A Performance Analysis of Parallel Differential Dynamic Programming on a GPU

Brian Plancher and Scott Kuindersma

Harvard University, Cambridge MA 02138, USA,
`brian_plancher@g.harvard.edu`, `scottk@seas.harvard.edu`

Abstract. Parallelism can be used to significantly increase the throughput of computationally expensive algorithms. With the widespread adoption of parallel computing platforms such as GPUs, it is natural to consider whether these architectures can benefit robotics researchers interested in solving trajectory optimization problems online. Differential Dynamic Programming (DDP) algorithms have been shown to achieve some of the best timing performance in robotics tasks by making use of optimized dynamics methods and CPU multi-threading. This paper aims to analyze the benefits and tradeoffs of higher degrees of parallelization using a multiple-shooting variant of DDP implemented on a GPU. We describe our implementation strategy and present results demonstrating its performance compared to an equivalent multi-threaded CPU implementation using several benchmark control tasks. Our results suggest that GPU-based solvers can offer increased per-iteration computation time and faster convergence in some cases, but in general tradeoffs exist between convergence behavior and degree of algorithm-level parallelism.

Keywords: optimization and optimal control; motion and path planning; differential dynamic programming; parallel computing; GPU

1 Introduction

The impending end of Moore's Law and the rise of GPU architectures has led to a flurry of research focused on designing algorithms that take advantage of massive parallelism. This trend is prominent in the machine learning literature, and recent work in robotics has demonstrated real-time collision checking for a manipulator using custom voxel grids on an FPGA [1] and fast sample-based planning on a GPU [2,3,4]. For dynamic trajectory optimization, there has been a historical interest in parallel strategies [5] and several more recent efforts lend support to the hypothesis that significant computational benefits are possible [6,7,8,9].

However, there remains much to learn about the algorithmic principles that lead to improvements in dynamic robot control tasks and how large-scale parallel execution of existing algorithms compares to state-of-the-art CPU implementations. This paper aims to add to our understanding of the benefits and limitations of instruction-level and algorithm-level parallelization for a particular family of trajectory optimization algorithms based on Differential Dynamic Programming

(DDP) [10]. We focus on DDP and its variants, particularly the iterative Linear Quadratic Regulator (iLQR) [11], as they have recently received increased attention due to growing evidence that online planning for model predictive control (MPC) is possible for high-dimensional robots [9,12,13,14,15,16].

We describe our parallel implementation strategy on a modern NVIDIA GPU and present results demonstrating its performance compared to an equivalent multi-threaded CPU implementation using benchmark trajectory optimization tasks on a quadrotor and a 7-DoF manipulator. Our results suggest that GPU-based solvers can offer faster convergence than equivalent parallelized CPU implementations in some cases, which could be important for realtime model-predictive control applications, but performance tradeoffs exist between convergence behavior and time per iteration as the degree of algorithm-level parallelism increases.

1.1 Related Work

Prior work on parallel nonlinear optimization has broadly focused on exploiting the natural separability of operations performed by the algorithm to achieve *instruction-level parallelism.* For example, if a series of gradients needs to be computed for a list of static variables, that operation can be shifted from a serial loop over them to a parallel computation across them. Alternatively, if block diagonal matrices must be inverted many times by the solver, each of these instructions can be broken down into a parallel solve of several smaller linear systems. These parallelizations do not change the theoretical properties of the algorithm and therefore can and should be used whenever possible. This research has led to a variety of optimized QP solvers targeting CPUs [17,18], GPUs [19,20], and FGPAs [21,22]. These approaches have also been used to specifically improve the performance of a subclass of QPs that frequently arise in trajectory optimization problems on multi-threaded CPUs [23,24,25,26] and GPUs [27,28]. Additionally, Antony and Grant [8] used GPUs to exploit the inherent parallelism in the "next iteration setup" step of DDP.

In contrast to instruction-level parallelism, *algorithm-level parallelism* changes the underlying algorithm to create more opportunities for simultaneous execution of instructions. In the field of trajectory optimization, this approach was first explored by Bock and Plitt [29] and then Betts and Huffman [5], and has inspired a variety of "multiple shooting" methods [30,31]. Recently, this approach has been used to parallelize both an SQP algorithm [6] and the forward [7,32] and backward passes [9] of the iLQR algorithm. Experimental results from using these parallel iLQR variants on multi-core CPUs represent the current state of the art for real-time robotic motion planning.

Our work aims to add to this literature by systematically comparing the performance of an identical parallel implementation of iLQR on a modern CPU and GPU to (1) better understand the performance implications of various implementation decisions that must be made on parallel architectures and (2) to evaluate the benefits and trade-offs of higher degrees of parallelization (GPU) versus a higher clock rate (CPU).

2 DDP Background

The classical DDP algorithm begins by assuming a discrete-time nonlinear dynamical system, $x_{k+1} = f(x_k, u_k)$, where x $\in \mathbb{R}^n$ is a state and u $\in \mathbb{R}^m$ is a control input. The goal is to find an input trajectory, $U = \{u_0, \ldots, u_{N-1}\}$, that minimizes an additive cost function,

$$J(x_0, U) = \ell_f(x_N) + \sum_{k=0}^{N-1} \ell(x_k, u_k), \tag{1}$$

where $x_1, \ldots, x_N$ are computed by integrating the dynamics from x_0.

Using Bellman's principle of optimality, the *optimal cost-to-go (CTG)*, $V_k(x)$, can be defined by the recurrence relation:

$$V_N(x) = \ell_f(x_N) \qquad V_k(x) = \min_u \ \ell(x,u) + V_{k+1}(f(x,u)). \tag{2}$$

DDP avoids the curse of dimensionality by optimizing over $Q(\delta x, \delta u)$, the second order Taylor expansion of the *local* change in the minimization argument in (2) under perturbations, $\delta x, \delta u$:

$$\begin{aligned} Q(\delta x, \delta u) &= \frac{1}{2} \begin{bmatrix} 1 \\ \delta x \\ \delta u \end{bmatrix}^T \begin{bmatrix} 0 & Q_x^T & Q_u^T \\ Q_x & Q_{xx} & Q_{xu} \\ Q_u & Q_{xu}^T & Q_{uu} \end{bmatrix} \begin{bmatrix} 1 \\ \delta x \\ \delta u \end{bmatrix}, \\ Q_{xx} &= \ell_{xx} + f_x^T V'_{xx} f_x + V'_x \cdot f_{xx}, \qquad Q_{uu} = \ell_{uu} + f_u^T V'_{xx} f_u + V'_x \cdot f_{uu}, \\ Q_{xu} &= \ell_{xu} + f_x^T V'_{xx} f_u + V'_x \cdot f_{xu}, \qquad Q_x = \ell_x + f_x^T V'_x, \qquad Q_u = \ell_u + f_u^T V'_x. \end{aligned} \tag{3}$$

Following the notation used elsewhere [12], a prime is used to indicate the next timestep, and derivatives are denoted with subscripts. The rightmost terms in the equations for Q_{xx}, Q_{uu}, and Q_{xu} involve second derivatives of the dynamics, which are rank-three tensors. These tensor calculations are relatively expensive and are often omitted, resulting in the iLQR algorithm [11].

Minimizing equation (3) with respect to δu results in the following correction:

$$\delta u = -Q_{uu}^{-1}\left(Q_{ux}\delta x + Q_u\right) \equiv K\delta x + \kappa, \tag{4}$$

which consists of an affine term κ and a linear feedback term $K\delta x$. Substituting these terms into equation (3) leads to an updated quadratic model of V:

$$\begin{aligned} V_x &= Q_x - K^T Q_{uu}\kappa - Q_{xu}\kappa - K^T Q_u \\ V_{xx} &= Q_{xx} - K^T Q_{uu} K - Q_{xu} K - K^T Q_{ux}. \end{aligned} \tag{5}$$

Therefore, a backward update pass can be performed starting at the final state and iteratively applying the above computations. A forward simulation pass, using the full nonlinear dynamics, is then performed to compute a new state trajectory using the updated controls. This forward-backward process is repeated until the algorithm converges.

3 Parallelizing DDP

3.1 Instruction-Level Parallelization

Since the cost (1) is additive and depends on each state and control independently, it can be computed in parallel following forward integration and summed in $\log(N)$ operations using a parallel reduction operator. In addition, instead of computing the line search during the forward pass by sequentially reducing α, we can compute all forward simulations for a set of possible α values in parallel. Furthermore, if all simulations are computed in parallel, then the algorithm could select the "best" trajectory across all α values, rather than the first value that results in an improvement. We employ the line search criteria proposed by Tassa [33] and accept an iterate if the ratio of the expected to actual reduction,

$$z = \left(J - \tilde{J}\right)/\delta V^*(\alpha) \quad \text{where} \quad \delta V^*(\alpha) = -\alpha\kappa^T H_u + \frac{\alpha^2}{2}\kappa^T H_{uu}\kappa, \tag{6}$$

falls in the range

$$0 < c_1 < z < c_2 < \infty. \tag{7}$$

Finally, since the dynamics are also defined independently on each state and control pair, the Taylor expansions of the dynamics and cost (the "next iteration setup" step) can occur in parallel following the forward pass. Since this is one of the more expensive steps, this is almost always parallelized in CPU implementations of DDP currently being used for online robotic motion planning (though the number of knot points often exceeds the number of processor cores).

3.2 Algorithm-Level Parallelization

Backward Pass We break the N time steps into M_b equally spaced parallel blocks of size $N_b = N/M_b$. We compute the CTG within each block by passing information serially backwards in time, as is done in standard DDP. After each iteration we pass the information from the beginning of one block to the end of the adjacent block, ensuring that CTG information is at worst case $(M_b - 1)$ iterations stale between the first and last block as shown in Figure 1.

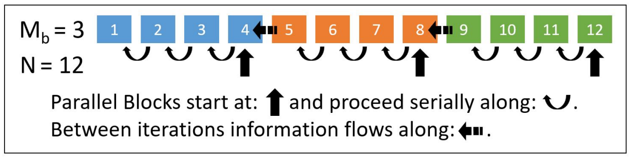
M_b = 3
N = 12
1
2
3
4
5
6
7
8
9
10
11
12
Parallel Blocks start at: and proceed serially along: .
Between iterations information flows along: .

Fig. 1. Graphical representation of the backward pass algorithm-level parallelizations.

Farshidian et al. [9] note that this approach may fail if the trajectory in the next iterate is far enough away from the previous iterate as the stale CTG approximations are defined in relative coordinates and are only valid locally.

Therefore, they propose a linear coordinate transformation of the quadratic CTG approximation at iterate i to re-center it about the current iterate:

$$V_{xx}^{i+1} = V_{xx}^{i}, \qquad V_{x}^{i+1} = V_{x}^{i} + V_{xx}^{i}(x^{i+1} - x^{i}). \tag{8}$$

While this process will still generally converge [34,35], in practice, some forward passes will fail to find a solution because either the CTG information was "too stale," or the new trajectory moved too far from the previous trajectory, rendering the controls at later time steps sub-optimal. Therefore, on the next pass we use the failed iterate's CTG approximation, as it is a less stale estimate, and again follow Tassa [33] and add a state regularization term ρI_n to V'_{xx} in the computation of Q_{uu} and Q_{xu} to stay closer to the last successful trajectory:

$$\begin{aligned} Q_{uu} &= \ell_{uu} + f_u^T \left(V'_{xx} + \rho I_n\right) f_u + V'_x \cdot f_{uu} \\ Q_{xu} &= \ell_{xu} + f_x^T \left(V'_{xx} + \rho I_n\right) f_u + V'_x \cdot f_{xu}. \end{aligned} \tag{9}$$

Forward Pass Giftthaler et al. [7] introduced Gauss-Newton Multiple Shooting, which adapts iLQR for multiple shooting through a fast consensus sweep with linearized dynamics followed by a multiple shooting forward simulation from M_f equally spaced states of $N_f = N/M_f$ time steps as shown in Figure 2.

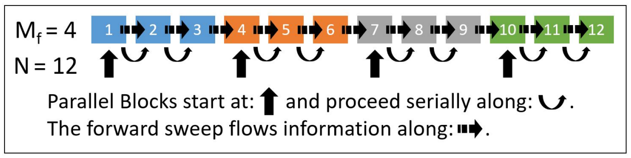
$M_f = 4$
N = 12
1
2
3
4
5
6
7
8
9
10
11
12
Parallel Blocks start at: and proceed serially along: .
The forward sweep flows information along: .

Fig. 2. Graphical representation of the forward pass algorithm-level parallelizations.

This parallel simulation leads to defects d between the edges of each block and changes the one step dynamics to $x_{k+1} = f(x_k, u_k) - d_k$. Incorporating this change into Equations 2-3 results in a modified version of $Q\left(\delta x, \delta u\right)$:

$$Q_x = Q_x + f_x^T V'_{xx} d \qquad Q_u = Q_u + f_u^T V'_{xx} d. \tag{10}$$

We also update our line search criteria to include the following:

$$0 < \max_k ||d_k||_1 < c_3 < \infty, \tag{11}$$

excluding any trajectories that have large defects which represent an artificial mathematical reduction in cost that is infeasible in practice.

Finally, in order to update the start states of each block, a serial consensus sweep is performed by integrating the state trajectory using the previously computed linearized dynamics ($A = f_x$, $B = f_u$) and feedback controls (K, κ), updating state $k+1$ for iterate $i+1$ (using a line search parameter α):

$$x_{k+1}^{i+1} = x_{k+1}^{i} + \left(A_k^i + B_k^i K_k^i\right)\left(x_k^{i+1} - x_k^i\right) + \alpha B_k^i \kappa_k^i + d_k^i. \tag{12}$$

Thus, the forward pass now becomes the serial consensus sweep followed by a parallel forward simulation on each of the M_f blocks.

Parallel DDP (Algorithm 1) combines the instruction-level parallelizations, forward sweep, M_f multiple shooting intervals, and M_b backward pass blocks into a single algorithm with parametrizable levels of parallelism.

4 Implementation

Our implementations were designed to target modern multi-core CPUs and GPUs in order to take advantage of their respective parallel processing power. A multi-core CPU can be roughly viewed as a handful of modern CPUs that are designed to work together, often on different tasks, leveraging the multiple-instruction-multiple-data (MIMD) computing model. In contrast, a GPU is a much larger set of very simple processors, optimized for parallel computations of the same task, leveraging the single-instruction-multiple-data (SIMD) computing model. Therefore, as compared to a CPU processor, each GPU processor has many more arithmetic logic units (ALUs), but reduced control logic and a smaller cache memory (see Figure 3).

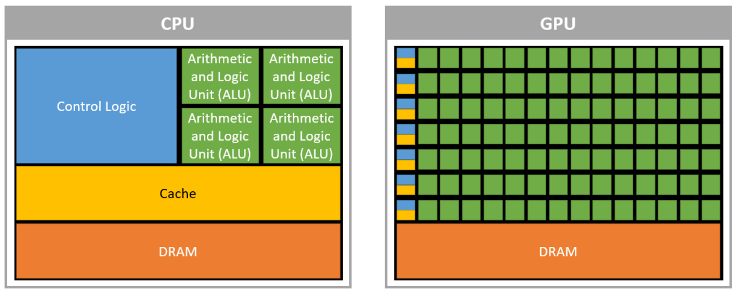
CPU
Control Logic
Arithmetic and Logic Unit (ALU)
Arithmetic and Logic Unit (ALU)
Arithmetic and Logic Unit (ALU)
Arithmetic and Logic Unit (ALU)
Cache
DRAM
GPU
DRAM

Fig. 3. High level architecture of a single CPU and GPU processor [36].

In this work we specifically targeted NVIDIA GPUs by using the CUDA extensions to C++ and the NVCC compiler. CUDA is built around functions called *Kernels* which can be launched using parallel blocks of threads on the GPU. Each block is guaranteed to run all of its threads on the same processor but the order of the blocks is not guaranteed. Each processor's limited cache is split into standard L1 cache memory and *shared memory*, which is managed by the programmer and accessible by all threads in the same block. Each kernel launch will naturally run sequentially but can also be placed in parallel *streams*.

We minimized memory bandwidth delays by doing most computations on the GPU. The CPU is instead in charge of high level serial control flow for kernel launches. We also condense as many computations onto as few kernels as possible, a process known as *kernel fusion* [37], to minimize kernel launch overhead. Finally, we make heavy use of streams and asynchronous memory transfers to increase throughput wherever possible. For example, by computing the Taylor approximations of the dynamics and cost in separate streams, the

Algorithm 1 Parallel DDP

```
 1: Initialize the algorithm and load in initial trajectories
 2: while cost not converged do
 3:     for all $M_b$ blocks $b$ do in parallel
 4:         for $k = b_{N_b} : b_0$ do
 5:             $d_k$, (3), (9), (10) $\rightarrow Q^k$
 6:             if $Q^k_{uu}$ is invertible then
 7:                 (4) $\rightarrow K_k, \kappa_k$
 8:                 (5) $\rightarrow V_k$ and derivatives                    Backward
 9:             else                                                         Pass
10:                 Increase $\rho$ go to line 3
11:             end if
12:         end for
13:     end for
14:     for all $\alpha[i]$ do in parallel
15:         $\tilde{x}_0[i] = x_0$
16:         if $M_f > 1$ then
17:             for $k = 0 : N - 1$ do
18:                 $\tilde{x}_k[i]$, (12) $\rightarrow \tilde{x}_{k+1}[i]$      Consensus
19:             end for                                                  Sweep
20:         end if
21:         for all $M_f$ blocks $b$ do in parallel
22:             for $k = b_0 : b_{N_f} - 1$ do
23:                 $\tilde{u}_k[i] = u_k + \alpha[i]\kappa_k + K_k(\tilde{x}_k[i] - x_k)$
24:                 $\tilde{x}_{k+1}[i] = f(\tilde{x}_k[i], \tilde{u}_k[i])$
25:                 $\tilde{d}_k[i] = 0$
26:             end for
27:             $k = b_{N_f}$                                                          Forward
28:             if $k < N$ then                                                        Pass
29:                 $\tilde{u}_k[i] = u_k + \alpha[i]\kappa_k + K_k(\tilde{x}_k[i] - x_k)$
30:                 $\tilde{d}_k[i] = x_{k+1} - f(\tilde{x}_k[i], \tilde{u}_k[i])$          Forward
31:             end if                                                       Simulation
32:         end for
33:         $\tilde{X}[i], \tilde{U}[i]$, (1), (6) $\rightarrow \tilde{J}[i], \tilde{z}[i]$
34:     end for
35:     $i^* \leftarrow \arg\min_i \tilde{J}[i]$ s.t. $\tilde{z}[i], \tilde{d}[i]$ satisfy (7), (11)
36:     if $i^* \neq \emptyset$ then
37:         $X, U, d \leftarrow \tilde{X}[i^*], \tilde{U}[i^*], \tilde{d}[i^*]$
38:     else
39:         Increase $\rho$ go to line 3
40:     end if
41:     Taylor approximate the cost at $X, U$                                      Next
42:     Taylor approximate the dynamics at $X, U$                                 Iteration
43: end while                                                                      Setup
```

throughput of the next iteration setup step was much closer to the maximum of the running times for those calculations than the sum.

We also found that the general purpose GPU matrix math libraries (e.g., cuBLAS) are optimized for very large matrix operations, while DDP algorithms require many sets of serial small matrix operations. We implemented simpler custom fused kernels which provide a large speedup by keeping the data in shared memory throughout the computations. We further optimized our code by precomputing serial terms during parallel operations. For example, during the backward pass, A, B, K, and κ were loaded into shared memory, so computing $A + BK$ and $B\kappa$ only added a small overhead to the paralellizable backward pass, while greatly reducing the time for the serial consensus sweep.

Our multi-threaded CPU implementation leveraged the `thread` library which supports the launching of parallel threads across CPU cores. Threads run with their own set of registers and stack memory. Like GPU blocks, allocation of CPU threads to processor cores, and context switches between threads running on the same core, are scheduled by the operating system. Also, since these threads run on standard CPUs, there is no explicit management of cache memory.

The CPU implementation reused the same baseline code to leverage the optimizations made during the GPU implementation and to provide an equivalent implementation for comparison. However, for optimal performance, we had to introduce serial loops within threads to limit the number of threads to a small multiple of the number of CPU cores.

We made use of hand derived analytical dynamics and cost functions for the quadrotor system to maximize performance. For the manipulator, we implemented a custom GPU optimized forward dynamics kernel based on the Joint Space Inertia Inversion Algorithm, the fastest parallel forward dynamics algorithm for open kinematic chain robots with a very small number of rigid bodies [38]. For better direct comparisons, we used a looped version of that code for the CPU implementation.[1] Finally, following the state of the art, we implemented the iLQR variant of DDP.

5 Results

Three sets of experiments were conducted to evaluate the performance of parallel iLQR across different problems, computing architectures, and levels of parallelism. We ran our experiments on a laptop with a 2.8GHz quad-core Intel Core i7-7700HQ CPU, a NVIDIA GeForce GTX 1060 GPU, and 16GB of RAM. In our experiments we initialized the algorithm with a gravity compensating input. Both the GPU and CPU implementations used the same scheme for updating ρ and the same set of options for α.

We report convergence results (cost as a function of time and iteration), the time per iteration, and tracking error where appropriate. Total cost reduction as a function of time is a particularly useful metric when deploying algorithms in

[1] We note that while the Joint Space Inertia Inversion Algorithm is not the fastest serial forward dynamics algorithm, the difference is not large for a 7-Dof manipulator.

MPC scenarios where there is typically a fixed control time budget. To ensure our results were representative for each experiment, we ran 100 trials with noise $\sim \mathcal{N}(0,\, \sigma^2)$ applied to the velocities of the initial trajectory. Our solver implementations and these examples can be found at `http://bit.ly/ParallelDDP`.

5.1 Quadrotor

We first considered a quadrotor system with 4 inputs corresponding to the thrust of each rotor and 12 states corresponding to the position and Euler angles, along with their time derivatives. We solved a simple flight task from a stable hover 0.5 m above the origin to a stable hover at the same height and at 7 m in the x and 10 m in the y direction. We used a quadratic cost function of the form:

$$J = \frac{1}{2}(x_N - x_g)^T Q_N (x_N - x_g) + \sum_{k=0}^{N-1} \frac{1}{2}(x_k - x_g)^T Q (x_k - x_g) + \frac{1}{2} u_k^T R u_k, \quad (13)$$

setting $Q = \texttt{blkdiag}(0.01 \times I_{3x3}, 0.001 \times I_{3x3}, 2.0 \times I_{6x6})$, $R = 5.0 \times I_{4x4}$, $Q_N = 1000 \times I_{12x12}$. We solved over a 4 second trajectory with $N = 128$, $M_F = M_B = M = 1, 2, 4, 8, 16, 32, 64$, a 3rd-order Runge-Kutta integrator, and $\sigma = 0.001$.

Figure 4 reveals that the delayed flow of information due to the algorithm-level parallelizations (stale CTG information, fixed starting state of each simulation block) generally leads to smaller steps and therefore slower cost reduction per iteration. For example, for the CPU implementation, the median line search depth for $M = 1$ was between 0 and 1, while for $M = 4$ it was 5. It also shows that the GPUs ability to run a fully parallel line search, as compared to the CPUs partially parallel approach (due to limited number of hardware cores), allows the GPU to select a better "best line search" option and descend faster while avoiding local minima. For example, for the GPU implementation, the median line search depth for $M = 1$ was also between 0 and 1, while for $M = 4$ it was only 3.[2]

The median time per iteration for each of the parallelization options for both implementations is shown in Figure 5. Our observed per-iteration times of under 3 ms for all GPU and CPU cases are comparable to state-of-the-art reported rates of 5-25 ms [15] on a similar UAV system. These results also match our expectations that the higher clock rate would allow the CPU to compute the serial consensus sweep faster while the GPU is able to leverage its increased number of cores to compute the parallel next iteration setup step faster.

For the CPU implementation, we also observed that parallelization can improve the speed of the backward pass until the number of available cores (4) is saturated at which point performance stagnates. For the forward simulation, we found that slower paths to convergence, and thus deeper line searches, quickly outweighed the running time gains due to parallelism. This is most evident in

[2] These trends were also mirrored in the success rate of the algorithm. While 0% of CPU and GPU runs failed for $M = 1, 2, 4$, and on the GPU only 5% failed for $M \geq 8$, on the CPU over 30% failed for $M \geq 8$.

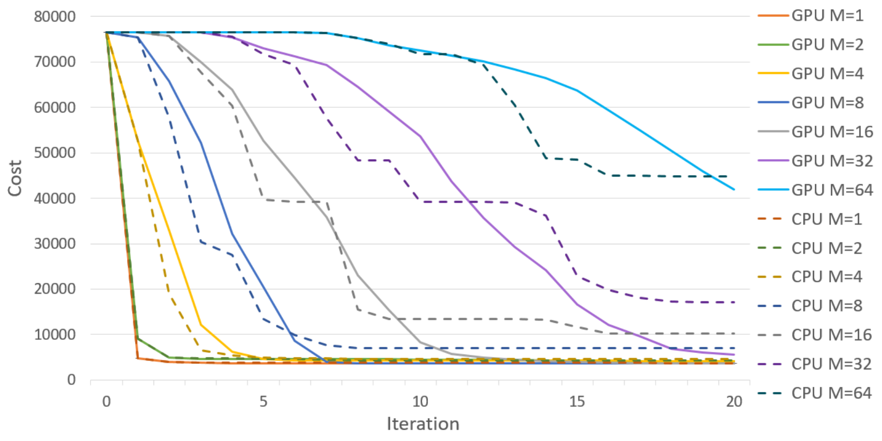

Fig. 4. Median cost vs iteration for the quadrotor experiment.

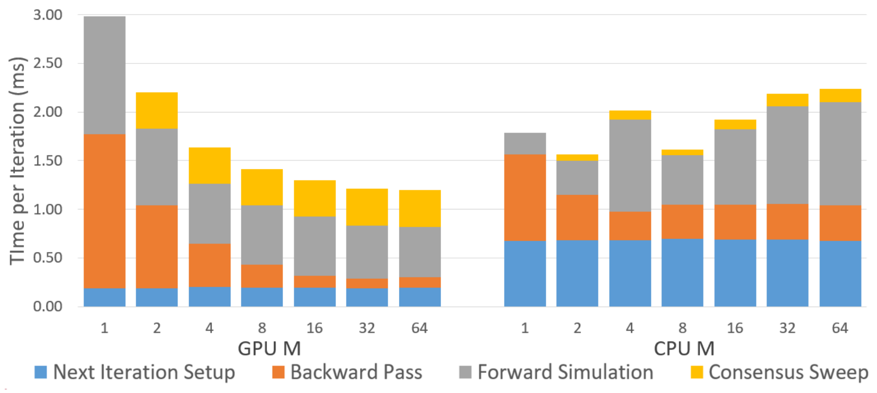

Fig. 5. Median time per iteration for the quadrotor experiment.

the increase in the time for the forward simulation as M grows from 8 to 64. By contrast, the GPU implementation is able to run the line search fully in parallel, which led to reductions in running time for both the backward pass and the forward simulation as M increases. However, there are diminishing returns. First, kernel launch overhead begins to dominate the running time as parallelization is increased. Second, since the next iteration setup is always fully parallelized, and for each line search option the consensus sweep cannot be parallelized, the running times for both steps remain constant. In fact, by the $M = 64$ case, the consensus sweep was almost a third of the total computational time as compared to only 17 percent for the $M = 2$ case.

This increased speed per iteration and decreased convergence rate leads to a level of parallelism which optimizes the time to convergence, as shown in Figure 6. There we find that in the $M = 1$ and $M = 2$ cases, the CPU is able to leverage its higher clock rate to outperform the GPU. However, the GPU is able to better exploit the algorithm-level parallelism and outperform the CPU for $M > 2$. For this experiment the dynamics computations required are simple enough, and the problem size is small enough, that the fastest approach is CPU $M = 1$ indicating that on simple problems the overhead from parallelism outweighs the gains.

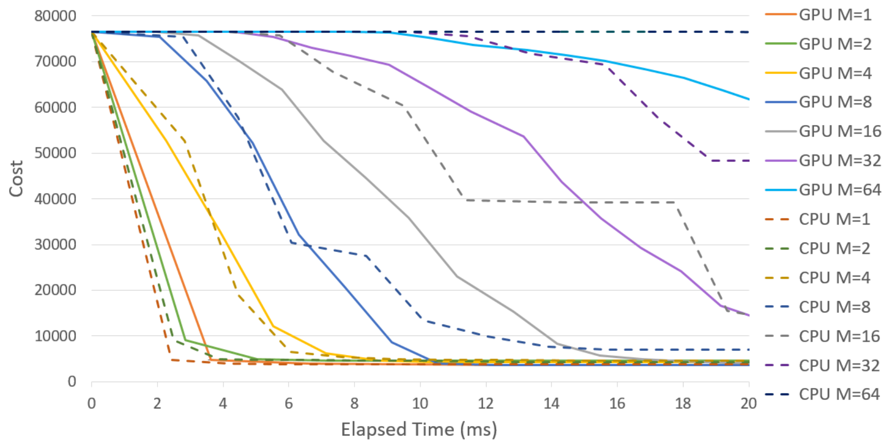

Fig. 6. Median cost for the first 20 milliseconds of the quadrotor experiment.

5.2 Manipulator

We then consider the Kuka LBR IIWA-14 manipulator which has 7 inputs corresponding to torques on the 7 joints. The nominal configuration is defined as the manipulator pointing straight up in the air. We solved a trajectory optimization task from a start state to a goal state across the workspace depicted in Figure 7. We set $Q = \texttt{blkdiag}(0.01 \times I_{7x7}, 0.001 \times I_{7x7})$, $R = 0.001 \times I_{7x7}$, $Q_N = 1000 \times I_{14x14}$. We solved the problem over a 0.5 second trajectory with $N = 64$, $M_F = M_B = M = 1, 2, 4, 8, 16, 32$, a 1st-order Euler integrator, and $\sigma = 0.001$.

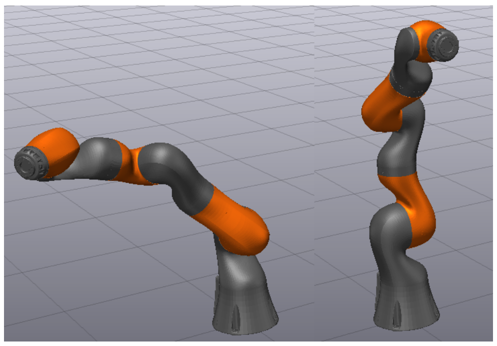

Fig. 7. Start (left) and goal (right) states for the manipulator experiment.

Figure 8 shows that parallelism leads to speedups in time per iteration on the GPU and CPU. We see that again on the GPU both the forward simulation and backward pass decrease in time as M increases. On the CPU we again see that the backward pass decreases in time until the CPU runs out of cores at $M = 4$, while the forward simulation time varies non-monotonically for different values of M depending on the parallelization speedup and the depth of line search slowdown. With all cases having a median time per iteration under 5 ms, both our GPU and CPU implementations are able to perform at speeds reported as state-of-the-art [9,12,13,14,15,16].

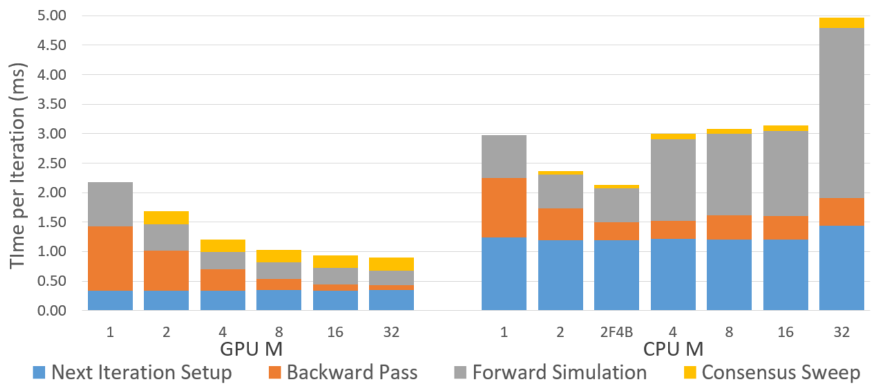

Fig. 8. Median time per iteration for the manipulator experiment.

Unlike in the quadrotor example, the more computationally expensive forward dynamics and increased problem size in this example led to performance gains from parallelism as shown in Figure 9. We find that the GPU is able to successfully exploit the algorithm-level parallelism with faster convergence from $M = 2, 4, 8, 16$ than $M = 1$ and that $M = 32$ on the GPU converges in about the same time as the CPU's fastest standard option, $M = 2$. We also tested various combinations of the number of blocks for the forward and backward passes until we found the best possible CPU variant for this problem ($M_f = 2$ and $M_b = 4$) and found that GPU $M = 2, 4, 8, 16$ still converge faster.

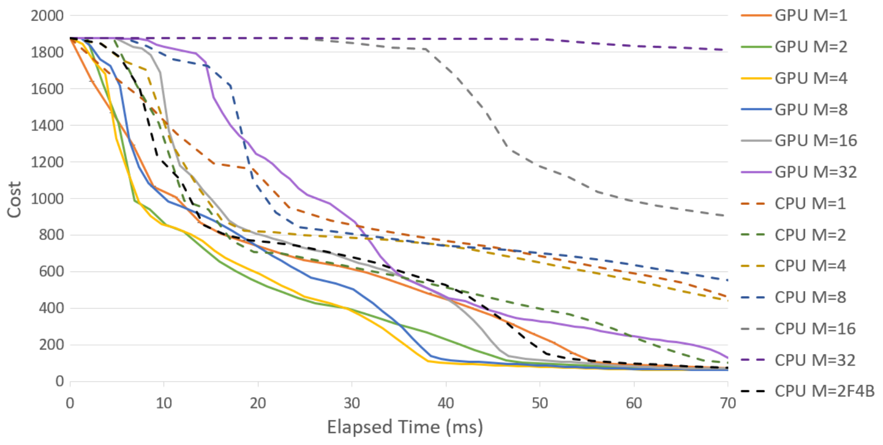

Fig. 9. Median cost for the first 70 milliseconds of the manipulator experiment.

Taken together, the results from the quadrotor and manipulator suggest that practical performance improvements can be obtained through large-scale parallelization and GPU implementations, but that the tradeoffs between the degree of parallelism and convergence speed are strongly dependent on system dynamics and problem specification.

5.3 Model-Predictive Control for the Manipulator

To better understand these results and the applicability of our GPU implementation for online model-predictive control, we conducted a goal tracking experiment with the manipulator in simulation. At each control step we ran our fastest solver, the GPU $M = 4$ implementation, with a maximum time budget of 10 ms. We warm started the iLQR algorithm by shifting all variables from the previous solve by the control duration and then rolling out a new initial state trajectory starting from the current measured state (with a gravity compensating input in the trailing time steps). During optimization periods, we simulated the system forward in realtime using the previously computed solution.

We considered a end effector pose tracking task where the goal moved continuously along a figure eight path. We modified our cost function to include the end effector error:

$$\begin{aligned} J =& \frac{1}{2}(\mathrm{ee}(q_N) - \mathrm{ee}_{goal})^T Q_N (\mathrm{ee}(q_N) - \mathrm{ee}_{goal}) \ + \frac{1}{2}\dot{q}_N^T \dot{Q}_N \dot{q}_N \\ & \sum_{k=0}^{N-1} \frac{1}{2}(\mathrm{ee}(q_k) - \mathrm{ee}_{goal})^T Q (\mathrm{ee}(q_k) - \mathrm{ee}_{goal}) + \frac{1}{2}\dot{q}_k^T \dot{Q} \dot{q}_k + \frac{1}{2} u_k^T R u_k, \end{aligned} \tag{14}$$

where we include the quadratic penalty on $\dot{q}$ to encourage a stable final position. We set $Q = \texttt{blkdiag}(0.01 \times I_{3x3}, 0 \times I_{3x3})$, $R = 0.0001 \times I_{7x7}$, $Q_N = \texttt{blkdiag}(1000 \times I_{3x3}, 0 \times I_{3x3})$, $\dot{Q} = 0.1 \times I_{7x7}$, $\dot{Q}_N = 10 \times I_{7x7}$. At each control step we solved the problem using a first-order Euler integrator and $N = 64$ knot points over a 0.5 second trajectory horizon. The full figure eight trajectory we were tracking had a period of 10 seconds. To initialize the experiment, we held the first goal pose constant until both $||\mathrm{ee}(q) - \mathrm{ee}_{goal}||_2^2$ and $||\dot{q}||_2^2$ were both less than 0.05 at which point the goal began moving along the figure eight path.

Figure 10 shows the trajectory computed by running the MPC experiment starting from an initial vertical state. Aside from confirming that good tracking performance is possible, we observed that the bookkeeping needed to implement MPC on a GPU does add delays in the control loop. In particular, the shifting of the previous variables and rolling out from the updated starting state takes almost 1.4 ms on average. Since GPU $M = 4$ is able to compute iterations in about 1.2 ms, this MPC initialization step is quite expensive. We expect there are potential avenues for improving this overhead through better software engineering (e.g., by using circular buffers to reduce memory copy operations). We plan to investigate this further in future experiments.

6 Conclusion and Future Work

We presented an analysis of a parallel multiple-shooting iLQR algorithm that achieves state-of-the-art performance on example trajectory optimization and model-predictive control tasks. Our results show how parallelism can be used to increase the convergence speed of DDP algorithms in some situations, but

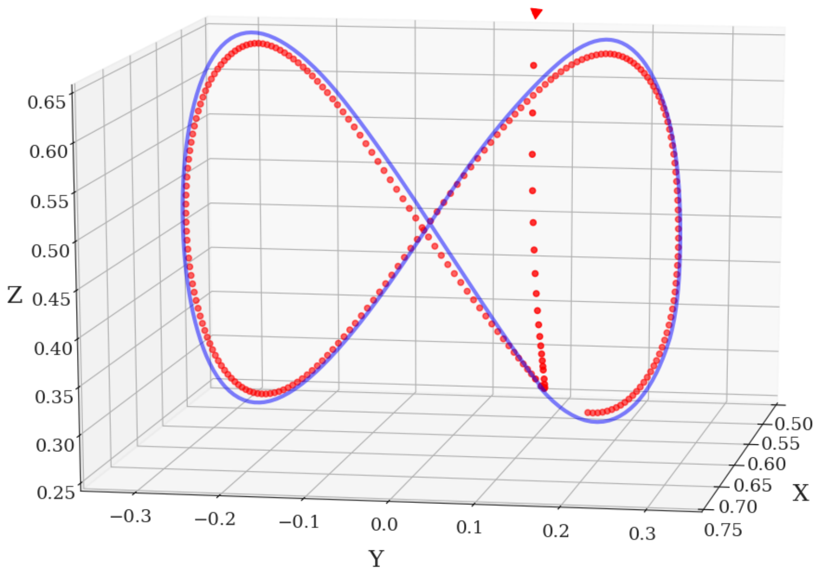

Fig. 10. Executed trajectory (red) vs. goal trajectory (blue) for the MPC experiment.

that tradeoffs between per-iteration speed and convergence behavior manifest in problem-specific ways.

Several directions for future research remain. First, we used analytical and numerical dynamics methods specific to the systems we considered to ensure good performance on the GPU. More general robot dynamics packages designed for GPUs that can achieve a satisfactory level of physical realism would be a valuable tool for the development and evaluation of parallel trajectory optimization methods. Second, it would be interesting to consider the performance impact of adding nonlinear constraints to parallel iLQR using augmented Lagrangian [39] or QP-based methods [40].

Other types of trajectory optimization formulations and algorithms may be more suitable for large-scale parallelization on GPUs, such as direct transcription and solvers based on the alternating direction method of multipliers [41]. We intend to broaden our investigation beyond DDP algorithms in future work. Finally, we would like to evaluate parallel trajectory optimization algorithms for MPC on hardware using low-power mobile parallel compute platforms such as the NVIDIA Jetson.

Acknowledgments

This work was supported by a Draper Internal Research and Development grant and by the National Science Foundation Graduate Research Fellowship (under grant DGE1745303). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding organizations. The authors would like to thank the members of the Harvard Agile Robotics Lab for their useful feedback and insights.

References

1. Sean Murray, Will Floyd-Jones, Ying Qi, Daniel J. Sorin, and George Konidaris. Robot Motion Planning on a Chip. In *Robotics: Science and Systems.*
2. C. Park, J. Pan, and D. Manocha. Real-time optimization-based planning in dynamic environments using GPUs. In *2013 IEEE International Conference on Robotics and Automation*, pages 4090–4097.
3. S. Heinrich, A. Zoufahl, and R. Rojas. Real-time trajectory optimization under motion uncertainty using a GPU. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3572–3577.
4. B. Ichter, E. Schmerling, A. a Agha-mohammadi, and M. Pavone. Real-time stochastic kinodynamic motion planning via multiobjective search on GPUs. In *2017 IEEE International Conference on Robotics and Automation.*
5. John T. Betts and William P. Huffman. Trajectory optimization on a parallel processor. 14(2):431–439.
6. Dimitris Kouzoupis, Rien Quirynen, Boris Houska, and Moritz Diehl. A Block Based ALADIN Scheme for Highly Parallelizable Direct Optimal Control. In *Proceedings of the American Control Conference.*
7. Markus Giftthaler, Michael Neunert, Markus Stuble, Jonas Buchli, and Moritz Diehl. A Family of Iterative Gauss-Newton Shooting Methods for Nonlinear Optimal Control.
8. Thomas Antony and Michael J. Grant. Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures. 54(5):1081–1091.
9. F. Farshidian, E. Jelavic, A. Satapathy, M. Giftthaler, and J. Buchli. Real-time motion planning of legged robots: A model predictive control approach. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics.*
10. D. H. Jacobson and D. Q. Mayne. *Differential Dynamic Programming.* Elsevier.
11. Weiwie Li and Emanuel Todorov. Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems. In *Proceedings of the 1st International Conference on Informatics in Control, Automation and Robotics.*
12. Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and Stabilization of Complex Behaviors through Online Trajectory Optimization. In *IEEE/RSJ International Conference on Intelligent Robots and Systems.*
13. T. Erez, K. Lowrey, Y. Tassa, V. Kumar, S. Kolev, and E. Todorov. An integrated system for real-time model predictive control of humanoid robots. In *2013 13th IEEE-RAS International Conference on Humanoid Robots.*
14. Jonas Koenemann, Andrea Del Prete, Yuval Tassa, Emanuel Todorov, Olivier Stasse, Maren Bennewitz, and Nicolas Mansard. Whole-body Model-Predictive Control applied to the HRP-2 Humanoid. In *Proceedings of the IEEERAS Conference on Intelligent Robots.*
15. M. Neunert, C. de Crousaz, F. Furrer, M. Kamel, F. Farshidian, R. Siegwart, and J. Buchli. Fast nonlinear Model Predictive Control for unified trajectory optimization and tracking. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1398–1404.
16. M. Neunert, F. Farshidian, A. W. Winkler, and J. Buchli. Trajectory Optimization Through Contacts and Automatic Gait Discovery for Quadrupeds. 2(3):1502–1509.
17. D.B. Leineweber. Efficient reduced SQP methods for the optimization of chemical processes described by large sparse DAE models.
18. Daniel P. Word, Jia Kang, Johan Akesson, and Carl D. Laird. Efficient Parallel Solution of Large-scale Nonlinear Dynamic Optimization Problems. 59(3):667–688.

19. Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schroder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 917–924. ACM.
20. Leiming Yu, Abraham Goldsmith, and Stefano Di Cairano. Efficient Convex Optimization on GPUs for Embedded Model Predictive Control. In *Proceedings of the General Purpose GPUs*, GPGPU-10, pages 12–21. ACM.
21. K. V. Ling, S. P. Yue, and J. M. Maciejowski. A FPGA implementation of model predictive control. In *2006 American Control Conference*.
22. M. S. K. Lau, S. P. Yue, K. V. Ling, and J. M. Maciejowski. A comparison of interior point and active set methods for FPGA implementation of model predictive control. In *2009 European Control Conference (ECC)*, pages 156–161.
23. H J Ferreau, A Kozma, and M Diehl. A Parallel Active-Set Strategy to Solve Sparse Parametric Quadratic Programs arising in MPC. 45(17):74–79.
24. Janick V. Frasch, Sebastian Sager, and Moritz Diehl. A parallel quadratic programming method for dynamic optimization problems. 7(3):289–329.
25. Gianluca Frison. Algorithms and Methods for Fast Model Predictive Control.
26. Rien Quirynen. Numerical simulation methods for embedded optimization.
27. Yunlong Huang, K. V. Ling, and Simon See. Solving Quadratic Programming Problems on Graphics Processing Unit.
28. Duc-Kien Phung, Bruno Hriss, Julien Marzat, and Sylvain Bertrand. Model Predictive Control for Autonomous Navigation Using Embedded Graphics Processing Unit. 50(1):11883–11888.
29. Hans Georg Bock and Karl-Josef Plitt. A multiple shooting algorithm for direct solution of optimal control problems. 17(2):1603–1608.
30. David Marcelo Garza. Application of automatic differentiation to trajectory optimization via direct multiple shooting.
31. M. Diehl, H. G. Bock, H. Diedam, and P.-B. Wieber. Fast Direct Multiple Shooting Algorithms for Optimal Robot Control. In *Fast Motions in Biomechanics and Robotics*, pages 65–93. Springer, Berlin, Heidelberg.
32. Etienne Pellegrini and Ryan P Russell. A Multiple-Shooting Differential Dynamic Programming Algorithm. In *AAS/AIAA Space Flight Mechanics Meeting*.
33. Yuval Tassa. Theory and Implementation of Biomimetic Motor Controllers.
34. Martin Zinkevich, John Langford, and Alex J. Smola. Slow Learners are Fast. In *Advances in Neural Information Processing Systems 22*, pages 2331–2339. Curran Associates, Inc.
35. Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. 2013.
36. NVIDIA. *NVIDIA CUDA C Programming Guide*. Version 9.1 edition.
37. J. Filipovi, M. Madzin, J. Fousek, and L. Matyska. Optimizing CUDA Code By Kernel Fusion—Application on BLAS. 71(10):3934–3957.
38. Yajue Yang, Yuanqing Wu, and Jia Pan. Parallel Dynamics Computation Using Prefix Sum Operations. 2(3):1296–1303.
39. Brian Plancher, Zachary Manchester, and Scott Kuindersma. Constrained Unscented Dynamic Programming. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
40. Z. Xie, C. K. Liu, and K. Hauser. Differential dynamic programming with nonlinear constraints. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 695–702.
41. Stephen Boyd. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. 3(1):1–122.