2024 European Control Conference (ECC)
June 25-28, 2024. Stockholm, Sweden

# Multilevel parallel GPU implementation of SQP solvers for Nonlinear MPC

P. C. N. Verheijen[1], A. H. Derkani[1], Y. A. Agarwal[1], M. Lazar[1] and D. Goswami[1]

***Abstract*— In recent literature, it has been shown that the number of steps in a sequential quadratic programming algorithm for a non-linear model predictive control (NMPC) problem can be greatly reduced by a parallel shooting method. The efficiency of such a parallel shooting method further depends on how the algorithm is implemented on parallel computing platforms such as Graphics Processing Units (GPUs). The GPU implementation should consider the degree of parallelism necessary for higher time efficiency as well as the hardware resource consumption/limitation at the GPU for a given problem size. In this paper, we present a multilevel parallel GPU implementation for sequential quadratic programming and an (Alternating Direction Method of Multipliers) ADMM solver. First, we introduce a GPU implementation enabling parallel computing of many quadratic programs (QPs) by *functional parallelism*. Next, we parallelize each QP solver using *data parallelism* of basic linear matrix operations. We show that the proposed GPU implementation greatly scales with the degree of parallelism in the parallel shooting method. Further, we show how a GPU implementation can be configured for a given problem size avoiding resource overprovisioning.**

***Index Terms*— Nonlinear MPC, Sequential Quadratic Programming, CUDA programming, Parallel Shooting**

## I. Introduction

Nonlinear MPC is a key enabling technology for the future; it started in the process industry but now enters high-tech, safety–critical industries with fast dynamics: mechatronics, electrical machines, power electronics, power systems, automotive systems, robotics - UAVs, water networks [1] or manufacturing systems, to name a few. In these new application areas, some common characteristics challenge the real–time implementation of NMPC at a large scale: strongly nonlinear dynamics leading to non-convex nonlinear programs to be solved online and large state-space dimensions which in combination with long horizons yield medium to large–scale nonlinear programs that need to be solved efficiently.

Nonetheless, a decent amount of off-the-shelve solvers are available to solve the corresponding NLPs [2]. These include strategies such as interior point methods, projected gradient methods, or Sequential Quadratic Programming (SQP) [3] [4]. To expand on the latter, SQP solves an NLP by successively linearizing the problem and solving the resulting Quadratic Program (QP) using any favored QP solver. With a wide range of QP solvers available (such as Hildreth, OSQP [5], qpOASES [6], etc), the efficiency of the SQP algorithm can be tailored to the NLP problem size and structure.

In the domain of MPC, a vast body of literature offers insights into implementing diverse optimization algorithms on single- and multi-core CPUs, however, few studies have focused on GPU implementations. For example, [7] demonstrated the improved performance of the matrix-free interior point method with GPU acceleration. The GPU implementation of the OSQP [8] algorithm, which is based on the Alternating Direction Method of Multipliers (ADMM), accelerates solving large-scale problems on a GPU [8]. Other GPU-based implementations for solving QP problems include qpDUNES [9], parallel Interior Point [10], and PQP [11]. These methods accelerate solving a QP through *data parallelism*, i.e., computing basic linear algebra operations within a single iteration of the corresponding algorithm in parallel. To delve a bit deeper into the implementation side of these algorithms, consider the details in [8], which include CUDA libraries such as Thrust, cuBLAS, and cuSPARSE to efficiently perform complex operations such as reduction, linear algebra, and sparse matrix operations. On the other hand, [12] shows that by splitting the problem over the input channels (thus generating multiple problems), the reduced problems can each be computed in parallel. Although they then implement it for an FPGA, [13] expands this idea to a GPU implementation. The idea of computing multiple QPs in parallel, i.e., *function parallelism* allows for another dimension of achieving faster NMPC solvers. The results reported by [13] also indicate that both parallel approaches can achieve improved performance and increased throughput compared to sequential solvers. However, it is unclear how such implementation can take into account the limitations of the GPU resources given the QP problem size. We present an alternative perspective on how to use the GPU capabilities for solving multiple QPs simultaneously using the OSQP algorithm using both levels of parallelism while offering configuration knobs concerning the problem size.

In this work, we introduce multilevel parallel GPU implementation for sequential quadratic programming and an ADMM algorithm targeting an NVIDIA architecture (most commonly used in industrial platforms). At the QP level, we present functional parallel implementation enabling a large number of QPs to be solved in parallel. Within each QP, we present a data parallel GPU implementation significantly reducing the execution time of a single QP problem. In

This research was performed within the framework of the research program AquaConnect, funded by the Dutch Research Council (NWO, grant-ID P19-45) and public and private partners of the AquaConnect consortium and coordinated by Wageningen University and Research.

[1]Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands, Corresponding E-mail: p.c.n.verheijen@tue.nl

combination, the proposed method shows great scalability concerning the degree of parallelism. Furthermore, we show how to configure a GPU implementation for a given problem size to avoid resource overprovisioning.

## II. Preliminaries and Problem Statement

Sequential Quadratic Programming (SQP) solves a nonlinear problem by sequentially linearizing the problem over its current operating point. The operating point (that we shall further refer to as a "trajectory") is updated with the optimal solution from the linearized Quadratic Program (QP). For the next part, we re-use the explanation in [14], adapted from [4]. Consider the following nonlinear MPC problem:

$$\begin{aligned}\min_{x_{i|k},u_{i|k}} \quad & \sum_{i=0}^{N-1} f(x_{i|k},u_{i|k}) + f_T(x_{N|k}) \\ \text{s.t.} \quad & g(x_{i|k},u_{i|k}) \leq 0, \qquad \forall i=\{0,\dots,N-1\} \\ & g_T(x_{N|k}) \leq 0, \\ & x_{0|k} = x(k), \\ & h(x_{i|k},u_{i|k}) = x_{i+1|k}, \quad \forall i=\{0,\dots,N-1\},\end{aligned} \tag{1}$$

which minimizes the sum of the running cost $f(\cdot)$ over the prediction horizon $N$ and the terminal cost $f_T(\cdot)$. Problem (1) computes a control input $u(k) = u_{0|k}$ for a discrete nonlinear system

$$x(k+1) = h(x(k),u(k)), \qquad k \in \mathbb{N}, \tag{2}$$

where $u(k) \in \mathbb{R}^q$ and $x(k) \in \mathbb{R}^n$. To simplify the notation in the remainder of the paper, consider

$$\begin{aligned} z_{i|k} &= \begin{bmatrix} x_{i|k}^T & u_{i|k}^T \end{bmatrix}^T, \quad \forall i=\{0,\dots,N-1\} \\ z_{N|k} &= x_{N|k}. \end{aligned}$$

With this notation, we linearize (1) over an estimated guess trajectory $z_{i|k}^g$ [4], which results in the following QP problem:

$$\begin{aligned}\min_{\Delta z_{i|k}} \quad & \sum_{i=0}^{N} \frac{1}{2}\Delta z_{i|k}^T Q_i \Delta z_{i|k} + \Delta z_{i|k}^T F_i \\ \text{s.t.} \quad & M_i \Delta z_{i|k} \leq -s_i, \qquad \forall i=\{0,\dots,N-1\} \\ & M_N \Delta z_{N|k} \leq -s_N, \\ & E\Delta z_{0|k} = x(k) - E z_{0|k}^g \\ & E\Delta z_{i+1|k} - A_i \Delta z_{i|k} = r_{i+1}, \quad \forall i=\{0,\dots,N-2\} \\ & \Delta z_{N|k} - A_{N-1}\Delta z_{N-1|k} = r_N,\end{aligned} \tag{3}$$

where the optimization variable $\Delta z_{i|k}$ is the optimal step direction with respect to $z_{i|k}^g$, $E = \begin{bmatrix} I_{n\times n} & \mathbf{0}_{n\times q}\end{bmatrix}$ and

$$\begin{aligned} Q_i &= \left.\frac{\partial^2 f(z)}{\partial z^2}\right|_{z_{i|k}^g}, F_i = \left.\frac{\partial f(z)}{\partial z}\right|_{z_{i|k}^g}, M_i = \left.\frac{\partial g(z)}{\partial z}\right|_{z_{i|k}^g}, \\ Q_N &= \left.\frac{\partial^2 f_T(z)}{\partial z^2}\right|_{z_{N|k}^g}, F_N = \left.\frac{\partial f_T(z)}{\partial z}\right|_{z_{N|k}^g}, s_i = g(z_{i|k}^g), \\ M_N &= \left.\frac{\partial g_T(z)}{\partial z}\right|_{z_{N|k}^g}, s_N = g_T(z_{N|k}^g), \\ A_i &= \left.\frac{\partial h(z)}{\partial z}\right|_{z_{i|k}^g}, r_i = h(z_{i|k}^g) - E z_{i+1|k}^g, \\ r_N &= h(z_{N-1|k}^g) - z_{N|k}^g. \end{aligned} \tag{4}$$

To express the cost function without the prediction time index $i$, consider the following augmented vectors and matrices:

$$\begin{aligned} \mathbf{z}^g &= \begin{bmatrix} z_{0|k}^g \\ \vdots \\ z_{N|k}^g \end{bmatrix}, \mathcal{A} = \begin{bmatrix} E & & & \\ -A_0 & \ddots & & \\ & \ddots & E & \\ & & -A_{N-1} & I \end{bmatrix}, \\ \mathcal{Q} &= \begin{bmatrix} Q_1 & & \\ & \ddots & \\ & & Q_N \end{bmatrix}, \mathcal{M} = \begin{bmatrix} M_0 & & \\ & \ddots & \\ & & M_N \end{bmatrix}, \\ \mathcal{F} &= \begin{bmatrix} F_1 \\ \vdots \\ F_N \end{bmatrix}, \mathbf{r} = \begin{bmatrix} x(k) - E z_{0|k}^g \\ r_1 \\ \vdots \\ r_N \end{bmatrix}, \mathbf{s} = \begin{bmatrix} s_1 \\ \vdots \\ s_N \end{bmatrix}, \end{aligned} \tag{5}$$

where $\mathbf{z}^g \in \mathbb{R}^p$, with $p = N(n+q)+n$, and corresponding control problem

$$\begin{aligned}\min_{\Delta \mathbf{z}} \quad & \Delta \mathbf{z}^T \mathcal{Q} \Delta \mathbf{z} + \Delta \mathbf{z}^T \mathcal{F} \\ \text{s.t.} \quad & \mathcal{M}\Delta \mathbf{z} \leq -\mathbf{s} \\ & \mathcal{A}\Delta \mathbf{z} = \mathbf{r}.\end{aligned} \tag{6}$$

Solving (6) using any QP solver and updating the guess trajectory $\mathbf{z}_+^g(k) = \mathbf{z}^g(k) + \alpha \Delta \mathbf{z}$. By repeating this cycle (of linearizing the problem, solving the problem and updating the guess trajectory), the corresponding guess trajectory will converge to the solution of the NMPC problem if the initial guess trajectory is "close" to the optimal solution or, the step size $\alpha$ is chosen to guarantee convergence of the linearization error [3]. These two conditions will play a crucial role in what we explain next.

*Remark 1:* We use the apostrophes around "close" to emphasize that this is not a measurable bound. Instead, this vastly varies per problem and operating point, and more specifically, how quickly the linearization error increases when $\|\mathbf{z}^*(k) - \mathbf{z}^g(k)\|_2$ increases.

### A. Parallel Shooting SQP

In Parallel Shooting SQP (PS-SQP), the convergence of the SQP algorithm is accelerated through the shooting method [14]. This exploits shooting in two phases, each designed to strengthen convergence concerning the aforementioned conditions. Furthermore, since each step of each of the "shot" SQP is independent, they can (and should) all be executed in parallel. We will use Figure 1 to illustrate the algorithm explained next.

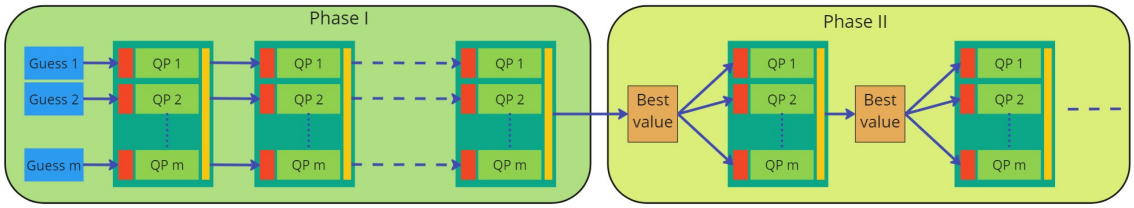


Fig. 1. Parallel Shooting SQP

*a) Phase 1: Shooting in trajectories:* In this first phase, we generate $m$ initial trajectories, each randomly displaced from our "personal best-selected candidate" (most commonly, the shifted optimal solution from the last time–index). This shooting principle enhances the chance of having an initial trajectory close enough to the solution for fast convergence (fixing $\alpha = 1$). More specific strategies on how to generate these trajectories are discussed in [14]. Yet, as is mentioned there as well, since $m$ must be a finite number, there exists no guarantee that any of the generated initial trajectories is "close". In that case, the PS-SQP algorithm switches to phase 2, which is designed to guarantee convergence.

*b) Phase 2: Shooting in step sizes:* In the standard SQP algorithm, convergence is guaranteed through selecting the right step–size $\alpha$, often obtained with a searching strategy over the Wolfe Conditions and the Armijo Constraint [3]. To prevent a lengthy search of the optimal step–size, we shoot over many values of $\alpha$, in parallel. At the beginning of a new SQP iteration, we only consider the best (i.e., the most converged) trajectory from the previous iteration and linearize all parallel NLPs using the best trajectory and a different step–size. Assuming plenty of parallel QPs are employed, a converging step–size will always be tested.

In both phases, having the resources to execute many QPs in parallel is key to convergence. However, the limiting number of cores present in a CPU causes a problem in running this algorithm efficiently on the CPU (given that most QP solvers exist as a single-thread CPU code).

### B. ADMM Algorithm

For each SQP step, the corresponding QP can be solved using any preferable solver. We decided to use the OSQP solver as the iterations can be highly parallelized and the underlying ADMM algorithm shown in [8] provided more than the necessary details to implement the solver. The OSQP solver is used to solve QP problems of the form [8]:

$$\begin{aligned} \min_{\Delta \mathbf{z}} \quad & \frac{1}{2}\Delta \mathbf{z}^T \mathcal{Q} \Delta \mathbf{z} + \mathcal{F}^T \Delta \mathbf{z} \\ \text{s.t.} \quad & l \leq \mathcal{B}\Delta \mathbf{z} \leq u, \end{aligned} \tag{7}$$

which equals problem (6) by stating that,

$$\mathcal{B} = \begin{bmatrix} \mathcal{M} \\ \mathcal{A} \end{bmatrix}, \quad l = \begin{bmatrix} -\infty \\ \mathbf{r} \end{bmatrix}, \quad u = \begin{bmatrix} -\mathbf{s} \\ \mathbf{r} \end{bmatrix}. \tag{8}$$

ADMM iteratively solves a QP problem by robust dual decomposition of the primal variable. An iteration of the algorithm is shown in Algorithm 1 [8].

**Algorithm 1** ADMM, rephrased from [8, Alg. 1]

**Input:** Problem matrices as in (7)

1: **while** convergence criteria **do**
2: $\quad \Delta\tilde{\mathbf{z}}^{t+1} \leftarrow \left(\mathcal{Q} + \sigma I + \mathcal{B}^T R \mathcal{B}\right)^{-1} (\sigma \Delta \mathbf{z}^t - \mathcal{F}^T + \mathcal{B}^T (R\mu^t - \lambda^t)$
3: $\quad \tilde{\mu}^{t+1} \leftarrow \mathcal{B}\Delta\tilde{\mathbf{z}}^{t+1}$
4: $\quad \Delta \mathbf{z}^{t+1} \leftarrow \alpha \Delta\tilde{\mathbf{z}}^{t+1} + (1-\alpha)\Delta \mathbf{z}^t$
5: $\quad \mu^{t+1} \leftarrow \alpha\tilde{\mu}^{t+1} + (1-\alpha)\mu^t + R^{-1}\lambda^t$
6: $\quad \mu_i^{t+1} \leftarrow \min(\max(\mu_i^{t+1}, l_i), u_i) \quad \forall i = \{0, \dots n_d\}$
7: $\quad \lambda^{t+1} \leftarrow \lambda^t + R\left(\alpha\tilde{\mu}^{t+1} + (1-\alpha)\mu^t - \mu^{t+1}\right)$
8: **end while**

**Return:** $\Delta \mathbf{z}$

The ADMM algorithm discussed here applies a preconditioning step using the Modified Ruiz Equilibration algorithm [8, Alg. 2]. Furthermore, $\left(\mathcal{Q} + \sigma I + \mathcal{B}^T R \mathcal{B}\right)^{-1}$ only needs to be (re)computed if $R$ is changed, showing that most iterations only consist of linear matrix operations and element-wise clipping. Both of these operations can be extensively parallelized.

## III. GPU Programming Model

In this section, we illustrate various programming abstraction layers of GPU architecture and Compute Unified Device Architecture (CUDA).

### A. GPU Architecture

Figure 2 illustrates a GPU's multi-core architecture. The parallel cores, as depicted in the figure, are organized as an array of streaming multiprocessors (SMs). Arithmetic and other instructions are executed by the SMs. Each SM comprises several streaming processors (SPs), usually multiple of 32, schedulers, registers, and shared memory. The shared memory is exclusively accessible by SPs within an SM. In addition, GPUs also have DRAM, which is globally accessible by all SMs and thus often referred to as global memory. The capacity of this memory varies depending on the GPU model but typically ranges in gigabytes. In comparison to shared memory, it has lower bandwidth and longer latency, yet the global memory bandwidth is as high as 1550 GB/s on the A100 GPU.

### B. GPU Programming Abstractions and Mapping

GPUs were originally developed with a primary focus on handling graphical tasks. However, considering its parallel computing power makes it suitable for general-purpose applications as well. Hence, NVIDIA developed the CUDA platform to streamline the utilization of GPUs for programmers. In this section, the main concepts of GPU programming are described.

Threads, warps, thread blocks, and grids are fundamental abstract layers of GPU programming and play a vital role in how tasks are organized and executed on the GPU. The definitions of these four concepts are provided below:

1) **Thread:** Thread is the smallest execution unit on a processor. Each thread has a unique thread ID to access memory and synchronize with other threads. Threads can communicate with each other using shared memory. The concept of GPU threads is similar to CPU although the number of threads in GPUs is much more compared to CPUs. A thread is assigned to a single SP.
2) **Warp:** A warp is a group of 32 consecutive threads that execute in parallel on a single SM. The SPs in an SM follow the Single Instruction Multiple Data (SIMD) policy and run a wrap in parallel. Therefore, a warp requires 32 SPs to be executed. An SM executes one or more warps. To avoid waste of SPs, the SMs have a number of SPs multiple of 32.
3) **Thread Block:** As its name implies, it is a group of threads that is mapped to an SM. As described before within each SM there are registers and shared memory by which threads inside a thread block can communicate. Multiple thread blocks can be mapped to one SM.
4) **Grid:** Grid is a collection of thread blocks organized as a two-dimensional array that may be executed by one or more SMs.

Abstract programming layers and how they are mapped on actual GPU resources are depicted in Fig. 2. For the given GPU architecture, the maximum number of SPs implies the highest level of parallelism achievable. In our case, we use an NVIDIA RTX 3070 Ti, which has 6144 CUDA cores, subdivided into 48 SMs, each partitioned into 4 blocks of 32 SPs (note that for our purpose, 1 SP $\approx$ 1 CUDA core).

### C. CUDA Kernels

CUDA kernel is a function executed on GPU resources using the programming abstraction layers. *Host* and *Device* are two primary terms in CUDA programming and refer to *CPU* and *GPU*, respectively. Executing a CUDA kernel involves three essential steps, which are explained below:

1) **Host-to-Device (h2d) Transfer:** The process of copying input data from host to device.
2) **Execute GPU Program:** The GPU program is loaded and is called by the host to be executed as per various programming abstraction layers.

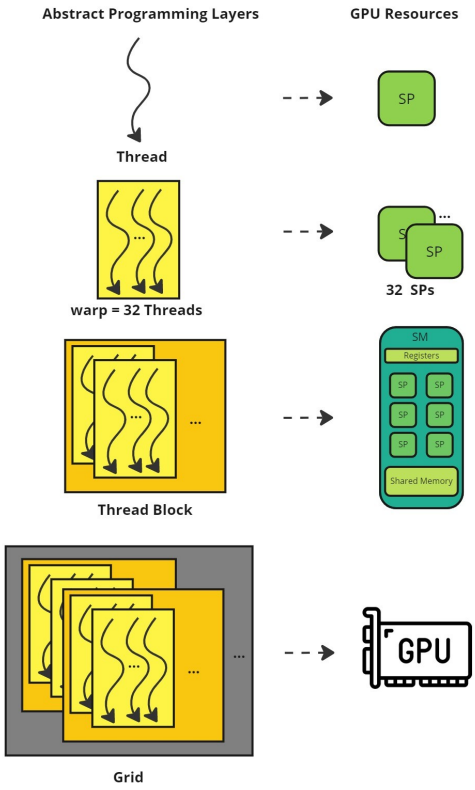


Fig. 2. CUDA Programming Layers and GPU Resources

3) **Device-to-Host (d2h) Transfer:** The process of copying the results from device to host.

For illustration, consider the following operation between two $100 \times 1$ vectors, which can be implemented on CPU and GPU by a function and kernel `add`, respectively,

$$\mathbf{c}_{100\times1} = \mathbf{a}_{100\times1} + \mathbf{b}_{100\times1}. \tag{9}$$

First of all, how function `add`, listed in Listing 1, is executed on a CPU is explained to get a better understanding of how tasks are executed differently on GPUs compared to CPUs. The function `add` gets arrays `a` and `b` as the inputs and stores the results in an array `c`. Line 3 of the function `add` is executed sequentially $n$ times by iterating in the loop.

```
void add(int *a, int *b, int *c, int n){
    for(int i = 0; i < n; i++){
        c[i] = a[i] + b[i];
    }
}
```

Listing 1. Array Addition Function on CPU

The `add` function can be written as a CUDA kernel as shown in Listing 2. This CUDA kernel runs on a device (or GPU) when it is invoked by a host (or CPU) as shown in Listing 3. The host uses two parameters `blkcnt` and `threadspblk` to determine the number of blocks and threads per block, respectively, that the kernel will be mapped on. Here, `d_a`, `d_b`, and `d_c` are the arrays that are stored on the GPU's global memory. In the remainder of this section, how threads and blocks are mapped and executed on the GPU resources is explained.

```
__global__ void add(int *a, int *b, int *c, int n){
    int tid = blockIdx.x*blockDim.x+threadIdx.x;
    if (tid < n){
        c[tid] = a[tid] + b[tid];
    }
}
```

Listing 2. Array Addition Kernel on GPU

```
int blkcnt = 2;
int threadspblk = 50;
add<<<blkcnt, threadspblk>>>(d_a, d_b, d_c, 100);
```

Listing 3. Calling CUDA Kernel from Host

In Listing 2, each addition operation is assigned to a thread and is executed in parallel. Each thread is assigned to an SP. For the CUDA kernel shown in Listing 2, we need at least $n$ threads to run all the addition operations in parallel. All the threads are divided into several blocks, `blkcnt`. Hence, in each block, there are `threadspblk` threads. Therefore, we need

$$\texttt{blkcnt} \times \texttt{threadspblk} \geq n.$$

Each block is assigned to an SM to be executed. An SM follows the Single Instruction Multiple Data (SIMD) policy and runs 32 operations in parallel. Therefore, `threadspblk` threads in the block are divided into groups of 32 threads (called warp). The number of warps in the corresponding SM (where the block is assigned to) is given by,

$$\#\texttt{warps} = \texttt{ceil}\left(\frac{\texttt{threadspblk}}{32}\right).$$

SMs schedule the execution of warps. The choice of the parameters `blkcnt` and `threadspblk` implies different GPU resource mapping.

These concepts are used to deploy multilevel parallelism for the PS-SQP problem and show the limiting factors that have to be considered when designing such a GPU implementation.

## IV. Multilevel Parallel SQP solver on a GPU

In this section, we illustrate the idea of executing multiple QP solvers concurrently on a GPU. First, we introduce how to parallelize QPs on the GPU and then how the solution of each QP is computed using parallel steps. Here we elaborate on two levels of parallelism, first, in Section IV-A, we illustrate how to compute QPs in parallel, also referred to as *function parallelism*. Then, in Section IV-B, independent data operations are executed inside each QP in parallel as well, which is known as *data parallelism*.

### A. Function Parallel QP solving

To refer back to Section II, by using a parallel shooting method, we can accelerate convergence by increasing the odds of having an initial trajectory closer to the optimal trajectory. Therefore, we now illustrate how to compute the corresponding QPs in parallel. As per Equation 7, the ADMM solver takes $\mathcal{Q}$, $\mathcal{F}$, $\mathcal{B}$, $l$ and $u$ as inputs and gives $\Delta\mathbf{z}$ as output. The ADMM solver is deployed as `ADMMSolver()` (CUDA) kernel. To solve $m$ QPs in parallel, we invoke `ADMMSolver()` kernel $m$ times. Each kernel

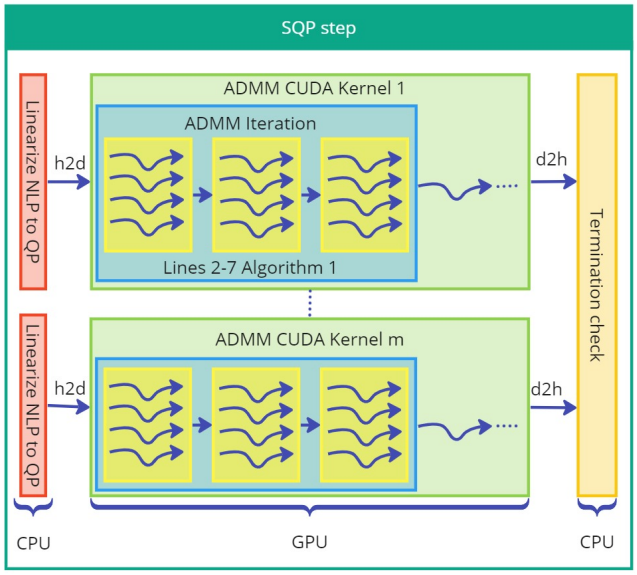


Fig. 3. Paralellism in a single Parallel Shooting SQP step

solves a QP with different values of input parameters. For example, `Q[0]` in Listing 4 is an input for the first instance of the `ADMMSolver()` kernel. Furthermore, the inputs for each QP are stored in GPU global memory and are used by the kernel. Each kernel is executed on a single thread in a block with `blkcnt=1` and `threadspblk=1` as shown in Listing 4.

```
cudaStream_t kernel_id[m];
for (int i = 0; i < m; i++){
    ADMMSolver<<<1,1,0,kernel_id[i]>>>(Q[i], f[i],
    B[i], l[i], u[i], z[i], lam[i], nPrim, nDual);
}
cudaDeviceSynchronize();
```

Listing 4. $m$ `ADMMSolver()` kernels in parallel

Listing 4 shows the invocation of $m$ `ADMMSolver()` kernels on the GPU. This is deployed by defining $m$ `cudaStream_t` objects and the kernels are invoked in a for loop and are synchronized by the command in Line 5 in Listing 4. There are four configuration arguments passed to the kernel. The first two configuration arguments (`blkcnt` and `threadspblk`) are the number of blocks and the number of threads per block as explained in Section III. Since this kernel is the main code of the ADMM and should be executed mostly sequentially, the kernel is executed as a single thread. Then, the third configuration argument defines the amount of shared memory that needs to be allocated, but this is kept at zero as we use global memory. The last configuration argument is the index of the kernel ID. The degree of parallelism in the PS-SQP method is dictated by the number of $m$, increasing numbers of $m$ implies a higher degree of parallelism, but it is further restricted by the degree of data parallelism, which can be linked to the problem size (i.e., smaller problem require less threads for parallel computing). In Figure 3, the function parallelism is displayed by the multiple light green blocks, each a different ADMM

kernel, but executed in parallel on a GPU.

### B. Data Parallel QP solver kernel

Running the `ADMMSolver()` kernel on a single thread implies sequential execution of the ADMM algorithm, which is time–inefficient. Therefore, we aim to parallelize basic linear matrix operations within a single QP solver by invoking *child* kernels from the parent kernel – the `ADMMSolver()` kernel. Such kernel hierarchy allows for efficient management and allocation of the global memory. This is also illustrated in Figure 3, where inside each ADMM iteration of the ADMM CUDA Kernel, a child kernel is illustrated as a yellow block, executed by many threads. The child kernels are defined by the two parameters, `blkcnt` and `threadspblk`, as explained in Section III and shown in Listing 6. The choice of these parameters should be matched to the size of the QP problem or can be linked to the hardware limits of the GPU used, in relation to the amount of parallel QPs that are desired to be executed.

Consider Listing 5, which is a child kernel invoked by the `ADMMSolver()` kernel as shown in Listing 6 that computes Line 4 in Algorithm 1:

```
__global__ void update_z(double *z, double *zh, int
     nPrim, double alpha){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    for(int i = tid; i < nPrim; i+=step) {
        z[i] = alpha * zh[i] + (1 - alpha) * z[i];
    }
}
```

Listing 5. Kernel to update z

```
update_z<<<32,64>>>(z, zh, nPrim, alpha);
```

Listing 6. Calling child kernel to update z

The operation of this function is the addition of two column vectors $\Delta\tilde{\mathbf{z}} =$`zh` and $\Delta\mathbf{z} =$`z`, whilst multiplying each with a scalar $\alpha$. This operation is row-independent and can thus be executed in parallel. The parallelism is achieved by executing each thread to compute just one row of the resulting column vector. Let us consider the QP problem size to be 2000 (i.e., $\mathbf{z} \in \mathbb{R}^{2000}$), which implies that we need at least 2000 threads to completely parallelize the operation. As explained in Section III the number of threads must be selected as a multiple of 32. For this problem, this can be achieved by setting `blkcnt=32` and `threadspblk=64` as shown in Listing 6. Alternatively, `blkcnt=16` and `threadspblk=128`, however, the NVIDIA Ampère architecture only allows for 64 threads per block. Furthermore, as also shown in Listing 5, if the number of threads is less than the number of elements in $\Delta\mathbf{z}$, (some) threads will compute more than one value, increasing the execution time. The product of these two parameters is therefore a configuration knob to design the degree of data parallelism.

## V. Illustrative Example

To demonstrate the workings of the GPU implementation, we considered the control of an inverted pendulum using NMPC. This problem, with all the chosen control and system parameters, is equal to the problem presented in [14].

Consider the following continuous time dynamics [15]:

$$\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= \frac{u\cos x_1 - (M+m)g\sin x_1 + ml(\cos x_1 \sin x_1)x_2^2}{ml\cos^2 x_1 - (M+m)l} \\
\dot{x}_3 &= x_4 \\
\dot{x}_4 &= \frac{u + ml\sin x_1 x_2^2 - mg\cos x_1 sinx_1}{M + m - m\cos^2 x_1}
\end{aligned} \tag{10}$$

where $x_1$ is the pendulum angle, with $x_1 = 0$ corresponding to the upright position, $x_2$ is the angular velocity, $x_3$ is the position of the cart, $x_4$ is the cart velocity. The model in (10) is discretized using backwards Euler with a sampling period of $T_s = 0.02s$. However, in the simulation, the system response is computed using an `ode45` solver for the continuous time dynamics. The cart's position is constrained by $-10 \leq x_1 \leq 10$, and the input force is constrained to $-500 \leq u \leq 500$. The control objective is to center the cart from a starting position of $x_0 = \begin{bmatrix}\pi & 0 & -5 & 0\end{bmatrix}^T$ while swinging the pendulum upright (and keeping it there). For this, we construct a standard quadratic cost function over a prediction horizon of $N = 25$, with weights $Q = \text{diag}(100, 0.1, 500, 0.1)$, $R = 0.001$ and $Q_T = \text{diag}(1000, 10, 500, 10)$. We assume that the full state is measurable. The pendulum starts initially in the downward position. Considering the state dimension and the prediction horizon, the corresponding NMPC problem size becomes:

$$\Delta\mathbf{z} \in \mathbb{R}^{129}, \quad \mu \in \mathbb{R}^{204}, \text{ and } \lambda \in \mathbb{R}^{204}. \tag{11}$$

For the PS-SQP algorithm, we used 16 parallel QPs (i.e., $m = 16$). Each QP is then solved with `blkcnt=16` and `threadspblk=32`, which is sufficient to ensure full data parallelism. The simulation is executed on a workstation with an NVIDIA RTX 3070 Ti GPU, which has 6144 CUDA Cores and a boost clock frequency of 1.77GHz. The results of the NMPC closed–loop response are shown in Fig. 4. Here we observe that the PS-SQP algorithm achieved the desired control goal. Furthermore, the non–smooth behavior of the input visible when the system is almost settled is likely caused by the low tolerances of the SQP algorithm, chosen to favor fast convergence.

### A. QP-level parallelism

See Figure 5 for the results of the QP kernel execution time for different numbers of parallel QPs ($m$ as explained in Subsection II-A), in comparison to a sequential QP kernel execution. We configured the QP solver kernel with `blkcnt = 16` and `threadspblk = 32`, which provided the problem size in (11) enables full data parallelism. Figure 5 also shows the time needed for host-to-device and device-to-host communication, where the latter is negligible compared to the kernel execution time. Indeed, as the entire problem has to be passed to the GPU during the host-to-device communication, this takes noticeably longer and exceeds the execution time for a single QP for increasing $m$. Between a

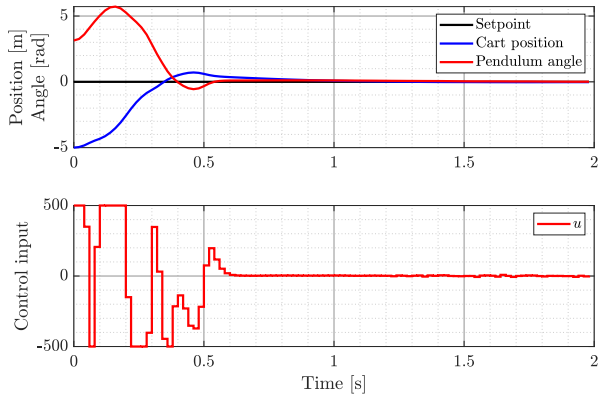


Fig. 4. NMPC closed loop response

single QP and 150 parallel QPs, the execution time increases by a factor of about 11 (considering the `blkcnt=16` graph in Fig. 5).

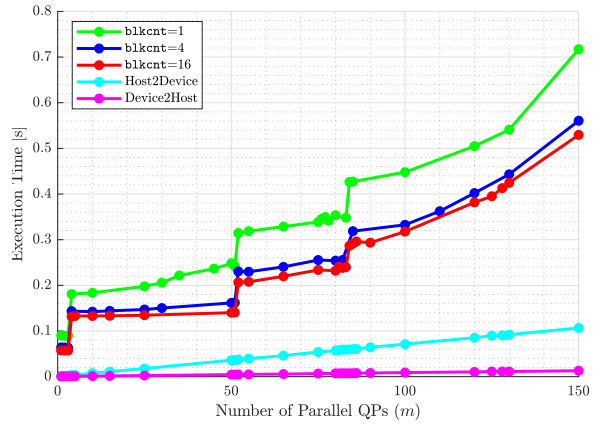


Fig. 5. GPU time per number of parallel QPs solved (`threadspblk=32`)

### B. Parallelism at a QP kernel

To play with the degree of parallelism, we have two parameters, `blkcnt` and `threadspblk`. Next, we test the effect on the computation time when considering different values for `blkcnt`, while fixing `threadspblk=32` (i.e., one warp) for all cases. Here we see that the execution is indeed parallel, noticeable by the lack of time increase between 5 and 50 parallel QPs. Figure 5 also shows a couple of odd jumps. While uncertain of the cause, it can result from the slow global memory management in relation to the occupancy of the warps on the GPU. Although the timing differences between `blkcnt=4` and `blkcnt=16` are minor, using `blkcnt=1` shows a significant execution time increase. Furthermore, while `blkcnt=1` uses only one thread block for data parallelism, it shows no comparable improvement when a large number of parallel QPs are employed.

## VI. Conclusions and Future Work

We presented an implementation with multilevel parallelism for solving many QPs on a GPU. Our implementation shows that parallelism greatly improves the timing efficiency and scalability, compared to running the same QPs sequentially. The presented GPU implementation considers global memory access for simplicity, which is time–inefficient. In our future work, we aim to introduce advanced memory management exploiting different levels of cache to improve the efficiency of a single QP execution.

## References


[1] Y. Wang, V. Puig, and G. Cembrano, "Non–linear economic model predictive control of water distribution networks," *Journal of Process Control*, vol. 56, pp. 23–34, 2017.

[2] M. Diehl, H. J. Ferreau, and N. Haverbeke, "Efficient Numerical Methods for Nonlinear MPC and Moving Horizon Estimation," *Int. Workshop on Assess. and Future Directions of NMPC*, 2008.

[3] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York: Springer Series in Op. Res. and Fin. Eng., 2006.

[4] S. Gros, M. Zanon, R. Quirynen, A. Bemporad, and M. Diehl, "From linear to nonlinear MPC: bridging the gap via the real-time iteration," *Int. Journal of Control*, vol. 93, pp. 1–19, 2016.

[5] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An Operator Splitting Solver for Quadratic Programs," *arXiv:1711.08013*, 2020.

[6] H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl, "qpOASES: a parametric active-set algorithm for quadratic programming," *Mathematical Programming Computation*, vol. 6, no. 4, pp. 327–363, 2014.

[7] E. Smith, J. Gondzio, and J. A. J. Hall, "Gpu acceleration of the matrix-free interior point method," School of Mathematics and Maxwell Institute for Mathematical Sciences, The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom, Tech. Rep. Technical Report ERGO-11-008†, May 2011.

[8] M. Schubiger, G. Banjac, and J. Lygeros, "Gpu acceleration of admm for large-scale quadratic programming," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 55–67, 2020.

[9] J. V. Frasch, S. Sager, and M. Diehl, "A parallel quadratic programming method for dynamic optimization problems," *Mathematical Programming Computation*, vol. 7, pp. 289–329, 2015.

[10] X. Hu, C. C. Douglas, R. Lumley, and M. Seo, "Gpu accelerated sequential quadratic programming," in *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*, 2017, pp. 3–6.

[11] M. Brand, V. Shilpiekandula, C. Yao, S. A. Bortoff, T. Nishiyama, S. Yoshikawa, and T. Iwasaki, "A parallel quadratic programming algorithm for model predictive control," *IFAC Proceedings Volumes*, vol. 44, pp. 1031–1039, January 1 2011.

[12] J. L. Jerez, G. A. Constantinides, E. C. Kerrigan, and K.-V. Ling, "Parallel mpc for real-time fpga-based implementation," *IFAC Proceedings Volumes*, vol. 44, no. 1, pp. 1338–1343, 2011, 18th IFAC World Congress.

[13] Y. Huang, K. V. Ling, and S. See, "Solving quadratic programming problems on graphics processing unit," *ASEAN Eng. Journal*, pp. 76–86, 12 2010.

[14] P. C. N. Verheijen, M. Haghi, M. Lazar, and D. Goswami, "Parallel Shooting Sequential Quadratic Programming for Nonlinear MPC Problems," *2023 IEEE Conference on Control Technology and Applications (CCTA)*, pp. 605–611, 2023.

[15] L. Prasad, B. Tyagi, and H. Gupta, "Optimal control of nonlinear inverted pendulum system using pid controller and lqr: Performance analysis without and with disturbance input," *Int. J. Autom. Comput*, vol. 11, p. 661–670, 2014.