Journal of Computational Science 79 (2024) 102317

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

CPU and GPU oriented optimizations for LiDAR data processing

Felipe Muñoz [a], Rafael Asenjo [a,*], Angeles Navarro [a], J. Carlos Cabaleiro [b]

[a] Department of Computer Architecture, Universidad de Málaga, Bulevar Louis Pasteur, 35, Málaga, Spain
[b] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Dpto. de Electrónica y Computación, Universidade de Santiago de Compostela, Santiago de Compostela, Spain

ARTICLE INFO

Dataset link: https://github.com/PPMC-DAC/FastOWM

Keywords:
LiDAR data processing
Digital Terrain Model
Tree data structures
Parallel optimization
GPU
SYCL
CUDA
oneAPI

ABSTRACT

Digital Terrain Models (DTM) can be accurately obtained from clouds of LiDAR points but the corresponding cloud processing time can be prohibitive. This paper describes several optimization techniques that have been applied to the Overlap Window Method (OWM) that is a key component in DTM applications. OWM was originally implemented in R which translates into serious limitations in terms of the size of the LiDAR point cloud that can be processed. We have ported the code to C++, significantly optimized the data structure to minimize memory accesses, and developed parallel implementations for CPU and GPU commodity devices using oneAPI libraries and tools. This results in CPU and GPU versions that are up to 19x and 83x faster, respectively, than an OpenMP baseline that uses eight CPU cores. Most importantly, the proposed optimizations for CPU and GPU can be paramount to get the most out of other LiDAR-based algorithms in which the careful selection of the right data structure, parallelization strategies and memory access reduction techniques will certainly result in significant performance improvements.

1. Motivation and summary

LiDAR (*Light Detection And Ranging*) is an active sensing technique capable of measuring distance between the sensor device and the target object [1] that has attracted the attention of the scientific community, as well as public administrations and private companies. LiDAR has become an established method for collecting accurate information of the Earth's landscape. The record of elevation data and the ability to penetrate through the canopy are some of the advantages over traditional acquisition methods like aerial imagery. The required equipment consists of an *Airborne Laser Scanner* (ALS), an *Inertial Measurement Unit* (IMU) and a GPS receiver. These devices are installed in an aircraft that flies over the ground surface obtaining a georeferenced point cloud. On land a GPS system is synchronized with the GPS receiver on the aircraft. For each point, a set of features are recorded: (x, y) coordinates, elevation (z), intensity, pulse id, number of pulses, and scan angle. Photogrammetry is another sensing technique, which obtains point clouds from aerial images. The density of the point clouds is usually higher, because it can be configured by software, but the accuracy is worse, particularly in flat areas and wooded regions, due to the impossibility of penetrating the tree canopy and the need to establish ground control points, which are essential for providing the georeferencing parameters and suppressing undesirable error propagation [1,2]. Due to the huge size of the collected information with these techniques, and the ubiquity of heterogeneous parallel computing systems nowadays, the optimization and parallelization of the algorithms that process LiDAR point clouds are paramount in order to achieve efficient solutions for these architectures.

Computing the *Digital Terrain Model* (DTM) from ALS point clouds is one of the first steps to automatically obtain relevant information from the surface that has been scanned using LiDAR technology, and DTMs are becoming standard products available from national geoportals and private companies. The first step for DTM computation is the ground point extraction from the point cloud, i.e. the identification of the points in the cloud that correspond to the ground surface. After that, an interpolation step to derive the DTM is needed [1]. In this work, we focus on a recent alternative in the process of ground point extraction, the *Overlap Window Method* (OWM) [3]. This algorithm was initially implemented in R, which resulted in that only small point clouds could be analyzed (less than 50k points in [3]). In order to cope with larger datasets, we have ported the code to C++ and optimized it to make the most of the current heterogeneous parallel architectures, featuring multicore CPUs and GPUs. We believe our optimizations are also applicable to other algorithms for processing LiDAR data point clouds because the data structures to represent this type of point clouds and their traversal are similar to the ones implemented in our work.

oneAPI [4] is a promising framework that simplifies programming heterogeneous parallel architectures by providing several libraries, profiling and analyzing tools, but most importantly a core programming

* Corresponding author.
E-mail addresses: felipeml@uma.es (F. Muñoz), asenjo@uma.es (R. Asenjo), angeles@ac.uma.es (A. Navarro), jc.cabaleiro@usc.es (J.C. Cabaleiro).

https://doi.org/10.1016/j.jocs.2024.102317
Received 11 May 2023; Received in revised form 8 April 2024; Accepted 30 April 2024
Available online 6 May 2024

language named Data Parallel C++ (DPC++) [5]. DPC++ is based on ISO C++ and the Khronos Group's SYCL standard programming language. SYCL [6] is a royalty-free, cross-platform abstraction layer that allows software developers to write a single code and compile it for a collection of heterogeneous parallel processors ("write once, run everywhere"). Currently, oneAPI's SYCL compiler is able to target multicore CPUs, GPUs and FPGAs. SYCL 2020 is the latest version of the standard and provides support for *Unified Shared Memory* (USM), ordered queues, reductions, hierarchical parallelism, subgroups (on CPU and GPU implementations), and data flow pipes (for FPGAs). The main benefit of using SYCL over other languages for heterogeneous programming, such as OpenCL or CUDA, is the portability of a single source code able to target multiple devices, which translates into cleaner and more easy to maintain code. Although oneAPI's SYCL compiler was initially devised to target Intel CPUs, GPUs and FPGAs, as an open source project, new backends have been added to also target architectures from other vendors. Among them, NVIDIA discrete GPUs, dGPUs, can be exploited, either by instructing the SYCL compiler to generate code for the dGPU or by embedding CUDA code inside SYCL code. In this work we evaluate these programming models and their impact in performance in a platform based on commodity CPU and GPU devices.

Summarizing, this paper covers the following contributions:

- We present several optimized parallel implementations of the OWM algorithm in C++, SYCL and CUDA, which are able to process point clouds of several million points on commodity devices (CPUs and GPUs) in less than a second.
- We discuss the tradeoffs between different data structures to store the LiDAR points (binary tree, quadtree and octree) as well as different techniques inherited from *branch-and-bound* and *ray tracing* algorithms to optimize the construction and traversal of the tree. Our findings are of interest to other applications that process LiDAR data point clouds.
- We contribute with parallel optimizations for CPU and GPU towards reducing memory bandwidth requirements which is key for the data structure construction and traversal in LiDAR memory bound problems.
- We compare SYCL, SYCL+CUDA and pure CUDA programming models in terms of performance and portability on our target platform, and provide the source code and results of all our implementations.[1]

The rest of the paper is organized as follows. Section 2 briefly introduces the OWM problem, suitable data structures, the testbed and the baseline performance that we strive to beat. Sections 3 and 4 describe CPU and GPU oriented optimizations, respectively. The experimental evaluation is presented in Section 5 to finally conclude in Section 6 summarizing the main take-home messages.

2. Background

As commented before, the DTM is one of the most important products to be obtained from 3D point clouds from LiDAR and photogrammetry. A DTM is a continuous function that maps from 2D planimetric position to terrain elevation $z = f(x, y)$ [7]. This function is stored digitally, together with a method on how to evaluate it from the stored geometrical and topological entities. DTM refers to the bare earth surface, without man-made or natural objects such as buildings and trees. Although DTM generation methods vary, most of them share several main steps (though conducted in different orders): data preprocessing, ground point filtering (the ground point extraction from the point cloud) and interpolation to derive the DTM. In [8] the general principles of DTM generation are introduced and diverse mainstream DTM generation methods are reviewed.

Although several ground filtering methods have been proposed based on different techniques [7], it is difficult to find one particular method that satisfies the requirements of every type of terrain. The OWM proposed in [3] is one of the most accurate filters in different scenarios that has been tested with the reference samples from the International Society of Photogrammetry and Remote Sensing (ISPRS), as is shown in the analysis performed in the mentioned work.

However, the main problem of this algorithm is its low computational performance, since it was initially programmed in R. In a previous work, the code was ported to C, demonstrating a good performance of the algorithm even without parallelization. It was also parallelized using OpenMP, achieving a considerable improvement in processing time, but high execution times (>25 s.) were still obtained when using dense clouds with a large number of points (>20 Mpts and >7 pts/m^2). In this work we propose to conduct a complete study of the algorithm, characterizing the bottlenecks and seeking solutions to overcome them, as well as leveraging current heterogeneous programming models like CUDA and SYCL for exploiting GPU and shared memory architectures.

Some works about the implementation in GPU of algorithms processing LiDAR data point clouds can be found in the literature. In [9] the Semi-Global Filtering (SGF) to classify points as either ground or non-ground is proposed. The authors claim that the SGF algorithm offers high classification accuracy and also it was parallelized using a GPU processing approximately 3 million points per second. In [10] a scan-line-based algorithm to detect local lowest points first and treat them as the seeds to grow into ground segments in presented. This algorithm was implemented in GPUs by processing each scan line independently. The test results show that the proposed algorithm is fast and effective in urban areas, but does not work well in mountainous areas. An efficient method for ground filtering of airborne LiDAR data based on scan-line processing is proposed in [11]. The algorithm was ported into a low-cost development board to demonstrate its feasibility to run in embedded systems, where throughput was improved by using programmable logic hardware acceleration. Analysis shows that real-time filtering is possible in a high-end board prototype, as it can process the amount of points per second that current lightweight scanners acquire with low-energy consumption.

Related to obtaining the DTM, but using satellite instead of LiDAR, a procedure for building a DTM from the satellite stereo images is proposed in [12]. The procedure generates a DTM that may be less accurate than the one achieved with the use of the ENVI software, but it offers a significantly shorter processing time. In [13] an algorithm for the generation of DSMs (Digital Surface Models) from satellite images was implemented on a GPU, increasing performance a 900% while having a negligible decrease in accuracy.

Our proposal is to optimize the OWM function, which is the most computationally intensive part of a reliable DTM algorithm proposed in [3], and to port it to GPU. More precisely, we present data structure and memory oriented optimizations that can be applied to other algorithms that process LiDAR point clouds. The resulting CPU and GPU optimized versions can process up to 68 and 285 million LiDAR points per second, respectively, which to the best of our knowledge is a performance not seen before in the context of robust ground surface detection algorithms.

2.1. The algorithm: OWM

The OWM function is part of the *Hybrid Overlap Filter* (HyOF) application [3], a hybrid algorithm for obtaining the DTM consisting of three processing blocks that are executed sequentially, from which the OWM is by far the most computationally intensive (consuming more than 80% of the total execution time). For this reason, in this paper we

[1] Github repository: https://github.com/PPMC-DAC/FastOWM.

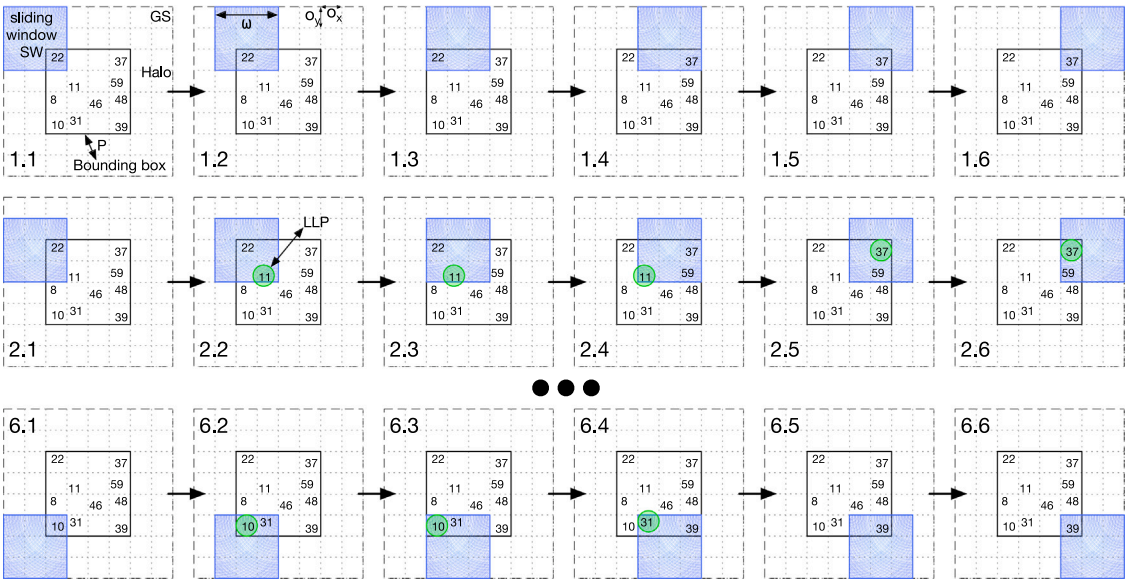


Fig. 1. OWM process to select LLPs (identified by a green circle). The point cloud, P, is depicted by the numbers (the z coordinate of each point). Only 18 sweeping steps out of the required 36 are depicted.

tackle the problem of optimizing the OWM function, but please refer to [3] for more information about the other processing blocks.

As commented in Section 1, the OWM function aims at defining an initial reference surface from which to obtain the DTM, starting from a LiDAR point cloud. This is achieved by adjusting some parameters so that, in the successive iterations or steps, the points of the cloud that do not belong to the ground (P_{ng}; *non-ground point*; points that may belong to a rooftop, the top of a tree, etc.) are filtered out and the seed-points (P_S; *seed-points*, the key points that represent the ground surface preserving all the details of the relief) are collected, and from them the initial reference surface ($\varphi_{t=0}$) can be generated.

OWM is defined by 5 arguments: input data (P; *Point Cloud*); Bounding box (x_{min}, y_{min}, x_{max}, y_{max}) defined by the maximum and minimum values of the (x, y) coordinates of the points in P; sliding window size (ω) for the selection of the *Local Lowest Point* (LLP); and horizontal (O_x) and vertical (O_y) overlaps between two consecutive windows. Both overlaps, O_x and O_y are expressed as a fraction of the window size so that we can define the overlap distances, $o_x = (1-O_x)\cdot\omega$ and $o_y = (1 - O_y) \cdot \omega$.

Fig. 1 illustrates the OWM function for a toy example. We assume a point cloud P comprising just 10 points with coordinates (x, y, z) confined in a bounding box. In the figure we depict each point by the value of its z coordinate (point altitude) at approximated (x, y) coordinates. To avoid an edge effect in the LLP selection, the area to be scanned is enlarged by $\omega \cdot O_x$ meters on both sides of the X dimension and $\omega \cdot O_y$ meters on both sides of the Y dimension. As a result, an empty halo surrounds the bounding box and the extended area is referred to as the "Geographic scope" (GS). In our example, $O_x = O_y = 2/3$ so that $o_x = o_y = \omega/3$ and GS becomes a grid of $o_x \times o_y$ boxes. The bounding box is a $4 \cdot o_x \times 4 \cdot o_y$ square, and GS dimensions are $8 \cdot o_x \times 8 \cdot o_y$.

With that, the OWM process goes by sweeping the whole geographic scope, GS, in a number of steps in which the sliding window, SW, scrolls in steps of o_x or o_y, so the overlapping area of the SW when moving in the x direction is $O_x \cdot w^2$. In Fig. 1 we show only the first 12 and the last 6 steps out of the total 36 steps that are required to cover the whole GS area. At each step, the point with the lowest elevation inside the sliding window, SW, is selected as an LLP (Local Lowest Point, and depicted inside a green circle in the figure) if the number of points under the SW is larger than $minNumPoints$. In this example we set the threshold $minNumPoints = 1$. Note that each $o_x \times o_y$ cell falls inside the sliding windows a number of times (9 times in our example and in general $1/(1 - O_x) \times 1/(1 - O_y)$ times). This means that a local minimum can be selected as LLP more than once, and up to 9 times in our running example. Clearly, the more times a point is selected as LLP, the higher the probability that it belongs to the ground (P_g; *ground point*). Although not explicitly shown in the figure, it is an easy exercise to validate that the point with $z = 11$ is selected 5 times, whereas the points with $z = 22$ and $z = 46$ are never selected (never surrounded by a large enough number of larger values). Therefore, the point with $z = 11$ is more likely to be a ground point than the point with $z = 46$. After 36 steps, points with z equal to 11, 37, 8, 39, 31 and 10 are found as representative minimums or LLPs, 5, 4, 9, 4, 2 and 2 times, respectively. On the other hand, points with z equal to 22, 59, 46 and 48 are never found as LLP inside a dense enough sliding window. In [3] it is demonstrated that the best heuristic consists in considering as a seed-point of the ground surface, $\varphi_{t=0}$, only the minimums that have been found more than once, since they are the most "reliable" minimums. Therefore, in our example the set of ground seed-points is $P_S = \{11, 37, 8, 39, 31, 10\}$ (using their z value as their ids).

Algorithm 1 OWM traversal phase

Require: P, ω, O_x, O_y. ▷ Point cloud, window size, horizontal and vertical overlaps
1: $nRows, nCols = ComputeSteps(P, \omega, O_x, O_y)$
2: $countLPP = 0$
3: **for** $i = 0, nRows$ **do**
4: **for** $j = 0, nCols$ **do**
5: $SW\_Center = ComputeSlidingWindowCenter(P, \omega, O_x, O_y, i, j)$
6: $coveredPoints = search(P, \omega, SW\_Center)$
7: **if** $coveredPoints.size() > minNumPoints$ **then**
8: $LPPid = Min(coveredPoints)$
9: $LPPsFound[countLPP] = LPPid$
10: $countLPP++$
11: **end if**
12: **end for**
13: **end for**
14: $P_S = ComputeSeedPoints(LLPsFound)$ ▷ Less than 0.07% of total execution time
15: **return** $P_S = P_S \cup FillEmptyCells(P, B, GroundSeedPoints)$ ▷ Less than 0.57% of total execution time

In our implementation we consider two phases in the OWM function: a data structure construction phase and a OWM traversal phase. The first phase is the input of the second one, which represents the main functionality of OWM. Algorithm 1 shows the pseudocode of the OWM

traversal phase. The function receives as input the point cloud P, the sliding window size ω, and the horizontal and vertical overlaps O_x and O_y. The code computes first the number of steps required to cover the whole point cloud area (in negligible execution time) and then iterates over all the steps. At each step, the center of the sliding window is computed (line 5), and the points that fall inside this window are stored in the set $coveredPoints$ (line 6). If the number of covered points is large enough (greater than $minNumPoints$) the id of the point with the lowest z value in this set is found and stored in the array $LLPsFound$. The value of $minNumPoints$ is computed as $P.density \times \omega^2/2$ and therefore an LLP is the minimum z-value of a sufficiently dense window. After the loop nesting (line 14), the set of seed-points of the ground surface, P_S, is computed from the array $LLPsFound$ by identifying the LLPs selected more than once. This function takes less than 0.07% of the total execution time for large point clouds since it only needs to sort the $LLPsFound$ array and iterate it by comparing consecutive entries.

In high-slope or very sparse zones, it is unlikely that the same point will be identified as an LLP more than once, and the resulting seed-point cloud will have some zones without points. In order to minimize the lack of detail in these zones, a grid of square cells of size $B \times B$ is overlapped to the geographical scope (see [3]). For those cells without points, the lowest point is selected from the original point cloud and is added directly to the seed-point cloud achieving the final OWM result, as can be seen in line 15 of Algorithm 1. In our experiments (see later), this phase takes less than 0.57% of the total execution time. Other steps to construct the final ground surface out of the seed-point cloud are described in [3], but they are not considered here since they represent a negligible amount of execution time and do not require any optimization strategy.

The loop nesting starting at line 3 of Algorithm 1 is the most time-consuming part of the OWM traversal phase taking more than 99.4% of the total execution time. A first optimization consists in parallelizing the outer loop using OpenMP adding a parallel directive to the "i" loop, with the `schedule(dynamic)` clause (to account for load unbalance when traversing the point cloud data structure) and a critical section protecting lines 9 and 10. This results in a 2-6x speedup in our 8-core platform. Other OpenMP strategies could be implemented in order to improve the parallel performance. For instance, we used the "collapse" clause to linearize the loop nesting and explored different scheduling policies: static, guided, and dynamic. Moreover, for dynamic we looked at different scheduler granularities (chunk sizes), from 1 to 14. Using "colapse" and the best chunk size we got OWM traversals that were ≈24% faster for dense clouds but ≈17% slower for sparse ones. With these findings, we will consider the OMP version without "collpase" and dynamic scheduler with chunk size of 1 as our baseline because it is representative enough of a reasonable OpenMP implementation that an average developer would consider acceptable. The corresponding traversal parallel times are shown in Section 2.5.

2.2. The data structure: trees

The point cloud, either stored in a file or arriving as a stream from a LiDAR device, is unsorted, and there is not a clear sorting criterion. However, as described in the previous section, at each step of the OWM algorithm we have to find the point with the minimum value of z from the set of points for which the two other coordinates, (x, y), fall into the corresponding sliding window. Therefore, to optimize each OWM step it is required to efficiently access and identify the points inside a sliding window. This is a memory-bound problem, so data locality should be exploited, thus it is a must to keep close in the data structure the points that are geographically close.

To solve this issue, hierarchical data structures are widely used in fields like graphics, image processing, geographical information systems, and robotics among others. These hierarchical data structures are based on the divide and conquer principle and on keeping related data close in the data structure. At the top of the data structure we have a handle that gives access to all the elements stored. Using recursive decomposition, lower levels of the data structure store fewer elements, which tend to be more closely related if they share a node than if they are in different nodes. With this idea, it is possible to traverse the data structure, from top to bottom, to different regions of related data, and back bottom-up, to gather information obtained at lower levels. A wide breadth of hierarchical data structures have been studied, but in this work, tree-like data structures (binary, quadtree and the $\mathbb{R}^3$ alternative *octree*) have proven their suitability for our problem. In a space storing 3D regions objects and point clouds are usually recursively partitioned in 8 sub-cubes per cube, which naturally translate into octrees.

Fig. 2 shows a possible octree for our running example and the corresponding C data structure declaration. The root node, 0, is the handle to access the whole point cloud. The bounding box, $[(0,0,0),(100,100,100)]$, is bipartitioned along the three axis in eight equal sub-cubes or octants that are represented by 8 nodes (A to H). In the data structure, each octant could have been identified by the minimum and maximum coordinates of the corresponding bounding box $[(x_{min}, y_{min}, z_{min}), (x_{max}, y_{max}, z_{max})]$, but each node occupies less memory storing only four values: center (x_c, y_c, z_c) and radius (that is actually half the side length of the sub-cube). In this example, three octants are empty and the number of points in the other ones varies between 1 and 3. Dense enough octants can be further subdivided recursively, but in our toy example this is not recommended if we do not want to waste more memory in empty nodes.

In memory-bound problems like the one at hand, minimizing the memory footprint is key to reduce data movement and improve locality, so in the next section we will propose several data structure optimization alternatives targeted to minimize execution time in both CPU and GPU architectures.

2.3. The target architectures: CPU and GPU

CPU and GPU architectures, and their memory models, are quite different. Thus, to get the most out of each device, algorithms and data structures should be tailored accordingly. GPUs are accelerators that excel at exploiting massive data-level parallelism but at the price of banning some operations that are available in general purpose processing units, like dynamic memory allocation and recursion, among others.

On a CPU, recursion and dynamic memory are usually leveraged to construct and traverse recursive data structures like trees (a node points to other nodes). On the other hand, the GPU works more efficiently with arrays, which means that tree-like data structures are usually implemented on GPUs using one or more arrays, and indices are used instead of raw pointers. Besides, total space required to store the tree has to be preallocated in advance in GPU memory.

Solving the OWM function implies constructing the tree out of the LiDAR point cloud, to later traverse it finding the Local Lowest Points (LLPs) that will serve to identify the ground points of the Digital Terrain Model. The next section describes the tree construction and OWM traversal phases for both the CPU and GPU, but first, we elaborate next on the suitable programming model able to target both CPU and GPU architectures.

2.4. The programming model: oneAPI vs. CUDA

Traditionally, NVIDIA GPUs and the CUDA programming language have dominated the GPGPU (*General Purpose GPU*) arena, especially for HPC problems. CUDA has evolved for more than 20 years and provides high level APIs, which are used by several DSLs (*Domain Specific Languages*) such as Tensorflow, PyTorch, etc., but also offers low-level and highly optimized routines that make the most out of the NVIDIA architecture. However, CUDA is a proprietary language targeting only NVIDIA GPUs.

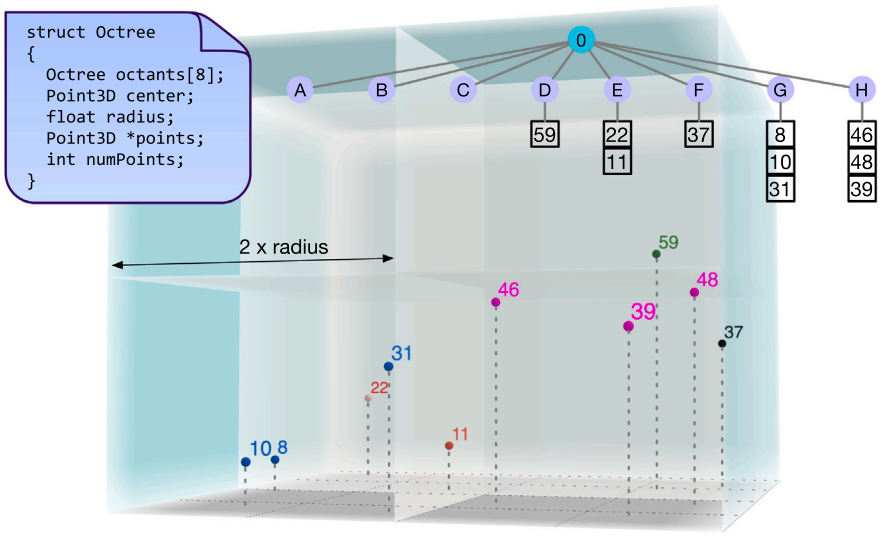


Fig. 2. Octree example and C declaration of the data structure. The color of each point in the cloud depends on the occupied octant.

A promising alternative comes within the oneAPI [4] open industry standard that strives to provide a single ecosystem comprised of languages, libraries and tools able to target several architectures (CPU, GPU, FPGA, etc.) in a friendly and productive way. Regarding the languages, the Intel implementation of the oneAPI initiative, includes the `icpx` compiler based on LLVM, which can compile SYCL [6] code, potentially augmented with some Intel extensions, to binaries that can run on Intel CPU, GPU or FPGAs. Code written in SYCL can also run on NVIDIA GPUs if they are targeted as OpenCL devices. Besides, thanks to a recent CUDA backend for SYCL, it is also possible to compile SYCL code that includes native CUDA kernels. In this paper we optimize OWM for CPU multicores, integrated GPUs, and for the different alternatives available to run on NVIDIA discrete GPUs (including native CUDA), comparing portability and performance. The results are presented in the experimental section.

Besides, oneAPI includes the library oneTBB (Threading Building Blocks) [14] that provides a parallel programming model based on task parallelism and on the work-stealing scheduler that helps in dealing with workload unbalance. This library is based on the C++17 standard and features a set of parallel templates, as the `parallel_for` one, that we will use to productively implement the parallel version of the OWM algorithm.

2.5. The testbed: HW, dataset and baseline

Our platform includes a 9th generation Intel processor with Coffee Lake architecture. More precisely, the CPU is the Core i9-9900K with 8 cores at 3.60 GHz and 8 × 256 KB L2 cache plus 16 MB shared L3 cache and a 32 GB DDR4 memory (37.7 GB/s BW[2]). This processor also features an integrated GPU Intel UHD Graphics 630 @ 1200 MHz (24 EUs). Additionally, the board includes a NVIDIA discrete GPU GeForce RTX 2060 @ 1755 Mhz (30 SMs) and 6 GB of VRAM GDDR6 (131.2 GB/s BW[3]). The operating system is Ubuntu 20.04.2 LTS, kernel 5.11.0 and we use the GCC 9.3.0 compiler and `icpx -fsycl` (previously known as `dpcpp`) compiler from oneAPI version 2024.0. We compile with optimization flag -O3 and reported execution times are the average of 5 runs in which the server is not running any other application.

The dataset comprises 4 LiDAR point clouds[4] from different regions with the following names: Alcoy, Arzua, Brion Foresta and Brion Urban. Table 1 summarizes the main characteristics of these point clouds, including the number of points, bounding box (x_{min}, y_{min}, x_{max}, y_{max}), spatial dimensions and point density (pts/m^2). It should be noted that this is the average density, although in the Brion clouds (obtained by photogrammetry) there are areas with very high density and others with very low one.

In our experiments we have fixed $O_x = O_y = 4/5$ and $\omega = 10$ m. This results in $o_x = o_y = 2$ m for the sliding window displacement. This means that the overlapping area is 80 m^2 and also that the same LLP can be selected up to 25 times. Under these conditions, Table 1 also shows the execution time of the C sequential code and OpenMP version that was described in Section 2.1 specifying both the tree construction sequential time, OWM traversal sequential and OWM traversal parallel times with 8 threads. Note that the tree construction time grows with the number of points in the cloud, however, the OWM traversal time depends more on the bounding box and therefore the number of steps. For example, Alcoy requires 714 × 959 steps and although it has lower density than Brion urban, the former requires more OWM traversal time than the later larger cloud that only requires 372 × 276 steps. We also see that the parallel speedup is particularly poor for the larger clouds. As we will see later, that is due to the memory-bound nature of this algorithm. We would like to remark that this baseline implementation is already a step forward in comparison with the original R implementation of the OWM function. This original code was evaluated in [3] with small point clouds (less than 52,000 points) and out of the clouds of Table 1 only Alcoy can be processed in R, taking 15 min (the other larger clouds run out of memory).

We will consider the times reported in Table 1 as the baseline that we tackle to outperform with the optimizations described in the next section, studying their impact on the OWM traversal phase -that is slower-, but also on the tree construction phase that becomes the bottleneck once the OWM phase is optimized. Our optimizations are organized in four categories and are adapted to target the specific features of each device on our platform: a CPU multicore and a GPU.

[2] Reported by the Intel Advisor profiler.
[3] Reported by the NVIDIA Nsight profiler.
[4] Provided by Babcock International: https://www.babcockinternational.com.

Table 1
Details of the LiDAR point clouds used in the experiments.

	Number of points	Bounding box [m]	Dimensions [m]	Density [pts/m^2]	Tree const. seq. time [s]	OWM trav. seq. time [s]	OWM trav. par. time (8ths) [s]
Alcoy	20.380.212	714 947.985, 4 286 501.934, 716 361.062, 4 288 406.229	1413.08 × 1904.30	7.57	4.74	22.55	3.73 (6.0×)
Arzua	40.706.503	568 000.00, 568 999.99, 4 752 320.00, 4 753 319.99	999.99 × 999.99	40.7	5.75	22.94	4.11 (5.6×)
Brion forestal	42.384.876	526 964.093, 4 742 610.292, 527 664.647, 4 743 115.738	700.55 × 505.45	119.7	5.97	19.75	6.23 (3.2×)
Brion urban	48.024.480	526 955.908, 4 742 586.025, 527 686.445, 4 743 124.373	730.54 × 538.35	122.11	6.76	21.30	7.33 (2.9×)

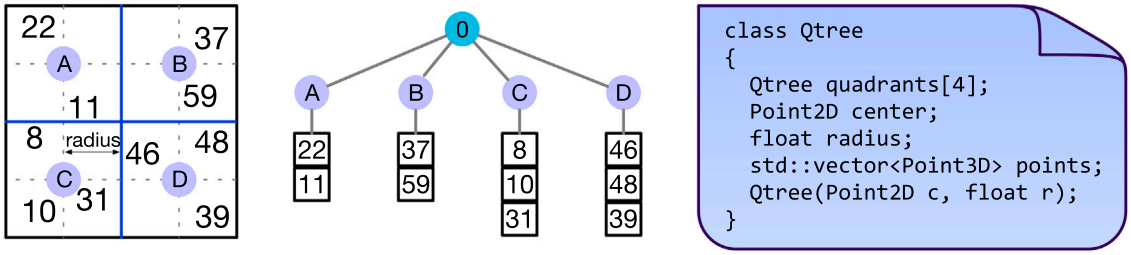


Fig. 3. Quadtree example and C++ class alternative for the data structure.

3. Optimizations targeting the CPU

In this section, we present different optimizations aimed at reducing the CPU execution time by devising better data structures, improved parallelization techniques and memory access reductions.

3.1. Optimization 1: 2D-space bipartition based tree data structure

In airborne LiDAR, since the aircraft is usually hundreds of meters above the area of study, the point clouds are, so called, 2.5D because the range of values in the z-axis is small in comparison with the range in the other two dimensions. Consequently, using a 3D bipartion of the space results in many empty octants (as we saw in Section 2.2), which translates in a lot of memory wasted for large point clouds. To alleviate this problem, we propose to project the points into the X-Y plane (using the (x, y) coordinates of the points) and to partition this 2D space instead. Now, the bounding box is recursively partitioned into quadrants and therefore a quadtree is the natural data structure to store the point cloud.

Fig. 3 shows a possible quadtree for the same running example of Fig. 2. Now we use C++ so we also show the corresponding C++ class declaration (including the constructor declaration in the last line). The root node, 0, gives us access to the whole point cloud, but now the 2D bounding box is partitioned into four equal quadrants that are represented by 4 internal nodes (A to D). Again, instead of identifying each quadrant by its bounding box coordinates, [(x_{min}, y_{min}), (x_{max}, y_{max})], it is cheaper to store only the three values needed for the center (x_c, y_c) and radius. In this example, there is no point in further subdividing each of these nodes into smaller ones because seven of them would be empty. The alternative is to consider nodes A to D as leaf nodes that include a vector of the 3D points that fall in each quadrant.

3.2. Optimization 2: CPU parallelization

Regarding the parallel programming model for the CPU, we moved from OpenMP (where a `parallel for` was used in the baseline implementation) to Threading Building Blocks (oneTBB) [14]. The main advantage of TBB relies on the `parallel_for` template that we have used to parallelize the main loop of the OWM traversal phase. The mentioned template is based on a work-stealing task scheduler that naturally deals with load unbalance and locality. It is, if a worker thread finishes its tasks earlier, it can steal tasks from other busier worker threads (usually the coldest in the cache used by the stolen task). This is a key feature in our problem where the OWM steps are processed in parallel, but each step has an unknown amount of workload depending on the corresponding slide window coordinates and on the density of the point cloud at this region of the 2D space. In addition to the OWM traversal phase parallelization, we also explore how to parallelize the tree construction phase.

3.2.1. OWM traversal phase parallelization

With respect to the baseline OWM traversal phase described in Algorithm 1 of Section 2.1, we have identified three main optimization strategies that are sketched in the pseudocode of Algorithm 2.

The first one consists in fusing the two nested loops that were traversing the rows and columns of the 2D space of the point cloud into a single loop that traverses all the steps, as we see in line 2 in Algorithm 2. This is now the loop that we parallelize with the TBB `parallel_for` template. This loop fusion enlarges the loop trip count which in turn increases the parallel scheduling opportunities for the loop. As the second optimization strategy, we have eliminated the critical section (that was protecting lines 9 and 10 in Algorithm 1) because now each thread is writing in the `step` (loop counter) position of the $LLPsFound$ array instead of using the induction variable `countLPP`. Finally, instead of first gathering the points that fall inside the sliding window and then finding the minimum if the number of points is large

Algorithm 2 Optimized OWM traversal phase

```
Require: P, ω, O_x, O_y.   ▷ Point cloud, window size, horizontal and vertical overlaps
1: NumSteps = ComputeNumSteps(P, ω, O_x, O_y)
2: for step = 1, NumSteps do                ▷ In parallel using tbb::parallel_for
3:     SW_Center = ComputeSlidingWindowCenter(P, ω, O_x, O_y, step)
4:     LLPid, numPts = searchMin(SW_Center, ω, PointCloud)
5:     if numPts > minNumPoints then
6:         LLPsFound[step] = LLPid
7:     end if
8: end for
9: GroundSeedPoints = ComputeSeedPoints(LLPsFound)
10: return GroundSeedPoints = FillEmptyCells(P, B, GroundSeedPoints)
```

enough, now we directly compute the minimum and count the number of points, `numPts`, at the same time (function `searchMin` in line 4). Only if `numPts` is larger than the threshold, we store the index of the minimum point in the $LLPsFound$ array (which is initialized with -1's so that we can identify the updated positions).

3.2.2. Tree construction phase parallelization

When constructing the tree on the CPU, two critical issues have to be efficiently tackled: (i) minimize the synchronization among the threads and (ii) partition the load carefully. Note that each LiDAR point initially stored in the input array has to be placed in the right tree leaf according to its (x, y) coordinates and that each branch depth may depend on the cloud density at the corresponding 2D region.

Our tree construction implementation consists of three stages as follows:

1. First stage (sequential): We sequentially construct the tree but only up to a given tree level, lev. For example, if $lev = 2$, only the root (level 0) and levels 1 and 2 are created, this is $4^0+4^1+4^2$ internal nodes. This is executed very fast, and it is not worth the overhead of creating and synchronizing the threads to parallelize it. This first stage uses the node radius as the tree splitting criterion.
2. Second stage (parallel): We traverse in parallel the array of LiDAR points. Each thread only visits a chunk of the input array and inserts the points in the internal nodes of the last level created in the previous stage. This is, these internal nodes are temporally used as leaf-nodes and they have a `tbb::concurrent_vector` data member in which the points can be stored. We rely on the TBB's `concurrent_vector` container because it allows for the concurrent insertion of elements in the vector. This is necessary because several threads may found in parallel that a point belongs to the same (temporary) leaf-node and the concurrent insertion has to be done in a thread-safe way. This is, this parallel stage can suffer from some parallel overhead due to potential collisions at the insertion of points.
3. Third stage (parallel): We traverse in parallel the (temporary) leaf-nodes (those at level lev) to finish up the tree construction. Basically each thread takes care of a temporary leaf-node and takes it as if it were a root node with a private vector of points to be inserted. That way each thread, in parallel, is able to construct the subtree (or treelet) hanging from the temporary leaf-node assigned to it, without any synchronization with the other threads. This final construction can be done considering different recursive cut-off criteria (to stop the recursive bipartition) that we explore in Section 3.4.

In essence, the first and second stages are implementing a pre-sorting phase in which the LiDAR points are classified according to the treelet to which they belong. This enables a fully parallel final stage that finish the construction of the tree. The caveat here is the proper selection of the level, lev, parameter. If lev is too small, there will be less (temporary) leaf-nodes after the first stage which means more potential collisions, less parallelism available at the third stage, and larger temporary vector sizes. On the other hand, if lev is too large, the first stage will take longer, and it can create nodes associated to 2D regions without points or with very low density. In the experimental section we study the impact of lev in the tree construction time.

3.3. Optimization 3: reduce the number of accesses by memoization

In general, the smaller the depth and number of nodes of the tree, the smaller the memory footprint, but then the higher the number of accesses to check the points stored in the leaf-nodes. One strategy to reduce the memory bandwidth pressure consists in reducing this number of accesses. The function `searchMin()` (see line 4 of Algorithm 2) has to look for all the points inside the sliding window to find the minimum value. Due to the sliding window overlap there are points that are re-visited in different steps of the traversal. We can save some of these repeated accesses via memoization (storing values that have been computed before and reusing them when needed).

In our problem we can keep the value of the minimum of a previous step and, if this minimum falls inside the sliding window of the next step, avoid searching in the area overlapped by both consecutive steps. For example, in Fig. 4(a) the minimum in step 3.2, 8, is still inside the sliding window in step 3.3. This means that we can just search for the minimum in the new region (the reddish area depicted in step 3.3) and keep the minimum of both regions (8 again, that is smaller than 46). However, this is not always the case as we can see in step 3.4, where the previous minimum, 8, is outside the new sliding window and the whole area has to be considered when finding the minimum. Note that the larger the overlap values (O_x, O_y), the more likely is that the previous minimum is inside the next sliding window and therefore that this strategy has an impact on the execution time.

Another alternative consists in storing at each tree node the minimum value of all the elements hanging from that node. This minimum can be computed at tree construction or during the OWM traversal. With that, if the sliding window fully overlaps within the bounding box of a node, the corresponding subtree is not traversed because the node already provides the minimum of that subtree. Similarly, the number of points hanging from each node are also stored in that node, which avoids traversing the current subtree to gather this required information. The drawback of this approach is that the memory footprint of the tree is increased by the additional information stored in each node (minimum and number of points covered by each node).

This strategy is illustrated in Fig. 4(b) where we see the sliding window at the position corresponding with step 4.3 of our running example and the corresponding quadtree. Now the nodes of each quadrant (previously identified with letters A to D) store the minimum of the points of that subregion in addition to the `points` vector. In the figure we identify with an "R" in a red circle the read accesses. Note that the points of node C are not accessed because this quadrant is fully covered by the sliding window and the minimum of the subtree, 8, is already stored in node C. The vector of points of the other quadrants is completely traversed to check if each point falls inside the window and compute the minimum in such a case. This technique can be useful also for partially overlapped nodes. For example, node A identified now with the minimum 11, is partially overlapped by the sliding window, but the minimum 11 is inside the overlapped region so we skip the minimum search. However, we still have to traverse the `points` vector to count the number of points that fall in this overlapped region.

These two approaches ((a) and (b)) are not exclusive, so we have put to work both of them during the minimum search at each step of the OWM algorithm. First, we identify the search region, which can be a fraction of the sliding window (as in Fig. 4(a) Step 3.3) or the whole sliding window (as in step 3.4). Then, we traverse the nodes of the tree from the root. If the region represented by a tree node

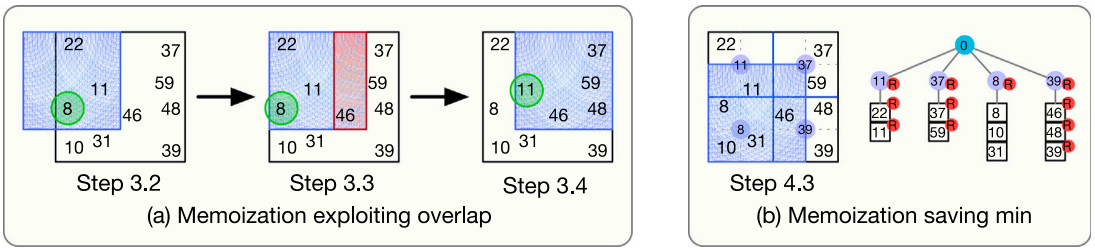


Fig. 4. Two alternatives to exploit memoization in the OWM traversal phase.

is fully overlapped by the search region, we take the minimum and number of points stored in that node and do not navigate deeper in this tree branch. Otherwise, we visit the four children of the node and recursively do the same query. The impact of these approaches is analyzed in the experimental section.

3.4. Optimization 4: tune the granularity

Recursive data structures, as recursive functions, require a cut-off or base criterion (stop condition) to avoid infinite recursion. By default, tree data structures that store space related objects or points, stop the recursive partitioning of the space when the leaves of the tree store a single element/point. However, in our OWM case, having one point per leaf-node is too fine-grained, resulting in very deep trees with too many internal nodes (that need to be traversed) and the corresponding memory overhead. We propose two alternatives that help in this regard:

- **MinRadius**: a node of the tree is not further subdivided in quadrants if the resulting $radius$ of the leaves is smaller than $MinRadius$. In other words, all the tree nodes have $radius \geq MinRadius$. Note that this criterion does not consider the point cloud density so the sparsest regions of the cloud point have less populated leaf nodes which translates into load unbalance. Times reported in Table 1 were obtained with this technique and $MinRadius = 0.1$ m. In the same conditions and as an example, reducing $MinRadius$ to 0.01 m increases tree construction time up to 3x and OWM time between 2x to 6x depending on the point cloud size.
- **MaxNumber**: a node of the tree is not further subdivided in quadrants if the number of points contained in this node (`points.size()` in the data structure of Fig. 3) is smaller than MaxNumber. This is, MaxNumber is the maximum number of points that can be stored in a leaf-node. If a new point has to be added to an already full leaf-node, that node turns into an internal-node, splitting the region in four quadrants (new leaf-nodes) and distributing the points (including the new one) among these children nodes. As a consequence, the depth of the branches depends on the density of the point cloud in the region represented by these branches but, at the same time, the load balance is better preserved for point clouds with significant differences in the densities of the subregions. On the other hand, converting a leaf-node into an internal node with four leaves and redistributing the points of the original leaf between the new leaves is a costly operation that it not always compensated by the extra load balance achieved later during traversal.

Fig. 5 illustrates the differences in constructing the quadtree for these two covered alternatives. The blue dots represent the points of the cloud, which will be stored in the tree. As we can see in the tree with $MinRadius = o_x/2$, nodes A, C and D have to be split into quadrants with $radius = MinRadius$, resulting in leaf-node $d4$ storing the three points of the most dense region of the cloud. Note that node B does not have children because it is empty and that many leaf-nodes are also empty in sparse regions of the space. On the other hand, the tree with $MaxNumber = 3$ has a more balanced distribution of the points among the leaf-nodes and has fewer nodes and fewer empty nodes.

Either $MinRadius$ or $MaxNumber$ can be used to control the granularity of leaf-nodes and should be carefully tuned. Small values of $MinRadius$ or $MaxNumber$ result in a deeper tree with fine-grained leaf-nodes, more memory overhead and more time consumed at tree construction. On the other hand, large values lead to large `points` vectors in the leaves. These vectors will need to be fully traversed to read the (x, y) coordinates and check if they fall inside the sliding window, and in such case, read the z value to find the minimum.

Depending on the point cloud and target architecture (CPU or GPU) one criterion can be better than the other. In our experiments, we have found that $MinRadius$ is better for the CPU (the tree is constructed faster, and the load unbalance can be handled on the CPU with dynamic or task-stealing scheduling), whereas $MaxNumber$ is more suitable for the GPU (load unbalance has a deeper impact in this architecture and the tree is constructed differently so that there is no leaf-node to internal-node conversion overhead). In Section 5 we will show the results of the experiments with both alternatives.

4. Optimizations targeting the GPU

The GPU kernels for the tree construction and OWM traversal have been implemented in SYCL and CUDA. The SYCL code can be compiled for the CPU multicore device (S-CPU),[5] the integrated GPU (S-iGPU), or the discrete NVIDIA GPU (S-dGPU), as sketched in Fig. 6. This code can be compiled using -DSCPU, -DSiGPU or -DSdGPU, and depending on the compilation flag the `queue` object will feed the CPU, the integrated GPU or the NVIDIA GPU, respectively. This feature greatly simplifies the development and maintenance of a single code base that can run in three different devices just by setting the desired compilation flag. For the CPU, iGPU and dGPU versions we use the `icpx -fsycl` compiler. Also note in Fig. 6 that the dGPU `queue` is constructed (in line 7) using a lambda that returns true for the device using the CUDA backend.

We also target the discrete NVIDIA GPU in two more ways. The first one is with a pure CUDA code that we name CUDA and that is compiled using the CUDA compiler `nvcc`. The second one is a SYCL-CUDA hybrid in which CUDA kernels are called from SYCL leveraging SYCL interoperability features. This version is called S-CUDA (more details are provided in Section 4.2).

4.1. Optimization 1 (O1): GPU oriented tree data structure

We have already discussed the tree data structure used on CPU, which requires pointers and dynamic heap memory allocations and is usually built via recursive functions. On GPUs this same approach is either unfeasible (if recursion and/or dynamic allocation are not allowed) or suboptimal (due to high data divergence). On the contrary, GPU oriented trees are stored in statically created arrays and indices are usually used as pointers to connect parents and children of the tree.

[5] "S" stands for SYCL.

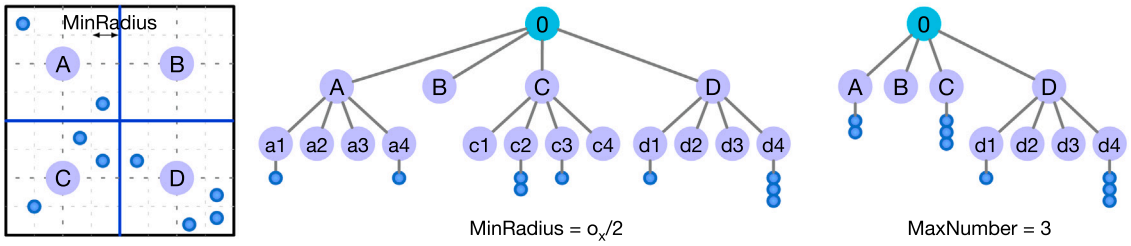


Fig. 5. MinRadius vs. MaxNumber example.

```
#ifdef SCPU
    sycl::queue queue{sycl::cpu_selector_v};
#elif SiGPU
    sycl::queue queue{sycl::gpu_selector_v};
#elif SdGPU
    sycl::queue queue{[](auto& d)
    {return (d.get_platform().get_backend() == sycl::backend::ext_oneapi_cuda);}};
#endif
```

Fig. 6. Construction of the SYCL queue depending on the desired device (CPU, iGPU or dGPU).

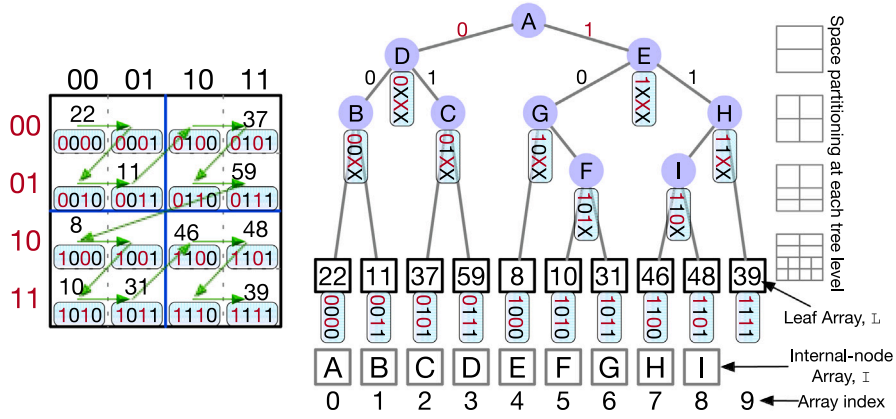


Fig. 7. GPU oriented data structure: Binary radix tree based on Morton codes.

Our approach incorporates ideas that were first used in the graphics domain to solve problems as real-time ray tracing, 3D collision detection, voxel-based global illumination, etc. These ideas were first proposed for the GPU implementation of Bounding Volume Hierarchies, BVH, by [15] in a method called LBVH. This strategy was later improved by [16] and optimized to increase the GPU occupancy (exploited parallelism) by [17,18]. We have taken ideas from all of these works and adapted them to work with 2D point clouds instead of 3D objects/primitives (usually triangles). Our method also departures in that we store several points per leaf (instead of one object per leaf) following the MaxNumber strategy mentioned before. MinRadius is discarded on the GPU because it is not suitable for our GPU oriented data structure based on Morton codes that we describe next. We rely on Fig. 7 that revisits our running example to illustrate our GPU data structure and the steps required to build it assuming a single point per leaf node.

In order to store a 2D space into a 1D array we use a "space filling curve" that keeps locality thanks to indexing the 2D points with Morton codes [19]. The first step consists in computing the Morton codes associated to the points. This is done by first discretizing the (x, y) coordinates of each point and then interleaving the bits of both coordinates. For example, in the figure, the point with $z = 59$ has a discretized $x = 11$ and $y = 01$ which results in a Morton code $m = 0111$. If we sort the points using as sorting key the Morton codes, we can linearize in a 1D array of leaf nodes, L, the cloud of points maintaining the locality (neighbor points in the 2D space are still close in the 1D array). In our implementation and for the point clouds that we evaluate, we use 32 bits unsigned integers to store the Morton codes which give us 16 bits for each coordinate. For bigger point clouds we could use 64 bit unsigned integers instead. In any case, with this strategy we get an array of leaf nodes in which each node also includes the bounding box of the points stored at each leaf node.

A sorted list of Morton codes has several interesting properties, being the most relevant one that it allows us to build a binary radix tree (BRT) in a single pass. Fig. 7 shows the BRT built from the sorted Morton codes of the running example. One of the characteristics of a BRT is that the number of nodes is known in advance and equal to the number of leaves minus one. In our example we have 10 leaves and 9 internal nodes. This is a very important property because it enables us to pre-allocate the node array and also construct the tree in parallel. Each node has to store the indices (pointers) of the children and the parent as well as the bounding box and minimum value of all the points that hang from that node.

When linearizing the tree nodes in the array of internal nodes, I, it is key to also exploit locality by keeping related nodes (children, siblings, parent) close in the node array. To this end we leverage some additional properties of the BRT based on Morton codes that we will illustrate with our example in Fig. 7. We have named the nodes in the node array with the sorted sequence of letters from A to I, aligned with the same indices of the sorted list of leaves in the leaf array. Now, to establish the parent–children relationships we use the following rules:

1. Each internal node corresponds to the longest common prefix shared by the keys in its respective subtree, and singleton nodes (nodes with a single child) are avoided. For example, node D is ancestor of points $22, 11, 37$ and 59, all with Morton codes of the class $0XXX$. There is not an internal node for the class $100X$ because only the point 8 belongs to that class.
2. Each internal node partitions its Morton codes according to the first differing bit (starting from the most significant one). For example, the root node A partitions the Morton codes in two classes: $0XXX$ and $1XXX$. Note that it also partitions the 2D space by the y coordinate.
3. Each internal node has an index in the node array, `I`, that corresponds either to the end of a class (if it is a left child) or to the beginning of a class (if it is a right child). For example, the node D is the left child of A and has the index 3 which corresponds to the point 59 that is the end of the class $0XXX$ in the leaf array, `L`. Similarly, node E is the right child of A and points to the beginning of the class $1XXX$. The root node is always the first node and points to the beginning of the whole array `L`.

With these rules, in parallel, one thread per tree node can find its two children and update the indices/pointers with the parent–child relationship as detailed in [17]. The last step is to compute the bounding boxes and minimum values of the nodes required by the memoization strategy presented in Section 3.3. This is done in a single parallel bottom-up pass from the leaves to the root. The bounding box of a node is the union of the bounding boxes of its children and similarly, the minimum value of a node is computed from the minimum value of its children. The parallel computation starts with a GPU thread per leaf-node that computes the bounding box and minimum for that node. Then the first thread that visits a parent node sets to one an atomic variable initialized to zero and dies. The thread that comes from the other children finds the atomic variable already modified and takes care of updating in the parent node the bounding box and minimum value from the two children.

An additional optimization strategy consists in reducing the depth of the tree by constructing an octree out of the binary tree. The idea is to collapse each block of three tree levels in the binary tree into a single octree node. Note that this new octree partitions the 2D space in contrast with the 3D octree that was described in Section 2.2. For example, in Fig. 7 the nodes A, B, C, D, E, G, and H, can be collapsed into a single octree root node with 8 children ($22, 11, 37, 59, 8, F, I$, and 39).

4.2. Optimization 2 (O2): GPU parallelization

As we said, we have three code implementations able to run on the GPU: (i) the SYCL implementation that can generate two versions to run either on the iGPU or the dGPU (S-iGPU, S-dGPU); let us not forget that this implementation can build a version to run on the CPU, S-CPU; (ii) the CUDA implementation running on the dGPU (CUDA); and the SYCL-CUDA hybrid also running on the dGPU (S-CUDA). In all the cases the kernel that executes the OWM traversal phase is based on the same data-parallel paradigm: the body of the OWM parallel loop (line 2 in Algorithm 2) is implemented as a kernel that is executed by the GPU threads. The SYCL syntax for that implementation is sketched in Fig. 8(a). In line 1 we invoke the `submit` member function of the `queue` object to submit a command group to the corresponding device. This command group includes a `parallel_for` invocation (line 3) specifying the number of iterations (`NumSteps`) and the lambda function with the kernel code (line 5) that is executed for each iteration, `it`.

The main differences with the SYCL-CUDA hybrid implementation are highlighted in Fig. 8(b). We first note that the command group calls a `host_task` member function instead of the `parallel_for` one. This enables the execution of the interoperability code required to properly get access to the CUDA kernel arguments and data. After that, we can just call the CUDA kernel as usual (line 5) using the `<<<grid, block>>>` syntax to specify the number of blocks and threads per block, respectively. In the pure CUDA implementation we do not need the `queue.submit()` nor the `host_task` member functions, as we can just call the CUDA kernel directly from the host code.

4.3. Optimization 3 (O3): reduce the number of accesses by memoization

Out of the two memoization strategies implemented on CPU and described in Section 3.3, only the memoization saving min approach (see Fig. 4(b)) is applicable to the GPU. The first strategy (memoization exploiting overlap) is based on temporal locality (one thread processing one step of the OWM algorithm can take advantage of the minimum found on the previous step), but on the GPU implementation there is a thread per step, which renders this approach unfeasible. The second strategy (b) is effectively implemented for the GPU OWM traversal, which requires storing at each tree node the number of points and minimum of all the points hanging from that node.

4.4. Optimization 4 (O4): tune the granularity

The GPU architecture is quite sensible to data and control divergence, which translates in that achieving load balance is more relevant on this kind of device than on the CPU. For that reason, keeping a balanced number of points per leaf-node is crucial to achieve good performance. For that reason we have implemented on the GPU an optimized version of the MaxNumber strategy described in Section 3.4 for the CPU. The idea is to force all the leaf-nodes of the tree to store exactly MaxNumber points, being MaxNumber a power of 2. This can be achieved once all the points of the LiDAR cloud have been sorted according to their Morton codes (see Section 4.1). Then, the sorted list of points is divided into chunks of MaxNumber points each, and each chunk is assigned to a leaf-node. The data members of each leaf-node are later updated with the information related to the bounding box and minimum of the points it contains. Note that with this static partition of LiDAR points among leaf-nodes, the bounding boxes of sibling nodes can overlap and this has a negative impact on the OWM traversal (that has less opportunities to prune the nodes). However, the alternative implies variable size leaf-nodes, and we found that this irregularity degraded times on the targeted GPU devices, so we did no explore that option further.

5. Experimental results

Before analyzing the performance of the different CPU and GPU versions, we first validate the final results of our proposed solution to assess that the implemented optimizations do not affect the quality of the detected ground surface.

5.1. Validation

We consider as the "gold" ground surface, $G\varphi_{t=0}$, the one that results from running the baseline sequential version of the OWM code. This is the one that is used to validate the results of the other versions studied in this paper. However, it should be noted that the implementation of some optimizations turns the resulting output into a non-deterministic one. For example, the selection of the minimum value depends on the order in which it is found. Several LiDAR points, with different ids can have the same z-value,[6] so different search orders

[6] This is quite frequent in the LiDAR clouds used in our experiments because the z-value is provided with maximum precision of 0.001 and spatially closed points can have the same altitude.

```
queue.submit([&](sycl::handler &h) {
  ... // host-to-device data management
  h.parallel_for(NumSteps,[=](auto it) {
    ...
    searchMinSYCL(it, ...)});
});
```

(a)

```
queue.submit([&](sycl::handler& h) {
  ... // host-to-device data management
  h.host_task([=](sycl::interop_handler ih) {
    ... // reinterpret with interop_handler
    searchMinCUDA<<<grid, block>>>(...);});
});
```

(b)

Fig. 8. Submitting a SYCL kernel (a) or a CUDA kernel (b) from SYCL code.

Table 2
Improvement of each optimization w.r.t. the previous one.

Cloud	O1 vs. OpenMP baseline			O2 vs. O1			O3 vs. O2			O4 vs. O3		
	Tree C.	OWM	Total	Tree C.	OWM	Total	Tree C.	OWM	Total	Tree C.	OWM	Total
Alcoy	1.49×	1.88×	1.64×	6.27×	1.07×	2.18×	0.73×	3.31×	1.88×	4.14×	2.04×	2.83×
Arzua	1.52×	2.42×	1.80×	5.52×	1.23×	2.65×	0.83×	3.68×	1.73×	2.49×	1.39×	2.00×
BrionF	1.48×	2.03×	1.72×	5.15×	1.63×	2.66×	1.02×	6.81×	2.55×	1.00×	1.00×	1.00×
BrionU	1.50×	1.88×	1.68×	4.89×	1.76×	2.68×	1.06×	6.55×	2.60×	1.00×	1.00×	1.00×

can result in different ids of the minimum. This hampers a direct comparison between the gold version and the results of the different optimized codes.

There are several strategies that have been used to estimate the error produced when processing LiDAR point clouds [11,20]. In [20] absolute error arising from computing digital terrain models are provided and errors in the order of centimeters are accepted. In our work we consider a valid ground point of the resulting ground surface, $\varphi_{t=0}$, if in the nearby (using $o_x/2 = 1$ m as radius) there is a corresponding point in the gold ground surface, $G\varphi_{t=0}$, with a difference in altitude of less than 1 cm. With this criterion, in all our experiments we always obtain more than 97% of valid ground points.

5.2. Performance analysis on CPU

In this section we discuss the performance improvements that our optimization strategies achieve on our target CPU. We start evaluating optimization 1 (O1) with respect to the baseline algorithm (see sequential tree construction and parallel OWM traversal times – based on OpenMP – in Table 1). By default, this baseline uses as base criterion $MinRadius = 0.1$ in the tree construction phase, and we keep this criterion till optimization 4 (O4), where we explore other strategies and values to control the leaf-nodes granularity. After O1, we evaluate the performance improvement that each new optimization adds to the previous one (+O2, +O3, +O4). Our study breaks down the impact of each optimization in the tree construction (Tree C.) and OWM traversal (OWM) phases, as well as in the total execution (Total) times. Table 2 summarizes the corresponding improvements for each optimization w.r.t. the previous one, always using 8 threads. In the next subsections, we analyze each optimization in more detail.

5.2.1. Optimization 1 (O1): 2D-space bipartition data structure

As we see in Table 2, O1 improves the tree construction times by around 50% for all clouds, while parallel OWM traversals are between 1.88x and 2.42x faster. Thus, we see that the change from a 3D bipartition to a 2D bipartition data structure has a greater impact in the traversal phase. More importantly, this optimization takes up to 90% less memory footprint. For instance, using a heap profiler (Massif from Valgrind[7]), we found that for the smallest point cloud in our dataset (Alcoy) the octree requieres 1.23 GB whereas the quadtree only 192 MB. For larger datasets we save even more memory space. In any case, the O1 optimization improves total execution times up to 1.8x.

[7] https://valgrind.org/docs/manual/ms-manual.html.

Table 3
Tree features for each cloud: Percentage of empty leaf-nodes at $lev = 5$, number of leaf-nodes at $lev = 7$ vs. $lev = 5$, and number of leaf-nodes at $lev = 9$ vs. $lev = 5$.

Cloud	% empty leaf-nodes	Ratio no_7/no_5	Ratio no_9/no_5
Alcoy	51%	1.00	1.01
Arzua	11%	1.00	1.01
BrionF	2%	1.08	1.19
BrionU	2%	1.08	1.20

5.2.2. Optimization 2 (O2): Parallelization

Here we evaluate the different parallelization strategies (O2) that we propose in Section 3.2. All parallel executions run with 8 threads. Let us recall that from now on, we use TBB as programming model to implement our optimizations.

Fig. 9 shows the times of the parallel tree construction described in Section 3.2.2 for different values of the lev parameter (tree level at which the initial part of the tree is created sequentially). As discussed in the mentioned subsection, small values of lev enable little parallelism in the final tree construction stage, while large values penalize the first sequential stage because it takes longer, but also the deeper the tree the more leaf-nodes are created, so a trade-off value must be selected for the corresponding cloud and number of threads. In the figure we see that for our clouds and 8 threads, the optimal lev is between 5 and 7. Table 3 represents some interesting features related to the trees created at different levels. Recall from Table 1 that Alcoy and Arzua cover a larger area but with less points than BrionF and BrionU clouds, and therefore the latter have a higher density of points. In clouds with a lower density (Alcoy, Arzua), increasing lev practically does not affect the number of leaf-nodes, because there is already a high ratio of empty leaf-nodes at $lev = 5$, and therefore the cost for creating the corresponding temporary vector of points in each non-empty node is similar. However, in clouds with a higher density, increasing lev also increases the number of leaf-nodes and therefore the cost for creating the corresponding temporary vector of points for each non-empty leaf-node.

Fig. 9 also shows the times of our TBB OWM traversal version described in Section 3.2.1 for different values of lev. Now, we notice that increasing the depth of the tree slightly decreases the parallel traversal times, because increasing the depth decreases the size of the leaf-nodes and thus the number of check operations per node. From the figure we see that the values of lev that achieve the optimal total execution times are 7, 7, 5, and 5 for Alcoy, Arzua, BrionF and BrionU, respectively. For these values, the parallel TBB OWM traversal incrementally improves OpenMP OWM traversal from 1.07x to 1.76x. More interestingly, for the same lev values the parallel tree construction strategy has a more significant impact on performance: up to a 6.27x of improvement. In any case, the O2 optimization incrementally improves the total execution times up to 2.68x w.r.t. O1 (see Table 2).

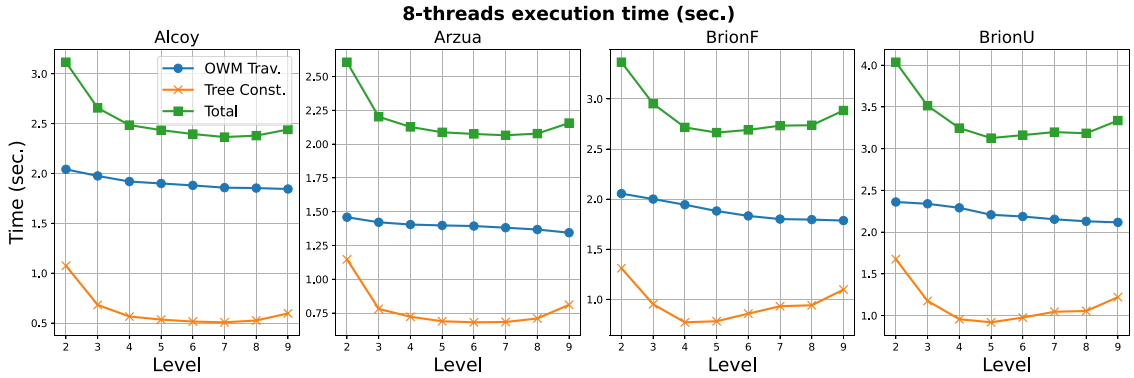


Fig. 9. Optimization O2: tree construction, OWM traversal and total times depending on *lev* using 8 threads.

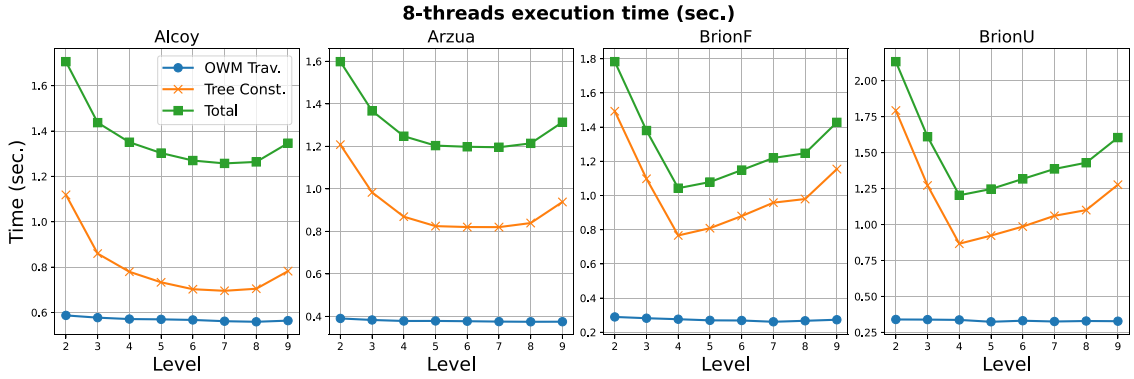


Fig. 10. Optimization O3: tree construction, OWM traversal and total times depending on *lev* using 8 threads.

5.2.3. Optimization 3 (O3): reduce the number of accesses by memoization

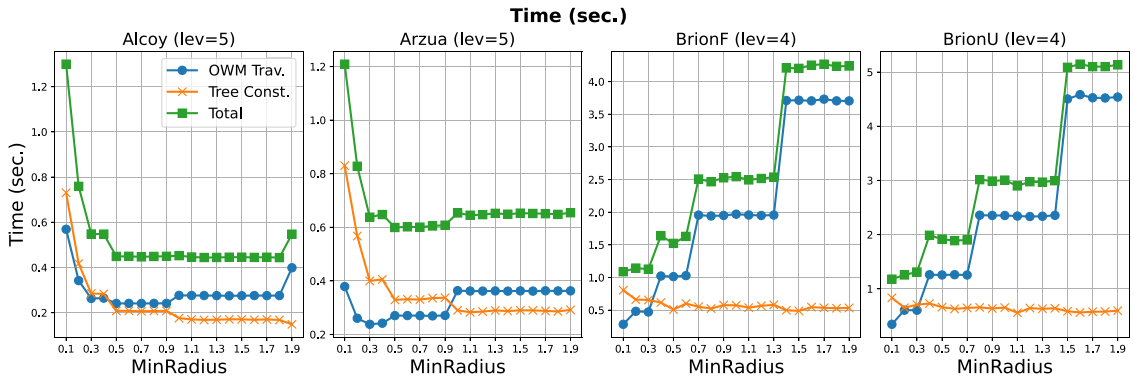
In this subsection we evaluate the impact of the memoization strategies (O3) proposed in Section 3.3 in our parallel codes (parallel executions with 8 threads again). Fig. 10 shows the times of the parallel tree construction, parallel OWM traversal and total time for different values of the *lev* parameter, which we explore again when we apply all the memoization strategies proposed in the mentioned section. As expected, there is a trade-off *lev* value that optimizes the times: 7, 7, 4 and 4 for Alcoy, Arzua, BrionF and BrionU, respectively. These *lev* values are used to report the O3 improvements w.r.t. O2 that we see in Table 2. Now, the tree construction phase takes longer because building the augmented tree with the memoization feature requires to add two new data members to each tree node: the minimum and the number of points hanging from the node. As we notice in Table 2, the tree construction times can degrade up to 27%. However, O3 has a more profound impact on the OWM traversal times, which improve from 3.31x to 6.81x. We should note that we achieve higher incremental improvements in the clouds with higher density. We have also factored out the impact that each memoization strategy achieves in the traversal times: strategy (a) - memoization exploiting overlap - improves the parallel traversal times up to 1.76x, while strategy (b) - memoization saving min - is the one with the highest impact, because it further improves these times up to 2.93x.

Interestingly, Fig. 10 tells us that the tree construction phase represents now the bottleneck in the total execution time. In any case, thanks to the combination of the memoization strategies (a) and (b) and the resulting reduction in the memory bandwidth requirements, O3 optimization incrementally improves the total execution times up to 2.6x (see Table 2).

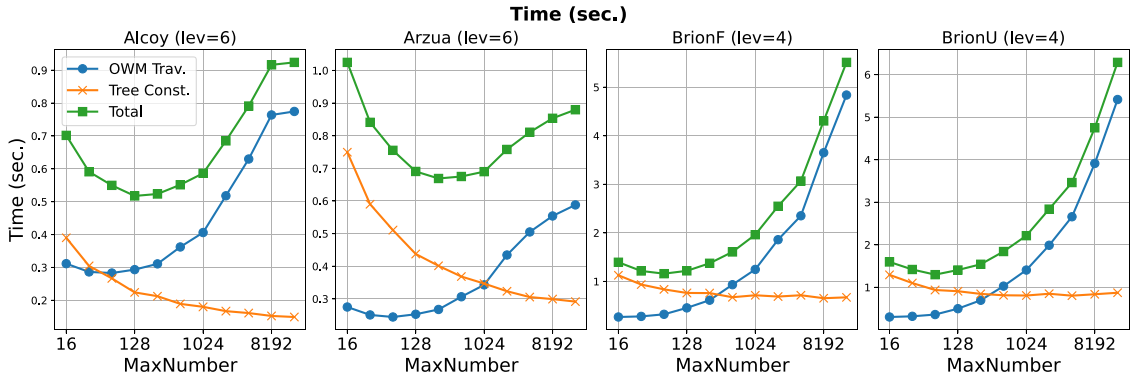
5.2.4. Optimization 4 (O4): tune the granularity

In this section we evaluate the impact of the strategies for tuning the granularity of the tree leaf-nodes, which we propose in Section 3.4: MinRadius and MaxNumber. For both strategies we have explored again the *lev* parameter, and in Fig. 11 we show the parallel times (tree construction phase, OWM traversal phase and total times) for the optimal *lev* value found for each cloud and strategy (indicated in the figure). In particular, Fig. 11(a) represents the times for MinRadius when this parameter varies between 0.1 and 0.9, whereas Fig. 11(b) shows the times for MaxNumber when this parameter goes from 32 to 65 536. Four main conclusions can be extracted from the figures:

- Tree construction times decrease when MinRadius or MaxNumber increases. Clearly, increasing the granularity of the leaf-nodes decreases the depth of the tree and the number of intermediate nodes created, what have an impact in the construction times. For this phase, the optimal MinRadius times are up to 33% more efficient than MaxNumber times.
- However, OWM traversal times tend to increase when MinRadius (higher than 0.2) or MaxNumber increases. Now, increasing the granularity of the leaf-nodes requires to invest more time checking all the points stored in these nodes. Again, for this phase, optimal MinRadius times are more efficient than optimal MaxNumber times: up to 17% faster. Interestingly, more sparse clouds (Alcoy, Arzua) show little degradation of times when increasing the granularity, contrary to more dense clouds (BrionF, BrionU).
- Therefore, for each strategy and cloud there is a sweet point at which the total time is optimal. The MinRadius and MaxNumber values that achieve that optimal for each cloud are shown in Table 4, along with the *lev* parameter. In any case, the optimal MinRadius total times always outperform the optimal MaxNumber ones: from 1.08x to 1.17x. We have found that MinRadius generates shallower and wider trees populated with smaller leaf-nodes than MaxNumber, and that type of tree is better exploited (creation and traversal) on our target CPU.



(a) MinRadius changes between 0.1 and 1.9



(b) MaxNumber changes between 16 and 16584 in powers of 2

Fig. 11. Optimization O4: tree construction, OWM traversal and total times using 8 threads.

Table 4
Optimization O4: Optimal parameters for strategies MinRadius and MaxNumber.

Cloud	MinRadius		MaxNumber	
	lev	*MinRad* value	*lev*	*MaxNum* value
Alcoy	5	1.8	6	128
Arzua	5	0.5	6	256
BrionF	4	0.1	4	64
BrionU	4	0.1	4	64

- With these results augmented by some additional experimentation we can recommend as a rule of thumb to use level = 5 and minRad = 0.3, resulting in a maximum performance loss of 14.63% compared to the optimal configuration for each individual cloud. In addition, we should not fright about getting an approximation around these recommended values since the potential performance loss is not significant in most cases.

Once we have selected the optimal strategy, MinRadius, and tuned the optimal granularity for each cloud, we see in Table 2 the impact that the O4 optimization has in the tree construction, OWM traversal and total times when compared to O3 with MinRadius = 0.1. The incremental improvement in performance is higher in clouds with lower density (Alcoy, Arzua): up to 2.83x. However, in clouds with higher density (BrionF, BrionU) the incremental improvement is nonexistent because the baseline MinRadius = 0.1 ends up being the best one.

5.2.5. Summary of each optimization and scalability analysis on CPU

Table 5 summarizes the improvement factor of each optimization w.r.t. the parallel OpenMP baseline implementation (see Table 1), for tree construction, OWM traversal and total execution times. Note that optimizations are cumulative, this is, optimization 2 (O2) also includes optimization 1 (O1), and optimization 3 (O3) encompasses the two previous ones. We can see larger improvements thanks to O3 in the OWM traversal phase, which is originally the most time-consuming part of the algorithm, but with this optimization the tree construction can become the bottleneck. On the other hand, O2 and O4 also show significant improvements for the tree construction times.

Fig. 12 shows the relative weight of each optimization over the OpenMP baseline (in %) for the tree construction, OWM traversal and total times. This figure gives us some interesting insights about the impact of the optimizations regarding the density of the cloud. We see that the clouds with a lower density of points, Alcoy and Arzua, are more affected by optimization O4 (thanks to the relevant impact of this optimization in their tree construction phase), while the clouds with a higher density, BrionF and BrionU, are more affected by optimization O3 (thanks to its significant impact on the OWM traversal phase).

Finally, Fig. 13 shows the parallel speedup of the O4 TBB implementation for different number of cores. We also break down the speedup numbers for the tree construction and the OWM traversal phases. While the tree construction scales for all the clouds (although parallel efficiency drops below 60% on 8 cores), the traversal phase does not scale after 6 cores for the dense point clouds. Note that after all the implemented optimizations the scalability of this phase has hardly improved (see the 8 cores speedup of the OpenMP baseline implementation in the last column of Table 1). This is, both the sequential and the 8 cores execution times have improved at the same ratio (more than 11×). Although our optimizations have reduced the memory footprint and bandwidth requirements, the problem is still highly memory bound and in some cases, mainly for the denser Brion's clouds, where the trees are larger and deeper, adding cores does not help. In Section 5.3.5 we collect different metrics from the roofline model to discuss this issue in further detail.

5.3. Performance analysis on GPU

In this section we study now the impact of the GPU optimizations proposed in Section 4. For the evaluation we use now as reference

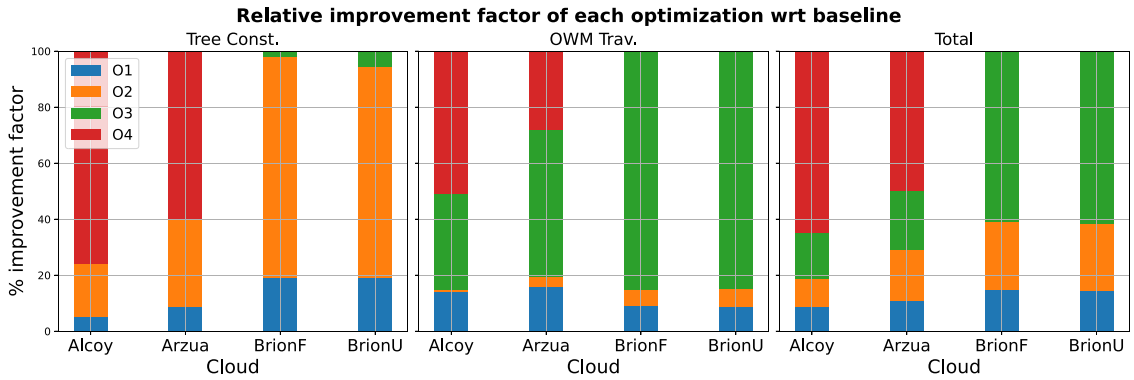


Fig. 12. Relative improvement factor of each optimization over the OpenMP baseline (%) using 8 threads.

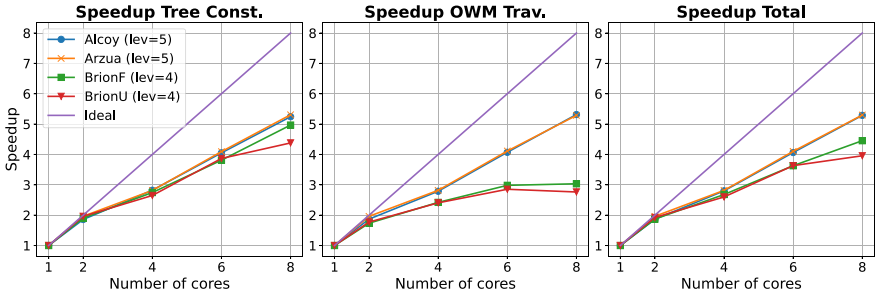


Fig. 13. Speedup plot for different core/thread counts.

Table 5
Improvement over the OpenMP baseline (see Table 1) for each optimization.

Cloud	O1			O2			O3			O4		
	Tree	OWM	Total	Tree	OWM	Total	Tree	OWM	Total	Tree	OWM	Total
Alcoy	1.49×	1.88×	1.64×	9.33×	2.01×	3.58×	6.81×	6.65×	6.74×	28.20×	13.54×	19.09×
Arzua	1.52×	2.42×	1.80×	8.41×	2.97×	4.77×	7.01×	10.93×	8.24×	17.47×	15.19×	16.44×
BrionF	1.48×	2.03×	1.72×	7.64×	3.31×	4.58×	7.79×	22.55×	11.70×	7.79×	22.55×	11.70×
BrionU	1.50×	1.88×	1.68×	7.36×	3.32×	4.51×	7.80×	21.75×	11.71×	7.80×	21.75×	11.71×

Table 6
Portability across devices for our implementations.

Prog. model/device	CPU	i-GPU	d-GPU
TBB	✓ (TBB base)	✗	✗
SYCL	✓ (S-CPU)	✓ (S-iGPU)	✓ (S-dGPU)
CUDA	✗	✗	✓ (CUDA)

TBB base, i.e., the optimal CPU times obtained after applying all optimizations discussed in the previous section (O1 to O4) to the TBB implementation using 8 threads, and compare them with the different GPU implementations described in Section 4.2: SYCL on iGPU and dGPU devices (S-iGPU, S-dGPU), and CUDA on the dGPU device (NVIDIA discrete GPU). We also include the SYCL implementation running on the CPU device (S-CPU). Table 6 summarizes the portability across devices for our implementations.

5.3.1. Optimizations 1 and 2 (O1 & O2): 2D-space bipartition and parallelization

Here we consider the impact of the 2D-space bipartition and parallelization optimizations in the GPU execution times, because these two features come together naturally and are actually a must in an efficient GPU implementation, as explained in Sections 4.1 and 4.2. The breakdown of the tree construction and OWM traversal times are shown in Fig. 14 for the CPU versions: TBB base and the SYCL CPU, S-CPU, as well as for the GPU versions: S-iGPU, S-dGPU, and CUDA. The SYCL and CUDA results do not include optimizations O3 (memoization) and O4 (tuning granularity) that will be studied in next subsections. In these results we consider as base criterion MaxNumber = 512 points per leaf-node.

Four main conclusions can be extracted from Fig. 14:

- The CUDA implementation running on the dGPU is the fastest for Brion's clouds, but the second fastest for Alcoy and Arzua. The total execution times improve from 1.39x to 3.56x when compared to the TBB base. Notably, tree construction times show the highest improvement, ranging from 11.34x (Alcoy) to 14.23x (BrionF). However, it is worth noting that the OWM traversal times for CUDA show a slight degradation of around 4% compared to the TBB base traversal times. It can be inferred from this result that the new tree construction phase, where the GPU oriented data structure is built, effectively takes advantage of the device architecture.
- From the GPU SYCL implementation (S-iGPU, S-dGPU), S-iGPU performs the worst due to the lower computational capabilities of the integrated GPU. On the other hand, S-dGPU is the fastest implementation for Alcoy and Arzua clouds, even faster than the native CUDA implementation. It achieves improvements of 2.49x and 1.80x, respectively. In the case of the Brion's clouds, S-dGPU achieves improvements of 2.90x and 2.86x for BrionF and BrionU, respectively, but it does not improve the CUDA

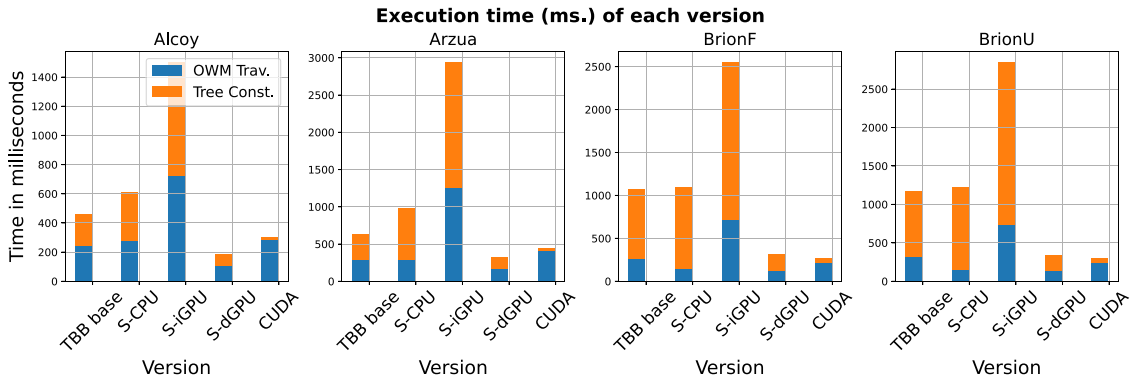


Fig. 14. Optimizations O1 & O2: Execution times comparison of TBB base, SYCL and CUDA implementations, without memoization and MaxNumber = 512.

implementation total times. Interestingly, in contrast to the CUDA implementation, S-dGPU shows a great improvement in the OWM traversal times that range from 1.41x to 2.27x when compared to TBB base.

- The efficiency of the SYCL implementation can be evaluated by comparing S-dGPU and CUDA. The SYCL abstraction layer introduces only a 4% degradation in the total execution time compared to CUDA on average, with slightly improved execution times for the Alcoy and Arzua clouds. The main source of inefficiency in the SYCL version lies in the tree construction phase, which can be up to 4 times slower than CUDA. This is attributed to the inefficient radix-sort operation called twice in our code, which cannot compete with the highly optimized CUDA counterpart specifically tailored to the NVIDIA architecture. However, as mentioned before, the OWM traversal times show a notable improvement over CUDA for all clouds, ranging from 25% to 61% faster. Despite the fixed scenario of MaxNumber=512 without memoization, which can lead to higher warp divergence on the NVIDIA architecture, the SYCL driver handles this inconvenience more effectively.
- The portability of the SYCL implementation can be analyzed by comparing TBB base and S-CPU. Let us recall that the data structure constructed in TBB is based on a quadtree, whereas the SYCL version builds a radix tree based on Morton codes, so these phases are not comparable, because while the second implementation exhibits higher parallelism, it does so by increasing the computational cost. However, the OWM traversals are similar, although memoization and tuning of granularity optimizations have not yet been applied in the SYCL implementation. Even though, for this phase, the SYCL implementation is already on average 25% faster than TBB base, thanks precisely to the less memory pressure that the traversal of the SYCL data structure offers and because the SYCL compiler does a better job exploiting the CPU SIMD units.

Due to the poor performance of the S-iGPU version, we skip this one in the rest of our analysis in the next subsections, unless otherwise noted.

5.3.2. Optimization 3 (O3): reduce the number of accesses by memoization

In this subsection we study the impact of the memoization strategy presented in Section 4.3, where we highlight again that only the saving min approach is applicable (case (b)). OWM traversal times are reduced thanks to this optimization. Table 7 shows the OWM traversal times before and after the memoization strategy has been applied. As we see, the incremental improvement (O3 vs. O2) is higher in the clouds with higher density for all versions. Specifically, for the SYCL versions, we observe up to a 17% and 53% gain for S-CPU and S-dGPU, respectively. The CUDA version also benefits from memoization, with improvements up to 13%. Overall, this optimization accounts for up to 12% and 5% improvement in the total execution times for S-dGPU and CUDA, respectively, while providing moderate improvements for the S-CPU version.

5.3.3. Optimization 4 (O4): tune the granularity

Here we evaluate the impact of the strategy for tuning the granularity of the tree leaf-nodes (see Section 4.4) in the GPU implementations. For it, we explore all possible values for the MaxNumber parameter: from 4 to 1024, and in Table 8 we report the MaxNumber value (MaxN) that gives the best performance in each cloud. Using that optimal value, we also report the incremental improvement (Total) for both the tree construction and the OWM traversal phases for S-CPU, S-dCPU and CUDA.

Table 8 shows that the optimal MaxN parameter is smaller than the default value of 512 used in the previous optimization studies. This leads to deeper data structures and, consequently, a more costly tree construction phase. However, the degradation of the tree construction times is relatively small, although more noticeable for S-dGPU and CUDA, where even smaller MaxN values are found to be optimal. The discrete GPU device is more likely to reduce warp divergences and improve memory coalescence with these smaller MaxN values. Optimization O4 has a more significant impact on the OWM traversal phase, particularly on the versions running on the discrete GPU, where the improvement reaches up to 1.61x for S-dGPU and 2.80x for CUDA. In summary, optimization O4 accounts for up to 6%, 15%, and 136% incremental improvement in the total execution times for S-CPU, S-dGPU, and CUDA, respectively, with the most substantial gains observed in the CUDA version.

5.3.4. Overall improvement factor of SYCL and CUDA implementations

Fig. 15 summarizes the total improvement of the SYCL and CUDA implementations when all optimizations have been incorporated (from O1 to O4) with respect to the best TBB base code. In the figure, we also include S-CUDA, a hybrid SYCL-CUDA implementation that embeds CUDA kernel calls inside the SYCL code using the SYCL interoperability features (see Section 4.2) and run on the discrete GPU.

As we see in Fig. 15, after applying all optimizations we outline some relevant findings:

- Optimizations O3 and O4 improve OWM traversal times in both SYCL and CUDA implementations, as explained earlier. However, the S-CPU version does not benefit from the new GPU-oriented tree construction implementation, which aligns with the discussion in Section 5.3.1. The SYCL implementation exhibits higher parallelism but incurs increased computational complexity, resulting in a 28% average performance degradation w.r.t. the CPU oriented TBB implementation. With this, the tree construction is the bottleneck in the CPU (TBB baseline and S-CPU), whereas the OWM traversal is the bottleneck in the CUDA versions (S-CUDA and CUDA), leaving the S-dGPU version between the other two

Table 7
Optimization O3: OWM traversal times in ms with (M) and without memoization (NM) and improvement factor (O3 vs. O2).

Cloud	S-CPU			S-dGPU			CUDA		
	NM	M	O3 vs. O2	NM	M	O3 vs. O2	NM	M	O3 vs. O2
Alcoy	279.64	278.18	1.01×	108.17	106.17	1.02×	279.69	282.07	0.99×
Arzua	324.69	294.20	1.10×	204.09	172.44	1.18×	419.25	407.55	1.03×
BrionF	174.05	149.03	1.17×	184.22	125.30	1.47×	245.52	219.86	1.12×
BrionU	184.44	157.15	1.17×	208.64	135.93	1.53×	268.15	237.50	1.13×

Table 8
Optimization 4 (O4): Improvement factor (O4 vs. O3) using the best MaxNumber (MaxN) in O4, in the tree construction (Tree), OWM traversal (OWM) and total times (Total).

Cloud	S-CPU				S-dGPU				CUDA			
	MaxN	Tree	OWM	Total	MaxN	Tree	OWM	Total	MaxN	Tree	OWM	Total
Alcoy	64	0.97×	1.16×	1.05×	64	0.89×	1.28×	1.08×	16	0.77×	2.60×	2.26×
Arzua	128	0.98×	1.35×	1.06×	64	0.90×	1.47×	1.13×	32	0.85×	2.80×	2.36×
BrionF	128	1.00×	1.49×	1.05×	128	0.95×	1.61×	1.14×	32	0.88×	2.47×	1.81×
BrionU	128	1.01×	1.43×	1.05×	128	0.97×	1.59×	1.15×	32	0.90×	2.45×	1.80×

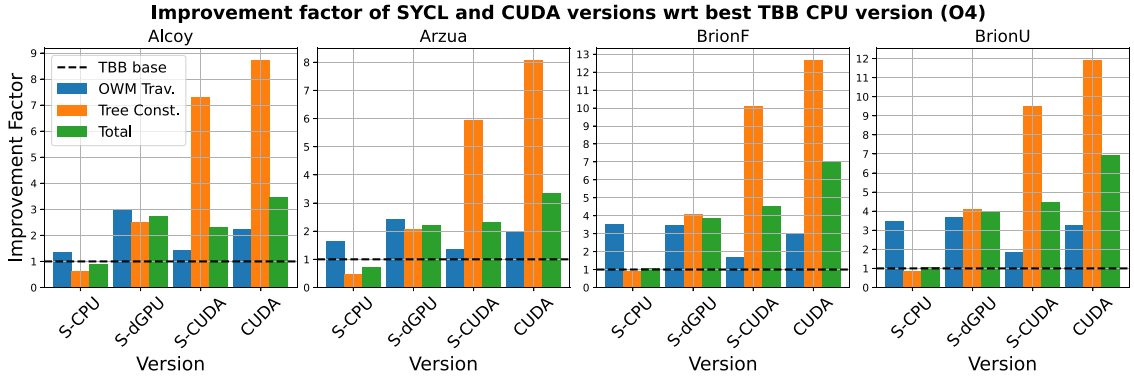


Fig. 15. Total improvement factor of SYCL and CUDA implementations w.r.t. best TBB CPU base.

cases (tree construction and OWM traversal require almost the same time).

- If you would rather target a GPU device (be it from NVIDIA, Intel, or AMD), SYCL is a viable choice in our case study. For the NVIDIA device, the native CUDA implementation still delivers the fastest performance, with speedups ranging from 3.36x to 7.05x compared to TBB on the CPU. However, the SYCL S-dGPU implementation is quite competitive, outperforming CUDA in the OWM traversal by 21% on average. Despite being 36% slower than CUDA in terms of total execution time, the SYCL implementation offers the advantage of portability across different GPU vendors. Developers who prioritize code portability, and wish to avoid vendor lock-in, may find the SYCL implementation more appealing, even with the performance trade-off.
- If on the contrary a multicore CPU suits your needs, your GPU oriented SYCL implementation is valid as well. The SYCL S-CPU implementation outperforms the best TBB by 150% in the OWM traversal. However, the S-CPU implementation experiences a slight degradation in overall performance, with an average 5% increase in the total execution time. Note that S-CPU is more competitive with denser clouds (Brion).
- S-CUDA suffers from the degradation induced by the runtime of SYCL when compared to the pure CUDA version. The SYCL runtime introduces two sources of overhead: the invocation of the CUDA kernels via the interoperability functionality, and the data movement from/to host-device through SYCL functions. This degradation accounts for around 50% of the total time. Anyway, the S-dGPU can be faster than S-CUDA (see Alcoy) and only 3% slower than S-CUDA on average (attributed primarily to the more efficient CUDA-based tree construction phase). For that reason, we only recommend an S-CUDA implementation as an intermediate step in the translation of an existing CUDA code to SYCL.

With all this we can conclude that the SYCL programming model, in our testbed, is getting closer to the "performance portability dream". A single SYCL implementation compiled for two devices, S-CPU and S-dGPU, reports competitive execution times (compared to highly optimized CPU-oriented, TBB, and GPU-oriented, CUDA, implementations).

For the sake of completeness, in Table 9 we summarize the total improvement of the SYCL and CUDA implementations with respect to the original baseline implemented in OpenMP with 8 threads. These results can be compared with those shown for TBB in Table 5. The best CPU version (TBB with O4) can process between 40 to 68 million points per second, whereas the best GPU version (CUDA) improves this throughput to the range 153–285 million points per second. This is a huge improvement with respect to the 3 million LiDAR points processed per second reported in [9] using Semi-Global Filtering (SGF) also on GPU.

5.3.5. Limiting factors and power efficiency considerations for TBB, SYCL and CUDA implementations

In order to better understand the performance bottlenecks for each implementation (TBB, SYCL and CUDA) and how much performance is left on the table because of them on each device, we carried out a roofline analysis for TBB and S-CPU using the Intel Advisor profiler, as well as for CUDA using the NVIDIA Nsight tool. We could not get the S-dGPU data because Nsight does not support SYCL. Table 10 shows the roofline data for the function that represents the hotspot in the tree creation (Tree) and traversal (OWM) phases, respectively. From the table, we see that the Arithmetic Intensity (AI) metric confirms the memory bound nature of the algorithm in the different implementations: almost all the analyzed functions are capped by the memory

Table 9
Total improvement factor of SYCL and CUDA implementations w.r.t. OpenMP CPU baseline.

Cloud	S-CPU			S-dGPU			S-CUDA			CUDA		
	Tree	OWM	Total	Tree	OWM	Total	Tree	OWM	Total	Tree	OWM	Total
Alcoy	14.02×	20.56×	16.30×	55.02×	44.95×	50.08×	160.28×	21.81×	42.28×	191.99×	34.26×	63.49×
Arzua	8.08×	23.50×	11.11×	34.35×	34.68×	34.48×	98.03×	19.21×	36.24×	133.70×	28.03×	52.10×
BrionF	6.57×	84.41×	12.69×	29.57×	82.94×	44.58×	73.62×	40.57×	51.62×	92.31×	72.33×	80.65×
BrionU	6.63×	80.99×	12.82×	32.00×	86.33×	47.83×	74.20×	43.12×	53.81×	92.96×	76.21×	83.33×

Table 10
Roofline metrics for TBB, S-CPU and CUDA: Arithmetic Intensity -AI-, memory/compute ceiling -Cap- and Attainable Performance -AP-; for Alcoy and BrionU: Performance -Perf- and headroom to the attainable Peak in % (%2Peak, the lower the better).

Metric	TBB		S-CPU		CUDA	
	Tree	OWM	Tree	OWM	Tree	OWM
AI (FLOP/Byte)	0.003	0.071	0.075	0.41	0.85	0.10
Cap (mem./compute)	L3	L3	L3	DDR4	FP32	GDDR6
AP (GFLOPS)	1.88	40	42	15.7	165	32
Alcoy						
Perf (GFLOPS)	0.23	4.67	5.2	8.68	104	4.6
%2Peak	87%	88%	87%	44%	40%	85%
BrionU						
Perf (GFLOPS)	0.2	1.27	5.76	14.3	104	6.4
%2Peak	89%	96%	86%	9%	40%	80%

system (either the L3 cache or the main memory -DDR4 on the CPU, GDDR6 on the discrete GPU-). Only the tree creation phase of the CUDA implementation is compute bound (capped by the FP32 unit) for our discrete GPU. We also show the Attainable Performance (AP) for the corresponding AI/Cap because that is the peak performance that each function could achieve in the corresponding device. Then we show the achieved Performance (Perf) and the headroom to the attainable Peak performance (%2Peak) for a sparse and a dense cloud (Alcoy, BrionU), respectively.

One important insight that AI gives is that the implementation targeting the GPU (CUDA) reduces significantly the traffic with memory compared to TBB, and therefore it increases the room for better performance. However, the traversal in this implementations is capped by the main memory ceiling, what hints that there could be explored more aggressive strategies for exploiting the memory hierarchy in the traversal of our binary radix tree based on Morton codes. On the other hand, the traversal of the quadtree in the TBB implementation already partially exploits cache hierarchy (it is capped by L3). Still, the measured performance of the TBB traversals in our clouds is far from the peak and the %2Peak metric says that there is room for locality improvement, in particular for the dense data clouds.

The Perf metric gives one important insight discussed earlier: the tree construction phase is the bottleneck in TBB and S-CPU, while the OWM traversal is the bottleneck in CUDA. However, the AP metric tells that the SYCL compiler has room for improvement for the tree construction in S-CPU, and in that ideal case the OWM traversal would be the bottleneck in both the SYCL and CUDA versions.

We also want to compare the different architectures in terms of power efficiency. Despite the limited computational resources available on the iGPU, the S-iGPU version achieves impressive efficiency values ranging from 1.11 to 1.34 Mpoints/s/W across the tested clouds. These efficiency figures surpass the best TBB CPU version, which ranges from 0.41 to 0.68 Mpoints/s/W, and are comparable to the efficiency of the CUDA version, which ranges from 0.96 to 1.78 Mpoints/s/W. The high power efficiency of the S-iGPU version can be attributed to the low power consumption of the device, with a TDP of only 15 W, compared to the 95 W TDP of the CPU and the 160 W TDP of the dGPU. This power efficiency makes the S-iGPU version an attractive option for scenarios where energy consumption is a critical concern, such as in mobile or embedded systems, or in large-scale deployments where cumulative power savings can be significant.

6. Conclusions

We have explored different optimization strategies adapted to the specific capabilities of the devices that we target in this study (CPU, iGPU and dGPU), as well as different programming environments that currently represent the state-of-the-art in heterogeneous programming (OpenMP, TBB, SYCL, CUDA) using as case study a C++ tree based application that processes LiDAR data point clouds (the OWM function).

Regarding the optimizations proposed in this work, our results hint that optimizations oriented to minimize memory access such as O3 (memoization) and O4 (tuning granularity) have a greater impact on the CPU, while optimizations oriented to tune the tree data structure to the device (O1) and exploit all available parallelism (O2) have a greater impact on the GPU.

Regarding the programming environments, SYCL is a promising abstraction that promotes programmers productivity by enabling the execution over different heterogeneous devices (CPU, integrated GPU or discrete GPU) with a single source code. Moreover, we have found that SYCL is highly competitive (mainly if you care about code portability) with device-specific programming environments such as TBB (CPU) or CUDA (GPU) for the type of application studied here. For future research lines we will consider hybrid implementations that use the CPU and the GPU at the same time, as well as the evaluation of the implementation in low-power devices that can be coupled with the LiDAR sensor.

CRediT authorship contribution statement

Felipe Muñoz: Formal analysis, Investigation, Methodology, Software, Writing – original draft, Writing – review & editing, Validation. **Rafael Asenjo:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Software, Supervision, Writing – original draft, Writing – review & editing, Visualization. **Angeles Navarro:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Project administration, Supervision, Visualization, Writing – original draft, Writing – review & editing. **J. Carlos Cabaleiro:** Conceptualization, Data curation, Funding acquisition, Investigation, Resources, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data and codes that support the findings of this study are available at https://github.com/PPMC-DAC/FastOWM.

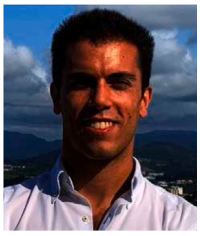
Acknowledgments

This work was supported by the Ministry of Economy and Competitiveness, Government of Spain (Grant Numbers PID2019-104834GB-I00, PID2022-141623NB-I00 and TED2021-131527B-I00), Junta de Andalucía, Spain (Grant Number P20-00395-R) and National FPU Grant FPU20/03735. This work has received financial support from the Consellería de Cultura, Educación e Ordenación Universitaria, Spain (accreditation 2019–2022 ED431G-2019/04, reference competitive group 2019–2021, ED431C 2022/16) and the European Regional Development Fund (ERDF), which acknowledges the CiTIUS-Research Center in Intelligent Technologies of the University of Santiago de Compostela as a Research Center of the Galician University System.

The authors would like to thank Babcock International for providing the point clouds and LaboraTe group (USC) for their help. Funding for open access charge: Universidad de Málaga / CBUA. The authors report there are no competing interests to declare.

References

[1] J. Shan, C.K. Toth (Eds.), Topographic Laser Ranging and Scanning: Principles and Processing, second ed., CRC Press, 2018.
[2] A.L. Gil, L. Núñez-Casillas, M. Isenburg, A.A. Benito, J.J.R. Bello, M. Arbelo, A comparison between LiDAR and photogrammetry digital terrain models in a forest area on tenerife island, Can. J. Remote Sens. 39 (5) (2013) 396–409.
[3] S. Buján, M. Cordero, D. Miranda, Hybrid overlap filter for LiDAR point clouds using free software, Remote Sens. 12 (2020) 1051, http://dx.doi.org/10.3390/rs12071051.
[4] Intel Corporation, oneAPI specification 1.2 rev. 1, 2023, https://spec.oneapi.io/versions/1.2-rev-1/. (Accessed 01 May 2023).
[5] J. Reinders, B. Ashbaugh, J. Broadman, M. Kinsner, J. Pennycook, X. Tian, Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL, A Press, 2021.
[6] The Khronos SYCL Working Group, SYCL 2020 specification (revision 7), 2023, https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html. (Accessed 01 May 2023).
[7] X. Meng, N. Currit, K. Zhao, Ground filtering algorithms for airborne LiDAR data: A review of critical issues, Remote Sens. 2 (3) (2010) 833–860.
[8] Z. Chen, B. Gao, B. Devereux, State-of-the-art: DTM generation using airborne LiDAR data, Sensors 17 (1) (2017) http://dx.doi.org/10.3390/s17010150.
[9] X. Hu, L. Ye, S. Pang, J. Shan, Semi-Global filtering of airborne LiDAR data for fast extraction of digital terrain models, Remote Sens. 7 (2015) 10996–11015, http://dx.doi.org/10.3390/rs70810996.
[10] X. Hu, X. Li, Y. Zhang, Fast filtering of LiDAR point cloud in urban areas based on scan line segmentation and GPU acceleration, IEEE Geosci. Remote Sens. Lett. 10 (2) (2013) 308–312, http://dx.doi.org/10.1109/LGRS.2012.2205130.
[11] J.M. Sánchez, Á.V. Álvarez, D.L. Vilariño, F.F. Rivera, J.C. Cabaleiro, T.F. Pena, Fast ground filtering of airborne LiDAR data based on iterative scan-line spline interpolation, Remote Sens. 11 (2019) 23, http://dx.doi.org/10.3390/rs11192256.
[12] V. Fursov, Y. Goshin, A. Kotov, The hybrid CPU/GPU implementation of the computational procedure for digital terrain models generation from satellite images, Comput. Opt. 40 (5) (2016) 721–729, http://dx.doi.org/10.3390/rs70810996.
[13] van der Merwem Dirk, J. Meyer, Towards automatic digital surface model generation using a graphics processing unit, in: AFRICON 2009, 2009, pp. 1–6, http://dx.doi.org/10.1109/AFRCON.2009.5308102.
[14] M. Voss, R. Asenjo, J. Reinders, Pro TBB : C++ Parallel Programming with Threading Building Blocks, A Press, 2019, p. 822.
[15] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast BVH construction on GPUs, Comput. Graph. Forum (2009) http://dx.doi.org/10.1111/j.1467-8659.2009.01377.x.
[16] K. Garanzha, J. Pantaleoni, D. McAllister, Simpler and faster HLBVH with work queues, in: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 59–64, http://dx.doi.org/10.1145/2018323.2018333.
[17] T. Karras, Maximizing parallelism in the construction of BVHs, octrees, and k-d trees, in: Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics, in: EGGH-HPG'12, Eurographics Association, Goslar, DEU, 2012, pp. 33–37.
[18] T. Karras, T. Aila, Fast parallel construction of high-quality bounding volume hierarchies, in: Proceedings of the 5th High-Performance Graphics Conference, HPG '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 89–99, http://dx.doi.org/10.1145/2492045.2492055.
[19] M. Bern, D. Eppstein, S.-H. Teng, Parallel construction of quadtrees and quality triangulations, Internat. J. Comput. Geom. Appl. 09 (06) (1999) 517–532.
[20] S. Smith, D. Holland, P. Longley, The importance of understanding error in LiDAR digital elevation models, Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci. 35 (2004) 996–1001.



Felipe Muñoz received the engineering degree in telecommunications systems in 2019 from the University of Málaga and the Master degree in high performance computing in 2020 from the University of Santiago de Compostela. He is currently pursuing a Ph.D. degree in mechatronics engineering in the Department of Computer Architecture at the University of Málaga. His research interests include high performance computing, computer architecture, time series analysis, and the application of statistical physics and information theory to algorithm design and optimization for low-power devices.

Rafael Asenjo is Professor of Computer Architecture at the University of Málaga. He obtained a Ph.D. in Telecommunication Engineering in 1997. He has been using TBB since 2008 and over the last five years, he has focused on productively exploiting heterogeneous chips leveraging TBB as the orchestrating framework. In 2013 and 2014 he visited UIUC to work on CPU+GPU chips. In 2015 and 2016 he also started to research into CPU+FPGA chips while visiting the University of Bristol. He served as General Chair for ACM PPoPP'16 and as an Organization Committee member as well as a Program Committee member for several HPC related conferences (PPoPP, SC, PACT, IPDPS, HPCA, EuroPar, and SBAC-PAD). His research interests include heterogeneous programming models and architectures, parallelization of irregular codes and energy consumption. He co-authored the latest book (open access) on Threading Building Blocks (Pro TBB), is oneAPI Innovator, SYCL Advisory Panel member and ACM member.

Angeles Navarro obtained a Ph.D. in Computer Science from the Universidad de Málaga, Spain, in 2000. She is a Full Professor in the Department of Computer Architecture at Universidad de Málaga. She has been a Research Visiting Scholar in the University of Illinois at Urbana-Champaign (UIUC), the Technical University of Munich (TUM), the EPCC at the University of Edinburgh, the University of Bristol, and a Research Visitor in IBM T.J. Watson Research Center at New York and in Cray Inc at Seattle. She has served as a program committee member for several High Performance Computing related conferences as PPoPP, SC, ICS, PACT, IPDPS, ICPP, EuroPar, ISPA and ISC. She is the co-leader of the Parallel Programming Models and Compilers group at the Universidad de Málaga. Her research interests are in programming models for heterogeneous systems, analytical modeling, compiler and runtime optimizations.

J. Carlos Cabaleiro got a BS in Physics at the University of Santiago de Compostela (Spain) in 1989 and obtained a Ph.D. in Physics at the same university in 1994. From 1990 until 1994 he was an associate professor in the Faculty of Computing of the University of A Coruña (Spain). From 1994 he was an associate professor in the area of Computer Architecture in the Department of Electronics and Computing at the University of Santiago de Compostela, until 2022, when he became a full professor. Since 2010 he has been a member of CiTIUS. His research interests include the development of parallel applications like 3D point cloud processing, prediction and improvement of the performance of parallel systems, development of applications and middleware for cloud, and Big Data technologies.