Cell-based Update Algorithm for Occupancy Grid Maps and Hybrid Map for ADAS on Embedded GPUs

Jörg Fickenscher, Jens Schlumberger, Frank Hannig, Jürgen Teich
Hardware/Software Co-Design, Department of Computer Science,
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany

Mohamed Essayed Bouzouraa
Concept Development Automated Driving,
Audi AG, Ingolstadt, Germany

***Abstract*—Advanced Driver Assistance Systems (ADASs), such as autonomous driving, require the continuous computation and update of detailed environment maps. Today's standard processors in automotive Electronic Control Units (ECUs) struggle to provide enough computing power for those tasks. Here, new architectures, like Graphics Processing Units (GPUs) might be a promising accelerator candidate for ECUs. Current algorithms have to be adapted to these new architectures when possible, or new algorithms have to be designed to take advantage of these architectures. In this paper, we propose a novel parallel update algorithm, called cell-based update algorithm for occupancy grid maps, which exploits the highly parallel architecture of GPUs and overcomes the shortcomings of previous implementations based on the Bresenham algorithm on such architectures. A second contribution is a new hybrid map, which takes the advantages of the classic occupancy grid map and reduces the computational effort of those. All algorithms are parallelized and implemented on a discrete GPU as well as on an embedded GPU (Nvidia Tegra K1 Jetson board). Compared with the state-of-the-art Bresenham algorithm as used in the case of occupancy grid maps, our parallelized cell-based update algorithm and our proposed hybrid map approach achieve speedups of up to 2.5 and 4.5, respectively.**

I. Introduction and Related Work

ADASs and autonomous driving are becoming more and more important the car makers. Besides electrical vehicles, most innovations in the automotive sector arise in the area of ADASs at the moment, and customers are looking for the best driving experience not only in the premium segment. To realize such functionality, for example, an adaptive cruise control or in the future autonomous driving, a detailed knowledge of the vehicle's environment is necessary any time. Such information is stored in environment maps, like the *occupancy grid map* [4]. Various sensors, such as radar and lidar are used for creating detailed environment maps. These sensors produce an enormous amount of data, which has to be processed and fused. At this moment, the challenge is that most of the ECUs in nowadays cars have only a single core Central Processing Unit (CPU), which fails to provide the required computing power for those tasks. Another hitch is that today's performance gains are achieved mostly by more cores and not by a higher single core performance. Here, new architectures, like embedded GPUs emerge, which have hundreds of cores. Yet to successfully employ such architectures, the predominant single-threaded programming model in the automotive industry has to be switched to a multithreaded programming model. Thus, existing sequential algorithms have to be either adapted to the multithreaded programming model or new parallel algorithms have to be developed from scratch. In this realm, we propose in this paper a new update algorithm for the occupancy grid map, which takes advantage of the highly parallel architecture of a GPU. Our algorithm overcomes the disadvantages of current state-of-the-art update algorithms, such as the Bresenham algorithm [2] when executed on many-core architectures. Another difficult task when creating occupancy grid maps is their granularity. Fine-grained grids, like the occupancy grid map, require a high computational effort, because regions farther away from the own vehicle have the same resolution as areas close to the car, but usually do not need that level of detail. Maps that entirely consist of coarse-grained grid cells instead may miss relevant information in the environment. Thus, a perfect solution would be to combine both approaches into a *hybrid map*, which has a higher resolution close to the vehicle and a smaller resolution for areas farther away from the vehicle. The unique selling point of our proposed hybrid map is that it is particularly designed for the characteristics of GPUs.

The remainder of the paper is structured as follows: In the next paragraph, we discuss the differences of our approach compared to related work. In Section II, an overview of our novel update algorithm and the new hybrid map is given as well as a brief introduction to programming embedded GPUs. In Section III, experimental results are presented and discussed. Finally, we conclude the paper in Section IV.

The research for environment maps started nearly three decades ago for mobile robots. In 1989, Elfes introduced the occupancy grid map [4] for robotics and his approach is now one of the standards in environment mapping [16]. It is also used by Simultaneous Localization and Mapping (SLAM) algorithms [8] and for motion planning of robots [10]. Elfes 2D approach was extended to a third dimension in [13]. Besides robotics, environment maps are also used in the automotive sector, e.g., for lane detection [11], to compute the free space [1] of the environment for path planning or to avoid dynamic obstacles [7]. Homm et al. [9] parallelized an occupancy grid map for automotive use for the first time on a desktop GPU. Instead, we do evaluate our algorithms also on an embedded GPU, which comes close to those used later in vehicles. To update the occupancy grid map, the authors in [9] used a Bresenham algorithm [2]. All other works, like, e.g., [5], [21], [18], which are dealing with updating an occupancy grid map, used also a Bresenham algorithm. The work in [15] introduced an improved Bresenham algorithm to update the occupancy grid map. Our approach presented in the following rather updates every cell by looking back to the laser scanner. A disadvantage of the occupancy grid map is that it has the same resolution over the whole environment map. A few works dealt with this problem. In [3], an occupancy grid map was introduced, whose cell sizes are not determined before the creation of the map. Instead, the cell size is determined on the fly depending on the sensor measurements. Normally, all grid cells have the same length in all directions of a 3D occupancy grid, but in [17], a concept was introduced, where the dimension in height can have different sizes, depending on the probability if there is an obstacle or not. Some works also tried to reduce the memory consumption of 3D grid maps. In [20] and [12], an

978-3-9819263-0-9/DATE18/©2018 EDAA

octree and a quadtree were used, respectively. Some works also changed nearly the concept of an occupancy grid map. In [19], the environment is discretized in the longitudinal direction like by occupancy grid maps. Instead, a continuous value is stored in the lateral direction. This approach was used for highway scenarios and was parallelized for an embedded GPU by [5]. Yet, a big disadvantage of all the works above is that all objects have to be separately compensated by the ego-motion of the vehicle. Contrary, in our proposed hybrid map, we can compensate the ego-motion by a 2D ring buffer such as in traditional occupancy grid maps, which is shown to be much faster.

II. Methods

A. Occupancy Grid Mapping

An occupancy grid map represents the environment around a vehicle through a fine-grained grid. For this purpose, the environment is rasterized in squares, so-called grid cells. For every grid cell, a probability is calculated, whether the cell is occupied or free, based on sensor measurements. Typically, the posterior probability is used in the *occupancy grid map algorithm* [16]

$$p(m|z_{1:t}, x_{1:t}) \tag{1}$$

where m is the map, $z_1, ..., z_t$ are the measurements from the first to the measurement at time t, and x denotes the corresponding known poses of the vehicle also from the first to the measurement at time t. Due to the high-dimensional space, the posterior cannot be determined easily. Thus, the problem is reduced to calculate the posterior of each grid cell separately:

$$p(m_i|z_{1:t}, x_{1:t}) \tag{2}$$

Due to numerical instabilities close to probabilities near zero or one, the so-called log-odds form is applied:

$$p(m_i|z_{1:t}, x_{1:t}) = \log \frac{p(m_i|z_{1:t}, x_{1:t})}{1 - p(m_i|z_{1:t}, x_{1:t})} \tag{3}$$

In addition, to eliminate some terms that are hard to compute, the Bayes' rule is applied to the posterior $p(m_i|z_{1:t}, x_{1:t})$. Finally, we get:

$$p(m_i|z_{1:t}, x_{1:t}) = \frac{p(z_t|z_{1:t-1}, m_i) \cdot p(m_i|z_{1:t-1})}{p(z_t|z_{1:t-1})} \tag{4}$$

An occupied grid cell has the probability $p(m_i) = 1.0$ and an empty grid cell $p(m_i) = 0.0$.

B. Programming Embedded GPUs

GPUs are most efficiently used for solving problems with a high arithmetic density and where data parallel computing is possible. In former times, this meant mostly image processing. But nowadays, GPUs can be used for a broad variety of compute-intensive problems, such as artificial intelligence or numerical flow simulation. The difference between a parallelized algorithm for a CPU and a GPU is that on CPUs, different tasks on different data are executed in parallel. Instead, on a GPU, the same instruction is executed in parallel, but on different data. GPU architectures are quite different from CPU architectures. GPUs consist on the hardware level of several *streaming processors*, which further contain multiple processing units. A streaming processor spawns, manages, and executes threads in groups of 32, so-called *warps*. In a warp, every thread executes the same instruction at the same time but on different data. This execution model is called Single Instruction, Multiple Thread (SIMT), which is close to Single Instruction, Multiple Data (SIMD). As general-purpose programming paradigm for GPUs, Nvidia introduced the CUDA framework [14] in 2006. Program blocks, which should be executed in parallel on a GPU, are called *kernels*. Kernels in CUDA are similar to functions defined over an iteration space in the programming language C. Every kernel has its own grid. They are executed N times in parallel by N CUDA threads. Multiple threads are combined to logical blocks and blocks are combined to a logical grid. One important difference between an embedded and a desktop GPU is the memory architecture. On a desktop GPU, the memory for the GPU is separated from the system memory, and the data has to be explicitly copied to the GPU and back (e.g., over PCI Express). Instead, in an MPSoC with an embedded GPU (such as Nvidia Tegra, ARM Mali, or Qualcomm Adreno), the processor system and the GPU share the same memory (also known as unified memory architecture, see, e.g., [6] for further details). Thus, no explicit data transfers to the GPU and back are necessary. Consequently, memory transfer times can be saved.

C. Cell-based Update Algorithm

The state-of-the-art algorithm for updating occupancy grid maps is the Bresenham line drawing algorithm [2]. The algorithm selects the cells which have to be updated in the occupancy grid map based on the sensor measurements. For each laser beam of the sensor, a line is drawn in the occupancy grid map. The line is drawn between the origin of the laser beam, the sensor, and the measured obstacle, indicated by the black arrows in Fig. 1.

The reason for the wide usage is because the algorithm uses only integer arithmetics and is quite efficient. Another advantage is that also rounding errors are quite small. However, on GPUs, the algorithm does not perform well due to several reasons. One is the non-coalesced memory access. The grid cells of the occupancy grid map are stored in the memory as indicated by the yellow arrows in Fig. 1. In our example, one object was recognized by a sensor. For updating the occupancy grid map, a CUDA thread is started for every laser beam of the sensor. If there is no obstacle in the grid cell, the cell is marked in green. If there is an obstacle in the grid cell, the cell is marked in red. To indicate no measurement, i. e., the occupancy is unknown, the cell is marked in brown. For every grid cell which has to be updated, a new line (corresponding

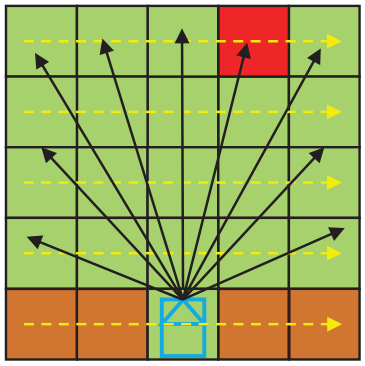

Figure 1: Schematic figure of the non-coalesced memory access on a GPU, using a Bresenham algorithm to update an occupancy grid map. The black arrows show which cells need to be accessed and updated in case a Bresenham algorithm is used. The yellow arrows indicate the storage of the cells in the memory.

Algorithm 1 Overview over the cell-based update algorithm.

```
1: Input: Vector vel  // velocity vector
          LaserDataArray laserData  // Array with the laser measurements
          float* matrixOld  // Occupancy grid map before updated with the new laser measurement
2: Output: float* matrixNew  // Occupancy grid map updated with the new laser measurement
3: function PUTLASERDATAINTOMATRIX(Vector vel, LaserDataArray laserData, float* matrixNew, float* matrixOld)
4:     for (y=verticalMin; y ≤ verticalMax; y++) do
5:         for (x=horizontalMin; x ≤ horizontalMax.x; x++) do
6:             float newValue = getCellState(laserData, x, y, vel)
7:             int indexOldValue = getGridCoordsIndex(x-vel.x, y-vel.y)
8:             float oldValue = matrixOld[indexOldValue]
9:             float updatedValue = 1/(1+exp(log((1−newValue)/newValue)+log((1−oldValue)/oldValue)))
10:            int gridCurIndex = getGridCoordsIndex(x, y)
11:            matrixNew[gridCurIndex] = updatedValue
12:        end for
13:    end for
14: end function
```

Algorithm 2 The core function of the cell-based update algorithm. It determines the laser ray which goes through a given grid cell.

```
1: Input: Vector vel, LaserDataArray laserData, float* matrixOld
2: Output: LaserDataArray laserData
3: function GETRAYFORCOORDINATES(LaserDataArray laserData, float x, float y, Vector vel)
4:     float angleBetweenVectors_deg = convToDegree(arccos((x·vel.x+y·vel.y)/(√(x²+y²)·√(vel.x²+vel.y²))))
5:     float rotationDirection = y·vel.x − x·vel.y
6:     if (rotationDirection < 0) then
7:         angleBetweenVectors_deg = angleBetweenVectors_deg · (-1)
8:     end if
9:     int rayIndex = angleBetweenVectors_deg / resolutionOfRays + (numberOfRays-1)/2
10:    return laserData[rayIndex]
11: end function
```

to the beam/ray of the laser, see Fig. 1) has to be loaded from memory. Because of the irregular memory access for different laser beams, the data is hardly cacheable. If the sensor would be on the entire left side (parallel projection), the Bresenham update would be executed over the yellow arrows, and for the whole update, only one memory line would have to be loaded into the cache to draw one line. Our algorithm instead updates the occupancy grid map by determining for each occupancy grid cell if a sensor measurement hits it or not. For every grid cell, a CUDA thread is started that calculates by which laser beam the occupancy grid cell was hit. Another challenge is the warp efficiency of the Bresenham algorithm. Let n threads in a warp draw the lines respective update the cells of the occupancy grid map between the n beams of a laser sensor measurement z_t and the origin of the beams at the sensor. Every thread in that warp has to wait until the thread has finished updating the grid map with the measurement farthest away from the sensor. This means in the worst case 31 threads of the warp are idle because they have a laser measurement close to the sensor and have to wait until the thread with the measurement farthest away has finished. A further advantage of our algorithm is that it always has the same execution time, for the same grid size, no matter how many and how far the measured objects are away from the vehicle. Instead, the Bresenham algorithm suffers from varying execution times, due to the different number of grid cells that have to be updated. If the measured object is farther away from the own vehicle, more cells have to be updated as if the object would be closer to the own vehicle.

In Algorithm 1, our new *cell-based update algorithm* is described. It iterates over all grid cells of the occupancy grid map. The important part of our algorithm is the function *getCellState*. In this function, the ray for a given cell coordinate is retrieved and checked if, based on the laser beam, the cell state is free, taken, or unknown. This is achieved by taking the current velocity vector of the car and computing the corresponding ray for every coordinate with the help of the *getRayForCoordinates* function, described in Algorithm 2, which is called in *getCellState*. The first step in *getRayForCoordinates* is to compute the angle between the given coordinate and the velocity vector. After the angle is calculated, the rotation direction can be determined if the cell is clockwise or counterclockwise turned from the velocity vector. The information is used to compute the correct index in the *LaserDataArray*. This array contains the laser measurements. After the index is computed, the correct laser ray can be returned, i. e., which hits the chosen cell in the environment map. Afterwards, the length of this ray can be compared, in the Cartesian coordinate system, with the distance of the grid cell to the origin of the laser beam, the lidar sensor. If the length of the ray is longer than the Euclidean distance of the grid cell to the sensor, the grid cell is free. If it is shorter, the occupancy of the grid cell is unknown, and if it is equal, there is an obstacle at exactly this grid cell. Finally, in line nine of the algorithm, the previous probability value in the map is merged (updated) with the corresponding new measurement, using Eq. (4).

The accuracy of our cell-based update algorithm and the Bresenham update algorithm, which is used, e.g., in [9], [5], [21], remains the same. In the Bresenham algorithm, a laser beam lb from the sensor measurement z_t is drawn from the sensor to the measured objects. As a result of this, the probability of all cells $p(m_i)$, which are between the sensor origin S and the measured object T with coordinates (x_t, y_t), is updated $p(m_i|z_{1:t}, x_{1:t})$. In our algorithm, we look back to the sensor and check which laser

beam is relevant for the cell and update the probability of cell $p(m_i)$ accordingly. Therefore, our parallelized implementation can be considered as a kind of *inverted* Bresenham algorithm.

For comparison, to update the environment map, one thread is created per cell in CUDA, and each thread independently executes the above cell-based update algorithm. To update these maps using the Bresenham algorithm, we created for every laser beam a thread on the GPU, similar as in the work by Homm et al. [9].

D. Hybrid Map

A disadvantage of the classic occupancy grid map is its uniform resolution over the whole map. This motivates us to propose a new hybrid map *hm* in the following. Surely, the occupancy grid map has also some advantages, e.g., the ego-motion of the own vehicle can be easily compensated using a 2D ring buffer, which corresponds to shifting two pointers in software. One for the movement in the lateral and one for the movement in the longitudinal direction. Rotations of the vehicle are compensated by rotating the own vehicle on the map and not the other vehicles in the surroundings. Normally, in hybrid maps, other objects in the environment map have to be rotated by the rotation of the own vehicle, like in the *interval map* [19]. But this requires a considerable amount of computing power, which results in long update cycles of such types of environment maps, maybe longer than the sensor's sampling rate.

The grid cells of the hybrid map are also squares, like in the occupancy grid map. Grid cells close to the sensor are smaller than cells farther from the sensor (see Fig. 2). The edge length between two cell sizes always doubles, e.g., 0.2m, 0.4m and 0.8m for a hybrid map with three different grid cell sizes. Therefore, the area of one grid cell between two different grid cell sizes gets four times bigger. Our *hybrid map hm* is defined as:

$$hm = \{\cup_{i\in[1,n]} hm_{i,j} | \forall i : \exists j \in [1,m]\} \tag{5}$$

The index i indicates the section of the hybrid map and index j the cell in the corresponding section i. The sections are ordered by the cell size, starting with i_1, which is the section with the smallest cells and ending with i_n, which is the section with the biggest cell size. The smallest size of the hybrid map has a size of 16×16 grid cells, also called *unit hybrid map* due to the doubling of the cells' edge length. In the following example, we assume that $m = 16$. The smallest index for the hybrid map, with three sections, is eight, like shown in Fig. 2. For three different cell sizes, a corresponding hybrid map is depicted in Fig. 2. For programming a GPU, it is important to store the data efficiently, especially to have coalesced memory access. Storing the whole map in a 2D array is not efficient because neither every row nor every column has an equal number of values. For example, as seen in Fig. 2, row eight has only four cells, and row eight has eight cells. Thus, for every different cell size, a separate array is created, which has the same size. So every cell size in the hybrid map has the same number of grid cells. Finally, on the GPU, all 2D arrays are combined into one 3D array. If there are three different sizes of cells, we have one array with the size of $16 \times 16 \times 3$. If the chosen size of a hybrid map is, for example, 1024×1024, and therefore different from the *unit hybrid map*, it is necessary to calculate the cell distribution and index for the different sections of a hybrid map. Hence, it is necessary to scale the unit hybrid map to bigger grid sizes. It is desired that the hybrid map always covers the same area as a comparable occupancy grid map. Similarly, the number of grid cells in each section of the hybrid map should be the same. Therefore, the number of grid cells in each section has to be known. l_i indicates the last index in section i, and it is calculated from the origin in the map, i. e., the vehicle's position. To calculate l_i, a scale factor s is introduced. For the unit hybrid map, we have for the section with the smallest cells $l_1 = 2$, for the mid-sized cells $l_2 = 4$, and finally for the section with the biggest cells $l_3 = 8$. For a hybrid map that has equal size as an occupancy grid map, for example, 1024×1024, we have to resize the unit map. The *scale s* is computed out of the maximal size of the map *maxSize* and the number of sections #*S*:

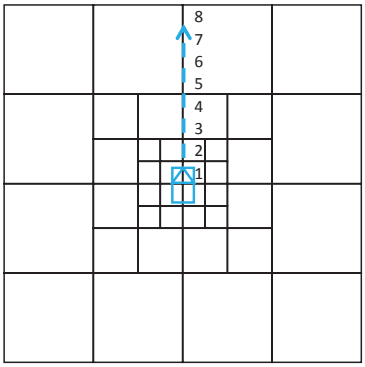

Figure 2: Hybrid map with a granularity of the different grid cell sizes and the used indices to calculate the size of each section.

$$s = \frac{(maxSize/2)}{2^{\#S}} \tag{6}$$

To calculate l_i, its corresponding section has to be considered:

$$l_i = s \cdot 2^i \tag{7}$$

For example, for a hybrid map equal to 1024×1024, we get $l_1 = 128$ for the section with the smallest cells, $l_2 = 256$ for the section with the mid-sized cells, and $l_3 = 512$ for the section with the biggest cells. For example, as shown in Fig. 3, the single maps cover the same area up to three times, although only one time would be necessary. The areas covered more than one time are relatively small compared to the conventional occupancy grid map. Always a quarter of the cells of the bigger size is computed needlessly. If we take a hybrid map with a size of 1024×1024, it has actually $256 \times 256 \times 3 = 196,608$ grid cells. $2 \times 256 \times 256/4 = 32,768$ grid cells are calculated unnecessary, which is 16.7% of the cells. Compared to the original number of cells $1024 \times 1024 = 1,048,576$ of the occupancy grid map, the hybrid map reduces the number of grid cells by 82%. Then, only 3.75% of the grid cells are calculated multiple times compared to the original occupancy grid map.

Since there are fewer cells as in a classic occupancy grid map, the accuracy in some regions of the hybrid map also decreases. In the important area, close to the vehicle environment, the resolution of the environment map remains the same, as in the conventional occupancy grid map. Cells further away have less resolution, which leads to a certain inaccuracy. Let *hm* be a hybrid map with two different cell sizes $h_{1,j}$ and $h_{2,j}$. If there is no object or an object which has the size of a cell in $h_{2,j}$ the accuracy of the occupancy grid map and a classic hybrid map remains the same. The worst case would be that a current measured object would

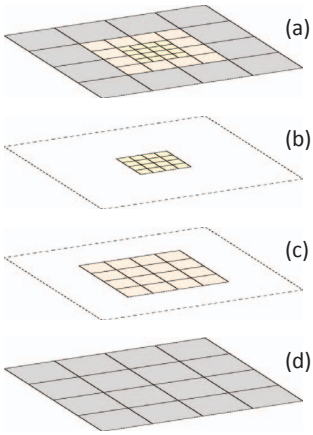

Figure 3: The upper hybrid map (a) shows the final stage of the environment map. The maps with the finest granularity (b), with the medium granularity (c) and coarsest granularity (d) are calculated and combined to a hybrid map on the GPU. As shown in the figure, the individual maps cover some areas of the final hybrid map up to three times, for computational reasons, as described in the text.

have the size of a cell in $h_{1,j}$, but is located in $h_{2,j}$. Then, the object would have the size of the cell in $h_{2,j}$, which leads to a maximum error of 75%. However, for the rough planning of a vehicle's route, this error is not critical because routes are only planned in free areas. If the route of the vehicle is planned in detail, the object is in the area with the high resolution.

III. Experiments

A. Evaluation Environment

To evaluate our algorithms we used a lidar scanner, which has a 360° horizontal field of view. It has a 0.08° angular resolution, which results in 4,500 laser beams. For the experiments, two GPU platforms where used. A desktop computer, equipped with an Intel Core i5-4670K processor at 3.4 GHz and an Nvidia GTX970 GPU with 1,664 CUDA Cores at 1,114 MHz. The experimental embedded platform was a Jetson K1 board. It embodies a quad-core ARM Cortex-A15 CPU with 2.3 GHz and Nvidia Tegra K1 GPU with 192 CUDA Cores and 850 MHz. Our experiments have been performed for several sizes for environment maps to evaluate the scalability of our algorithms. The shown measured times were the average times over several cycles.

B. Evaluation

In the first experiment, we compared our new update algorithm, the cell-based update algorithm, with the state-of-the-art Bresenham algorithm on an occupancy grid map, as well as using our hybrid map approach. The experiments were executed on the GPU of an Nvidia Jetson board, and the results are illustrated in Table I. All algorithms were real-time capable, except one. For a grid size of 2048×2048, the cell-based update algorithm slightly missed the update rate of 25 Hz and reached only an update rate of 22.5 Hz. But such a high number of grid cells is usually not used in practice. In Fig. 4, the experiments given in Table I are normalized to the Bresenham algorithm on the occupancy grid map for each grid size for the sake of better comparability. This means, only values for the same grid size are comparable. For small and medium occupancy grid sizes, like the widely used 512×512 occupancy grid size in the automotive sector [9], the cell-based algorithm is

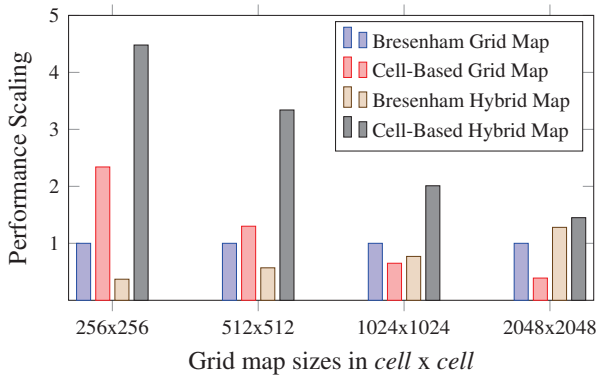

Figure 4: Performance scaling of the algorithms in Table I. The values were normalized to the Bresenham algorithm on the grid map for each size of the grip map, executed on the GPU of the Jetson board.

faster than the state-of-the-art Bresenham algorithm. For big grid sizes, it is slower because for every grid cell, a thread is created, which calculates the occupancy of the grid cell. Instead, in case of the Bresenham algorithm, for every laser beam, one thread is created, which updates the probability in the map. For the grids in our examples, the number of grid cells grows exponentially, but the sensor resolution stays the same. Thus, the computational effort of our method also nearly becomes exponential, while in the Bresenham algorithm it only grows linear.

The computation of the hybrid map with the cell-based update algorithm is faster than the computation of the standard occupancy grid. For a grid size of 1024×1024, the hybrid map has only about 20% of the grid cells as the occupancy grid map with of the same size; thus, considerably fewer calculations have to be performed. The Bresenham algorithm on the hybrid map was always slower, even on maps with a significant number of cells than the cell-based update algorithm. If the Bresenham algorithm updates the cell with a measurement, it always has to be checked, which size the current cell has, due to the different grid cell sizes in the hybrid map. This adds an extra overhead to the algorithm, which slows it down in comparison to the others.

We also performed the same experiments on a desktop computer equipped with the discrete Nvidia GTX970 GPU. The results are shown in Table II. The settings were the same as in the first experiments (Table I). The execution times on the desktop GPU were only marginally higher as on the embedded GPU. This is mainly caused by the fact that data has to be explicitly copied to the discrete GPU and back, while this is not the case for the embedded GPU. Obviously, these data transfers add additional time to the total execution time of the algorithm. Also, the desktop GPU cannot make full use of its much higher computing power due to a larger amount of CUDA cores than the embedded GPU,

Table I: Execution times in ms of the update of the occupancy grid (Gr. M.) and hybrid map (H. M.) using the Bresenham (Br.) and the cell-based update algorithm (Cell B.) on the GPU of the Jetson board.

	grid size			
algorithm	256	512	1024	2048
Br. Gr. M.	2.06	3.81	7.24	17.47
Cell B. Gr. M.	0.87	2.92	11.20	44.49
Br. H. M.	5.50	6.72	9.40	13.66
Cell B. H. M.	0.45	1.13	3.61	12.01

Table II: Execution times in ms of the occupancy grid (Gr. M) and hybrid map (H. M.) with the Bresenham (Br.) and the cell-based update algorithm (Cell B.) on the desktop PC. The experiment was executed on an Nvidia GTX970 GPU.

	grid size			
algorithm	256	512	1024	2048
Br. Gr. M.	2.26	4.78	8.99	17.98
Cell B. Gr. M.	0.73	2.78	10.86	43.05
Br. H. M.	10.11	20.32	41.54	80.92
Cell B. H. M.	0.62	1.36	4.12	13.79

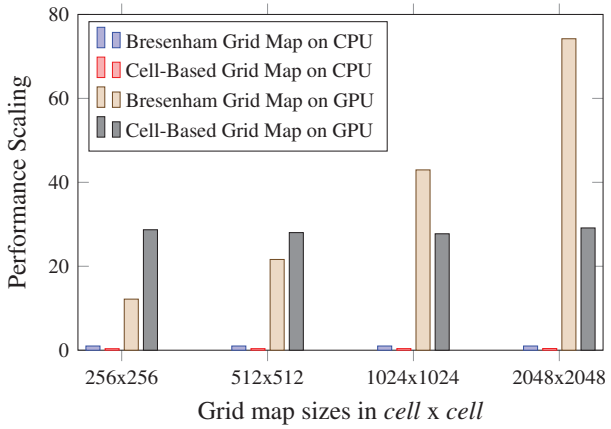

Figure 5: Speedup measured on the Nvidia Jetson board between the CPU and GPU versions of the algorithm. The values are normalized for each size of the grid map to the Bresenham algorithm.

but the algorithm has no intense arithmetic calculations. Thus, the desktop GPU cannot benefit from it.

In our last experiment, we executed the cell-based update algorithm on the CPU of the Nvidia Jetson board and compared it to the parallel version of that algorithm. The results are normalized to the CPU version of the Bresenham algorithm on the occupancy grid map. They are normalized for each grid size for a better comparison and shown in Fig. 5. The CPU version is a sequential algorithm. The speedup of the GPU version, compared to the CPU version, increases with bigger grid sizes for the Bresenham algorithm. The individual threads have a higher workload for larger grid sizes because one thread has to update more cells in average. The cell-based update algorithm has nearly a constant speedup. Finally, the Bresenham algorithm is faster on the CPU as our cell-based update algorithm on the CPU. This is not surprising, as the Bresenham algorithm has lesser arithmetic operations than our algorithm, which is advantageous on the CPU.

IV. Conclusion

In this paper, we introduced a new parallel update algorithm for the occupancy grid map, which serves as the basis for automated driving. We showed that this algorithm could outperform state-of-the-art GPU implementations of the Bresenham algorithm used to update the occupancy grid map by 2.5 for small and mid-sized occupancy grids. Further, our proposed algorithm has always a constant execution time for updating the environment map, whereas the Bresenham algorithm has a varying latency depending on the laser measurement. Only on single core CPU is the standard Bresenham algorithm to update the environment maps faster than our single-core cell-based algorithm due to fewer arithmetic operations. As a second contribution, the introduced hybrid map cuts down execution times up to a factor of 4.5 while having an accuracy as high as in the case of an occupancy grid map, in close proximity of to the own vehicle. In the future, we want to investigate power tradeoffs between CPU and GPU implementations.

References

[1] H. Badino, U. Franke, R. Mester, and F. A. Main, "Free space computation using stochastic occupancy grids and dynamic programming", in *Proc. of the Intl. Workshop on Dynamical Vision at Eleventh IEEE Intl. Conference on Computer Vision (ICCV)*, (Rio de Janeiro, Brazil), Oct. 20, 2007.

[2] J. E. Bresenham, "Algorithm for computer control of a digital plotter", *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[3] E. Einhorn, C. Schröter, and H. M. Gross, "Finding the adequate resolution for grid mapping – Cell sizes locally adapting on-the-fly", in *Proc. of the IEEE Intl. Conference on Robotics and Automation (ICRA)*, (Shanghai, China), May 9–13, 2011, pp. 1843–1848.

[4] A. Elfes, "Using occupancy grids for mobile robot perception and navigation", *Computer*, vol. 22, no. 6, pp. 46–57, Jun. 1989.

[5] J. Fickenscher, O. Reiche, J. Schlumberger, F. Hannig, and J. Teich, "Modeling, programming and performance analysis of automotive environment map representations on embedded GPUs", in *Proc. of the IEEE Intl. High-Level Design Validation and Test Workshop (HLDVT)*, (Santa Cruz, CA, USA), IEEE, Oct. 7–8, 2016, pp. 70–77.

[6] J. Fickenscher, S. Reinhart, M. Bouzouraa, F. Hannig, and J. Teich, "Convoy tracking for ADAS on embedded GPUs", in *Proc. of the IEEE Intelligent Vehicles Symposium (IV)*, (Redondo Beach, CA, USA), IEEE, Jun. 11–14, 2017, pp. 959–965.

[7] C. Fulgenzi, A. Spalanzani, and C. Laugier, "Dynamic obstacle avoidance in uncertain environment combining PVOs and occupancy grid", in *Proc. IEEE Intl Conference on Robotics and Automation (ICRA)*, (Roma, Italy), Apr. 10–14, 2007, pp. 1610–1616.

[8] G. Grisettiyz, C. Stachniss, and W. Burgard, "Improving grid-based SLAM with Rao-Blackwellized particle filters by adaptive proposals and selective resampling", in *Proc. of the IEEE Intl. Conference on Robotics and Automation (ICRA)*, (Barcelona, Spain), Apr. 18–22, 2005, pp. 2432–2437.

[9] F. Homm, N. Kaempchen, J. Ota, and D. Burschka, "Efficient occupancy grid computation on the GPU with lidar and radar for road boundary detection", in *Proc. of the IEEE Intelligent Vehicles Symposium (IV)*, (La Jolla, CA, USA), Jun. 2010, pp. 1006–1013.

[10] X.-J. Jing, Ed., *Motion Planning*. InTech, Jun. 2008. [Online]. Available: https://www.intechopen.com/books/motion_planning.

[11] S. Kammel and B. Pitzer, "Lidar-based lane marker detection and mapping", in *Proc. of the IEEE Intelligent Vehicles Symposium (IV)*, (Eindhoven, The Netherlands), Jun. 4–6, 2008, pp. 1137–1142.

[12] G. K. Kraetzschmar, G. P. Gassull, and K. Uhl, "Probabilistic quadtrees for variable-resolution mapping of large environments", in *Proc. of the IFAC/EURON Symposium on Intelligent Autonomous Vehicle (IAV)*, (Lisboa, Portugal), Jul. 5–7, 2004.

[13] H. Moravec, "Robot spatial perception by stereoscopic vision and 3d evidence grids", Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. CMU-RI-TR-96-34, Sep. 1996.

[14] NVIDIA Corp., *Programming guide – CUDA toolkit documentation*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2016.

[15] T. Rakotovao, J. Mottin, D. Puschini, and C. Laugier, "Integration of multi-sensor occupancy grids into automotive ECUs", in *Proc. of the ACM/EDAC/IEEE Design Automation Conference (DAC)*, (Austin, TX, USA), Jun. 5–9, 2016, 27:1–27:6.

[16] D. F. Sebastian Thrun Wolfram Burgard, *Probabilistic Robotics*. Cambridge, Massachusetts and London, England: The MIT Press, 2005.

[17] A. Souza, R. S. Maia, R. V. Aroca, and L. M. G. Gonçalves, "Probabilistic robotic grid mapping based on occupancy and elevation information", in *Proc. of the Intl. Conference on Advanced Robotics (ICAR)*, (Montevideo, Uruguay), Nov. 25–29, 2013, pp. 1–6.

[18] A. Souza and L. Gonçalves, "Occupancy-elevation grid: An alternative approach for robotic mapping and navigation", vol. 1, p. 18, Apr. 2015.

[19] T. Weiherer, S. Bouzouraa, and U. Hofmann, "An interval based representation of occupancy information for driver assistance systems", in *Proc. of the Intl. IEEE Conference on Intelligent Transportation Systems (ITSC)*, Oct. 2013, pp. 21–27.

[20] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: A probabilistic, flexible, and compact 3d map representation for robotic systems", in *Proc. of the ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*, (Anchorage, AK, USA), 2010.

[21] M. Yguel, O. Aycard, and C. Laugier, "Efficient GPU-based construction of occupancy grids using several laser range-finders", in *International Journal of Vehicle Autonomous Systems*, vol. 6, Oct. 2006, pp. 105–110.