Model Predictive Control for Autonomous Navigation Using Embedded Graphics Processing Unit

Duc-Kien Phung, Bruno Hérissé, Julien Marzat, Sylvain Bertrand

ONERA - The French Aerospace Lab, F-91123 Palaiseau, France (duc_kien.phung,bruno.herisse,julien.marzat,sylvain.bertrand@onera.fr)

Abstract: The objective of this work is to implement a Model Predictive Control (MPC) algorithm on an embedded Graphics Processing Unit (GPU) card. A MPC model for the autonomous navigation of a ground mobile robot is proposed. GPU CUDA code implementation and CUDA optimization techniques are discussed for this specific problem. The GPU-accelerated application permits extending the prediction horizon and evaluating more future trajectories compared to usual time-constrained CPU implementations. Simulation results and a preliminary experiment are presented to demonstrate the efficiency of the real-time algorithm.

Keywords: Model Predictive Control, Graphics Processing Unit, Embedded Control, Mobile Robot, Autonomous Navigation

1. INTRODUCTION

Model Predictive Control (MPC) is an appealing control strategy for real-time navigation of many systems, including mobile robots, Unmanned Aerial Vehicles (UAV), etc. MPC strategy uses the system dynamical model to predict the future state of the system. At each time step, a performance criterion is optimized for computing control inputs to reach pre-defined goals (Findeisen et al., 2003). Unlike most other methods, MPC considers a realistic dynamical model of the system and may also consider the changes in environment in real-time.

Some of popular optimization methods for MPC include Sequential Quadratic Programming (SQP), Active Set or Interior Point methods (Bartlett et al., 2000; Martinsen et al., 2004). However, a global solution can be hard to find because of potential local minima. Another basic strategy for computing MPC (sub-)optimal control input is systematic search (Frew, 2005; Bertrand et al., 2014). It has several advantages over traditional optimization procedure. Firstly, systematic search strategy can be less sensitive to local optimal problems, since the entire control space is explored (depending on the exhaustivity of control space discretization). Secondly, the computational load to find control sequence is constant in all situations leading to constant computation time. This is a necessary property to design real-time systems that require deterministic time response. Finally, the systematic search does not require an initialization of the optimization procedure. However, when many control candidates are evaluated for optimization, the computational load can be heavy. The usual solutions are to use a simpler parameterization of control sequence (e.g. constant input control over limited control horizon) or to limit the prediction horizon (Bertrand et al., 2014; Roggeman et al., 2016) but these approaches limit the potential of MPC. In some situations, a limited prediction horizon can lead to vehicle blockage near obstacles.

In order to maximize the prediction horizon as well as the number of control candidate sequences and to handle heavy computational load, a dedicated calculator like Graphics Processing Unit (GPU) can be used. Traditionally, GPU is designed with more and more rapid cores to satisfy the increasing demands of graphics/gaming industry. Nowadays, the modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor with very fast floating-point calculation and very high memory bandwidth that surpasses its CPU counterpart. Moreover, recent advances in semiconductor technology open up the possibility to embed high-performance yet low-power-consumption GPUs on small mobile robots or mini-UAVs. The objective of this work is to propose a MPC approach using GPU to control such robotic systems.

The paper is organized as follows. Section 2 recalls GPU programming model and other parallelized control algorithms available in the literature. Section 3 provides the general predictive control approach. Section 4 applies the MPC approach on a more specific case of mobile robot. The GPU code implementation details and optimization techniques are presented in Section 5. Simulation results are provided in Section 6 and compared with a nonparallelized MPC. Section 7 demonstrates a preliminary experiment on our mobile robot platform. Section 8 provides a brief summary of the paper and perspectives.

2. RECALLS ON PROGRAMMING GPUS AND PARALLELIZED CONTROL ALGORITHMS

2.1 CUDA Programming Model

In order to facilitate GPU programming, NVIDIA has been developing the leading proprietary programming model for GPUs, called CUDA (Compute Unified Device Architecture). CUDA provides an abstract scalable programming model. It is designed to be an extension to C. CUDA also supports a subset of C++, such as templates. The GPU is programmed by implementing device functions, called kernels. When a kernel is called, a thread grid is created to execute the kernel on the GPU. A thread grid is a 3D grid of thread blocks. Each block in turn is a 3D grid of CUDA threads. The programmer can specify the dimension of thread blocks and thread grids.

Threads within the same thread block are able to synchronize execution and share data by using the shared memory, but there is no synchronization available between different thread blocks. Thread blocks are required to execute independently: it must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of core (NVIDIA, 2015).

The thread blocks can allocate memory from shared memory and use it to share data between the threads in a thread block. Since shared memory is located inside on-chip streaming multiprocessor, the access latency is very fast. Utilizing shared memory is key to improving performance of memory-bound kernels.

2.2 Review on parallelized control algorithms

Generally, there are two approaches to accelerate a control algorithm (Huang et al., 2011). The problem-parallel approach consists in doing multiple execution of the same algorithm on different data. The data-parallel approach consists in exploiting the intrinsic property of the algorithm (e.g. structure of matrices).

In problem-parallel approach, the usual requirements are the problems have the same size and they are solved using the same algorithm. Typical examples are projectile Monte-Carlo trajectory analysis (Ilg et al., 2011) and real-time projectile guidance for impact area constraint (Rogers, 2013). In these examples, the same Monte Carlo dynamic simulation can be run hundreds or thousands of times in parallel using different initial conditions. This “single-program-multiple-data” framework is suitable for implementation on GPU.

Most of the algorithms in data-parallel approach exploit the structure of matrix calculation. Soudbakhsh and Annaswamy (2013) make use of the structure of the computations and the matrices (cost and constraint matrices) in the MPC to reduce the computational time. It can be shown that the computation time of MPC with prediction horizon H_p can be reduced to $O(\log_2 H_p)$. Specifically, a plant model with time delays was reformulated to get a form with sparse matrices. The computational time was analyzed to detect computational bottlenecks. The matrices in primal-dual method have block diagonal and block tridiagonal structures. Exploiting this sparsity, computations were parallelized to minimize the execution time.

In order to maximize the use of GPU/FPGA, the data-parallel approach can be implemented together with problem-parallel approach. The implementations are quite diverse. Examples include MPC implementation with interior point method (Constantinides, 2009; Gade-Nielsen et al., 2012, 2014), solving quadratic programming problem (Huang et al., 2011), RRT/RRT* motion-planning algorithm for a robotic manipulator (Bialkowski et al., 2011).

In this work, we have implemented a MPC control algorithm with the hybrid data-parallel and problem-parallel approach for a nonlinear system.

3. GENERAL MPC APPROACH

We consider the general discrete model of a vehicle dynamics:

$$\boldsymbol{x}(k+1) = f(\boldsymbol{x}(k), \boldsymbol{u}(k)), \tag{1}$$

where $\boldsymbol{x}$ is the state vector and $\boldsymbol{u}$ is the control vector. We define H_p as the prediction horizon and H_c ($1 \le H_c \le H_p$) as the control horizon.

Using the dynamical model (1), future control inputs and the resulting state trajectories of the vehicle can be computed as follows.

$$\begin{aligned} \boldsymbol{U} &= (\boldsymbol{u}(k)^\top, \boldsymbol{u}(k+1)^\top, ..., \boldsymbol{u}(k+H_c-1)^\top)^\top, \\ \boldsymbol{X} &= (\boldsymbol{x}(k+1)^\top, \boldsymbol{x}(k+2)^\top, ..., \boldsymbol{x}(k+H_p)^\top)^\top. \end{aligned} \tag{2}$$

For each future trajectory, a cost function J which represents the objectives of the mission is computed. Concretely, the optimization problem at time k is the following:

$$\begin{array}{ll} \text{minimize} & J(\boldsymbol{U}, \boldsymbol{X}) \\ \text{over} & \boldsymbol{U} \in \mathcal{U} \\ \text{subject to} & \forall t \in [k+1; k+H_p], \boldsymbol{x}(t) \in \mathcal{X} \end{array} \tag{3}$$

where $\mathcal{U}$ is the considered control space and $\mathcal{X}$ is the entire space free of obstacles. One possible solution for problem (3) consists in considering a finite set of predefined feasible control sequences, from which the one minimizing the cost function will be selected. Finally, the first command of that optimal sequence is applied to the system and the operation is performed again at the next time step.

Note that in our proposed method, at time step k, future trajectories over entire prediction horizon H_p and the cost function J are calculated based solely on current state $x(k)$ and predefined control sequences $\boldsymbol{U}$. This method is thus not suitable if there are changes in constraints on intermediate states $x(k+1), x(k+2), ..., x(k+H_p)$.

4. SPECIFIC MPC MODEL

We consider here the discrete dynamics of a mobile robot in 2D:

$$\begin{cases} x(k+1) = x(k) + \Delta t\, v(k) \cos(\theta(k)), \\ y(k+1) = y(k) + \Delta t\, v(k) \sin(\theta(k)), \\ \theta(k+1) = \theta(k) + \Delta t\, \omega(k). \end{cases} \tag{4}$$

where $\boldsymbol{x} = (x, y, \theta)^\top$ is the state vector containing the linear coordinates $(x, y)^\top$ and direction angle θ, $\boldsymbol{u} = (v, \omega)^\top$ is the control vector with linear speed v and angular speed ω, Δt is the sampling time step.

The constraints on the control inputs of system (4) are:

$$|v| \le v_{\max},\ |\omega| \le \omega_{\max}. \tag{5}$$

4.1 Cost function

The cost function J consists of a speed control cost J_v, an angular speed control cost J_ω, a speed regulation cost J_r, a navigation cost J_{nav}, and a safety cost J_{safe}. Each cost function includes a corresponding weighting factor, denoted by W. These factors are tuned by trial and error in simulation. The formulation of each cost function is detailed in the followings.

The control cost aims to limit the control effort and therefore the energy consumption. Both the speed and the angular speed control costs are defined as:

$$J_v(k) = \sum_{n=k}^{k+Hc-1} W_v v^2(n),\ J_\omega(k) = \sum_{n=k}^{k+Hc-1} W_\omega \omega^2(n). \quad (6)$$

The speed regulation cost aims to regulate the speed of the robot near a nominal value v_{nom} (can be negative):

$$J_r(k) = \sum_{n=k}^{k+Hc-1} W_r \frac{(|v(n)| - |v_{\text{nom}}|)^2}{(|v_{\text{nom}}| + v_{\max})^2}. \quad (7)$$

The robot is required to visit a certain waypoint at position $\boldsymbol{p}_{\text{goal}}$. The navigation cost is then proportional to the sum of square of distance between predicted position $\hat{\boldsymbol{p}}$ and the goal $\boldsymbol{p}_{\text{goal}}$:

$$J_{\text{nav}}(k) = W_{\text{nav}} \sum_{n=k+1}^{k+Hp} \|\hat{\boldsymbol{p}}(n) - \boldsymbol{p}_{\text{goal}}\|^2. \quad (8)$$

As for the safety cost, we define a safety function as follows:

$$f_{\text{safe}}(k) = \frac{1 - \tanh(\alpha(d(k) - \beta))}{2}, \text{ where} \quad (9)$$

$$\alpha = \frac{6}{d_{\text{des}} - d_{\text{sec}}},\ \beta = \frac{d_{\text{des}} + d_{\text{sec}}}{2}, \quad (10)$$

$d(k)$ is the distance between the robot position and the closest obstacle at time step k and d_{des} is the desired distance to the obstacles. Beyond this distance, the influence of the obstacles is ignored, d_{sec} is the security distance to obstacle that the robot must not cross. This continuous function f_{safe} is equal to 1 when $d(k) < d_{\text{sec}}$ and equal to 0 when $d(k) > d_{\text{des}}$. Finally,

$$J_{\text{safe}}(k) = W_{\text{safe}} \sum_{n=k+1}^{k+Hp} f_{\text{safe}}(n). \quad (11)$$

4.2 Control candidate selection

We define a certain number of control sequences where v and ω are varied. Let $N_{cs} \geq 3$ be an odd number of speed variations which are uniformly distributed from $-v_{\max}$ to $v_{\max}$. Define a positive integer $n_s = (N_{cs}-1)/2$. The speed control candidates are then:

$$v_i = \frac{i v_{\max}}{n_s} \text{ with } i = -n_s, -n_s+1, ..., n_s-1, n_s. \quad (12)$$

Similarly, let $N_{cy} \geq 3$ be an odd number of angular speed variations which are uniformly distributed from $-\omega_{\max}$ to $\omega_{\max}$. Define $n_y = (N_{cy}-1)/2$. The angular speed control candidates are then:

$$\omega_j = \frac{j \omega_{\max}}{n_y} \text{ with } j = -n_y, -n_y+1, ..., n_y-1, n_y. \quad (13)$$

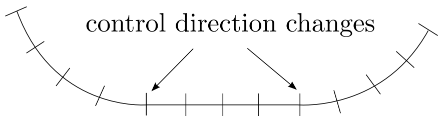

Fig. 1. Example of changing the control direction with control horizon $H_c = 12$, $D = 3$, $C = 4$

In a general case, the control candidate can be different in all predicted control steps. However, that makes the number of sequences extremely large. Instead, we defined a positive integer D as a number of possible different control values in the control horizon. Therefore, the control values will be unchanged in $C = H_c/D$ steps. The change of control values (change of angular speed in this case) is illustrated in Fig. 1.

Let consider for example a case where the prediction horizon is $H_p = H_c = 24$ steps. Choose the number of speed variations $N_{cs} = 7$ and angular speed variations $N_{cy} = 11$. We consider $D = 3$ different control values. Then, the number of possible control candidate sequences is $P = (N_{cs}N_{cy})^D = 456533$. The speed control candidate data will be a 2D matrix with column height P and row width H_p. In this example, the size of the 2D matrix is $L = P \times H_p = 10956792$. The angular control candidate and other intermediate prediction and cost matrices have large size as well. The computation of these huge data for prediction and cost is relatively slow on traditional CPU. This is where the advantages of GPU computing come into play. Well-designed CUDA program can handle these data in an efficient manner.

5. CUDA CODE IMPLEMENTATION

Below is the overview of algorithm steps with the parts that are executed on GPU in bold blue texts:

- Initialization
- Control vectors generation
- Copy control vectors to GPU device vectors
- Loop for each step until visited all waypoints
    - · Supervision (check waypoints achievement)
    - · Update environment map data to GPU
    - · **MPC**
        1. **Calculate predicted state using** (4)
        2. **Calculate cost components as in Section 4.1**
        3. **Find control candidate sequence that entails minimum cost**
    - · Publish the optimal control (speed and angular speed)

In order to obtain an efficient program, we implemented the following CUDA optimization techniques:

A) 1D vector representation for 2D array. As seen in the previous section, very large 2D arrays are used in our algorithm. Hence, it is imperative that 2D arrays are represented in efficient way. We choose to represented 2D arrays as 1D vector, with stride length equal to the width of 2D array. This is also the way 2D arrays are represented in memory (ISO, 2011, Section 6.5.2.1). This representation is also convenient for data manipulation within primitive operation library (CUDPP) below.
B) Efficient calculation of primitive parallel operations. Many intensive operations on CPU can be replaced

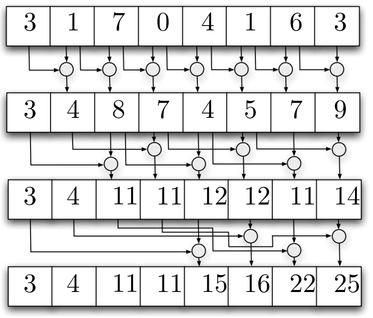

Fig. 2. Example of inclusive sum scan of $L = 8$ elements

by equivalent but more efficient parallelized CUDA codes. We illustrate this point by a simple example. In MPC algorithm, it is required to calculate the predicted values $\hat{\theta}$ of direction angle based on past angle and angular speed control as per (4):

$$\hat{\theta}_{iH_p+j} = \theta_0 + \Delta t S_j, \tag{14}$$

where θ_0 is the initial angle at this iteration step, and S_j is a cumulative sum defined as:

$$S_j = \sum_{k=0}^{j} \omega_{iHp+k}. \tag{15}$$

Predicted angle $\hat{\theta}$ and ω are essentially 2D matrices with size $L = P \times H_p$ and are represented as 1D vector with length L, as explained earlier. Implementations of simplified C++ code and CUDA code are shown in Fig. 3. Equation (15) is implemented by line 4 in C++ code, or equivalently by line 4 in CUDA code. Equation (14) is implemented by line 5 in C++ code, or equivalently by line 7 in CUDA code. The difference between them is that *for* loops are not needed in CUDA code since the calculations of equations (15) and (14) are implemented in parallel. The important operation here is to calculate the cumulative sum (running sum or scan) of the angular speed control ω (line 4 in C++ code in Fig. 3). The step complexity and work complexity of serial scan algorithm are both $O(L)$. When L grows big, the calculation time can be relatively long.

At first look, scan operation seems like a sequential operation since the running sum depends on past values. However, scan operation can be calculated very efficiently in parallel. An example of well-known parallel scan by Hillis and Steele (1986) is shown in Fig. 2. This algorithm requires $\log_2 L = 3$ steps, compared to 7 steps (7 additions) in serial scan. However, this algorithm applies the sum operator $O(L \log_2 L)$ times, which is asymptotically inefficient compared to the $O(L)$ applications performed by serial scan. Another principal difficulty in our application is calculating the running sums for P chunks of data with length H_p in parallel, i.e. a parallel segmented scan operation. For this operation, we make use of one of the most powerful scan algorithms designed for GPU in a specialized library called CUDPP (CUDA Data Parallel Primitives) (Sengupta et al., 2011). CUDPP is an utility library for data-parallel algorithm primitive operations. Step complexity of CUDPP scan algorithm is $O(\log_2 L)$ and work complexity is $O(L)$, both of which are asymptotically optimal. Using efficient parallelized algorithms like CUDPP segmented scan has greatly reduced the calculation time in our CUDA application, as will be shown in Section 6.

C) Fine-grained data-parallelism To efficiently utilize the GPU, a program must expose substantial amounts of fine-grained parallelism. For example, launching a kernel on the GPU is a relatively expensive operation. For kernels that perform simple operations like scaling or addition, the calculation time is very low and comparable to the time delay required to call the kernel itself. In order to maximize the GPU utilization, we write kernels that group data together to hide the kernel call latency. As an example, each thread in *vectorScaleAdd4* kernel (Fig. 3) groups and handles the calculation of 4 sets of data. Specifically, each thread performs 4 additions and 4 scaling operations. Simple optimization like this is important since CUDA program is required to handle very large amount of data. In practice, each kernel is optimized using profiling (time measurement using Eclipse Profiling Tool) to select suitable parameters like amount of data to be handled by each thread, kernel grid/block size, etc.

Other optimizations include extensive shared memory utilization and kernel concurrency. Large data such as occupancy maps are copied into fast on-chip shared memory to minimize access time. Whenever possible, kernels are executed concurrently to minimize overall execution time.

D) Limit memory management overhead The CUDA memory management functions *cudaMalloc* and *cudaFree* are more than two orders of magnitude more expensive than the equivalent C standard library functions *malloc* and *free*. In our code, whenever possible, memories (for example control candidate vectors) are allocated before MPC calculation loop. The memory is reused in each kernel invocation in the control loop. The memory is deallocated only when the control loop finishes.

E) Limit memory transfer overhead One of the most limiting overhead in parallel computing with GPU is the need to explicitly transfer data between CPU memory and GPU memory. Also, transfer a large chunk of data once is often faster than transfer small data many intermittent times. In our program, the data are grouped and transferred as few times as possible before MPC calculation.

F) Make use of fast intrinsic CUDA functions CUDA framework provides intrinsic functions (*_sinf*, *_expf*, etc) that are less accurate than standard math functions but execute faster (as they map to fewer native instructions). We trigger the option *use_fast_math* in the compiler option to make use of these intrinsic functions.

There are some challenges in the development with CUDA. For instance, CUDA does not support some standard libraries, e.g. *std::vector*. However, usually there are some high-level libraries available for the developers, e.g. Thrust library that can be used in replacement of these convenient tools. One challenge is that when using different libraries together, sometimes it is required to convert from one format compatible with a library to another - a task not so straightforward especially if the conversion has to be done in low-level device codes to optimize execution time.

6. MOBILE ROBOT MPC CONTROL SIMULATION

In this simulation, we run our GPU CUDA algorithm on a Jetson TK1 card with Tegra K1, a NVIDIA Kepler "GK20a" GPU with 192 CUDA cores (up to 326 GFLOPS). In order to compare the performance, we also

```
for (i=0;i<P;++i) {
  cum_sum_omega = 0.0;
  for (j=0;j<Hp;++j) {
    cum_sum_omega += omega[i*Hp+j];
    predicted_theta[i*Hp+j] = theta_0 + delta_t *
         cum_sum_omega;
  }
}
```

```
L = P*Hp;
//Using CUDPP library with a class planner scanplan
//iflags: flag vector to mark the beginning of a segment
cudppSegmentedScan(scanplan, cum_sum_omega, omega,
  iflags, L);

vectorScaleAdd4<<<gridSize,blockSize>>>(predicted_theta,
     cum_sum_omega, L, delta_t, theta_0);
```

Fig. 3. Comparison between C++ code (left) and CUDA code (right)

$v_{\max}$	1.0 m/s
v_{nom}	0.7 m/s
$\omega_{\max}$	0.5 rad/s

Table 1. Speed parameters

W_v	5	W_{safe}	150
W_ω	5	d_{des}	0.8
W_r	2	d_{sec}	0.6
W_{nav}	5		

Table 2. Cost parameters

H_p	24	N_{cy}	11
H_{cs}	24	D	3
H_{cy}	24	C	8
N_{cs}	7		

Table 3. Search procedure parameters

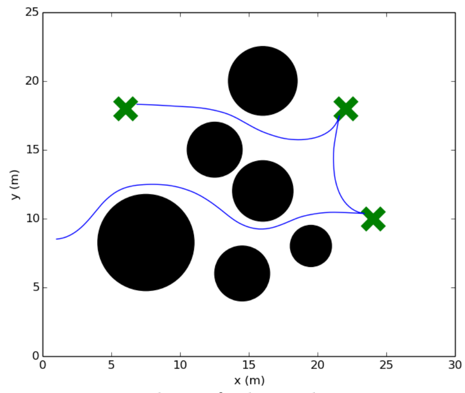

Fig. 4. Trajectory plot of the robot passing through 3 waypoints (green crosses) while avoiding obstacles (black circles)

run the equivalent CPU C++ code on a PC workstation Intel Xeon W3520 @ 2.67 GHz.

We consider a known environment with many fixed obstacles. The goal of the mobile robot is to visit a few waypoints one by one while avoiding obstacles. The sampling time is $\Delta t = 0.25$s. The speed parameters are shown in Table 1. The cost parameters are listed in Table 2.

In a specific example, we consider the search procedure parameters as shown in Table 3. The data vector length is $L = (N_{cs}N_{cy})^D \times H_p = 10956792$. Fig. 4 shows the trajectory plot (blue line) of the robot. The robot is able to find the waypoints (green crosses) while avoiding obstacles (black circles).

Fig. 5 compares the average execution time for the MPC code executed by CPU versus GPU for different data vector length L. For smaller data, the CPU C++ program executes faster since the clock rate of CPU is fast and the CPU is not limited by the memory transfer overhead. However, as the data grow larger and larger, the CPU C++ program takes exponentially longer. The GPU CUDA program execution time only increases slightly and it runs in real time (i.e. less than $\Delta t = 250$ms) for all tested lengths of data vector.

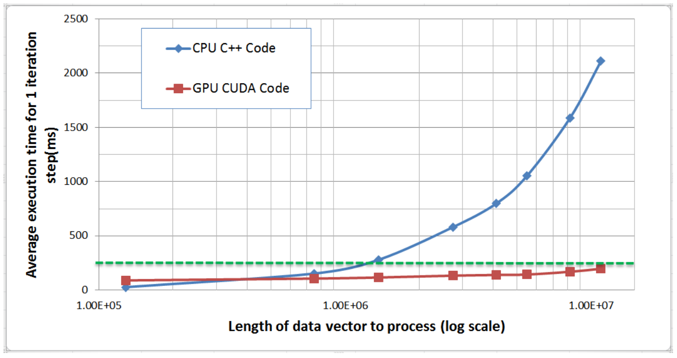

Fig. 5. Comparison of execution time of CPU C++ versus GPU CUDA programs with different data vector length L. Green dashed line represents the sampling time $\Delta t = 250$ms.

In some other simulations, we reduce data vector length L so that the CPU C++ program can run in real-time. However, the quality of predictive control degrades as compared to CUDA program (abrupt direction change near obstacles due to short prediction horizon, robot blockage between several obstacles due to fixed control values over control horizon, etc).

7. PRELIMINARY EXPERIMENT

We performed a preliminary experiment on a four-wheel Robotnik Summit XL robot (Fig. 6). The robot is equipped with an Asus Xtion depth sensor to create an occupancy map. The localization of the robot uses wheel odometry and Inertial Measurement Unit.

The Tegra card is mounted on the front part of the robot. It is powered by a 12V portable battery. The Tegra card is connected to the mobile robot on-board PC by LAN cable. Communication between Tegra card and other modules is facilitated using ROS (Robot Operating System by Quigley et al. (2009)) middleware. The Tegra card receives odometry data and projected occupancy map from appropriate topics published by the on-board PC program. Then, the control algorithm is executed on Tegra card as described in Section 5. After finding the optimal trajectory, control signals (speed and angular speed) are published to a ROS topic. The on-board PC program then subscribes to this topic and control the motors accordingly.

The experiment was carried out in a parking in our research center (Fig. 7). A short video of this experiment is available at `http://bit.ly/2nwgXmy`.

The robot was able to successfully find 2 waypoints while avoiding the obstacles (mainly 2 pillars) as depicted in Fig. 8. This experiment shows that our algorithm can perform in real-time and achieve safe autonomous navigation. In the near future, we will perform experiments with more

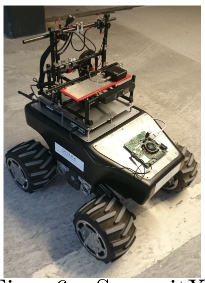
Fig. 6. SummitXL robot

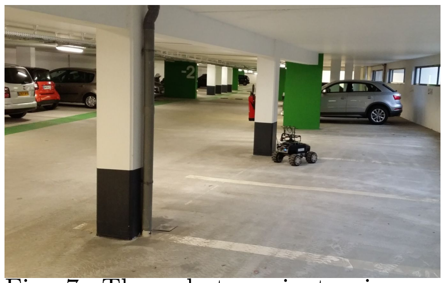
Fig. 7. The robot navigates in our experiment

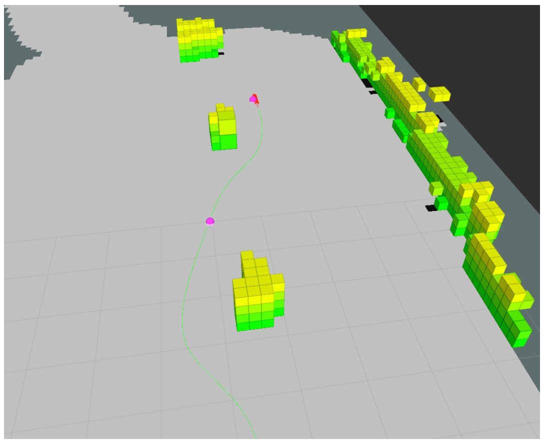
Fig. 8. RVIZ visualization of the experiment: yellow and green cubes are obstacles detected by the depth sensor; two magenta hemispheres represent the waypoints; green line represents the trajectory of the robot; zones in dark gray are unexplored region.

complex scenario (e.g. more obstacles) to further test the efficiency of our parallelized CUDA algorithm.

8. CONCLUSION

We have proposed a MPC approach for autonomous navigation using GPU. It is shown that our CUDA program permits extending the capability of MPC compared to more constrained CPU implementations (longer control/prediction horizon, more control candidates, etc). Our application can run successfully in real-time on mobile robot. Moreover, this work permits to evaluate the system performance in more realistic condition using small, low-power-consumption GPU that can be integrated into small embedded systems. Future works include more extensive experiments on mobile robot and other platforms such as mini-UAVs. This application can be further optimized with more advanced embedded GPUs (such as the new Tegra X1) with enhanced capabilities (e.g recursive CUDA kernel call, more efficient unified memory handling).

REFERENCES

Bartlett, R. A., Wachter, A., Biegler, L. T., 2000. Active set vs. interior point strategies for model predictive control. In: American Control Conference. Vol. 6. Chicago, IL, USA, pp. 4229–4233.

Bertrand, S., Marzat, J., Piet-Lahanier, H., Kahn, A., Rochefort, Y., 2014. MPC Strategies for Cooperative Guidance of Autonomous Vehicles. AerospaceLab (8), pp. 1–18.

Bialkowski, J., Karaman, S., Frazzoli, E., Sep. 2011. Massively parallelizing the RRT and the RRT*. In: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). St Louis MO, USA, pp. 3513–3518.

Constantinides, G. A., 2009. Tutorial paper: Parallel architectures for model predictive control. In: IEEE European Control Conference. Budapest, Hungary, pp. 138–143.

Findeisen, R., Imsland, L., Allgower, F., Foss, B. A., Jan. 2003. State and Output Feedback Nonlinear Model Predictive Control: An Overview. European Journal of Control 9 (2), 190–206.

Frew, E., 2005. Receding Horizon Control Using Random Search for UAV Navigation with Passive, Non-Cooperative Sensing. In: AIAA Guidance, Navigation, and Control Conference and Exhibit. Portland OR, USA.

Gade-Nielsen, N. F., Dammann, B., Jørgensen, J. B., 2014. Interior Point Methods on GPU with application to Model Predictive Control. Ph.D. thesis, Technical University of Denmark.

Gade-Nielsen, N. F., Jørgensen, J. B., Dammann, B., 2012. MPC Toolbox with GPU Accelerated Optimization Algorithms. In: The 10th European Workshop on Advanced Control and Diagnosis (ACD 2012). Lyngby, Denmark.

Hillis, W. D., Steele, Jr., G. L., Dec. 1986. Data Parallel Algorithms. Communications of the ACM 29 (12), 1170–1183.

Huang, Y., Ling, K. V., See, S., 2011. Solving Quadratic Programming Problems on Graphics Processing Unit. ASEAN Engineering Journal.

Ilg, M., Rogers, J., Costello, M., 2011. Projectile Monte-Carlo Trajectory Analysis Using a Graphics Processing Unit. In: AIAA Atmospheric Flight Mechanics Conference. Portland OR, USA.

ISO, 2011. C programming language standard ISO/IEC 9899:201x N1570 Committee draft.

Martinsen, F., Biegler, L. T., Foss, B. A., Dec. 2004. A new optimization algorithm with application to nonlinear MPC. Journal of Process Control 14 (8), 853–865.

NVIDIA, 2015. CUDA C Programming Guide.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A. Y., 2009. ROS: an open-source Robot Operating System. In: ICRA workshop on open source software. Vol. 3. Kobe, Japan, p. 5.

Rogers, J., 2013. GPU-enabled projectile guidance for impact area constraints. Vol. 8752. Baltimore, MD, USA, pp. 87520I 1–23.

Roggeman, H., Marzat, J., Bernard-Brunei, A., Le Besnerais, G., 2016. Prediction of the scene quality for stereo vision-based autonomous navigation. IFAC-PapersOnLine 49 (15), 94–99.

Sengupta, S., Harris, M., Garland, M., Owens, J. D., Jan. 2011. Efficient Parallel Scan Algorithms for Many-core GPUs. In: Kurzak, J., Bader, D. A., Dongarra, J. (Eds.), Scientific Computing with Multicore and Accelerators. Chapman & Hall/CRC Computational Science. Taylor & Francis, pp. 413–442.

Soudbakhsh, D., Annaswamy, A. M., 2013. Parallelized model predictive control. In: IEEE American Control Conference (ACC). pp. 1715–1720.