Efficient Point Cloud Pre-processing using The Point Cloud Library

Marius Miknis *Marius.Miknis@southwales.ac.uk*
Faculty of Computing, Engineering and Science
University of South Wales
Pontypridd, CF37 1DL, UK

Ross Davies *Ross.Davies@southwales.ac.uk*
Faculty of Computing, Engineering and Science
University of South Wales
Pontypridd, CF37 1DL, UK

Peter Plassmann *Peter.Plassmann@southwales.ac.uk*
Faculty of Computing, Engineering and Science
University of South Wales
Pontypridd, CF37 1DL, UK

Andrew Ware *Andrew.Ware@southwales.ac.uk*
Faculty of Computing, Engineering and Science
University of South Wales
Pontypridd, CF37 1DL, UK

Abstract

Robotics, video games, environmental mapping and medical are some of the fields that use 3D data processing. In this paper we propose a novel optimization approach for the open source Point Cloud Library (PCL) that is frequently used for processing 3D data. Three main aspects of the PCL are discussed: point cloud creation from disparity of color image pairs; voxel grid downsample filtering to simplify point clouds; and passthrough filtering to adjust the size of the point cloud. Additionally, OpenGL shader based rendering is examined. An optimization technique based on CPU cycle measurement is proposed and applied in order to optimize those parts of the pre-processing chain where measured performance is slowest. Results show that with optimized modules the performance of the pre-processing chain has increased 69 fold.

Keywords: Point Cloud, Point Cloud Library, Point Data Pre-processing.

1. INTRODUCTION

Point clouds are sparse spatial representations of 3D object shapes. Algorithms such as the ones in the frequently used RANSAC [1] method can then be applied to reconstruct the complete object shapes from the point clouds.

A popular library for storing and manipulating point cloud data is the Point Cloud Library (PCL) [2]. The PCL is a large scale open source project that is focused on both 2D and 3D point clouds and includes some image processing functionality. Currently the Library has over 120 developers, from universities, commercial companies and research institutes. The PCL is released under the terms of the BSD license, which means that it is free for commercial and research use. It can be cross compiled for many different platforms including Windows, Linux, Mac OS, Android and iOS. This allows the library to also be used in embedded systems. The main algorithm groups in the PCL are for segmentation, registration, feature estimation, surface reconstruction, model fitting, visualization and filtering.

In the work presented in this paper stereo-photogrammetry is used as the main method of 3D data acquisition. This method is based on stereoscopy where two spatially separated images are obtained from different viewing positions [3]. The analysis of disparity (separation) between corresponding points in both images encodes the distance of object points which are then stored in a disparity map.

This paper is organized as follows: section 2 presents related work in the field of 3D data acquisition and point cloud processing, followed in section 3 by a description of PCL modules and their optimization, while conclusion and future work are discussed in sections 1 and 1.

2. RELATED WORK

There are many uses for 3D data ranging from environmental perception for robots via autonomous car navigation, playing video games to medical uses such as wound measurement, facial reconstruction and more. A number of ways to capture 3D data have been proposed and implemented. Many existing technologies rely heavily on the use of structured or infrared lighting to extract the depth data [4]. The technique of structured lighting is widely used in computer vision for its many benefits [5] in terms of accuracy and ease of use. Over the last 15 years 3D laser scanners have been developed [6] as active remote sensing devices. Such scanners can quickly scan thousands or even millions of 3D cloud points in a scene. Time of flight cameras are also widely used in computer vision. The principle behind these cameras is similar to that of a sonar, but with light replacing sound. Such cameras were introduced into the wider public domain by the Microsoft Xbox One console [7] to replace its older structured lighting based Kinect sensor.

Once 3D data has been acquired by the above systems some kind of processing needs to be applied to extract useful information as well as to remove noise, outliers or any unnecessary information. With the number of points that can be sampled point clouds can get extremely large and contain noise as well as outliers and errors. Thus the pre-processing stage is important [8] [9] as it deals with noise, error and outlier removal through the use of filters as well as smoothing the point cloud and reducing the point count while still keeping the relevant feature information. There are software tools available for such processing [10] [11] [12] but very few provide a complete library framework to incorporate into software projects. 3DReshaper [13] is such a library that provides point cloud processing capabilities. The PCL is the most commonly used library for point cloud processing, thus the PCL was used as the main development library in this research.

The current application focus of the PCL library is in the field of robotics. For robots to sense, compute and interact with objects or whole scenes a way to perceive the world is needed, which is why the PCL is used as a part of the Robot Operating System (ROS). Using the PCL as a part of ROS, robots can compute a 3D environment in order to understand it, detect objects and interact with them. Due to space and power restrictions such systems rarely use desktop-like computing devices and are therefore in most cases implemented on relatively small embedded systems. In these systems the universal nature of the PCL (many operating systems, many 3D data formats, etc.) results in slow performance. The following section III proposes a range of optimizations in order to improve performance.

3. POINT CLOUD PROCESSING OPTIMISATIONS

Four key algorithm areas were selected for optimization: point cloud creation (section 3.1), rendering (section 3.2), voxel grid down-sampling (section 3.3), pass through filtering (section 3.4) and the pre-processing chain (section 3.5). For the stereo test data the New Tsukuba Stereo Dataset [14] was used. This is a collection of synthetic stereo image pairs created using computer graphics. Additionally, the OpenCV (Open Source Computer Vision Library) was used for image loading. The project code was run on a desktop Intel i7 machine. The first set of tests used the Microsoft Visual Studio 2013 code analyzer for inspecting code and its performance statistics. The purpose of the tests was to identify which parts of the code are using the most of the CPU calls and then to optimize those.

3.1. Point Cloud Creation Speed Improvements

When using a stereo camera setup depth values are represented as a disparity map which in most cases is a greyscale image where the brightness of pixels represents depth values. A second output is a color image that stores information of the actual color value of the point. From the disparity and color images a point cloud can be produced. The PCL provides the *OrganisedConversion<>::convert()* method which uses the disparity map, color image and the focal length of the camera to produce a point cloud.

Point cloud generation is in 3 stages: first the input images are loaded into memory using OpenCV which converts them to vectors that can be passed as parameters to the second stage, PCL point cloud creation. The point cloud is then rendered on screen in the third stage. Using Microsoft Visual Studio 2013 code profiler CPU cycles were measured per line of code. In order to average-out operating system specific random overheads all following test were performed three times. Results are shown in FIGURE 1.

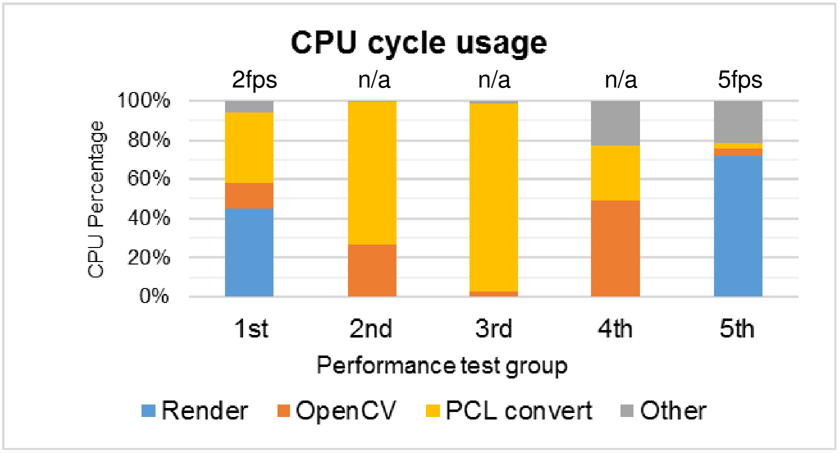

FIGURE 1: Test figures for CPU usage of different stages. Test 1 – 3 show pre-optimised PCL code, while tests 4 and 5 show optimised conversion.

- For the first test OpenCV was used to read the Tsukuba dataset as a sequence of images, loaded one at a time. OpenCV, PCL point cloud generation and rendering algorithms were used ‘as is’ without changes and as provided from public repositories. The results are shown in the first bar in FIGURE 1. PCL point cloud generation required 36% of CPU cycles, rendering 45%. This resulted in a processing speed of 2 frames per second (fps).
- In the second test rendering was disabled to identify CPU load more accurately when OpenCV loaded images one at a time.
- This is contrasted by the third test where OpenCV loaded images not as individual stills but as a video sequence. Encoding the still images into a video sequence was achieved using the OpenCV Intel IYUV. This had a dramatic effect as OpenCV CPU cycles reduced from 27% to only 3%, leaving the remaining almost 97% to the PCL conversion.
- In order to improve PCL performance numerous optimizations were made. In particular, these were a) bit-shifting pointer incrementation of color values to allow faster access and modification of values, b) vector clear and resize checks to avoid clearing and resizing a new vector when it is the same size as the previous one c) vector access optimizations through the use of data pointers which allowed the optimization of vector pushback overhead and d) several minor optimizations. The source code and documentation of these changes are available in the PCL developer’s forum [15]. The 4th bar in FIGURE 1 shows that as a result the CPU cycles needed for PCL conversion reduced by 66% to less than the cycles needed for image loading by OpenCV.
- The two improvements documented in tests 3 and 4 were finally tested in the same way as in the first test of this series, i.e. with rendering switched on again. With image loading replaced by video loading and conversion optimized the total cycle usage of these two

components now consume less than 10% of processor cycles while rendering now takes 72%. Importantly, the overall frame rate increased to 5 frames per second.

3.2. Rendering Speed Improvements

Since rendering was now the new bottleneck, steps were taken to improve its performance.

By default, rendering for the PCL is done by The Visualization Toolkit (VTK) which is an open source library for 3D computer graphics and image processing. This was replaced with a shader (i.e. graphics processor) based OpenGL rendering implementation for desktop PCs.

The basic data structure inside the PCL is the point cloud. This is an assembly of sub-fields. The main ones are 'width', 'height' and 'points'. 'Points' is a vector that stores points of PointT type which in turn can be PointXYZ, PointRGB, PointRGBA (and some other basic types). Under the existing PCL data structure non-colored point clouds of type PointXYZ could be rendered with our new OpenGL implementation but not colored ones. To enable this several changes were made to the PCL:

- A fourth float value was added to the point cloud type union. This was easy to do since the union already had memory allocated for four float values but only x, y and z floats were declared. The forth parameter added now stores the color value to be passed to the OpenGL shaders.
- To store the color values the three constituent independent integer values were bit-shifted into a single float which was then stored as the fourth value of the above union. This was done to avoid integer calculations having to be performed in the shaders while at the same time having minimal impact on the PCL.
- However, OpenGL shaders do not support bit shifting. The color values were therefore extracted in the shader by manipulating the known structure (8 bits for each of the channel). In the vertex shader the *floor()* method was used to extract each color channel separately as the return value is an integer.

The result of the above manipulations are shown in FIGURE 2. The two bars labelled 'VTK' are unchanged re-runs of the first and fifth group tests from the previous section (see FIGURE 1). When in the first test VTK is replaced by OpenGL the frame rate increases by a modest 50% to 3 fps. When, however, this is done in the optimized system produced in the previous section the speed improvement is considerable: 38 fps. In this final system where all three components are optimized, OpenGL rendering uses only 8.5% of the processor cycles while before VTK used up 72%.

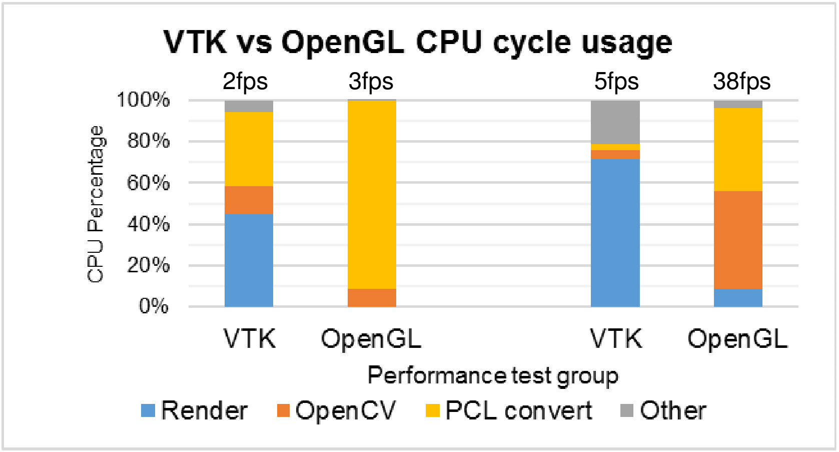

FIGURE 2: 1st and 5th re-tests (using the standard VTK renderer) compared to new OpenGL renderer. The first 2 bars represent performance of the non-optimised PCL code and the 3rd and 4th bar the optimised PCL/OpenCV code.

3.3. Voxel Grid Downsample Filter Improvements

After the point cloud has been produced further processing is usually required, e.g. for data reduction and filtering operations. A relatively low resolution point cloud of 640 x 480 (e.g. produced by the Kinect) results in 307,200 points. While for some operations (e.g. thresholding) point processing follows an O(n) notation a more complex algorithm (e.g. k nearest neighbor filtering) becomes O(nk). This can place a heavy workload on the processor.

One of the methods frequently used to lower the amount of points in a point cloud and unnecessary complexity while retaining detail and information is voxel grid down sampling. The down sampling is performed using an octree to sub-divide the point cloud into multiple cube shaped regions (voxels). After processing, all points in the voxel are reduced to a single one. This results in a point cloud that is smaller in size and complexity but is still precise enough to work with and has a smaller cost in terms of CPU performance. The PCL has a dedicated method for this called *voxelGrid.filter()*. For testing the leaf size values of the filter were 0.03f, 0.03f, 0.03f (3x3x3cm). Three groups of tests were performed as shown in FIGURE 3.

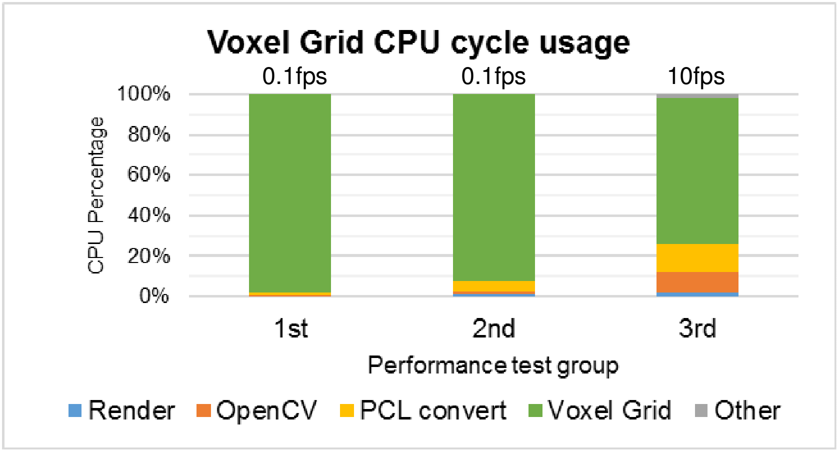

FIGURE 3: Test figures for CPU usage of voxel grid. Test 1 shows the stock code, test 2 shows results with Quicksort algorithm implemented and test 3 shows overall optimised voxel grid performance.

- In the first group of tests voxel filtering was added to the optimized processing chain developed in the previous two sections A and B. Voxel grid computation proved to be very CPU intensive with overall CPU cycle usage of 98%. This also resulted in a poor frame rate of under 0.1 fps (8.6 seconds per frame). Analysis of the filter code revealed that 30% of the processing was spent on sorting the points using a standard C++ library vector sort method.
- The second group of tests was therefore performed with the sort method replaced by a Quicksort algorithm [16]. This algorithm takes on average O(n log n) steps to sort n points, but in the worst case scenario when a chosen pivot value is the smallest or largest of the points to sort the algorithm has to make $O(n^2)$ comparisons. To avoid this possible issue a mean value is computed before the sorting to avoid using very small or very large values as the pivot. Compared to the standard C++ sort with 30% of processor cycles used, Quicksort was significantly more efficient, using only 0.9%. This unfortunately improved the overall filter method by only 5.2% as the computation shifted to different parts of the algorithm, mostly to vector access overheads.
- For the third test group vector access was therefore optimized by replacing vector pushback calls with pointer accesses and improving the centroid finding which together took up 65% of the processing. These changes reduced the voxel filter computation time by 26% to an overall contribution of that in total using only 72% of CPU cycles.

The combined changes to the sorting and vector processes increased the frame rate 91-fold to an average frame rate of about 10 fps.

3.4. Pass Through Filter Improvements

Another PCL provided post-processing method is *passthrough.filter()* which is as a means to allow the removal of points from the cloud which are not within a specified range. This allows the point cloud to be adjusted in any coordinate direction similar to a frustum cut-off. The *passthrough.filter()* method accepts parameters for upper and lower limits and a direction along the x, y or z axis. For the Tsukuba dataset the depth range values of 3 and 12 were used for testing in the z coordinate direction. Two groups of tests were performed with results shown in FIGURE 4.

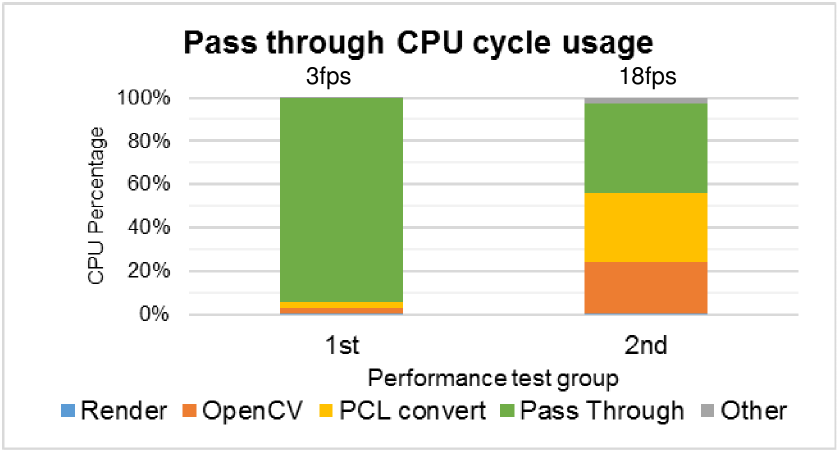

FIGURE 4: Test figures for CPU usage of pass through filter. The first and second test showing the stock code performance and second improved code performance respectively.

- In the first test the pass through filter was appended to the optimized processing chain outlined previously in sections A and B. The filter was very CPU intensive using 93.6% of cycles bringing down the frame rate to 3 fps. Analysis of the code showed that (as before with voxel filtering) vector accesses were inefficient.
- After vector access optimization along the lines outlined before with voxel filtering and improving the non-finite entries check (54%) as well and field value memory copy calls (24%) the pass through filter now only consumes 41% of CPU cycles with the frame rate rising to 18 fps.

3.5. Combined Pre-processing Chain

The PCL modules analyzed above when combined create the main pre-processing chain of the point cloud manipulation. The order in which these algorithms are applied makes a substantial performance difference.

Running the voxel filter first proved to be the slower combination as the down sampling had to be performed on the whole point cloud, in this case 307,200 points. Looking at FIGURE 5 it can be seen that voxel grid computation is the most CPU intensive task taking up 92% of all processing. Pass through filtering only took up 2% of CPU cycles and organized PCL conversion 4%. An optimized version saw a more balanced use of the processing with voxel grid processing lowered to 68% and pass through filtering at 12%. Organized conversion rose to 10% and OpenCV's contribution increased to 7% from 0.2% previously.

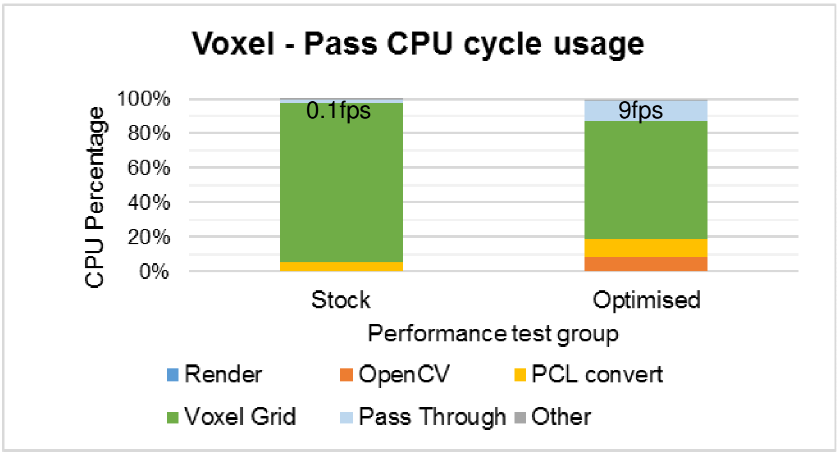

FIGURE 5: CPU usage shown for pre-processing applying voxel grid filter computations before pass through filtering.

When the pass through filter was applied first the performance changed by a great margin. As shown in FIGURE 6 it can be seen that the voxel grid process is still the most CPU intensive part but has improved over the previous order. It was found to only use 65% of processing steps instead of 92%. This led to CPU cycles being distributed more evenly between the pass through filter (13%) and organized PCL conversion (20%). The optimized version of the modules exhibits the most even distribution of processing with voxel grid contribution lowered to 21% and pass through filtering taking up 33% of CPU cycles. Organized conversion used up 24% and OpenCV 18% respectively.

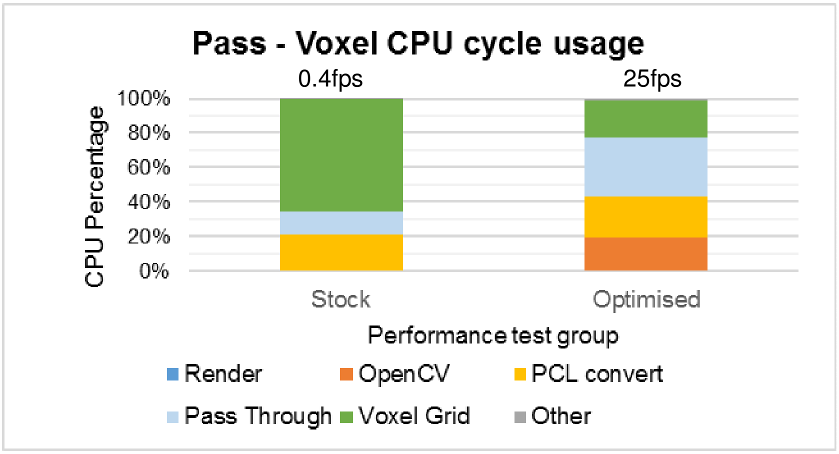

FIGURE 6: CPU usage shown for pre-processing applying pass through filtering before voxel grid computations.

The order of code execution has led to a significant change in performance (see FIGURE 7). When the voxel grid was processed before the pass through filter the stock code was not able to render more than 0.1 fps, i.e. it took around 9.1 seconds to render a single frame. This order when used with the optimized code has shown a significant improvement as the frame rate increased to 3fps, i.e. it only took 98 milliseconds on average to render a single frame, making it on average up to 93 times faster. Similar results were seen in the reverse arrangement. The stock code with pass through filtering being applied first was able to render 0.4 fps (2.5 seconds per frame) which is a four times better performance. The biggest change was seen in the overall optimised code frame rate which on average was 25 fps making it close to real time performance as it only took 37 miliseconds to render a frame. Overall this is a 69 times better performance compared to the original unaltered stock code.

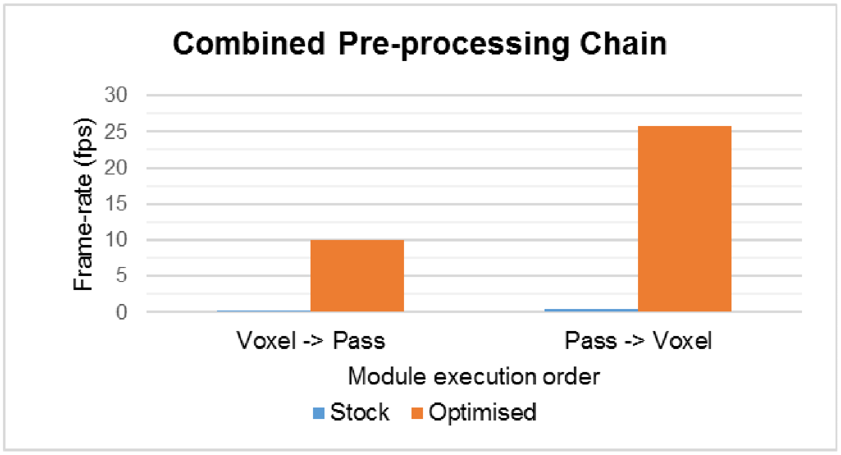

FIGURE 7: Frame rates shown of stock and optimised modules in different execution orders.

To further support and test the findings additional testing was performed on a wide range of devices which included embedded systems such as Raspberry Pi 1 and 2, tablets, laptops and powerful rendering machines. In total eighteen different machines were used to perform a comparative evaluation between the stock and optimised code, of which some ran a Linux operating system to give a full spectrum of hardware and software combinations. These results show that optimised code was able to increase the performance for every single machine tested. The embedded systems saw the smallest increase due to their lack of power on the ARM based processor, but still saw four times better performance with optimise code compared to stock. As the power of machines increased so did the optimised code performance while stock stayed almost level.

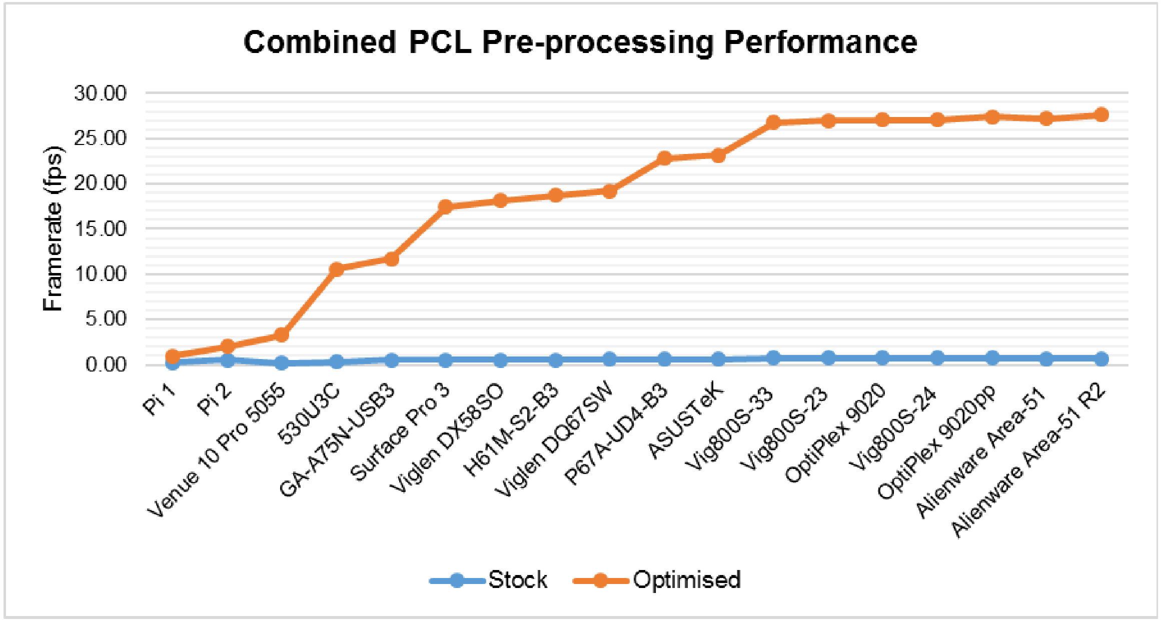

FIGURE 8: Comparative evaluation test results between stock and optimised code on eighteen different machines sorted from least powerful(left) to most powerful(right).

4. CONCLUSIONS

Since the PCL is a general purpose and multi-platform library many of its internal aspects are generalized, not all parts are optimized and performance can suffer on time sensitive processing. As shown in section 3 optimized PCL modules provide significant performance gains over the stock modules. When neglecting the minimal cost of performance testing measurement overheads speed increased 2.4 times for the organized PCL conversion, 91 times for voxel grid filtering and 7.8 times for pass through filtering. As seen in section 3.5 this allows for the use of multiple PCL modules together while still maintaining near real-time frame rates giving an average of 69 times improved performance for the pre-processing of the point clouds. It is important to note that the optimized code is still generalized, not specific to a particular platform and backwards compatible with existing stock code. The optimized modules in this paper have not been changed since libraries release 2011 showing the need for the update and improvement. The point cloud pre-processing optimizations are important for various point cloud tasks such as registration, object recognition and segmentation. Part of these improvements are already being implemented to the library project by the community.

5. FUTURE WORK

Future plans focus on working with PCL developer community, and to contribute optimized algorithms to the official PCL code repository. Another part of research has already been started to allow the PCL to be used with embedded devices to perform real time point cloud processing.

6. REFERENCES

[1] S. Ruwen, W. Roland and R. Klei, "Efficient RANSAC for Point-Cloud Shape Detection," *Computer Graphics Forum,* vol. 26, no. 2, p. 214–226, 2007.

[2] S. C. Rusu Radu Bogdan, "3d is here: Point cloud library (pcl)," in *Robotics and Automation (ICRA), 2011 IEEE International Conference*, Shanghai, 2011.

[3] C. Sun, "A Fast Stereo Matching Method," in *Digital Image Computing: Techniques and Applications*, Auckland, 1997.

[4] S. Izadi, D. Kim and O. Hiliges, "Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera," in *24th annual ACM Symposium on User Interface Software and Technology*, New York, NY, 2011.

[5] D. Lanman, D. Crispell and G. Taubin, "Surround Structured Lighting for Full Object Scanning," in *Sixth International Conference on 3-D Digital Imaging and Modeling*, Montreal, Aug. 2007.

[6] A. Zhang, S. Hu, Y. Chen, H. Liu, F. Yang and J. Liu, "Fast Continuous 360 Degree Color 3D Laser Scanner," in *The Internal Archives of the Photogrammetry, Remote Sensing and Spatial Information sciences, Volume XXXVII*, Beijing, 2008.

[7] Microsoft, "Kinect for Windows," Microsoft, [Online]. Available: https://www.microsoft.com/en-us/kinectforwindows/develop/. [Accessed 2 June 2015].

[8] I. Budak, D. Vukelić, D. Bračun, J. Hodolič and M. Sokovi, "Pre-Processing of Point-Data from Contact and Optical 3D Digitization Sensors," *Sensors,* vol. 12, no. 1, pp. 1100-1126, 2013.

[9] X. Zhang, C. K. Sun, C. Wang and S. Ye, "Study on Preprocessing Methods for Color 3D Point Cloud," *Materials Science Forum,* Vols. 471-472, pp. 716-721 , 2004.

[10] Bentley Systems, "Bentley Pointools V8i," Bentley Systems, [Online]. Available: http://www.bentley.com/en-US/Promo/Pointools/pointools.htm. [Accessed 16 June 2015].

[11] Mirage-Technologies, "Home: PointCloudViz," Mirage-Technologies, [Online]. Available: http://www.pointcloudviz.com/. [Accessed 16 June 2015].

[12] Faro, "Home: PointSense," Faro, [Online]. Available: http://faro-3d-software.com/CAD/Products/PointSense/index.php. [Accessed 16 June 2015].

[13] E. K. Stathopoulou, J. L. Lerma and A. Georgopoulos, "Geometric documentation of the almoina door of the cathedral of Valencia.," in *Proceedings of EuroMed2010 3rd International Conference dedicated on Digital Heritage*, Cyprus, 2010.

[14] S. Martull, M. Peris and K. Fukui, "Realistic CG stereo image dataset with ground truth disparity maps," *Trak-Mark,* 2012.

[15] Point Cloud Library, "Point Cloud Library (PCL) Developers mailing list," Naddle, [Online]. Available: http://www.pcl-developers.org/. [Accessed July 2015].

[16] C. A. R. Hoare, "Quicksort," *The Computer Journal,* pp. 10-16 , 1962.

[17] Willow Garage, "Software: ROS," Willow Garage, 3 June 2015. [Online]. Available: https://www.willowgarage.com/pages/software/ros-platform.

[18] Itseez, "Home page: OpenCV," Itseez, [Online]. Available: http://opencv.org/. [Accessed 15 January 2015].

[19] Kitware, "Home: VTK," Kitware, [Online]. Available: http://www.vtk.org/. [Accessed 15 June 2015].

[20] GiHub, "Point Cloud Library Repository," [Online]. Available: https://github.com/PointCloudLibrary/pcl. [Accessed 23 June 2015].