Robotica (2017) volume 35, pp. 85–100. © Cambridge University Press 2015
doi:10.1017/S0263574714002914

Real-time obstacle detection using range images: processing dynamically-sized sliding windows on a GPU

Caio César Teodoro Mendes*, Fernando Santos Osório and Denis Fernando Wolf

Mobile Robotics Laboratory, University of São Paulo (USP), Av. Trabalhador São-Carlense, 400, P.O. Box 668, 13.560-970 São Carlos, Brazil. E-mails: fosorio@icmc.usp.br, denis@icmc.usp.br

(Accepted December 12, 2014. First published online: March 6, 2015)

SUMMARY
An efficient obstacle detection technique is required so that navigating robots can avoid obstacles and potential hazards. This task is usually simplified by relying on structural patterns. However, obstacle detection constitutes a challenging problem in unstructured unknown environments, where such patterns may not exist. Talukder *et al.* (2002, *IEEE Intelligent Vehicles Symposium*, pp. 610–618.) successfully derived a method to deal with such environments. Nevertheless, the method has a high computational cost and researchers that employ it usually rely on approximations to achieve real-time. We hypothesize that by using a graphics processing unit (GPU), the computing time of the method can be significantly reduced. Throughout the implementation process, we developed a general framework for processing dynamically-sized sliding windows on a GPU. The framework can be applied to other problems that require similar computation. Experiments were performed with a stereo camera and an RGB-D sensor, where the GPU implementations were compared to multi-core and single-core CPU implementations. The results show a significant gain in the computational performance, i.e. in a particular instance, a GPU implementation is almost 90 times faster than a single-core one.

KEYWORDS: Obstacle detection; Autonomous navigation; Stereo vision; Graphics processing unit (GPU).

1. Introduction
A mobile robot with autonomous navigation capabilities can serve many practical purposes. It can be a vehicle that drives itself carrying passengers and goods or an agricultural machine that frees individuals from tedious and dangerous tasks. To accomplish such useful feat, robots must sense the environment through sensors, detect transversable regions and move through them precisely and efficiently. However, autonomous navigation involves several technical challenges, which range from dealing with intrinsically noisy sensors to correctly identifying road signs.

A crucial component of an autonomous navigation system is the obstacle detection module. Regarding range images, a popular approach is to approximate the ground surface using a planar model.[1,2] Once the model parameters have been estimated, the distance from any point to the model can be calculated and, based on a threshold, points can be distinguished as either parts of the ground or obstacles. Although reasonable for indoor environments, this approach is not suitable for outdoor ones, where one may expect to find curved and hilly regions.

Another popular approach[3,4] relies on a transformation of the disparity/range image called V-disparity map, which is essentially a lateral projection of the range image where the longitudinal road profile is expected to appear as a line segment. With the road longitudinal profile extracted,

* Corresponding author. E-mail: caiom@icmc.usp.br

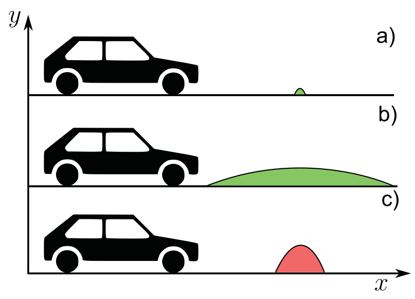

Fig. 1. 2D illustration of the intuitive notion of an obstacle. Scenario (a): a rapid, but not significant change in the terrain height; Scenario (b): a significant, but not rapid change in height; Scenario (c): a rapid *and* significant change, thus an obstacle.

obstacles are pixels that do not belong to the road profile. This approach can handle curved road longitudinal profiles, but its transversal profile is expected to be flat otherwise the road profile will appear blurred in the V-disparity map. This significantly compromises the profile extraction, hence the obstacle detection. A camera roll angle relative to the ground surface will exert the same effect even on roads with a flat transversal profile.

To circumvent such problems, some authors[5,6] create a digital elevation map (DEM), which is a grid where each cell corresponds to a part of the terrain. These approaches tend to be computationally costly and do not clearly answer the question: "what is an obstacle?". Rather, they tend to apply a number of heuristics (i.e. standard deviation of heights, average slope in x) to create a cost map and usually end up overfitting the robot setup (robot size and shape, sensor characteristics, etc.).

What is an obstacle? It is something that prevents the progress or passage of some movable entity. Regarding wheeled robots and range images, we have this intuitive notion that an obstacle is a rapid *and* significant change in the surface height, as illustrated in Fig. 1.

Talukder *et al.*[7] proposed an obstacle definition and detection method for range images using precisely this notion. According to the authors, for two 3D points to be considered an obstacle, there should be a significant difference in the height and slope between them. The method has several advantages over the ones previously mentioned: it provides and applies a clear point-wise definition of obstacle, enables a number of post-processing steps, such as clustering and temporal integration and, in practice, can handle rough terrain. It was successfully employed in a long autonomous navigation experiment, in which few geometric assumptions could be made.[8] In practice, its only drawback is its high computational cost; it may take the method seconds to detect obstacles in a practical-sized range image. Researchers usually rely on approximations,[8–10] so it can perform within a reasonable time setting.

Our main goal is to significantly reduce the processing time of the method, more specifically, achieve real-time obstacle detection in practical-sized range images without using approximations. We hypothesize that the method of Talukder *et al.* could benefit from a parallel implementation using a *GPU*. Unlike a CPU, which is mainly a serial processor, a GPU has a parallel structure that makes it especially suitable for graphics-related processing. Despite starting as a fixed-function unit, recent changes in its architectural design have increased the flexibility of its previous rigid pipeline. Such changes have enabled the exploration of its computational power for massive data-parallel (i.e. same task on different pieces of data) applications. Scientists have recognized this potential and have been applying it to speed up scientific computations.

Performance and accessibility were some of the factors that contributed towards the choice of a GPU. Modern GPUs are widely available in the market and their processing capacity can overcome the one of high-end CPUs by an order of magnitude . However, to benefit from this potential, the programmer must take into account the architecture and workflow of the GPU to implement the target application.

This article describes and evaluates single-core, multi-core, and GPU implementations of Talukder *et al.* method, sharing general and specific optimization techniques. It also addresses some issues related to the implementation suggested by the original authors of the method. The study is not concerned with the method detection quality, but its *proper* CPU implementation and possible speedup due to the use of a GPU. The article is an extension of a previous conference paper.[11]

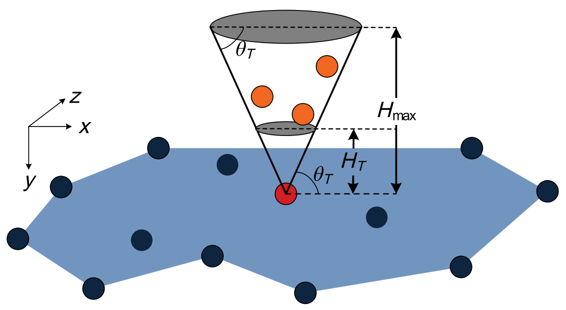

Fig. 2. Illustration of a search for compatible points. Orange points are compatible and blue ones are not, regarding the pivot point (red point), adapted from ref. [7].

The paper is organized as follows: Section 2 presents Talukder *et al.* obstacle detection method; Section 3 provides an overview of general-purpose computing using GPUs; the implementations are detailed in Section 4; processing times are presented in 5; Section 6 discusses the results and related work; finally, Section 7 draws the conclusions and suggests future work.

2. Obstacle Detection

Obstacle detection methods that apply for range images may work in two different domains: the range image itself or the corresponding point cloud. The latter can be generated by transforming the range image based on the intrinsic parameters of the sensor. A point cloud that can be indexed by its projection (i.e. range image) index space is called an organized point cloud. A range image indexed by u and v returns a single value d corresponding to the depth, i.e. $I_{u,v} = (d_{u,v})$, where I is the range image. An organized point cloud can be indexed by the same u and v and returns three values x, y and z usually in real world units, i.e. $P_{u,v} = (x_{u,v}, y_{u,v}, z_{u,v})$, where P is the point cloud. Here we will assume that both domains are available and the point cloud is indexed by the same index space of the range image.

By extending the intuition shown in Fig. 1 for a three-dimensional space,[7] proposed a point-wise obstacle definition for organized point clouds, where $\mathbf{p}_1 = (x_1, y_1, z_1)$ and $\mathbf{p}_2 = (x_2, y_2, z_2)$ are called compatible *and* considered obstacles if they satisfy the following conditions:

1. $H_T < |y_2 - y_1| < H_{max}$;
2. $|y_2 - y_1|/||\mathbf{p}_2 - \mathbf{p}_1|| > \sin(\theta_T)$;

where H_T, H_{max} and θ_T are empirically determined constants.

The first condition refers to the difference in the height between two points, i.e. it checks the significance of the height difference. The second regards the angle between the line segment containing the two points and the ground plane, i.e. it checks the significance of the slope. Parameters H_T and θ_T correspond to the thresholds used for determining if the height difference and slope are respectively significant. Parameter H_{max} plays a more subtle role, as it checks if the two points are connected. Imagine a point cloud referent to a city intersection, where there are some points referent to a traffic light and right below it some points referent to the road. Between the points of the traffic light and the road there is a large height difference and a high slope, but they do not represent obstacles because nothing connects them and parameter H_{max} prevents this mistake.

Given a pivot point $\mathbf{p}$, its effective search area for compatible points resembles an upside–down cone, as shown in Fig. 2. A straightforward way to implement the method would be to compare every point with every other point, which would result in a high computational cost and an O(N^2) complexity, where N is the number of points. Assuming that the coordinate system of the camera and the point cloud are aligned (*projection assumption*), the computational cost can be reduced by projecting the search area to a region of the reference image plane taking into account parameters H_{max}, θ_T, and the focal length f (in pixels) of the camera. The search area for compatible points can be projected to a triangle on the image plane, as shown in Fig. 3. The height of the triangle is equal to fH_{max}/p_z and the opening angle of the triangle is equal to $180 - 2\theta_T$ degrees. Now the complexity

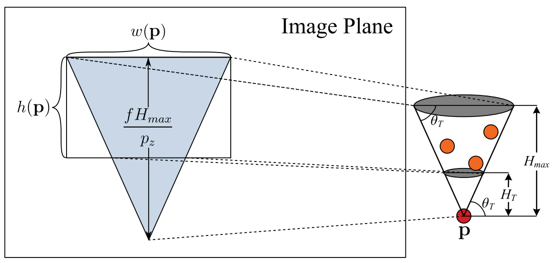

Fig. 3. Projection of the search area for a triangle in the image plane, adapted from ref. [7].

is reduced to $O(NK_{mean})$, where N is the number of points and K_{mean} is the mean area of triangle projections onto the image plane.

This method assumes that the coordinate system of the point cloud is aligned with the ground plane (*method assumption*), i.e. the y-axis of the point cloud is aligned with the ground plane height. For the most part, this assumption may be false. While navigating, the vehicle undergoes pitch and roll motions. Such motions may affect the camera coordinate system, which may affect the point cloud coordinate system. A way to tackle this issue is to connect the camera to a pan-and-tilt system that compensates for the vehicle motions based on the feedback of some inertial sensor. Another approach is to use the same feedback to virtually tilt the point cloud. While avoiding the need for a physical pan-and-tilt system, this approach violates the projection assumption (the coordinate system of the camera and the point cloud are aligned).

In practice, there are four manners to handle the method and projection assumptions:

1. Ignore the method assumption while maintaining the projection one;
2. Use a physical pan-and-tilt system to compensate for the vehicle motions maintaining both assumptions;
3. Virtually align the point cloud and ignore the projection assumption;
4. Virtually align the point cloud and derive a projection that accounts for the misalignment between the camera and the point cloud coordinate system.

The first option is the most popular and has been employed by the original authors and also in refs. [8] and [10]. It is also worth mentioning that by reading the original authors' papers,[7,12] one may infer that by increasing the projection size the method assumption would be somehow alleviated. Enlarging the projected size only makes sense if the point cloud is virtually aligned with the ground and the projection accounts for it (option 4), which apparently was not the case in refs. [7] and [12]. However, we acknowledge it is a natural assumption since we made it ourselves in ref. [11].

We have also chosen option 1. In practice, the method is robust and yields great results even when the "method assumption" is violated, as the results from refs. [7], [8], [10], and [12] suggest. We also have the benefit of the projection assumption but, as the original authors, rather than projecting a triangle or trapezoid we employ a rectangular region (see Fig. 3) according to:

$$h(\mathbf{p}) = \left\lceil \frac{f.(H_{max} - H_T)}{z} \right\rceil, \tag{1}$$

$$w(\mathbf{p}) = \left\lceil \frac{2f\,\tan(90 - \theta_T)H_{max}}{z} \right\rceil, \tag{2}$$

where h is the function that returns the height of the projected rectangle to a point $\mathbf{p}$ and w is the function that returns its width. We believe that a rectangular region provides the best trade-off between computational complexity and number of unnecessarily processed points.

As previously discussed, there is no need to create larger projections than the ones calculated by these functions. We provide evidence that this is the case in Section 5.

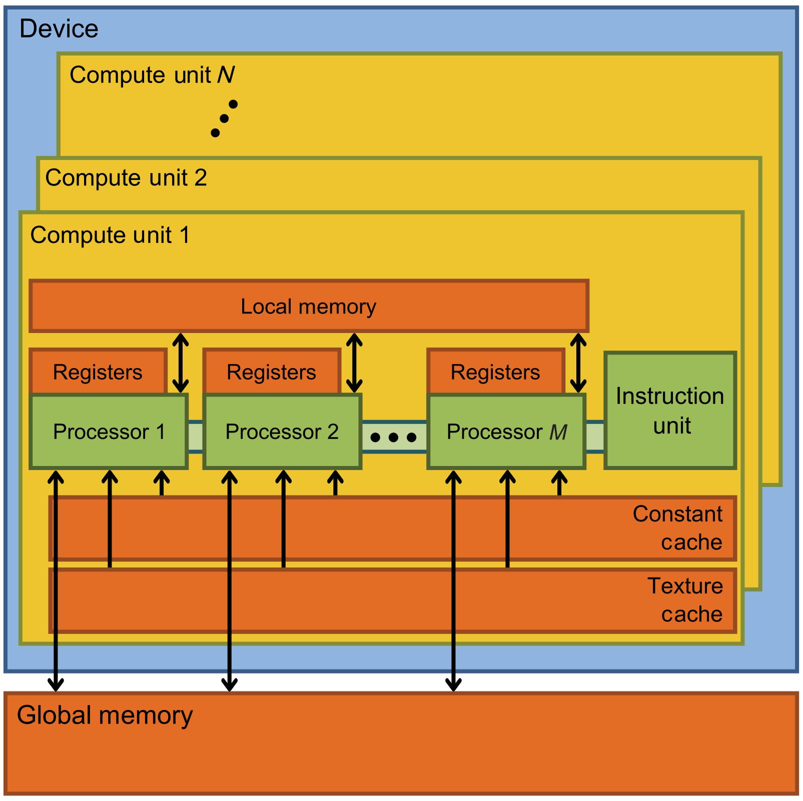

Fig. 4. Simplified GPU architecture consisting of N compute units with M processors each, adapted from ref. [13].

3. General-Purpose Computing on Graphics Processing Units

This section presents a brief overview of the architecture of modern GPUs and concepts related to general-purpose computing on GPUs (GPGPU). The OpenCL terminology and abstraction layer were used to provide a device-independent overview and the CUDA counterparts of the main concepts are provided in parentheses.

While CPUs rely on few and sophisticated processors, GPUs employ numerous (hundreds) simplified processors that operate at relatively low frequencies. This architecture is motivated by its main application, i.e. graphics-related processing, which consists in applying the same operation over and over to different chunks of data.

As shown in Fig. 4, a GPU is divided into *compute units* (streaming multiprocessors). Each compute unit consists of several *processing elements* (scalar cores) that can perform a *work-item* (thread) each. Work-items within a compute unit can communicate via local memory (shared memory) and synchronize. Work-items from different compute units can communicate only through the global memory (or device memory) and there is no explicit way of synchronizing them *on* the GPU. Global synchronization may be achieved either via CPU or implicitly on the GPU by handcrafted strategies.[14]

Work-item is a fundamental concept in GPU programming. It represents an instance of a *kernel*, which is a function (as in programming) specified by the developer and running on a processing element. Work-items are mapped onto an n-dimensional index space called *NDRange* (grid), used by the GPU to schedule the execution of work-items. The processing is finished when all elements of the *NDRange* have been processed. Work-items can be grouped into work-groups; all work-items belonging to the same group will be performed by a single compute unit, therefore they can share local memory and synchronize.

The concept of *wavefront* (warp) must also be introduced. Instead of scheduling single work-items for execution, the GPU schedules wavefronts. A wavefront refers to a part of a work-group that will share the same control unit, therefore its instructions are executed strictly in parallel (lock-step).

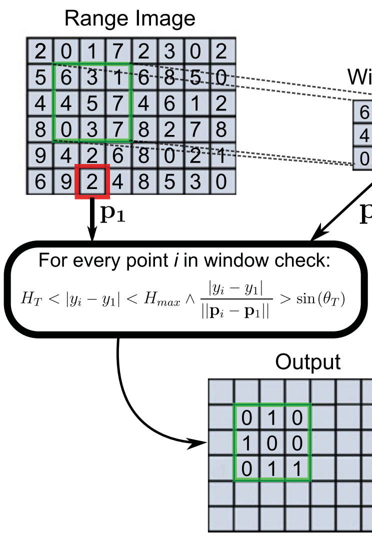
Range Image
Window
$\mathbf{p_1}$
$\mathbf{p_i}$
For every point *i* in window check:
$H_T < |y_i - y_1| < H_{max} \wedge \frac{|y_i - y_1|}{||\mathbf{p}_i - \mathbf{p}_1||} > \sin(\theta_T)$
Output

Fig. 5. Computation of Talukder *et al.* method: the pivot point ($\mathbf{p_1}$) is checked against every element of its respective search area (window) and the result is written as boolean on an output matrix, i.e. 1 for obstacle and 0 for non-obstacle.

3.1. Considerations on performance

Some performance issues must be addressed when developing for a GPU. This section provides an overview of the most significant and general performance considerations.

Hiding memory access latency is one of the major challenges in GPGPU. When work-items within a wavefront access global memory, some patterns produce a coalesced memory operation. Such an operation uses the full bandwidth of the memory subsystem. These patterns and the way they use the bandwidth are device-dependent. However, the general idea is that multiple fetch operations can be translated to a single 'wide' fetch by the GPU, i.e. at most 16 fetch operations (half wavefront) can be replaced with a single fetch on NVidia GPUs. A simple and widely used pattern that produces a coalesced operation in most cases is the use of adjacent work-items to access adjacent memory addresses within a wavefront.

Another technique to hide memory access latency is the use of the local memory (shared memory). Since local memory has much lower latency than the global one and does not depend on coalesced operations to be efficient, its use can yield significant performance boost. This is especially the case when there is data reuse within a work-group.

Each device has its wavefront size, which is usually 32 for NVidia GPUs and 64 for AMD ones, i.e. if we do not want part of our wavefront idle, we need a work-group size multiple of the wavefront size. Since wavefronts are processed in lock-step, no flow deviations should occur within one. If a flow deviation occurs within a wavefront, the processing time will be equivalent to the whole wavefront processing all possible paths.

Here is a summary of what we believe to be the most significant optimization recommendations:

1. Select a work-group size multiple of the wavefront size to avoid idle work-items;
2. Avoid flow deviations within the wavefront;
3. When accessing global memory, use a pattern that yields a coalesced memory operation, e.g. adjacent work-items accessing adjacent memory addresses;
4. Use local memory to hide global memory access latency.

4. Implementations and Optimization

The main limitation of the obstacle detection method proposed by Talukder *et al.* is its computational cost. Even with the search area projection, the processing time of a point cloud can be too long for several applications. Therefore, the original method has been parallelized to efficiently utilize GPUs and CPUs.

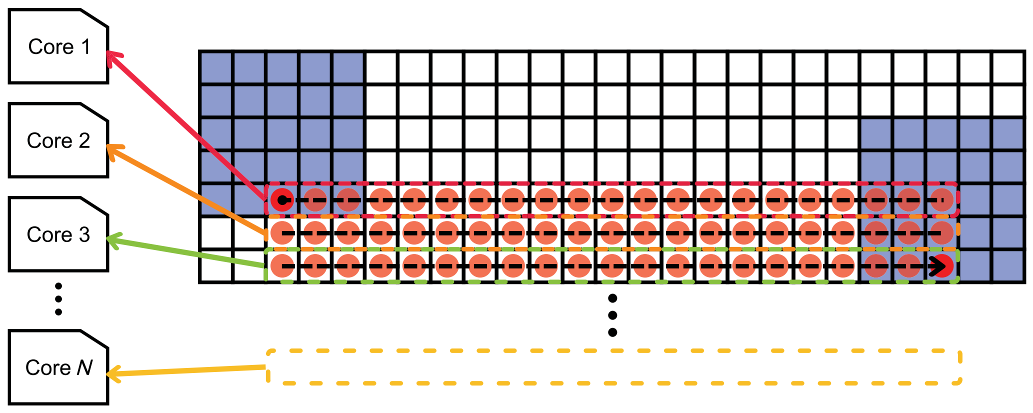

Fig. 6. Multi-core implementation. The pivot lines are processed in parallel by N CPU cores and the search areas of the first and last pivot are colored in light blue.

The computation of the target method employing projection is similar to the computation of an image convolution. Both cases involve a sliding window guided by a pivot point. In an image convolution the window is displaced around the pivot, i.e. the pivot is the center of the window, whereas in Talukder *et al.* method, the window is displaced above the pivot, as shown in Fig. 5. The range image is used only for indexing purposes and the computation itself is performed in its referent point cloud domain (i.e. 3D Euclidean space). A major difference is that image convolutions are usually performed with a fixed window size, whereas the target method uses a variable window size that depends on the z coordinate of the pivot. This difference is what makes a GPU implementation of Talukder *et al.* method challenging.

4.1. CPU implementations

We started by implementing the "full" version of the method, that is, without considering the projection. This implementation consists in checking the two conditions of the method for every pair of points, i.e. N^2 times, where N is the number of points. This implementation will serve as a reference to validate the others.

A natural scheme for implementing the "projected" version of the target method is the use of four aligned loops, which can also be employed for image convolution. The two outer loops correspond to the pivot line and column and the inner ones are related to the window. The ranges of the outer loops correspond to the data (point cloud) size and the ranges of the inner ones refer to the window size. We implemented the CPU single-core version of the method based on this idea and the window sizes were calculated by Eqs. (1) and (2).

The multi-core parallelization was performed with the *open multi-processing* (*OpenMP*)[1] library, which provides a directive that automatically parallelizes and distributes a loop between multiple cores using threads. The multi-core implementation was developed by inserting such directive before the most outer loop. This loop refers to the pivot lines. As a result, the lines will be processed in parallel using threads (Fig. 6). By default, *OpenMP* uses static scheduling to divide the workload (in this case, the pivot lines) across threads, i.e. the workload of each thread is determined before the loop execution. However, as each pivot line takes a different processing time to finish due to the dynamic nature of window sizes, some threads will finish their workload sooner than others, which results in idle threads and the under-use of the available processing. That being the case, we changed the schedule type for dynamic. Where the workload of each thread is determined on-the-fly: each new chuck (pivot lines) is assigned to a thread as its finishes its workload. The size of each chunk, i.e. the number of pivot lines assigned each time, can also be defined, but we chose to leave that option automatically determined by the library. Unless specified, the number of threads created by the *OpenMP* library is determined automatically based on the number of threads the CPU can execute in parallel (for the Intel Core i7-3770 this number is 8).

[1] http://www.openmp.org/

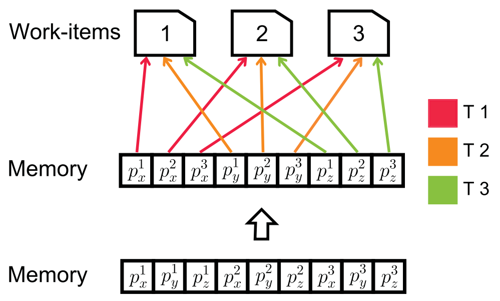
Work-items
1
2
3
T 1
T 2
T 3
Memory
p_x^1 p_x^2 p_x^3 p_y^1 p_y^2 p_y^3 p_z^1 p_z^2 p_z^3
Memory
p_x^1 p_y^1 p_z^1 p_x^2 p_y^2 p_z^2 p_x^3 p_y^3 p_z^3

Fig. 7. Redistribution of coordinates in the memory so that a wavefront accesses contiguous memory addresses (from array of structures to structure of arrays); the memory accesses are represented by arrows and colored according to the time of the event.

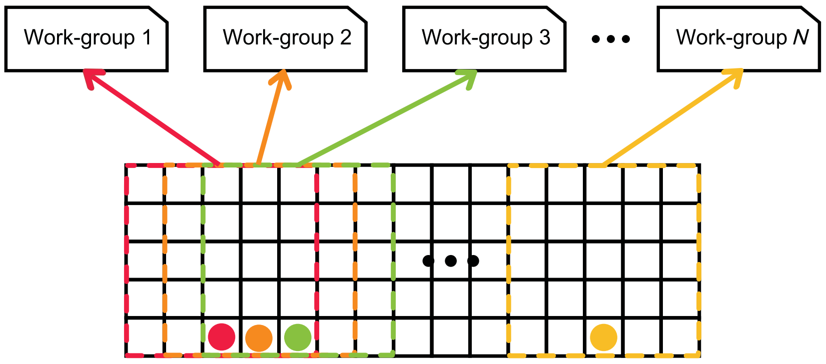

Fig. 8. Each search area is computed by a work-group.

The CPU implementations are based on *c++* language and compiled by *g++*[2] with general and CPU-specific (instruction sets and cache size) optimization flags. Unlike a convolution, the projected version does not require padded data; instead, the inner loops ranges can be calculated and limited (if they are larger than the image) as soon as the pivot has been fetched.

4.2. GPU implementations

The OpenCL specification was employed for the GPU-based implementation due to its interoperability across GPU manufacturers. The OpenCL specification comprises an application programming interface (API) and a programming language similar to the C language. The API is used to communicate with the GPU whereas the language is used to develop the kernel that effectively runs on the GPU.

In a GPU implementation, constraints and trade-offs must be dealt with, as GPUs rely on systematic branch/memory schemes to be efficient. We started with a simple kernel meeting most of this constraints, and increased its complexity as necessary. Three different kernels were implemented in the process.

Initially, we reorganized the data, as shown in Fig. 7. This change enabled a wavefront to access memory addresses sequentially. Unlike the CPU implementation, we padded the data with large negative values. To avoid copying the padding to the GPU every time, we write the padded buffer once and in later write calls we employ function `clEnqueueWriteBufferRect`, writing only useful data.

A simple approach to distribute the workload is to process a window (search area) per work-group (Fig. 8). Within a work-group each work-item could process a point, however, in practice, the maximum size of a work-group is easily reached: a search area of 20×20 results in a work-group with 400 work-items, above the 256 limit of some GPUs (e.g. AMD Radeon 5850).

This limit can be suppressed by increasing the workload of the work-items: instead of checking the compatibility of a point, each work-item checks the compatibility of a column of points. For

[2] http://gcc.gnu.org/

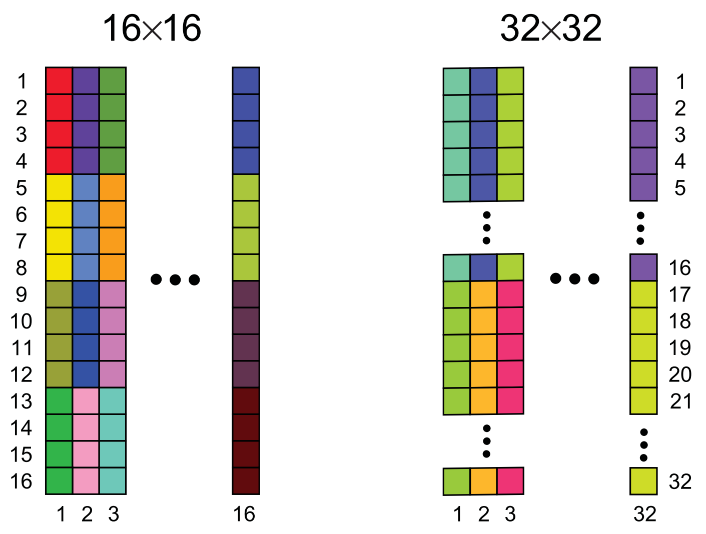

Fig. 9. Distribution of 16×16 and 32×32 search windows within a work-group of size 64. Each color represents the workload of a work-item.

example, in a window of 20×20 points, each work-item must check the compatibility of 20 points and the size of a work-group is reduced from 400 to 20.

A limitation of this approach is that the size of a work-group is fixed, but the window size is not. Moreover, the size of a wavefront in GPUs *HD79xx* series is 64, which means that a work-group smaller than 64 would result in a critical under-utilization of the GPU. A naive solution would be to use only windows of size 64×64 and ignore the proper window size, as given by Eqs. (1) and (2). The advantage of this approach is its simple kernel with few things to compute and only one fixed size `for` loop which can be unrolled. The whole wavefront also accesses memory addresses sequentially. However since the proper window sizes are ignored, in some cases one might expect that we have a larger window size than unnecessary, processing unnecessary points, and in other cases, smaller ones, missing compatible points. We will call this implementation *Kernel 1* for further references.

A partial solution to avoid this unnecessary workload and maintain a simple kernel has been the use of three fixed window sizes: 16, 32 and 64. The lateral size of each window would be computed according to Eq. (3), where s is the function that returns the side size of a window, h is from Eq. (1) and w from Eq. (2).

$$s(\mathbf{p}) = \begin{cases} 16 & \text{if} \quad max\{w(\mathbf{p}), h(\mathbf{p})\} \leq 16 \\ 32 & \text{if} \quad 16 < max\{w(\mathbf{p}), h(\mathbf{p})\} \leq 32 \\ 64 & \text{if} \quad 32 < max\{w(\mathbf{p}), h(\mathbf{p})\} \end{cases} . \tag{3}$$

This function is computed once for each work-group. Although the size of the window is variable, the work-group size of 64 was kept to match the wavefront size. In a 64×64 window, a work-item would compute the compatibility of 64^2 points. In this case, the initial scheme is maintained and each processor computes the compatibility of a column whose number of points is 64.

The points of windows sized 32×32 (32^2 points) and 16×16 (16^2 points) must be adequately distributed among a group of 64 work-items (Fig. 9). Each work-item processes 16 points within a 32×32 window and 4 points within one of 16×16. A fast way to determine the points that should be computed by a work-item is to use the “?” operator, since it does not cause flow divergence.[15] A noteworthy fact is that 64 points are processed in parallel, which is consistent with a wavefront of size 64. However, the whole wavefront is no longer accessing continuous memory addresses. We will call this implementation *Kernel 2* for further references.

Although this solution minimizes part of the unnecessary processing relative to the *Kernel 1* solution, it does not address cases in which $max\{w(\mathbf{p}), h(\mathbf{p})\} > 64$ and misses compatible points. For smaller window sizes, there are only three possible windows sizes; hence it is highly probable that it will process unnecessary points.

To address such cases, we consider Eqs. (1) and (2), but directly using them to calculate window sizes is not feasible and we must obey at least the following constraints: it should be possible to distribute the workload of a work-group equally among its work-items, i.e. $xy \bmod 64 = 0$, where x is the width of the window and y is its height. Another desirable constraint is that the workload of a work-item ($xy/64$) should be divisible by its height, or its height should be divisible by its workload, i.e. the workload can be processed by at most two `for` loops, which minimizes the kernel complexity. We formalized such constraints as the following optimization problem:

$$\begin{aligned}
\underset{x,y}{\arg\min} \quad & g(x,y) = (x - w(\mathbf{p}))^2 + (y - h(\mathbf{p}))^2 \\
\text{subject to} \quad & x \geq w(\mathbf{p}), \quad y \geq h(\mathbf{p}), \quad (x,y) \in \mathbb{N}, \\
& \frac{xy}{64} \bmod y = 0 \ \text{ OR } \ y \bmod \frac{xy}{64} = 0, \\
& xy \bmod 64 = 0,
\end{aligned} \tag{4}$$

where g is the function that calculates the distance between the calculated window sizes and the ones we are trying to approximate. The constraints are GPU-related and we believe they represent the minimal set so as to enable a reasonable kernel implementation.

To the best of our knowledge, this problem has no simple analytic solution but we can approximate it by rounding up $w(\mathbf{p})$ to the nearest multiple of 4 and y to the nearest number from set $\mathbf{M4} = \{16, 32, 64, 128, 192, 256\ldots\}$. We can calculate x and y using:

$$x = \left\lfloor \frac{w(\mathbf{p}) + 3}{4} \right\rfloor 4, \tag{5}$$

$$\begin{aligned}
\underset{y}{\arg\min} \quad & r(y) = (y - h(\mathbf{p}))^2 \\
\text{subject to} \quad & y \geq h(\mathbf{p}), \quad y \in \mathbf{M4}.
\end{aligned} \tag{6}$$

Although this is not the optimal solution, it is a good approximation and has a simple computational implementation. Since there is a natural limit to y (the range image height), we implemented Eq. (6) by using a look-up table, which can be stored in the constant memory of the GPU. We will call this implementation *Kernel 3* for further references.

An optimization step applied to all implementations is related to the pivot access; every work-item within a work-group must load the same pivot. According to ref. [15], the best way to carry this access pattern is to use only one work-item to load the pivot and share it with the others through the local memory. In common, all kernels must perform the following steps:

1. Obtain their work-group number and position within the work-group;
2. Calculate the pivot index;
3. If it is the first work-item: load the pivot into the local memory;
4. Synchronize work-items so that they can access the pivot;
5. Calculate the window size and the ranges for processing;
6. Use a "for" loop to process the designated points and save the results in the global memory.

A number of windows, work-items and work-groups combinations were not explored in this study and there is probably one that would yield lower processing times than the ones proposed. Nevertheless, Section 5.3 provides evidence that this is a reasonable implementation in absolute terms.

5. Results

Experiments were conducted using range images from a stereo camera and an RGB-D sensor (*Microsoft* Kinect). Two logs were captured for this purpose: one from an agricultural environment by a stereo camera and an indoor one by the Kinect sensor. We chose a resolution of 320×240 for

Table I. Talukder *et al.* method parameter selection for both sensors.

Parameter	Stereo	Kinect
H_T	27 cm	7 cm
H_{max}	40 cm	20 cm
θ_T	45 degrees	45 degrees
f	630 pixels	420 pixels

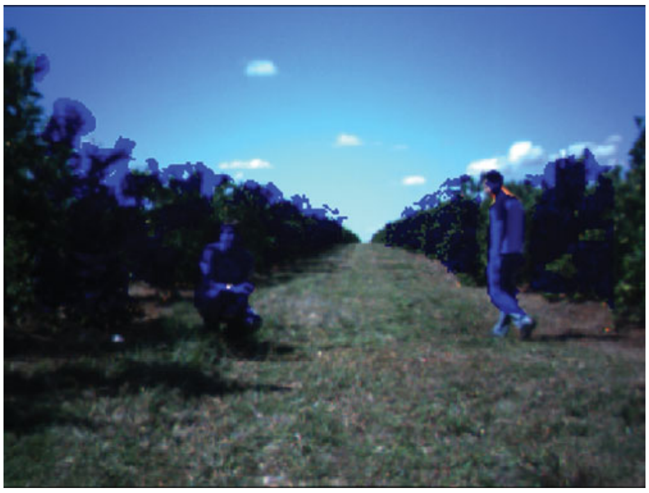

Fig. 10. Comparison between the results of the "full" CPU implementation and a projected one with a frame of the stereo log. Blue pixels represent obstacles detected by both implementations and red pixels are the ones detected only by the "full" CPU implementation.

the stereo camera and a semi-global stereo method[16] for stereo matching. The Kinect sensor tests were performed using its full resolution, i.e. 640×480. Table I shows the parameter selection for the method in each case. We employed the following hardware for experiments: Intel Core i7-3770 CPU, AMD Radeon 5850 GPU and AMD Radeon 7950 GPU. All tests were conducted in a Linux 32 bits environment.

5.1. *Validation*

We validated our implementations by comparing each one with the "full" CPU implementation. The objective was to assure that single-core, multi-core and *Kernel 3* implementations were correct, i.e. if they yield the same results as the "full" one, and also to highlight the possible effects of having a limited window size, as in the *Kernel 1* and *Kernel 2* implementations.

Figure 10 shows the comparison between the results of the "full" CPU implementation and a projected one in the stereo log. No obstacle was missed. The same holds for all (projected) GPU implementations, which revealed that, for this log, $max\{w(\mathbf{p}), h(\mathbf{p})\} \leq 64$ and, in cases in which $max\{w(\mathbf{p}), h(\mathbf{p})\} > 64$, no obstacles were found. The results in Fig. 10 were the same throughout the stereo log.

The same validation was performed using the Kinect log. Two relevant differences between this log and the stereo one: (i) the indoor environment tends to yield larger window sizes since the points are closer to the sensor; and (ii) the sensor is clearly tilted towards the ground plane, therefore the "method assumption" is violated. The GPU implementations *Kernel 1* and *Kernel 2* have a limited window size and the first difference causes them to miss near obstacles (Fig. 11). The proper projected implementations (i.e. single-core, multi-core and *Kernel 3*) produced the same results as the "full" CPU implementation during the log, i.e. the same obstacles were found in both cases. Therefore, we can conclude that the proper projected implementations are correctly implemented, and that using projection sizes larger than the ones from Eqs. (1) and (2) do not alleviate the "method assumption". The later can be inferred because the "full" implementation uses the maximum possible window size, i.e. the image size. These results were expected from the discussion in Section 2.

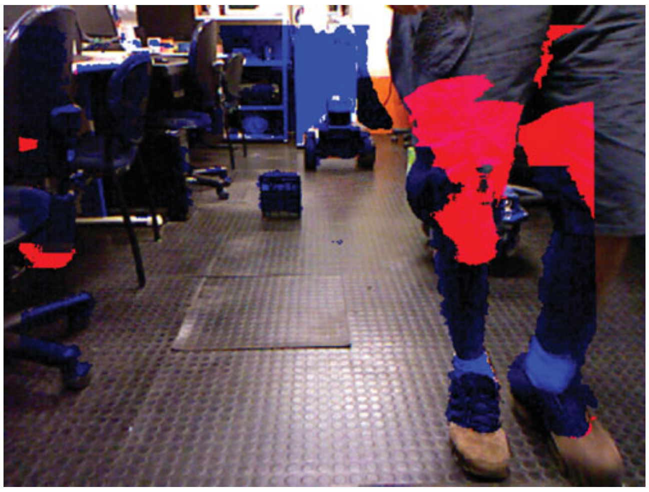
Fig. 11. Comparison between the results of the “full” CPU implementation and GPU *Kernel 1* one with a frame of the kinect log. Blue pixels represent obstacles detected by both implementations and red pixels are the ones detected only by the “full” CPU implementation. This is not the case of the proper projected implementations (i.e. single-core, multi-core and *Kernel 3*).

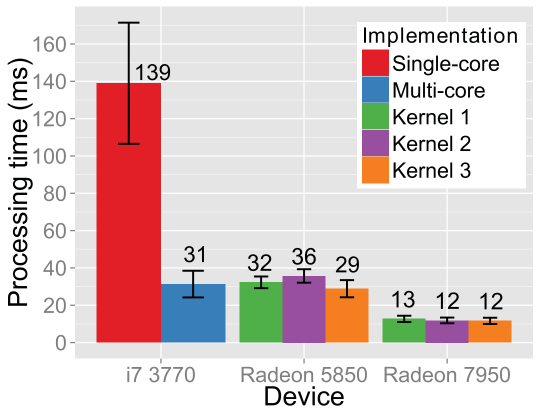

Fig. 12. Mean processing times for each device and implementation obtained using the stereo log. Only the projected CPU implementations are presented.

5.2. *Processing times*

This section compares the processing times of each implementation. The results were calculated by running each log with each implementation. Since the processing times are valid only for our setup, we also provided processing times relative to the number of points processed. This is important because even with the same sensor and resolution, the density of the point cloud may change. This change is especially significant in stereo cameras, where both the stereo method and the environment affect the number of valid depth measurements.

A bar chart displaying the mean processing times for each implementation on the stereo log in shown in Fig. 12 and Table IV provides further details. The multi-core implementation is 4.5 times faster than the single-core one, which has revealed the benefit of simultaneous multithreading (SMT) in the tested CPU. Concerning GPU implementations, there was a significant difference between the two tested GPUs, but not between kernels.

Table IV and Fig. 13 show that the processing times were higher for the Kinect sensor. There are two main reasons for the result: the point cloud is four times larger and the indoor environment tends to have closer points (smaller z) and generate larger search areas (window sizes). The CPU single-core implementation was almost 27 times slower than the equivalent one from stereo data while the 7950 *Kernel 3* GPU implementation scaled much better, i.e. only 3.6 times slower, and provided a speedup

Table II. Mean number of points per ms computed by each device.

Device	Points per ms	Speedup
CPU Single-core	2.72×10^5	1
CPU Multi-core	1.42×10^6	5.2
Radeon 5850	2.70×10^6	9.9
Radeon 7950	1.95×10^7	71.9

The GPU devices refer to *Kernel 3* implementation.

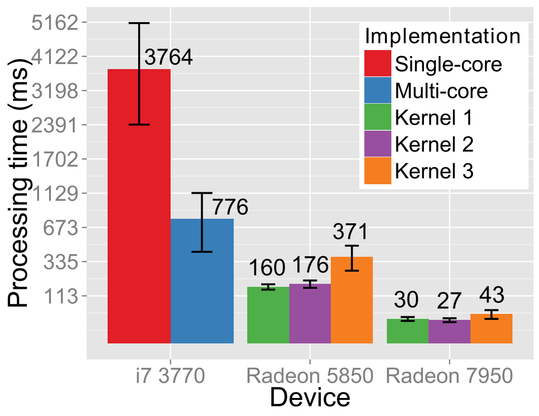

Fig. 13. Squared scaled mean processing times for each device and implementation obtained using the kinect log. Only the projected CPU implementations are presented.

of 87 relative to single-core. A larger gap was observed between the two tested GPUs and, again, no significant difference was found between kernels.

We finally provide a setup independent measure of computational cost, showing, for each device, the number of points and processing times in Fig. 14, where each dot of the graph represents a frame and the number of points of a frame was calculated according to:

$$np = \sum_{\mathbf{p} \in V} w(\mathbf{p}) h(\mathbf{p}), \tag{7}$$

where V is the set containing all valid pivot points. Table II shows the number of points processed per millisecond (ms), calculated using the mean points per ms of each frame. Both table and chart show only the implementations that yielded the same results of the "full" one, i.e. single-core, multi-core and *Kernel 3*. Such data provide setup independent results, which can serve as a reference for other researchers in the field to check if our implementation and devices can attend their demand.

5.3. Kernel analysis

While processing times alone enable the assessment of relative speedups, they provide few clues on how efficient the implemented kernels are in absolute terms. Although we started attending some of the GPU performance recommendations in *Kernel 1*, this was not the case in subsequent kernels. Coalesced memory operations were compromised as *Kernel 2* and *Kernel 3* do not necessarily access adjacent memory address within a wavefront, which may result in idle arithmetic logic units (ALUs) while work-items wait fetch operations.

AMP APP Profiler enabled the gathering of information on the performance of each kernel. It organizes this information in "performance counters", which reveal a different aspect of the kernel. Our focus is on two of them:

VALUBusy: Percentage of GPU time spent in vector ALU operations;
MemUnitBusy: Percentage of GPU time the memory unit is active.

Table III. Performance counters for each kernel in the Kinect log with the AMD Radeon 7950.

Kernel	Counter	Value in %
Kernel 1	VALUBusy	82.06
	MemUnitBusy	95.10
Kernel 2	VALUBusy	82.98
	MemUnitBusy	94.18
Kernel 3	VALUBusy	80.14
	MemUnitBusy	91.20

These counters can reveal whether a kernel is CPU or memory bounded. We tested only the AMD Radeon 7950 in the Kinect log since they generally apply to our kernels and this GPU is the only that provides such counters.

Table III shows the performance counter for each kernel in the Kinect log with AMD Radeon 7950. There is little room for memory access improvements as the ALUs are kept significantly busy, i.e. over 80% in all cases, and no significant variation in the counters across kernels, probably because although memory accesses are less efficient in *Kernel 2* and *Kernel 3*, their ALU/fetch instruction ratio is higher, i.e. they have more ALU instructions. These instructions hide the latency of inefficient fetches.

6. Discussion

The GPU implementation has successfully provided a speedup for the method and enabled real-time obstacle detection for both sensors and scenarios. We were especially interested in cases where a single-core implementation provides a prohibitive computational time, as in our Kinect setup. Fortunately our implementation could accelerate the processing frequency from ~0.26 Hz (single-core) to ~23 Hz (*Kernel 3* and Radeon 7950). Concerning the stereo setup, the small granularity (window sizes) and amount of data penalized the GPU implementations.

A crucial aspect of our approach is that no approximations were employed, differently from other works. In ref. [10], the authors used a conditional reduction in the point cloud resolution, in which sub-sampling is performed until an obstacle is found. A similar approach was used in ref. [8], where the image is divided into horizontal segments and sub-sampled according to the distance represented by such segments. In,[9] a set of discrete parameters was employed. Although these approaches are justifiable for noisy and low resolution range images (e.g. the ones from our stereo setup), they assume that obstacles are formed by a reasonable set of points, which is not necessarily true, can conceal obstacles and reduce the detection accuracy. This is especially the case of relative high-resolution range images, where the sub-sampling rate should be high, and on precise sensors, that provide pixel-wise reliable measurements.

Although computation times are not directly comparable due to different hardware, parameters and test scenarios, we could achieve the second lowest computational time stated in the literature without using approximations (Table V). It is also worth mentioning that the approximation techniques employed by other authors can be used in conjunction with our approach and yield even higher speedups.

7. Conclusions and Future Work

This paper has presented a GPU-based parallel version of an obstacle detection method. Proposed in ref. [7], the method has been widely used, however researchers have to deal with its main limitation, its computational cost.

One of the main challenges to adapt code for parallel execution on a GPU is the need of a proper distribution of the workload among multiple processors/compute units. The memory access is also a critical point and must be performed carefully to avoid potential bottlenecks. Our target algorithm requires the use of dynamic-sized windows, which makes parallelization even more complex.

Table IV. Processing times for each different combination of log, device and implementation.

Log	Device	Implementation	Mean time	Max. time	Min. time	Speedup
Stereo	i7 3770	Single-core	139 ms	240 ms	105 ms	1
		Multi-core	31 ms	54 ms	21 ms	4.5
	Radeon 5850	*Kernel 1*	32 ms	41 ms	23 ms	4.5
		Kernel 2	36 ms	46 ms	27 ms	3.9
		Kernel 3	29 ms	42 ms	24 ms	4.8
	Radeon 7950	*Kernel 1*	13 ms	19 ms	9 ms	10.7
		Kernel 2	12 ms	18 ms	8 ms	11.6
		Kernel 3	12 ms	17 ms	8 ms	**11.6**
Kinect	i7 3770	Single-core	3764 ms	9487 ms	1479 ms	1
		Multi-core	776 ms	2230 ms	287 ms	4.8
	Radeon 5850	*Kernel 1*	160 ms	179 ms	98 ms	23.5
		Kernel 2	176 ms	199 ms	82 ms	21.4
		Kernel 3	371 ms	798 ms	176 ms	10.1
	Radeon 7950	*Kernel 1*	30 ms	53 ms	20 ms	125.5
		Kernel 2	27 ms	42 ms	16 ms	139.4
		Kernel 3	43 ms	93 ms	22 ms	**87.5**

Speedup relative to the single-core implementation of each log.

Table V. Comparison of computation times found in the literature.

Approach	Resolution	Proc. time
Sub-sampling[10]	640×480	200 ms
Sub-sampling[8]	500×320	25 ms
Discrete parameters[9]	640×480	62 ms
GPU (Ours)	640×480	43 ms

The processing times are not directly comparable due to variations in hardware, environment and parameters; therefore it should serve only as a reference.

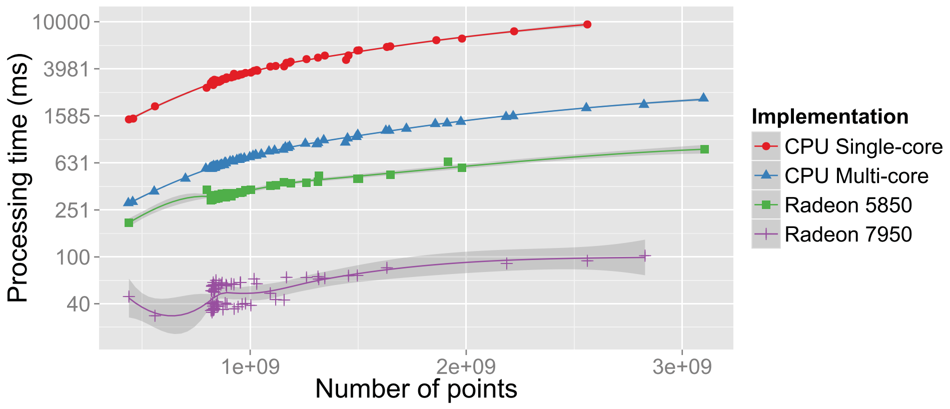

Fig. 14. Logarithmic scaled processing times relative to the number of points for each device. The GPU devices refer to *Kernel 3* implementation.

We formalized the GPU constraints as an optimization problem and derived an approximation for it. Satisfactory results were achieved, i.e. the *Kernel 3* implementation was almost 90 times faster than the single-core one with the use of Radeon 7950 in the Kinect log. An important aspect of the GPU option is that the CPU becomes available to perform other tasks and responsible only for reading and writing buffers.

The proposed solution for processing dynamically-sized sliding windows on GPU can be seen as a general framework and applied to other tasks that require similar computation, benefiting a wide range of applications.

As future work, we aim at using our implementation along with a clustering method and a filter based on machine learning techniques to minimize noise and improve the classification results.

References

1. R. Hadsell, P. Sermanet, J. Ben, A. Erkan, M. Scoffier, K. Kavukcuoglu, U. Muller and Y. LeCun, "Learning long-range vision for autonomous off-road driving," *J. Field Robot.* **26**(2), 120–144 (2009).
2. K. Konolige, M. Agrawal, M. R. Blas, R. C. Bolles, B. Gerkey, J. Solà and A. Sundaresan, "Mapping, navigation, and learning for off-road traversal," *J. Field Robot.* **26**(1), 88–113.
3. A. Broggi, C. Caraffi, R. Fedriga and P. Grisleri, "Obstacle Detection with Stereo Vision for Off-Road Vehicle Navigation," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition – Workshops*, San Diego, California, USA (2005) p. 65.
4. C. Caraffi, S. Cattani and P. Grisleri, "Off-road path and obstacle detection using decision networks and stereo vision," *IEEE Trans. Intell. Transp. Syst.* **8**(4), 607–618 (2007).
5. J. Kolter, M. Rodgers and A. Ng, "A Control Architecture for Quadruped Locomotion Over Rough Terrain," *IEEE International Conference on Robotics and Automation*, Pasadena, California, USA (2008) pp. 811–818.
6. S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb and R. Chatila, "Autonomous Rover Navigation on Unknown Terrains Functions and Integration," **In**:*Experimental Robotics VII*, Lecture Notes in Control and Information Sciences, vol. 271 (Springer, Berlin Heidelberg, 2001) pp. 501–510.
7. A. Talukder, R. Manduchi, A. Rankin and L. Matthies, "Fast and Reliable Obstacle Detection and Segmentation for Cross-Country Navigation," *IEEE Intelligent Vehicles Symposium*, Versailles, France (2002) pp. 610–618.
8. A. Broggi, M. Buzzoni, M. Felisa and P. Zani, "Stereo Obstacle Detection in Challenging Environments: The viac Experience," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, California, USA (2011) pp. 1599–1604.
9. W. van der Mark, J. van den Heuvel and F. Groen, "Stereo Based Obstacle Detection with Uncertainty in Rough Terrain," *IEEE Intelligent Vehicles Symposium*, Istanbul, Turkey (2007) pp. 1005–1012.
10. P. Santana, P. Santos, L. Correia and J. Barata, "Cross-Country Obstacle Detection: Space-Variant Resolution and Outliers Removal," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nice, France (2008) pp. 1836–1841.
11. C. C. T. Mendes, F. S. Osorio and D. F. Wolf, "An Efficient Obstacle Detection Approach for Organized Point Clouds," *IEEE Intelligent Vehicles Symposium (IV)*, Gold Coast City, Australia (2013) pp. 1203–1208. Available at: http://dx.doi.org/10.1109/IVS.2013.6629630 doi:10.1109/IVS.2013.6629630.
12. R. Manduchi, A. Castano, A. Talukder and L. Matthies, "Obstacle detection and terrain classification for autonomous off-road navigation," *Auton. Robots* **18**(1), 81–102 (2005).
13. NVIDIA, OpenCL Programming Guide for the CUDA Architecture (May 2010). Available at: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf
14. S. Xiao and W. chun Feng, "Inter-Block GPU Communication Via Fast Barrier Synchronization," *IEEE International Symposium on Parallel Distributed Processing*, Atlanta, Georgia, USA (2010) pp. 1–12.
15. AMD, AMD Accelerated Parallel Processing OpenCL Programming Guide (Jul 2013). Available at: http://developer.amd.com/tools/hc/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf.
16. H. Hirschmuller, "Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2, San Diego, California, USA (2005) pp. 807–814.