Implementing Trajectory Optimization on GPUs

Wei Chen

1 Introduction

Trajectory optimization is a powerful method for generating goal-directed trajectory. Differential Dynamic Programming (DDP) is an indirect method which optimizes only over the unconstrained control-space and is therefore fast algorithm. Graphics Processing Unit (GPU) is a modern device for accelerating the algorithm. In this work, we try to implement the trajectory optimization algorithm on GPU.

2 Method

This work currently focuses on solving a pendulum motion problem with indirect shooting method, which specifically is Differential Dynamic Programming algorithm. The algorithm is both implemented on CPU and GPU. In this section, we will go through details of algorithm and regularization method.

2.1 Local Dynamic Programming

We consider a system with discrete-time dynamics. The dynamics is modeled by generic function $\mathbf{f}$

$$\mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) \tag{1}$$

which describes the evolution from time i to $i+1$ of the state $x \in R^n$, given the control $u \in R^m$. A trajectory $\{\mathbf{X}, \mathbf{U}\}$ is a sequence of states $\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1 \ldots, \mathbf{x}_N\}$, and corresponding controls $\mathbf{U} = \{\mathbf{u}_0, \mathbf{u}_1 \ldots, \mathbf{u}_{N-1}\}$.
The total cost denoted by J is the sum of running costs l and final cost l_f, incurred when starting from $\mathbf{x}_0$ and applying $\mathbf{U}$ until the horizon N is reached:

$$J(\mathbf{x}_0, \mathbf{U}) = \sum_{i=0}^{N-1} l(\mathbf{x}_i, \mathbf{u}_i) + l_f(\mathbf{x}_N)$$

The indirect methods represent the trajectory implicitly using only the controls $\mathbf{U}$. The states $\mathbf{X}$ are recovered by integration of (1) from the initial state $\mathbf{x}_0$. The solution of the optimal control problem is the minimizing control sequence

$$\mathbf{U}^* = \underset{\mathbf{U}}{\operatorname{argmin}} J(\mathbf{x}_0, \mathbf{U})$$

Letting $\mathbf{U}_i = \{\mathbf{u}_i, \mathbf{u}_{i+1} \dots, \mathbf{u}_{N-1}\}$ be the tail of the control sequence, the cost-to-go J_i is the partial sum of costs from i to N:

$$J_i(\mathbf{x}_0, \mathbf{U}_i) = \sum_{j=i}^{N-1} l(\mathbf{x}_j, \mathbf{u}_j) + l_f(\mathbf{x}_N)$$

The Value at time i is the optimal cost-to-go starting at $\mathbf{x}$:

$$V_i(x) = \min_{\mathbf{U}_i} J_i(\mathbf{x}_0, \mathbf{U}_i)$$

The Value of the final time is defined as $V_N(x) = l_f(\mathbf{x}_N)$. The Dynamics Programming Principle the reduces the minimization over a sequence of controls $\mathbf{U}_i$, to a sequence of minimizations over a single control, proceeding backwards in time:

$$V(x) = \min_{\mathbf{u}} [l(\mathbf{x}, \mathbf{u}) + V'(\mathbf{f}(\mathbf{x}, \mathbf{u}))] \tag{2}$$

2.2 Quadratic Approximation

DDP involves iterating a forward pass which integrates (1) for a given $\mathbf{U}$, followed by a backward pass which compute a local solution to (2) using a quadratic Taylor expansion. Let $Q(\delta\mathbf{x}, \delta\mathbf{u})$ be the change in the argument of (2) as a function of small perturbations of the i-th nominal $(\mathbf{x}, \mathbf{u})$ pair:

$$Q(\delta\mathbf{x}, \delta\mathbf{u}) = l(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u}) + V'(\mathbf{f}(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u})) \tag{3}$$

The second-order expansion of Q is given by:

$$Q_{\mathbf{x}} = l_{\mathbf{x}} + \mathbf{f}_{\mathbf{x}}^T V'_{\mathbf{x}} \tag{4a}$$
$$Q_{\mathbf{u}} = l_{\mathbf{u}} + \mathbf{f}_{\mathbf{u}}^T V'_{\mathbf{x}} \tag{4b}$$
$$Q_{\mathbf{xx}} = l_{\mathbf{xx}} + \mathbf{f}_{\mathbf{x}}^T V'_{\mathbf{xx}} \mathbf{f}_{\mathbf{x}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{xx}} \tag{4c}$$
$$Q_{\mathbf{ux}} = l_{\mathbf{ux}} + \mathbf{f}_{\mathbf{u}}^T V'_{\mathbf{xx}} \mathbf{f}_{\mathbf{x}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{ux}} \tag{4d}$$
$$Q_{\mathbf{uu}} = l_{\mathbf{uu}} + \mathbf{f}_{\mathbf{u}}^T V'_{\mathbf{xx}} \mathbf{f}_{\mathbf{u}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{uu}} \tag{4e}$$

where the last terms of (4c, 4d, 4e) denote the product of a vector with a tensor. The optimal control modification $\delta\mathbf{u}^*$ for some state perturbation $\delta\mathbf{x}$, is obtained by minimizing the quadratic models:

$$\delta\mathbf{u}^*(\delta\mathbf{x}) = \operatorname*{argmin}_{\delta\mathbf{u}} Q(\delta\mathbf{x}, \delta\mathbf{u}) = \kappa + \mathbf{K}\delta\mathbf{x} \tag{5a}$$

This is a locally-linear feedback policy with

$$\kappa = -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{u}} \qquad\qquad \mathbf{K} = -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}} \tag{5b}$$

the feed-forward modification and feedback gain matrix, respectively. Plugging this policy back into the expansion of Q, a quadratic model of V is obtained. After simplification it is

$$\Delta V = -\frac{1}{2}\kappa^T Q_{uu}\kappa \tag{6a}$$

$$V_{\mathbf{x}} = Q_{\mathbf{x}} - \mathbf{K}^T Q_{uu}\kappa \tag{6b}$$

$$V_{\mathbf{xx}} = Q_{\mathbf{xx}} - \mathbf{K}^T Q_{uu}\mathbf{K} \tag{6c}$$

The backward pass begins by initializing the Value function with the terminal cost and its derivatives $V_N = l_f(\mathbf{x}_N)$, and then recursively computing (5) and (6). Once it is completed, a forward pass computes a new trajectory:

$$\hat{\mathbf{x}}(1) = \mathbf{x}(1) \tag{7a}$$

$$\hat{\mathbf{u}}(i) = \mathbf{u}(i) + \kappa(i) + \mathbf{K}(i)(\hat{\mathbf{x}}(i) - \mathbf{x}(i)) \tag{7b}$$

$$\hat{\mathbf{x}}(i+1) = \mathbf{f}(\hat{\mathbf{x}}(i), \hat{\mathbf{u}}(i)) \tag{7c}$$

2.3 Regularization and Line Search

As in Gauss-Newton approximation, care must be taken when the Hessian is not positive-definite or when the minimum is not close and the quadratic model inaccurate. As described in[1], regularization that penalizes deviations from the states rather than controls is introduced:

$$\tilde{Q}_{ux} = l_{\mathbf{ux}} + \mathbf{f}_{\mathbf{u}}^T(V'_{\mathbf{xx}} + \mu\mathbf{I}_n)\mathbf{f}_{\mathbf{x}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{ux}} \tag{8a}$$

$$\tilde{Q}_{uu} = l_{\mathbf{uu}} + \mathbf{f}_{\mathbf{u}}^T(V'_{\mathbf{xx}} + \mu\mathbf{I}_n)\mathbf{f}_{\mathbf{u}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{uu}} \tag{8b}$$

$$\kappa = -\tilde{Q}_{\mathbf{uu}}^{-1} Q_{\mathbf{u}} \tag{8c}$$

$$\mathbf{K} = -\tilde{Q}_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}} \tag{8d}$$

This regularization amounts to placing a quadratic state-cost around the previous sequence. The feedback gains $\mathbf{K}$ do not vanish as $\mu \to \infty$, but rather force the new trajectory closer to the old one, significantly improving robustness.
The improved Value update is therefore

$$\Delta V = -\frac{1}{2}\kappa^T Q_{\mathbf{uu}}\kappa + \kappa^T Q_{\mathbf{u}} \tag{9a}$$

$$V_{\mathbf{x}} = Q_{\mathbf{x}} + \mathbf{K}^T Q_{\mathbf{uu}}\kappa + \mathbf{K}^T Q_{\mathbf{u}} + Q_{\mathbf{ux}}^T\kappa \tag{9b}$$

$$V_{\mathbf{xx}} = Q_{\mathbf{x}} + \mathbf{K}^T Q_{\mathbf{uu}}\mathbf{K} + \mathbf{K}^T Q_{\mathbf{ux}} + Q_{\mathbf{ux}}^T\mathbf{K} \tag{9c}$$

Also, as described in[1], to overcome the divergence, a backtracking line-search parameter $0 < \alpha \leq 1$ are used to form a new integration method

$$\hat{\mathbf{u}}(i) = \mathbf{u}(i) + \alpha\kappa(i) + \mathbf{K}(i)(\hat{\mathbf{x}}(i) - \mathbf{x}(i)) \tag{10}$$

For $\alpha = 0$ the trajectory would be unchanged, but for intermediate values the resulting control step is not a simple scaled version of the full step, due to the presence of feedback.

We use the expected total-cost reduction in the line-search procedure, but using the improved formula (9a), we can derive a better estimate:

$$\Delta J(\alpha) = \alpha \sum_{i=1}^{N-1} \kappa(i)^T Q_{\mathbf{u}}(i) + \frac{\alpha^2}{2} \sum_{i=1}^{N-1} \kappa(i)^T Q_{\mathbf{uu}}(i)\kappa(i)$$

When comparing the actual and expected reductions

$$z = [J(\mathbf{u}_{1\ldots,N-1}) - J(\hat{\mathbf{u}}_{1\ldots,N-1})]/\Delta J(\alpha)$$

we accept the iteration only if

$$0 < c_1 < z < c_2 < \infty$$

If we are near the minimum we would like μ to quickly go to zero to enjoy fast convergence. If the back-pass fails (a non-PD $\tilde{Q}_{\mathbf{uu}}$), we would like it to increase very rapidly, since the minimum value of μ which prevents divergence is often very large. Finally, if we are in a regime where some $\mu > 0$ is required, we would like to accurately tweak it to be as close as possible to the minimum value, but not smaller. Our solution is to use a quadratic modification schedule. Defining some minimal value μ_{min} (we use $\mu_{min} = 0.01$), maximal value $\mu_{max} = 10^6$ and a minimal modification factor Δ_0 (we use $\Delta_0 = 1.25$), we adjust μ as follows:

$$\begin{aligned} &\textit{increase } \mu: \\ &\qquad \Delta \leftarrow max(\Delta_0, \Delta \cdot \Delta_0) \\ &\qquad \mu \leftarrow max(\mu_{min}, \mu \cdot \Delta) \\ &\textit{decrease } \mu: \\ &\qquad \Delta \leftarrow min(\frac{1}{\Delta_0}, \frac{\Delta}{\Delta_0}) \\ &\qquad \mu \leftarrow max(\mu_{min}, \mu \cdot \Delta) \end{aligned}$$

3 Results

3.1 Dynamics and Model

The example used to verify the function of program is pendulum model. It has two state variables $\mathbf{x} = (x^{(1)}, x^{(2)})$, angle of the pendulum from start position $x^{(1)}$ and the angular velocity of pendulum $x^{(2)}$. There is only one control variable u, which is the tangential force on the tip of pendulum. The dynamic of model is

$$\begin{bmatrix} x^{(1)} \\ x^{(2)} \end{bmatrix} = \begin{bmatrix} x^{(1)} + x^{(2)}\Delta t \\ x^{(2)} + (u + g \cdot sin(x^{(1)}))\Delta t \end{bmatrix}$$

We solved a simple swinging problem by drive pendulum move from downside to upside ($\mathbf{x} = (0, 0)$ to $\mathbf{x} = (\pi, 0)$). And the cost function is

$$J = \frac{1}{2}(x_N - x_g)^T Q_N (x_N - x_g) + \sum_{k=0}^{N-1} \frac{1}{2}(x_k - x_g)^T \mathbf{Q}(x_k - x_g) + \frac{1}{2} u_k^T \mathbf{R} u_k$$

where x_g is the terminal state, $\mathbf{R}$ and $\mathbf{Q}$ are diagonal matrix for running cost. Q_N is used for calculate terminal cost. For the pendulum problem, we set $Q_{1,1} = 1$ and $Q_{2,2} = 0.1$, $Q_N = 1000$. We solved over a 4 second trajectory with 128 steps, Euler integrator (first-order Runge-Kutta integrator).

3.2 Implement on GPUs

Notice that for the iLQR method need to calculate cost respect to the states and control, also the line search method requires performing forward pass couple of times to find out suitable line search parameter α. However, these process can be done in parallel to decrease the time cost.
In order to make program run on GPU, codes are wrapped as "kernel" which is called from CPU and runs on GPU. Apart from this, before performing iLQR method, all the matrix calculation methods and data are transferred from CPU to GPU at the certain allocated space. Specifically, forward pass function is wrapped as 32×1 grids with 1×1 blocks kernel function, and function to calculate dynamics is wrapped as 1×1 grid with 128×1 blocks kernel function.
The result is shown in the Fig.1. From the plot we can see that the time cost of program on GPU is still larger than the time cost on CPU. This is caused by data transferring back and forth from CPU to GPU. But the ratio between backward pass and forward pass is closer to 1 than program on CPU, and this means the parallel calculation works.

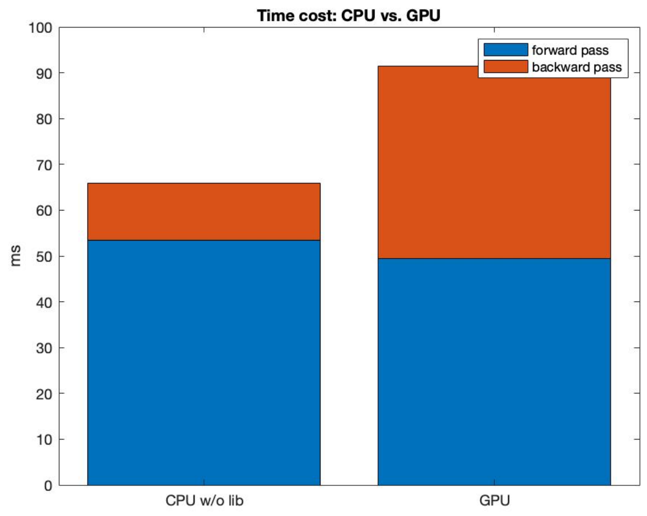

Figure 1: Time cost comparison between program on CPU and GPU

4 Further ideas

From this simple example we can hardly see the advantage of using GPU to implement algorithm in parallel. One reason is there is still a lot of sequential calculation in our method, and we should divide the problem into smaller and independent one. Another reason is the example we used is just a simple case to verify the code, so we should further update the program into better version which can be used to solve more complex problems.
We can use indirect multiple shooting method to further parallelizing the problem. The trajectory can be divided into different segments and at the boundary of each segments the trajectory should be continuous so that $\mathbf{x}(t_i^-) = \mathbf{x}(t_i^+)$. Suppose we have L segments, and each segment has H steps, we can fold this constrain into cost function:

$$J = \frac{1}{2}(x_N - x_g)^T Q_N (x_N - x_g) + \sum_{k=0}^{N-1} \frac{1}{2}(x_k - x_g)^T \mathbf{Q}(x_k - x_g) + \frac{1}{2}u_k^T \mathbf{Q} u_k$$

$$J_{boundary} = \sum_{i=1}^{L} \frac{1}{2}(x_{i\cdot H} - \mathbf{f}(x_{i\cdot H-1}, u_{i\cdot H-1}))^T \mathbf{P}(x_{i\cdot H} - \mathbf{f}(x_{i\cdot H-1}, u_{i\cdot H-1}))$$

$$J_{total} = J + J_{boundary}$$

where $\mathbf{P}$ is cost matrix for boundary constraint.
With this method, we can wrap the forward pass function and backward pass function as 32×1 grids with $L \times 1$ blocks kernel function and 1×1 grids with $L \times 1$ blocks kernel function. And each iteration the expected time to run forward pass and backward pass could possibly decrease as $\frac{1}{L}$ of original time. But in fact every time when the forward pass or backward pass is done, the program should synchronize to wait for all parallel segments finishing calculation. So the time cost should be the time of worst parallel segment.

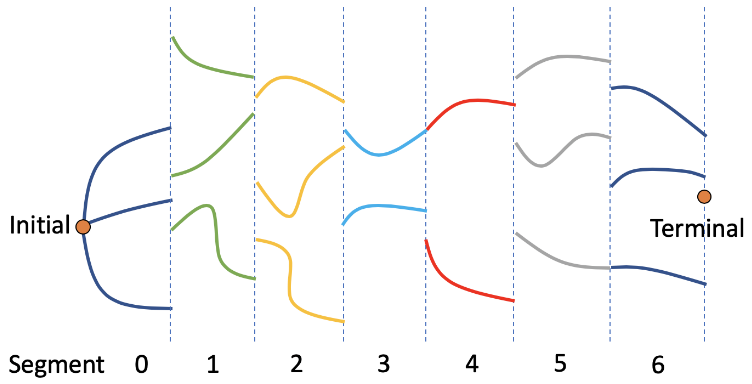

Figure 2: Scheme of trajectory in each segment

There is another approach to deal with the synchronization issue. As illustrated in fig.2, we can store all the segments with different initial states and terminal states in a pool. We choose some reasonable random initial states and perform parallel calculation with same time step for each segments. The program does the calculation for a while, and each time it gets a new segments, it stored the segments in the segments pool then run calculation again with another initial states. Once the segment pool has enough segments, another part of the program selects a trajectory with lowest cost from the segment pool. Then modified initial state of each segment is placed back for new segments calculation. We do this procedure several times until the total cost is barely decrease.

5 Conclusion

So far I tried to implement the iLQR algorithm, and learned related knowledge of trajectory optimization. As a approximation method, iLQR algorithm should be carefully regularized to prevent divergence. I also tried a simple kernel function to wrap the algorithm, and make the algorithm run on GPU. Apart from these, there are still a lot of work to do, such as making trajectory parallel with multiple shooting.