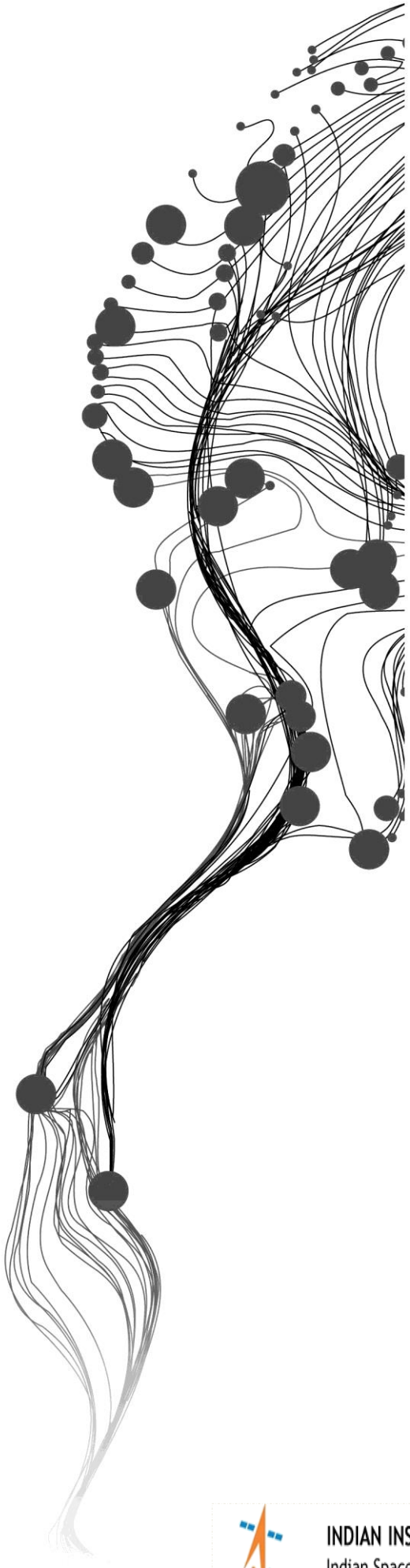
Object segmentation and surface creation from massive point cloud data using High Performance Computing

Vaibhav Raj
June, 2017

IIRS Supervisor: **Mr Ashutosh Kumar Jha**

ITC Supervisor: **Dr Ir. S. J. Oude Elberink**



Object segmentation and surface creation from massive point cloud data using High Performance Computing

VAIBHAV RAJ
Enschede, The Netherlands, June, 2017

Thesis submitted to the Faculty of Geo-Information Science and Earth Observation of the University of Twente in partial fulfilment of the requirements for the degree of Master of Science in Geo-information Science and Earth Observation.
Specialisation: Geoinformatics

SUPERVISORS:
IIRS Supervisor: Mr Ashutosh Kumar Jha
ITC Supervisor: Dr Ir. S. J. Oude Elberink

THESIS ASSESSMENT BOARD:
Chairperson (ITC): Prof. Dr Ir. M.G. Vosselman
External Examiner: Mr Arul Raj (NRSC, India)

OBSERVERS:
IIRS Observers: Dr Sameer Saran
ITC Observers: Dr V. A. Tolpekin

INDIAN INSTITUTE OF REMOTE SENSING
Indian Space Research Organisation

ITC
Faculty of Geo-Information Science and Earth Observation

DISCLAIMER

This document describes work undertaken as part of a programme of study at the Faculty of Geo-Information Science and Earth Observation of the University of Twente. All views and opinions expressed therein remain the sole responsibility of the author and do not necessarily represent those of the Faculty.

Dedicated to my Parents and my Sister.....

ABSTRACT

A decade ago processing remote sensing data was not as computationally demanding as it is now, even though 'technology' advanced with time, the 'velocity' and 'volume' of data increased at a much faster rate. In this literature, the scope of High Performance Computing solutions for point cloud processing of massive datasets was studied and compared with sequential processing of same, based on factors like speedup ratio, implementation effort, and lines of code. 'DEM generation from point clouds', was the considered 'use case' for this work. A point cloud processing tool was created, where many code optimisation measures were incorporated. The usability of this tool was expanded by adding customisable functionalities in it so that it's not limited to a specific data source or dataset.

Segmentation based filtering approach is considered, where multiple orientations of 'virtual profiles' are used. A new logic (function based weighted mean) is proposed. This is used to segment the ground object (or ground surface points) in each profile. After segmentation of points, filtration of points is performed. Where a 'point' is filtered out, only if it is a part of the surface object segment in majority of the orientations. This is followed by rasterization of filtered point clouds. The accuracy of proposed approach is also assessed, using the ISPRS filter dataset as a reference.

A detailed Inter-comparison report is created for serial and parallel implementations. The results reveal that Multi-core implementation is more viable in terms of implementation effort. But if 'speedup' alone is considered, then GPU implementation is better. And, no matter which high performance computing solution is used, optimisation steps and programming related choices play an important role, as the speedup in processing time not only depends on the 'hardware' but also on, factors like memory management, choice of programming language and related specifics like which version of compiler is used, container type, method type etc. Even a small tweak, matters when massive datasets are processed.

The proposed approach requires a minimum of two parameters to start with. The result of accuracy assessment shows that it works best for urban terrains (without sudden elevation changes) as compared with other terrains. The logic can also be used for post-processing of already filtered ground points, to remove any 'low vegetation' present in it. It's computationally much lighter than other popular algorithms, but it is terrain specific. The whole program is not hard coded for the proposed logic, hence it can be wrapped around any other logic to inherit all the useful aspects of parallelisation, customisability, memory management and code optimisations, based on end user's requirement.

Keywords: High Performance Computing, Filtering, GPU, Multi-core processing, Point clouds

ACKNOWLEDGEMENTS

To begin with, I would like to thank my parents. I had selected this topic of research based on my new found interest in high performance computing solutions for processing remote sensing big data, and there were moments (quite a lot, I would say) when things were not looking good, but the constant support from my parents and my sister helped me to stay positive and motivated.

I express my deep gratitude towards 'Dr. Sameer Saran' and 'Dr. V. A. Tolpekin' for providing all the necessary facilities and support a student can hope for. Next, I would like to thank both of my supervisors, I feel fortunate to have my IIRS supervisor as 'Mr. Ashutosh Kumar Jha' and my ITC Supervisor as 'Dr. Ir. S. J. Oude Elberink'.

A teacher plays an important role, in generating curiosity and interest in a particular subject for a student. And helps the student in streamlining, efforts into meaningful outcomes. Thanks 'Mr. Ashutosh Kumar Jha' for being that teacher for me. My ITC supervisor 'Dr. Ir. S. J. Oude Elberink' is one of the most polite and humble teacher I have ever met. I would like to thank him for his guidance and support.

"And so it turned out only a life similar to the life of those around us, merging with it,
Without a ripple, is genuine life, and that an unshared happiness is not happiness."
- From the novel 'Doctor Zhivago' by Boris Pasternak.

On that note, I want to thank all of my friends with whom I shared my joy and grief alike. Kunwar, Prathistha, Rocky, Sanath, Archika, Sayali, Neha and Vaibhav Pathak thanks for believing in me and bringing a smile, when I needed it the most. Kaaviya, Sansar, Raga, Vikas, Nixon, Yunmeng, Fredo, Dio and Jefferson thanks for helping me during my stay at ITC. Arijit, Soham, Daman, Hati, J Bhai, Meryl, Vipul, Pratyusha, Ramya and Ajit, thanks for making IIRS memorable.

Vaibhav Raj

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATION

LiDAR - Light Detection and Ranging

DEM - Digital Elevation Model

GPU - Graphical Processing Unit

SIMD - Single Instruction Multiple Data

MIMD - Multiple Instruction Multiple Data

MIC - Many Integrated Cores

RAID - Redundant Array of Independent Disks

SSD - Solid State Drive

PTX - Parallel Thread Execution

LOC - Lines Of Code

STL - Standard Template Library

1. INTRODUCTION

1.1. Background

In the field of remote sensing, spatial, spectral and measured details of data are increasing day by day, which in turn increases the volume of data. For such a high volume of remote sensing data, most of the analysis/operations performed, are becoming a challenging task. Making the 'data processing' a concern in terms of 'processing time', I/O overhead and data handling inside the memory. In particular, processing of point cloud data and Hyperspectral data, is slow and difficult to handle on commodity hardware (Plaza & Chang, 2007).

There are ways in which one can tackle this problem of slow data processing, like by reducing the I/O overhead, code optimisation, increasing the computing power of the hardware etc. The first solution one can think of is, making the processor more competent. This will result in speed up, but then for a processor with a single core, power consumption and heating becomes an issue if we keep on increasing the clock speed of the processor, which also has its limit. This is the reason why there is not much improvement in clock speed in the last decade (Ross, 2008). Therefore, multi-core architecture evolved and other options like offloading to GPU, co-processers etc also came into existence, thus cementing the place of parallelisation as a means to speed up data processing (Navarro, Hitschfeld-Kahler, & Mateu, 2014). Hence, the solution to decrease the processing time for remote sensing data can be parallel computing.

There are many options one can follow for parallelisation. Like which implementation to use viz., multi-core, GPU, CPU cluster, cloud computing, or a combination of these. Choosing between computational models like task parallelism and data parallelism or a combination of both. Selecting between shared or distributed or distributed shared memory approach, needs to be considered. So, while speeding up the processing of data using parallelisation, it is useful to know all the pros and cons involved, along with the specifics, like which kind of parallelism or which type of parallel implementation is well suited for a specific work. Along with this, the optimisation of I/O and data handling in memory can't be overlooked.

Research work in the field of remote sensing over the last decade concentrated on speeding up the processing of different types of remote sensing data. Few examples of parallelisation for different types of remote sensing data are, parallelisation of conversion of point clouds to raster image done by

Han, Heo, Sohn, & Yu, (2009), parallelisation of spectral unmixing for Hyperspectral datasets done by Delgado, Martin, Plaza, Jimenez, & Plaza, (2016). Work of Yang, Zhu, & Pu, (2008) on parallel implementation of image processing algorithm, and parallelisation of vector operations done by (Mineter, 2003).

In this literature, the usability of Parallelisation is examined by comparing it with the serial implementation based on some parameters (like speedup, Lines of code etc), and also the data handling in the 'memory' is optimised. The use case considered is, segmentation based filtering of massive point clouds for DEM generation using multiple orientations of virtual profiles, inspired by the work of Sithole & Vosselman, (2005). A new logic is proposed, which is used to segment surface object within a profile. In a scene, there are many objects like buildings, trees, cars, ground etc. But for DEM generation, only ground surface is needed, and all these non-ground object types are not significant. So, here we only focus on this single, surface object (or ground object). After segmentation, points are filtered out, based on whether a 'point' is a part of the 'surface' object, in majority of the orientations. These filtered points are also rasterized. The functionality of the algorithm was extended, to process any point cloud dataset created using photogrammetry methods, airborne LiDAR, mobile or terrestrial LiDAR. Lastly, the accuracy assessment of proposed algorithm is performed using ISPRS filter test datasets.

1.2. Motivation and problem statement

In order to justify the need for any research, 'top to bottom' view about the problem which motivates that research should be well understood and the 'research output' should be towards, solving the problems in society, industry or the scientific community.

If the source of point clouds is LiDAR or a photogrammetric technique like SFM (Structure From Motion) the detection or extraction of surfaces from point clouds are one of the most common tasks. And these tasks become more problematic due to the ever increasing size of point cloud datasets, thus making the processing of these datasets expensive for commodity hardware. The increasing volume of point cloud datasets in recent years is due to many factors.

Like,

- Improved sensor related technologies like Geiger-mode LiDAR sensor (Duffy, 2015).
- Increasing 'extent' of the coverage area.

- Requirement for more detailed surface (higher resolution) for heritage preservation, thus increasing the point density.
- New photogrammetry methods for DEM generation (Uysal, Toprak, & Polat, 2015).
- Requirement of high-quality DEM, produced with higher point density datasets (Chu et al., 2014).
- Need for fast processing (low latency) of data for some of the near real-time applications viz. military and emergency response (Sene, 2008).

These application driven factors are increasing the velocity (analysis of streaming data) and volume (scale of data) of point cloud datasets.

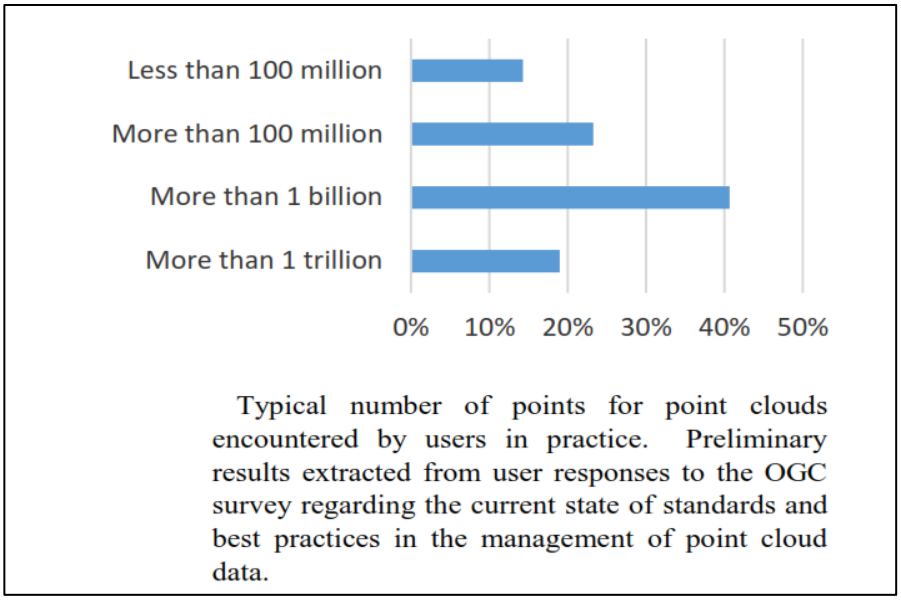


Figure: 1: Results from OGC survey (in 2016) | Ref: (Boehm, Liu, & Alis, 2016)

Dutch AHN2 dataset has around 400 billion points (Van der Sande, Soudarissanane, & Khoshelham, 2010). The size of the point cloud dataset released by UK's environmental agency was in the order of terabytes (Whitworth, 2015). Boehm, Liu, & Alis, (2016) showed the early results of the latest OGC survey (shown in, Figure: 1) about the usage related to point clouds among users.

Therefore sooner or later, the problem of surface creation from huge dataset needs to be dealt with. Not only this, but most of the approaches used for DEM generation are iterative and computationally costly (Q. Chen, 2007). While conventional algorithm when sequentially executed needs to load all the data at once, which creates a problem as the size of the data itself, exceeds the size of computer memory when dealing with huge datasets (Isenburg, Liu, Shewchuk, & Snoeyink, 2006). These drawbacks about existing algorithms, adds up to the issue.

Hence, there is a need for improving the new or existing algorithms, in terms of processing speed and data handling to speedup the whole process. At the same time, maintaining a balance in trade-offs (as shown in, Figure: 2) between 'accuracy achieved', 'resources spend', and 'processing time'.

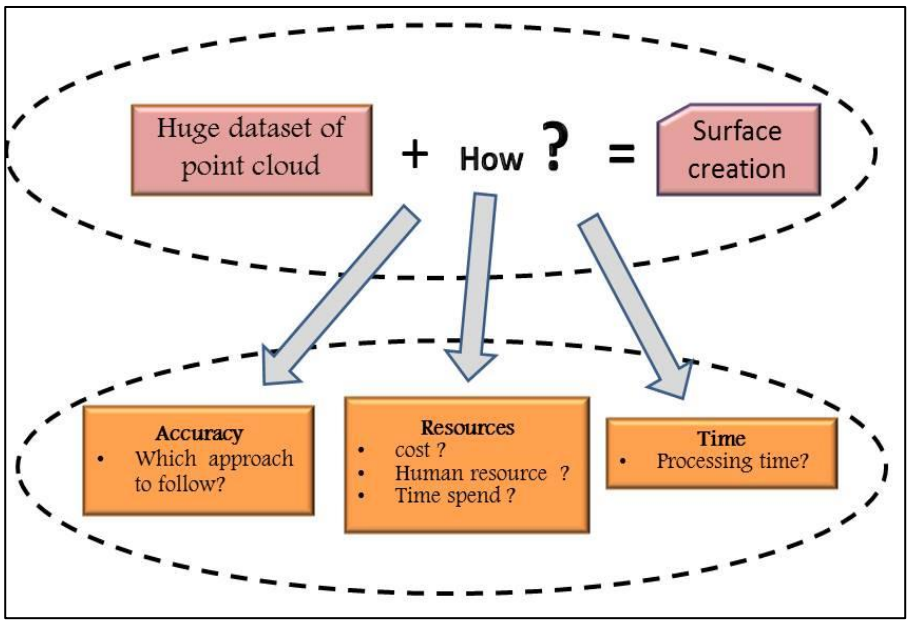


Figure: 2: Balance in Trade-offs between 'Accuracy', 'Resources' and 'Time'.

By only improving one aspect, we should not compromise the other but rather come up with a solution to balance them. For example, if there is a way to get exceptionally high accuracy for surface extraction but a lot of hardware resources and human resources are required then it is not worth the cost and power consumption if there exists a more balanced way, at a comparable 'accuracy'. Keeping these arguments and discussed problems in mind, and the following objectives were formulated.

1.3. Research objectives

1) Propose an algorithm for segmentation based filtering of massive point clouds for generating DEM (surface), where multiple orientations of virtual profiles are used, which is partially based on the previous work of Sithole & Vosselman, (2005).

    - Optimise the memory handling and processing speed of the algorithm.

    - Making the algorithm (tool) customisable, so that it is no longer hard-coded for a specific type of dataset or data source of point clouds and to give more control in terms of governing parameters or in-built options.

2) Parallel implementation of the algorithm using high performance computing solutions.

    - Inter-comparing the implementations (serial, multicore & GPU), based on parameters like speedup ratio, implementation effort, lines of code (LOC).

1.4. Research questions

1) What is the performance of the proposed approach of segmentation based filtering, where multiple orientations of virtual profiles are considered, when used on different terrain types?

2) How can we optimise the processing speed and memory handling in the involved steps of DEM generation process?

3) How to increase the usability of the algorithm and make the extraction of 'surface' more generic, so it can also be used for other use cases too, apart from ground point filtering?

4) Which is the better high performance computing solution for the algorithm's parallel implementation?

1.5. Innovation aimed at

For this study the innovation is aimed at:

- The logic of segmentation based filtering, which is used in virtual profiles of multiple orientations.

- Finding an efficient way for memory handling and code optimisation for overall speedup.

1.6. Research approach

In this research, a step by step approach was followed to fulfil the objectives, where we first develop the logic, which is used in virtual profiles of multiple orientations. Later, the code was written while incorporating all the optimisation measures.

Then, algorithm's functionality was extended and it was made customisable, like changing the no. of orientations of virtual profiles, giving weight to the result of a particular orientation, user-defined angle of orientation, scaling of points, translation of points, noise removal etc. Memory profiling and time profiling of the program was done for detecting memory leaks and identifying the bottlenecks.

Based on time profiling, parallelisation target and approach was decided. Then, a comparative report was formed for the serial and parallel implementations. A program was also written to evaluate the accuracy (point to point) of the proposed algorithm, where ISPRS reference datasets were used.

1.7. Thesis structure

This thesis is divided into five chapters, starting with chapter one, where the background and motivation for the work are discussed, then research objective and research questions, lastly innovation, research approach and thesis structure. Chapter two contains the literature review and chapter three has detailed methodology along with details about study area and materials used. Chapter four contains results and discussion. Chapter five talks about conclusion and recommendations. The appendix contains some extra information about this work, like conceptual diagrams, code snippets etc.

2. LITERATURE REVIEW

2.1. Introduction

In this section, the work related to DEM generation are discussed. Later sections discuss about, filtering techniques for point cloud processing and high performance computing implementation in the field of point cloud processing.

The application of 'how the earth surface is' and 'how it changes with time and space' is growing day by day. Digital surface or DEM are directly or indirectly used in many kinds of scientific and engineering applications like hydrology, flood modelling, 3-D city modelling, designing structures, or even studying an archaeological site. And in order to create a digital surface, we need reliable ground point measurements (point clouds). This can be achieved by using technologies like light detection and ranging (LiDAR), Digital photogrammetry, microwave remote sensing or with the help of traditional ground surveying techniques (Li, Zhu, & Gold, 2004).

For DEM generation, the choice of algorithm for filtering out the terrain points largely depends on the study area rather than using the popular algorithm, and in the work done by Jian et al., (2015) for the same reason linear prediction method was preferred and Hadoop™ software framework was used to generate the DEM. Another work for fast processing and DEM generation was done by Han et al., (2009) where PC clusters were used and key aspects like linearity and efficiency of parallel processing were studied.

Terrain specific work of Qi Chen, Wang, Zhang, Sun, & Liu, (2016) where point clouds from satellite image matching were used and it was found that the accuracy of their proposed algorithm was better than the classic progressive tin densification method, for both urban and mountainous terrains. This is another example, showing that the type of algorithm one should select largely depends on the use case.

The algorithm proposed by Qi Chen et al., (2016) for DEM generation had the lowest average type I error in all test cases as compared to the classic algorithms compared in ISPRS test. It was a combination of multi-level kriging interpolation in morphology-based filtering approach.

2.2. Filtering Techniques

A lot of research has been done on filtering of ground points from point clouds in past 20 years. Broadly these filtering algorithms can be divided into 4 types: slope based, morphology based, surface based, and segmentation based. First slope based filter was proposed by Vosselman (2000) and later on, many modified versions of this approach were created. This approach is conceptually easy and it is not computationally expensive. But still, it's not suitable for all kind of terrains.

The unique aspect about Morphological filters is that they are raster image based filters. While initially it was proposed by Zhang et al., (2003), later on, a simple yet improved form of this concept was developed by Pingel, Clarke, & McBride, (2013), among many other modified workflows. But same as slope based filters these were also not suitable for all kinds of terrains, especially for steep mountain areas. For surface based filters, progressive TIN densification (PTD) is the most widely used filter and is also a part of commercial software *TerraScan*™. Still, progressive TIN densification is quite consistent when considering all kinds of terrains (Sithole & Vosselman, 2004).

Segmentation based filters produce a good result in some cases but they also fail when it comes to complex terrains. These filtering algorithms usually have two stages, first being segmentation of points, which is followed by filtering based on the segments (Lin & Zhang, 2014). Sithole & Vosselman, (2005) proposed an iterative segmentation based filtering approach based on minimum spanning tree (MST) within a scanline, and profile intersection was used. In this work, the concept of multiple profiles is taken from their work. A hybrid segment based filtering approach, which is a combination of two methods gives a better result (Lin & Zhang, 2014). But it all depends on the result of segmentation, which is a drawback for these type of filters. Melzer, (2007) segmented the point clouds using clustering approach directly on the dataset, where mean shift technique was used.

Segmentation means the labelling of points which are a part of same surface or region, based on some sort of criteria, where each segment has a unique identifier, and it may be possible that some points do not belong to any segment (Rabbani, van den Heuvel, & Vosselman, 2006). In this study, only a single 'object type' was considered. Which is the ground object (ground surface points), segmented in a virtual profile, rest of the non-ground objects were discarded.

Then there are some new innovations too, like the use of a virtual cloth on inverted terrain to separate ground and non-ground points, where the number of parameters are not many and are also simple to set, also the developed software is released as open source (W. Zhang et al., 2016). But one fact can be clearly understood that when it comes to filtering algorithms, none of them can be considered the

best, all of them have their advantages and disadvantages depending upon the application and type of terrain.

In the case of datasets from the Terrestrial laser scanner, new filtering approach has been developed as the algorithms meant for processing airborne point cloud does not work in the same way. One of them is the work done by Rodríguez-Caballero, Afana, Chamizo, Solé-Benet, & Canton, (2016), where spectral information is also considered while using morphology filter based approach and final RMSE error was found to be reduced by 40 percent.

Type I, type II and total error is preferred for judging the performance of algorithms, where type I error is the number of wrongly classified terrain points per total number of ground points in the dataset and type II error means the number of wrongly classified non-ground points per total number of non-ground points in the dataset. And the total error is the number of wrongly classified points per all points in the dataset (Sithole & Vosselman, 2004).

2.3. High performance computing

While, when it comes to optimising the speed of these algorithms a lot of work has been done and is being done. There are many successful implementations of parallelising different filtering algorithm, but only a few are needed to mention, in order to justify the usability of HPC solutions. Very good speed up results were achieved when GPU was used to speed up scan-line segmentation based filtering, it worked well for urban terrain but failed for mountainous areas, also the procedure can evolve to real-time processing (Xiangyun Hu, Xiaokai Li, & Yongjun Zhang, 2013). GPUs works really well with data decomposition type of parallelisation problems.

Some literature can also be found for multi-core implementations of filtering algorithm. Krishnan et al., (2010) applied a big data approach on huge point cloud datasets and applied Hadoop implementation for optimisation of processing speed he also compared this technology in terms of cost-performance and programming effort, similar Hadoop approach was successfully attempted by Jian et al., (2015). While Hadoop is an open source implementation of MapReduce programming model, it is currently maintained by Apache™.

Streaming data approach, which is an improvement over the drawbacks of MapReduce model, was used by (Kang, Liu, & Lin, 2014). It was there attempt on improving the processing speed of PTD algorithm by one by one feeding parts of a scene and processing it. This does not intensify the usual I/O overhead which occurs when the whole dataset is read at once for processing and also maintains an

optimum memory usage, which was one of the shortcomings of their previous work on hybrid filtering approach (Lin & Zhang, 2014).

It is also worth mentioning the use of PC cluster by (Han, Heo, Sohn, & Yu, 2009) for accelerating the rasterization of point clouds, where minimum local filtering followed by inverse distance weighting was used by them. A hybrid framework of HPC implementation in which multiple GPUs were used using MPI and a balanced approach in terms of I/O, computations, intercommunication time was proposed by Danner, Breslow, Baskin, & Wilikofsky, (2012).

Lastly, the latest work by Boehm, Liu, & Alis, (2016) based on Apache™ Spark framework for point cloud data classification, where they used 'random forest', and also check the scalability of their proposed method. where Apache™ Spark is an open source cluster computing framework based on resilient distributed dataset (RDD), it is considered an improvement over Hadoop for some applications.

3. STUDY AREA, MATERIAL USED AND METHODOLOGY

3.1. Study area

This work is not a study area specific work. But Netherland's AHN3 dataset was used for airborne LiDAR datasets. It is available in tiles (an example is shown in, Figure 3), and the download is possible in .laz format. Data is freely available with a high point density (over 10 points/m^2), where RIEGL LMS-Q680i was used having accuracy up to 20 mm. As they have a huge number of points, therefore it is well suited for this use case. For terrestrial LiDAR datasets, RIEGL VZ-400 was used, which has an accuracy of 5 mm. Another dataset created via structure from motion is also taken. After manual cleaning of data, only coordinates of a point are preserved rest of the information about a point like colour, scan angle etc are discarded.

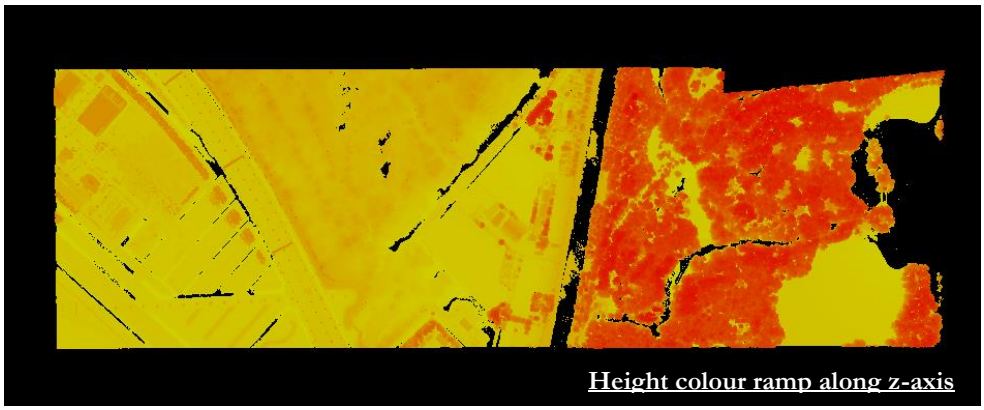


Figure 3: An example of a AHN3 dataset's tile
('C_25DN2' tile with 20,594,973 points)

3.2. Material used

- System specifications:
  - Processor: Intel® Core™ i3-4000M Processor
  - No. of physical cores: 2 || No. of logical cores: 4
  - Base frequency of processor: 2.40 GHz
  - RAM: 12 GB DDR3 RAM.
  - Storage : 500 GB SATA HDD @ 5400 RPM.

- GPU specifications:
  - NVIDIA GEFORCE 820M (details are shown in, Figure 4 & Figure 5)
  - No. of Cuda cores: 96
  - Dedicated video memory: 2 GB DDR3.

Figure 4: GPU specification - device information

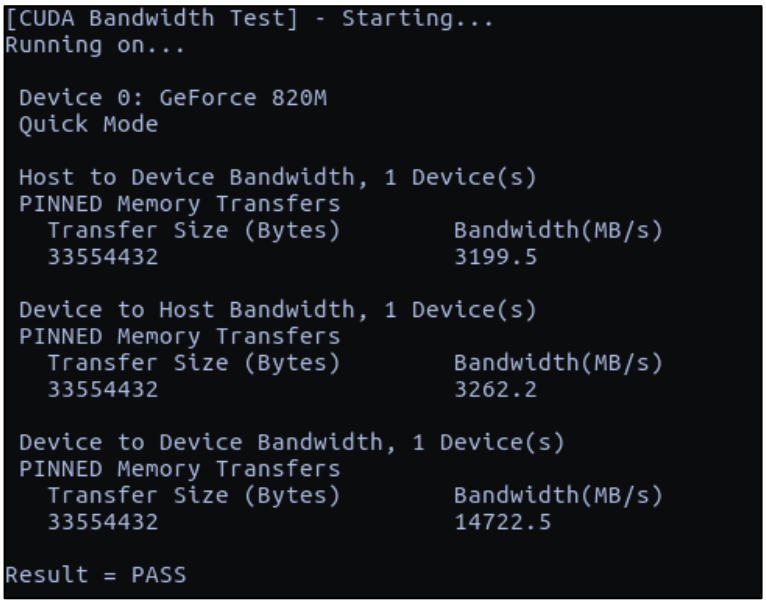


Figure 5: GPU specification - device Bandwidth

- C++ was selected for programming and C++ standard library was used.
- OpenMP 4.5 and OpenACC 2.5 directives are used for multi-core and GPU implementations.

3.3. Methodology

This chapter contains the detailed description of various steps involved in the research work. In the following paragraphs, all those steps are covered briefly and then, later on, they are explained in depth section wise. A simplified flowchart for the algorithm is also shown.

The workflow of this research starts with developing a logic or criteria which can work with multiple orientations of virtual profiles and can segment the surface object in a profile, keeping the first objective in mind.

The code was written using C++11 language where only in-built standard C++ library was used, rather than any 3rd party library, custom made for point cloud processing like PDAL, PCL etc. The reason for this was, to have more control over background processes like memory allocations, type of data container, choice of method/function/data types etc. As applying tweaks and optimising a stand-alone program can be done in a more efficient and faster way rather than understanding a whole library and optimising it. Where a lot of functions are inter-depended and tweaking takes a lot of time, sometimes a single tweak requires, re-writing of a large part of the library. This can be due to the way the library was written or due to the evolving functionalities in languages with each version (like between C++98 and C++11).

After incorporating different measures (discussed later, in Section 3.3.2) needed at programming level for the optimisations. The functionality and usability of the developed algorithm are extended beyond the selected use case of DEM generation from airborne datasets. So it can be used to extract the top or bottom surface (DEM, in the case of airborne datasets) from any data resulting from TLS, airborne LiDAR or SFM.

When the coding was completed, Memory profiling of code was done for checking memory leaks. Followed by time profiling for understanding, which step/portion of code consumes how much time. This information helps in deciding what, the parallel implementation will be for, as the portions of code which are significantly slow should be the prime target for parallel implementation. It also gives a better understanding of time spent per line of code.

Some test cases are taken to understand the working of the algorithm and the results are discussed. Working towards the second objective, parallel implementation of the filtering step was done for multi-core and GPU using OpenMP 4.5 API and OpenACC 2.5 API. Then serial and parallel implementations are compared based on quantitative parameters. The accuracy of the outputs is also checked, for the proposed algorithm using the ISPRS filter test datasets.

3.3.1. Developing the filtering logic

The approach of multiple orientations of virtual profiles is inspired by the concept of multiple scanline approach used in the work of Sithole & Vosselman, (2005). Multiple orientations of virtual profiles is illustrated in, Figure 8. Within these profiles, surface object is segmented based on a 'criteria' (or logic). After segmentation, a 'point' is filtered out as a ground point if it is a part of the 'surface object' segment in majority of the orientations.

The logic which was used within a scanline was minimum spanning tree (MST), in the work of Sithole & Vosselman, (2005). Where in an iterative process by varying the parameters like threshold weight and user defined parameter 'k', objects are detected and separated to finally end up with bare earth.

We propose a new logic, where function based weighted mean (elevation based) was used to segment the surface object within a virtual profile. This approach (single iteration) is not iterative in nature, unlike MST. While MST is also more computationally expensive and has a much higher 'time complexity' than our proposed approach. Slope is not considered in the proposed logic, rather it is based on elevation information. Its effect on the accuracy can only be understood after testing it, with different terrains. Proposed approach can only separate one 'object type', unlike MST, which is the 'surface object'. Moreover, only the 'surface points' matter, in order to fulfil the first objective of 'DEM/surface generation' rather than the non-surface objects, which are also present in a scene.

Some other logic can also be applied to the multiple orientations of virtual profiles, apart from MST and the one proposed. Like, 'slope' and elevation of points within a window with respect to the local lowest point where the size of the window, elevation and slope thresholds are manually set, as proposed by Xiangyun Hu et al., (2013). Han, Lee, & Yu, (2007) applied a logic based on proximity and elevation thresholds. Though it is worth mentioning that, in these two works, logic/criteria is applied only in the native scan lines and not in virtual profiles in multiple orientations.

Function based weighted mean (based on elevation) is proposed to segment the surface object within a virtual profile. The lowest point is searched (illustrated in, Figure 6), and a weighted mean is calculated, where the weight of a point is a function defined as shown in, eq. - 3.3.1.1. Function based weighted mean of 'elevation value' of all the points within a profile is defined as shown in, eq. – 3.3.1.2.

$$W_{pi} = 1/e^{(pi - pm)/2} \qquad \text{eq. - 3.3.1.1}$$

$$M = \sum W_{pi} * p_i / \sum W_{pi} \qquad \text{eq. - 3.3.1.2}$$

Here, p_i is a point's elevation within a virtual profile, W_{pi} is the weight of that point, e is Euler's constant, p_m is the lowest point in that virtual profile and M is the weighted mean for that profile.

Points in a profile, who's elevation is below the calculated weighted mean (based on e^x function, graph shown in, Figure 7) are included in the surface object segment. A 'point' is filtered out as a ground point (surface point) based on, whether that point is a part of a surface object segment, in majority of the considered orientations. For example, if a 'point' was a part of the surface object in an orientation then for each 'true case', a value of '1' is associated with it or added if previously a value was associated. So if the total no. of orientation is '3' then if this value is more than equal to '2', then only that point is filtered out as a ground point.

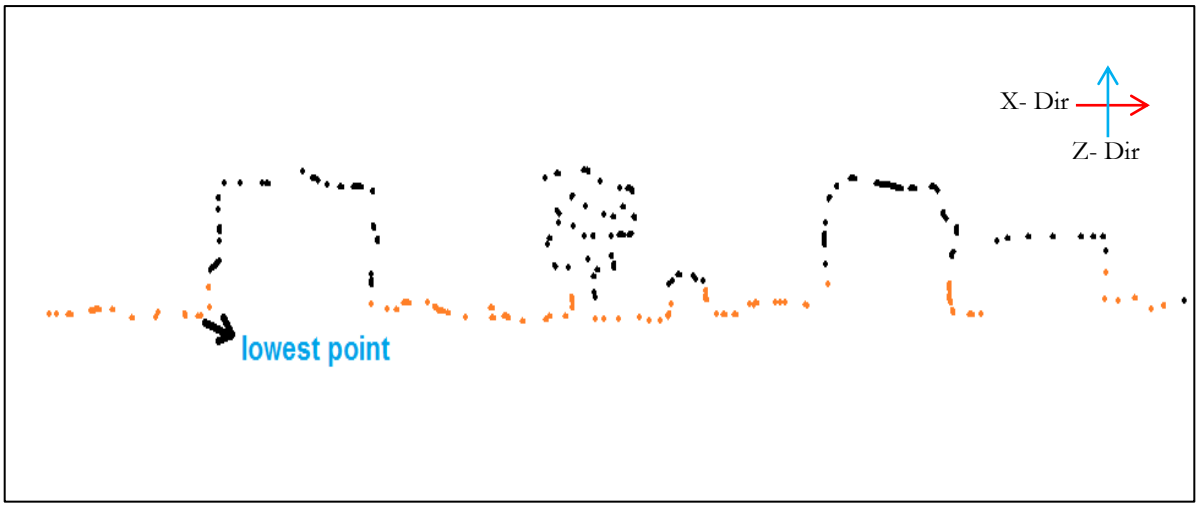


Figure 6: Cross-section of a profile, where the lowest point is searched for calculating, the function based weighted mean of elevation.

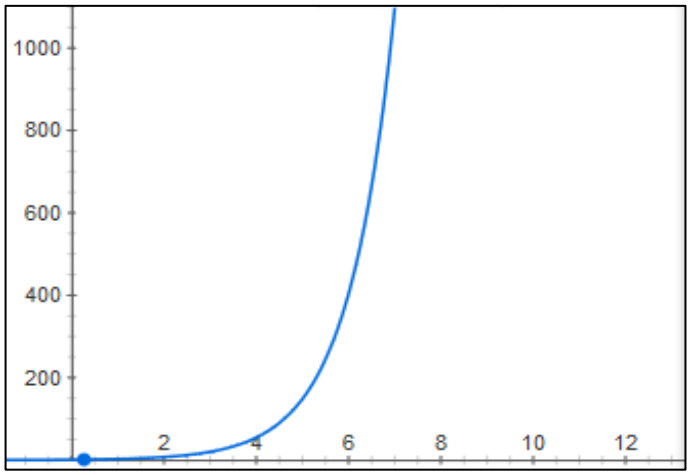


Figure 7: Large-scale graph of e^x function

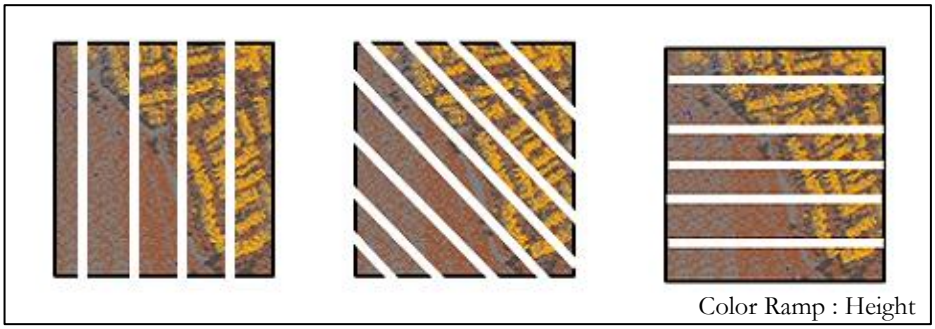


Figure 8: Illustration of multiple orientations of virtual profiles for a scene.

3.3.2. Programming, optimisation and customisation

For programming C++ (C++11 ISO standard) was used, as it gives more control over memory allocations and is widely supported by various APIs for parallel implementations. Only C++ standard library was used, instead of any third party library like PCL, PDAL, LASlib etc to minimises the no. of dependencies, and to increase the transparency and flexibility for implementation of various optimisation measures.

GCC ver. 6.2 (GNU compiler collection) was used for compiling the program, a total of '13579' lines of code were written, which includes the portion of code responsible for the two parallel implementations. The program was written in a robust way, capable of handling any runtime exceptions and user input errors. For parallel implementation, OpenMP and OpenACC APIs were used. And other ways of implementation were also explored to comment on the implementation effort in terms of time spend, hardware, software etc. The code is provided as open source (link provided in, APPENDIX D).

Following are the optimisation measures incorporated in the program for increasing the robustness, processing speedup, and memory handling:

- Only one-dimensional container is used to store data, to have minimum allocation and de-allocation overhead, and locality of data in the cache (illustrated in APPENDIX B).

- The program automatically determines the maximum and safest size of a container it can create, according to the system it is running on (code snippet is shown in, APPENDIX C). And based on that decides the size of containers it will be creating, later on. And prompts the user to decrease the extent of a 'tile' if that many points can not be processed at the same time. The situation of 'out of memory' never occurs, which can happen when processing massive point cloud datasets (calling this optimisation as the 'safest container size criteria' for future reference).

- Sufficient memory is pre-reserved per container, before actually copying anything in it, thus filling of the container is faster (push back method is only used after initialising a container to the required size). So, no resizing occur which happens when pre-initialised container's capacity is not enough and more memory is required, creating additional allocation and deallocation overhead with each 'resize'. This optimisation makes the program stable in terms of memory usage and improves the 'total time' taken to process a dataset.

- Variables are dynamically allocated. All the manual inputs by the user are validated, and exceptions handling is also taken care of. The program is structured in such a way to, have minimum data transfer overhead in parallel implementations (explained in later Sections) and avoid thrashing (fast exchange of data between memory and disk).

- There is an option to define tiles, doing so will result in streaming of data (tiles) and processing it, this results in reducing memory usage and reduces the otherwise I/O bottleneck at the end of processing.

- Memory is properly allocated and de-allocated to avoid memory leaks, which is later verified by memory profiling of code. This is also taken care of, within each loop, in parallel implementations.

The program was then customised in many ways and some functionalities were added, to increase the usability of this tool. The program is written in such a way that it is not hard coded only for the logic proposed in this work, but any user can wrap this program around any logic and inherit all the memory

optimisations and parallel implementations if needed. This can be done, just by editing few lines of code, where the logic is defined and the rest of the code remains unchanged. Even if the user's logic is not compatible with multiple virtual profile based approach, still the memory optimisations are inherited and other features can be used. A whole tile can be executed at once without the profiling approach (by inputting the size of profile width, equal to the tile size).

'Weights' can be given to prioritise a particular orientation of virtual profiles. So, when filtering the ground points if a 'point' is a part of the surface object segment in a 'prioritised' orientation then even if this is not true for other orientations the point will still be filtered out. Noise removal feature is also added based on standard deviation where upper and lower sigma is taken as input. Data can be processed irrespective of its coordinate system, and the filtering plane can also be changed. 'Scaling' and 'translation' operations are also added as an additional option. If needed only one orientation with a user-defined angle of orientation can be chosen instead of three. For the 'output type', user can opt for classified point clouds, into surface and non-surface class. Or can get filtered points in a separate file. The threshold can also be soft adjusted if needed, or the logic can be modified to segment the surface from the top, using the 'logic inverse' option which can be useful in some specific cases. The 'precision' of the output coordinates can also be changed.

3.3.3. Code profiling

Memory profiling was performed, in order to check for memory leaks, and understand the memory usage. And this was followed by time profiling to find out the contribution (in respect of time) of various portions of code and overall workflow.

For memory profiling, many open source code and tools are available like Valgrind, memcheck, Mtuner etc or the inbuilt feature of Visual Studio community edition (used here). After running the profiler it was found that there were no memory leaks present in the program. All the allocation and deallocation operations in the code were accounted for. Figure 9, illustrates the memory usage during the runtime when a sample data was processed on a clean system.

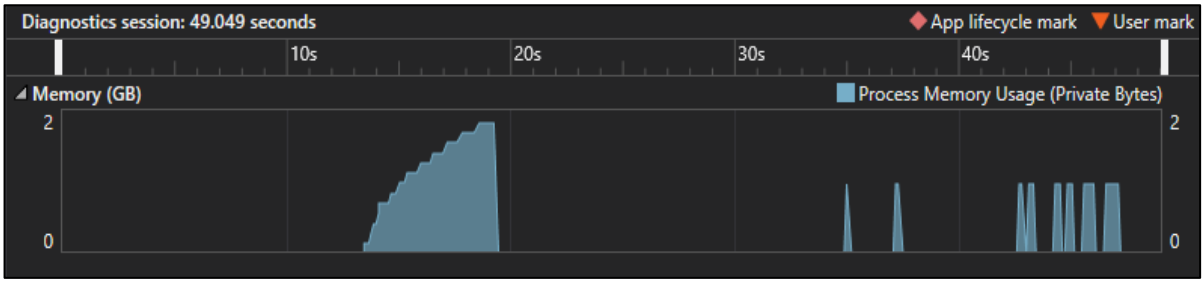


Figure 9: Memory profiling - understanding the memory usage

By observing this memory usage graph of the serial run, we can deduce and comment on many aspects of the program during runtime. Which are listed below:

- The first rising peak in the graph is because of, one of the optimisations considered in the program ('safest container size criteria'). Where it finds the practically possible and safest size of the container that can be created on that system. Apart from the system configuration, the outcome depends on factors like the type of OS (governs the default heap size) running on a system, current memory usage of background processes, implementation specifics of the library (here it's STL) like the theoretical limit of maximum no. of elements in a container.

- Also, it is worth mentioning that this particular optimisation step runs in the background while the user inputs the runtime specifics like file path, implementation type, profile width etc. And the outcome is displayed during the input phase itself before deciding the tile size. So in this example, the user was providing the inputs, up till 34 seconds (approx.).

- The consecutive peaks after the optimisation step are the processing of individual tiles. Within each peak, multiple virtual profiles of different orientations are processed.

- As all the containers are pre-initialised to the required size, the peak (memory usage) is same for all the tiles, instead of changing peaks due to different no. of points in each tile. That is, optimum memory usage is maintained without any sudden variation and avoiding any resizing of containers (explained previously).

- The irregular gaps between peaks are the 'time taken to write the processed points'. These are due to the fact that each tile will contain a different number of points. Hence, after processing a tile, the points are written to the file, so if the no. of points are more in that tile as compared to others, then the writing time will also be higher than others. In some cases, this can also be caused when system resources are busy.

Next step is time profiling, which is crucial in deciding the target and framework of parallel implementation. For this step, the overall workflow was considered. Which is, the whole DEM generation process. There are two main steps of filtering and rasterization in it. After writing the code responsible for filtering step, few lines of code were added for rasterization of filtered point cloud where GDAL was used.

And it was observed that the portion of code responsible for 'filtering step' was a bigger bottleneck as compared to 'rasterization step'. This was true, in terms of 'processing time' for the whole workflow, and in terms of ' data size', because after filtering the no. of points gets reduced significantly in the case of DEM generation scenarios. Complex computations are involved in the filtering step, rather than the rasterization, which is generally true in all cases. Moreover, there are plenty of 'open source' means for the conversion of point clouds to raster, while some of them are accelerated as well. Based on these observations, in this study, the filtering step is targetted for parallel implementation. For rasterization, one can use GDAL library or any of the pre-accelerated open source tools like CloudCompare, LAStool etc.

Another 'time profiling' is done (in detail), but only for 'filtering step', rather than whole workflow. The major events that occur at runtime which can be significant in terms of total processing time are, reading of data, processing of data, writing of data, memory allocation and deallocation overheads.

It is worth noting that, some of the optimisations were aimed at reducing allocation and deallocation overhead. Also, It was found that time consumed for memory allocation and deallocation is negligible as compared to other portions. That leaves 'reading', 'processing' and 'writing' as the major parts of 'total time' consumed (shown in, Figure 10).

Figure 10: Time Profiling - sections of code

For time profiling actual time was considered rather than CPU time, to get precise results. The time taken by these sections for a sample run are shown below in, Figure 11. Here, the 'time required to write' is much larger than expected, as it depends on the precision and no. of significant digits (decided by the user) for the coordinates of point clouds, this affects the 'time to write the processed points'.

```
time_to_read...   284.9099
time_to_write...   1715.0588
time_to_process...  1152.1033
```

Figure 11: Time profiling - sample run (time in seconds)

As we can see here, to reduce the 'total time' we can either reduce the I/O overhead (read & write time) or the processing time or both. I/O overhead (read and write time) mostly depends on the hardware, to a large extent. Key factors are HDD spindle speed (eg - 5400 rpm, 7200 rpm, etc) or HDD configuration (eg -RAID 0 is faster than RAID 1 config.) or changing the hardware technology itself like Solid State Drive (SSD). Or the solution is with Data Intensive Computing (like Hadoop™, Spark™).

Based on the above arguments we only aim to accelerate the 'processing' section, thus reducing the overall time. For this, high performance computing solution is used (these are process intensive rather than data intensive), where multicore processing and GPU processing is utilised for parallelisation of filtering step.

3.3.4. Working of algorithm

The multiple virtual profile approach is illustrated in, Figure 12. And in Figure 13 and Figure 14, the algorithm is explained in a simplified way. As shown in the flowchart the algorithm takes the input in ASCII file format, in that file x, y, z coordinates of a point are written on each line separated by space. Further, the user can choose between the types of parallel implementation.

The main plane can be selected, that is across which segmentation and filtration will be performed. This step enables the program to process even if the surface which has to extracted does not lies in the conventional XY plane (in the case of airborne datasets). After that, the region of processing can be selected in three ways. First, by inputting the minimum and maximum coordinates of the plane of segmentation. Second, by inputting the four corner points of whole region or a subset of the whole region. Third, the algorithm automatically searches for corner points by reading the file.

Next, the tile size and profile width are asked for. If the user wants to process the whole scene at once, tile size greater than equal to the size of scene can be selected, also if the system is incapable of processing the whole scene at once, then the user will be prompted and can input a smaller tile size which can be processed as per the system's configuration. When processing the scene using tiles, the recommended minimum size of tile should be more than the maximum size of the non-ground object in the scene, like buildings, cars etc (explained later). Next, when asked for profile width if the user does not want multiple profiles but wants to apply the logic in the whole scene. Then this can be done by providing the profile width equal to tile size provided in the previous input. The program is written, keeping these specific cases in mind thus, making the workflow flexible in nature.

Then a ground threshold is asked, this parameter is for those virtual profiles which comprise of only ground points, in that case, all the points in that profile are considered as a part of the surface object. Threshold defines the difference between the highest and lowest point in a particular profile, if the actual difference is below this value, all the points in that profile are included in the surface object.

Next, the 'advance options' can be selected if needed. which includes, choosing the no. of orientations (one, three or five), like if a particular angle of profiles can give a better result then only one orientation can be selected. Or weight can be given to a particular orientation to emphasise the dominance of that orientation's result. A standard deviation based noise removal can be done if needed. The logic searches for the 'lowest point' for calculating weighted mean, this can be changed to 'highest point', this will extract the top most surface if there is such a case where the scene is inverted or, natively a surface is present at the top of a scene. Standard operations like scaling and translation can be performed and the threshold for segmentation can be soft adjusted if needed.

The output format can be selected as a file with only filtered ground points or a file with classified points. Finally, the user defines the angle of the third or the single orientation of virtual profiles, based on whatever option selected previously. After this, the processing starts and later the output file is created and a summary of 'processing time', the time required to 'read' and 'write' the data is displayed. This output of filtered point cloud is then converted to a raster with the help of open source means like GDAL, CloudCompare, LAStool etc, using the appropriate grid size.

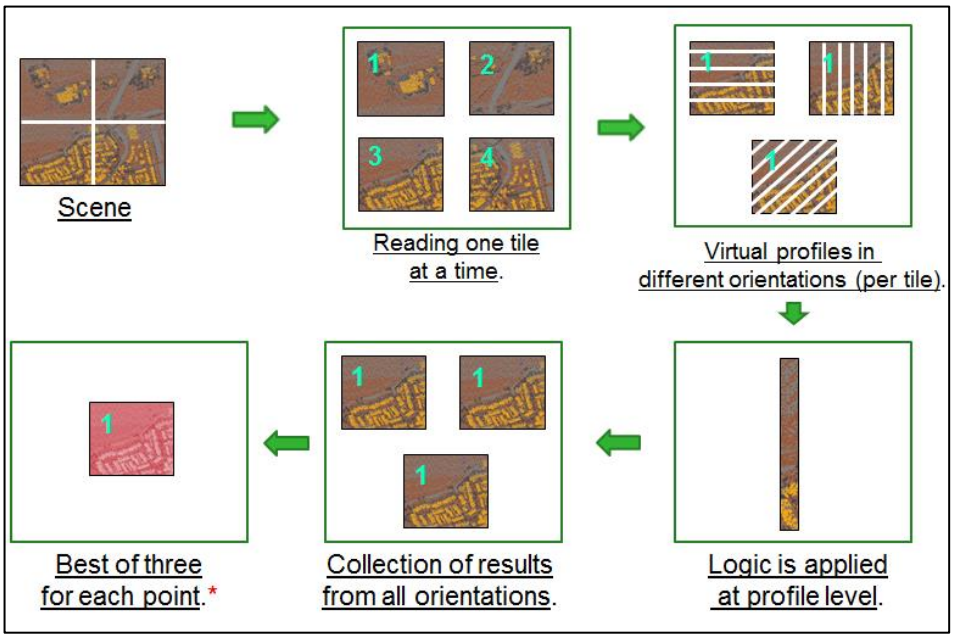


Figure 12: Illustration of Multiple virtual profiles based approach

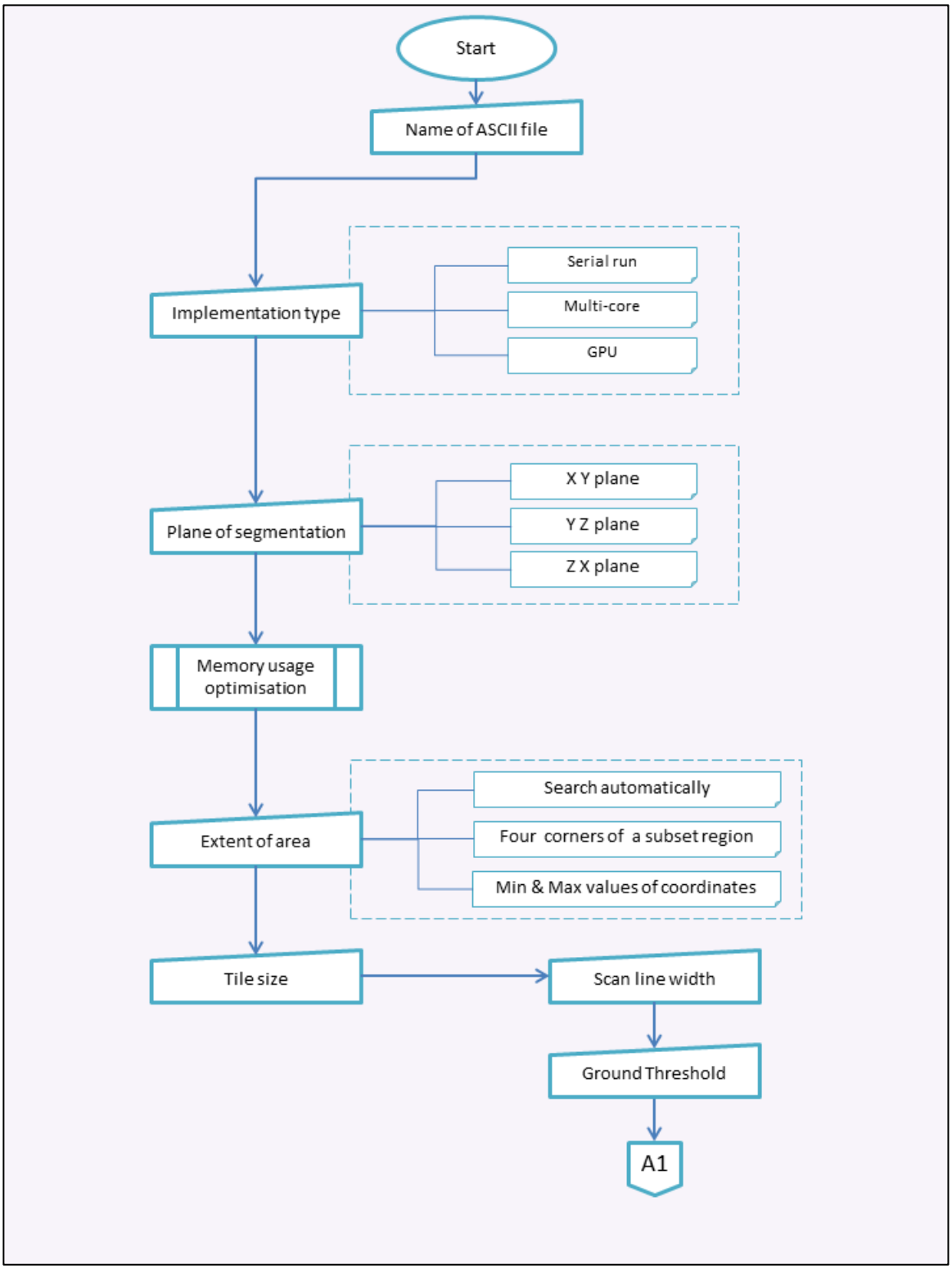


Figure 13: Flowchart of algorithm – I

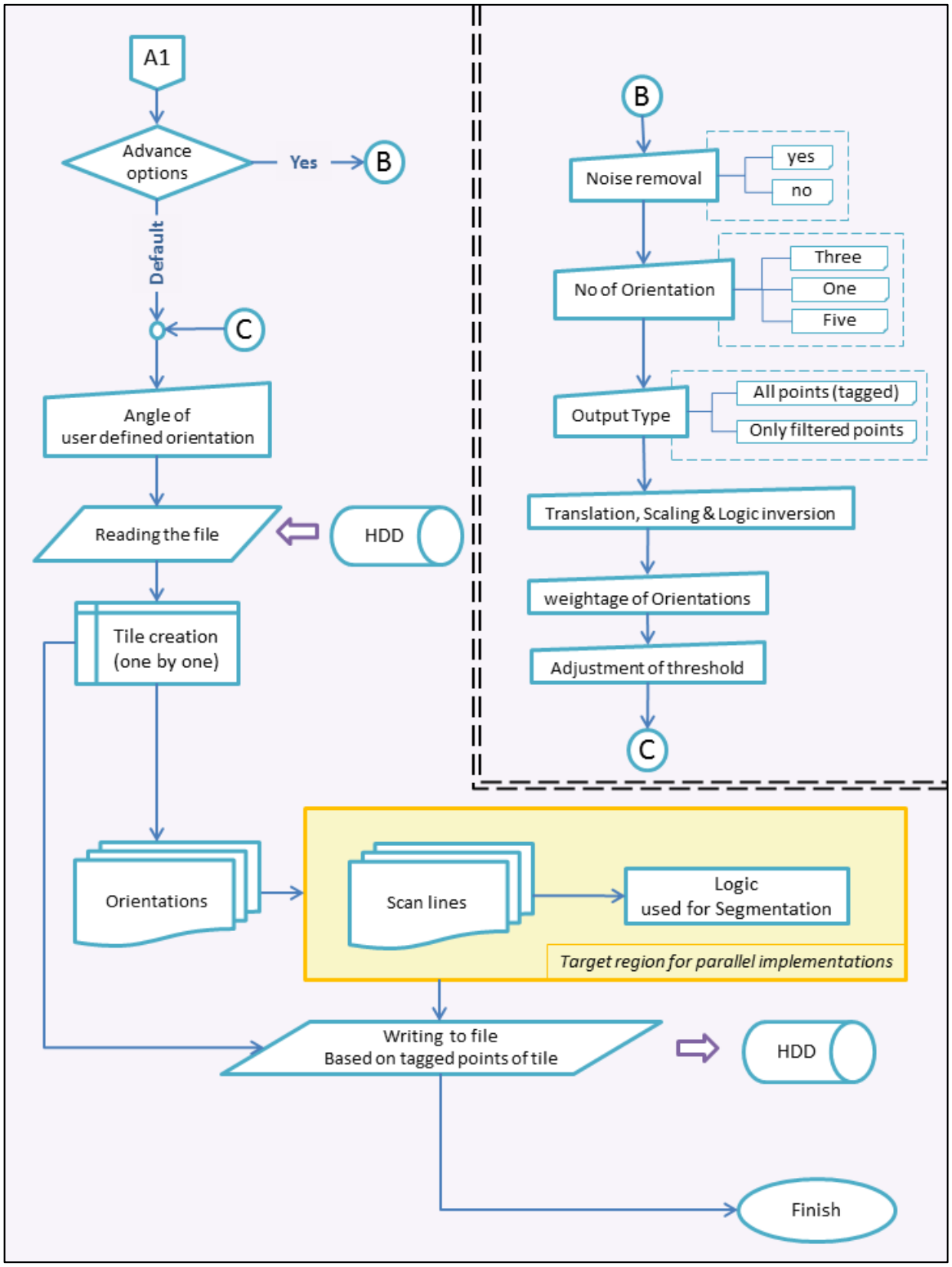


Figure 14: Flowchart of algorithm – II

In this section, the working of the program is explained from a programming point of view (shown in, Figure 15). For a tile, three 1-D containers save x, y, and z coordinates of the points and for each orientation, a unique container is created of data type 'char' which will save the 'segment id' of a point in a tile. For each orientation, virtual profiles also contain one 'location id' container which saves the position of a point in the 'tile'.

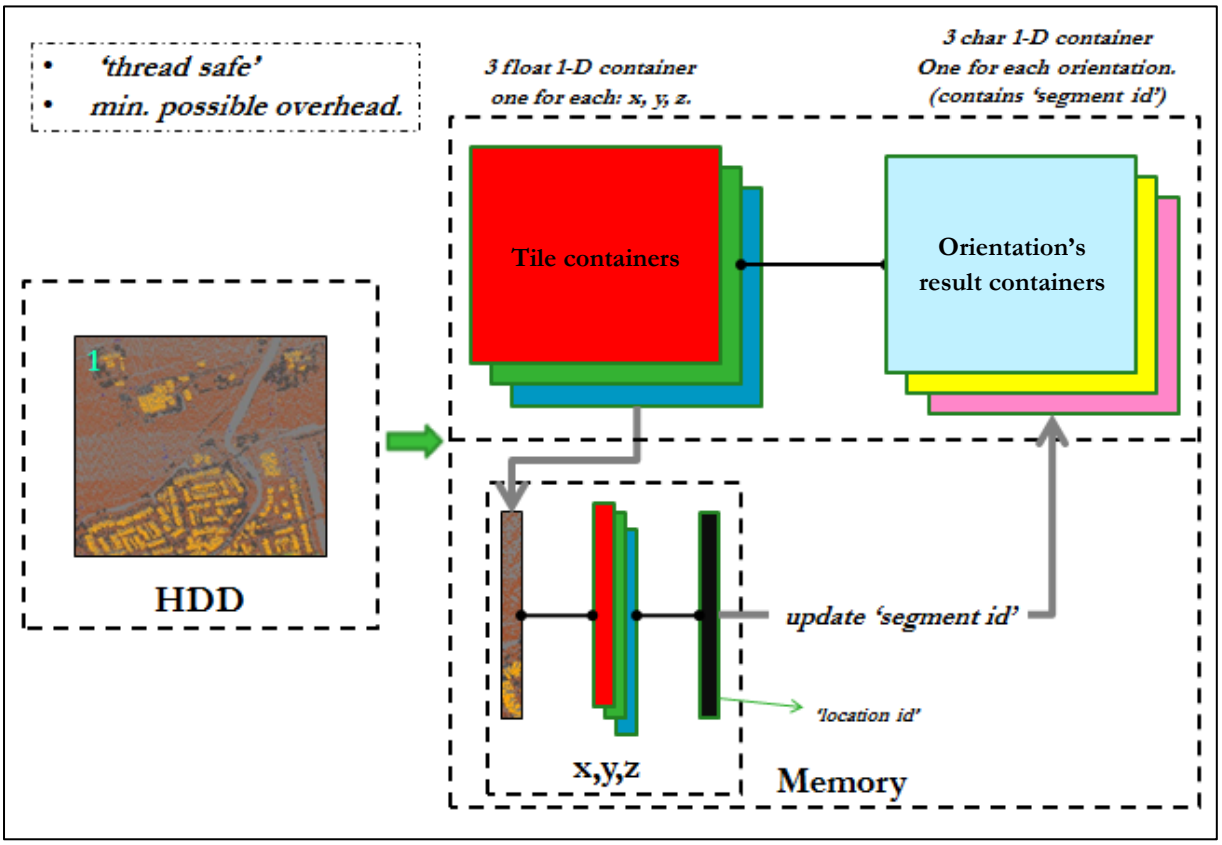


Figure 15: Working, illustrated from Programming point of view

It can be noted here that, this way of having a 'segment id' containers explicitly for orientations removes any chance of data races during the parallel implementations, making the program thread-safe and reduces the overhead of copying data in GPU, which is useful for parallel implementation. Where the simplified meaning of thread safe is that the shared data between the threads are modified in a safe way. In the next section 'user interface' is illustrated, which will further, throw some light on the working of the developed algorithm.

3.3.5. Illustration of user interface

In this section, the command-line user interface (CLI) of the developed program is illustrated. As shown in Figure 16, the user selects the implementation type after providing the input file path, where the format in which the points are written in the input file should be like the example shown in, Figure 17. All these user inputs are validated and verified (Figure 24), before the processing starts, increasing the robustness of the program, and eliminating any chance of failure in processing thus making the program reliable. In a similar fashion, other options are also selected, like, main plane, or how to specify the region to be processed (shown in, Figure 18).

```
     : An open source point cloud processing tool !!



Implementation specifics: GPU/co-pro and Multi-Core run
 Segmentation criteria : Function based weighted mean
 Note: make sure the ASCII file is without any header information,
 and contains the standard format: x, y, z separated by whitespace in each line.


 Enter the file path (include the extension
 Input file path -> test.txt

 Select the implementation you want to use :

 ¦ 'a' for serial run.
 ¦ 'b' for Multi-core.
 ¦ 'c' for GPU/co-pro.
response ->
```

Figure 16: Interface - selection of implementation type

```
179.71 -96.9146 131.932
179.873 -96.6036 131.856
180.2 -96.3028 131.814
180.558 -96.5059 131.947
180.905 -96.6578 132.051
181.335 -96.4714 131.932
181.684 -96.4714 131.795
181.86 -96.1265 131.698
178.638 -96.1382 131.719
178.915 -96.1382 131.71
```

Space seperated coordinates

Figure 17: Format of input file

```
How do you want to specify the region which will be processed ? :

 ¦ press 'a' , if you want to specify minimum and maximum value of
   x and y or y and z or z and x depending upon the selected major plane.

 ¦ press 'b' , if you want to explicitly specify all the four corner points
   [useful for sub-setting the region]

 ¦ press 'c' , if you want the program to automatically detect the bounds
   w.r.t. selected major plane.

response ->
```

Figure 18: Interface - specifying the region

The user is made aware of the safest size of the container as well as the extent along the main plane from the display, before asking for 'tile size' (illustrated in Figure 19 & Figure 20). An approximate estimation of equivalent no. of points corresponding to this container size, and 'point density' is also displayed along with the above information. This helps the user in deciding the appropriate 'tile size'. The option of using 'tiles' is also useful when the system can't handle the whole scene, tiling enables the processing of data in a streaming manner.

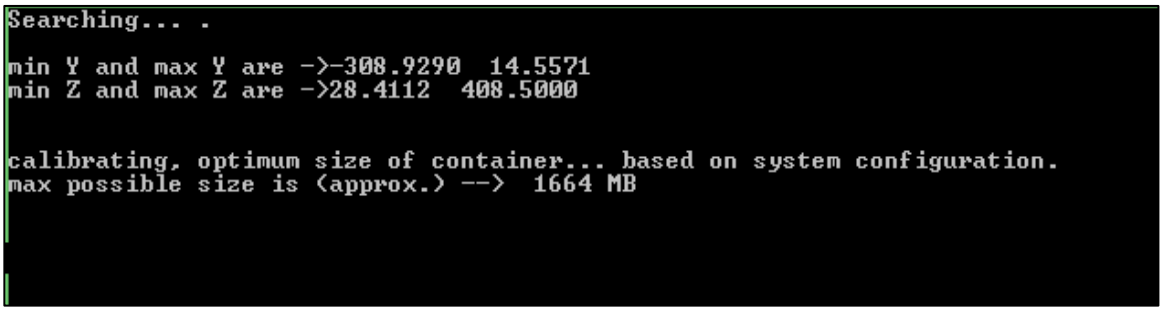


Figure 19: Interface - safe size of container

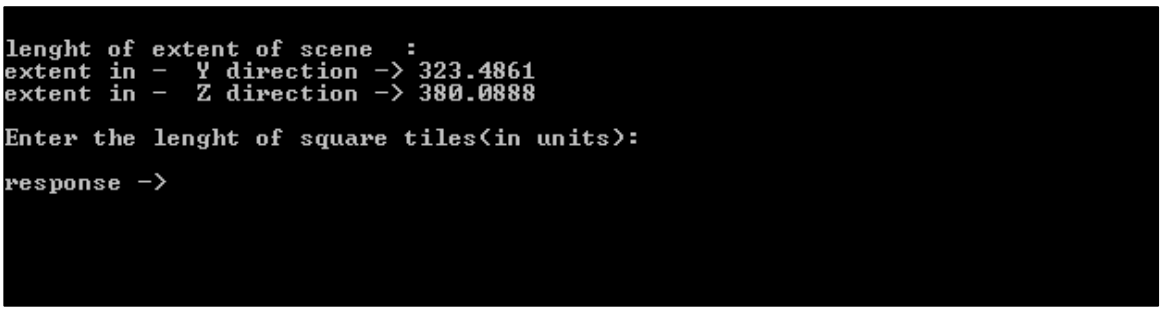


Figure 20: Interface - extent & length of tile

All other inputs are asked in a similar manner, along with 'advance options' (shown in Figure 21 & Figure 22), and output file can contain 'filtered points' in the same format of input (Figure 17) or 'classified points' in the format shown in, Figure 23 . Lastly, if the user decides to process the scene tile wise then, after each tile the 'processed points' are written to the file, this info is also displayed on screen (shown in, Figure 24).

```
Advance settings/options :
  | 'y' for yes
  | 'n' for no
response ->  y

Do you want filtered points and non filtered points in separate file ?

response, yes(y) or no(n) ->  n

Is your scene inverted ?

response, yes(y) or no(n) ->  n

Do you want a single orientation with user defined angle of orientation ? :

response, yes(y) or no(n) ->  n

Do you want 5 orientation,
 viz., vertical, horizontal, 3rd(user-defined), 4th at 45 %, 5th at 135 %  :

response, yes(y) or no(n) (default is 3 orientation) ->  n
```

Figure 21: Interface - advance options - I

```
Enter the angle for 3rd(user-defined) orientation :

response with angle(degree) ->  45

Do you want to prioritize a particular orientation ?

response, yes(y) or no(n) ->  n

The algorithm is sensitive to outliers. Press 'y' if you want to manually remove
 outliers

response  yes(y) / no(n)->  n

Do you want to adjust the threshold for function based weighted mean ?

response, yes(y) or no(n)  ->  n
```

Figure 22: Interface - advance options – II

```
12.419746398926 -0.121159009635 2.186000108719 1
12.350053787231 -0.179752007127 2.187000036240 1
12.284005165100 -0.236415997148 2.187999963760 1
12.206877708435 -0.301692992449 2.186000108719 0
12.140905380249 -0.356943011284 2.187000036240 0
12.071956634521 -0.414869010448 2.187000036240 0
12.604799270630 0.042448002845 3.532000064850 1
12.659689903259 0.089086003602 3.569000005722 1
12.662664413452 0.091759003699 3.586999893188 1
12.663409233093 0.092427998781 3.605000019073 1
12.661176681519 0.090421997011 3.621999979019 1
```


Figure 23: Output with point's 'segment id'

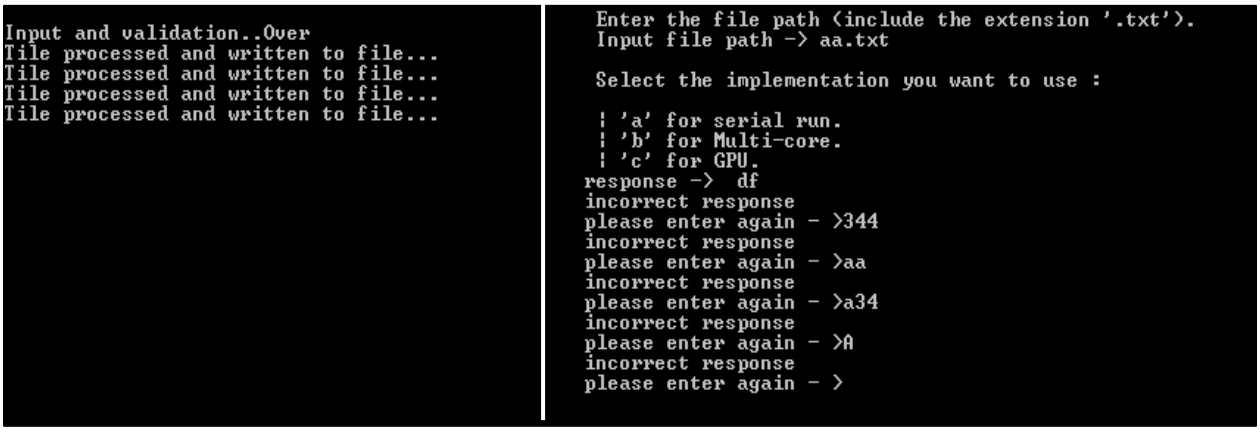


Figure 24: Interface - showing display while processing & an example of 'Input verification'

3.3.6. Running test cases, parallel implementation and accuracy assessment

Smooth execution of the program and its robustness is checked, also various examples where the tool can be used, are illustrated. Then using sample datasets from different sources, the results are discussed in detail.

For parallel implementation, OpenMP and OpenACC are used, while there are other options too but these APIs are the more suitable. Where the same code can be used for offloading the computation to MIC or GPU just by selecting the appropriate 'device id' in the 'compiler directive'. Though a lot depends on the compiler which is to be used, like for GCC, another compiler is needed to be built (by the user) from 'source' with appropriate flags meant for offloading computation to GPU or MIC (Many Integrated Core), same measures are needed for compiling the code where OpenACC was used. Though, this extra effort is not required in some commercial compilers (further explained in Section 4.1.2).

In multi-core implementation (Figure 25), virtual profiles are distributed among the threads. Different scenarios where SIMD, MIMD, no. of threads are varied and results are discussed. Also, the way these profiles are distributed (dynamic or static), also affects the parallelisation (discussed in Section 4.1.2).

In GPU implementation (Figure 26), apart from offloading the computations, global variables are also copied to GPU memory and virtual profiles are distributed. The data is copied into the GPU and then the results are copied back. That is, only 'segment id' containers of orientations are copied back from the GPU in an 'async' manner, this also enables the continuous processing of different orientation, without waiting for the transfer/copy to finish.

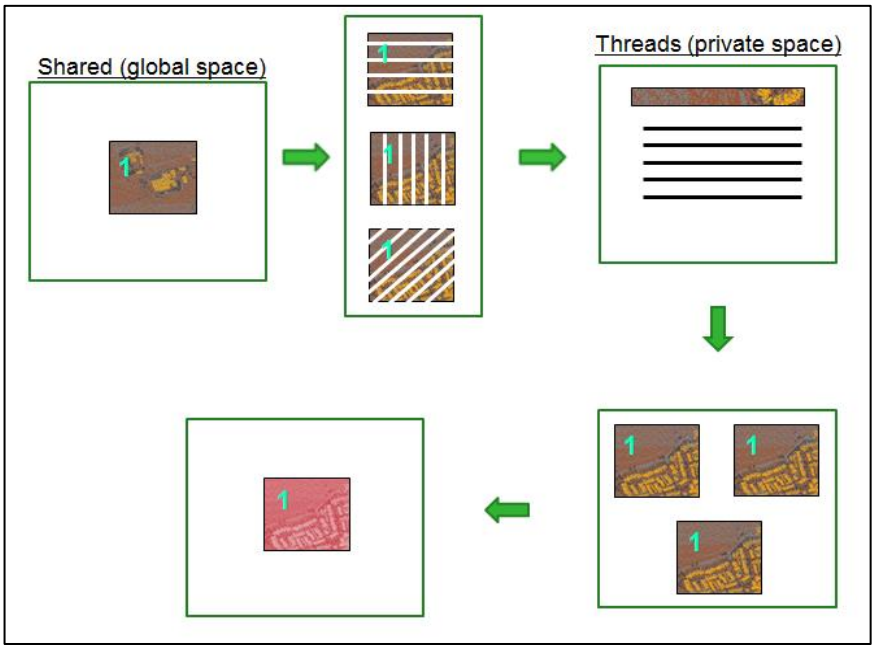


Figure 25: Multicore implementation

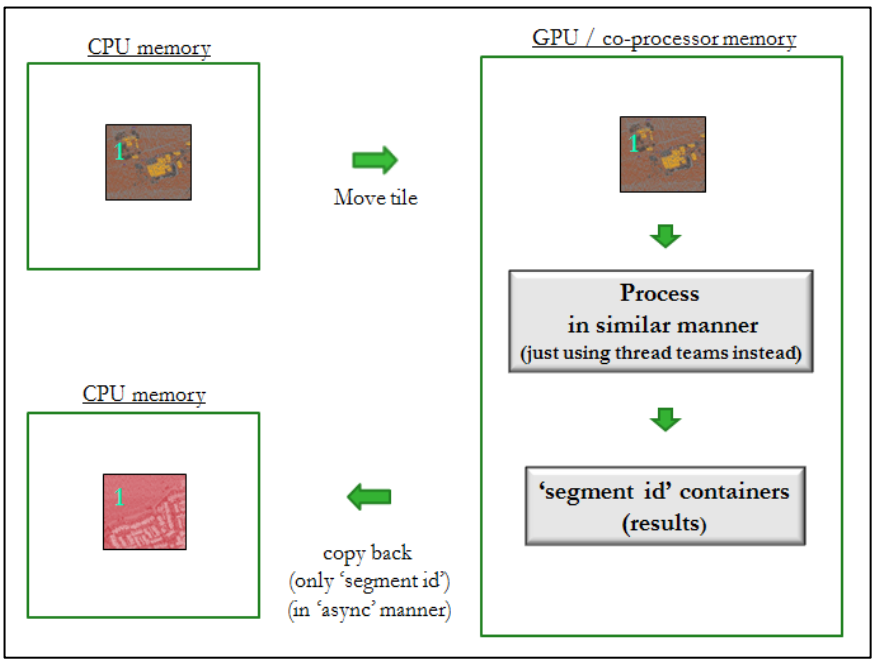


Figure 26 : GPU implementation

After this inter-comparison is done based on processing speedup, implementation effort, lines of code, etc. Lastly, the performance of proposed logic is tested with different types of ISPRS filter test datasets. For this, a program is written to test the accuracy of each point in a scene, the result of which is then discussed (in section 4.1.3 and section 4.2.3).

4. RESULTS AND DISCUSSION

4.1. Results

4.1.1. Test cases

In this section, datasets from different sources which are processed using the proposed algorithm are shown, including the cases when the algorithm does not produce good results. Next, intercomparison of serial and parallel implementations is done, followed by accuracy assessment.

In 'Case 1' we take an airborne dataset (shown in, Figure 27). This LiDAR dataset was taken from the AHN3 public dataset. And, using the proposed algorithm, shown below are the filtered surface points (in, Figure 28) along with the generated DEM from it (in, Figure 29). In Figure 30, the classified output is shown, where the surface points are shown in blue colour and non-surface points are shown in red colour. This dataset was processed by using the tiling approach. The 'tile size' taken has to be more than the largest non-surface object and ground threshold was taken as 0.4 units. The tiles are processed one by one, and we can notice the seamless integration of tiles, shown in Figure 31 and Figure 32.

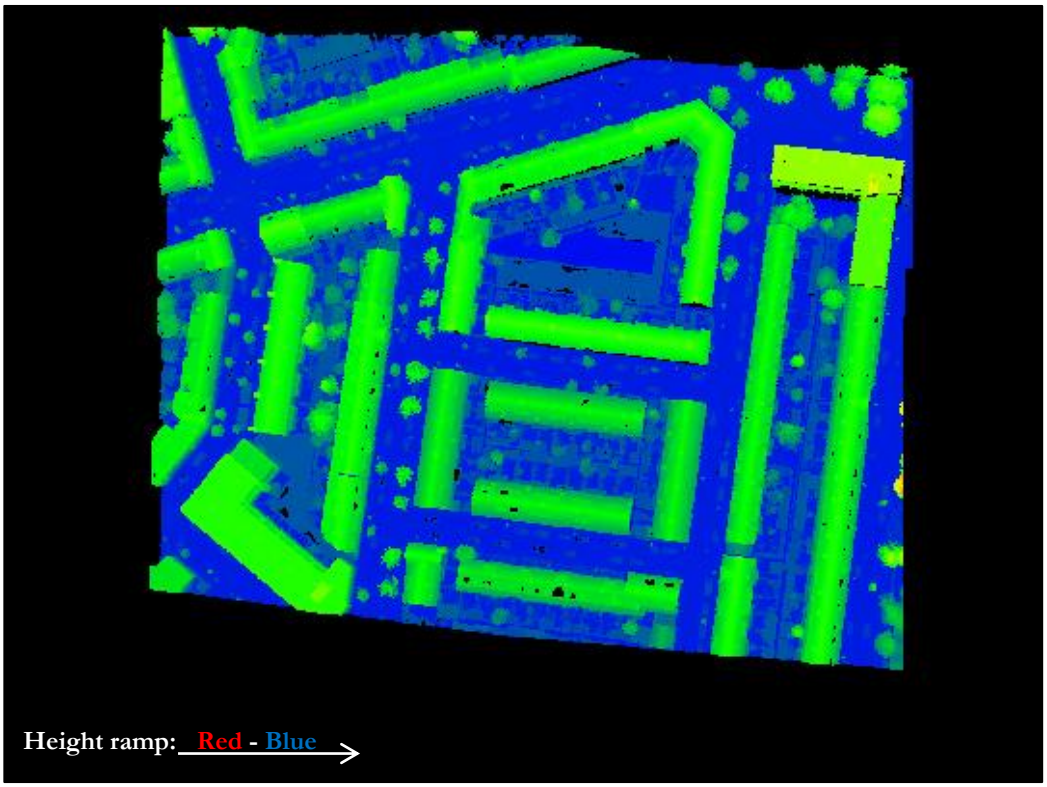


Figure 27: Case 1 – airborne dataset |colour – height ramp

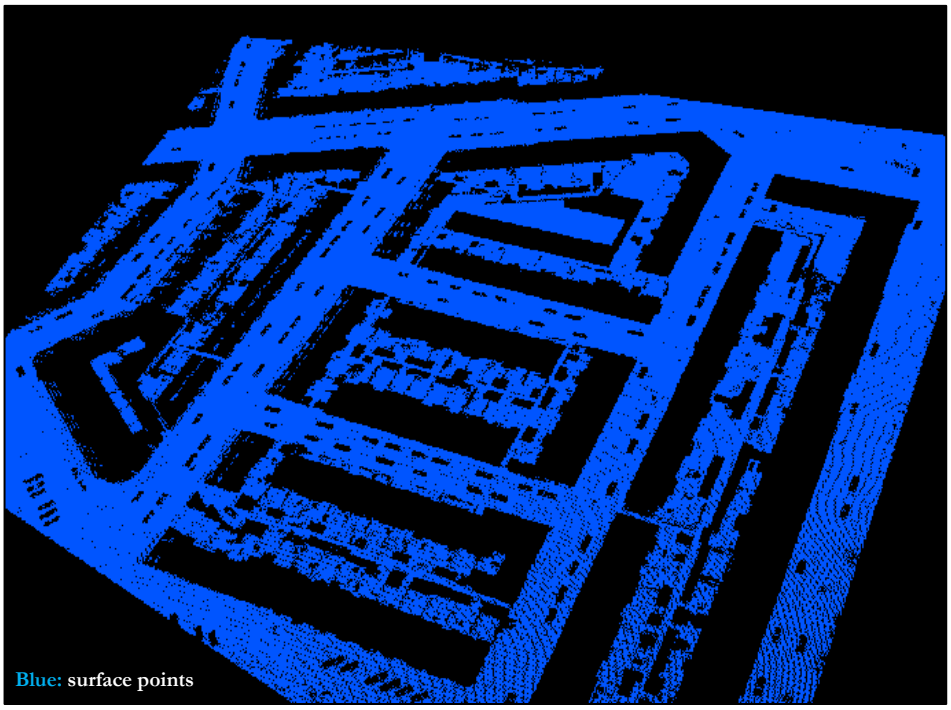


Figure 28: Case 1 – surface points, filtered out from the scene

Figure 29: Case 1 – DEM made from the extracted surface points (rasterization)

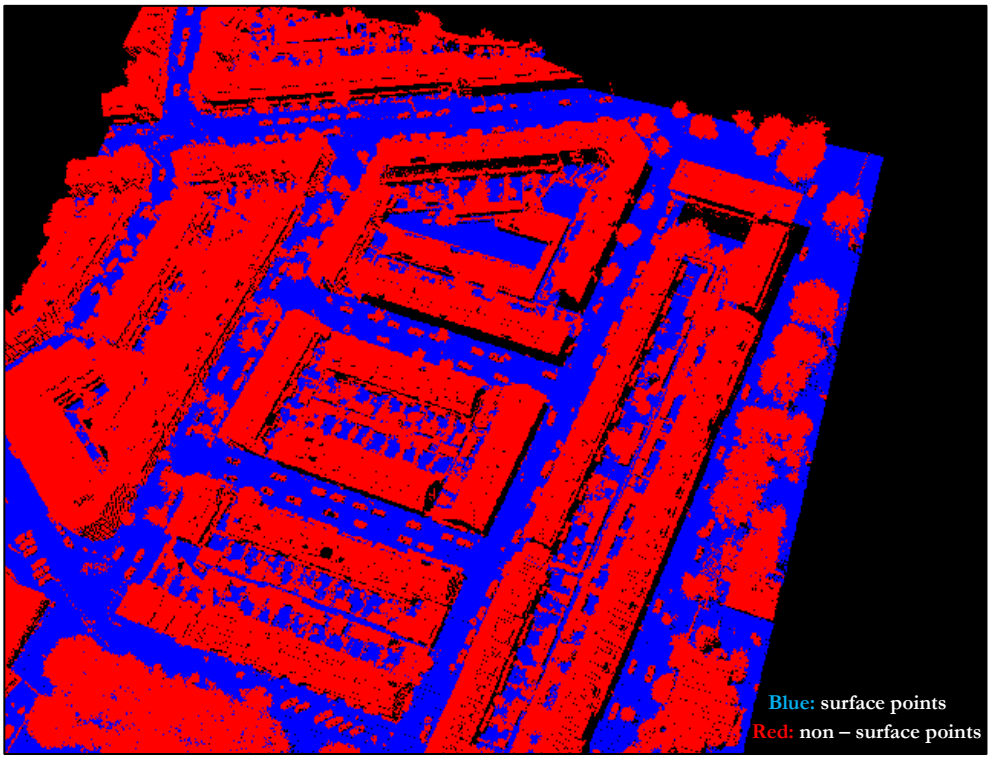


Figure 30: Case 1 – classified points, using the proposed algorithm

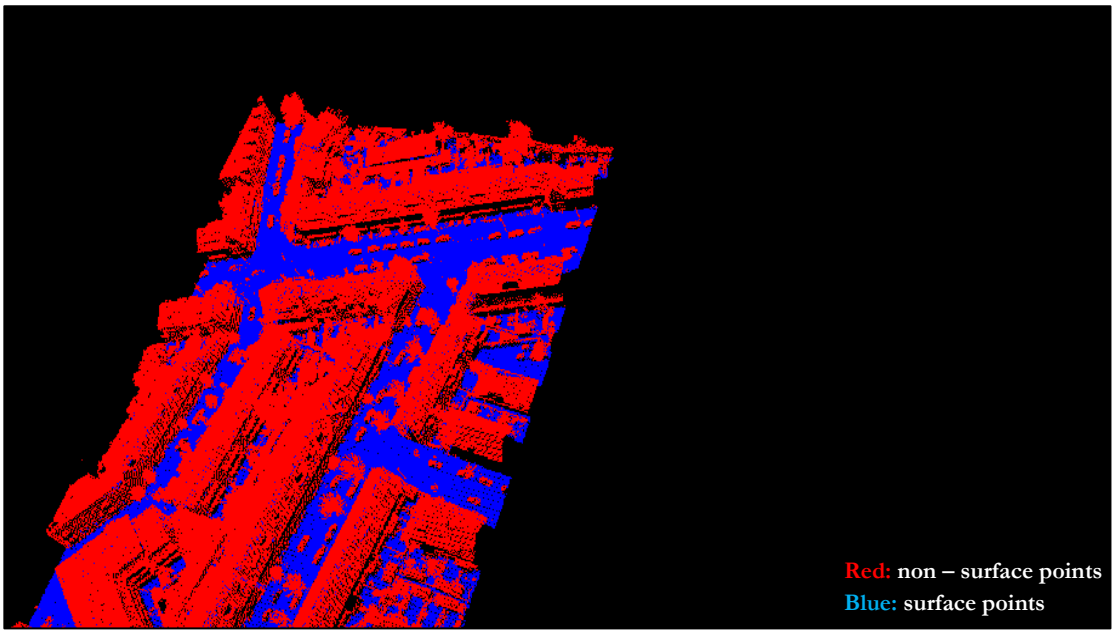


Figure 31: Case 1 – seamless integration of tiles – I

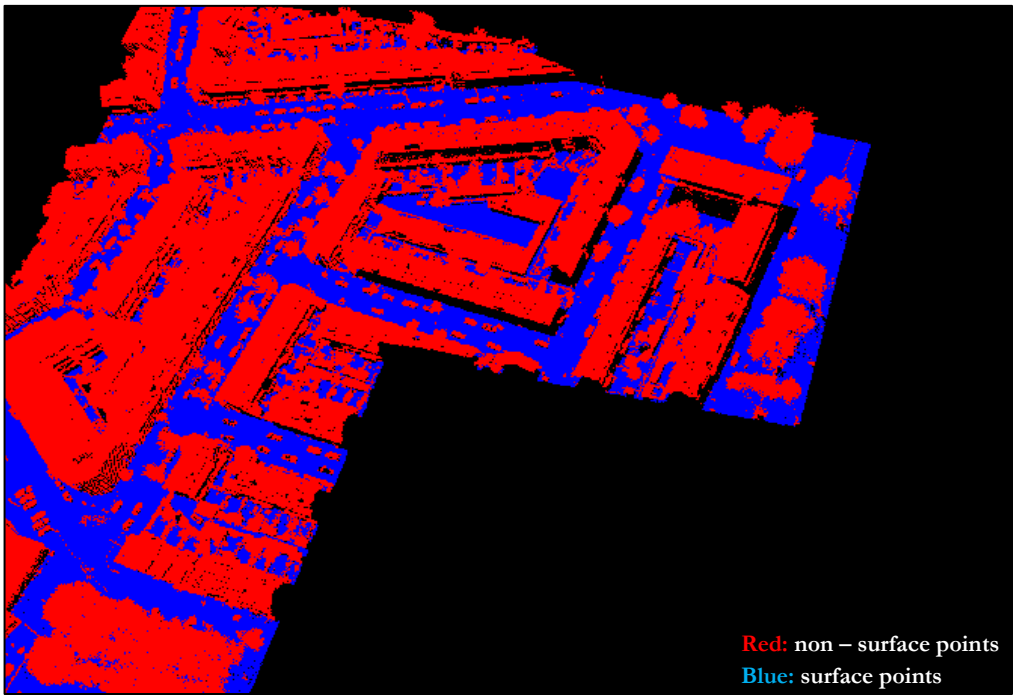


Figure 32: Case 1 – seamless integration of tiles – II

In 'Case 2' we take a façade of a building (shown in, Figure 33), and use the proposed algorithm. This data was acquired using RIEGL VZ-400. The dataset is captured from an oblique view. Figure 34 and Figure 35 shows the extracted surface and the rasterized image of the surface. Figure 36, shows the classified output where 'blue' points are the non-surface points and 'red' belong to the surface.

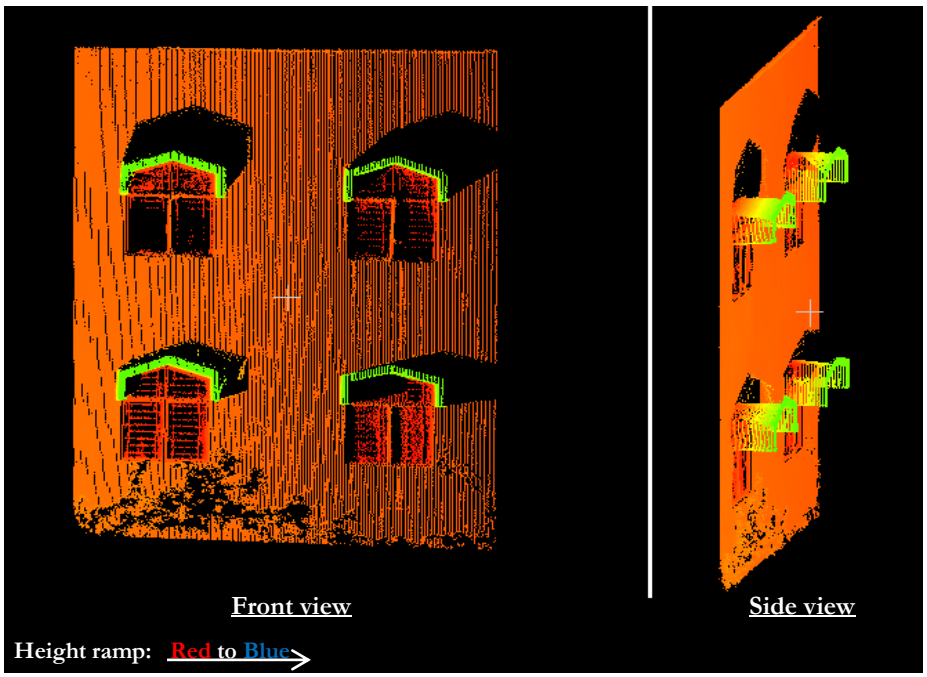


Figure 33: Case 2 – façade |colour – height ramp

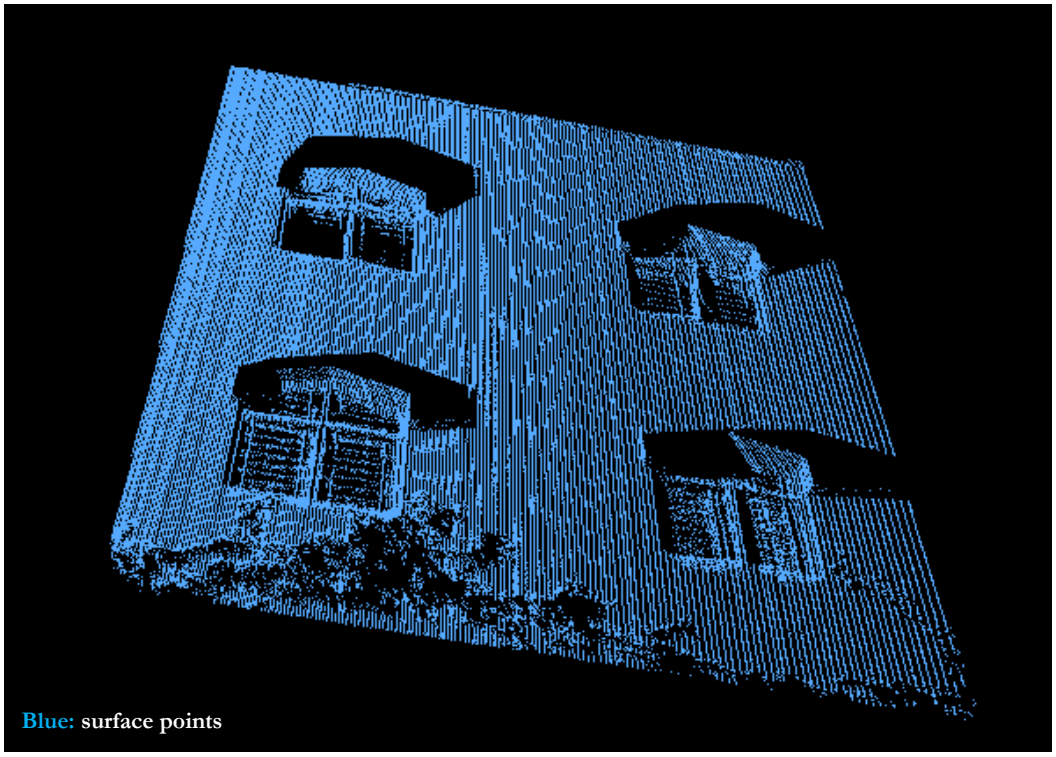


Figure 34: Case 2 – surface points, filtered out from the scene

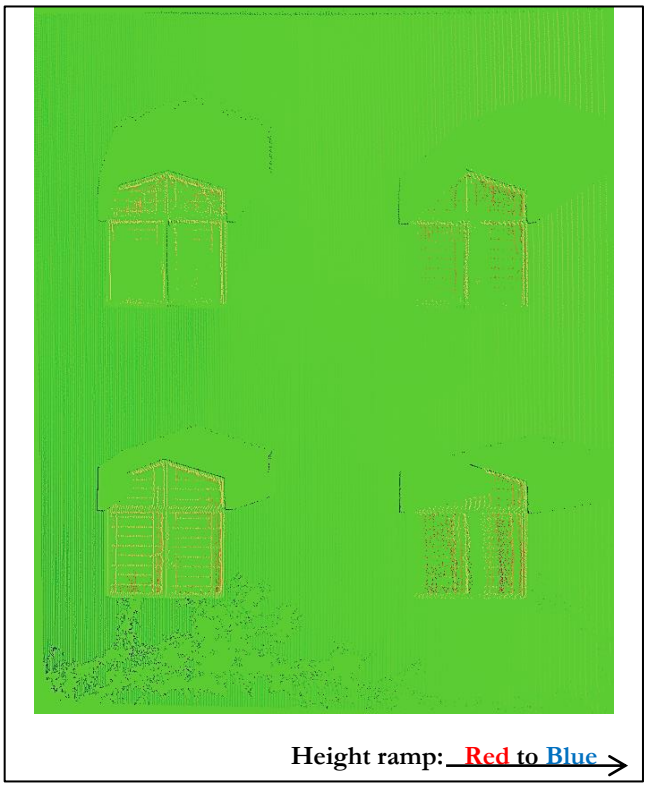


Figure 35: Case 2 – Rasterization of surface points

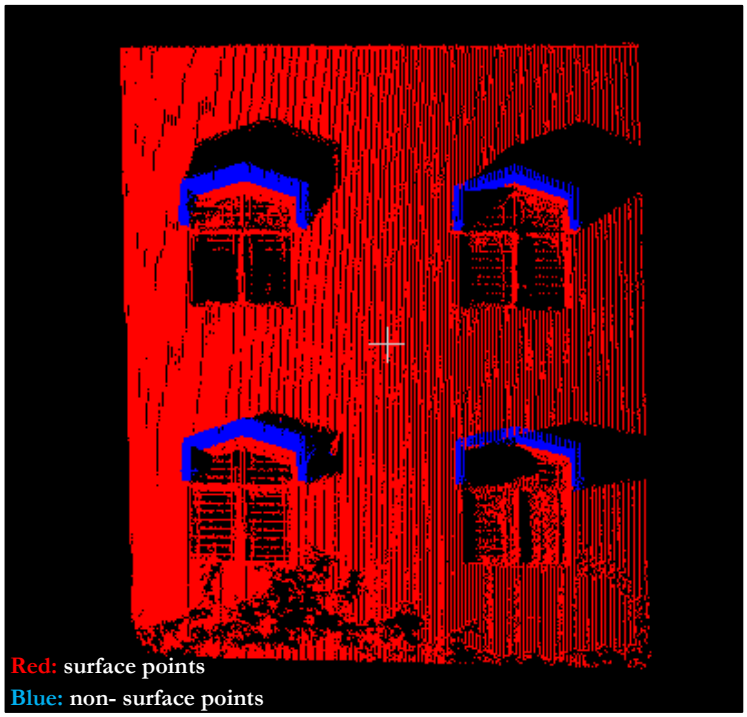


Figure 36: Case 2 – classified points, using the proposed algorithm

In 'Case 2', data is processed using 'single orientation' option. While 'Y-Z plane' was selected as 'main plane' as the wall was along that plane. The whole scene is considered for processing, 'surface threshold' was also provided (or ground threshold). Though in this case, similar results can be obtained even when the dataset is processed using tiling approach. Provided that in the latter case, 'tile size' is more than the largest object to be separated, here the 'projections' around the windows decides that.

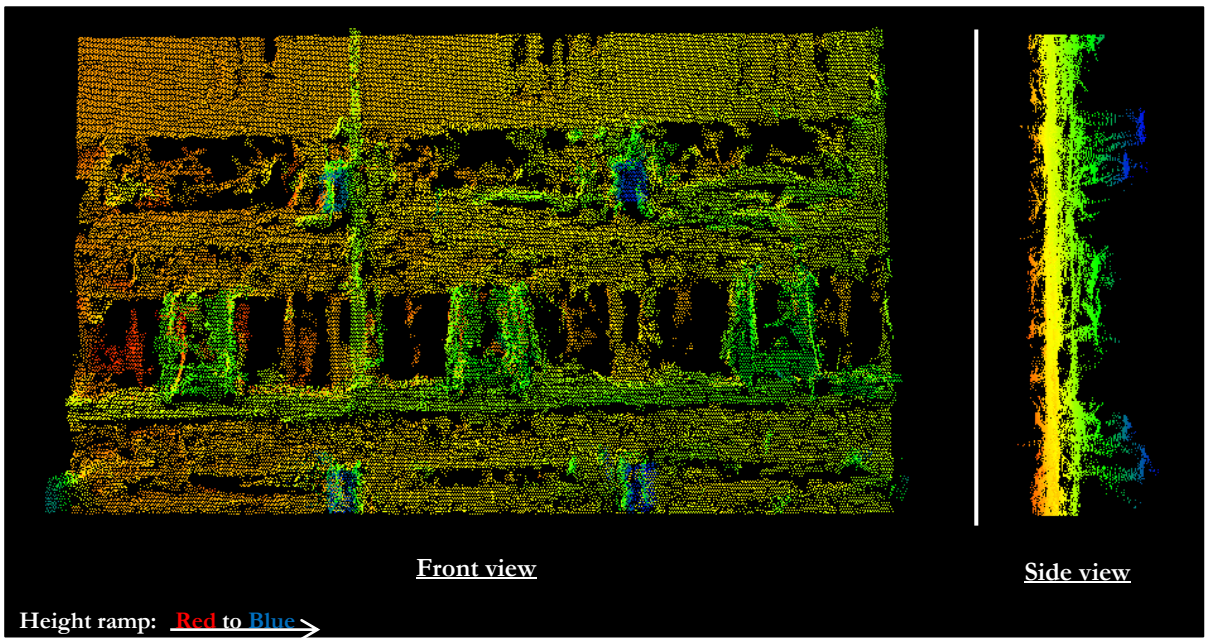


Figure 37: Case 3 - SFM generated dataset of a façade, with side view showing the 'projections'

In 'Case 3', a point cloud of a façade of a building is used. This dataset is generated using SFM technique, and it natively has noise in it (shown, in Figure 37). Single orientation is used and the surface threshold is provided, to process this data. Tiling is not used and the main plane is selected as Y-Z plane because in the coordinate system of dataset the surface is along Y-Z plane. Figure 38 shows the filtered surface points, and Figure 39 shows the rasterization of these points. While in Figure 40, we can see the classified points where blue points are the surface points and red are the non-surface points.

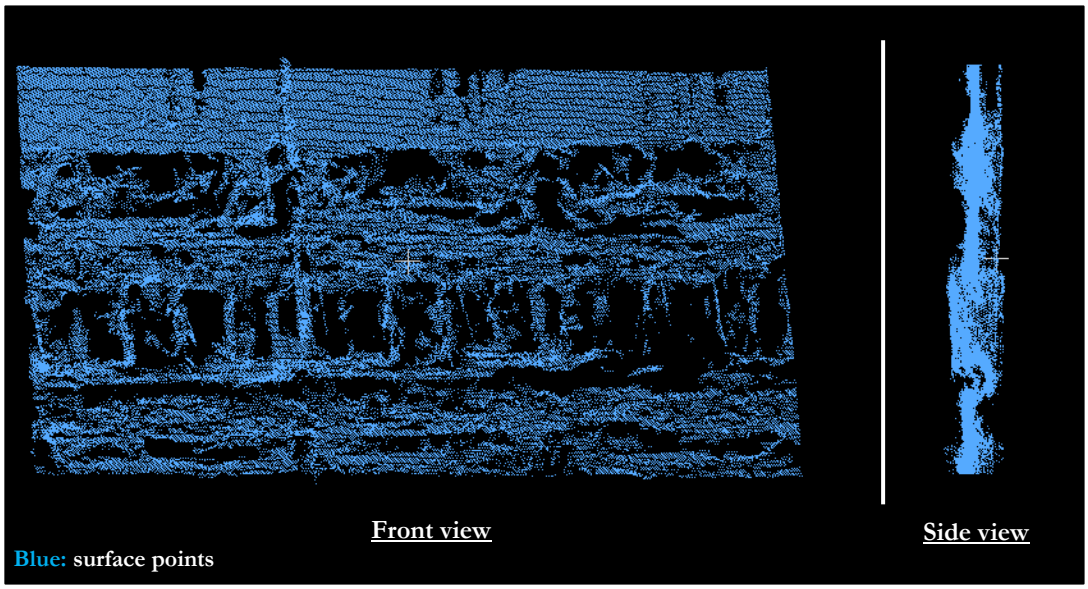


Figure 38: Case 3 – surface points, filtered out from the scene

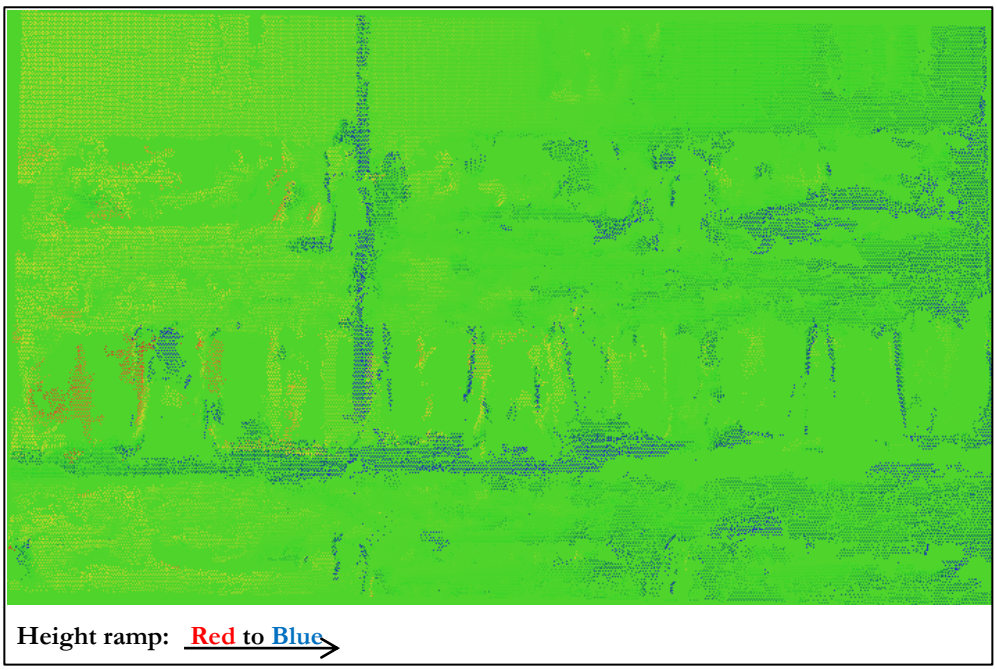


Figure 39: Case 3 – Rasterization of surface points

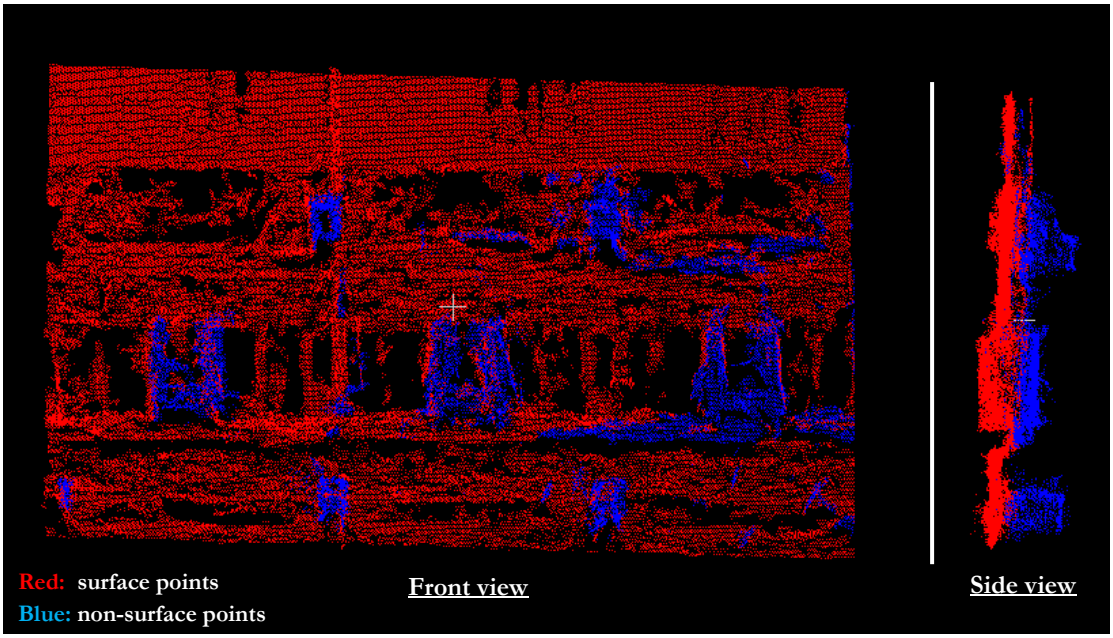


Figure 40: Case 3 – classified points, using the proposed algorithm

Apart from these possible data sources, the algorithm is tested on some other samples, but this time intentionally varying key input parameters. This will help in understanding the significance and effect of these input parameters viz., profile width, tile size, surface threshold. The effect of point density, point distribution and noise is also studied.

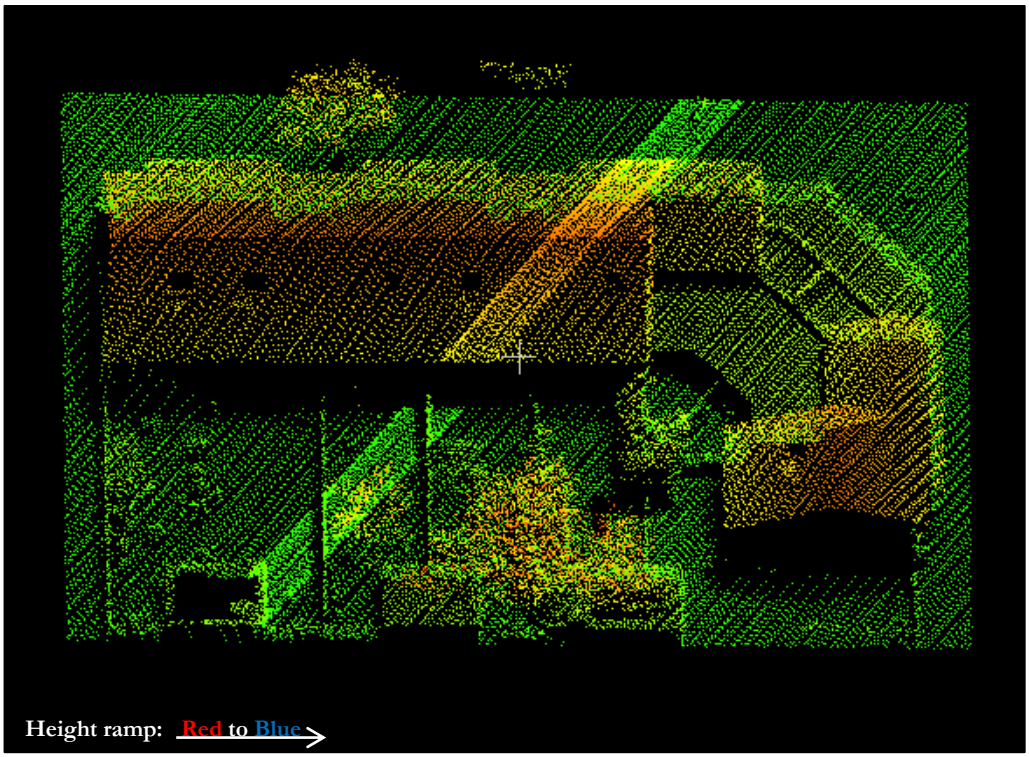


Figure 41: Point cloud dataset of a house

Figure 41 shows a general scenario of point clouds of a house and its surrounding. In the first run (Figure 42) the tile size is taken bigger than the largest non-surface object in the scene. In the second run (Figure 43), the tile size is taken smaller than the largest non-surface object. While the tile size is varied other parameters are kept constant. In Figure 42 and Figure 44 profile width is varied.

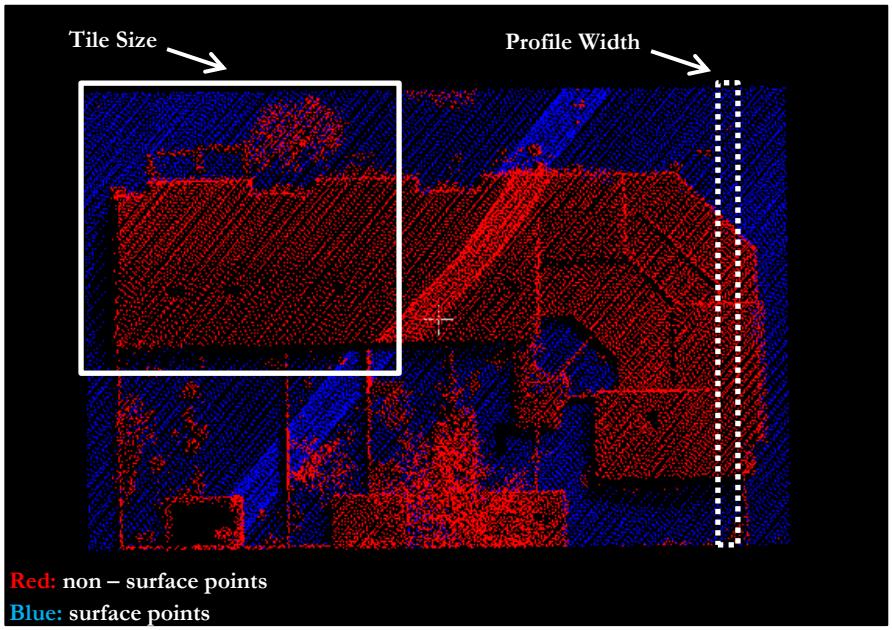


Figure 42: First test run - understanding the input parameters

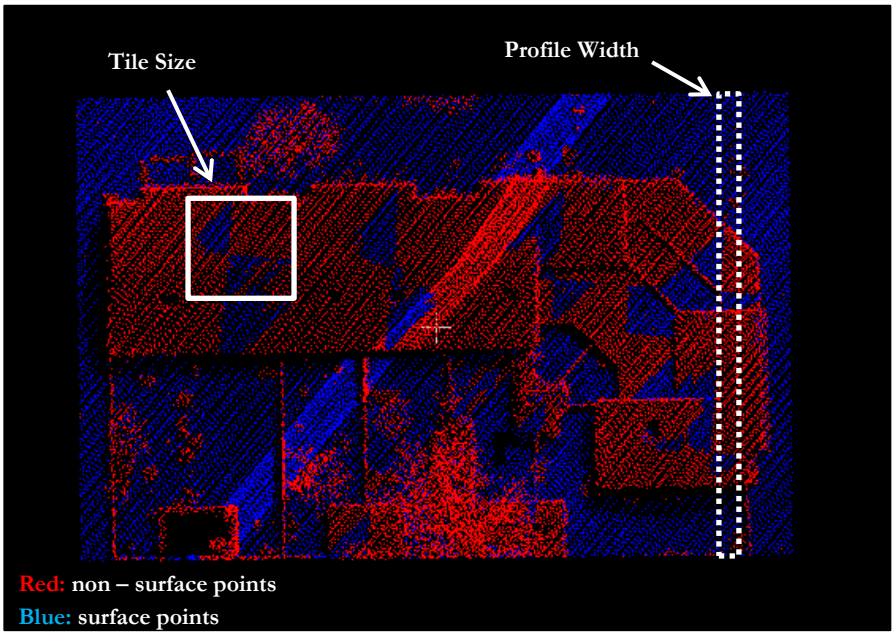


Figure 43: Second test run - understanding the input parameters

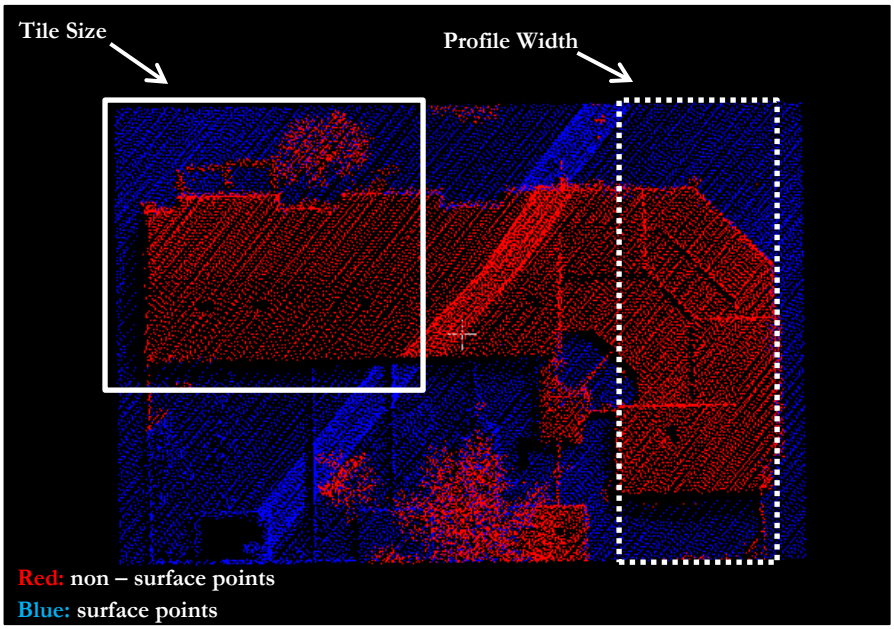


Figure 44: Third test run - understanding the input parameters

Figure 45 and Figure 46, illustrate the effect of the surface threshold, use of multiple orientations or single orientation of virtual profiles, and weight given to a particular orientation's result.

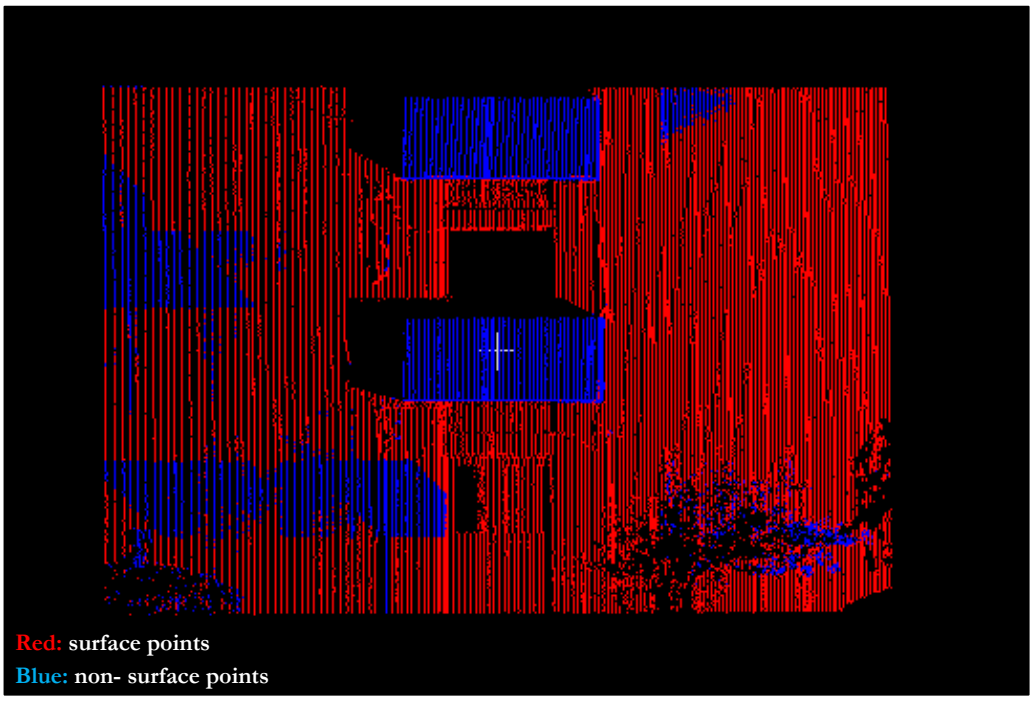


Figure 45: Example 1 - significance of parameters

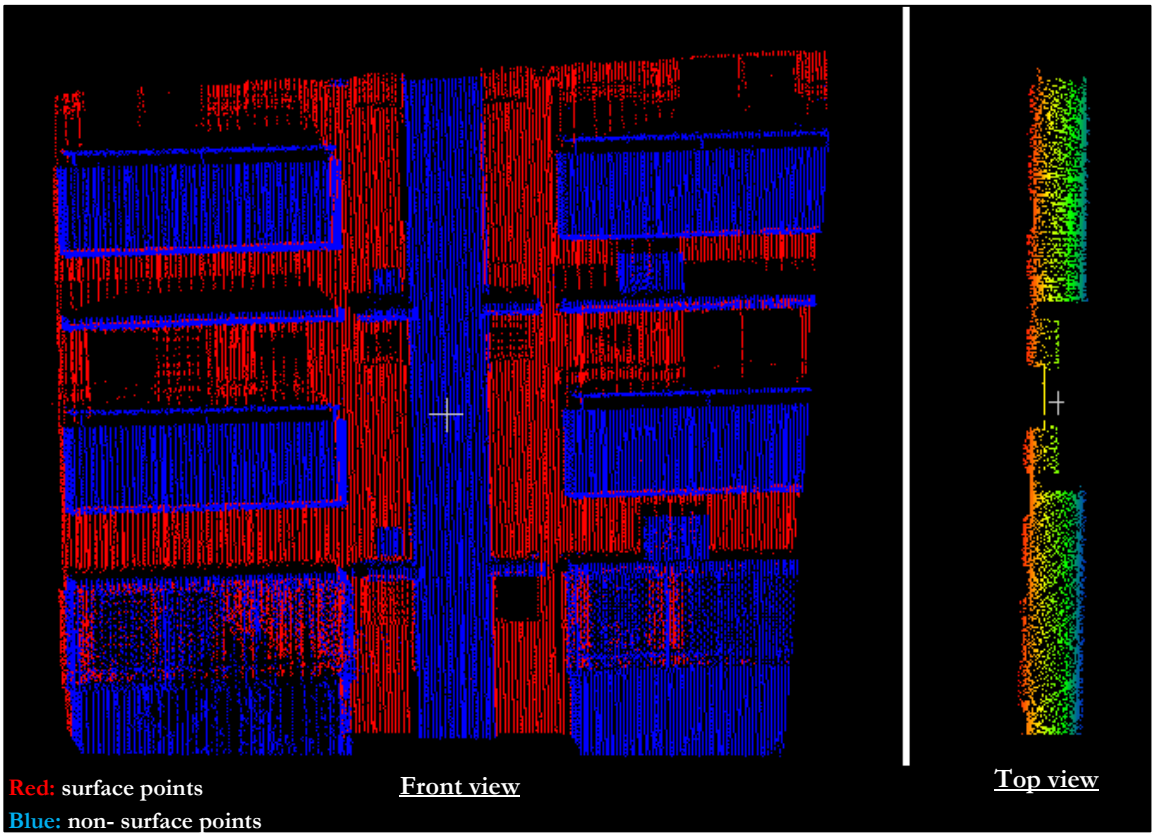


Figure 46: Example 2 - significance of parameters

4.1.2. Parallel implementations - Inter-comparison report

In this section, the results related to parallel implementations are shown. Figure 47 shows, an example where execution time of serial and multicore processing is compared. In this case, MIMD approach was used, without optimising the scheduling of threads. Parallel processing time shows a significant improvement over serial run. Figure 48 shows the effect of increasing the number of threads in multicore implementation and Figure 49 shows the speedup comparison of the serial run, multicore-MIMD, multicore-SIMD and GPU (explained later in Section 4.2.2).

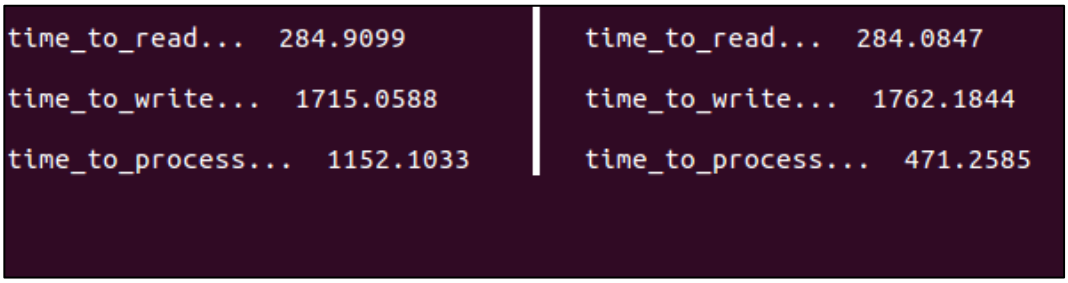


Figure 47: Execution time - serial vs parallel run

4.1.2.1. Speedup comparison

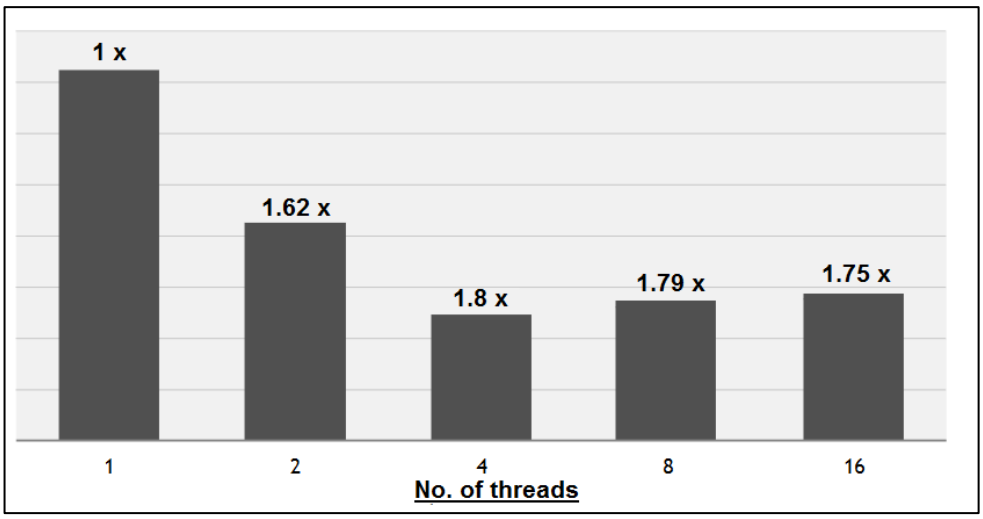


Figure 48: Parallel implementation - variation in no. of threads

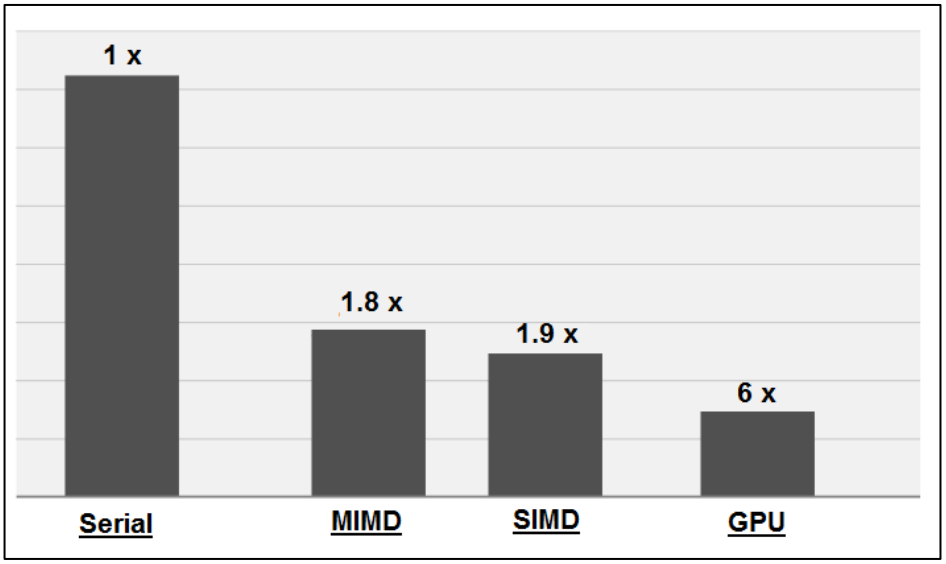


Figure 49: Speedup comparison of parallel vs serial run

4.1.2.2. Lines of Code (LOC)

The summary of actual 'lines of code', is shown in, Table 1. Here for each implementation the 'lines of code' are mentioned, all these implementations are merged into a single program, thus making total '13579' lines of code for the program, which is also released as open source (link provided in, APPENDIX D). Approx. 30 to 40 lines of code are API related for each parallel implementation.

Implementation type	Lines of code (approx.)
Serial	4503
multicore	4533
GPU/co-processor	4543

Table 1: lines of code - for individual implementations

4.1.2.3. Implementation effort

In terms of implementations effort required for parallel implementation, the specifics are discussed in this section. Options like C++11 STL's Threads, Boost, OpenMP or OpenACC (APIs), Intel threading building block (TBB), Intel Cilk Plus or natively using Pthreads are preferred for multicore implementations. A combination of OpenMP with 'threads' is also viable. The reason why OpenMP is preferred for this work is, due to the day by day increasing community support for it. With time this lead to addition of many functionalities in it, thus making it the top contender here. The repetitive tasks are automatically handled in OpenMP like the creation of threads or something like load balancing. Another plus is the advantage while debugging, due to a clean and straightforward syntax. Many compilers are adding support for OpenMP and with time this will result in increasing the usability of the code written in this work. Using OpenACC instead of CUDA for GPU implementations, have similar arguments, like automation of repetitive steps, straightforward syntax etc. Both of these APIs are open source, use compiler directives, and both supports GPU and CPU (syntax illustrated in, APPENDIX C). A compiler ignores the 'directives' if it does not supports these APIs, but the code is still compiled successfully.

Implementation effort required for OpenMP and OpenACC is more feasible and convenient as compared to above-mentioned options. But even after that, one can't directly run a code written using OpenMP or OpenACC directives. If commercial compilers are used like NVIDIA's PGI compiler (ver. 17.5, at the moment) then full support of OpenACC 2.5 can be availed, but PGI does not support OpenMP's GPU offloading functionalities (instead OpenMP 3.1 standards is supported). Conversely, Intel compiler (ver. 17.0, at the moment) fully supports OpenMP 4.5 standard, including offloading but only for MIC, not for GPUs and does not support OpenACC at all. This is due to the 'split' in the industrial support provided by Intel and NVIDIA for these open source APIs. That is, Intel supports OpenMP and NVIDIA supports OpenACC, moreover, as Intel's MIC is marketed as comparable to NVIDIA's GPU.

For open source compiler, the two most preferred are LLVM Clang and GCC. As far as GCC goes, currently (ver. 7.1) OpenMP 4.5 is supported and up to OpenACC 2.5 standards is partially supported. Another aspect is that for GCC, old architecture of GPUs is not supported (Fermi). While, for

LLVM Clang (ver. 4.0, at the moment) GPU offloading is in the development stage, MIC offloading is possible and some features of OpenMP 4.0 and 4.5 are already available.

When using GCC, additional offload compiler is needed. That can be built, from the source with appropriate flags (commands & flags are shown in, APPENDIX C). The main compiler runs the code and passes the portion of parallel code based on OpenMP/OpenACC directives to this offload compiler, which generates the ptx code (pseudo assembly language used in GPUs) that will execute on GPU.

4.1.2.4. Linearity (scalability) and Parallel efficiency

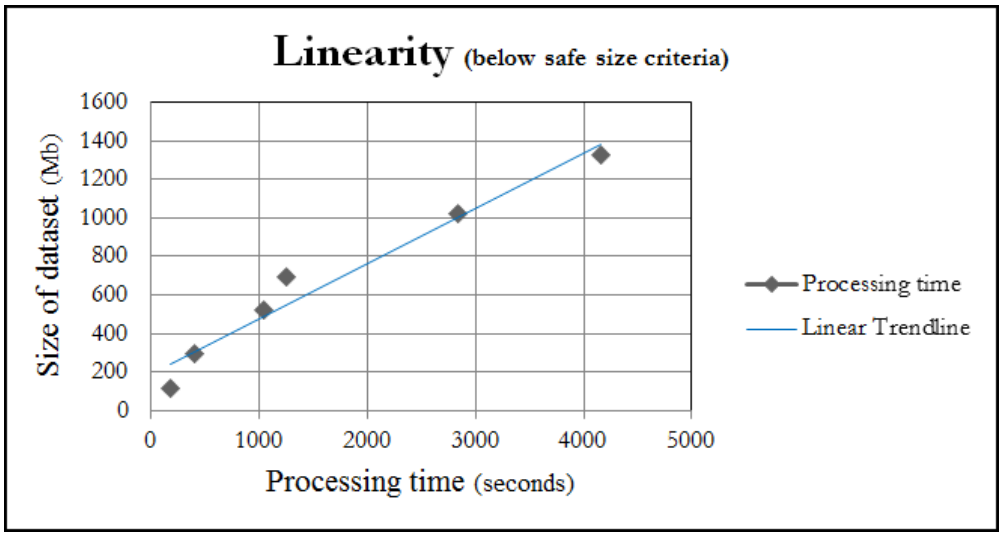


Figure 50: Checking linearity, when tile size is as per 'below safe size' criteria

Checking the linearity or load scalability of the program is important when parallelly processing, massive datasets. Here, load scalability means the stable performance of a program with increasing the load, while linearity (better criteria) is the linear increase in processing time with an increase in data size (linear time complexity).

Due to the feature of tiling, any size of data can be processed by the tool, making it scalable in terms of 'load' if only 'processing time' portion of 'total time' is considered. Because the 'total time' will increase gradually if no. of tiles is increased as this will increase the I/O overhead. As long as the 'tile size' is as per the 'safest container size criteria', the parallel implementation of algorithm scales well in terms of 'processing time' as illustrated in Figure 50, where 'linearity' is used judge the load scalability. The 'parallel efficiency' of multicore implementation is 0.95, which can be deduced from Figure 49 (explained later).

4.1.3. Accuracy assessment

In this section, the performance of proposed logic in terms of accuracy is assessed. For this, the reference dataset of ISPRS filter test is used (Sithole & Vosselman, 2004). All the fifteen samples are processed using the proposed algorithm. The 'Type I error', 'Type II error' and 'Total error' were calculated and are mentioned in Table 2. Where 'Type I error' is the percentage of misclassified surface points per total no. of surface points. 'Type II error' is the percentage of misclassified non-surface points per total no. of non-surface points. And 'Total error' is the percentage of total no. of misclassified points per total no. of points in the sample.

Nature of terrain	Sample Id	Type I Error	Type II Error	Total Error
terraced slopes	samp11	39.54	14.56	28.88
terraced slopes	samp12	21.83	8.26	15.21
gentle slope	samp21	10.44	12.69	10.94
terraced slopes	samp22	22.40	15.71	20.32
terraced slopes	samp23	34.47	8.24	22.06
terraced slopes	samp24	25.14	6.41	19.99
gentle slope	samp31	8.97	7.55	8.32
terraced slopes	samp41	25.76	26.59	26.18
gentle slope	samp42	5.46	15.42	12.50
gentle slope	samp51	14.32	27.47	17.19
steep slopes	samp52	14.45	68.92	20.18
steep slopes	samp53	4.87	65.30	7.31
gentle slope	samp54	10.82	14.72	12.92
steep slopes	samp61	2.12	52.90	3.86
steep slopes	samp71	17.54	20.23	17.85

Table 2: Type I, Type II and Total error for all samples

For each sample, the optimum value (discussed later) of mandatory parameters viz., profile width, tile size, 'angle of third orientation' (user defined), and optional parameters viz., surface threshold, no. of orientations were used. Outputs and parameter values which were taken for each sample are shown in APPENDIX A.

Among all the samples, the algorithm performs most consistent in urban terrains where there is no sudden change in elevation (gentle slope), which can be clearly seen in samp31, samp21, samp42 and samp54. Also, low 'type I error' but large 'type II error' can be seen for mountainous scenes (samp53 & samp61) where mostly vegetation is removed.

The average 'total error' is '16.24' with a standard deviation of '7.02' for the proposed algorithm. While for other algorithms, like for cloth simulation based algorithm proposed by W. Zhang et al., (2016), the average 'total error' came to be '4.39' with a standard deviation of 2.76. Or for the morphological filter based algorithm proposed by Pingel et al., (2013), the average 'total error' came to be '2.97' with a standard deviation of '2.00'.

4.2. Discussion of Results

4.2.1. Discussion of Test cases

In 'Case 1' (Figure 27) an airborne dataset is processed, taken from AHN3 dataset. Figure 28 and Figure 29 shows the filtered points and rasterized output of the scene respectively. Processing using the tiling approach is demonstrated in this case. While in Figure 31 and Figure 32 we can observe the seamless integration of tiles this is possible due to the nature of the logic which is used to segment the surface. The function based weighted mean depends on the variability of points in a profile. So if a point lies at the corner, it will be detected correctly. And in cases where few points are detected incorrectly, that does not affect the final result, as chances are that point is correctly segmented in other orientation, so the problem at borders of a tile gets automatically corrected most of the time if appropriate tile size and ground threshold are considered.

Seamless integration of tiles won't be possible where the terrain changes abruptly, and this holds true for the correctness of surface object segments too. So when it comes to a terrain where ground changes are large or abrupt, the logic fails to provide good results, which also results in the poor merging of tiles where seams are clearly visible. The above arguments can also be verified by the results obtained in the accuracy assessment.

In 'Case 2' (Figure 33) the dataset taken is from the Terrestrial laser scanner, and the coordinate system of point cloud was in the local coordinate system. The program is capable of processing point clouds in any coordinate system, and along any 'plane', both positive and negative X-Y, Y-Z, Z-X plane. This can be seen in this case where the façade is along Y-Z plane and no rotation is required to bring it to 'X-Y plane' which is the processing plane in airborne datasets. For processing this dataset appropriate surface threshold, tile size, and profile width were used. In this case, the surface object segment is properly detected and same with the 'projections' coming out from the façade (Figure 36).

In 'Case 3' (Figure 37) the dataset was created using SFM technique, where pictures of a wall were taken and point clouds were generated. This dataset contains noise in it. And we can notice that the surface is not properly extracted (Figure 38), the main reason for this is the noisy dataset. Again the appropriate 'main plane' (Y-Z plane) is selected along which the surface objects will be detected.

The program can handle any point cloud dataset from any source, irrespective of the coordinate system of the point clouds being local or global. Also, customisable option to select the main plane and the logic of the algorithm being unaffected by translation or scaling operation makes the program capable of processing datasets without the need to drastically rotate, scale, or translate.

In case of the house (Figure 41) where the significance of tile size and profile width is observed. In Figure 42 we can see the tile size taken larger than the extent of the roof of the house which in this particular scene is the largest non-surface object in the scene. This is crucial while selecting the tile size to ensure that there is not a single tile where all the points are actually non-surface points.

In Figure 43 where tile size is taken smaller than the extent of the roof, hence there came an iteration during tiling where a tile had all the points which lie on the roof, thus as per the nature of the logic, in the virtual profiles of this tile, surface object segment created were incorrect. This resulted in oblique like patterns (corresponding to the angle of 3rd orientation), seen throughout the scene, as the best of three is the criteria for deciding when processed using three orientations.

As for the 'profile width', we can observe in Figure 44, that selecting a broader profile width made the surface object segments more generalised in nature. Some non-surface points became a part of the surface object.

The optimum width is approximately the average spacing between the points, that is the average distance between two points in that scene. As 'profile width' finer than that may result in profiles, where along the length of the virtual profile from end to end point are not evenly available. And this will again produce incorrect surface objects.

It was found that the algorithm is sensitive to 'noise', which was also observed when processing 'case 3' and other datasets. The point density does not affect the competence of the logic used (seen in, Figure 51). But if the points are not uniformly distributed, the accuracy drops. This is due to the function based weighted mean. For example, if we consider the dataset shown in Figure 41, here if points were not uniformly distributed, but heavily cluttered on the roof, then this would have resulted in shifting the mean, thus reducing the accuracy.

In example 1 (Figure 45) we can understand the significance of surface/ground threshold. This is the output when no surface threshold was used. As we can see here that there is little variability in this dataset. Only the central region of the façade has 'non-surface' points.

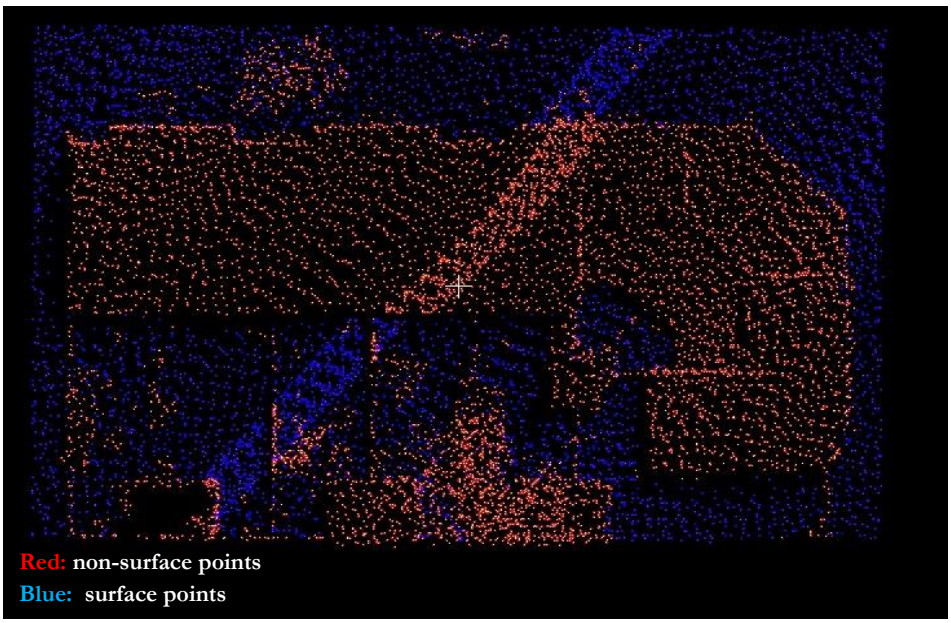


Figure 51: An example, where the point density was intentionally reduced

In this case, there will be cases where a profile is only made of surface points, then within those surface points, logic will further separate them (as seen in, Figure 45), which should not happen. To prevent this, this surface threshold is used. After defining it, the program will skip the profiles where the difference between the highest and lowest point is lower than the defined threshold and then all those points in that profile are considered as the part of the surface object. In Figure 52, we can see an example of such a virtual profile where all the points actually belong to the 'surface', and cases like these, are handled with the help of surface threshold.

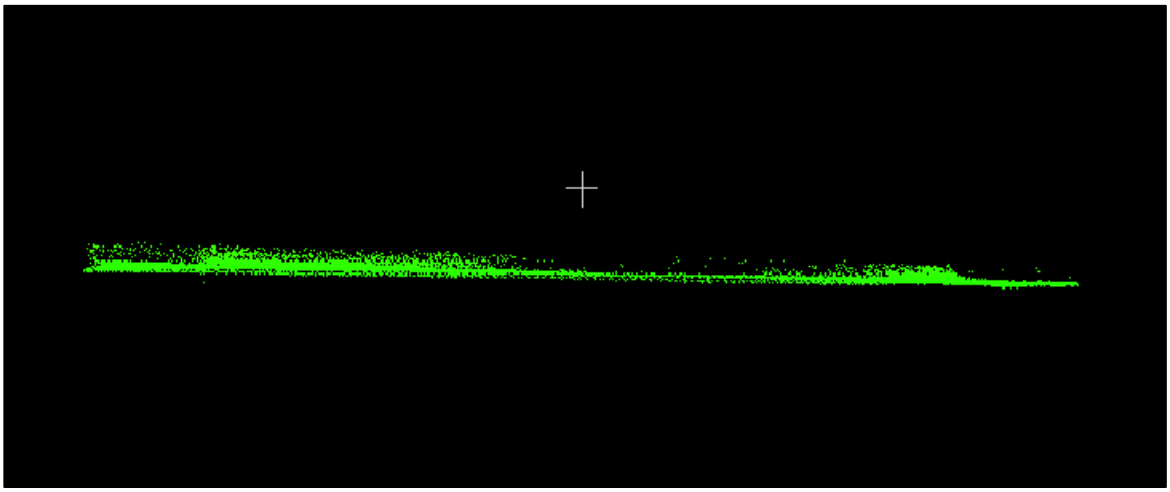


Figure 52: Side view of a profile, with only surface points

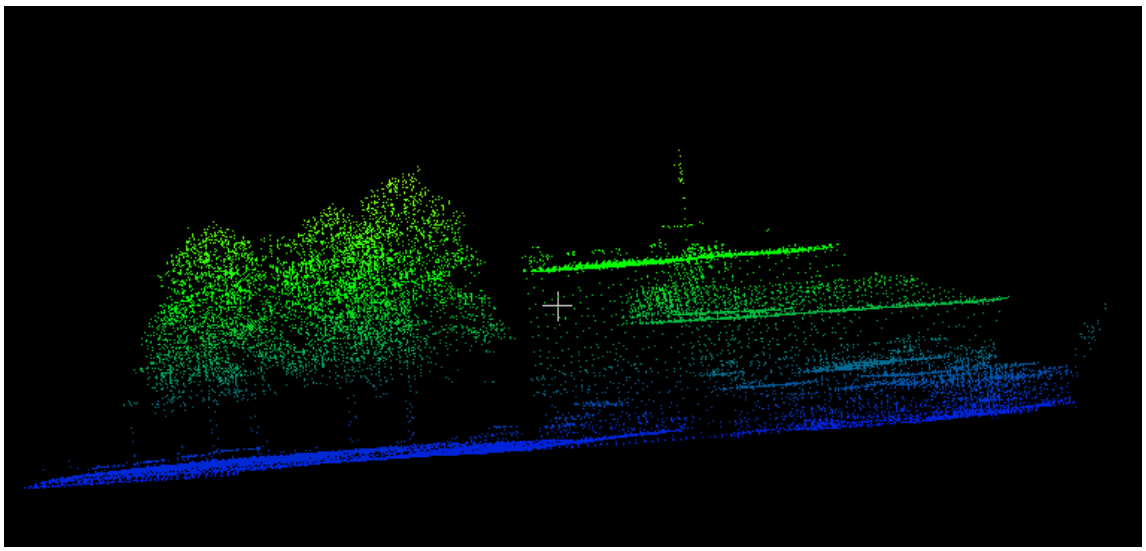


Figure 53: Use case where single orientation with user defined angle is useful

In Figure 53, we can see the use case of single orientation where the terrain is slightly tilted. In these cases, if the orientation of profiles is considered across the tilt of the plane then better results can be obtained, rather than using three orientation where accuracy will drop. As this tilt will result in shifting the function based weighted mean for the profiles which are along the tilt of the plane. But this effect will be least, across the tilt/incline of the terrain, hence making the option of single orientation with user defined angle a useful feature. These inferences are also applicable for datasets from other sources., TLS, SFM etc.

In Figure 46, we can see the use case of, the option of giving weightage to a particular orientation and of 'single orientation' option. Where if the feature of interest which needs to be extracted, is prominent in a particular direction then, more weightage can be given to the result of the orientation of virtual profiles which are across that direction. This functionality might also be useful when filtering out some features from a curved wall.

In Figure 54, we can see an example where we use the 'inversion of logic' feature, which will enable the program to separate out the top surface, without the need to rotate the scene.

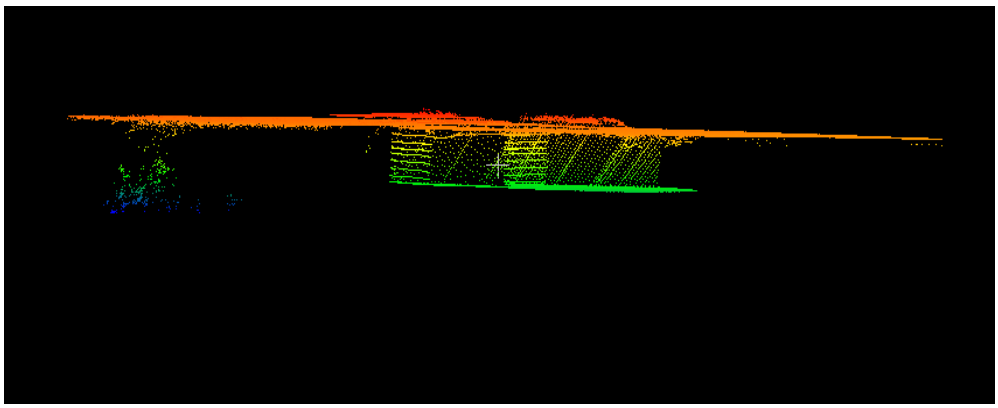


Figure 54: Use case for, 'inversion of logic' option

4.2.2. Discussion of code optimisation and parallel implementation

Code optimisations, are important and contributes to the overall robustness and performance of the program. This can be justified using Figure 55, where the improvement in performance is shown. Where containers are pre-initialised as per the requirement thus, helping in speeding up the task, as well as maintaining an optimum memory usage (code snippet shown in, APPENDIX C).

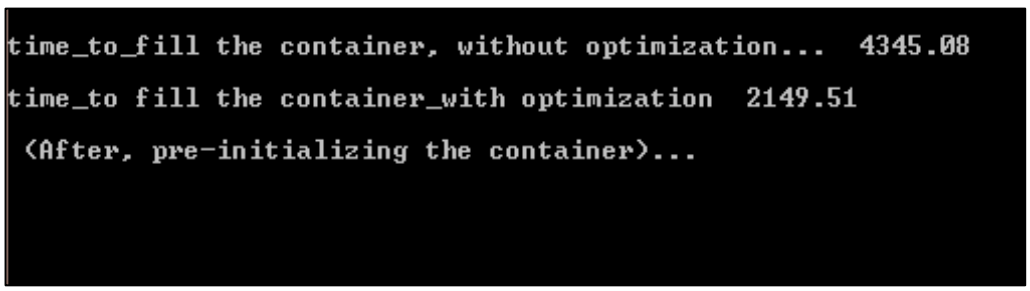


Figure 55: Significant Improvement in processing time, due to one of the optimisation measures

In the inter-comparison report (Section 4.1.2), implementation effort is discussed in detail, and in Figure 47 the comparison of 'total execution time' for 'serial' and 'parallel' run is shown, where significant improvement in 'processing time' is visible.

In Figure 48, we can see the effect of 'no. of thread' on the speedup. Speedup in processing time is maxed out at 4 threads with 1.8x. If we further increase the number of threads then the speedup reduces. This is due to the Thread generation overhead, and as the hardware used has only two physical cores (and four logical cores), thus the maximum speedup is around twice, that too achieved at four threads, two running at each core and using the hyper-threading technology to effectively have four logical cores. Beyond four threads, the cost of creating a thread keeps on increasing without any net speedup achieved thus the reduction in speedup is observed. Parallel efficiency is never '1', due to factors like communication overheads, thread management cost etc. Thus we can't see a true 'twice speedup' improvement here, and the term 'parallel efficiency' is the speedup per no. of cores.

In Figure 49, we can observe the speedup achieved from different parallel implementations. Here, MIMD and SIMD stands for multiple instruction and multiple data, single instruction and multiple data respectively. And they are different types of parallel architectures. An easy to understand analogy would be, in MIMD we have different workers working on a different chunk of a work in an orderly or random manner, while in SIMD we have one instruction decoder through which all the parts of work is passed in an orderly or random manner. SIMD is also called vectorization and is faster and simpler to design, but MIDM can solve complex problems unlike, SIMD. Slightly better results were achieved with SIMD than MIMD. And for GPU, around 6 times speedup was achieved and 'async' directive was used to copy results back to the CPU memory in an 'asynchronous' manner. Also considering the transfer overhead, GPU implementation is not efficient in terms of "total time", when processing very small datasets.

Another important factor is optimised in both the parallel implementations is load balancing of threads. For example, suppose there are total 60 profiles and 10 of them are assigned to each thread (in order), where total no. of threads is '6'. And it might be possible that a thread finished its job and is idle, as all these profiles will never result in same load (proportional to, no. of points) so this static scheduling of load distribution is not efficient. Thus dynamic scheduling of load distribution is used. so in the example discussed above, where we had '6' threads, as soon as a thread finishes the processing of a profile another profile is assigned to it, that too in random order. This further increases the efficiency as the order of processing the profiles in not important. Lastly, linearity is used to judge the scalability of the program, the Trendline shows that the processing time vs size curve is almost linear (Figure 50). Few concepts related to parallelisation are also shown in, APPENDIX B.

4.2.3. Discussion of accuracy assessment

Point to point accuracy is checked, so both, 'no. of points' as well as the 'location' of a point are accommodated. The accuracy of the proposed logic (Table 2), is consistent and better for 'urban areas' with low changes in elevation (gentle slopes). It can effectively remove low vegetation from the mountainous region but fails at mountainous slopes (steep slopes) thus resulting in large 'Type II error' but low 'Type I error'.

According to the result of accuracy assessment shown in Table 2 and its output which are shown in, APPENDIX A, we can also conclude that the proposed logic can effectively remove any 'low vegetation' from terrains without any sudden elevation changes. This use case occurs mostly at the time of post-processing when there is a need to further refine an already filtered point clouds by removing any 'low vegetation' present in it.

When compared to other algorithms it's not versatile in terms of terrain types, hence we can see a larger average total error (16.24) and standard deviation (7.02) for the proposed logic. In Table 4 and Table 5 (shown in APPENDIX A) taken from (W. Zhang et al., 2016) if we look at the performance of other algorithms in terms of 'total error', for individual samples rather than the overall average 'total error' and its standard deviation. It can be seen that the proposed algorithm outperforms some algorithms in specific terrains. Showing the potential of the proposed logic.

The logic is elevation based and if merged with 'slope' or some other criteria, the accuracy can be improved. Which is possible as the program can be wrapped around some other logic or clubbed with some other logic while retaining all the unique features, flexibility, code optimisation and parallel implementation benefits. If the whole process is repeated (multiple iterations), where the output of the first iteration is used as an input of the 'second' iteration, while defining appropriate parameters at each iteration, results can be further improved.

5. CONCLUSIONS AND RECOMMENDATIONS

5.1. Conclusion

The proposed approach performs well for the urban regions without any sudden elevation changes. It also has potential be used as a post-processing step to remove 'low vegetation'. The average 'total error' came to be '16.24' with a standard deviation of '7.02'. It is not as versatile as other algorithms in terms of terrain type. But it has potential, as a better accuracy was achieved for few terrain as compared with some other algorithms. If this logic is integrated with other logics, or multiple iterations are performed, then results can be further improved. The flexibility or customisable nature of the algorithm, as well as the unique features, came to be useful while processing different datasets. The algorithm can process any dataset irrespective of its coordinate system. It can process along any standard plane, from positive X-Y, Y-Z, Z-X to negative.

The program takes two mandatory input parameters, which are tile size and profile width, where the tile size should be more than the size of the largest non-surface object (buildings in airborne cases), this will result in a lower error and will reduce the Type I error. The tile size can also be used to processes data in a stream, enabling processing of large datasets even on low-end systems. The profile width is optimum around the average point spacing of the scene. Other optional parameters like surface/ground threshold, the angle of third orientation etc can be used to reduce the error due to profiles with surface points only, the optimum surface threshold depends on the type of surface and the scale of points. Rest of the customisation options depends on the specific dataset or on user's need.

Code optimisations like a different method, memory management, container type etc can make a huge difference when processing large datasets. The program is scalable in terms of 'processing time' and the use of open source parallel processing APIs, as well as the structure of the program, makes it easy to compile, modify or understand the code.

The results of parallel implementations are convincing along with the many optimisation measures, resulting in a robust processing framework. The speedup achieved were 1.9x and 6x for multicore and GPU respectively, a detailed report was generated for parallel implementation based on parameters such as speedup, implementation effort, lines of code, parallel efficiency, and linearity. When improving the algorithm in future, the addition of some other logic or a combination of two or more is possible. As the program is not hard coded for a single logic and was written keeping the expansion of proposed logic, in mind. All the benefits of this work like memory management, optimisations, flexibility and parallel implementation will be inherited when this program is wrapped around another logic.

5.2. Answers to research questions

1) What is the performance of the proposed approach of segmentation based filtering, where multiple orientations of virtual profiles are considered, when used on different terrain types?

Ans. The answer to the above question is addressed in, Section 4.1.3 and 4.2.3.

2) How can we optimise the processing speed and memory handling in the involved steps of DEM generation process?

Ans. The optimisation starts at code level with choosing the right method, compiler version, careful allocation and deallocation, and efficient memory management. Further, in-depth discussion can be found in Section 3.3.2.

3) How to increase the usability of the algorithm and make the extraction of 'surface' more generic, so it can also be used for other use cases too, apart from ground point filtering?

Ans. The answer to this research question is answered in, Section 3.3.2. This is also illustrated throughout the considered examples shown in Section 4.1.1, where multiple datasets are used from different sources viz., TLS, SFM airborne LiDAR.

4) Which is the better High performance computing solution for the algorithm's parallel implementation?

Ans. This depends on the 'implementation effort' one can put in, along with the size of the dataset. Though GPU implementation has better speedup, thus it is a more viable option. But one has to keep in mind about the multiple factors involved like 'compiler', language to be used, code level optimations, etc. An in-depth inter-comparison report, which answers this more precisely is given in Section 4.1.2, which is later discussed in Section 4.2.2.

5.3. Recommendations

The logic used is capable of separating a single bottom/top plane from a point cloud, but work can be extended to separate multiple planes. Further improvement in the proposed elevation based logic can be achieved, by merging it with other logics like window based or slope based, to produce better results. Other recommendations are:

- Distributed memory model can be tested. An integration of OpenMP and MPI is a viable option, in that direction. Apache hadoop™ and Spark™ should also be tested to further include them in the inter-comparison of parallel implementations.

- Distributed file system can be explored to make up for the I/O overhead.

- Multiple file format support can be added apart from the native 'ASCII' support. To increase the interoperability of the tool. This will remove the additional task of converting the .las, .pcd, .ply or some other format to ASCII file.

LIST OF REFERENCES

Boehm, J., Liu, K., & Alis, C. (2016). Sideloading – Ingestion of large point clouds into the Apache Spark Big data engine. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLI-B2*, 343–348. http://doi.org/10.5194/isprs-archives-XLI-B2-343-2016

Chen, Q. (2007). Airborne LiDAR data processing and information extraction. *Photogrammetric Engineering & Remote Sensing*, *73*, 109–112.

Chen, Q., Wang, H., Zhang, H., Sun, M., & Liu, X. (2016). A point cloud filtering approach to generating DTMs for steep mountainous areas and adjacent residential areas. *Remote Sensing*, *8*(1), 71. http://doi.org/10.3390/rs8010071

Chu, H., Wang, C., Huang, M., Lee, C.-C., Liu, C.-Y., & Lin, C.-C. (2014). Effect of point density and interpolation of LiDAR-derived high-resolution DEMs on landscape scarp identification. *GIScience & Remote Sensing*, *51*(6), 731–747. http://doi.org/10.1080/15481603.2014.980086

Danner, A., Breslow, A., Baskin, J., & Wilikofsky, D. (2012). Hybrid MPI/GPU interpolation for grid DEM construction. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems - SIGSPATIAL '12* (p. 299). New York, New York, USA: ACM Press. http://doi.org/10.1145/2424321.2424360

Delgado, J., Martin, G., Plaza, J., Jimenez, L. I., & Plaza, A. (2016). Fast Spatial Preprocessing for Spectral Unmixing of Hyperspectral Data on Graphics Processing Units. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, *9*(2), 952–961. http://doi.org/10.1109/JSTARS.2015.2495128

Duffy, L. (2015). Geiger-Mode: The Latest and Greatest in LiDAR. Retrieved August 15, 2016, from http://www.pobonline.com/articles/100273-geiger-mode-the-latest-and-greatest-in-lidar

Han, S. H., Heo, J., Sohn, H. G., & Yu, K. (2009a). Parallel Processing Method for Airborne Laser Scanning Data Using a PC Cluster and a Virtual Grid. *Sensors*, *9*, 2555–2573. http://doi.org/10.3390/s90402555

Han, S. H., Heo, J., Sohn, H. G., & Yu, K. (2009b). Parallel Processing Method for Airborne Laser Scanning Data Using a PC Cluster and a Virtual Grid. *Sensors*, *9*(4), 2555–2573. http://doi.org/10.3390/s90402555

Han, S. H., Lee, J. H., & Yu, K. Y. (2007). An approach for segmentation of airborne laser point clouds utilizing scan-line characteristics. *ETRI Journal*, *29*(5), 641–648. http://doi.org/10.4218/etrij.07.0106.0316

Isenburg, M., Liu, Y., Shewchuk, J., & Snoeyink, J. (2006). Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics*, *25*(3), 1049. http://doi.org/10.1145/1141911.1141992

Jian, X., Xiao, X., Chengfang, H., Zhizhong, Z., Zhaohui, W., & Dengzhong, Z. (2015). A hadoop-based algorithm of generating DEM grid from point cloud data. *International Archives of the Photogrammetry,*

Remote Sensing and Spatial Information Sciences - ISPRS Archives, *40*(7W3), 1209–1214. http://doi.org/10.5194/isprsarchives-XL-7-W3-1209-2015

Kang, X., Liu, J., & Lin, X. (2014). Streaming Progressive TIN Densification Filter for Airborne LiDAR Point Clouds Using Multi-Core Architectures. *Remote Sensing*, *6*(8), 7212–7232. http://doi.org/10.3390/rs6087212

Krishnan, S., Baru, C., & Crosby, C. (2010). Evaluation of MapReduce for gridding LIDAR data. In *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010* (pp. 33–40). IEEE. http://doi.org/10.1109/CloudCom.2010.34

Li, Z., Zhu, C., & Gold, C. (2004). *Digital Terrain Modeling. CRC Press*. Baton Rouge: CRC Press. http://doi.org/10.1201/9780203357132

Lin, X., & Zhang, J. (2014). Segmentation-based filtering of airborne LiDAR point clouds by progressive densification of terrain segments. *Remote Sensing*, *6*(2), 1294–1326. http://doi.org/10.3390/rs6021294

Melzer, T. (2007). Non-parametric segmentation of ALS point clouds using mean shift. *Journal of Applied Geodesy*, *1*(3), 159–170. http://doi.org/10.1515/jag.2007.018

Mineter, M. J. (2003). A software framework to create vector-topology in parallel GIS operations. *International Journal of Geographical Information Science*, *17*(3), 203–222. http://doi.org/10.1080/13658810210149443

Navarro, C. A., Hitschfeld-Kahler, N., & Mateu, L. (2014). A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. *Communications in Computational Physics*, *15*(02), 285–329. http://doi.org/10.4208/cicp.110113.010813a

Pingel, T. J., Clarke, K. C., & McBride, W. A. (2013). An improved simple morphological filter for the terrain classification of airborne LIDAR data. *ISPRS Journal of Photogrammetry and Remote Sensing*, *77*, 21–30. http://doi.org/10.1016/j.isprsjprs.2012.12.002

Plaza, A. J., & Chang, C.-I. (2007). *High Performance Computing in Remote Sensing*. Philadelphia, PA: CRC Press.

Rabbani, T., van den Heuvel, F. a, & Vosselman, G. (2006). Segmentation of point clouds using smoothness constraint. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences - Commission V Symposium "Image Engineering and Vision Metrology,"* *36*(5), 248–253. Retrieved from http://www.isprs.org/proceedings/XXXVI/part5/paper/RABB_639.pdf

Rodríguez-Caballero, E., Afana, A., Chamizo, S., Solé-Benet, A., & Canton, Y. (2016). A new adaptive method to filter terrestrial laser scanner point clouds using morphological filters and spectral information to conserve surface micro-topography. *ISPRS Journal of Photogrammetry and Remote Sensing*, *117*, 141–148. http://doi.org/10.1016/j.isprsjprs.2016.04.004

Ross, P. (2008). Why CPU Frequency Stalled. *IEEE Spectrum*, *45*(4), 72–72. http://doi.org/10.1109/MSPEC.2008.4476447

Sene, K. (2008). *Flood warning, forecasting and emergency response. Flood Warning, Forecasting and Emergency Response*. Berlin, Heidelberg: Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-540-77853-0

Sithole, G., & Vosselman, G. (2004). Experimental comparison of filter algorithms for bare-Earth extraction from airborne laser scanning point clouds. *ISPRS Journal of Photogrammetry and Remote Sensing*, *59*(1-2), 85–101. http://doi.org/10.1016/j.isprsjprs.2004.05.004

Sithole, G., & Vosselman, G. (2005). Filtering of airborne laser scanner data based on segmented point clouds. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, *36*(3/W3-4), 66–71. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.1111&rep=rep1&type=pdf

Uysal, M., Toprak, A. S., & Polat, N. (2015). DEM generation with UAV Photogrammetry and accuracy analysis in Sahitler hill. *Measurement*, *73*, 539–543. http://doi.org/10.1016/j.measurement.2015.06.010

van der Sande, C., Soudarissanane, S., & Khoshelham, K. (2010). Assessment of relative accuracy of AHN-2 laser scanning data using planar features. *Sensors*, *10*(9), 8198–8214. http://doi.org/10.3390/s100908198

Vosselman, G. (2000). Slope based filtering of laser altimetry data. *International Archives of Photogrammetry and Remote Sensing*, *33*(B3/2; PART 3), 935–942.

Whitworth, M. (2015). Laser surveys light up open data | Creating a better place. Retrieved August 20, 2016, from https://environmentagency.blog.gov.uk/2015/09/18/laser-surveys-light-up-open-data/

Xiangyun Hu, Xiaokai Li, & Yongjun Zhang. (2013). Fast Filtering of LiDAR Point Cloud in Urban Areas Based on Scan Line Segmentation and GPU Acceleration. *IEEE Geoscience and Remote Sensing Letters*, *10*(2), 308–312. http://doi.org/10.1109/LGRS.2012.2205130

Yang, Z., Zhu, Y., & Pu, Y. (2008). Parallel image processing based on CUDA. In *International Conference on Computer Science and Software Engineering* (pp. 198–201). IEEE. http://doi.org/10.1109/CSSE.2008.1448

Zhang, K., Chen, S. C., Whitman, D., Shyu, M. L., Yan, J., & Zhang, C. (2003). A progressive morphological filter for removing nonground measurements from airborne LIDAR data. *IEEE Transactions on Geoscience and Remote Sensing*, *41*(4 PART I), 872–882. http://doi.org/10.1109/TGRS.2003.810682

Zhang, W., Qi, J., Wan, P., Wang, H., Xie, D., Wang, X., & Yan, G. (2016). An Easy-to-Use Airborne LiDAR Data Filtering Method Based on Cloth Simulation. *Remote Sensing*, *8*(6), 501. http://doi.org/10.3390/rs8060501

APPENDIX A

sample Id	parameters	total points	misclassified surface pt	misclassified non surface pt	actual no. of surf pt	actual no. of non surf pt
samp11	tile - 40 , profile - 1, s.thresh- 5, angle - 45	38010	8615	2363	21786	16224
samp12	tile - 40 , profile - 1, s.thresh- 2, angle - 45	52119	5827	2101	26691	25428
samp21	tile - 35 , profile - 1, s.thresh- 2, angle - 45	12960	1053	365	10085	2875
samp22	tile - 50 , profile - 1, s.thresh- 3, angle - 45	32706	5042	1603	22504	10202
samp23	tile - 45 , profile - 1, s.thresh- 2, angle - 45	25095	4558	978	13223	11872
samp24	tile - 45 , profile - 1, s.thresh- 2, angle - 45	7492	1366	132	5434	2058
samp31	tile - 45 , profile - 1, s.thresh- 2, angle - 45	28862	1396	1005	15556	13306
samp41	tile - 40 , profile - 1, s.thresh- 2, angle - 45	11231	1443	1497	5602	5629
samp42	tile - 40 , profile - 1, s.thresh- 2, angle - 45	42470	679	4631	12443	30027
samp51	tile - 45 , profile - 1, s.thresh- 4, angle - 45	17845	1998	1070	13950	3895
samp52	tile - 10 , profile - 1, s.thresh- 2, angle - 45	22474	2907	1628	20112	2362
samp53	tile - 10 , profile - 1, s.thresh- 2, angle - 45	34378	1607	907	32989	1389
samp54	tile - 40 , profile - 1, s.thresh- 2, angle - 45	8608	431	681	3983	4625
samp61	tile - 10 , profile - 1, s.thresh- 2, angle - 45	35060	717	638	33854	1206
samp71	tile - 40 , profile - 1, s.thresh- 2, angle - 45	15645	2434	358	13875	1770

Table 3: Accuracy assessment – details and parameters for all samples

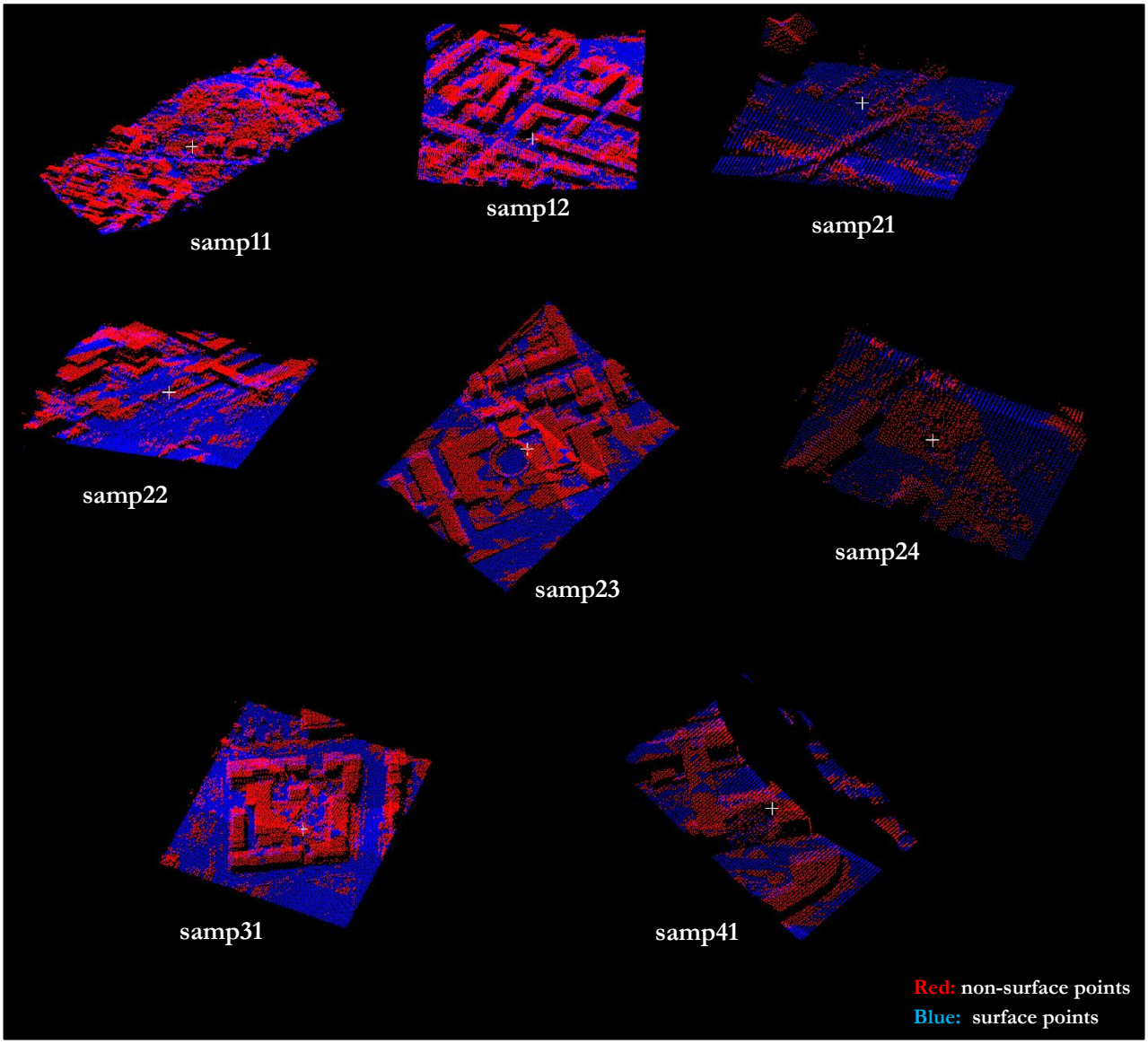


Figure 56: Outputs of ISPRS test samples used in accuracy assessment - I

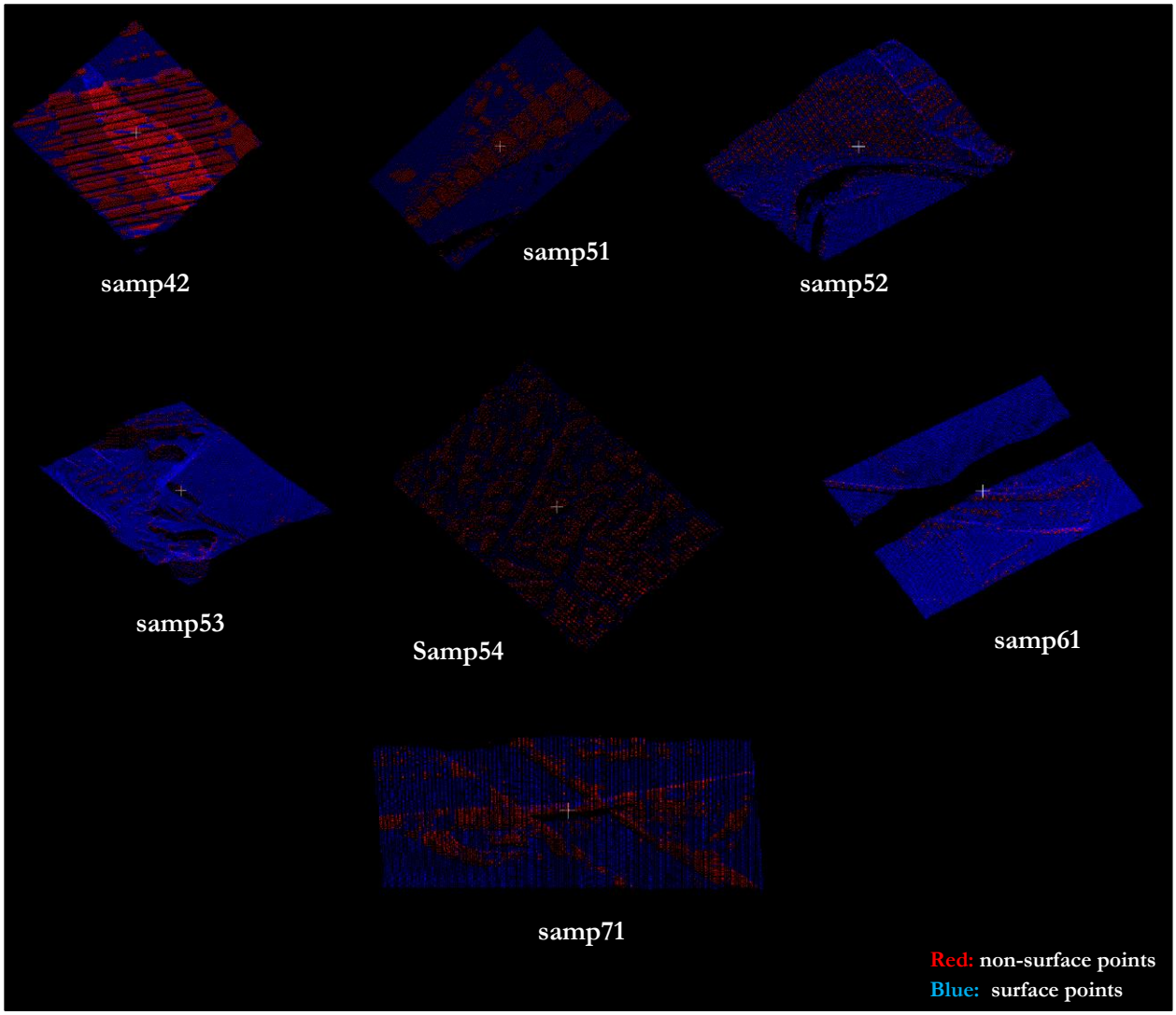


Figure 57: Outputs of ISPRS test samples used in accuracy assessment - II

Samples	Axelsson (1999)	Elmqvist (2000)	Pfeifer (2001)	Mongus (2012)	Li (2013)
samp11	10.76	22.4	17.35	11.01	12.85
samp12	3.25	8.18	4.5	5.17	3.74
samp21	4.25	8.53	2.57	1.98	2.55
samp22	3.63	8.93	6.71	6.56	4.06
samp23	4.00	12.28	8.22	5.83	6.16
samp24	4.42	13.83	8.64	7.98	5.67
samp31	4.78	5.34	1.8	3.34	2.47
samp41	13.91	8.76	10.75	3.71	6.71
samp42	1.62	3.68	2.64	5.72	3.06
samp51	2.72	21.31	3.71	2.59	3.92
samp52	3.07	57.95	19.64	7.11	15.43
samp53	8.91	48.45	12.6	8.52	11.71
samp54	3.23	21.26	5.47	6.73	3.93
samp61	2.08	35.87	6.91	4.85	5.81
samp71	1.63	34.22	8.85	3.14	4.58
Avg.	4.82	20.73	8.02	5.62	6.18
Std.	3.44	15.92	5.09	2.39	3.84

Table 4: Result (I) of accuracy assessment of other algorithms using ISPRS filter test samples, here 'total error' per sample is shown along with average 'total error' and its standard deviation (W. Zhang et al., 2016)

Samples	Chen (2013)	Pingel (2013)	Zhang (2013)	Hu (2014)	Hui (2016)	W. Zhang (2016)
samp11	13.01	8.28	18.4	8.31	13.34	12.01
samp12	3.38	2.92	5.92	2.58	3.5	2.97
samp21	1.34	1.1	4.95	0.95	2.21	3.42
samp22	4.67	3.35	14.1	3.23	5.41	8.94
samp23	5.24	4.61	12.0	4.42	5.11	4.79
samp24	6.29	3.52	20.2	3.80	7.47	2.87
samp31	1.11	0.91	2.32	0.90	1.33	1.61
samp41	5.58	5.91	20.4	5.91	10.6	5.14
samp42	1.72	1.48	3.94	0.73	1.92	1.58
samp51	1.64	1.43	5.31	2.04	4.88	3.08
samp52	4.18	3.82	12.9	2.52	6.56	3.93
samp53	7.29	2.43	5.58	2.74	7.47	5.2
samp54	3.09	2.27	6.4	2.35	4.16	3.18
samp61	1.81	0.86	16.1	0.84	2.33	1.49
samp71	1.33	1.65	10.4	1.50	3.73	5.71
Avg.	4.11	2.97	10.6	2.85	5.33	4.39
Std.	3.06	2.00	6.01	2.03	3.23	2.76

Table 5: Result (II) of accuracy assessment of other algorithms using ISPRS filter test samples, here 'total error' per sample is shown along with average 'total error' and its standard deviation (W. Zhang et al., 2016)

APPENDIX B

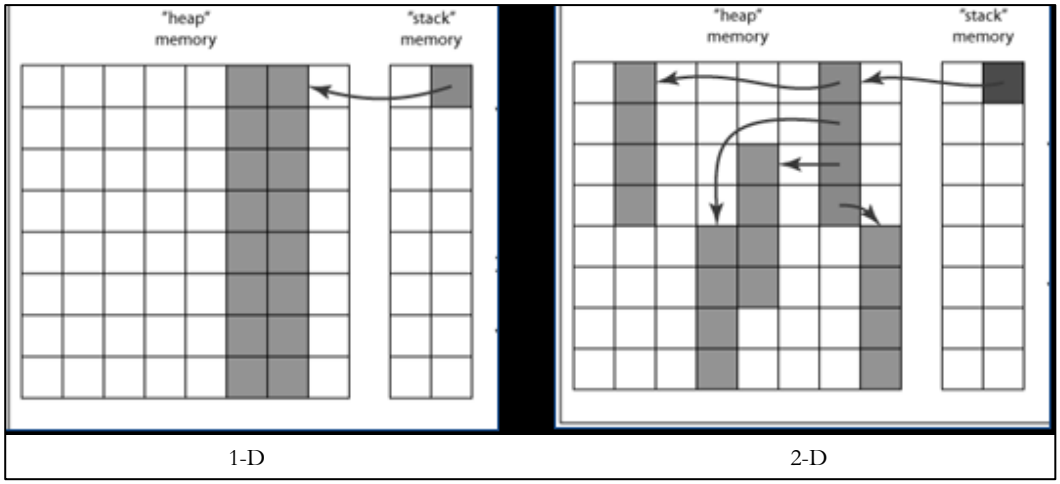


Figure 58: Locality in cache - for 1-D container vs 2-D container

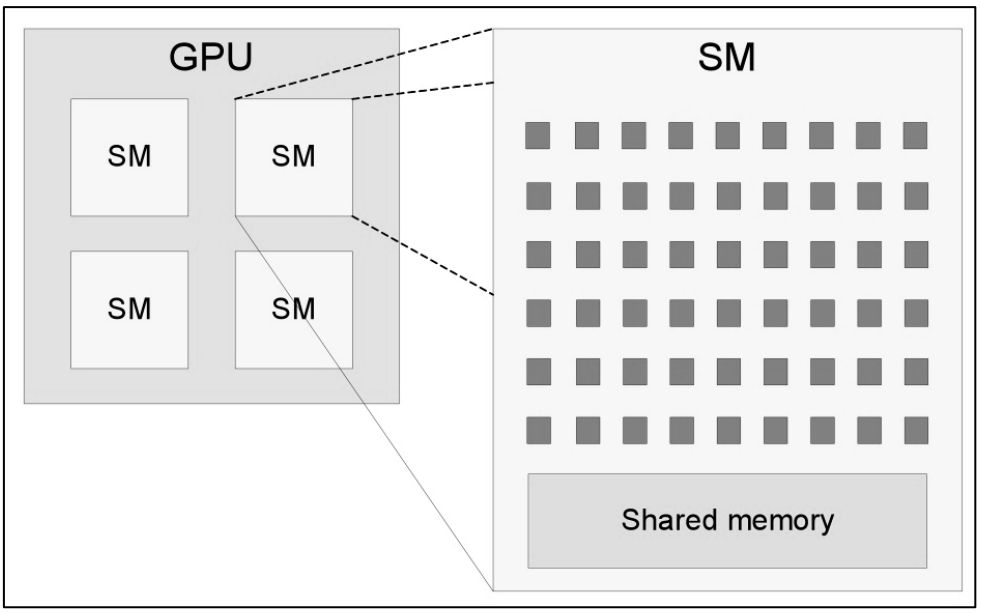


Figure 59: GPUs are made of multiple SMs – streaming multiprocessors

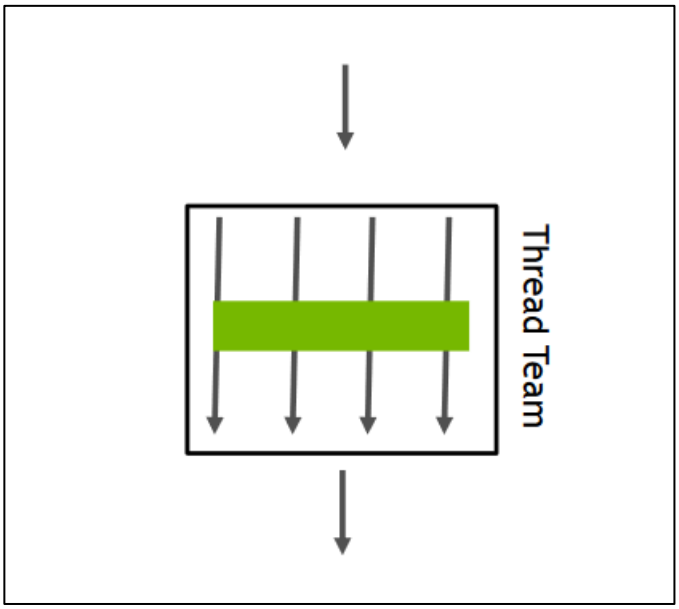


Figure 60: Distribution of work in CPU per parallel region

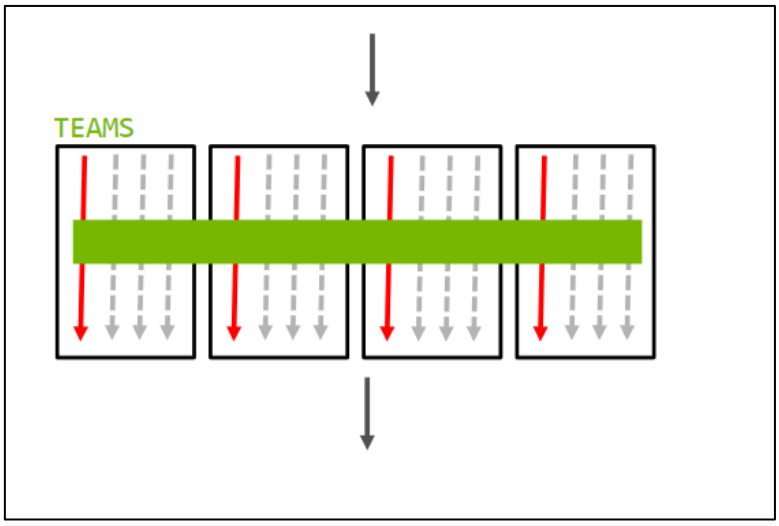


Figure 61: Distribution of work in GPU / coprocessor per parallel region

APPENDIX C

The portion of code (code snippet) below, shows the approximate estimation of 'safest size criteria'. Thus an appropriate size is determined, which is later used for pre-initialising the containers.

```
int optimum_single_vector_lenght=0;
int sz = 0, a = 34739242;        // elements in approx. 128 MB float  vector.

        for (sz = 1;  ; sz++) {
                try {
                        vector<float> v;
                        v.reserve(a*sz);
                        cout<<"attempting for... "<<sz*128<<" MB float vector"<<endl;
                        v.clear();
                }
                catch (bad_alloc) {
                cout << "max possible size is (approx.) -->  " << sz * 128 << " MB" << endl;
                optimum_single_vector_lenght = ((sz - 3)*a);  //3 iteration before, just to be safe.
                break;
                }
                catch (length_error) {
                cout << "max possible size is (approx.) -->  " << sz * 128 << " MB" << endl;
                optimum_single_vector_lenght = ((sz - 3)*a);
                break;
                }
        }
```

Figure 62: Code snippet, showing how the 'safest container size criteria' is determined

Below is the portion of code, which was used to illustrate how, 'pre-initialising' a container before filling it, speeds up the task.

```
float time_w_opt =0, time_wout_opt =0;

        vector<float> v;   // pre-initialising this vector  to a required size
        vector<float> x;
         int a = 60473923; // no. of elements to be filled
         v.reserve(a);

auto t_start = std::chrono::high_resolution_clock::now();
              for (int sz = 1;sz<a ; sz++) {
                    v.push_back(3);
                    }
auto t_end = std::chrono::high_resolution_clock::now();
time_w_opt =  std::chrono::duration<double, std::milli>(t_end-t_start).count();
auto t_start1 = std::chrono::high_resolution_clock::now();
              for (int sz = 1;sz<a ; sz++) {
                 x.push_back(3);
                 }
auto t_end1 = std::chrono::high_resolution_clock::now();
time_wout_opt =  std::chrono::duration<double, std::milli>(t_end1-t_start1).count();

cout << endl << "time_to_fill the container, without optimization...  " << time_wout_opt<<endl;
cout << endl << "time_to fill the container with optimization  " << time_w_opt <<endl;
cout << endl << " ( that is, after, pre-initialising the container)...  " <<endl;
```

Figure 63: Code snippet, used for showing how 'pre-initialising' a container, speeds up the task

Below are some syntax of, the 'compiler directives' of OpenMP and OpenACC.

```
// syntax of  openMP compiler directives

#pragma omp parallel for simd private(x_scn,y_scn,z_scn,id_in_tile,int_temp_3,scline_vec_lenght)
        For(  ;  ;  ) {
                ----parallel region----
                ----parallel region----
                ----parallel region----
                }

// syntax of  openACC compiler directives

#pragma acc data copy (tag_1), create (x_scn)
#pragma acc kernels loop
        For(  ;  ;  ) {
                ----parallel region----
                ----parallel region----
                ----parallel region----
                }
```

Figure 64: Syntax of compiler directives of OpenMP and OpenACC

Illustration of commands, flags and dependencies required, for building the GCC offload compiler.

(Ubuntu 16.10)

- **Version of Nvidia driver and linux header should be compatible with each other.**
- **Check the current linux header and install a compatible version.**

To list the currently installed headers:

```
dpkg --list | grep linux-image
dpkg --list | grep linux-header
```

After installation of CUDA toolkit, paths should be added:

```
vi ~/.bashrc
export CUDA_HOME=/usr/local/cuda-8.0
export LD_LIBRARY_PATH=${CUDA_HOME}/lib64
PATH=${CUDA_HOME}/bin:${PATH}
PATH=${CUDA_HOME}/bin/nvcc:${PATH}
export PATH
```

For nvptx tools:

cd /home/invoker/test/build/nvptx-tools

With these flags:

/home/invoker/test/src/nvptx-tools/configure --prefix=/home/invoker/test/install --target=nvptx-none --with-cuda-driver-include=/usr/local/cuda-8.0/include --with-cuda-driver-lib=/usr/local/cuda-8.0/lib64 --with-cuda-runtime-include=/usr/local/cuda-8.0/include --with-cuda-runtime-lib=/usr/local/cuda-8.0/lib64 CC="gcc -m64" CXX="g++ -m64"

make
make install

Figure 65: Dependencies, commands and flags needed for building GCC offload compiler from source- I

For offload compiler :

Dependencies: nvptx-tools and nvptx-newlib (downloadable from github).

Linking nvptx-newlib with gcc-newlib:

```
ln -vs /home/invoker/test/src/nvptx-newlib/newlib /home/invoker/test/src/gcc-6.2/newlib

ln -vs . /home/invoker/test/install/nvptx-none/usr

cd /home/invoker/test/build/ol-compiler

/home/invoker/test/src/gcc-6.2/configure --target=nvptx-none --prefix= --enable-languages=c,c++ --enable-checking=yes,df,fold,rtl --enable-as-accelerator-for=x86_64-pc-linux-gnu --with-sysroot=/home/invoker/test/install/nvptx-none --with-build-sysroot=/home/invoker/test/install/nvptx-none --with-build-time-tools=/home/invoker/test/install/nvptx-none/bin --disable-multilib --disable-sjlj-exceptions --enable-newlib-io-long-long CC="gcc -m64" CXX="g++ -m64"

make
make DESTDIR=/home/invoker/test/install install
```

For main compiler:

```
cd /home/invoker/test/build/core-compiler

/home/invoker/test/src/gcc-6.2/configure --prefix= --disable-bootstrap --enable-languages=c,c++ --disable-multilib --enable-offload-targets=nvptx-none=/home/invoker/test/install --with-cuda-driver-include=/usr/local/cuda-8.0/include --with-sysroot= CC="gcc -m64" CXX="g++ -m64"

make
make DESTDIR=/home/invoker/test/install install
```

After this add appropriate 'Path' and use 'update-alternatives' to switch between default compiler and this compiler.

Figure 66: Dependencies, commands and flags needed for building GCC offload compiler from source- II

APPENDIX D

Github link for code - https://github.com/vbhv14