CUDA Accelerated Robot Localization and Mapping

Haiyang Zhang, Fred Martin
Computer Science Department
University of Massachusetts Lowell
Lowell, MA 01854, USA
hzhang@cs.uml.edu, fredm@cs.uml.edu

***Abstract* — We present a method to accelerate robot localization and mapping by using CUDA (Compute Unified Device Architecture), the general purpose parallel computing platform on NVIDIA GPUs. In robotics, the particle filter-based SLAM (Simultaneous Localization and Mapping) algorithm has many applications, but is computationally intensive. Prior work has used CUDA to accelerate various robot applications, but particle filter-based SLAM has not been implemented on CUDA yet. Because computations on the particles are independent of each other in this algorithm, CUDA acceleration should be highly effective. We have implemented the SLAM algorithm's most time consuming step, particle weight calculation, and optimized memory access by using texture memory to alleviate memory bottleneck and fully leverage the parallel processing power. Our experiments have shown the performance has increased by an order of magnitude or more. The results indicate that offloading to GPU is a cost-effective way to improve SLAM algorithm performance.**

Keywords* — *robot; localization; mapping; SLAM; GPU; GPGPU; parallel; CUDA

I. INTRODUCTION

For a mobile robot in unknown environments, it is important to simultaneously localize itself and generate maps of the environments. The SLAM (Simultaneous Localization and Mapping) algorithm [1, 2] is usually used in these cases. Based on a probabilistic model, the SLAM algorithm estimates the robot state from its prior state, the current motor commands, and sensor readings. Particle filter-based SLAM is easy to implement and applicable to non-linear and non-Gaussian systems.

The particle filter is a sequential Monte Carlo method, in which system state is represented by a set of particles. Each particle is a data object containing one of the hypothetical robot states from the distribution and a "weight" value. In each sensing cycle, we calculate the weight value according to how closely this state matches the current sensor readings, and re-sample the particle set based on their weights.

To maintain an accurate representation of the state distribution, we must have a large number of particles, which makes the particle filter computationally intensive. But most computing steps in the particle filter are done independently on each particle, so it is inherently suitable for parallel processing.

CUDA (Compute Unified Device Architecture) [3] is a parallel computing platform running on NVIDIA GPUs (Graphics Processing Units). It is one of the popular GPGPU (General-Purpose computing on GPU) platforms. CUDA includes the compiler and driver to build and run CUDA C, which is an extended C/C++ language supporting both CPU and GPU, and communication between them.

The present work extends prior work in this area. Here is a brief review of related research done to accelerate particle filters and other robot applications with CUDA.

To efficiently utilize CUDA, Chao *et al.* describe an algorithm to implement a particle filter on CUDA [4]. Two enhancements are used—Finite-Redraw Importance-Maximizing (FRIM) prior editing and localized resampling. FRIM prior editing increases the coverage of the particles to important region of the state distribution. And, localized resampling reduces the overhead to access global memory. They use bearings-only tracking (BOT) problem for the performance benchmarking. The optimizations have increased performance by 5.73 times than a direct implementation on a GPU. This paper shows CUDA can effectively accelerate particle filter used in BOT problem, where the resampling step is the slowest step. But, in our work on SLAM problem, we have found that the weight calculation is the most time consuming step, and focused on accelerating this part of the particle filter.

Xu *et al.* present an implementation of the "saliency map model" on CUDA [5]. The saliency map model is a popular computational model for robotic vision to extract interesting objects from camera inputs. But the computational cost is high, and it is not efficient to run on CPU. This paper implements the saliency map model on CUDA-based GPU, and can process high speed camera inputs in real time, which is much faster than a standard CPU implementation. The implementation uses different memory types in the CUDA memory hierarchy according to the different requirement in each part of the algorithm.

GPU computing is used by Tuck *et al.* to accelerate a mobile robot control system [6]. The map-merging step involves combining laser rangefinder data with stereovision inputs, which is slow on a CPU. After porting this and some other steps to be run on GPU, and optimizing with a GPU-targeting compiler, Bacon, the overall performance has increased to near real time. The computing steps that are parallel in nature, including laser data processing and map merging, are accelerated greatly by GPGPU.

Also, Par and Tosun describe CUDA acceleration for localization based on GPS and map matching [7]. The vehicle location is estimated by a GPS reading first. Then current GPS

978-1-4673-6225-2/13/$31.00 ©2013 IEEE

and speed data are combined with odometer data and history locations to find best matches on the pre-loaded map by particle filter. The particle weight is based on zones and map topology. Running these procedures on GPU has largely improved performance.

These prior examples utilized GPGPU on various algorithms and robot applications, but have not implemented the SLAM algorithm on CUDA, which is the area we have developed in this paper. In our work, we analyzed the computing load of each step of the particle filter-based SLAM algorithm, and found that the particle weight calculation step consumes the majority of the CPU time. For comparison, we implemented this step on a multi-core CPU and a CUDA device. Because our access of memory is spatially localized, we were able to use the texture memory of the CUDA memory hierarchy to store the global map, thereby optimizing memory performance. We tested our program with data from a real robot, and found CUDA accelerated SLAM performs much faster than that on multi-core CPU with the same cost.

II. ALGORITHM AND OPTIMIZATION

The particle filter-based SLAM algorithm [2] is our localization and mapping algorithm. A particle is one of the possible robot states. Our robot runs on a building floor, and turns with differential steering, so the state includes three variables: x, y coordinates for its location on the floor and yaw for its orientation. For an unknown environment, the initial state is set to all zero. For each round of control and sensor data, the states are updated by motor commands including speed and turn-rate, electronic compass readings, and refined by the laser rangefinder data with the weighting and resampling. In this section, we first describe the algorithm in part A, and then introduce the CUDA memory hierarchy in part B, which is important for the optimizations in part C.

A. Particle filter based SLAM algorithm

1) Initialize particle states

We initialize all particle states to be x=0, y=0, and yaw=0, because we assume the robot is in an unknown environment. The particle set will be updated in the following step for each data cycle.

2) Read data from log file

The data in a new control and sensing cycle are read from a log file generated by a real robot for off-line processing. The data include the speed and turn-rate in motor control commands, the time elapsed since last data point, the electronic compass reading, and the laser rangefinder data.

3) Apply motion data with randomization

The motion data, including motor command and electronic compass reading, are converted to the distance and angle travelled by the robot. The hypothetical current state distribution is the conditional probability distribution:

$$p(X_t \mid X_{t-1}, U_t)$$

Of which, X_t is the current robot state at time t, and X_{t-1} is the previous robot state at time t-1, and U_t is the motion data including the distance and angle travelled by the robot since last time point.

But, we don't know the exact distribution of X_t, so the motion data including distance and angle are added into the state in each particle with some randomization to reflect the random error of the system and noise from the environment.

4) Copy data into the video card

The GPU can only access memory on the video card directly, so we need to copy the data, including the particle array and laser rangefinder readings from system (host) memory to GPU (device) memory.

5) Particle weight calculation

The particle weight W, reflecting how important this particle is among the state distribution, is the conditional probability of Z_t, assuming the robot state is X_t:

$$W = p(Z_t \mid X_t)$$

Of which, Z_t is the current observation data from the laser rangefinder, and X_t is the robot state in each particle.

It is not practical to calculate this conditional probability, because we do not know the error distribution. Instead, we use the matching score between the sensor readings and hypothetical states to estimate the conditional probability.

The laser rangefinder returns an array of ranges of the surrounding objects. For each particle, we calculate the object locations from the laser data and the particle state. These hypothetical locations are compared to the generated map in previous cycles, and the number of matches is counted as the particle weight.

6) Copy data from the video card

The CPU cannot access memory on the video card either, so we copy the particle weights back to host memory for other steps running on CPU.

7) Resample the particles

We resample the particles based on their weights. Particles with more weight are resampled with a higher probability. The resampling is done with replacement, and the total number of particles remains the same. The end result is that the particle set becomes more closely reflect the actual robot state distribution.

8) Estimate the robot location and update the map

We estimate the robot location by calculating the mean value of the resampled particle states. Then, the map is updated according to the current laser data. As the map representation, we used an occupancy grid map. The points reflecting laser rays are marked as blocked, and the region between laser rangefinder and these points is marked as open space.

Also, we transfer the map to the device memory to be accessed by GPU in the next cycle.

9) Go to step 2, read and process the next data point

See Fig. 1 for the algorithm flow chart.

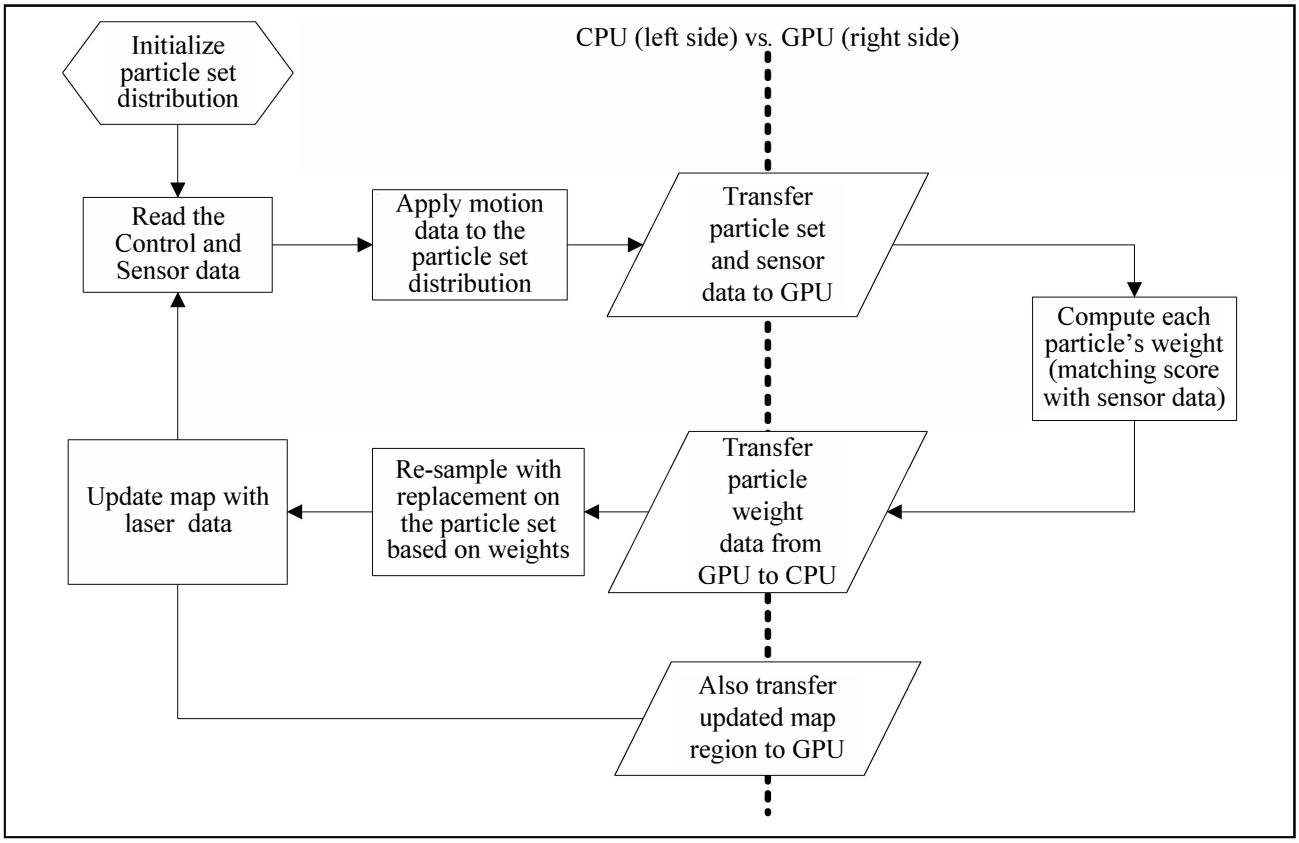

Fig. 1. Particle filter-based SLAM algorithm flow chart

B. CUDA memory hierarchy

Each NVIDIA GPU contains one or more streaming multiprocessors (SM), and each SM contains multiple CUDA cores [3, 8].

To alleviate memory bottleneck and fully leverage the GPU's parallel processing power, we also need to understand the CUDA memory hierarchy. As shown in Fig. 2, the registers, local, and shared memory are on the GPU chip, and provide fast access. The global memory is off-chip and larger, but is slower to access. The constant and texture memory is part of the global memory but is read-only by CUDA kernel (the code runs on GPU). Texture memory has a caching scheme optimized for spatial locality.

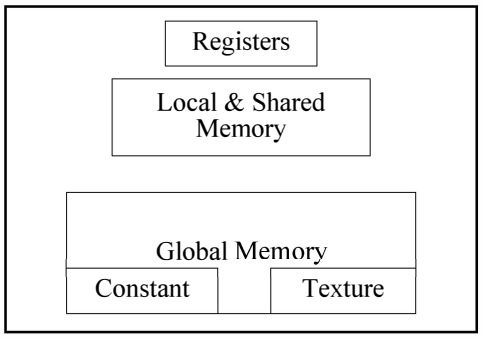

Fig. 2. CUDA memory hierarchy

C. Optimize SLAM for CUDA

To fully exploit the parallel computing power of GPU, we should have a large number of calls to the same function but using different data—Single Instruction Multiple Data (SIMD). These function calls should be independent of each other, so that one call will not require data from another call, and no locking or synchronization is necessary. The particle weight calculation step is computationally intensive, and each particle is independent of each other, so this step is selected to run on GPU.

The grid map is too large for on-chip memory, so it has to be on global memory. The particle set represents the hypothetical robot states, with random offsets added to reflect error range. The particles are usually close to each other on the grid map. The memory accesses to the map are localized in each particle set. And, the accesses are read only within one processing cycle. So, using texture memory should enhance the performance by leveraging the spatial locality of its caching.

Also, the memory copy between host and device is a relatively time consuming operation due to the large map size. If we copy the entire grid map (~36MB), it takes around 6ms in each cycle which is even longer than the weight calculation time on CUDA with texture memory as shown in Table 1. To minimize the overhead, we copy only a region of the grid map to the device at the end of each cycle. This region covers the area between the min and max addresses of the changed memory. We use only one region to cover all of the updated

map memory even there are some unchanged areas in between, because dividing them into multiple smaller regions will involve extra overhead for each region. With this optimization we have reduced map copy time from 6ms to 0.8ms.

The other data required for the weight calculation include laser scan pattern, which is a 181-length array (this is the same for every particle). The state includes 3 numbers (x, y, yaw) for each particle, and the returning weight is one number for each particle. Thus, the required data transfer between host and device for the weight calculation is not too large. See Fig. 3 for the algorithm pseudo code.

Algorithm 1: SLAM with CUDA Acceleration

```
1:   //GPU code:
2:   Function getWeight()
3:   i ← current thread index among all blocks
4:   step ← total number of threads on all blocks
5:   while i<N do  //N is the number particles
6:     calculate matching score of:
7:       X[i] + LaserData v.s. CurrentMap
8:     i ← i + step
9:   endwhile
10:
11:  //CPU code:
12:  Function main(datafile)
13:  X[1…N] ← {x=0, y=0, yaw =0}  //initialize all particles
14:  while (read(datafile) != EOF) do
15:    for i=1 to N do
16:      X[i] ← X[i] + Motion + RandomNoise
17:    endfor
18:    copyHostToDevice (X[1…N] )
19:    copyHostToDevice(LaserData)
20:    call getWeight() // on GPU
21:    copyDeviceToHost(W[1…N]) //copy particle weights
22:    resample(X[1...N]) based on W[1…N]
23:    Xmean = AverageState(X[1…N])
24:    updateMap(Xmean, LaserData)
25:    copyHostToDevice(ChangedMapRegion)
26:  endwhile
```

Fig. 3. Pseudo code of SLAM algorithm with CUDA acceleration

III. Implementation and Experiments

A. Hardware and development platform

To implement and test the performance of the SLAM algorithm on CUDA, we selected a GPU, the NVIDIA GeForce GTX 660, and a CPU, the Intel Core i5-3570K. These were mid-range devices with similar pricing when purchased.

NVIDIA GTX 660 [9] uses the GK106 "Kepler" GPU chip, which contains five streaming multiprocessors, and each multiprocessor includes 192 CUDA cores. There are 960 CUDA cores in total on the chip. The base clock rate is 980 MHz, and can boost up to 1033 MHz. The global memory size is 2 GB.

Intel Core i5-3570K CPU [10] is the third generation Intel Core series processors, which uses the "Ivy Bridge" microarchitecture. It contains four cores, and can run 4 parallel threads simultaneously. The clock frequency is 3.4 GHz, or 3.8 GHz in turbo mode.

The CUDA Toolkit 5.0 contains the latest compiler (nvcc), drivers, libraries, and code samples. It is used to compile our program which includes both CPU and GPU code.

Our experimental robot system is called "Stark," which was designed and built by our IGVC (Intelligent Ground Vehicle competition) team. The robot is four-wheel driving with skid steering. Its sensors include a SICK LMS200 laser rangefinder and electronic compass. The motion and sensor data were collected during our previous work [11], and saved into log files for offline analysis. The robot ran through the hallway in our computer science building. The motor commands (speed and turn rate), electronic compass readings, and laser scan data were recorded.

B. Identify the bottleneck on CPU implementation

To find the bottlenecks on CPU implementation, we first implemented the particle filter-based SLAM algorithm on the quad-core CPU. The weight calculation step runs on 4 cores in parallel, but still takes the majority of the computational time. As shown in Table 1, the weight calculation step on CPU consumes 95% of the time in each data cycle. So, to explore the massive parallel computing power of GPU, we focused on accelerating this step by using CUDA. And, the performance data on the quad-core CPU are used as the baseline to analyze the GPU acceleration results.

C. Implementation on CUDA

CUDA C is an extended C/C++ language, which has a similar syntax to standard C/C++, but with some extensions to call functions running on NVIDIA GPU, also to access the metadata such as the thread dimension and index. The CUDA platform consists of hundreds of cores — 960 in our case. The number of threads can be larger than the number of cores, and scheduled by the CUDA runtime automatically. The threads are further grouped into blocks; threads within each block can access a set of shared memory. A specialized function call is made to call CUDA kernel (the code running on GPU exclusively). The CUDA kernel code can make calls to other functions labeled as GPU code, or functions in CUDA libraries. But the kernel cannot call the regular functions in host code nor standard C libraries. The number of blocks and the number of threads in each block are specified in the call.

To reduce memory traffic between host memory and device memory, we only copy the required data in each processing cycle. These data include laser scan pattern, particle states before calling the kernel, particle weights after calling the kernel, and a region covering all changed portions of the grid map at the end of each cycle.

After passing the particle set and laser scan pattern, each GPU thread access its assigned particle state by using its block and thread indexes. The algorithm for weight calculation is same as the CPU version.

D. Optimization by using texture memory

As we discussed previously, the memory access of the grid map during weight calculation has spatial locality. Texture memory has the appropriate caching scheme to speed up caching efficiency for this case. Texture memory is still part of the global memory, but bound to texture mode after allocation.

It can be updated via host to device memory copy operations, but cannot be written by CUDA kernel. We used texture memory for the grid map, because the map is not updated during the weight calculation step.

E. Experiment results

As shown in Table 1, the weight calculation step is the most time consuming step in the algorithm, even it is already running in parallel on the quad-core CPU.

After ported the weight calculation step to CUDA, the average time of this step dropped from 100.5ms to 6.67ms, which is 15 times improvement in performance.

With texture memory, the performance improved even more. The weight calculation step has improved 28 times to 3.55ms, and the entire cycle has improved 11 times.

Fig. 4 shows two different paths traveled by the robot in our building. The maps generated by our SLAM algorithm during the two runs are shown in Fig. 5.

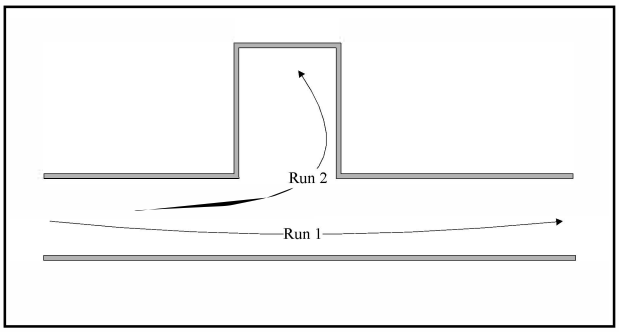

Fig. 4. The paths of the robot. In "Run 1", the robot goes from one end of the corridor to the other end. In "Run 2" it goes from one point from the left side to the middle area, which is the elevator hallway, with some turns.

TABLE I. PERFORMANCE COMPARISON BETWEEN CPU-ONLY AND CUDA-ACCELERATED IMPLEMENTATIONS

Average time spent to	**CPU only** (milliseconds)	**CUDA accelerated without Texture memory** (milliseconds)	**CUDA accelerated with Texture memory** (milliseconds)
Read a new data point from the log file	0.177893	0.188451	0.191382
Apply motion data with randomization	1.94319	1.91583	1.90274
Copy data from host to device	0	0.24669	0.249476
Weight calculation	100.502	6.66587	3.55373
Copy data from device to host	0	0.0852671	0.086652
Resample the particle set	1.34339	1.30413	1.30622
Update map and transfer to device memory	1.45296	2.24304	2.25671[a]
TOTAL	**105.41**	**12.65**	**9.53**[b]

[a] There is a slight time increase in the map update step, because the changed region of the grid map needs to be copied into device memory, which consumes about 0.8ms on average.

[b] The CUDA-accelerated version has the weight calculation, the most computationally intensive step, running on GPU. Considering this step alone, the CUDA without texture memory increased performance by 15 times, and CUDA with texture memory increased performance by 28 times. With the entire cycle taken into account, the CUDA with texture memory increased performance by 11 times totally.

IV. CONCLUSION

The particle filter-based SLAM algorithm is flexible for unknown error and noise distributions, and maintains accuracy when the number of particles is large enough. But that leads to intensive computation, especially in the particle weight calculation step during our performance tests.

In this paper, we explored the methods to accelerate the particle filter based SLAM algorithm by NVIDIA's general purpose GPU computing platform—CUDA. We have ported the particle weight calculation step, which consumes the majority of computational time on CPU, into CUDA. We also used the video card's texture memory, which has a caching scheme of spatial locality, to speed up our access to the grid map memory.

To test the performance on CUDA, we selected a quad-core CPU and a NVIDIA GPU at the same price to compare the computation time. Our tests have shown significant performance improvements of the CUDA-accelerated SLAM. The implementation on CUDA with texture memory increased performance by 28 times of the weight calculation step. With the entire data processing cycle taken into account, the CUDA with texture memory increased performance by 11 times. So, our results indicate using CUDA is a cost effective way to accelerate particle filter based SLAM algorithm.

FUTURE WORK

In this work, we have improved performance of the particle filter-based SLAM algorithm significantly by using CUDA. But, we believe there is still more potential to explore in this direction. In the near future, we are considering to accelerate other steps in the algorithm, and further improve the weight calculation step already ported into CUDA.

For example, in the step applying the motion data into the current particle set, the operations are independent in each particle. So it should be suitable for parallel processing on CUDA.

Regarding the weight calculation step, we are thinking about some methods to further improve the memory access spatial locality. Because the same sensor readings are shared among all particles, we can group the computations according to the laser scan angles, and check if the memory access efficiency can be further enhanced.

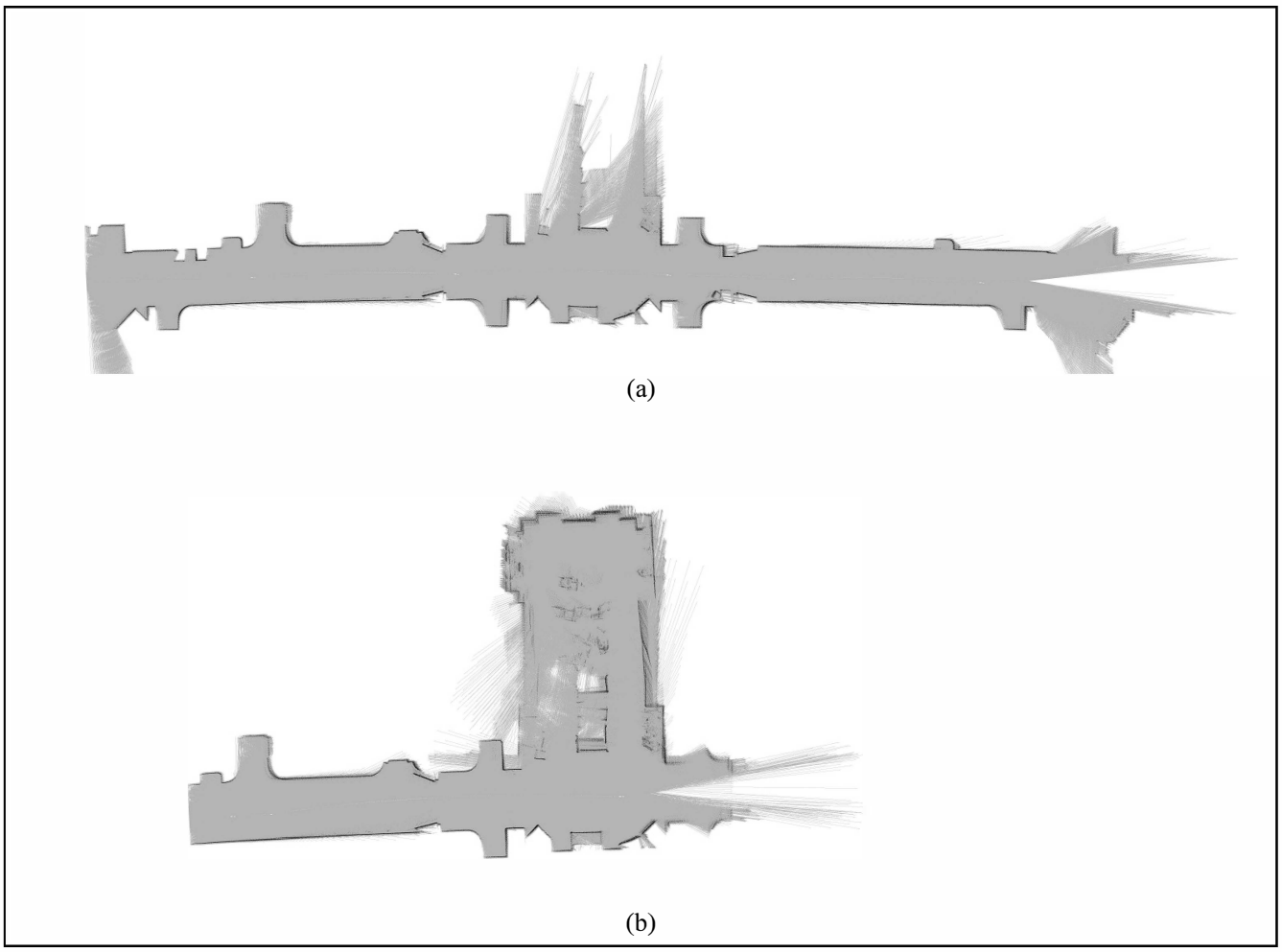

Fig. 5. Mapping results from SLAM algorithm when the robot runs through our building hallway. The gray area is open space, and the black lines are object boundaries, such as walls, and the white area are unknown space. (a) is the map from run 1. (b) is the map from run 2

ACKNOWLEDGMENTS

Thanks to Nat Tuck for providing some GPGPU related information. And, thanks to James Dalphond and John Fertitta for their assistance on the experimental robot system.

REFERENCES

[1] J.J. Leonard, H.F. Durrant-whyte, "Simultaneous map building and localization for an autonomous mobile robot," Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 1991), pp.1442-1447.

[2] S. Thrun, W. Burgard, and D. Fox, Probabilistic Robotics, MIT Press, 2006.

[3] J. Sanders and E. Kandrot, CUDA by Exmaple - An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2011.

[4] M. Chao, C. Chu, C. Chao, and A. Wu, "Efficient parallelized particle filter design on CUDA," Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS 2010), pp.299-304.

[5] T. Xu, T. Pototschnig, K. Kuhnlenz, and M. Buss, "A high-speed multi-GPU implementation of bottom-up attention using CUDA," Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2009), pp.41-47.

[6] N. Tuck, M. McGuinness, and F. Martin, "Optimizing a mobile robot control system using GPU acceleration," Proceedings of the Society of Photo-Optical Instrumentation Engineers Conference Series (SPIE 2011), vol.8301.

[7] K. Par and O. Tosun, "Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures," Proceedings of the 2011 IEEE Intelligent Vehicles Symposium (IV 2011), pp.820-826.

[8] R. Farber, CUDA Application Design and Development, Morgan Kaufmann, 2011.

[9] NVIDIA GeForce official website: http://www.geforce.com/

[10] Intel official website: http://www.intel.com/

[11] H. Zhang, F. Martin, "Robotic mapping assisted by local magnetic field anomalies," Proceedings of the IEEE International Conference on Technologies for Practical Robot Applications (TePRA 2011), p25-30.