Supplementary Material for Highly Parallelized Data-driven MPC for Minimal Intervention Shared Control

Alexander Broad*‡, Todd Murphey†, Brenna Argall*†‡
*Department of Computer Science
†Department of Mechanical Engineering
Northwestern University, Evanston, IL 60208
Email: alex.broad@u.northwestern.edu
‡Shirley Ryan AbilityLab, Chicago, IL 60611

I. Implementation Details

Sampling-based optimal control algorithms are a classic example of "embarrassingly parallel" computation. To compute an optimal control strategy these approaches integrate information from a large number of sampled trajectories that are generated completely independently of one another. Importantly, the expected optimality of the solution generated through these techniques increases with the number of trajectories considered. For this reason, massively parallelized computer architectures (e.g., GPUs and FPGAs) are a natural choice to improve the speed and efficacy of sampling-based optimal control algorithms for real-time applications. We begin this section with a brief introduction to how GPU architectures differ from standard CPU architectures (see Section I-A). We then describe what portions of our algorithm are parallelizable and provide a visualization for the aid of the reader (see Section I-B). Next, we discuss important features in the scalability of our algorithm (see Section I-C). We then describe alternative sampling strategies and detail how one could embed a model the human-in-the-loop in the underlying distribution (see Section I-D). Finally, we conclude with a discussion of how one could extend our system-idenfication algorithm to devices that exhibit hybrid dynamics (see Section I-E).

A. CPU vs GPU Architectures

Central processing units (CPUs) are designed for serial computation that requires potentially complex control logic, while graphics processing units (GPUs) are designed to perform highly parallel multi-threaded computation. From a hardware design perspective this means that CPUs have larger control blocks, larger on-board cache and a smaller number of arithmetic logic units (ALUs). In contrast, GPUs are designed with smaller control blocks, less on-board memory and a significantly larger number of ALUs (see Fig. 1).

GPUs are therefore well suited to solve arithmetic-heavy computations in data-parallel scenarios such as we find in the algorithm described in this paper. In the next section, we describe key implementation details we consider to properly take advantage of the GPU hardware and improve the speed of our shared control paradigm.

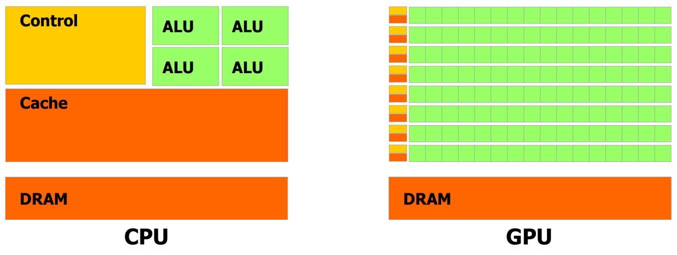

Fig. 1: Pictorial representation of generic CPU and GPU architectures [1].

B. GPU Implementation of MPMI-SC

In this section, we focus our description on how we achieve *instruction-level parallelism*. Parallel computation devices can also be used to speed up calculations through *algorithm-level parallelism*, however, we note that our shared control paradigm is already highly-parallelized at this level as it relies on sampling-based techniques. To detail our approach, we first reproduce our shared control paradigm in Algorithm 1.

The important insight from Algorithm 1 is that the *for loop* on Line 3 is performed in parallel. That is, each control sample (from a uniform distribution) is used to generate a potential trajectory that the human partner may wish to take. In this first study, z_i is set to 0 for each trajectory. Each resulting trajectory is therefore generated independently of the other trajectories and is done so in parallel on the GPU. Here, we note that the main part of the MPMI-SC algorithm happens inside this initial for loop, suggesting that the vast majority of the computation can be parallelized.

To implement this algorithm on a GPU, we rely on NVIDIA's CUDA API [1]. This platform refers to functions as *compute kernels* which are launched in parallel blocks of threads on an NVIDIA GPU. The order in which a given thread runs on a block is not guaranteed, but CUDA supports syncing functions to ensure that all threads have

Algorithm 1 MPMI-SC

```
 1: procedure MPMI-SC($t, x_t, u_h$)
 2:     $\xi \sim \mathbb{U}^{MxN}$                              ▷ unbiased control samples
 3:     for i in N in parallel do                         ▷ forward predict system
 4:         $t_p, x_p, safe \leftarrow t, x_t,$ True
 5:         while $t_p < t + T$ and $safe$ do                      ▷ prediction
 6:             $x_p \leftarrow f(x_p, \xi(i)) + z_i$       ▷ $z_i$ is Gaussian noise
 7:             $safe =$ isSafe($x_p$)                  ▷ system safe at $x_p$
 8:             $t_p = t_p + \Delta t$                        ▷ $\Delta t$ is the timestep
 9:         end while
10:         if $safe =$ True then               ▷ safe over full trajectory
11:             store $\leftarrow \xi(i)$
12:         end if
13:     end for
14:     $u_r \leftarrow$ argmin($cost(\xi(i), u_h)$) $\forall\, \xi(i) \in$ store
15:     return $u_r$              ▷ signal is safe and adheres to MIP
16: end procedure
```

finished running before moving on to additional computation. To improve the instruction-level parallelism of our code, we follow the principles collected by Plancher and Kuindersma [5]. In particular, key implementations features are:

- minimize memory bandwidth delays by reducing the number of cross-device copies,
- minimize kernel launches through *kernel fusion* [2], and
- use *shared memory* when possible to allow fast data access by all threads in a single block.

In contrast to Plancher and Kuindersma [5], we found the cuBLAS optimizations beneficial during the forward prediction step of our algorithm. This relates directly to our choice of system modeling technique—the Koopman operator [4]. Unlike related work that uses complex representations (e.g., Neural Networks) to perform sampling-based optimal control [7], the Koopman operator can be represented as a single matrix and can therefore forward predict the motion of a system through highly optimized matrix-multiplications. Additionally, we note that selecting the block size and number of blocks has a large impact on the efficiency of the algorithm. We use a principled formula to select the optimal number of blocks based on the block size and the number of samples (see code : https://github.com/asbroad/mpmi_shared_control).

Figure 2 is the data flow diagram used to guide our implementation and it highlights the principles defined above. Again, following the structure described by [5], we use the CPU as a high-level controller to ensure correct serial processing of the data and minimize the number of kernel calls to reduce overhead. Notably there are only two copy operations per iteration—once at the beginning and once at the end—to maximize throughput. Within the main for loop, there is a non-parallelizable **while loop** used to forward predict the state of the system. This computation must be done in sequence as the state of the system at each time-step depends on the prior state. Importantly, however, this computation can still be carried out on the GPU. Additionally, it only requires a single kernel call per time-step and zero memory copies. To ensure fast safety checks on the GPU, all relevant environmental information is loaded into *shared memory* so all parallel threads have access to the information. A final important implementation detail is that we re-project (on the GPU) the state into the Koopman space after each prediction iteration.

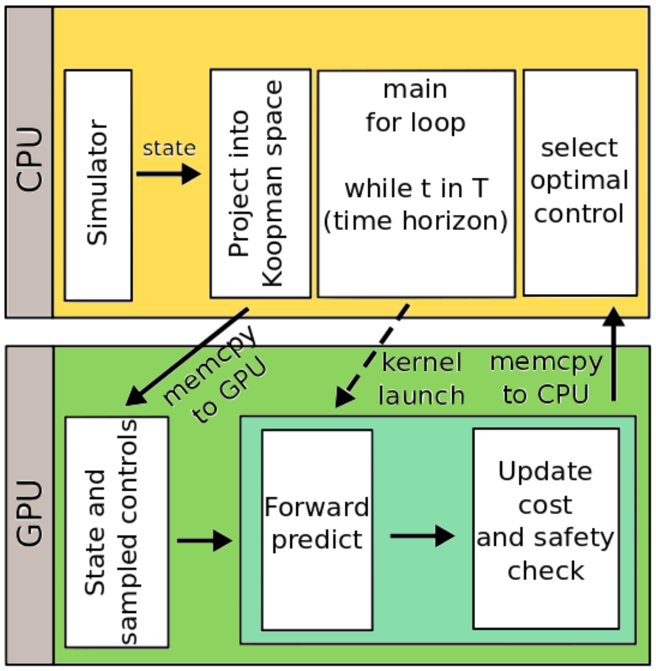
CPU
Simulator
state
Project into Koopman space
main for loop while t in T (time horizon)
select optimal control
memcpy to GPU
kernel launch
memcpy to CPU
GPU
State and sampled controls
Forward predict
Update cost and safety check

Fig. 2: Pictorial representation of the data flow of our algorithm on a CPU and GPU. This data flow can be observed in the main callback (i.e., **mpmi_cb**) in the two main pieces of code in our supplementary material (i.e., **shared_control_balance_bot.cpp** and **shared_control_race_car.cpp**).

C. Scalability

In this section we discuss the scalability of our proposed algorithm. The main factors we must consider here are:

- the number of control samples,
- the length of the receding horizon, and
- the size of the basis used to project the state into the Koopman space.

The number of control samples defines (1) the set of potential actions the user can choose from, and (2) the number of trajectories we must evaluate for safety. The later computation requires forward predicting the state of the system over a receding horizon. While each trajectory must propagate sequentially, the *set of trajectories* can be computed independently in parallel on a GPU. The limiting factor therefore is the number of blocks and threads on the particular GPU.

The length of the receding horizon also has a direct impact on the run-time of our algorithm. This computation must be performed in serial for each individual control sample. Serial computation is less efficient on a GPU then on a standard

CPU [1] which means that shorter time-horizons result in faster overall run-times. However, time-horizons that are too short will result in collision predictions that are not computed early enough to be useful when intervening to improve safety. The limiting factor here is the clockrate of the particular GPU.

Finally, the size of the basis used to project the state into the Koopman space also impacts the efficiency of our algorithm. The main effect of this variable is on the efficiency of the forward prediction computation which requires a series of matrix multiplications. It takes between $\mathcal{O}(n^{2.4}) - \mathcal{O}(n^3)$ to multiple two n x n matrices, depending on the matrix multiplication algorithm used. These computations can be done quicker on a GPU than on a CPU [6], but again, a larger basis (and therefore larger matrices) require more powerful GPU hardware. In this work we used an nVidia GeForce GTX 860M, a low-power 2GB GPU, and therefore note that the speed of this system can be further increased with more powerful hardware.

D. Sampling Strategy

One implementation detail that could be altered based on the goal of the shared control paradigm is the sampling strategy used to generate the trajectory rollouts. For example, if the human partner's desired goal is known *a priori*, or can be predicted, sampling-techniques from stochastic optimal control (e.g., importance sampling) can be used to optimally combine information from all control samples. A benefit of the choice made in this first piece of work is that an equally spaced discretization (or a uniform prior) allows us to provide the tightest bound on the maximum deviation between the user's input and the applied control signal when we have no knowledge of their desired motion. However, if knowledge of the user's behavior (or desired goal) is known, one could define an alternative set (or distribution) from which to generate samples, thereby embedding a model of the user in our MPMI-SC algorithm.

E. System Modeling

One final comment on our implementation relates to the method used for system identification. That is, a main challenge users faced in controlling the race car is that it can enter a *skidding state*. This behavior was not explicitly modeled in our system and therefore not accounted for by the autonomous partner. It is likely we would see further improvements in safety if we (1) explicitly considered the hybrid dynamics of the system [3] or (2) incorporated a cost to penalize the skidding behavior. To ensure safety in the real-world it would be important to explicitly account for these properties when implementing MPMI-SC on new systems.

References

[1] NVIDIA CUDA C Programming Guide. *Nvidia Corporation*, 120(18):8, 2011.

[2] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing CUDA Code by Kernel Fusion: Application on BLAS. *The Journal of Supercomputing*, 71(10):3934–3957, 2015.

[3] Aleksandra Kalinowska, Thomas A Berrueta, Adam Zoss, and Todd Murphey. Data-Driven Gait Segmentation for Walking Assistance in a Lower-Limb Assistive Device. In *International Conference on Robotics and Automation*, 2019.

[4] Bernard O Koopman. Hamiltonian Systems and Transformation in Hilbert space. *Proceedings of the National Academy of Sciences*, 17(5):315–318, 1931.

[5] Brian Plancher and Scott Kuindersma. A Performance Analysis of Parallel Differential Dynamic Programming on a GPU. In *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2018.

[6] Vasily Volkov and James W Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2008.

[7] Grady Williams, Nolan Wagener, Brian Goldfain, Paul Drews, James M Rehg, Byron Boots, and Evangelos A Theodorou. Information Theoretic Mpc for Model-based Reinforcement Learning. In *International Conference on Robotics and Automation (ICRA)*, 2017.