Exploiting GPU/SIMD Architectures for Solving Linear-Quadratic MPC Problems*

David Cole,1 Sungho Shin,2 François Pacaud,2 Victor M. Zavala,1,2 Mihai Anitescu2,3

Abstract—We report numerical results on solving linear-quadratic model predictive control (MPC) problems by exploiting graphics processing units (GPUs). The presented method reduces the MPC problem by eliminating the state variables and applies a condensed-space interior-point method to remove the inequality constraints in the KKT system. The final condensed matrix is positive definite and can be efficiently factorized in parallel on GPU/SIMD architectures. In addition, the size of the condensed matrix depends only on the number of controls in the problem, rendering the method particularly effective when the problem has many states but few inputs and moderate horizon length. Our numerical results for PDE-constrained problems show that the approach is an order of magnitude faster than a standard CPU implementation. We also provide an open-source Julia framework that facilitates modeling (DynamicNLPModels.jl) and solution (MadNLP.jl) of MPC problems on GPUs.

I. INTRODUCTION

The fundamental challenge of model predictive control (MPC) is solving optimal control (dynamic optimization) problems within short sample time intervals. The real-time computation load can be prohibitive, especially when the system dimension is high or the prediction horizon is long; such computational challenges have limited the application scope of MPC. Thus, the scalable solution of optimal control problems has been a long-standing challenge in MPC, with gradual improvements being made to the algorithm [1]–[7].

While numerical solvers have greatly improved thanks to strides in algorithms and computing hardware, the performance of single-core processors has started to stall in the past decade. Rather, progress in hardware has been primarily driven by parallel architectures like multicore processors, distributed computing clusters, and graphics processing units (GPUs). Hence, to continue leveraging advances in modern computing hardware, algorithms and software implementations that harness such capabilities must be developed. While the use of multicore processors and distributed computing has been widely studied in the field of mathematical optimization [8]–[11], relatively fewer contributions have been made regarding the use of GPUs and single instruction, multiple-data (SIMD) architectures. In this context, our work is devoted to accelerating the solution of linear-quadratic MPC problems by exploiting modern GPU/SIMD architectures.

The efficient solution of MPC problems (on either CPUs or GPUs) requires tailored linear algebra techniques that exploit the structure of the Karush–Kuhn–Tucker (KKT) systems arising in the optimization procedure. Classical linear algebra methods include Riccati-like recursions [3], sparse LU factorizations [4], or sparse LDLT factorizations [12]. These methods rely on direct linear solvers that exploit the sparsity or the block tridiagonal structure of the KKT systems. Despite being highly efficient on CPUs, most sparse matrix factorization routines are sequential and known to be difficult to parallelize. Furthermore, when the factorization of the sparse matrix requires a lot of fill-ins (e.g., as in PDE systems [13]), the sparse direct solver can become extremely slow.

An alternative approach to sparse methods is to formulate the problem in a reduced space by eliminating the state variables. This is readily done by writing each state as a function of the initial state and the previous controls [6] (reduction). By further eliminating the inequality constraints using a Schur complement technique (condensation), the Newton step computation can be performed by solving a small (the only variables are controls) positive definite system, which can be factorized efficiently with the Cholesky algorithm. Jerez et al. [6] highlight this method and compare the computational complexity of the step computation within sparse and dense methods. Their complexity analysis suggests that the dense method is particularly effective when the number of states is large, the number of controls is small, and the time horizon is short. We note that the dense formulation has been used in various contexts [5], [14], [15] and is implemented in the state-of-the-art MPC solver HPIPM [16].

All these methods transpose directly in the linear algebra routine employed inside the optimization solver and can be exploited in conjunction with a GPU-accelerated interior-point method (IPM) [4], [17]–[21]. Indeed, in direct contrast to active-set methods (which imply expensive reordering operations in the KKT systems, associated with changes in the active set), IPM involves the solution of a sequence of KKT systems with a fixed sparsity pattern. Hence, the computational burden lies primarily at the linear algebra level (i.e., parallelizing IPM requires solving KKT systems in parallel). To overcome the limitations of sparse direct indefinite factorization routines on GPUs [22], attempts to solve IPMs on GPUs have relied on alternative linear algebra routines such as iterative methods [18], CPU-GPU methods [17], or reduction methods [21]. In the context of MPC, both Gade-Nielsen and co-workers [23], [24] and Lee et al. [4] used GPUs to accelerate some steps of the IPM (e.g., matrix factorization) for linear or linear-quadratic MPC programs.

*)This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) under Contract DE-AC02-06CH11347.

1Department of Chemical and Biological Engineering, University of Wisconsin-Madison, Madison, WI 53706
2Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439
3Department of Statistics, University of Chicago, Chicago, IL 60637

*This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) under Contract DE-AC02-06CH11347.

1Department of Chemical and Biological Engineering, University of Wisconsin-Madison, Madison, WI 53706
2Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439
3Department of Statistics, University of Chicago, Chicago, IL 60637

979-8-3503-2806-6/$31.00 ©2023 AACC
and report subsequent speedups, but neither groups condensed the KKT system as we show herein.

A different approach for implementing optimization algorithms on GPU/SIMD architectures is to use iterative linear solvers. Recently, a new algorithm for general quadratic programs (QPs) based on the operator-splitting technique has been proposed [25]. The technique eliminates the necessity to factorize the KKT system in each iteration and greatly accelerates the solution. The algorithm was subsequently adapted for GPUs by replacing the initial sparse factorization step (difficult to parallelize) with an iterative method (in particular, preconditioned conjugate gradient method) [26].

It has been reported that more than 10 times speedup can be made through GPU accelerations [26].

Our goal is to demonstrate the capability of modern GPU architectures for solving dense linear-quadratic MPC problems. We formulate the problem by applying the reduction method presented in [6] and applying the condensed-space interior-point method with dense Cholesky factorization routines within the CUDA library. To the best of our knowledge, this is the first implementation of condensed-space IPM running on GPU for solving linear-quadratic MPC problems and with the potential to be extended to nonlinear MPC. The dense problem formulation is implemented in the modeling library DynamicNLPModels.jl, and the condensation is implemented in the nonlinear optimization solver MadNLP.jl. The proposed method is demonstrated by using case studies for PDE-constrained optimal control problems. Our results show that solving dense problems using the GPU can offer an order of magnitude speedup.

II. DENSE FORMULATION

We consider the following linear-quadratic MPC problem:

$$\min_{\{x_t\}_{t=0}^{T-1}, \{u_t\}_{t=0}^{T-1}} x_T^T Q_f x_T + \sum_{t=0}^{T-1} u_t^T R u_t$$

s.t. $x_0 = \pi$, $x_{t+1} = A x_t + B u_t + w_t$  $\forall t \in [0, T-1]$, $g^l \leq E x_t + F u_t \leq g^u$  $\forall t \in [0, T-1]$, $x^l \leq x_t \leq x^u$  $\forall t \in [0, T]$, $u^l \leq u_t \leq u^u$  $\forall t \in [0, T-1]$,  (1a)

$$\min_v \frac{1}{2} v^T H v + h^T v + h_0$$

s.t. $Jv \leq d$,  (4a)

where $H$, $J$, $h$, $h_0$, and $d$ are defined implicitly by the elimination procedure.

Observe that the problem is expressed only in terms of $v$. This dense formulation has $3n u_x$ variables and $2T(n_u + n_x + n_u)$ inequality constraints. Thus, the number of variables is significantly reduced, but now the problem has dense Hessian $H$ and Jacobian $J$. By exploiting the controllability, one can partially maintain the sparsity; for more details on the step elimination procedure, readers are pointed to [6].

III. CONDENSED-SPACE INTERIOR-POINT METHOD

We now describe the condensed-space IPM applied to the dense MPC problem in (4). First, we reformulate (4) by introducing the slack variable $s$:

$$\min_{v, s \geq 0} \frac{1}{2} v^T H v + h^T v + h_0$$

s.t. $Jv - d + s = 0$  (5a)

The IPM treats the inequality constraints by replacing the inequality $s \geq 0$ by the log barrier function $-\mu \sum_{i=1}^{m} \log(s_i)$,
where $\mu$ is the barrier term and $s_i$ is the $i$th value of $s$. The resulting barrier subproblem has the form
\[
\min \frac{1}{2} w^T Hv + h^T v + h_0 - \mu \sum_{i=1}^{m} \log(s_i) \quad (6a)
\]
s.t. $Jv - d + s = 0$. \hspace{1cm} (6b)

The resulting KKT conditions are then
\[
Hv + h + J^T \lambda = 0, \quad \lambda - z = 0, \quad Jv - d + s = 0 \quad (7a)
\]
where $z := \mu S^{-1} 1$, $\lambda$ is the Lagrange multiplier of (6b), $S := \text{diag}(s)$, and $I$ is the vector of ones.

Typical IPMs compute the step direction by applying Newton’s method to the primal-dual equation in (7). The step computation involves the solution of linear systems of the following form:
\[
\begin{bmatrix}
H & J^T \\
\Sigma_s & I \\
J & I
\end{bmatrix}
\begin{bmatrix}
p^v \\
p^s \\
p^\lambda
\end{bmatrix} =
\begin{bmatrix}
r_1 \\
r_2 \\
r_3
\end{bmatrix}
\]
where $p^v$, $p^s$, and $p^\lambda$ are the descent directions for $v$, $s$, and $z$, respectively; $r_1 := Hv + h + J^T \lambda$, $r_2 := \lambda - \mu S^{-1} 1$, $r_3 := Jv - d + s$; and $\Sigma_s := S^{-1} Z$ with $Z := \text{diag}(z)$. Note that $S$ must be invertible (always the case in IPMs since the primal iterate stays in the strict interior $s > 0$).

This KKT system can be further condensed by writing $p^\lambda$ as a function of $p^s$ and $p^v$ as a function of $p^v$. We then obtain a system of equations for $p^v$:
\[
(H + J^T \Sigma_s J) p^v = -r_1 + Jr_2 - J^T \Sigma_s r_3. \quad (9)
\]

Here, $H + J^T \Sigma_s J$ is positive definite and can be factorized with the Cholesky factorization, and $p^v$ can be computed by back triangular solve. Once $p^v$ is obtained, $p^s$ can be obtained from $p^s = -r_3 - Jp^v$, and $p^\lambda$ can be found from $p^\lambda = -r_2 + \Sigma_s r_3 + J p^v$. After the descent directions have been determined, a backtracking line search is performed to determine the step size, and the next iterate is obtained based on the step size and the direction. The new values of $v$, $s$, and $z$ at the end of this process are then used in the next iteration, and the process is continued until a stopping criterion is satisfied. The method is summarized in Algorithm 1; here, the superscript $(k)$ represents the iterate at $k$th iteration.

The system in (9) is (i) smaller ($n_x T$ rows and columns) than the system in (8) ($n_x T + 2(n_c + n_x + n_u) T$ rows and columns), (ii) dense, and (iii) positive definite. These features make it more efficient to perform Algorithm 1 on the GPU or other parallel architecture because the Cholesky factorization of small dense systems can be easily parallelized, and efficient implementation is available within the CUDA library. An illustration of the sparsity pattern of full, reduced, and condensed KKT systems with one-sided bounds on the state variables is shown in Figure 1.



**Algorithm 1** Condensed-space interior-point method

**Require:** $A, B, Q, R, S, K, N, x_0$

**Construct** $H$, $h$, $h_0$, $J$, and $d$

**Initialize** $v^{(0)}$, $s^{(0)}$, and $\lambda^{(0)}$

**while** Termination criteria are not met

**do**

\[
\Sigma_s^{(k)} \leftarrow (S^{(k)})^{-1} Z^{(k)}
\]

**Compute** $H + J^T \Sigma_s^{(k)} J$

**Factorize** $H + J^T \Sigma_s^{(k)} J$ and solve (9) for $p^{v^{(k)}}$.\hspace{1cm} $p^{v^{(k)}} \leftarrow -r_3 - J p^{v^{(k)}}$

\[
p^{\lambda^{(k)}} \leftarrow -r_2 + \Sigma_s r_3 + \Sigma_s J p^{v^{(k)}}
\]

\[
p^{z^{(k)}} \leftarrow -z^{(k)} - \Sigma_s p^{v^{(k)}}
\]

Determine the step length $\alpha$, $\alpha_z$ through line search $u^{(k)} \leftarrow u^{(k)} + \alpha p^{v^{(k)}}$\hspace{1cm} $s^{(k)} \leftarrow s^{(k)} + \alpha p^{v^{(k)}}$

\[
\lambda^{(k)} \leftarrow \lambda^{(k)} + \alpha p^{\lambda^{(k)}}$

\[
z^{(k)} \leftarrow z^{(k)} + \alpha z^{(k)}
\]

**end while**

**Fig. 1.** Full, reduced, and condensed system for linear-quadratic MPC problems for $n_x = 5$, $n_u = 3$, $n_c = 0$, and $T = 5$.

Julia package DynamicNLPModels.jl. This package takes data given in (1) and builds either the sparse or reduced forms of these problems. The package is open source, and the source code can be found at [27]. Further, the model returned by DynamicNLPModels.jl is a subtype of the generic NLP model data structure implemented in NLPMModels.jl [28] and so is compatible with other solvers in the JuliaSmoothOptimizer ecosystem. By building the reduced problem outside the solver, DynamicNLPModels.jl facilitates forming the reduced KKT system (8), which can then be further condensed (9) as the system in (9) within MadNLP.jl.

DynamicNLPModels.jl allows the user to easily define either the sparse (Equation (1)) or reduced (Equation (4)) linear-quadratic MPC problem by passing user-defined data. An example of how these models are built can be seen in the code snippet below.

```julia
using DynamicNLPModels
using Random, LinearAlgebra
Q = 1.5 * Matrix(I, (3, 3))
R = 2.0 * Matrix(I, (2, 2))
```

IV. IMPLEMENTATION

To facilitate building and solving sparse or reduced linear-quadratic MPC problems, we have developed the
We solved problems (i) over a range of \( N \) values at three different numbers of time steps: \( T = 50 \), \( T = 150 \), and \( T = 250 \) and (ii) over a range of \( T \) values for fixed \( N = 4 \). The results can be seen in Figures 2 and 3 and Tables IV–V. In the tables, “iter” is the number of iterations within IPMs reported by the solvers (Ipopt for the sparse problems and MadNLp.jl for the dense problems), “tot” is the total solver time, and “lin” is the linear solver time. The infinite-dimensional problem in (10) is discretized by using a finite difference method in space and an explicit Euler scheme in time to make a finite-dimensional MPC problem of the form in (1). We also treat the boundaries as inputs only (not as states), so all of the state variables are strictly internal temperatures. In addition, this problem can be scaled up by changing \( T \) (the time horizon) or \( N \) (the number of discretization points per dimension). Note that increasing \( N \) increases cubically the number of states because the spatial domain is three-dimensional.

**B. Results**

We solved problems (i) over a range of \( N \) values at three different numbers of time steps: \( T = 50 \), \( T = 150 \), and \( T = 250 \) and (ii) over a range of \( T \) values for fixed \( N = 4 \). The results can be seen in Figures 2 and 3 and Tables IV–V. In the tables, “iter” is the number of iterations within IPMs reported by the solvers (Ipopt for the sparse problems and MadNLp.jl for the dense problems), “tot” is the total solver time, and “lin” is the linear solver time. The infinite-dimensional problem in (10) is discretized by using a finite difference method in space and an explicit Euler scheme in time to make a finite-dimensional MPC problem of the form in (1). We also treat the boundaries as inputs only (not as states), so all of the state variables are strictly internal temperatures. In addition, this problem can be scaled up by changing \( T \) (the time horizon) or \( N \) (the number of discretization points per dimension). Note that increasing \( N \) increases cubically the number of states because the spatial domain is three-dimensional.

**V. NUMERICAL RESULTS**

We apply the condensed-space IPMs to a 3-D PDE temperature control problem to showcase the potential benefits and limitations of our approach. For problems with low numbers of inputs and high numbers of constraints, we show that solving the dense linear-quadratic MPC problem on the GPU using a condensed-space IPM can offer an order of magnitude speedup over solving the dense problem on the CPU with the condensed-space IPM or solving the sparse problem on the CPU. Also, our results suggest that the condensed-space GPU approach is less effective for problems with a low ratio of states to inputs. In all cases, we constructed the model using DynamicNLFLModels.jl. We solved the sparse form of the problem using Ipopt [29] with MA27 [12] as the linear solver. We solved the dense form of the problem on the CPU as well as the GPU using MadNLp.jl and dense Cholesky factorization using LAPACK (OpenBLAS) or CUSOLVER. Scripts and solver outputs are available at [30]. All the numerical results presented have been generated on our workstation, equipped with an Intel Xeon Gold 6140 CPU and an NVIDIA V100 GPU.

**A. Problem Formulation**

We adapted the 2-D thin plate temperature control problem given by [7] to a 3-D temperature control problem in a cube using the boundary inputs. In this problem, each face of the cube can be heated by control to try to reach the desired temperature profile. The continuous form of this problem can be stated as follows:

\[
\begin{align*}
\min_{x,u} & \quad \frac{1}{2} \int_0^T \int_{w \in \Omega} \left( q(x(w,t) - d(w,t))^2 dw \\
& \quad + \sum_{i \in \mathcal{U}} r_u(t_i)^2 \right) dt \\
\text{s.t.} & \quad \rho \Delta_t \frac{\partial x(w,t)}{\partial t} = k \Delta_{xx} x(w,t), \quad w \in \Omega, \quad t \in [0,T] \quad (10b) \\
& \quad x^l \leq x(w,t) \leq x^u, \quad w \in \Omega, \quad t \in [0,T] \quad (10c) \\
& \quad u^l \leq u_i(t) \leq u^u, \quad i \in \mathcal{U}, \quad w \in \Omega, \quad t \in [0,T] \quad (10d) \\
& \quad x(w,t) = u_i(t), \quad i \in \mathcal{U}, \quad w \in \partial \Omega, \quad t \in [0,T] \quad (10e) \\
& \quad x(w,0) = \hat{x}, \quad w \in \Omega, \quad (10f)
\end{align*}
\]

where \( \Omega = [0,L]^3 \subseteq \mathbb{R}^3 \) is the 3-D domain of a cube of length \( L \), \( \mathcal{U} := \{1,2,\cdots,6\} \) is the set of all cube faces, \( x : \Omega \times [0,T] \to \mathbb{R} \) is the temperature, \( d : \Omega \times [0,T] \to \mathbb{R} \) is the set point temperature, \( u_i : [0,T] \to \mathbb{R} \) are the boundary temperatures for \( i \in \mathcal{U} \) and \( \partial \Omega_i \subseteq \Omega \) for \( i \in \mathcal{U} \) are the boundaries. Note that (10e) enforces the Dirichlet boundary condition. The constant values are given in Table I.

| Parameter | Value |
|-----------|-------|
| \( \Delta_t \) | Temporal discretization mesh size | 0.1 sec |
| \( \Delta_w \) | Spatial discretization mesh size | 0.02 m |
| \( L \) | Length of cube face | \((N+1)\Delta_{w}\) |
| \( \rho \) | Density of copper | 8960 kg \cdot m\(^{-3}\) |
| \( C_p \) | Specific heat of copper | 386 J \cdot (kg \cdot K)\(^{-1}\) |
| \( k \) | Thermal conductivity of copper | 400 W \cdot (m \cdot K)\(^{-1}\) |
| \( q \) | Weight on temperature error | 10 \( \Delta_{w}^2 \) m\(^{-2}\) |
| \( r \) | Weight on input | \( \frac{\rho}{m} \Delta_{w}^2 \) m\(^{-2}\) |
| \( (x^l, x^u) \) | (Lower, upper) temperature bounds | (200, 550) K |
| \( (u^l, u^u) \) | (Lower, upper) input bounds | (300, 500) K |
| \( \hat{x} \) | Initial temperature | 300 K |


Table II
Solver Statistics for T = 50 and Varied N.

| N  | CPU Sparse | CPU Dense | GPU Dense |
|----|------------|-----------|-----------|
|    | iter       | tot lin   | iter       | tot lin   | iter       | tot lin   |
| 4  | 26         | 0.6       | 4.5        | 27        | 0.45 0.020 | 27 10.9 0.008 |
| 5  | 28         | 5.0       | 4.4        | 32        | 1.01 0.023 | 32 17.7 0.009 |
| 6  | 27         | 17.5      | 15.5       | 29        | 1.66 0.022 | 29 23.2 0.008 |
| 7  | 27         | 69.1      | 61.4       | 33        | 3.36 0.024 | 33 41.6 0.010 |
| 8  | 32         | 689.0     | 641.1      | 37        | 10.1 0.030 | 37 70.5 0.011 |
| 9  | 30         | 3,100.0   | 2,936.0    | 37        | 7.79 0.033 | 37 94.9 0.011 |
| 10 | -          | -         | -          | 36        | 9.69 0.033 | 36 17.1 0.011 |
| 11 | -          | -         | -          | 37        | 13.9 0.034 | 37 20.0 0.012 |
| 12 | -          | -         | -          | 37        | 18.1 0.034 | 37 24.3 0.012 |
| 13 | -          | -         | -          | 38        | 22.6 0.035 | 38 33.7 0.013 |
| 14 | -          | -         | -          | 38        | 25.6 0.033 | 38 40.8 0.013 |

Table III
Solver Statistics for T = 150 and Varied N.

| N  | CPU Sparse | CPU Dense | GPU Dense |
|----|------------|-----------|-----------|
|    | iter       | tot lin   | iter       | tot lin   | iter       | tot lin   |
| 4  | 25         | 1.7       | 1.4        | 27        | 8.90 0.031 | 27 68.6 0.023 |
| 5  | 29         | 17.2      | 15.4       | 32        | 17.4 0.054 | 32 126.0 0.028 |
| 6  | 29         | 55.9      | 49.9       | 31        | 28.9 0.037 | 31 193.0 0.027 |
| 7  | 30         | 238.2     | 214.1      | 36        | 53.0 0.043 | 36 419.0 0.034 |
| 8  | 31         | 4,953.0   | 4,636.0    | 37        | 155.5 0.045 | 37 612.0 0.034 |
| 9  | 31         | 10,357.0  | 9,703.0    | 38        | 110.5 0.040 | 38 805.0 0.034 |
| 10 | -          | -         | -          | 36        | 153.2 0.039 | 36 108.0 0.034 |
| 11 | -          | -         | -          | 38        | 221.1 0.043 | 38 151.0 0.037 |
| 12 | -          | -         | -          | 38        | 353.7 0.048 | 38 189.0 0.040 |
| 13 | -          | -         | -          | 39        | 404.1 0.050 | 39 252.0 0.045 |
| 14 | -          | -         | -          | 40        | 559.6 0.053 | 40 326.0 0.045 |

Fig. 3. Total solver time for N = 4 and varied T.

of variables in the dense problem remains unchanged (the number of inputs is independent of N), whereas the number of variables of the sparse problem scales cubically in N. Note that problems with rapidly growing numbers of states can easily arise in discretized PDE-constrained control problems. Also, because of the structure of the discretization of 3-D PDEs, the fill-ins in the factorization grow rapidly, and the sparse solver becomes inefficient as N grows. Although the number of variables in the condensed-space approaches does not change, the solution time is moderately affected by N. The reason is that the number of constraints increases with N, making the number of rows in the Jacobian grow linearly with N. While the dense system size is still independent of N, the condensation procedure, which involves computing \( J^T \Sigma J \), is affected by the dimension of \( J \).

In all three cases of changing N, the dense formulation on the GPU was much faster than the same formulation/algorithm running on the CPU. For \( T = 250 \), \( T = 150 \), and \( T = 50 \), the GPU was on average 22, 16, and 7 times faster respectively. It appears that the GPU operates proportionally faster than the CPU on the larger problem sizes, which is not surprising since the parallelizable computations increase with problem size. However, the solution time for the dense formulation increases faster than the sparse solution time for changing T, as can be seen in Figure 3 and Table V. Jerez et al. [6] report that the number of floating point operations (flops) for computing an interior-point iteration of the dense formulation is \( O(T n_u^2) \) while the number of flops for the sparse formulation is \( O(T(n_x + n_u)^2(n_x + n_u + n_c)) \). Thus, even though the number of flops for the sparse formulation goes as \( O(n_u^2) \), the sparse formulation can overtake the dense formulation (on either the CPU or GPU) since the dense formulation goes as \( O(T^3) \) while the sparse formulation goes only as \( O(T) \). This effect can be seen for \( N = 4 \) and \( T = 500 \) (see Figure 3 and Table V).

These results suggest that solving linear-quadratic MPC problems on the GPU using the above methods can reduce the solution times by an order of magnitude for certain cases. This speedup can enable applying MPC to new systems that previously would have been limited by time constraints. The types of problems to which this method can practically be applied are problems with a high ratio of states to inputs and moderate horizon length (e.g., some control systems modeled with PDEs). However, our results also suggest that this framework is less effective (and eventually impractical) as the ratio of states to inputs decreases. For example, in Figure 2 the sparse formulation with \( N = 4 \) (where the sizes between the sparse and dense problems are similar) is comparable to the GPU solution time and much faster than the dense formulation on the CPU. Numerous linear-quadratic MPC problems have much lower ratios of states to inputs than that of the temperature control problem presented here, and these problems would be less practical in this framework.
TABLE V
SOLVER STATISTICS FOR N = 4 AND VARIED T.

| CPU Sparse | CPU Dense | GPU Dense |
|------------|-----------|-----------|
| T          | iter tot lin | iter tot lin | iter tot lin |
| 10         | 26 0.10 0.079 | 28 0.02 0.001 | 28 0.03 0.004 |
| 20         | 25 0.21 0.170 | 27 0.05 0.002 | 27 0.04 0.005 |
| 30         | 24 0.30 0.249 | 26 0.10 0.005 | 27 0.05 0.005 |
| 40         | 25 0.43 0.356 | 27 0.21 0.009 | 27 0.06 0.006 |
| 50         | 26 0.55 0.460 | 27 0.34 0.016 | 27 0.08 0.007 |
| 75         | 27 0.85 0.724 | 29 1.02 0.051 | 29 0.16 0.010 |
| 100        | 25 1.07 0.900 | 27 2.31 0.096 | 27 0.29 0.015 |
| 125        | 25 1.35 1.14  | 27 3.76 0.162 | 27 0.60 0.021 |
| 150        | 25 1.60 1.36  | 27 6.63 0.243 | 27 0.60 0.021 |
| 200        | 26 2.30 1.97  | 28 20.0 0.529 | 28 1.35 0.029 |
| 250        | 26 2.99 2.58  | 28 27.0 0.979 | 28 1.64 0.036 |
| 300        | 28 3.63 3.15  | 30 48.0 1.61  | 30 2.54 0.047 |
| 350        | 27 4.17 3.61  | 29 64.6 2.29  | 29 3.62 0.062 |
| 400        | 27 5.04 4.16  | 29 216.4 3.30 | 29 4.48 0.053 |
| 450        | 27 5.89 5.14  | 29 123.3 4.35 | 29 5.41 0.067 |
| 500        | 27 6.12 5.30  | 29 176.2 5.64 | 29 7.49 0.067 |

VI. CONCLUSIONS AND FUTURE WORK

For problems with large numbers of states, few inputs, and a moderate horizon length, we can speed up linear-quadratic MPC by an order of magnitude by exploiting the structure using GPU/SMID architectures. We introduced the package DynamicNLPModels.jl, which reduces the linear-quadratic MPC problem based on user-defined data. The KKT system of this dense model is a reduced KKT system (as compared with the sparse formulation), which MadNLP.jl automatically constructs when the model is passed from DynamicNLPModels.jl to MadNLP.jl. The KKT system can be further condensed within MadNLP.jl, resulting in a dense positive definite matrix that can be efficiently factorized on the GPU. Based on the example problem of the heating of a cube, this method can result in significant speedup on the GPU.

REFERENCES

[1] S. J. Wright, “Partitioned dynamic programming for optimal control,” SIAM Journal on Optimization, vol. 1, no. 4, pp. 620–642, 1991.
[2] C. V. Rao, S. J. Wright, and J. B. Rawlings, “Application of interior-point methods to model predictive control,” Journal of Optimization Theory and Applications, vol. 99, no. 3, pp. 723–757, 1998.
[3] G. Frison, H. H. B. Sørensen, B. Dammann, and J. B. Jørgensen, “High-performance small-scale solvers for linear model predictive control,” in 2014 European Control Conference (ECC). IEEE, 2014, pp. 128–133.
[4] S. Lee, H. Lee, Y. Kim, and W. Choi, “GPU-accelerated PD-IPM for real-time model predictive control in integrated missile guidance and control systems,” Sensors, vol. 22, no. 12, p. 4512, 2012.
[5] J. A. Anderson, J. V. Frach, M. Vukov, and M. Diehl, “A condensing algorithm for nonlinear MPC with a quadratic runtime in horizon length,” Automatica, pp. 97–100, 2013.
[6] J. L. Jerez, E. C. Kerrigan, and G. A. Constantinides, “A sparse and condensed QP formulation for predictive control of LTI systems,” Automatica, vol. 48, no. 5, pp. 999–1002, 2012.
[7] S. Na, S. Shin, M. Anitescu, and V. M. Zavala, “On the convergence of overlapping Schwarz decomposition for nonlinear optimal control,” IEEE Transactions on Automatic Control, 2022.
[8] R. B. Schnabel, “A view of the limitations, opportunities, and challenges in parallel nonlinear optimization,” Parallel Computing, vol. 21, no. 6, pp. 875–905, 1995.
[9] S. J. Wright, “Solving optimization problems on computational grids,” 2001.