Implementing Communicable Measurement Grid Algorithms on Graphical Processing Units

Menna El-Shaer
The Ohio State University
205 Dreese Labs, 2015 Neil Ave, Columbus, OH 43210 USA
el-shaer.1@osu.edu

John M. Maroli
The Ohio State University
205 Dreese Labs, 2015 Neil Ave, Columbus, OH 43210 USA
maroli.2@osu.edu

Abstract

Occupancy grid maps are a commonly used and highly practical method of representing the occupancy state of space surrounding a vehicle across a probabilistic range. They serve as a suitable medium for representing the fusion of multiple sensors and can be transmitted between vehicles with lower data overheads compared to sharing raw data. Occupancy grid maps are constructed using measurement grids, which represent the spacial occupancy of a sensor as strictly free or occupied. This work introduces a compact measurement grid representation tailored toward vehicle to vehicle (V2V) and vehicle to infrastructure (V2I) communication. The representation's data size is not only lower, but can also be adjusted for varying data transmission rates. This new measurement grid representation can lower bandwidth requirements but must be reconstructed by the receiving party. It's structure, however, is ideal for parallel processing on a GPU (Graphics Processing Unit).

In this work, we demonstrate how grid maps can be reconstructed from LIDAR point cloud sensor data, that are usually large, on a multiprocessing system, using the CUDA programming model. We will discuss our implementation and give general tips for migrating serial CPU code to a GPU system.

2018

Keywords: occupancy grid map; measurement grid; vehicle-to-vehicle; vehicle-to-infrastructure; Graphical Processing Units; Parallel Programming; CUDA; LIDAR Sensors

1. Introduction

Occupancy grid maps serve as a valuable media for storing spatial occupancy data obtained from a time-series of sensor measurements and/or a fusion of data from different sensors. They are a probabilistic representation of space, often used in robotics and intelligent vehicles [1]. Fusing occupancy grid maps from multiple vehicles can extend the perception range of vehicles involved in sharing. The occupancy grid is beneficial in reducing data sharing bandwidth requirements in comparison to sharing raw data [2], however bandwidth reductions can still yet be beneficial since the current Dedicated Short Range Communication (DSRC) standard for sharing data may not be able to support large-scale deployment [3].

Reconstructing grid maps from the large amount of sensor data, to be shared between vehicles, is memory and time consuming on the processing system. The availability of multicore processors and System-on-chips (SoCs) nowadays has made the task more feasible. However being able to use said systems efficiently is necessary to achieve the real-time constraints, most intelligent vehicles require such as reconstructing occupancy grid maps.

In this paper, we start by giving a brief review of occupancy grid mapping methods, followed by an introduction to heterogeneous processors of which GPUs are an example, in section 2. The core of the paper is divided into two parts: In section 3, we describe our occupancy measurement algorithm along with results and discussion; the parallel implementation and on the GPU and its discussion are described in section 4, comprising the second half of the paper. We end with a general discussion and suggested future work.

2. Background

2.1. Occupancy and Measurement Grids

The occupancy grid is constructed from a number of measurement grids using a technique such as the binary Bayes filter [4], where each measurement grid represents the spatial occupancy from a single sensor at a single time step. The measurement grid can be represented as a 2D image with only one of two occupancy states for known cells; free or occupied. Similarly, the occupancy grid map represents spatial occupancy but each cell holds a value representing the probability of occupancy in a range. As a result, an occupancy grid map with an equal number of cells as a measurement grid will contain more occupancy information but at the cost of a larger data size.

We represent the occupancy grid as a 2D array of cells akin to a grayscale image, where a pixel value of 0 represents free space (white) while a pixel value of 255 represents occupied space (black). Any intermediate values represent the probability of spatial occupancy. We represent a measurement grid similarly as a 2D array of cells, except the cells can hold states representing the sensor measurement; free (white), occupied (black), or unknown (gray). All measurement and occupancy grids in this work are 2D where the focus is a ground vehicles plane of movement.

Since an occupancy grid can be constructed from a series of measurement grids, the measurement grids can be transmitted between vehicles in lieu of the occupancy grid and the occupancy grid constructed on the receiving vehicle. This may be advantageous depending on the transmittable size of the occupancy grid map and the number of measurement grids desired to be shared. If sharing only the latest measurement grid from a vehicle's LIDAR sensor for example, the measurement grid will yield lower data transmission costs and should be shared rather than the vehicle's occupancy grid map.

2.2. Heterogeneous Processing Model

GPUs are an example of heterogeneous processors, where two separate different architecture processors are connected via a PCIe bus. Each system has its own DRAM and resources. The GPU is comprised of a scalable array of *Streaming Multiprocessors* or SMs. Parallelism is achieved using NVIDIA's Single Instruction Multiple Thread (SIMT) model, where multiple threads are execute the same instruction in a group called *warp*. [5, 6, 7] The SMs are responsible for processing thread blocks of the kernel grid launched by the CPU co-processor which the programmer gets to control. Figure 1 shows the heterogeneous programming structure. [7]

3. Measurement Grid Algorithm

A compact measurement grid format is introduced in this paper that relies on the sectorized nature of many vehicle sensors, where the sensor values are obtained from some sector of the circle containing the sensor as its origin. Light Detection and Ranging (LIDAR) sensors are an ideal example of this, with many LIDAR sensors having a 360 degree field of view. The LIDAR sensor emits laser pulses and measures the return distance to each laser strike as well as information for determining the precise angle.

For creating the compact representation, the area within a specified radius around the vehicle or sensor is broken up into a designated number of sectors with the vehicle or sensor as the center of the grid at $(0, 0)$. Only the closest (x, y) spacial coordinate of objects is recorded for each sector, and empty sectors will be represented by an invalid data point such as $(0, 0)$. The compacted measurement grid can now be represented as an array of (x, y) coordinate pairs containing no occupancy information and sent over Dedicated Short Range Communication (DSRC) to neighboring vehicles.

To reconstruct the measurement grid from the compact representation, the receiving vehicle must process it to retrieve occupancy information, converting it into a standard measurement grid. The first step in processing the compact measurement grid is to connect points of neighboring segments using a line drawing algorithm such as Bresenham's line algorithm. Points on this line are assumed to be occupied space. Any points on the side of the line closer to the

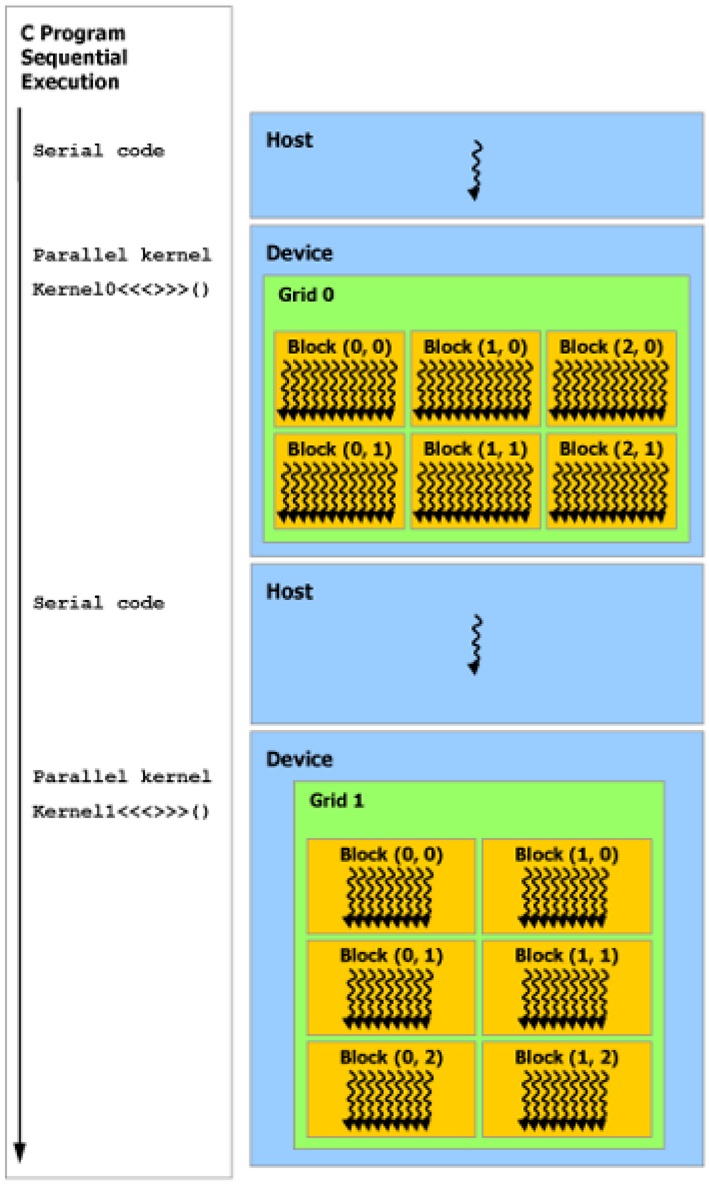


Figure 1: The Heterogeneous Programming Model [7]

vehicle or sensor are assumed to be free space while points on the side of the line further from the vehicle are marked as unknown space. Sectors containing points that border free space are split with a ray going from the vertex of the circle through the sector's coordinate and the edge of the map. Points on the side of this ray bordering the unoccupied sector are marked as free space, while points on the other space have unknown occupancy. The compressed and reconstructed measurement grid concepts are shown in 2 The computation overhead is not trivial, however measurement grids can be reconstructed in real time through parallel implementation on a GPU as shown in this work.

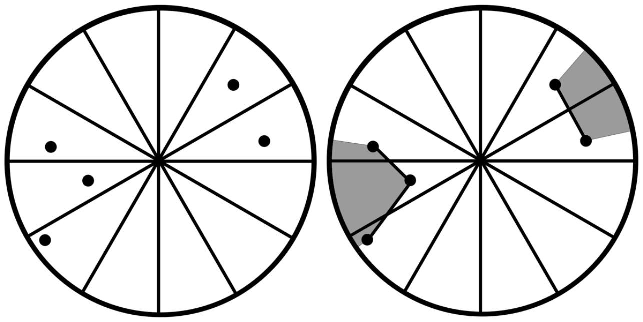


Figure 2: The compact measurement grid representation (left) and the reconstructed measurement grid (right)

3.1. Discussion and Results

The resulting measurement grid has inaccuracies incurred by the reconstruction process, but the closest occupied area in each sector has been preserved.The inaccuracies incurred in the reconstruction process may be insignificant for certain applications, especially when looking at the data bandwidth gained. For example, a compact measurement grid comprised of 360 sectors would have 360 coordinate pairs with each sector covering a 1 degree viewing angle. If using 16 bit floating point numbers, this compact representation would be 1,440 bytes in size. A similarly sized standard measurement grid using 2 bits per cell (to represent free, occupied, and unknown space) would be 1,444 bytes in size at a resolution of 76x76 pixels. For a 38 meter square grid, the cell size would be

50 centimeter square. The compact representation in this scenario can much more accurately represent the distance to objects since the exact point of closest distance to the vehicle or sensor is maintained in each sector. An example reconstructed measurement grid is shown in 3.

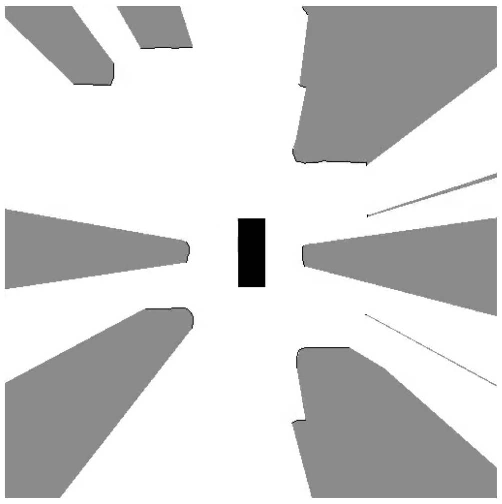


Figure 3: The reconstructed measurement grid of LIDAR data generated from a 180 sector compact measurement grid array. The resulting measurement grid is 512x512 cells. The black pixels in the center of the grid represent the vehicle that is capturing data while driving through a parking lot. White cells represent free space while black cells represent occupied space (vehicle edges in this scenario). The parked vehicles detected in this measurement grid occlude the space behind them, which is marked as unknown space and colored gray. The two thinnest occluded areas are the result of parking signs detected by the LIDAR sensor.

While Bresenham's line algorithm is used to connect the closest points from each sector in a smooth manner representing vehicle edges, it may lead to occupied space being marked as free space in sectors where neighboring points are further away than the closest point in the examined sector. To avoid occupied space being inferred as free space, the compact measurement grid can be represented as an array of n distances $[d_1, d_2, d_3, ..., d_n]$ to the closest sensor measurement in each sector. Cells along the circle of radius d_n in sector n are marked as occupied, while cells within the circle are marked as free and cells outside the circle are marked as unknown.

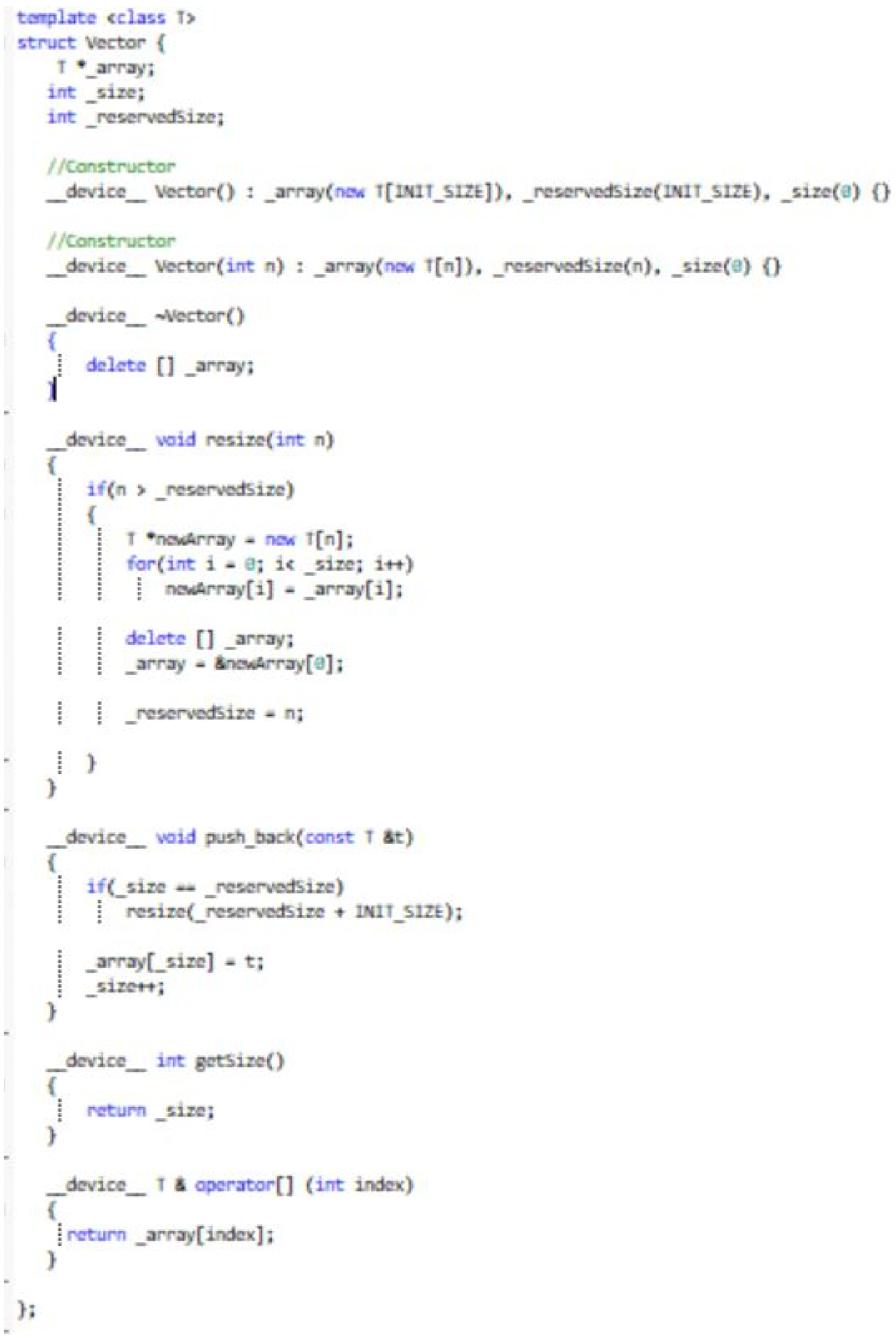
4. Parallel Implementation

In this section, we describe in detail how our parallel implementation of the measurement grid algorithm was done. Following the CPU-GPU memory organization and the CUDA programming model [7], to generate the grid sector data on the GPU, input sensor data must be copied from CPU memory to GPU global memory via the connecting PCIe bus. Global memory allocations and copies can be expensive, so choosing the appropriate data structures and containers is essential to ensure correct program execution and performance. The following subsection explains how we implemented our data structures for that purpose.

4.1. Parallel Data Structures

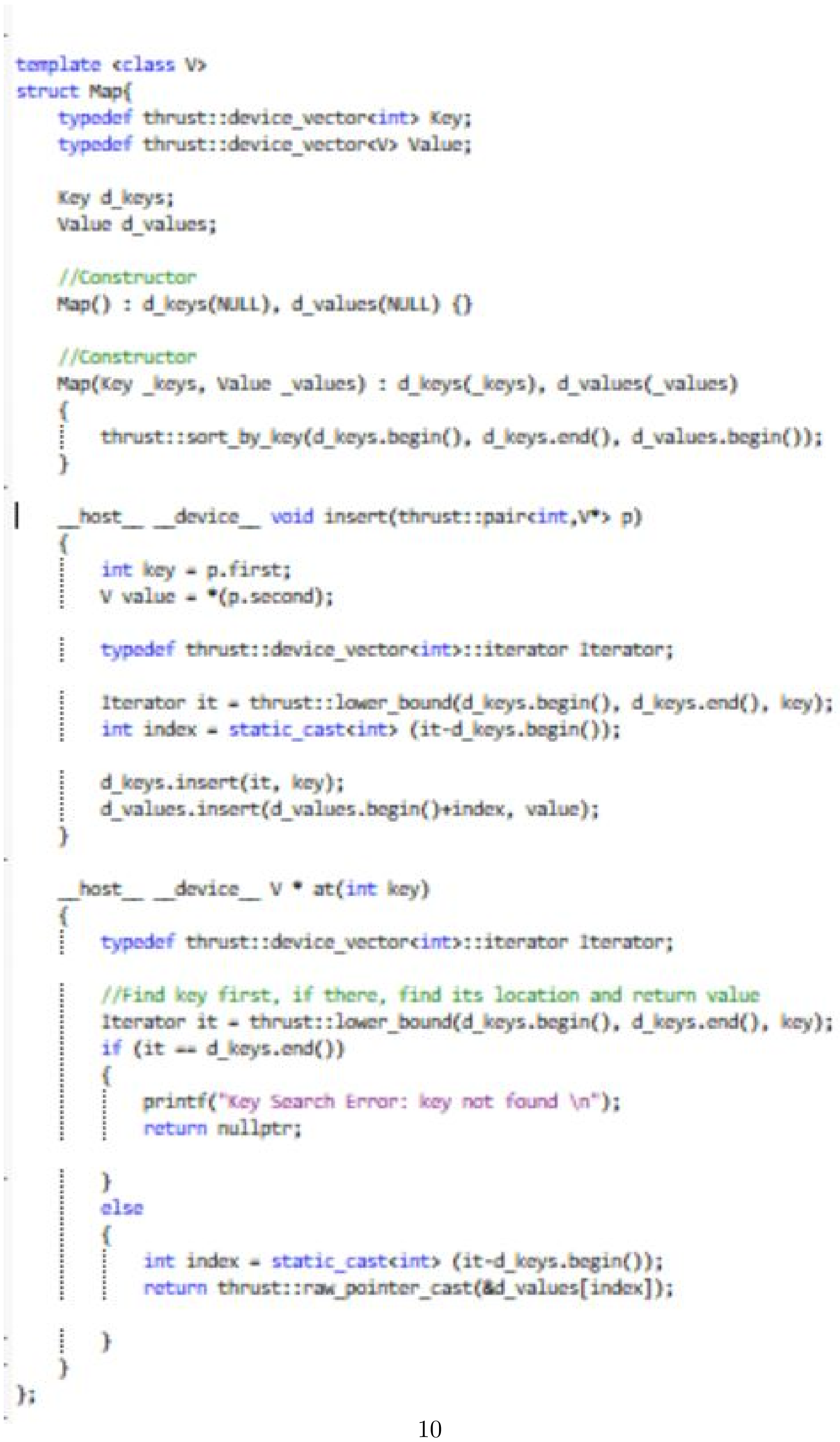
The C++ Standard Template Library (STL) [8] provides data structures and containers, that while convenient for writing serial code, are not supported by the NVIDIA CUDA Compiler Driver (NVCC) [9] to run on the device. As a result, we had to implement our own structures to run on the CUDA accelerated device. The two main basic containers that we needed were: vector containers to store 2D grid pixel coordinates in a segment; and map containers where the grid segment structures are sorted using their index as key. Although using already constructed containers from the STL templates was not feasible, the availability of the CUDA-accelerated Thrust library [10] was helpful, especially to allocate and handle host-side data.

As discussed in section 3, reconstructing the measurement grid is done in sectors *or segments*, this reconstruction is done in the SIMT fashion discussed previously. Using segment data in CUDA kernels requires them to be present in device memory, thus we needed portable containers that can be processed in both host and device memories. Figures 4 and 5 show our implementations of the vector structure implemented on the device and the map structure that could be implemented in both host and device memories, respectively.



```
template <class T>
struct Vector {
    T *_array;
   int _size;
   int _reservedSize;

   //Constructor
   __device__ Vector() : _array(new T[INIT_SIZE]), _reservedSize(INIT_SIZE), _size(0) {}

   //Constructor
   __device__ Vector(int n) : _array(new T[n]), _reservedSize(n), _size(0) {}

   __device__ ~Vector()
   {
       delete [] _array;
   }

   __device__ void resize(int n)
   {
       if(n > _reservedSize)
       {
           T *newArray = new T[n];
           for(int i = 0; i< _size; i++)
               newArray[i] = _array[i];

           delete [] _array;
           _array = &newArray[0];

           _reservedSize = n;

       }
   }

   __device__ void push_back(const T &t)
   {
       if(_size == _reservedSize)
           resize(_reservedSize + INIT_SIZE);

       _array[_size] = t;
       _size++;
   }

   __device__ int getSize()
   {
       return _size;
   }

   __device__ T & operator[] (int index)
   {
     return _array[index];
   }

};
```

Figure 4: Implementation of the Vector structure on the device



```
template <class V>
struct Map{
    typedef thrust::device_vector<int> Key;
    typedef thrust::device_vector<V> Value;

    Key d_keys;
    Value d_values;

    //Constructor
    Map() : d_keys(NULL), d_values(NULL) {}

    //Constructor
    Map(Key _keys, Value _values) : d_keys(_keys), d_values(_values)
    {
        thrust::sort_by_key(d_keys.begin(), d_keys.end(), d_values.begin());
    }

    __host__ __device__ void insert(thrust::pair<int,V*> p)
    {
        int key = p.first;
        V value = *(p.second);

        typedef thrust::device_vector<int>::iterator Iterator;

        Iterator it = thrust::lower_bound(d_keys.begin(), d_keys.end(), key);
        int index = static_cast<int> (it-d_keys.begin());

        d_keys.insert(it, key);
        d_values.insert(d_values.begin()+index, value);
    }

    __host__ __device__ V * at(int key)
    {
        typedef thrust::device_vector<int>::iterator Iterator;

        //Find key first, if there, find its location and return value
        Iterator it = thrust::lower_bound(d_keys.begin(), d_keys.end(), key);
        if (it == d_keys.end())
        {
            printf("Key Search Error: key not found \n");
            return nullptr;

        }
        else
        {
            int index = static_cast<int> (it-d_keys.begin());
            return thrust::raw_pointer_cast(&d_values[index]);

        }
    }
};
```

Figure 5: Implementation of the Map structure

4.2. Host Code

Global memory device allocations and copies have to be done by the host CPU [5, 6, 7, 11]. Allocating simple structures like arrays are usually done in a straighforward way using the CUDA API functions:*cudaMalloc()* and *cudaMemcpy()* [7]. However, using complex user-defined structures in device code requires a few more steps than straight *cudaMalloc()* and *cudaMemcpy()*. Detailed steps shown in figure 6 are explained below:

- First allocate space on host for any needed user-defined class instances.
- Allocate space on device using *cudaMalloc()* to store a copy of the instances defined on the host.
- Copy object from host to device using *cudaMemcpy()*.
- If object has a pointer member variable, the above steps should be repeated for each member variable. This ensures a *Deep Copy* process between the host and device.

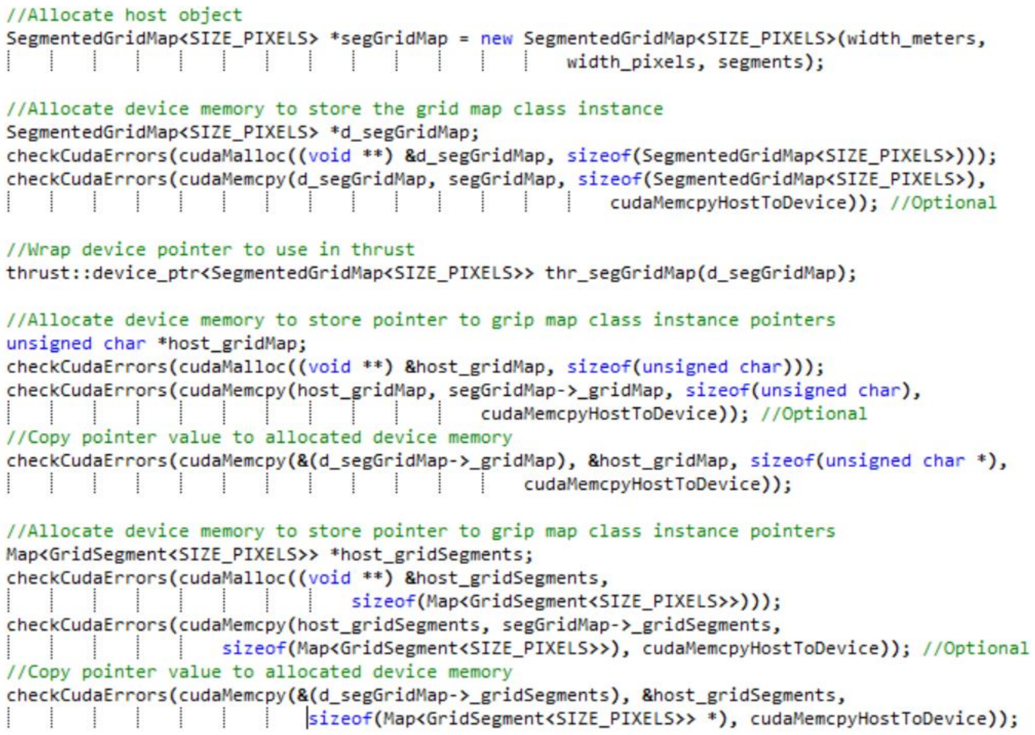
It is to be noted that Thrust library structures cannot be used directly on the device [10], thus a raw pointer to any those structures has to be passed to device kernels on launching. This was how we were able to use our input sensor data in our CUDA kernels.

4.3. Device Code

Filling up the grid map with input sensor data was done in parallel in a CUDA kernel. Input sensor data can be pretty large so dividing the input data among available threads was significant. As stated before, reconstructing the grid map along with all its computations were also implemented in device memory. Full code implementation can be found in the supplementary material.

4.4. Discussion and Results

The occupancy measurement grid algorithm was initially implemented in standard C++ to run on one CPU. However, map generation and sharing are



```
//Allocate host object
SegmentedGridMap<SIZE_PIXELS> *segGridMap = new SegmentedGridMap<SIZE_PIXELS>(width_meters,
                                                     width_pixels, segments);

//Allocate device memory to store the grid map class instance
SegmentedGridMap<SIZE_PIXELS> *d_segGridMap;
checkCudaErrors(cudaMalloc((void **) &d_segGridMap, sizeof(SegmentedGridMap<SIZE_PIXELS>)));
checkCudaErrors(cudaMemcpy(d_segGridMap, segGridMap, sizeof(SegmentedGridMap<SIZE_PIXELS>),
                                                         cudaMemcpyHostToDevice)); //Optional

//Wrap device pointer to use in thrust
thrust::device_ptr<SegmentedGridMap<SIZE_PIXELS>> thr_segGridMap(d_segGridMap);

//Allocate device memory to store pointer to grip map class instance pointers
unsigned char *host_gridMap;
checkCudaErrors(cudaMalloc((void **) &host_gridMap, sizeof(unsigned char)));
checkCudaErrors(cudaMemcpy(host_gridMap, segGridMap->_gridMap, sizeof(unsigned char),
                                             cudaMemcpyHostToDevice)); //Optional
//Copy pointer value to allocated device memory
checkCudaErrors(cudaMemcpy(&(d_segGridMap->_gridMap), &host_gridMap, sizeof(unsigned char *),
                                                 cudaMemcpyHostToDevice));

//Allocate device memory to store pointer to grip map class instance pointers
Map<GridSegment<SIZE_PIXELS>> *host_gridSegments;
checkCudaErrors(cudaMalloc((void **) &host_gridSegments,
                                 sizeof(Map<GridSegment<SIZE_PIXELS>>)));
checkCudaErrors(cudaMemcpy(host_gridSegments, segGridMap->_gridSegments,
                      sizeof(Map<GridSegment<SIZE_PIXELS>>), cudaMemcpyHostToDevice)); //Optional
//Copy pointer value to allocated device memory
checkCudaErrors(cudaMemcpy(&(d_segGridMap->_gridSegments), &host_gridSegments,
                               sizeof(Map<GridSegment<SIZE_PIXELS>> *), cudaMemcpyHostToDevice));
```

Figure 6: Allocating structures on the device and creating pointers to their members

both memory and time expensive, especially with increasing data size. Parallelizing the algorithm on the GPU presented commonly-faced challenges when migrating serial code for GPU processing: mainly system memory management and data dependencies.

While the GPU implementation of the measurement grid reconstruction reduced computational overhead; there are more optimization strategies that could be applied, but were not discussed in this paper [12], There is also room for creating original data structures that are optimized for parallel processing, for example, the commonly used "push back" style of appending vector element operations, that was used extensively in this work is not *parallel-friendly* since it usually involves an atomic or thread-locking process in parallel execution models which can cause unnecessary waits and hold-ups.

Occupancy [12] is important to ensure maximum resource usage given the amount of work required by each thread. Organizing our structures as Structure of Arrays (SOA) instead of Array of Structures (AOS) ensured device global memory coalesced access and efficient use of memory bandwidth. Utilizing the shared memories between threads in a block might result in faster execution times especially among neighboring thread segments.

Optimization for the Tegra X1 and X2 SoCs. Algorithms such as our grid mapping reconstruction are generally implemented in robotics and intelligent vehicle systems. These systems are usually run by System-on-chips (SoCs) as they consume much less power. Among our future interests is deploying our parallel algorithm on NVIDIA's Tegra X1 SoC [13] and optimizing it for real-time constraints.

5. Conclusion

As intelligent vehicles are equipped with more sensors each day, processing and extracting information from sensor data in a timely manner is essential to ensure their operation. The availability of faster system processors allows us to do just that. In this work, we have presented a sharable occupancy grid algorithm along with the parallel implementation of the grid reconstruction on Graphical Processing Units (GPUs).

6. References

References

[1] A. Elfes, Using occupancy grids for mobile robot perception and navigation, Computer 22 (6) (1989) 46–57.

[2] G. Ozbilgin, U. Ozguner, O. Altintas, H. Kremo, J. Maroli, Evaluating the requirements of communicating vehicles in collaborative automated driving, in: 2016 IEEE Intelligent Vehicles Symposium (IV), 2016, pp. 1066–1071. `doi:10.1109/IVS.2016.7535521`.

[3] Y. J. Li, An overview of the dsrc/wave technology (01 2012).

[4] K. C. J. Dietmayer, S. Reuter, D. Nuss, Representation of fused environment data (01 2016). `doi:10.1007/978-3-319-09840-1_25-1`.

[5] J. Sanders, E. Kandrot, CUDA by example : An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.

[6] D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors : A Hands-on Approach, Elsevier Science, 2012.

[7] NVIDIA, Cuda c programming guide (2018).
URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

[8] cplusplus.com (2000-2017).
URL `http://www.cplusplus.com/reference/stl`

[9] NVIDIA, Nvcc: Nvidia cuda compiler driver (2018).
URL `http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html`

[10] G. Barlas, Chapter 7 - the thrust template library, in: G. Barlas (Ed.), Multicore and {GPU} Programming, Morgan Kaufmann, Boston, 2015, pp. 527 – 573. doi:https://doi.org/10.1016/B978-0-12-417137-4.00007-1. URL https://www.sciencedirect.com/science/article/pii/B9780124171374000071

[11] Copyright, in: S. Cook (Ed.), {CUDA} Programming, Applications of GPU Computing Series, Morgan Kaufmann, Boston, 2013, pp. iv –. doi:https://doi.org/10.1016/B978-0-12-415933-4.02001-9. URL https://www.sciencedirect.com/science/article/pii/B9780124159334020019

[12] NVIDIA, Cuda c best practices guide (2018). URL http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[13] Nvidia tegra x1: Nvidia's new mobile superchip, Tech. rep., NVIDIA Corporation (Jan 2015).