverflow.com](https://stackoverflow.com/questions/38353269/how-to-merge-two-or-three-3d-arrays-in-python)
14. Combining 3 Arrays in One 3D Array in Numpy, [combining-3-arrays-in-one-3d-array-in-numpy](https://stackoverflow.com/questions/12395420/combining-3-arrays-in-one-3d-array-in-numpy)
15. Koji Nishio, Ken-ichi Kobori, Toshiro Kutsuwa, and Yoshikazu Nishikawa. Three Dimensional Morphological Filter Using Spatial. Dec. 12, 1999. Retrieved Sept. 24 2018. [https://onlinelibrary.wiley.com/doi/pdf](https://onlinelibrary.wiley.com/doi/pdf/10.1002/ecjb.1047)
16. Octree in Python, Retrieved Sept. 25, 2018, [http://www.anderswallin.net](http://www.anderswallin.net/2010/03/octree-in-python/)
17. Sparse Voxel Octrees, Retrieved Sept. 25, 2018, [sparse-voxel-octrees](https://github.com/tunabrain/sparse-voxel-octrees)
18. Check Current CPU and RAM Usage in Python, [how-to-get-current-cpu-and-ram-usage-in-python](https://stackoverflow.com/questions/276052/how-to-get-current-cpu-and-ram-usage-in-python)
19. Neighborhood and Block Operations, [http://radio.feld.cvut.cz](http://radio.feld.cvut.cz/matlab/toolbox/images/block.html)
20. Memory Management, <https://docs.python.org/3/c-api/memory.html>[g](http://www.roboticsproceedings.org/rss01/p48.pdf)
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](https://www.kdnuggets.com/2016/09/introducing-dask-parallel-programming.html)
24. Out of Core Data Frames, [out-of-core-dataframes-in-python/](https://jakevdp.github.io/blog/2015/08/14/out-of-core-dataframes-in-python/)
25. Openvdb, <http://www.openvdb.org/>
26. Scipy NDImage Morphology Binary, [scipy.ndimage.morphology.binary\_erosion.html](https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.ndimage.morphology.binary_erosion.html)
27. NDImage Morphology, [ndimage.html#morphology](https://docs.scipy.org/doc/scipy/reference/ndimage.html#morphology)
28. Image Measure Marching Cubes, [http://scikit-image.org](http://scikit-image.org/docs/dev/api/skimage.measure.html#skimage.measure.marching_cubes_lewiner)
29. Marching Cubes, <https://en.m.wikipedia.org/wiki/Marching_cubes>
30. Memory-Mapped, [http://www.roboticsproceedings.org](http://www.roboticsproceedings.org/rss01/p48.pdf)
31. OpenVDB - python, [http://www.openvdb.org](http://www.openvdb.org/documentation/doxygen/python.html)
32. Voxel Data Structure suggestions, [https://gamedev.stackexchange.com](https://gamedev.stackexchange.com/questions/17050/which-data-structure-should-be-used-to-represent-voxel-terrain)
33. Octree Storage VS Array Storage, [https://pdfs.semanticscholar.org](https://pdfs.semanticscholar.org/b2ec/9d0c9fadba4174e1ae5d29fccb7b344dc365.pdf)
34. Plot the 3d Image, <https://terbium.io/2017/12/matplotlib-3d/>
35. Numpy Matrix Memory Size, [numpy-matrix-memory-size-low-compared-to-numpy-array](https://stackoverflow.com/questions/52897457/numpy-matrix-memory-size-low-compared-to-numpy-array)