Optimizing a Mobile Robot Control System using GPU Acceleration

Nat Tuck, Michael McGuinness, and Fred Martin*

University of Massachusetts Lowell, 1 University Ave, Lowell, MA, US

ABSTRACT

This paper describes our attempt to optimize a robot control program for the Intelligent Ground Vehicle Competition (IGVC) by running computationally intensive portions of the system on a commodity graphics processing unit (GPU). The IGVC Autonomous Challenge requires a control program that performs a number of different computationally intensive tasks ranging from computer vision to path planning. For the 2011 competition our Robot Operating System (ROS) based control system would not run comfortably on the multicore CPU on our custom robot platform. The process of profiling the ROS control program and selecting appropriate modules for porting to run on a GPU is described. A GPU-targeting compiler, Bacon, is used to speed up development and help optimize the ported modules. The impact of the ported modules on overall performance is discussed. We conclude that GPU optimization can free a significant amount of CPU resources with minimal effort for expensive user-written code, but that replacing heavily-optimized library functions is more difficult, and a much less efficient use of time.

Keywords: robotics, stereo vision, GPU acceleration

1. INTRODUCTION

Autonomous robot navigation is an area that has undergone much research and development, leading to a wide range of robotic platforms. These range from mass-produced robotic vacuums, to mid-sized research platforms like ours, to full-size autonomous automobiles.[1] As advancement in sensor and computer technology continues and as sensor prices come down, smaller platforms are gaining access to sensors and that used to be reserved for larger platforms. Leveraging more sensors and more sophisticated sensors however requires more computation. This paper is principally inspired by the desire to incorporate more sensors and computation without going beyond the size envelope of a single desktop computer.

Our robotics platform, Stark, consists of a four wheel differential drive chassis with a standard desktop computer mounted on-board. It also includes a SICK rangefinder, GPS receivers, a compass, and a stereo camera rig. In this paper we performed testing in a simulated environment with a simulated version of Stark, consisting of the same chassis and sensor load-out.

The primary control program for Stark is designed for participation in the Association for Unmanned Vehicle Systems International (AUVSI) Intelligent Ground Vehicle Competition (IGVC). The competition consists of two primary events, the navigation challenge, and the autonomous challenge. The navigation challenge is an outdoor GPS-based navigation problem where a robot needs to autonomously navigate to a number of pre-defined GPS waypoints while avoiding positive obstacles such as construction barrels, sawhorses and mesh fences. The robot must do this while also strictly following a course bounding box which is indirectly given by GPS coordinates. The autonomous challenge is effectively a superset of the navigation challenge, requiring additional computer vision processing and logic to handle white lines, real and simulated potholes, ramps, and navigation flags.

2. OUR CONTROL PROGRAM

2.1 Full design

Our control system is built on the Robot Operating System (ROS).[2] ROS provides a networked communications scheme using the publisher/subscriber strategy. It also provides a number of pre-built components, such as simulators, visualizers, and relative geometric position tracking. ROS encourages a modular design by nature. Our full control program consists principally of the following modules:

*The authors can be contacted at {ntuck, mmcguinn, fredm}@cs.uml.edu

Four control modules:

- A navigation module which keeps track of which GPS waypoint the robot is traveling towards and a sequence of waypoints to visit in the future.
- A mapping module which builds a map of the challenge course. This is done using an occupancy grid approach with a fusion of the sensor modules discussed below.
- A pathfinding module which uses the map to find the best path to the next intermediate goal. This is accomplished by running A*[3] on the occupancy grid of the map.
- A obstacle avoidance module which issues actual motor commands to safely transition the robot to the next intermediate waypoint. We use the Working Obstacle Avoidance Heuristic (WOAH),[4] an algorithm developed earlier by the authors for obstacle avoidance using LIDAR rangefinders.

Five Sensor Modules:

- A position estimation module which keeps track of the position and orientation of the robot is based on GPS, compass, and wheel odometry readings. This is necessary for mapping and determining if the robot has successfully hit waypoints.
- A laser rangefinder module which feeds data from a laser rangefinder to provide very accurate obstacle distance data in a plane in front of the robot.
- A stereo vision module which provides depth information that is used as a secondary source of obstacle information. Stereo vision can also locate things such as negative obstacles which a single laser range finder traditionally cannot.
- A safe space detection module which uses computer vision to detect clear areas to drive in. This module recognizes painted white lines and simulated potholes for the purpose of avoidance, and is also capable of detecting ramps as a traversable surface.
- A flag detection module which detects nautical-style flags that signal the robot to avoid them on a specific side.

2.2 Reduced design

Although the full design is more interesting, due to time constraints we instead chose to analyze a reduced functionality version of the system that is only capable of solving the IGVC Navigation challenge. This allowed us to avoid the safe space detection and flag detection modules of which we did not have working versions. Because the navigation challenge is a subset of the autonomous challenge, our reduced system simply consists of a subset of the modules: navigation, mapping, pathfinding, obstacle avoidance, and stereo vision.

3. BACON

In order to run parts of our robot control system on a GPU we used an experimental GPU-targeting compiler called Bacon[5] developed by one of the authors. Bacon is built upon the existing OpenCL standard in order to make it easier for programmers to write high performance kernels for GPU accelerated applications. The OpenCL C syntax is extended into a new language, Bacon C, intended to make development significantly more convenient and enabling some additional compiler optimizations as this code is compiled via OpenCL at runtime. Bacon currently provides support only for AMD GPUs although it would be reasonably straightforward to modify it to work with with Nvidia GPUs or any other hardware with an OpenCL 1.1 implementation.

Bacon provides two core features over using OpenCL directly. First, it provides syntax improvements and automatic C++ interface generation that significantly improve the developer experience. Second, it uses user annotations to perform data-based optimizations that OpenCL compilers don't have sufficient information to easily perform.

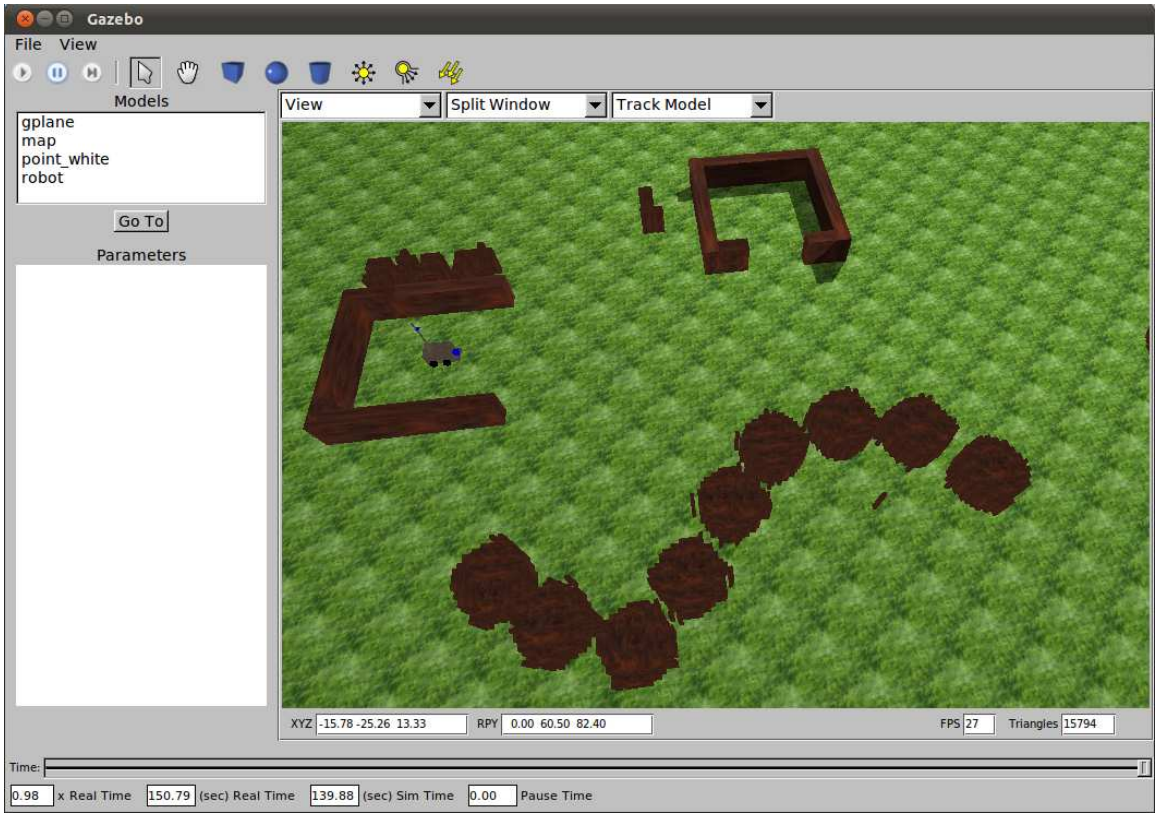

Figure 1. Our simulation environment using the Gazebo 3D simulator.

The key optimization performed by the Bacon compiler is just in time specialization. When a Bacon kernel is written, some integer arguments can be marked as specialization-time constants. When a kernel is first called with a given set of values for those arguments a specialized version of that kernel is generated with those variable arguments replaced with constant values. This allows for a variety of optimizations be performed by the OpenCL compiler such as constant propagation and loop unrolling. Further, Bacon can work around OpenCL's lack of support for variable size local arrays by transforming array declarations that depend only on these specialization-time constants to constant sized array declarations in the just-in-time generated OpenCL code.

Testing of Bacon on microbenchmarks has shown that runtime-constant loop unrolling can provide upwards of a 2x speedup over non-unrolled OpenCL code. Since the output of Bacon is roughly equivalent to hand-optimized OpenCL kernels our results would have been similar if we had written OpenCL code directly.

4. SIMULATION AND TESTING ENVIRONMENT

4.1 Gazebo Simulation

We performed our testing using the Gazebo simulation tool which is provided as a ROS package. Gazebo is a 3D simulator, which is necessary to provide meaningful simulated cameras for computer vision. This allowed us to simulate all of the sensors used by our control system design.

The simulated IGVC navigation course is built by Gazebo from a 2D image of obstacles extruded vertically into "walls" and "barrels" that are high enough to be detected by the simulated laser rangefinder but low enough that the simulated cameras can see over them at short distances. This configuration allowed us to get appropriately different behavior from our two sensors. The course includes a start box and nine clearly marked waypoints, organized into two sections separated by a fence consistent with the 2011 IGVC rules.[6]

Gazebo requires a significant amount of compute power to run, to the extent that it barely runs smoothly on a high end quad core workstation. We took advantage of the network transparency of the ROS system and ran Gazebo on its own dedicated machine (Table 1).

Machine Role	Processor	RAM	Graphics Card	OS
Gazebo	Intel Core i7 870 @ 2.93 GHz	4 GB	Nvidia GTS 450	Ubuntu 11.04
Test System	Intel Core i5 2500K @ 3.3 GHz	4 GB	AMD Radeon 6850	Ubuntu 11.04

Table 1. Hardware used for simulation and testing.

4.2 Test Machine

For our test machine we used a workstation with hardware (Table 1) that could feasibly be mounted on our robot platform. The GPU we tested with was an AMD Radeon 6850, a mid-range consumer graphics card that is available at the time of publication for under $200. The Intel i5 2500K CPU we used benchmarks at approximately 50 GFLOPS,[7] while the Radeon 6850 with 960 shader units and a gigabyte of GDDR5 graphics memory provides a theoretical 1,500 GFLOPS,[8] providing a significant theoretical opportunity for performance improvements by moving computations to the GPU.

5. PROFILING

Two techniques were used to profile the running robot system: CPU usage sampling and compute kernel timing.

The node-based design of the ROS system allowed us to split the different computationally intensive modules of the system out into separate nodes. Since each node is a separate operating system process, this allowed us to compare the overall CPU usage of the modules using the standard utility "ps". We wrote a script to sample the system CPU usage this way every 100ms and perform basic statistical analysis of the result. This measurement method does not provide perfect accuracy, but it was able to clearly identify the modules worth considering for optimization and provide a rough order of importance among those modules.

Once we identified the modules worth examining in depth, we added timing code around the core compute kernels that those modules depended on. Essentially, the timing code informs a special ROS node when each monitored section is started and completed. This provides more specific profiling information about which compute kernels take the largest amount of time to execute.

In general, our control system is designed to operate at 10Hz, which provides an upper bound on the time for any processing step of 100ms. The exception to this is the pathfinding module, which is designed to operate asynchronously and can take longer than 100ms to generate a path without degrading the overall performance of the control system.

5.1 Profiling Results

Kernel-based profiling revealed four compute kernels that consumed the majority of the compute time: map merging for the pathfinder, point cloud filtering, and stereo disparity calculation. See figure 2 for a pie chart representing the use of CPU resources by each component.

6. OPTIMIZATIONS

Based on our results from profiling our control system we considered three components for optimization: mapping and pathfinding, stereo disparity, and point cloud processing. These components represented a significant majority of the compute time used by our control system. While the A* algorithm we use for pathfinding doesn't parallelize easily, the mapping portion of that component does.

We had reasonable success at porting our mapping system to the GPU. We were unable to get a performance improvement on stereo disparity as explained below. Due to our experience with trying to beat heavily hand-optimized CPU code and time constraints, we did not attempt to optimize the point cloud processing component.

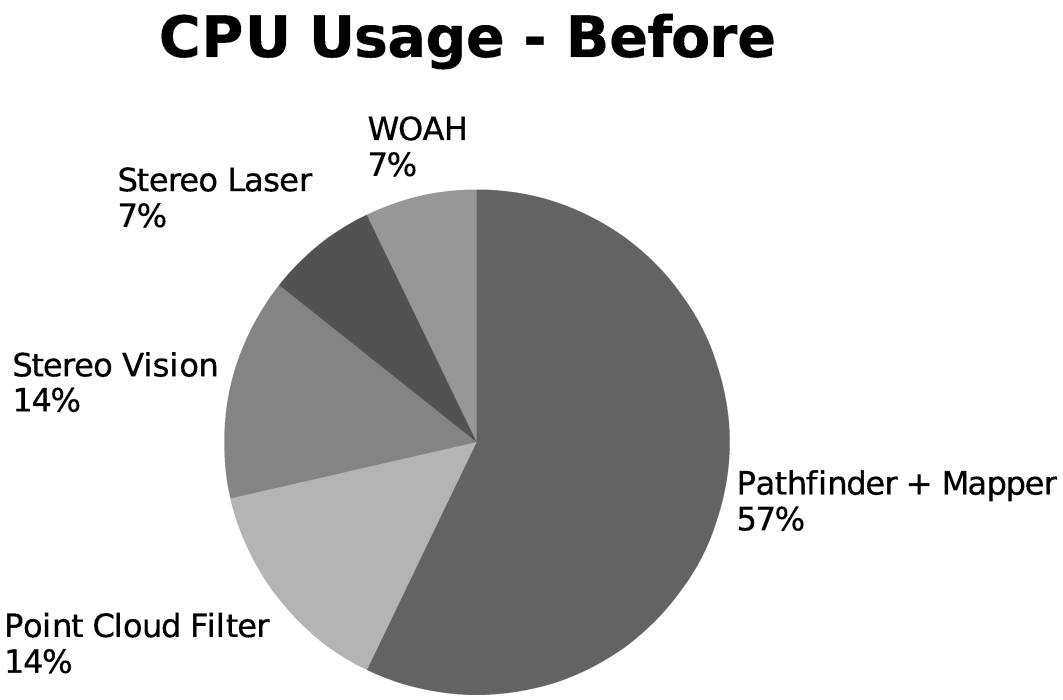

Figure 2. Pie chart representing CPU usage before optimization.

6.1 GPU Mapping

Our control system maps by generating an occupancy grid[3] for each sensor and then merging the grids. In the full system this would be several grids, but in the reduced system we profiled there were only two, one for the laser rangefinder and one for the stereo vision simulated laser.

Applying a laser scan to an occupancy grid can be parallelized by handling each ray in parallel. In order to mitigate conflicts due to parallel writes, updates are applied in two stages: first any cells that a ray passes through are marked as clear, then cells that a ray terminates in are marked as blocked. This means that writes can only conflict with identical writes, which is acceptable since at least one of the writes will occur last and be included in the result.

Parallel map merging is trivial. Each cell in the output grid is a function of the corresponding cells in the input grid, which can be computed in parallel. In our mapper we simply take the maximum cell value.

6.2 GPU Stereo Disparity

Profiling showed that stereo disparity and point cloud filtering were the next largest CPU users after the mapping module. Based on the authors' familiarity with the algorithms involved, we chose to work on stereo disparity next.

To calculate stereo disparity, we use the stereo block matching routine from the OpenCV[9] computer vision library. This compute kernel is heavily hand-optimized C++ code that uses a rolling buffer and takes advantage of the invertablity of addition in the sum of absolute difference cost function to avoid redundant computation of a disparity window while maintaining locality of reference.

After a significant amount of effort to determine exactly what computation the optimized OpenCV routine performed, we were able to produce a parallel version that ran slightly slower due to redundant computation that the serial implementation avoided. Due to the existence of multiple solid results in optimizing stereo disparity for GPUs,[1011] we decided not to pursue optimizing this component further.

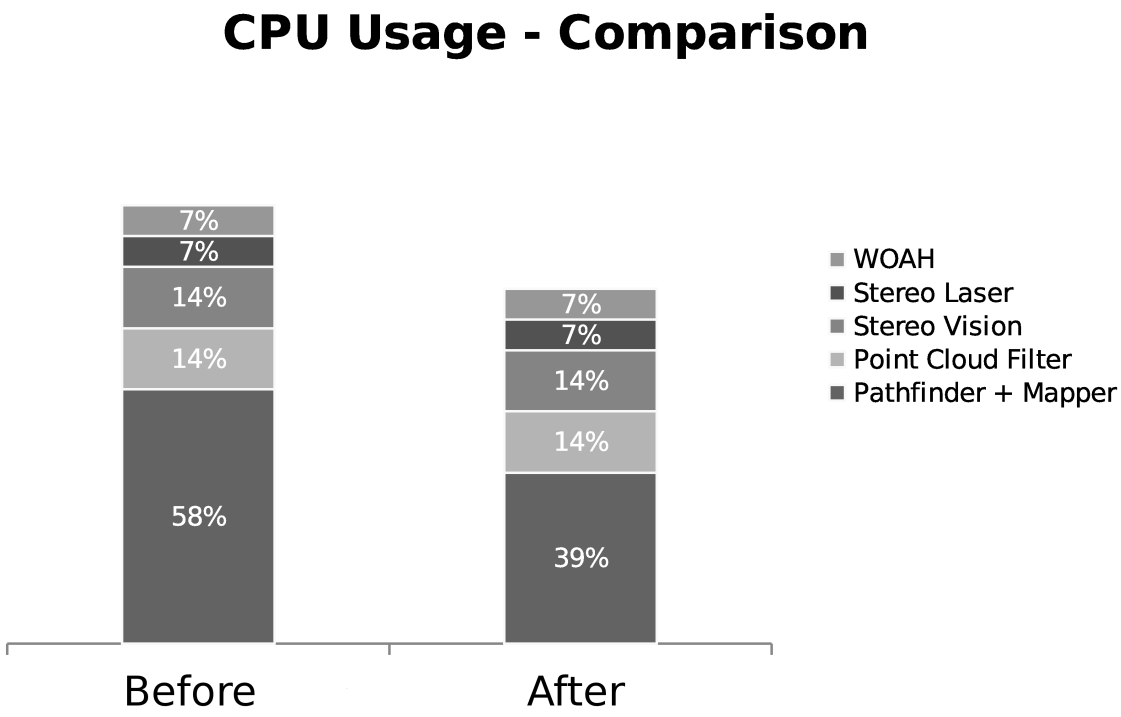

Figure 3. Stacked bar chart representing CPU usage before and after optimization. Percentages are based on baseline CPU usage of the control program before optimization.

Our point cloud filter used a sequence of routines from the Point Cloud Library (PCL).[12] Based on our experience with trying to parallelize the stereo disparity algorithm and the expected difficulty of beating heroically optimized C++ code in general we did not try to parallelize the point cloud filter.

7. OPTIMIZATION RESULTS

We again profiled the control system after replacing the mapper with the new optimized, GPU enabled version. Table 2 shows a comparison of compute kernel timings before and after optimization. The merge operation in particular shows a massive improvement from parallelization. The map operation itself also showed some improvement. Notably, the pathfinder's performance actually degraded. We speculate this degradation is due to a poor translation from an older pathfinder module that didn't natively support the latest map data format. The pathfinder should be able to be improved to run as quickly as the original version with further tweaks.

Table 2. Comparison of timings for compute kernels before and after optimization.

Compute Kernel	Time (before)	Time (after)
Pathfinding (CPU)	0.038	0.156
Mapping	0.039	0.006
Map Merging	1.150	0.004

See Figure 3 for a stacked bar chart built using CPU profiling from the optimized and unoptimized control programs. Note that even though the pathfinder is still not quite running in real time, we did manage to reclaim some (about 19% from the original baseline) CPU time that is now idle. Given the improvements we believe can be made in the pathfinding kernel, the control program should be able to run not only in less CPU time but also allow the pathfinder to run at the 10Hz rate the rest of the control program does.

8. FUTURE WORK

We would like to implement the full control program design for the IGVC autonomous challenge. This will the integration and development of additional sensors that can detect white lines, navigation flags, and new obstacles that are introduce in future IGVC contest rules. Integrating additional sensors should be straightforward, since we have shown the occupancy grid merging technique to work with two sensors and can merge such grids reasonably efficiently.

The profiling mechanisms we have developed for this paper will be used in the future to drive further optimization work. Getting a speedup by writing a GPU kernel with Bacon is significantly easier than hand-optimizing serial code for a CPU, so we expect to use this technique more to optimize our own code in the future.

9. CONCLUSION

Using GPU acceleration, we were able to reduce the execution time of our slowest compute kernel, map merging, by two orders of magnitude. The impact of this improvement on the overall control system is dramatic in that the pathfinding performance was improved to be nearly real-time within the given 10Hz control rate. This impact indicates to us that it is very worthwhile to budget in (power and space-wise) a GPU into our research platform in order to increase its performance without going beyond its single desktop computer footprint. It is also clear that while porting compute kernels to run on a GPU can grant a performance increase without much serious optimization work, popular scientific computing libraries often have highly optimized CPU implementations that cannot be bested by trivial GPU implementation. Although there is significant theoretical space for performance improvements from GPU versions of these algorithms, it will require considerable work to gain an improvement in practice.

ACKNOWLEDGMENTS

Thanks to James Dalphond for assistance with revising this paper and providing some testing hardware.

REFERENCES

[1] Urmson, C., Anhalt, J., Bartz, D., Clark, M., Galatali, T., Gutierrez, A., Harbaugh, S., Johnston, J., Kato, H., Koon, P., Messner, W., Miller, N., Mosher, A., Peterson, K., Ragusa, C., Ray, D., Smith, B., Snider, J., Spiker, S., Struble, J., Ziglar, J., and Whittaker, W., "A robust approach to high-speed navigation for unrehearsed desert terrain," in [*The 2005 DARPA Grand Challenge*], Buehler, M., Iagnemma, K., and Singh, S., eds., *Springer Tracts in Advanced Robotics* **36**, 45–102, Springer Berlin / Heidelberg (2007).

[2] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A., "ROS: an open-source Robot Operating System," in [*Conference on Robotics and Automation*], (2009).

[3] Thrun, S., "Robotic mapping : a survey," in [*Exploring artificial intelligence in the new millennium*], 1 –37 (2002).

[4] Tuck, N., McGuinness, M., and Martin, F., "Woah: an obstacle avoidance technique for high speed path following," in [*Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*], **7878**, 32 (2011).

[5] Tuck, N., "Bacon: A GPU programming language with just in time specialization (working draft)," (2011).

[6] "IGVC rules (2011)," http://www.igvc.org/rules.htm (Nov. 2011).

[7] "CPU charts," Tom's Hardware Guide, http://www.tomshardware.com/charts/desktop-cpu-charts-2010/Raw-Performance-SiSoftware-Sandra-2010-Pro-GFLOPS,2409.html (Nov. 2011).

[8] "AMD Radeon 6850 graphics," Advanced Micro Devices, http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/hd-6850/Pages/ amd-radeon-hd-6850-overview.asxp (Nov. 2011).

[9] Bradski, G. and Kaehler, A., [*Learning OpenCV: Computer vision with the OpenCV library*], O'Reilly Media (2008).

[10] Rosenberg, I. D., Davidson, P. L., Muller, C. M. R., and Han, J. Y., "Real-time stereo vision using semi-global matching on programmable graphics hardware," in [*ACM SIGGRAPH 2006 Sketches*], *SIGGRAPH '06*, ACM, New York, NY, USA (2006).

[11] Weber, M., Humenberger, M., and Kubinger, W., "A very fast census-based stereo matching implementation on a graphics processing unit," in [*Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*], 786 –793 (27 2009-oct. 4 2009).

[12] Rusu, R. B. and Cousins, S., "3D is here: Point Cloud Library (PCL)," in [*IEEE International Conference on Robotics and Automation (ICRA)*], (May 9-13 2011).