Efficient CPU-Multi-Threaded Dual Gradient-Projection Algorithm for Embedded Linear Model Predictive Control

Luke Nuculaj, Joseph Volcic
Department of Electrical and Computer Engineering
Oakland University
Rochester Hills, MI, USA
e-mail: {lukenuculaj | volcic}@oakland.edu

***Abstract*—This work considers the dual gradient-projection algorithm (GPAD), a lightweight quadratic programming (QP) solver for real-time embedded model predictive control (MPC) applications, and investigates the use of parallel processors in expediting its computation. Furthermore, we expand on the initial parallel implementation by taking into consideration the advantages that come with matrix compression. Our findings demonstrate that storing linear time-invariant (LTI) matrices in a compressed format and processing data in parallel reduces computation time, achieving a 61.0x speedup on a desktop computer, a 20.1x on a Jetson Nano compared to a sequential non-compressed CPU implementation.**

I. Introduction

As time has progressed, model predictive control (MPC) has proven to be a powerful control methodology, regarded for its ability to synthesize optimal control inputs while factoring in system constraints [1], [2]. Originally, MPC was widely adopted by chemical plants and oil refineries, where it saw initial success controlling multiple variables. As computational power has become more affordable, aerospace and automotive industries turned their attention to MPC as an attractive control technique to be implemented in high-speed environments. Specifically, an MPC algorithm that can run in real-time on an embedded platform and provide solutions within short time frames.

However, compute capability has recently plateaued due to the limitations the Moore's Law and Dennard scaling, hitting an insurmountable frequency and power-density barrier [3], [4]. To this end, there has been extensive research in the realm of using multiple cores – specifically, graphics processing units (GPUs) – to accelerate real-time embedded computing [5]–[8]. As for MPC, there have been various sub-problems tackled by GPU implementations in the literature. For instance, [9] in on parallelizing a warm-started, preconditioned conjugate gradient solver for computing the optimal trajectories of a simulated robot arm, which saw up to a 10x speedup compared to the CPU implementation. [10] aimed to expedite the computations of nonlinear evolutionary MPC on a GPU as applied to the control of pneumatically actuated continuum robot. The authors' experimental results, much like [10], also note a 10x speedup as compared to the sequential implementation.

Our work leverages the parallel capabilities of CPUs to implement the accelerated dual gradient projection algorithm (GPAD) [11], [12]. The GPAD algorithm is a recent development in QP solvers for MPC, hailed for its speed, simplicity, and small memory footprint. Section II constructs the convex QP problem as applied to linear MPC; thereafter, Section III summarizes the main steps of the GPAD algorithm, highlighting how we prepare the linear MPC construction for solving. Section IV considers a preliminary, step-by-step evaluation of the number of floating-point operations (flops) and the percent parallelism for the GPAD algorithm, identifying relevant bottlenecks. Subsequently, the various parallel approaches for each step are described (a linear, time-invariant system is assumed), accompanied by a directed acyclic graph (DAG) used for parallel task scheduling. Section VI compares the average execution times of the various parallel CPU implementations to one another as well as to the sequential GPAD implementation. Additional profiling metrics such as optimal thread-block usage and memory footprints are also considered. This work closes out with a discussion of the major takeaways and future work in Section VII.

II. Linear MPC

The formulation of the receding-horizon optimal control problem for MPC is described by the following

$$\min_{u_0,\dots,u_{N-1}} \quad \ell_N(x_N) + \sum_{k=0}^{N-1} \ell_k(x_k, u_k) \tag{1a}$$

$$\text{s.t.} \quad x_0 = p \tag{1b}$$

$$x_{k+1} = A_k x_k + B_k u_k \tag{1c}$$

$$F_k x_k + G_k u_k \leq c_k \tag{1d}$$

$$k = 0, \dots, N-1 \tag{1e}$$

$$F_N x_N \leq c_N \tag{1f}$$

where $p \in \mathbb{R}^{n_x}$ is the current state, $u_k \in \mathbb{R}^{n_u}$ is the control input at a particular time step k, and $x_k \in \mathbb{R}^{n_x}$ is the state. N is the prediction horizon, which is how many time steps into the future the MPC considers. Our stage and terminal costs $\ell_k(x_k, u_k) : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}$ and $\ell_N(x_N) : \mathbb{R}^{n_x} \to \mathbb{R}$ make

up our cost function, and for linear MPC, they are assigned to be quadratic functions like so

$$\ell_k(x_k, u_k) = \frac{1}{2}\begin{bmatrix} x_k^\top & u_k^\top \end{bmatrix}\begin{bmatrix} Q_k & 0 \\ 0 & R_k \end{bmatrix}\begin{bmatrix} x_k \\ u_k \end{bmatrix} \tag{2a}$$

$$\ell_N(x_N) = \frac{1}{2}x_N^\top Q_N x_N \tag{2b}$$

where weight matrices $Q_k, Q_N \in \mathbb{S}_+^{n_x}$, $R_k \in \mathbb{S}_{++}^{n_u}$.

Defining our optimization variable $z := \begin{bmatrix} u_0^\top & u_1^\top & \cdots & u_{N-1}^\top \end{bmatrix} \in \mathbb{R}^{n_u N}$, we can apply our stage costs (2) to the optimal control problem in (1) to get

$$z^* = \arg\min_z \quad \frac{1}{2}z^\top M z + (Cp + g)^\top z \tag{3a}$$

$$\text{s.t.} \quad Gz \leq Ep + b \tag{3b}$$

where the Hessian matrix $M \in \mathbb{S}_{++}^{n_u N}$, $C \in \mathbb{R}^{n_x \times n_u N}$, $g \in \mathbb{R}^m$, $G \in \mathbb{R}^{m \times n_u N}$, $E \in \mathbb{R}^{m \times n_x}$, and $b \in \mathbb{R}^m$. It should be noted that in the scenario where the model is linear time-invariant (LTI), several matrices in (3) can be computed offline – we will revisit this idea later on.

III. Dual Gradient Projection Algorithm

The crux of the GPAD algorithm was first developed in [13], but first applied to MPC in [11], [12]. For brevity, the derivations of these steps are omitted from this work, and said derivations can be located in the original literature. The steps, as laid out in [12], are the following

$$w_\nu = y_\nu + \beta_\nu(y_\nu - y_{\nu-1}) \tag{4a}$$

$$\hat{z}_\nu = -M_G w_\nu - g_P \tag{4b}$$

$$z_\nu = (1 - \theta_\nu)z_{\nu-1} + \theta_\nu \hat{z}_\nu \tag{4c}$$

$$y_{\nu+1} = [w_\nu + G_L \hat{z}_\nu + p_D]_+ \tag{4d}$$

$$y_0 = y_{-1} = 0, \;\; z_{-1} = 0$$

where $y \in \mathbb{R}^m$ is the dual vector, $M_G := M^{-1}G^\top$, $g_P := M^{-1}(Cp + g)$, $G_L := \frac{1}{L}G$, $p_D := -\frac{1}{L}(Ep + b)$, the iteration count $\nu \in \mathbb{N}$, and L is an upper bound on the maximum eigenvalue of $H := GM^{-1}G^\top$; this work considers the Frobenius norm $\|H\|_F = \sqrt{\sum_{i,j=1}^m |H_{i,j}|^2}$. θ_ν and β_ν are scalar quantities generated by the following recursions:

$$\theta_{\nu+1} = \frac{\sqrt{\theta_\nu^4 + 4\theta_\nu^2} - \theta_\nu^2}{2} \tag{5a}$$

$$\beta_\nu = \theta_\nu(\theta_{\nu-1}^{-1} - 1) \tag{5b}$$

$$\theta_0 = \theta_{-1} = 1. \tag{5c}$$

The GPAD is an iterative algorithm, meaning that (4) is run repeatedly until certain termination criteria are satisfied. The appeal of the GPAD algorithm, as [12] points out, is that we may designate a fixed, worst-case number of iterations $N_\nu \in \mathbb{N}$ that agrees with the sample time of the MPC controller. Such a limit is paramount for implementation in real-time embedded systems that must be equipped to handle worst-case scenarios. Consider Algorithm 1 – the listing for the algorithm with fixed N_ν.

In the case of LTI systems, matrices M_G and G_L can be computed offline and stored in a non-volatile memory source for easy online access. In fact, the only matrices that need to be computed online in all circumstances are p_D and g_P, which are dependent upon the initial state p. As stated in [12], the main computational workload comes from (4b) and (4d), which are matrix-vector multiplications on the order of $O(n_u N m)$. In the following section, we will confirm this claim with a comprehensive flops comparison as well as an evaluation of the level of parallelism of (4).

Algorithm 1 GPAD with fixed number of iterations

```
procedure GPAD(N_ν)
    init: y_0 = y_{-1} = 0, z_{-1} = 0;
    for ν = 0 to N_ν do
        compute w_ν, z_ν, y_{ν+1} as in (4);
    end for
    stop: z_ν = primal solution, y_{ν+1} = dual solution;
end procedure
```

IV. Methodology

A. Preliminary Analysis

To find a starting point for the parallel CPU implementation, an investigation of the preliminary flops and percent parallelism is vital in identifying easily parallelizable bottlenecks. For clarity, a flop is defined as a fundamental arithmetic operation (addition, subtraction, multiplication, and division) on floating-point values.

(4a) consists of a vector subtraction, a scalar multiplication, and a vector addition on vectors $\in \mathbb{R}^m$. Simply put, the subtraction is m flops, the scalar multiplication of the difference is m flops, and the vector addition is m flops, totaling $3m$ flops required for (4a). In fact, (4c) also requires $3m$ flops. As for (4b), there is a matrix-vector multiplication followed by a vector subtraction. Note that we can store the negated version of M_G in non-volatile memory, avoiding extraneous online scalar multiplications. Recalling the sizes of the matrices and vectors from Section III, the matrix multiplication is essentially a repeated vector dot product over each of the rows, resulting in a flop count of $2n_u Nm - n_u N$. Adding in $n_u N$ flops for the vector subtraction, and we see that (4b) requires $2n_u Nm$ flops. For (4d), the matrix-vector multiplication requires $2n_u Nm - m$ flops. Factoring in the two vector additions, the total flop count for this step is $2n_u Nm + m$ flops. As can be seen, these findings align with the claim set forth by [12] regarding time complexity on (4b) and (4d). For convenience, these results are listed in Table I.

TABLE I
GPAD Flop Counts

Step	Flop Count
(4a)	$3m$
(4b)	$2n_u Nm$
(4c)	$3m$
(4d)	$2n_u Nm + m$
Total	$4n_u Nm + 7m$

As for the level of parallelism, the operations we use in (4) are scalar multiplications, vector additions/subtractions,

matrix-vector multiplications, and a projection onto the non-negative orthant in (4d) (in Layman's terms, an elementwise ReLU operation on the input vector). Each of these operations have a percent parallelism of 100%, meaning there is absolutely no interdependence between the computations for distinct output elements. Phrased differently, in ideal conditions, the results for each step in (4d) can theoretically be computed at once. So, we can identify our largest computational bottlenecks residing with (4b) and (4d). The following subsections introduce the various parallel approaches explored by the authors for (4).

B. Parallel Approaches

Per the recommendation of [12], (4) "can easily be executed on m parallel processors". To leverage parallel processors for linear algebra operations, we begin by assigning one thread to compute a distinct section of the output data for each step. This approach provides an intuitive starting point for parallelization and highlights the division of work across threads. The operations involved in steps (4a) and (4c) are straightforward SAXPY operations on non-constant vectors with a with a relatively short length due to the column heavy nature of the matrices. Consequently, we did not heavily explore parallelizing SAXPY, as its relatively short vector lengths offer limited potential for speedup.. Matrix-vector operations in steps (4b) and (4d) account for the majority of the algorithm's workload, making them strong candidates for parallelization.

The matrices remain constant throughout the duration of the algorithm. Leveraging this property, we can partition the data into smaller chunks before launching the MPC. This preprocessing step eliminates the partitioning overhead during runtime, improving efficiency. This is advantageous because it allows us to remove the partitioning overhead during runtime.

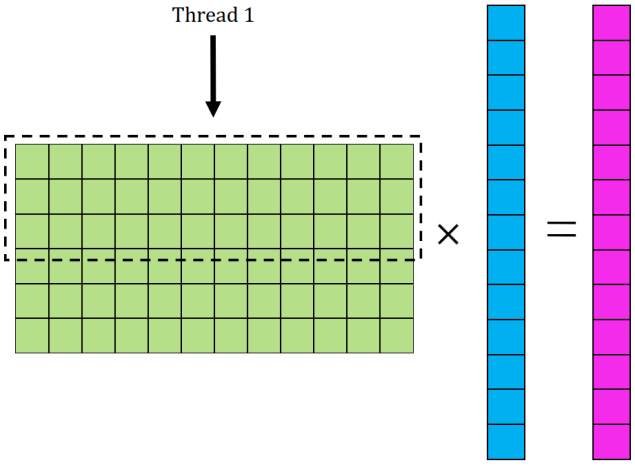

Fig. 1. Data Partitioning for Matrix Vector Multiplication

A further improvement to matrix vector multiplication can be made by compressing the sparse matrices into a compact matrix using ELL compression. ELL compression removes all zero elements and stores the column index of each remaining element. For matrices with a variable number of non zero elements per row, rows with less non zero elements are padded to match the length of the longest row. Eliminating zero elements significantly reduces computational workload but comes at the cost of unaligned data access. However, since the matrices are predominantly sparse, the impact of unaligned access is minimal.

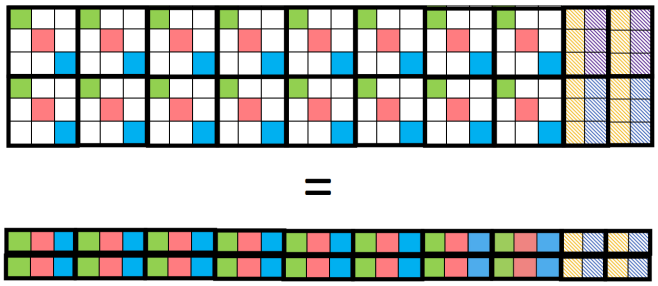

Fig. 2. Matrix Data Compression.

Upon closer examination of GPAD, it becomes clear that multiple steps can be computed concurrently. By assigning threads to these steps, we can reduce the total number of phases required for a prediction, further enhancing parallel efficiency. Using threads to calculate multiple steps at the same time has potential for speedup, by reducing the total number of phases GPAD takes to compute a prediction along with parallelizing work.

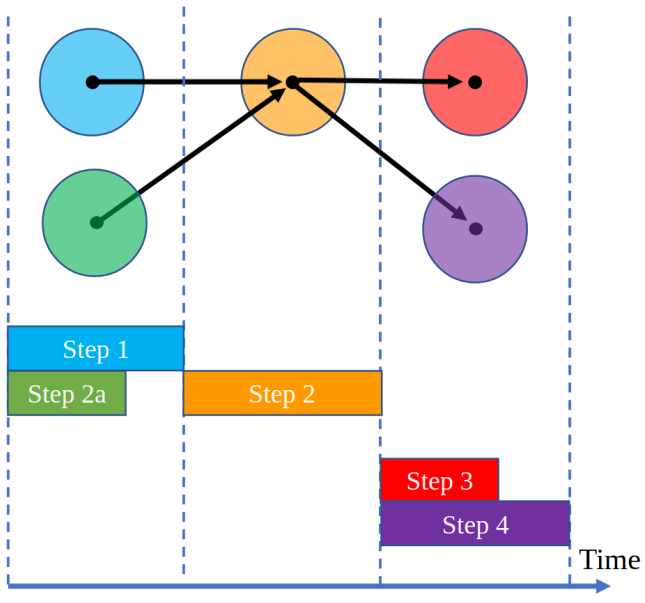

Fig. 3. GPAD Parallel Step Computation.

V. Implementation

For proof of concept, the theoretical parallel implementations discussed in Section IV-B are implemented in C++ –

using the pthreads standard library for parallelization – and executed on a NVIDIA Jetson Nano and a Desktop Computer, the Jetson was equipped with 4 cores, off of a 1.9 GHz clock speed. The desktop was equipped with 8 cores, off a 3.2 GHZ clock speed. The maximum number of cores was assigned to each device, and the parallel and sequential implementations were compared in the section VI.

VI. Results

The parallel design philosophy adapted to this problem did not meet the expected performance predicted in the preliminary analysis. Launching and joining groups of threads needs to be managed efficiently to achieve peak performance. This becomes highly apparent when assigning threads smaller tasks, such as SAXPY seen in steps (4b) and (4d). As indicated in Figures 4 and 5, which display the speed-up of multi-threading steps of the algorithm, at best a small speed-up was achieved on the NVIDIA Jetson Nano; however, the average case led to a net slowdown of the algorithm. For smaller tasks, such as SAXPY, the overhead dominates the execution time, negating any potential gains from parallelism.

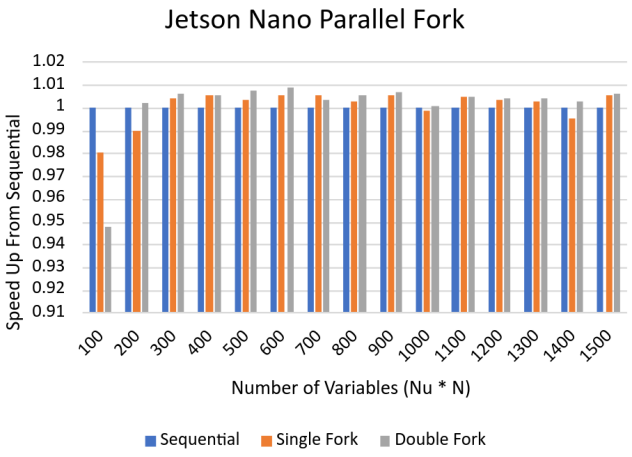

Fig. 4. Jetson Nano GPAD Speed Up from Parallel Step Execution.

Both optimization techniques operating on larger partitions of data performed as expected. Reducing the total amount of work done by a single thread is a generally good technique for decreasing the runtime of a function. Similarly, reducing the amount of work done by a thread when operating on large data sets also produces a significant speed-up. This is demonstrated in Figures 6 and 7, which compare the optimized implementations for larger data partitions on the Jetson Nano and desktop platforms. In all cases a significant speed-up can be observed.

The performance analysis of the parallel SAXPY and Matrix-Vector Multiplication highlights the critical role of balancing task granularity with thread execution efficiency. While smaller tasks, such as SAXPY, suffer from significant overhead due to frequent thread management, larger tasks benefit from parallel execution as the overhead becomes

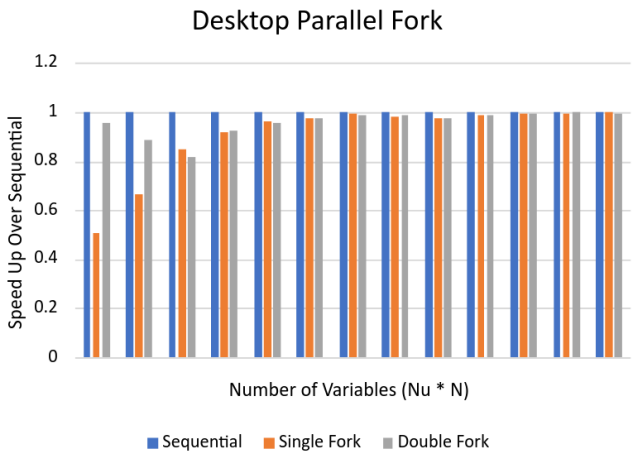

Fig. 5. Desktop GPAD Speed Up from Parallel Step Execution.

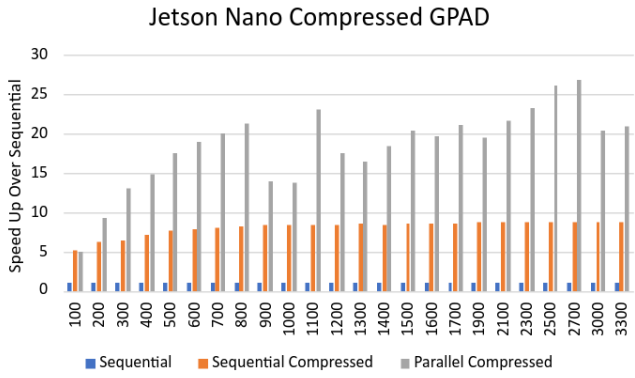

Fig. 6. Desktop GPAD Speed Up from Parallelization and Compression.

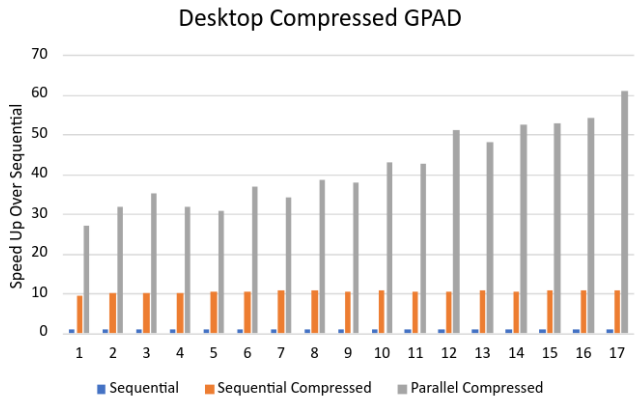

Fig. 7. Jetson Nano GPAD Speed Up from Parallelization and Compression.

negligible relative to the workload. Figure 8 and **??** illustrates this comparison, showing the total execution time of SAXPY and Matrix-Vector Multiplication across varying data sizes. The GPAD compressed parallel algorithm demonstrates this trend effectively: as data size increases, execution time scales more efficiently, leading to substantial speed-ups. This under-

scores the importance of optimizing workload distribution and task size when designing parallel algorithms, especially on resource-constrained platforms like the Jetson Nano.

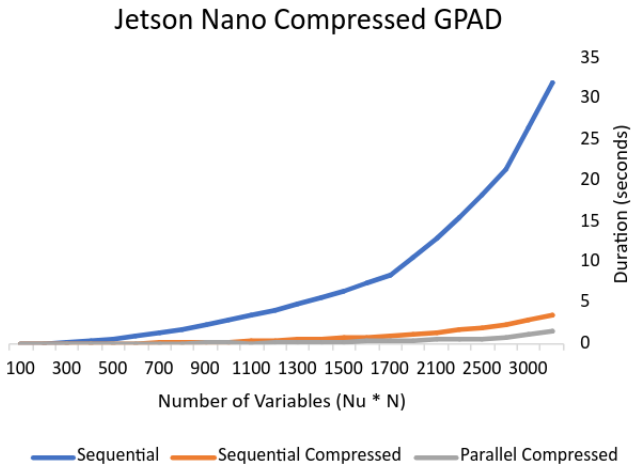

Fig. 8. Jetson Nano GPAD Duration Comparison.

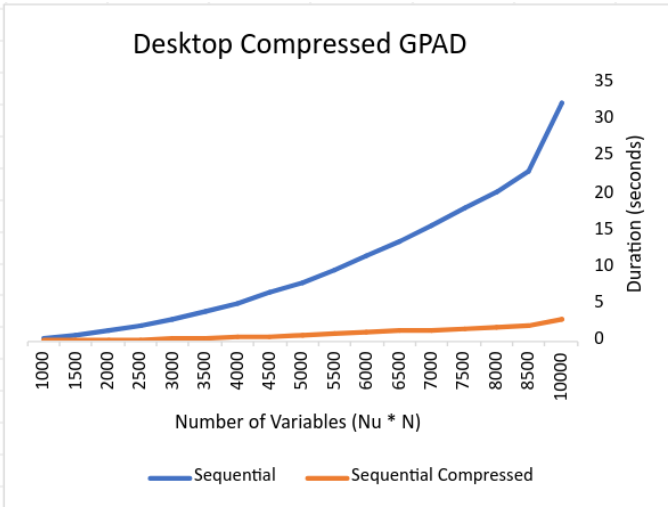

Fig. 9. Desktop GPAD Duration Comparison.

VII. Conclusion

According to Amdahl's Law, task speedup increases as more resources, such as CPU cores, are added to the system and a significant portion of the application is parallelized. However, the maximum achievable speedup is inherently constrained by the serial portions of the code. In the multithreaded CPU implementation of the GPAD algorithm, performance gains were achieved through effective task parallelization across multiple cores. Despite these gains, factors such as thread management overhead and memory access contention limited the scalability of the implementation. Applied optimizations, such as minimizing thread synchronization and improving cache locality, helped alleviate these bottlenecks and further enhanced performance, demonstrating the importance of balancing thread granularity with efficient resource utilization.

References

[1] D. Q. Mayne, "Model predictive control: Recent developments and future promise," *Automatica*, vol. 50, no. 12, pp. 2967–2986, 2014.

[2] M. Schwenzer, M. Ay, T. Bergs, and D. Abel, "Review on model predictive control: An engineering perspective," *The International Journal of Advanced Manufacturing Technology*, vol. 117, no. 5, pp. 1327–1349, 2021.

[3] L. Johnsson and G. Netzer, "The impact of moore's law and loss of dennard scaling: Are dsp socs an energy efficient alternative to x86 socs?" in *Journal of Physics: Conference Series*, vol. 762, no. 1. IOP Publishing, 2016, p. 012022.

[4] P. A. Gargini, "How to successfully overcome inflection points, or long live moore's law," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 51–62, 2017.

[5] L. Kosmidis, J. Lachaize, J. Abella, O. Notebaert, F. J. Cazorla, and D. Steenari, "Gpu4s: Embedded gpus in space," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2019, pp. 399–405.

[6] E. Güney, C. Bayilmiş, and B. Çakan, "An implementation of real-time traffic signs and road objects detection based on mobile gpu platforms," *IEEE access*, vol. 10, pp. 86 191–86 203, 2022.

[7] M. Díaz, R. Guerra, P. Horstrand, E. Martel, S. López, J. F. López, and R. Sarmiento, "Real-time hyperspectral image compression onto embedded gpus," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 12, no. 8, pp. 2792–2809, 2019.

[8] D. Hallmans, M. Åsberg, and T. Nolte, "Towards using the graphics processing unit (gpu) for embedded systems," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. IEEE, 2012, pp. 1–4.

[9] E. Adabag, M. Atal, W. Gerard, and B. Plancher, "Mpcgpu: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the gpu," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 9787–9794.

[10] P. Hyatt and M. D. Killpack, "Real-time nonlinear model predictive control of robots using a graphics processing unit," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 1468–1475, 2020.

[11] P. Patrinos and A. Bemporad, "An accelerated dual gradient-projection algorithm for embedded linear model predictive control," *IEEE Transactions on Automatic Control*, vol. 59, no. 1, pp. 18–33, 2013.

[12] A. Bemporad and P. Patrinos, "Simple and certifiable quadratic programming algorithms for embedded linear model predictive control," *IFAC Proceedings Volumes*, vol. 45, no. 17, pp. 14–20, 2012.

[13] Y. Nesterov, "A method of solving a convex programming problem with convergence rate mathcal {O}(1/kˆ {2})," in *Sov. Math. Dokl*, vol. 27, 1986.