GPU-Based Optimisation of 3D Sensor Placement Considering Redundancy, Range and Field of View

Joacim Dybedal* and Geir Hovland*

*University of Agder
Faculty of Engineering and Science
Jon Lilletunsvei 9, 4879 Grimstad, Norway

Abstract **This paper presents a novel and efficient solution for the 3D sensor placement problem based on GPU programming and massive parallelisation. Compared to prior art using gradient-search and mixed-integer based approaches, the method presented in this paper returns optimal or good results in a fraction of the time compared to previous approaches. The presented method allows for redundancy, i.e. requiring selected sub-volumes to be covered by at least n sensors. The presented results are for 3D sensors which have a visible volume represented by cones, but the method can easily be extended to work with sensors having other range and field of view shapes, such as 2D cameras and lidars.**

B.1 Introduction

The use of 2D cameras and 3D point cloud sensors and algorithms is a key enabler in autonomous systems. Historically, the automotive and self-driving vehicles industry have been leading domains, see for example [B1], [B2] and [B3]. Typical sensors used are radar, lidar and cameras. The use of computer vision sensors are now rapidly being applied in other industries as well. The paper [B4] presents an example where the intended application domain was in offshore drilling.

The optimal placement of 2D and 3D sensors is a challenging problem. Most of the publications found in the open literature are limited to solving the 2D problem, for example solving a surveillance problem of a 2D floor plan. Mixed-integer programming is one of the approaches successfully used in solving the 2D problem. Some previous work has attempted solving the 3D problem by using heuristic approaches such as Genetic Algorithms. However, such approaches usually end up in a local minimum. The authors previous work on this topic, [B4], contained several references of different methods used for both 2D and 3D sensor placement optimisation, see [B5], [B6], [B7], [B8] and [B9].

Examples of more recent work are [B10] and [B11] where the problem of optimal lidar configuration for self-driving cars was considered. Optimal spinning lidar placement is different from optimal camera placement, since the lidar's visible volume can not be described using the two parameters range and field of view, as is the case with cameras. Instead, both the above mentioned papers model the sensors using cone-like perception areas, but still discretise the considered volume into cuboids. In [B10] the mixed-integer programming approach is used, as in [B4] whereas in [B11] an artificial bee colony evolutionary algorithm is applied.

The main drawback of solutions based on mixed-integer linear programming is the inability to scale the problems. Nonlinear equations need to be linearised by introducing many new variables. In addition, the problem size grows very fast when a high accuracy is wanted (small cuboid size). The time required to find good solutions is many hours on powerful computers, even for relatively small problems. Gradient search algorithms have the disadvantage that they often end up in local minima.

In this paper the optimal placement of 3D sensors to cover a volume of interest is considered by exploiting the parallel processing and 3D architecture of a CUDA-based GPU. Such GPUs were originally intended for gaming and ray tracing applications, and hence are ideally suited for massive, parallel computations in 3D environments. The intended application considered in this paper is optimal 3D sensor placement in a volume illustrated by Fig. B.1 which shows two industrial robots mounted on linear track motion. The purpose of the 3D sensors is to monitor safety in a human-robot collaborative environment, as well as perform early collision detection and avoidance.

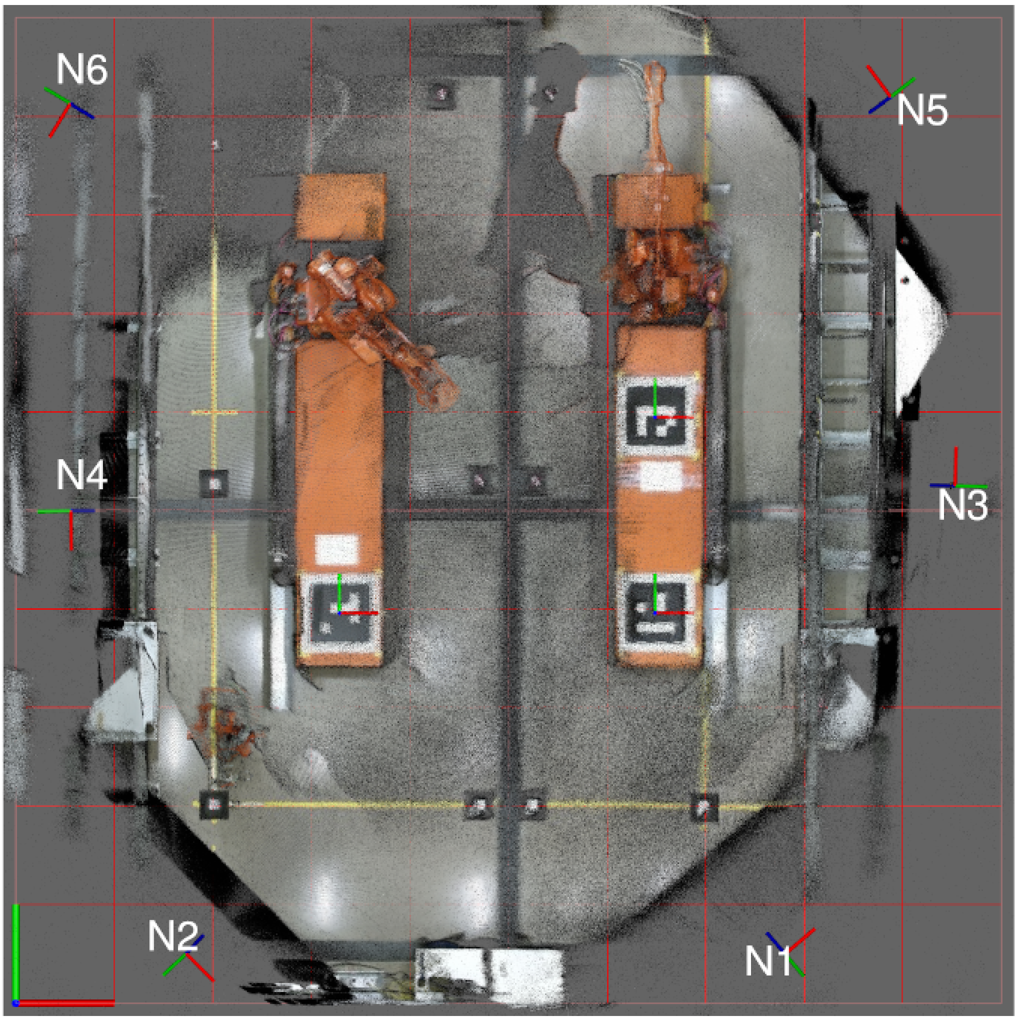

Figure B.1: A multi-sensor industrial robotics cell for human-robot collaboration. The positions of the six 3D sensors, N1-N6, are to be optimised.

B.2 Problem Formulation

The volume covered by a 3D sensor is modelled as a cone having a limited field of view S_f and range S_r as illustrated in Fig. B.2. The variables A and Q in the figure describe the position (x,y,z) and orientation (quaternion Q) of the sensor. A point P lies inside the cone covered by the sensor if the angle q is smaller than the field of view $S_f/2$ and if the distance between the two points A and P is less than S_r. The entire volume is divided into smaller cuboids.

The objective of the optimisation is to see as many cuboids as possible with the available sensors. In some cases, redundancy will be specified as a constraint, i.e. some cuboids must be seen by n sensors or more. Volumes requiring redundancy could for example be those covered by the linear track motions and the robots shown in Fig. B.1.

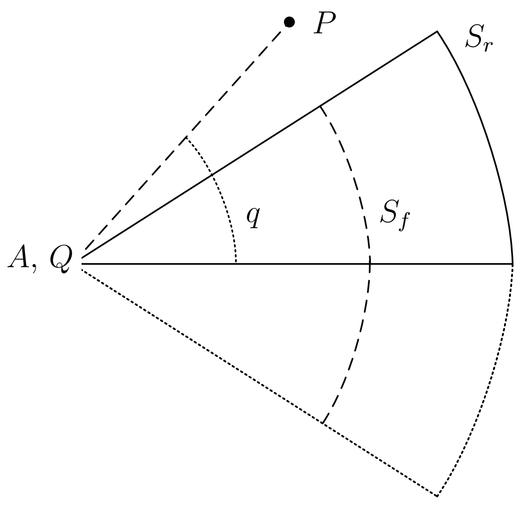

Figure B.2: Illustration of sensor range (S_r) and field of view (S_f). A and Q describe the sensor's location and orientation (quaternion) , respectively. P is a point outside the sensor's field of view and q is the angle between the sensor's centre line and the line PA.

B.3 Optimisation Method

The optimisation method used in this paper is similar to a brute-force search with the exception that random floating-point numbers are generated for the sensor locations, and not a discretised search space. A discretised search space may miss good solutions if the discretisation is coarse and may also take a long time before a good solution is found. With a randomised floating-point numbers search good solutions are found relatively fast. The demonstrated results presented in this paper allow one free variable per sensor, typically X- or Y-location of a sensor along a wall, or two free variables, for example the planes XZ or YZ.

As mentioned in the introduction, GPUs were originally intended for 3D gaming applications and are ideally suited for this type of problem. By exploiting the inherent capability in CUDA to structure threads in three dimensions, each cuboid in the volume of interest is assigned a thread running a CUDA kernel. The kernel calculates the angle q and distance from A to P, and checks if the centre coordinate of the cuboid is covered by the sensor's field of view and range as described in Section B.2.

Fig. B.3 shows the layout used in this paper when launching CUDA threads. The threads (each representing a cuboid) are organised using three-dimensional blocks of threads and a three-dimensional grid of blocks. In the case studies below, each block has a dimension of $8 \times 8 \times 8$ threads, i.e. 512 threads per block. The grid dimension (number of blocks) is scaled such that there is at least enough threads to cover the entire volume of interest. With such a layout, some blocks will only partially cover the volume of interest, as seen in the figure, and threads that fall outside the volume will do no work. However, structuring the blocks in this way,

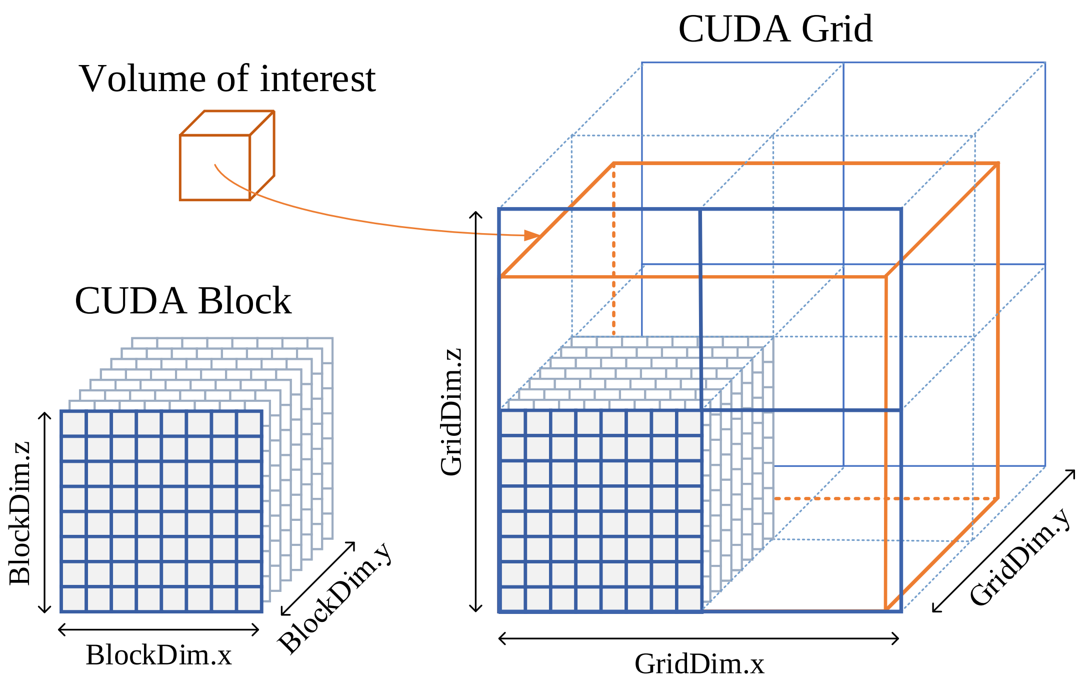

Figure B.3: Layout of CUDA threads.

with the number of threads in a block being a multiple of 32 (the number of warps in a multiprocessor) and not using the maximum capacity of a thread block, helps the GPU to efficiently schedule the workload across and within the multiprocessors. For more details on CUDA and GPU processing, see for example [B12]. For CUDA devices with compute capability 6.1 such as the GTX 1080 used in the case studies in this paper, the grid y and z dimensions are limited to 65 535. When using a block dimension of $8 \times 8 \times 8$, this means that volumes with up to 524 288 cuboids in each dimension can be processed using a single CUDA grid.

As seen in Fig. B.4, each sensor to be optimised gets its own set of threads, stacked on top of each other in the CUDA grid. The figure shows only two dimensions for simplicity. By structuring the threads this way, the coverage equations are calculated in parallel for all sensors. If a thread calculates that it is covered by the respective sensor, the thread writes to the corresponding cell in a global cover matrix, which contains a 32 bit integer for each cuboid in the volume. To be able to differentiate which sensor covers which cuboid, the thread adds 2^N to the number, where N is the sensor number, counting from 0. This means that bit N in the 32 bit integer will be set if sensor N covers the cuboid. In fact, the operation is implemented as an atomic bit-wise OR operation on the global cover matrix. By using a 32 bit integer, the number of sensors is currently also limited to 32.

In addition to writing to the global cover matrix, each thread also increments a counter such that the total number of covered cuboids can be extracted from

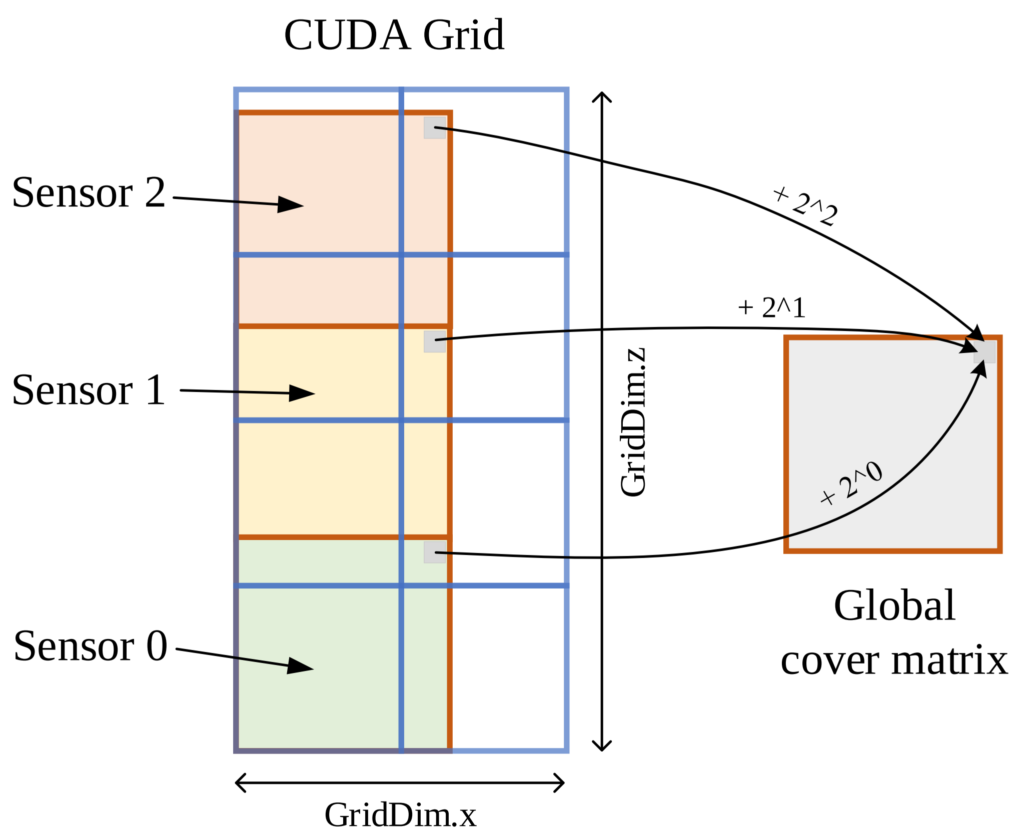

Figure B.4: CUDA threads for multiple sensors. Simplified into two dimensions.

GPU memory after all threads have finished. Specifically, each CUDA block uses a shared memory array for this task, which contains one integer for each thread in the block. To avoid counting cuboids more than once (if more than one sensor cover the same cuboid), each thread only increments its counter if no other thread has already written to the same cell in the global cover matrix. When all threads are finished writing to the shared memory, one thread in each CUDA block iterates over the shared memory and writes the total number of covered cuboids to global memory. This is done to limit the amount of global memory writes, which is slower than using shared memory which physically resides closer to the processor.

After the CUDA launch is completed, the total number of covered cuboids is found by adding the numbers from each CUDA block. If this result is better than any result found in previous iterations, the complete cover matrix is copied from GPU memory and stored as the best result, along with the randomly generated sensor location variables.

B.3.1 Redundancy Constraints

When optimising sensor placement, not only the maximum possible coverage in a volume is of interest. In practical applications there will be sub-volumes that are occluded or more critical and thus need to be monitored by a specific amount of sensors. To accommodate this, the developed method can also incorporate redundancy constraints. This is done by adding a new matrix (dimensionally identical to the cover matrix) as input to the CUDA program, where each cell representing a cuboid holds an integer. The matrix must be pre-filled, and any integer above 0 yields a redundancy constraint. E.g. the number 3 will enforce a constraint such that the cuboid must be covered by at least three sensors for the final solution to be considered valid. These are hard constraints, meaning a solution may not be obtainable depending on the sensor parameters and free variables.

In the CUDA kernel, this functionality is implemented by checking the cuboid's cell in the redundancy matrix, and in addition to writing to the cover matrix as described above, the value in the redundancy matrix is decreased (atomically) by 1. If the value is now 0, it means that the redundancy constraint for the cuboid has been fulfilled, and the number of fulfilled constraints is counted in the same manner as counting covered cubes. When the CUDA launch is finished, the total number of fulfilled redundancy constraints are checked against the total number of specified constraints. If the values match, the solution is considered valid.

B.4 Case Studies

The results from five different case studies are presented in this paper:

I. Two sensors, located at the ceiling, one on each side of a room of size $10\,\text{m} \times 10\,\text{m} \times 10\,\text{m}$, no redundancy requirement. This is the same setup as presented in [B4] except that the accuracy is increased (the size of the cuboids is halved to $0.5\,\text{m} \times 0.5\,\text{m} \times 0.5\,\text{m}$).

II. Same volume as above, except four sensors, two rotated downwards and two rotated upwards. The z coordinates for all sensors are also free variables.

III. Same volume as above, four sensors, two located at the ceiling and two located at the floor, but this time with a redundancy requirement at the centre of the volume.

IV. Setup similar to the real volume shown in Fig. B.1: Six sensors, all located at the ceiling, with a redundancy requirement of 1 for all cuboids near the floor. The room dimensions are $10\,\text{m} \times 10\,\text{m} \times 4.5\,\text{m}$

V. Same as above, six sensors representing the real volume, but with redundancy requirements for cuboids representing the robot tracks, forcing the sensors to focus on the robots. This case also uses a smaller cuboid size of $0.25\,\mathrm{m} \times 0.25\,\mathrm{m} \times 0.25\,\mathrm{m}$ for increased resolution.

Where nothing else is specified, all free variables have lower and upper bounds equal to the corresponding volume dimensions. In case study IV and V, the volume is in reality covered by Microsoft Kinect v.2 3D cameras, which have a field of view (for depth measurements) of 70×60 degrees. As an approximation, due to the cone shaped field of view used in this paper, the vertical field of view (60 deg) is used in the case studies. And even though the Kinect has a specified range of up to 4.2 meters, the sensors are used for measuring longer distances in this volume, as described in [B13].

One single Nvidia GTX 1080 GPU was used in the presented case studies, and every iteration was one CUDA-launch with one set of random variables. For each new iteration new random variables were generated.

B.4.1 Case Study I

Setup:

- 2 sensors, Cuboid size: 0.5 m, No. of cuboids: 8000
- Field of View: $\pi/5$, Range: 8.0 m
- Fixed variables: Rotation (45° down), y-coordinates, z-coordinates
- Free variables: x-coordinates
- Redundancy Constraints: None

Fig. B.5 shows the results of case study I. In the final solution both sensors cover 436 cuboids each and 872 cuboids were covered in total. The optimal sensor locations were found to be:

$$A_0 = [2.499, 10.0, 10.0]^T \tag{B.1}$$

$$A_1 = [7.499, 0.0, 10.0]^T \tag{B.2}$$

In total 1 million sensor location combinations were evaluated and the total computation time was 18.3 seconds. This result is a significant improvement over the approach based on mixed-integer linear programming presented in [B4] which took several hours to compute.

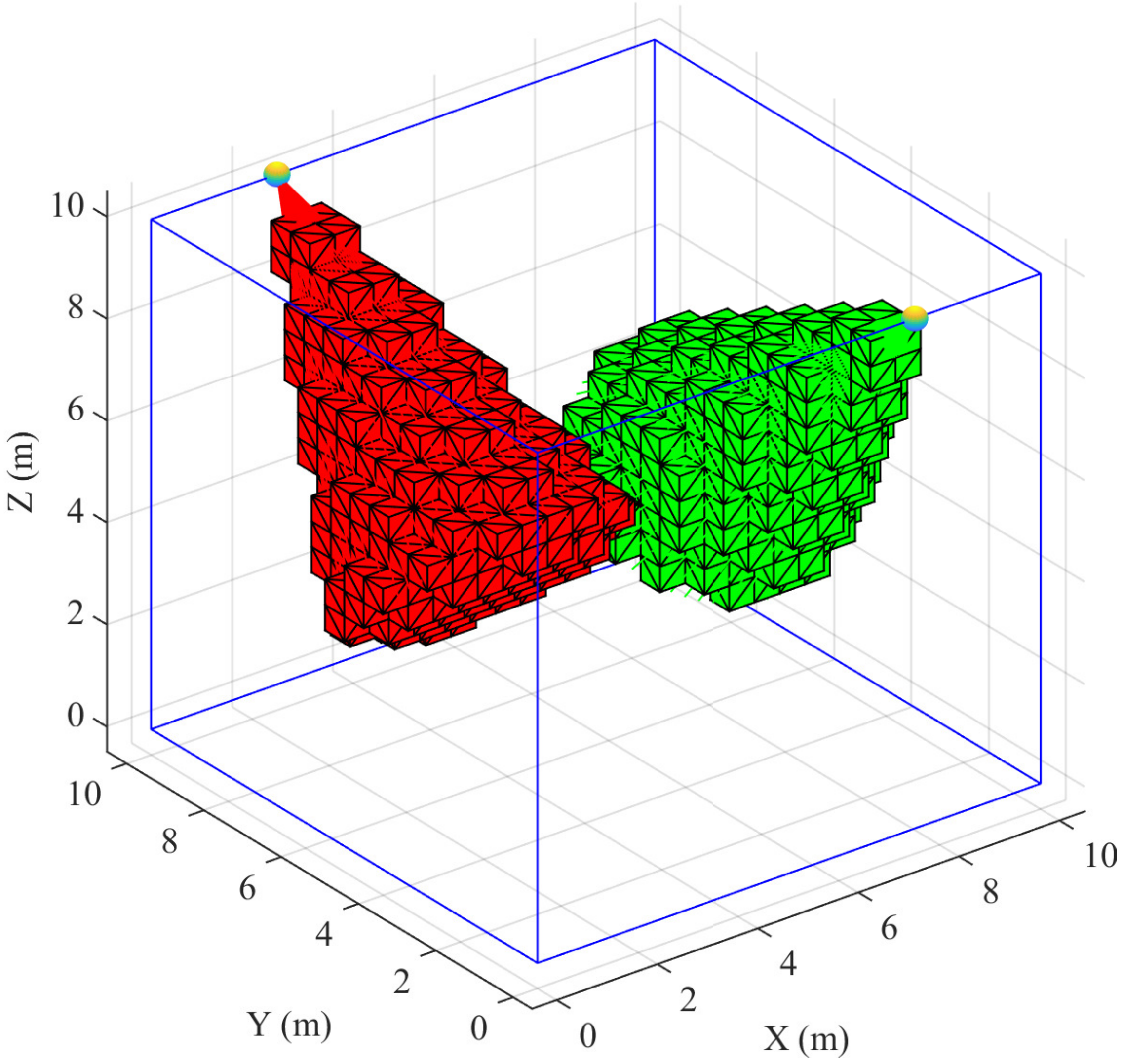

Figure B.5: Case Study I: Result of 3D sensor placement with two sensors, one on each side of the room. Discretised cuboid size equals 0.5 m × 0.5 m × 0.5 m.

B.4.2 Case Study II

Setup:

- 4 sensors, Cuboid size: 0.5 m, No. of cuboids: 8000
- Field of View: $\pi/5$, Range: 8.0 m
- Fixed variables: Sensor 0 and 1: Rotation (45° down), y-coordinates; Sensor 2 and 3: Rotation (45° up), x-coordinates
- Free variables: x-coordinates (sensor 0 and 1), y-coordinates (sensor 2 and 3), z-coordinates
- Redundancy Constraints: None

Fig. B.6 shows the results of case study II. In the final solution the sensors cover 423, 427, 420 and 427 cuboids, respectively. A total of 1691 cuboids were covered. The optimal sensor locations were found to be:

$$A_0 = [7.034, 10.0, 7.272]^T \tag{B.3}$$
$$A_1 = [2.495, 0.0, 7.101]^T \tag{B.4}$$
$$A_2 = [10.0, 2.734, 3.564]^T \tag{B.5}$$
$$A_3 = [0.0, 6.971, 2.747]^T \tag{B.6}$$

In total 5 million sensor location combinations were evaluated and the total computation time was 1 min and 33.8 s.

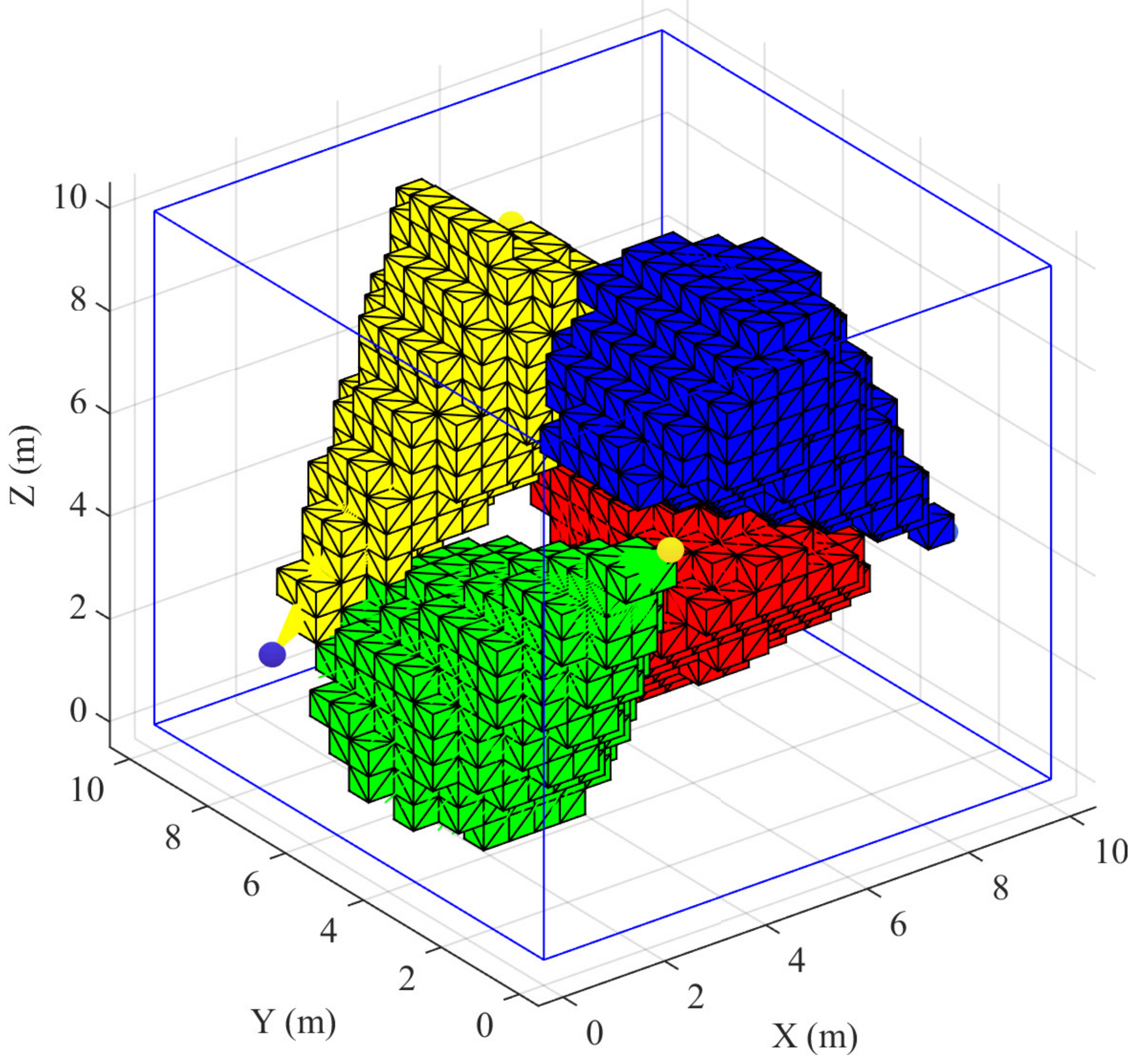

Figure B.6: Case Study II: Result of 3D sensor placement with four sensors, one on each wall with different heights, two pointing upwards and two downwards. Discretised cuboid size equals 0.5 m × 0.5 m × 0.5 m.

B.4.3 Case Study III

Setup:

- 4 sensors, Cuboid size: 0.5 m, No. of cuboids: 8000
- Field of View: $\pi/5$, Range: 8.0 m
- Fixed variables: Sensor 0 and 1: Rotation (45° down), y-coordinates, z-coordinates; Sensor 2 and 3: Rotation (45° up), x-coordinates, z-coordinates
- Free variables: x-coordinates (sensor 0 and 1), y-coordinates (sensor 2 and 3)
- Redundancy Constraints: A volume in the centre of size 2 m × 2 m × 2 m (64 cuboids) must be covered by at least three sensors.

Fig. B.7 shows the results of case study III. In the final solution the sensors cover 436, 436, 426 and 434 cuboids, respectively. A total of 1235 cuboids were covered. The optimal sensor locations were found to be:

$$A_0 = [3.998, 10.0, 10.0]^T \tag{B.7}$$

$$A_1 = [6.0, 0.0, 10.0]^T \tag{B.8}$$

$$A_2 = [10.0, 6.196, 0.0]^T \tag{B.9}$$

$$A_3 = [0.0, 3.994, 0.0]^T \tag{B.10}$$

In total 5 million different sensor location combinations were evaluated and the total computation time was 3 minutes and 7.9 seconds.

B.4.4 Case Study IV

Setup:

- 6 sensors, Cuboid size: 0.5 m, No. of cuboids: 3600
- Field of View: $\pi/3$ (60°), Range: 8.0 m
- Fixed variables: Rotation (all sensors rotated 60° down, orientation according to Fig. B.1), x-coordinates, z-coordinates
- Free variables: y-coordinates (sensor 0 and 1 have an upper limit of 5.0 m, sensor 4 and 5 have a lower limit of 5.0 m)
- Redundancy Constraints: All cuboids below $z = 1.0$ m must be covered by at least 1 sensor.

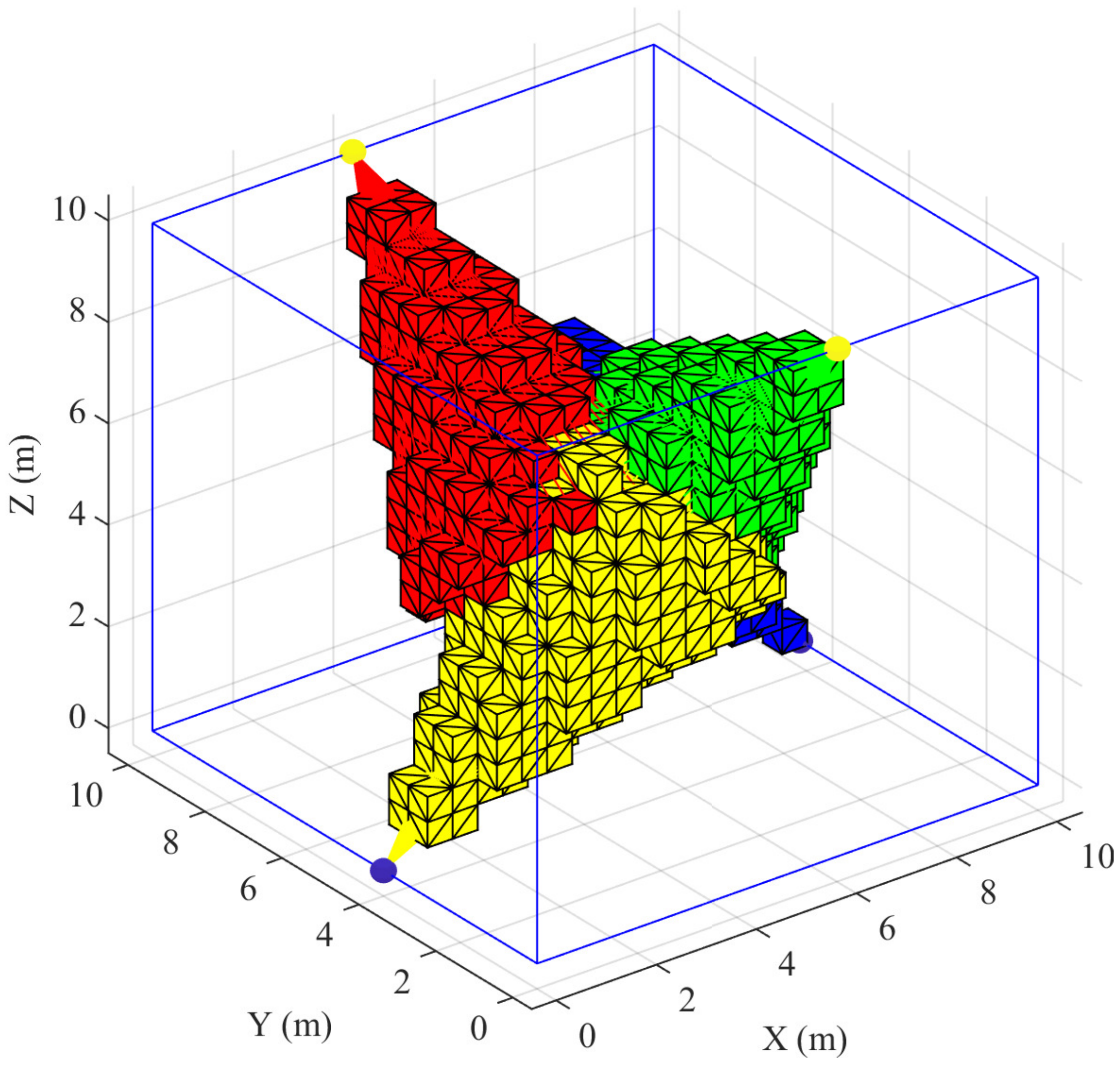

Figure B.7: Case Study III: Result of 3D sensor placement with four sensors, two located at the floor (yellow, blue) and two at the ceiling (green, red). Discretised cuboid size equals 0.5 m × 0.5 m × 0.5 m. 64 cuboids in the centre (a volume of 2 m × 2 m × 2 m) has a redundancy requirement of 3.

Fig. B.8 shows the results of case study IV. In the final solution the sensors cover 427, 416, 383, 369, 416 and 434 cuboids, respectively. A total of 1924 cuboids were covered. The optimal sensor locations were found to be:

$$A_0 = [10.0, 2.281, 4.5]^T \tag{B.11}$$

$$A_1 = [0.0, 0.132, 4.5]^T \tag{B.12}$$

$$A_2 = [10.0, 1.198, 4.5]^T \tag{B.13}$$

$$A_3 = [0.0, 8.91, 4.5]^T \tag{B.14}$$

$$A_4 = [10.0, 9.732, 4.5]^T \tag{B.15}$$

$$A_5 = [0.0, 7.564, 4.5]^T \tag{B.16}$$

In total 5 million different sensor location combinations were evaluated and the total computation time was 3 minutes and 0 seconds.

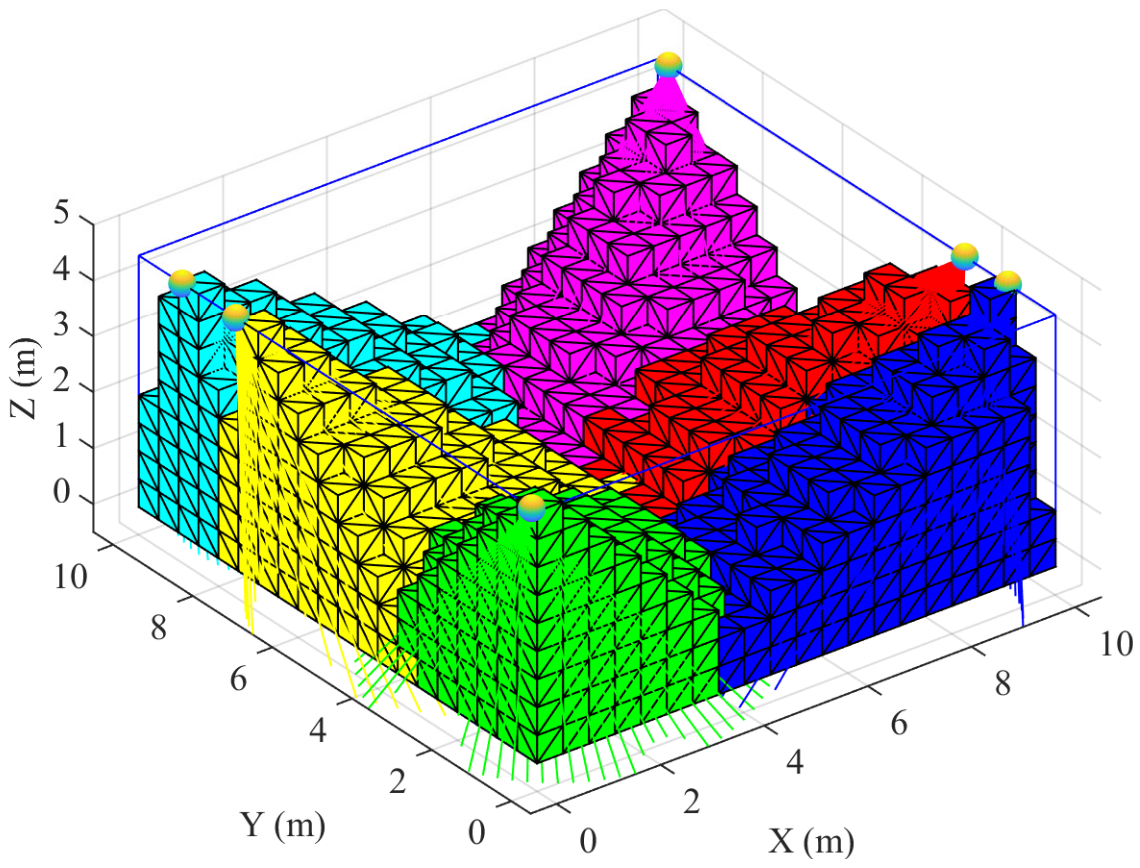

Figure B.8: Case Study IV: Result of 3D sensor placement with six sensors. Discretized cuboid size equals 0.5 m × 0.5 m × 0.5 m.

B.4.5 Case Study V

Setup:

- 6 sensors, Cuboid size: 0.25 m, No. of cuboids: 28 800
- Field of View: $\pi/3$ (60 deg), Range 8.0 m
- Fixed variables: Rotation (all sensors rotated 60° down, orientation according to Fig. B.1), x-coordinates, z-coordinates
- Free variables: y-coordinates (sensor 0 and 1 have an upper limit of 5.0 m, sensor 4 and 5 have a lower limit of 5.0 m)
- Redundancy Constraints: Cuboids representing the robot tracks (see Fig. B.1) must be covered by at least 3 sensors below $z = 1.5$ m.

Fig. B.9 shows the results of case study IV. In the final solution the sensors cover 3446, 3448, 3697, 3687, 3434 and 3470 cuboids, respectively. A total of 10 387 cuboids were covered. The optimal sensor locations were found to be:

$$A_0 = [10.0, 3.343, 4.5]^T \tag{B.17}$$
$$A_1 = [0.0, 3.091, 4.5]^T \tag{B.18}$$
$$A_2 = [10.0, 5.335, 4.5]^T \tag{B.19}$$
$$A_3 = [0.0, 6.696, 4.5]^T \tag{B.20}$$
$$A_4 = [10.0, 8.872, 4.5]^T \tag{B.21}$$
$$A_5 = [0.0, 8.522, 4.5]^T \tag{B.22}$$

In total 5 million different sensor location combinations were evaluated and the total computation time was 6 minutes and 21.7 seconds.

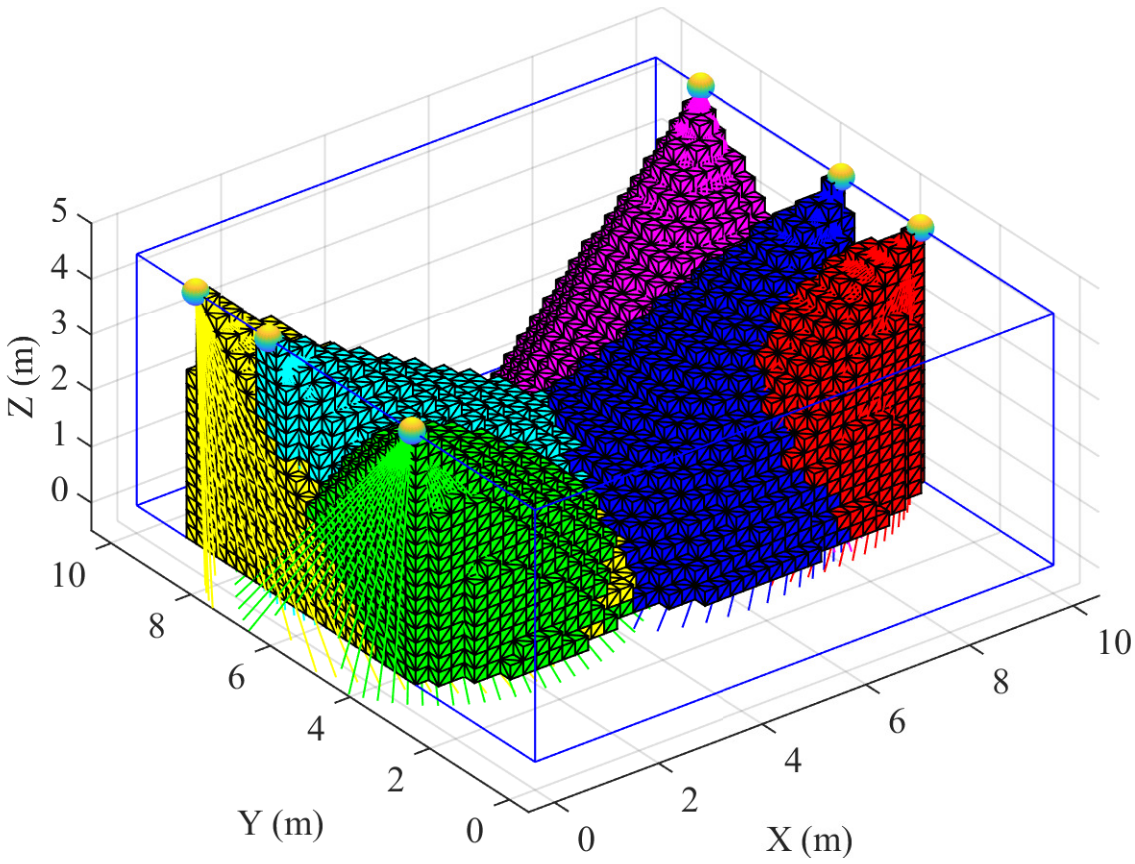

Figure B.9: Case Study V: Result of 3D sensor placement with six sensors. Discretised cuboid size equals 0.25 m × 0.25 m × 0.25 m.

The results from the five different sensor placement test cases are summarised in Table B.1.

Table B.1: Computation time for the different cases using an NVIDIA GTX 1080 GPU.

Sensors	Cuboids	Iterations	Redundancy	Comp.Time (s)
2	8000	$1 \cdot 10^6$	No	18.3
4	8000	$5 \cdot 10^6$	No	93.8
4	8000	$5 \cdot 10^6$	Yes	187.9
6	3600	$5 \cdot 10^6$	Yes	180.0
6	28 800	$5 \cdot 10^6$	Yes	381.7

B.5 Discussion and Conclusions

The presented solutions for the selected case studies are not trivial to find, in particular for the cases IV and V where 6 sensors are considered and redundancy requirements are included. The solution to the largest problem (case study V) was found in a few minutes compared to several hours or days for gradient-based and mixed-integer based approaches.

Due to more GPU memory read, writes and copies, using the redundancy functionality increases the computation time, as seen in Table B.1. However as seen when comparing case study IV and V, the computation time only scales by a factor of two even though the number of cuboids are scaled by a factor of 8. This is due to the large capacity of the GPU and that most of the added work is parallelised through CUDA kernels.

When adding more free variables it takes more iterations to find the best result when using random number generation. However, the computation times with the method presented in this paper are much lower than comparable gradient-search and mixed-integer based solutions. When optimising sensor placement, a calculation time of several hours or even days would normally be acceptable, which would allow for solving very large problems and compute a large number of iterations using GPUs.

Random sampling is common in many big data and machine learning algorithms, but it can not guarantee to find the best solution with a fixed number of iterations. With a brute-force search the best solution can be guaranteed to be found, but that could take a long time. The use of random numbers usually give good solutions within a short time period. With random numbers you can still continue the computations to make sure you did not miss the optimum, or try a gradient approach at the end with the current best solution as a starting point to find the closest (local) minimum.

In future expansions, the maximum number of sensors could be increased from 32 by e.g. using 64 bit integers in the cover matrix and/or running multiple CUDA grids concurrently on the GPU, where each grid writes to its own cover matrix. Dividing calculations for different sensors across different grids could also allow for processing volumes containing more cuboids, due to the limits on the CUDA grid dimensions.

Computation time could also probably be lowered further by moving more of the program into CUDA kernels, e.g. the generation of random variables.

Other planned future work is: graphical user interface (GUI) for intuitive definition of constraints, enabling sensor rotations as free variables, inclusion of "pyramidical" and other field-of-views (better representing lidars for example), return of several (the n best) solutions and simultaneous distribution of workload on multiple GPUs for scaling to larger problems. In the final version of the paper the source code will be made publicly available on GitHub.

B.6 Acknowledgement

The research presented in this paper has received funding from the Norwegian Research Council, SFI Offshore Mechatronics, project number 237896, https://sfi.mechatronics.no

References – Paper B

[B1] D. Göhring, M. Wang, M. Schnürmacher, and T. Ganjineh. Radar/lidar sensor fusion for car-following on highways. In *Proc. Intl. Conf. Automation, Robotics and Applications (ICARA)*, 2011.

[B2] Y. Alkhorshid, K. Aryafar, S. Bauer, and G. Wanielik. Road detection through supervised classification. In *Proc. Intl. Conf. Machine Learning and Applications (ICMLA)*, 2016.

[B3] E. Ward and J. Folkesson. Vehicle localization with low cost radar sensors. In *Proc. IEEE Intelligent Vehicles Symposium*, 2016.

[B4] Joacim Dybedal and Geir Hovland. Optimal placement of 3d sensors considering range and field of view. In *Proc. IEEE Intl. Conf. on Advanced Intelligent Mechatronics (AIM)*, 2017.

[B5] U.M. Erdem and S. Sclaroff. Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements. *Computer Vision and Image Understanding*, 103:156–169, 2006.

[B6] Nicolaj Kirchhof. Optimal placement of multiple sensors for localization applications. In *Proc. IEEE Intl. Conf. on Indoor Positioning and Indoor Navigation*, 2013.

[B7] E. Hörster and R. Lienhart. Optimal placement of multiple sensors for localization applications. In *Proc. 4th ACM Intl. Workshop on Video Surveillance and Sensor Networks*, 2006.

[B8] H. Topcuoglu, M. Ermis, I. Bekmezci, and M. Sifyan. A new three-dimensional wireless multimedia sensor network simulation environment for connected coverage problems. *Simulation: Trans. Society for Modeling and Sim. Intl..*, 88(1):110–122, 2006.

[B9] P. Rahimian and J.K. Kearney. Optimal camera placement for motion capture systems. *IEEE Trans. on Visualization and Computer Graphics*, 23(3):1209–1221, March 2017.

[B10] Shenyu Mou, Yan Chang, Wenshuo Wang, and Ding Zhao. An optimal lidar configuration approach for self-driving cars. *CoRR*, abs/1805.07843, 2018.

[B11] Zuxin Liu, Mansur Arief, and Ding Zhao. Where should we place lidars on the autonomous vehicle? - an optimal design approach. In *Proc. 2019 Intl. Conf. Robotics and Automation (ICRA)*, 2019.

[B12] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach 3rd Ed.* Morgan Kaufmann, 2016.

[B13] Atle Aalerud, Joacim Dybedal, and Geir Hovland. Automatic calibration of an industrial rgb-d camera network using retroreflective fiducial markers. *Sensors*, 19(7), 2019.