Modeling and Optimization of Control Problems on GPUs

Response to reviewers

We thank the reviewers for their careful reading of our manuscript and their insightful and constructive comments. Our responses are summarized below. Due to time constraints, we did not re-run numerical experiments in the three days available for the rebuttal; however, we outline the modifications that will be incorporated in the camera-ready version of the paper.

Reviewer 1 comments

(i) – Reasons to Accept and Areas of Improvement

(a) Interest of frameworks (in the sense of toolchains or workflows) to accelerate the solution of control problems on GPUs.

(b) Demonstration of benefits of exploiting GPUs.

(c) Interest of promoting Julia as an alternative scientific programming language.

(d) Very little details on the framework itself. Description of the components, but these are published elsewhere and are not a contribution of the work.

(e) No comparison with related work.

(ii) – Detailed feedback for the authors

(a) The paper lacks details on how the various components of the workflow are integrated. The description primarily refers to previously published components, which cannot be considered original contributions of this work. Simply combining these existing elements does not constitute a substantial scientific contribution; rather, it represents an engineering effort.

(b) The experimental evaluation is weak. No comparison is made with other alternatives. The authors simply execute their codes on CPU and GPU and compare the execution times.

(iii) – Questions for Rebuttal

(a) What is the scientific contribution of your work?

(b) What is the novelty of this work?

(iv) – Response of the authors

> - Regarding both scientific contribution and novelty: to the best of our knowledge, as of today `OptimalControl.jl` is the only solver for optimal control problems running on CPU and GPU. There might be alternatives using some older solvers (such as `CasADi` which is a remarkable piece of work, in particular regarding automatic differentiation) and adding some hand made tweaks to interface them

with optimizers running on GPU (which one?), but this seems to be far from being easily accessible to the standard user. We could only find one reference [1], now added to the bibliography, called "CusADi" that showcases such an attempt using a mix of Matlab and Python (with PyTorch) (No maintained software seems to be available). Plugging together our software, that previously was only able to use `ADNLPModels.jl` [4] as optimization modeller, with `ExaModels.jl` necessitated some intensive metaprogramming: the original high level DSL has entirely been compiled into an ExaModels.ExaModel description of the discretised problem, allowing to call the compatible solver `MadNLP.jl` that indeed runs on GPU. This is definitely not "combining existing elements", which we have now clarified in the text. A byproduct of that is also faster runs even on CPU, using the modeller-solver pair ExaModels + MadNLP instead of ADNLP + Ipopt, both with the MUMPS linear solver. These features were added to OptimalControl.jl v1.1.

- The motivation of the paper was to illustrate the possibility to solve on GPU such optimal control problems, not to provide a worked out benchmark which, of course, would require a larger collection of problems. (This collection of problems is currently being built within the package `OptimalControlProblems.jl` and detailed benchmarking will be published somewhere else in time.) The two chosen problems are actually significant as they are rich enough to exhibit important peculiarities of optimal control problems, which we tried to emphasize at the beginning of Section 7 (stating that the solution of the Goddard case, for instance, is complex with four different kind of subarcs). Moreover, the results obtained on these two examples (and on two different CPU / GPU) were clearly illustrative of the gain obtained when solving on GPU vs. CPU: for large enough (typically $N = 1000$) grid sizes, GPU does better (and keeps doing so for larger grid), with a typical speedup of magnitude 2 to 10. Our aim being the comparison between CPU and GPU for direct optimal control solvers (= that resort to discretizing the control problem into a mathematic program), the only available possibility in Julia is indeed `MadNLP.jl` on a model based on `ExaModels.jl`.

Reviewer 2 comments

(i) – Reasons to Accept and Areas of Improvement

(a) The paper addresses an important unmet need by introducing a GPU-first solution for nonlinear optimal control problems. This integrated approach is interesting and closes a clear gap in current tools. The proposed GPU pipeline achieves notable speed-ups on large-scale benchmarks – e.g. about 2× faster on a complex rocket control problem, and up to 5× faster on a quadrotor problem, once the problem size is sufficiently large. For the evaluation part, authors compare GPU vs. CPU performance across varying problem sizes, showing where GPU acceleration pays off and confirming that the GPU solver converges to the same solutions with a similar number of iterations as the CPU solver. Building on Julia DSLs and automatic code generation, it is a good combination of software and HPC, making advanced GPU optimization accessible to domain specialists. The framework has practical value for scientists and engineers who need high-level

modeling capability with GPU performance. The use of Julia's DSL makes the system accessible without requiring low-level CUDA programming.

(b) The core contribution is an engineering effort rather than a fundamentally new algorithm. The paper builds on existing methods (direct transcription, interior-point solving) and known Julia packages. The solution currently relies on NVIDIA-specific technologies (CUDA and cuDSS). As the authors note, the sparse linear solver is CUDA-specific, meaning the approach cannot yet run on AMD or other GPU platforms. This limits portability and adoption in heterogeneous environments. It would enhance the work to at least discuss plans for supporting alternative backends or demonstrate that the framework is designed to accommodate other GPUs when corresponding libraries become available. (The paper does mention the design is amenable to future backends and even multi-GPU extensions, which is good, but currently the scope is restricted to NVIDIA GPUs.) Additionally, the authors could elaborate slightly more on implementation details such as JIT compilation overhead or memory usage, to preempt questions about the practicality of the approach in different settings.

(ii) – Detailed feedback for the authors

(a) The experiments are thorough and show tangible performance gains (2×–5×) for large-scale control problems. The performance evaluation is systematic, and the authors correctly highlight where GPU acceleration becomes beneficial (larger-scale problems). The framework has practical value for scientists and engineers who need high-level modeling capability with GPU performance. The use of Julia's DSL makes the system accessible without requiring low-level CUDA programming.

(b) Consider clarifying why existing GPU acceleration approaches (e.g., GPU kernels in CasADi or TensorFlow-based differentiable control) are insufficient or fundamentally different. This will better justify the uniqueness of your work. It would also help to explain why performance gains appear only after a certain problem size threshold, perhaps through a scaling analysis or profiling insight. Since the current implementation depends on NVIDIA's CUDA stack and cuDSS, it would strengthen the paper to discuss how this design could be extended to other GPU platforms (AMD ROCm, Intel GPU, etc.).

(iii) – Questions for Rebuttal

Please see detailed feedback.

(iv) – Response of the authors

- To the best of our knowledge, there is no other established and well distributed optimal control solver on GPU. We only found an experimental attempt to combine CasADi API with Pytorch using a mix of Matlab and Python (plus *ad hoc* compilations), named CusADi, that we now cite [1].
- We added sentences describing GPU backend support in our framework. We have

officially added support for AMD GPUs: sparse linear systems are simply converted to dense ones and solved with rocSOLVER. We are currently working on similar support for Intel GPUs with oneMKL.

- Regarding JIT compilation, we added a precompilation setup in the optimization package `MadNLP.jl`, which should remove the JIT overhead. This precompilation workflow has not yet been integrated into `OptimalControl.jl`. We added sentences in the paper to clarify this difference compared to compiled-code execution.
- The memory usage is quite similar to what can be achieved in low-level languages such as C or Fortran, since Julia provides fine-grained control over memory reuse and allocation. However, the main overhead comes from the garbage collector, which we do not force for performance reasons, but which can put pressure on the GPU. A sentence describing these points will be added to the paper.

Reviewer 3 comments

(i) – Reasons to Accept and Areas of Improvement

(a) This paper introduces a Julia module for optimal control problems on GPU with performance portability.

(b) Insufficient performance analysis.

(c) Insufficient evaluation to claim the conclusion.

(ii) – Detailed feedback for the authors

(a) Detailed performance analysis is not presented. For example, what is the theoretical performance bottleneck (e.g. memory bandwidth or compute) in this problem? And, how much efficiency does this Julia tool achieve?

(b) The authors mentioned in the "8. Discussion" section as "Julia's combination ... performant yet expressive tools for optimal control"; however, they didn't demonstrate how much performance we can achieve in other tools or programming models/languages. In other words, I couldn't understand how performance-efficient.

(iii) – Questions for Rebuttal

I believe that an effective rebuttal would not be able to address the fundamental limitations of this work.

(iv) – Response of the authors

- Our aim is to illustrate the benefit of GPU solving over CPU, not to provide a worked out benchmark which would necessitate a much larger collection of test problems (this is being built in `OptimalControlProblems.jl` actually, and will open the way for other tests, not only on GPU vs. CPU). The two proposed

examples exactly do that by providing the typical GPU over CPU speedup for large enough gridsizes. Currently, the only available second-order optimization solver in Julia that runs on GPU is `MadNLP.jl` (combined with an ExaModels.jl model for the automatic differentiation). We could of course add comparison with CPU runs of other solvers, *e.g.* `Ipopt`, `KNITRO`, `Uno` on hand made JuMP discretizations (we actually have made these tests), but the added value for the paper would be close to zero: performances on CPU for `JuMP` + `Ipopt` are more or less the same as for `ExaModels` + `MadNLP`. (Actually the pair `ExaModels` + `MadNLP` often does slightly better, but not significantly.) We have added a comment Section 7 to clarify this point.

- We plan to provide a breakdown of the time spent in each stage of our workflow (model assembly, differentiation, linear algebra, other operations in the optimization solver) to support our performance analysis and strengthen the results presented in our conclusions.

Reviewer 4 comments

(i) – Reasons to Accept and Areas of Improvement

(a) Open source, well-thought-out, and documented Julia libraries made to be shared and used by the community.

(b) Leveraging HPC throughput with easy-to-use Julia syntax.

(c) Convincing performance analysis on the latest HPC NVIDIA GPUs.

(d) Pedagogical and informative article.

(e) The extent of the software contributions associated with this article is not stated clearly.

(f) Additional comments on the performance analysis are needed.

(ii) – Detailed feedback for the authors

General comment

In "Modeling and Optimization of Control Problems on GPUs", the authors present an easy-to-use software stack for the efficient solution of optimal control problems on GPUs. The first part of the article describes the different components of their toolchain, and explains in a tutorial fashion how this toolchain can be used. The second part of this article concerns the use of their software to solve two optimal control problems (of increasing sizes) on NVIDIA A100 and H100 GPUs.

Overall, I enjoyed reading the article. It is setting a nice milestone for a set of work spanning different Julia libraries, and that comes together in an efficient and easy-to-use framework for the benefit of the community. For this reason, I think this article will interest the public of the SIAM PP conference, and I recommend it for publication. I have, however, a set of queries and remarks that I would like the authors to address.

Various queries and remarks

(a) I am not sure to understand the contribution claim "Our work fills this gap with a GPU-first toolchain that unifies modeling, differentiation, and solver execution" in section 2. It can sound like "all the associated software of the toolchain are contributions of this article"? Maybe the authors can clarify which software developments and efforts are specific contributions of this article.

(b) I did not find the model and spec of the CPU used for the experiments in the article. Can you add it?

(c) Can you clarify which floating-point arithmetic you use to perform the experiments?

(d) Can you comment on why H100 needs more iterations than the A100 to beat CPU time? Also, can you explain why we do not see a significant performance improvement when using H100 over A100?

(e) Why did you pick the direct solver CuDSS over a GPU-capable iterative solver (e.g., Krylov.jl)? Is it simply for convenience, or are there mathematical/numerical reasons pushing for this choice?

(f) I found the Figures hard to read and not very convenient for comparing the runs and the hardware. For instance, in Fig. 7.4, it is difficult to even read the runtime's order of magnitude. The gray grids in the plots' backgrounds are too light and won't print well. I realize afterward that Tables with the detailed execution time are available in the appendix. But they are not advertised in the main text of the article and are hard to find. I think that using a big Table spanning the two columns of the article instead of the current plots would be preferable for instance.

(iii) – Questions for Rebuttal

Nothing

(iv) – Response of the authors

- The software dev associated with this article is more than a mere combination of existing tools (namely modeler + optimisation solver + linear algebra). In order to maintain a high-level (= close to the math) modeling at the optimal control level (which is one the thing we believe is very important for most users-we do already have feedback on that, and on the performance), we need to parse `OptimalControl.jl` DSL and compile it into an `ExaModels.jl` model, tailored for optimization on GPU with `MadNLP.jl`. This is done through an intensive use of metaprogramming in Julia to capture the description and do a syntactic + semantic pass to generate the code. We have a provided a few details on the generated code in Section 5 to illustrate the translation into GPU friendly modeling thanks to `ExaModels.jl`, but did not give any specific information on the translation process itself (metaprogramming + pattern matching + one single pass treating both syntactic and semantic) as we believe it does not belong exactly to the scope

of the conference.

- The ticks on the four figures have now been updated and should be more readable.
- Regarding plots vs. table(s): we believe it is important to be able to see graphically that CPU and GPU curves indeed cross around $N = 1e3$. Of course, we can use a large table spanning the two columns in the final version.
- We will add the CPU model used for the numerical results.
- The cluster with H100 has a slower CPU than the one with A100. This difference is not related to the GPU model.
- We do not observe a significant improvement when using H100 over A100 because, in the nonlinear optimization solver, most of the computational work is spent solving the sparse KKT systems. The main bottleneck for speed-up is memory communication rather than computation itself. More precisely, the comparison depends primarily on the memory architecture of each GPU, such as the HBM generation, bandwidth, on-chip shared memory, and interconnect topology, rather than on raw FLOPS or peak compute throughput. We plan to explain this in Section 7, which is dedicated to benchmarks. We will also show that once the memory cache is exceeded and the computation enters a regime dominated by memory bandwidth and transfer, the maximum speed-up is limited to around 10x due to the fundamental differences in RAM and VRAM architectures between CPUs and GPUs.
- We use a direct solver instead of Krylov solvers because the optimization solver implemented in `MadNLP.jl` [8, 9] is a primal-dual interior point method. The KKT systems are known to be very hard to solve with Krylov methods due to the clustering of eigenvalues, which is caused by some terms in the KKT systems that approach zero or infinity. Furthermore, we currently lack a robust, problem-independent preconditioner. Note that we have developed a hybrid KKT solver in `HybridKKT.jl` [7], which combines conjugate gradient with Cholesky from cuDSS to solve generic optimization problems, but a direct solver is still required for robustness. The only approach where we fully rely on Krylov solvers on GPU is in `CompressedSensingIPM.jl` [2], where we have a dedicated preconditioner and cannot materialize the KKT systems because they are based on FFT operators and are dense Vandermonde matrices. We added sentences to explain that Krylov solvers are not used in this particular context. However, for other projects, `Krylov.jl` [3] is used with `MadNLP.jl` to solve optimization problems on NVIDIA and AMD GPUs.

Reviewer 5 comments

(i) – Reasons to Accept and Areas of Improvement

This paper presents a GPU-accelerated, Julia-based workflow for solving large-scale sparse nonlinear optimal control problems (OCPs). While the work demonstrates solid engineering

and effective implementation, it offers limited novelty from a research perspective, despite the value of parallelizing optimal control problems on GPUs.

(ii) – Detailed feedback for the authors

(a) Provide more details about the algorithm, particularly which components can be parallelized and which cannot.

(b) Include additional experiments on different GPUs, such as an analysis of the time consumption across various stages of the algorithm.

(iii) – Questions for Rebuttal

What is the time breakdown across the different stages of the algorithm?

(iv) – Response of the authors

- A sentence to clarify that the main computational effort in the optimization solver lies in solving sparse KKT systems on the GPU and performing automatic differentiation to evaluate sparse Jacobians and Hessians, and that the former is harder to parallelize than the latter.
- We plan to add a table in the numerical experiments section showing the time spent in the four stages of the algorithm for our test problems, along with a sentence specifying which of these stages are GPU-friendly.

Other changes

We made other changes to the manuscript while it was in review.

- Added references to our newly supported GPU optimization solvers: `MadNCL.jl` [5], and `MadIPM.jl` [6], and used them as examples of the modularity in our workflow.
- Outlined upcoming work and research questions arising from the connection between these packages, such as solving batch optimization problems on GPU with varying initial conditions in optimal control problems.

A PDF diff file generated with LaTeX, showing replaced text in red and new text in blue, is also provided.

References

[1] S. H. Jeon, S. Hong, H. J. Lee, C. Khazoom, and S. Kim. CusADi: A GPU Parallelization Framework for Symbolic Expressions and Optimal Control, 2024.

[2] W. Kuang, A. Montoison, V. Rao, F. Pacaud, and M. Anitescu. Recovering sparse DFT from missing signals via interior point method on GPU. *arXiv preprint arXiv:2502.04217*, 2025.

[3] A. Montoison and D. Orban. Krylov.jl: A Julia basket of hand-picked Krylov methods. *Journal of Open Source Software*, 8(89):5187, 2023.

[4] A. Montoison, T. Migot, D. Orban, and A. S. Siqueira. ADNLPModels.jl: Automatic differentiation models implementing the nlpmodels api, 2021.

[5] A. Montoison, F. Pacaud, M. Saunders, S. Shin, and D. Orban. MadNCL: A GPU Implementation of Algorithm NCL for Large-Scale, Degenerate Nonlinear Programs. *arXiv preprint arXiv:2510.05885*, 2025.

[6] A. Montoison, F. Pacaud, S. Shin, and M. Anitescu. GPU Implementation of Second-Order Linear and Nonlinear Programming Solvers, 2025.

[7] F. Pacaud, S. Shin, A. Montoison, M. Schanen, and M. Anitescu. Condensed-space methods for nonlinear programming on GPUs. *arXiv preprint arXiv:2405.14236*, 2024.

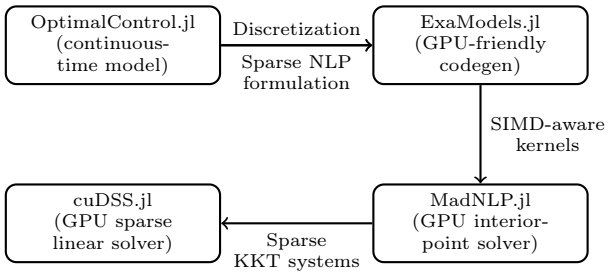
[8] S. Shin, C. Coffrin, K. Sundar, and V. M. Zavala. Graph-based modeling and decomposition of energy infrastructures. *IFAC-PapersOnLine*, 54(3):693–698, 2021.

[9] S. Shin, M. Anitescu, and F. Pacaud. Accelerating optimal power flow with GPUs: SIMD abstraction of nonlinear programs and condensed-space interior-point methods. *Electric Power Systems Research*, 236:110651, 2024.

Modeling and Optimization of Control Problems on GPUs

Abstract. We present a fully Julia-based, GPU-accelerated workflow for solving large-scale sparse nonlinear optimal control problems. Continuous-time dynamics are modeled and then discretized via direct transcription with `OptimalControl.jl` into structured sparse nonlinear programs. The direct transcription of ODE dynamics produces block-separable constraints with identical computational patterns across time steps. This regularity aligns naturally with SIMD execution on GPUs, enabling efficient batched evaluation of constraint and derivative computations. These programs are compiled into GPU kernels using `ExaModels.jl` ~~, leveraging SIMD parallelism~~ for fast evaluation of objectives, ~~constraints,~~ gradients, Jacobians, and Hessians. The resulting sparse problems are solved entirely on GPU using the interior-point solver `MadNLP.jl` ~~and~~ together with the GPU sparse linear solver cuDSS, yielding significant speed-ups over CPU-based approaches.

1 Introduction Solving large-scale nonlinear optimal control problems is computationally demanding, especially with fine discretizations or real-time requirements. While GPUs offer massive parallelism well-suited to these problems, fully exploiting their potential remains challenging due to the complexity of modeling, automatic differentiation, and solver integration. We present a fully GPU-accelerated workflow, entirely built in Julia [5]. Continuous-time dynamics are discretized with `OptimalControl.jl` [9] into structured, sparse nonlinear programs. These are compiled with `ExaModels.jl` [27] into GPU kernels that preserve sparsity and compute derivatives in a single pass, enabling efficient SIMD parallelism. Problems are solved on NVIDIA GPUs using the interior-point solver `MadNLP.jl` [28] and the sparse linear solver `CUDSS.jl` [16], enabling end-to-end acceleration from modeling to solving. We focus on NVIDIA hardware because efficient sparse KKT factorization is currently available only through cuDSS. Although our framework already runs on AMD and Intel GPUs, these backends rely on dense linear algebra for KKT solves and therefore serve mainly as proof-of-concept implementations. While Krylov methods from `Krylov.jl` [18] can be used on NVIDIA, AMD, and Intel GPUs, the KKT systems arising in the context of interior-point methods are notoriously difficult for Krylov solvers due to eigenvalue clustering in the system matrices, caused by terms approaching zero or infinity. Furthermore, a robust, problem-independent preconditioner for GPUs is currently lacking, which limits the practical use of iterative solvers in this setting. We demonstrate the performance of this approach on benchmark problems solved on NVIDIA A100 and H100 GPUs.



2 Background and limitations Optimal control problems (OCPs) aim to find control inputs for dynamical systems modeled by ODEs that optimize a given performance criterion. Direct transcription methods discretize these infinite-dimensional problems into large-scale nonlinear programs (NLPs). These NLPs exhibit a sparse structure arising from time discretization: each node introduces state and control variables linked by nonlinear equality constraints enforcing the system dynamics. Second-order methods, such as interior-point solvers, exploit this structure. Most existing optimal control toolchains target CPU execution. For example, CasADi [2] constructs symbolic expressions evaluated just-in-time or exported as C code, typically solved by CPU solvers like ~~IPOPT~~Ipopt [30], Uno [29], or KNITRO [7], which rely on CPU linear solvers such as PARDISO [26], MUMPS [1], or HSL [12]. Other frameworks, such as ACADO [13] and `InfiniteOpt.jl` [25], which cleverly leverage the modeling power of JuMP [11], also follow the same CPU-centric paradigm. This CPU focus limits scalability and real-time performance for large or time-critical problems that could benefit from GPU parallelism. While some libraries provide GPU-accelerated components, none deliver a fully integrated, GPU-native workflow for nonlinear optimal control. See, nonetheless, the nice attempt [14] trying to combine

the CasADi API with PyTorch so as to evaluate part of the generated code on GPU. Our work fills this gap with a GPU-first toolchain that unifies modeling, differentiation, and solver execution, addressing the challenges of solving large-scale sparse NLPs.

3 SIMD parallelism in direct optimal control When discretized by *direct transcription*, optimal control problems (OCPs) possess an inherent structure that naturally supports SIMD parallelism. Consider indeed an optimal control with state $x(t) \in \mathbf{R}^n$, and control $u(t) \in \mathbf{R}^m$. Assume that the dynamics is modeled by the ODE

$$\dot{x}(t) = f(x(t), u(t)),$$

where $f : \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}^n$ is a smooth function. Using a one-step numerical scheme to discretise this ODE on a time grid $t_0, t_1, \ldots, t_N$ of size $N+1$ results in a set of equality constraints. For instance, with a forward Euler scheme, denoting $h_i := t_{i+1} - t_i$, ~~one has (~~$X_i \simeq x(t_i)$, $U_i \simeq u(t_i)$~~)~~, we have

$$X_{i+1} - X_i - h_i f(X_i, U_i) = 0, \quad i = 0, \ldots, N-1.$$

Similarly, a general Bolza cost that mixes endpoint and integral terms as in

$$g(x(0t_0), x(t_f)) + \int_{0t_0}^{t_f} f^0(x(t), u(t))\,\mathrm{d}t \to \min$$

can be approximated by

$$g(X_0, X_N) + \sum_{i=0}^{N-1} h_i f^0(X_i, U_i).$$

Discretising boundary or path constraints such as

$$b\big(x(0t_0), x(t_f)\big) \le 0, \quad c\big(x(t), u(t)\big) \le 0$$

is obviously done according to

$$b(X_0, X_N) \le 0, \quad c(X_i, U_i) \le 0, \quad i = 0, \ldots, N-1.$$

The resulting NLP in the vector of unknowns $(X_0, \ldots, X_N, U_0, \ldots, U_{N-1})$ so involves only a few functions (*kernels*), namely f, f^0, g, b and c, that need to be evaluated on many state or control points, X_i, U_i. This massive SIMD parallelism allows for very efficient GPU-enabled computation. GPU acceleration thus facilitates real-time and large-scale optimal control computations critical, *e.g.*, to robotics and autonomous systems as in [23].

4 A Julia-based GPU optimization stack Julia offers a powerful and flexible environment for GPU programming, providing multiple levels of abstraction to suit different use cases. The package `CUDA.jl` [4, 3] provides direct access to NVIDIA GPUs, supporting easy array-based programming as well as explicit CUDA kernel writing and launching. For vendor-agnostic and portable GPU development, `KernelAbstractions.jl` [10] allows writing GPU kernels in Julia that target multiple GPU vendors (NVIDIA, AMD, Intel, Apple). This ecosystem leverages the LLVM compiler [15] and vendor APIs to generate efficient native GPU code directly from pure high-level Julia code. It allows users to exploit GPUs without requiring any knowledge of GPU programming. For instance, `ExaModels.jl` builds on `KernelAbstractions.jl` to automatically generate specialized GPU kernels for parallel evaluation of ODE residuals, Jacobians, and Hessians needed in optimal control problems. We build on this ecosystem to create a complete GPU-accelerated toolchain spanning modeling, automatic differentiation, and solving. This results into a fully Julia-native workflow for modeling and solving ODE-constrained optimal control problems on GPU. Key components of our stack include:

- `OptimalControl.jl`: a domain-specific language for symbolic specification of OCPs, supporting both direct and indirect formulations.
- `ExaModels.jl`: takes the discretized OCPs and produces sparse, SIMD-aware representations that preserve parallelism across grid points, compiling model expressions and their derivatives into optimized CPU/GPU code.
- `MadNLP.jl`: a nonlinear programming solver implementing a filter line-search interior-point method, with GPU-accelerated linear algebra support.
- `CUDSS.jl`: a Julia wrapper around NVIDIA's `cuDSS` sparse solver, enabling GPU-based sparse matrix factorizations essential for interior-point methods.

Together, these components form a high-level, performant stack that compiles intuitive Julia OCP models into efficient GPU code, achieving substantial speed-ups while maintaining usability.

Our approach offers several advantages:

- **Abstraction**: problems can be defined intuitively using Julia DSLs (Domain Specific Languages) without any requirement for GPU programming expertise.
- **Performance**: just-in-time compilation, SIMD parallelism, and GPU-accelerated sparse linear algebra provide substantial runtime improvements.

– **Portability**: symbolic modeling and kernel generation are backend-agnostic; the current limitation lies in sparse linear solvers, which are still CUDA-specific, but the framework is designed to integrate alternative backends as they become available.

One common concern in GPU-accelerated workflows is the overhead of just-in-time (JIT) compilation. In our stack, this overhead is minimal because we precompile all performance-critical code whenever possible using Julia 1.12 and tools like `PrecompileTools.jl`, thereby reducing warm-up latency. Only a small fraction of code that depends on dynamic types or runtime-specialized kernels is compiled just-in-time; this JIT overhead is negligible in practice due to the highly regular structure of GPU kernels generated from transcribed ODE constraints. The GPU ecosystem is also moving toward runtime PTX / JIT compilation (for example, NVIDIA CUDA 13.0 deprecates offline compilation for older architectures), which aligns with this strategy. Combined, this strategy allows our workflow to efficiently exploit GPUs for large-scale or real-time optimal control problems.

5 From optimal control models to SIMD abstraction To illustrate the transcription from the infinite dimensional setting towards a discretized optimization suited for SIMD parallelism, consider the following elementary optimal control problem with a state function, $x(t)$, valued in $\mathbf{R}^2$, and a scalar control, $u(t)$: minimize the (squared) L^2-norm of the control over the fixed time interval $[0, 1]$,

$$\frac{1}{2}\int_0^1 u^2(t)\,\mathrm{d}t \to \min,$$

under the dynamical constraint

$$\dot{x}_1(t) = x_2(t), \quad \dot{x}_2(t) = u(t),$$

and boundary conditions

$$x(0) = (-1, 0), \quad x(1) = (0, 0).$$

The strength of the DSL of the package `OptimalControl.jl` is to offer a syntax as close as possible to this mathematical formulation.[1] The translation of this optimal control problem so reads:

```
ocp = @def begin
    t in [0, 1], time
    x in R^2, state
    u in R, control
    x(0) == [-1, 0]
    x(1) == [0, 0]
    derivative(x1)(t) == x2(t)
    derivative(x2)(t) == u(t)
    integral( 0.5u(t)^2 ) => min
end
```

The intial and final times are fixed in this case but they could be additional unknowns (see, Appendix A.1, where the Goddard benchmark problem is modeled with a free final time. Users can also declare additional finite-dimensional parameters (or *variables*) to be optimized. Furthermore, extra constraints on the state, control, or other quantities can be imposed as needed. At this stage the crux is to seamlessly parse the abstract problem description and compile it on the fly into a discretized nonlinear optimization problem. We achieve this by exploiting two features. First, the DSL syntax is fully compatible with standard Julia, allowing us to use the language~~'~~s built-in lexical and syntactic parsers. Second, pattern matching via `MLStyle.jl` [31] extends Julia~~'~~s syntax with additional keywords such as `state` for declaring state variables, and implements the semantic pass that generates the corresponding discretized code. This discretized code ~~is~~ can now be represented as an `ExaModels.jl` model~~, which allows to~~. Previously, only the generic `ADNLPModels.jl` [17] was available, which requires more user effort and provides limited guidance for automatic differentiation, making it harder to generate highly efficient code on GPU. We have therefore added syntactic and semantic passes to generate `ExaModels.jl` models directly from the abstract problem description, allowing users to declare optimization variables (~~finite dimensional vector or~~ finite-dimensional arrays), constraints~~and cost.~~, and cost functions while providing the solver with detailed problem structure. Because `ExaModels.jl` builds on the `NLPModels.jl` abstraction [22], integrating support was straightforward. JuMP models can also be handled via `NLPModelsJuMP.jl` [19], which wraps JuMP problems to expose the `NLPModels.jl` interface. However, this backend is currently limited to CPU execution, highlighting `ExaModels.jl` as the preferred alternative for high-performance, GPU-enabled optimal control. Regarding constraints, `ExaModels.jl` uses *generators* in the form of `for` loop like statements to model the SIMD abstraction, ensuring that the function at the heart of the statement is mapped towards a *kernel* (this is where `KernelAbstractions.jl` comes into play) and efficiently evaluated by the solver. All in all, the process merely is a compilation from `OptimalControl.jl` DSL, well suited for mathematical control abstractions, into

[1] Note that one can actually use unicode characters to denote derivatives, integral, *etc.*, making this closeness even more striking. Check `OptimalControl.jl` documentation online.

ExaModels.jl DSL, tailored to describe optimization problems with strong SIMD potentialities. ~~(~~As explained in Section 3, this is indeed the case for discretizations of optimal control problems. ~~)~~ This transcription process is mostly parametrized by the numerical scheme used to discretize the ODE. A very important outcome of having a DSL for ExaModels.jl models is the ability for the package to automatically differentiate the mathematical expressions involved. Automatic differentiation (AD) is essential for modern second-order nonlinear solvers, such as ~~IPOPT~~ Ipopt and MadNLP.jl, which rely on first- and second-order derivatives.

Let us take a brief look at the generated code for this simple example. The code is wrapped in a function whose parameters capture the key aspects of the transcription process: the numerical scheme (here trapezoidal), the grid size (here uniform), the backend (CPU or GPU), the initial values for variables, states, and controls (defaulting to nonzero constants across the grid), and the base precision for vectors (defaulting to 64-bit floating point):

```
function def(; scheme=:trapeze, grid_size=250,
  backend=CPU(), init=(0.1, 0.1, 0.1),
  base_type=Float64)
```

The state declaration is compiled into an ExaModels.jl variable representing a $2 \times (N+1)$ array, where N is the grid size. Lower and upper bounds, plus initial values can be specified, and constraints are vectorized across grid points. Internally, the DSL uses metaprogramming to generate unique variable names and ensure proper initialization, while any syntactic or semantic errors are caught and reported at runtime.

```
x = begin
  local ex
  try
    variable(var"p_ocp##266", 2, 0:grid_size;
      lvar = [var"l_x##271"[var"i##275"]
      for (var"i##275", var"j##276") =
      Base.product(1:2, 0:grid_size)],
      uvar = [var"u_x##272"[var"i##275"]
      for (var"i##275", var"j##276") =
      Base.product(1:2, 0:grid_size)],
      start = init[2])
    catch ex
      println("Line ", 2, ": ",
        "(x in R^2, state)")
      throw(ex)
  end
end
```

The initial-state boundary constraint must be applied across all state dimensions. This is achieved using the for generator. A runtime dimension check ensures that the specified bounds match the length of the state vector being addressed:

```
length([-1, 0]) == length([-1, 0]) ==
  length(1:2) || throw("wrong bound dimension")
constraint(var"p_ocp##266", (x[var"i##283", 0]
  for var"i##283" = 1:2); lcon = [-1, 0],
  ucon = [-1, 0])
```

The first equation of the ODE system is discretized using the trapezoidal scheme, and the corresponding expression (here the right hand side is just $x_2(t)$) is declared thanks to the for generator tailored for SIMD abstraction:

```
constraint(var"p_ocp##266j",
  ((x[1, var"j##291" + 1] - x[1, var"j##291"])
  - (var"dt##268" * (x[2, var"j##291"] +
  x[2, var"j##291" + 1])) / 2
  for var"j##291" = 0:grid_size - 1))
```

The same goes on for the second dimension of the ODE, and for the Lagrange integral cost as well, where the same numerical scheme (trapezoidal rule again) is employed for consistency (defining two objectives actually computes their sum):

```
objective(var"p_ocp##266", ((1 * var"dt##268" *
  (0.5 * var"u##277"[1, var"j##299"] ^ 2)) / 2
  for var"j##299" = (0, grid_size)))
objective(var"p_ocp##266", (1 * var"dt##268" *
  (0.5 * var"u##277"[1, var"j##299"] ^ 2)
  for var"j##299" = 1:grid_size - 1))
```

The generated code returns an ExaModels.jl model that, when instanciated with the proper backend (*e.g.*, CUDABackend() from CUDA.jl) can be passed to MadNLP.jl to be solved on GPU.

6 Orchestrating model derivatives and sparse GPU linear solvers Our workflow executes both differentiation and linear algebra entirely on the GPU. The interior-point solver MadNLP.jl serves as the central orchestrator: it calls derivative kernels generated by ExaModels.jl, assembles KKT systems directly in device buffers, and invokes CUDSS.jl for ~~factorization~~ sparse factorizations and triangular solves. Apart from a one-time symbolic analysis on the host, primarily reordering to reduce fill-in in the factors of the $\mathrm{LDL}^\top$ decomposition, all operations remain on the device. Robustness is further ensured by scaling, inertia-corrected primal–dual regularization, and optional iterative refinement. When GPU vendor libraries lack suitable routines, MadNLP.jl falls back on KernelAbstractions.jl to implement custom kernels.

Each interior-point iteration therefore stays entirely on the device: derivative evaluation, KKT assembly, sparse factorizations and triangular solves, and step updates. All of these components are orchestrated on GPU by MadNLP.jl.

7 Benchmark problems We evaluate our GPU-accelerated workflow on two challenging nonlinear optimal control problems:

- the Goddard rocket problem,
- a quadrotor control problem.

These problems have already been discretized using, for example, Euler or trapezoidal schemes, and can be found in standard optimization problem collections such as COPS [6], as well as in the Julia port `COPSBenchmark.jl` [24]. Their infinite dimensional (non discretized) counterparts are also available in the more recent open collection of control problems `OptimalControlProblems.jl` [8]. The full models using `OptimalControl.jl` are detailed in the supplementary material (check Appendix A.1). We compare GPU and CPU performance, to assess both raw speed-ups and the effect of sparsity on GPU d parallelism on solver performance. For the considered setups (see Appendix A.2 for details on the hardware and on the Julia packages used), as expected performance is better on H100 than A100, with comparable GPU over CPU speed-ups on both problems. Note that the aim of the paper being this CPU-GPU comparison, with `ExaModels.jl` + `MadNLP.jl` the currently only modeler-solver pair available to work on GPU, we deliberately do not provide comparisons on CPU with other such pairs (*e.g.*, JuMP + Ipopt).

On the Goddard problem, the total dimension of the discretized problem (variables plus constraints) is about $10N$, where N is the grid size. Although the problem is standard in the control literature, its solution exhibits an intricate structure for the choosen parameters with a control that is comprised of four subarcs (a concatenation of bang-singular-boundary-bang arcs). In particular, the existence of a singular arc is well known to be a source of difficulty for direct methods. Convergence towards the same objective (up to minor changes in precision depending on the number of points) is nonetheless obtained for all solves with the same (trivial) initialization, and the number of iterates remains very close on CPU and GPU for all the tested grid sizes. On the cluster with A100, GPU is faster than CPU after $N = 2000$, see Figure 7.1. On the cluster with H100, GPU beats CPU after $N = 5000$, see Figure 7.3. On both architectures, a speed-up about 2 is obtained. The largest problem solved on the H100 has size about two millions with a run time about 15 seconds.

On the Quadrotor problem, the total dimension of the discretized problem (variables plus constraints) is about $20N$, where N is the grid size. The particular instance of the problem is unconstrained (neither control nor state constraint, contrary to the previous Goddard example) but has a strongly nonlinear dynamics. As before, convergence is obtained for all architectures and grid sizes with comparable number of iterates between CPU and GPU. On the A100, GPU is faster than CPU after $N = 500$, see Figure 7.3. On the H100, GPU beats CPU after $N = 750$, see Figure 7.4. On both architectures, a speed-up about 5 is obtained. The largest problem solved on the H100 has size about $4e6$ with a run time about 13 seconds.

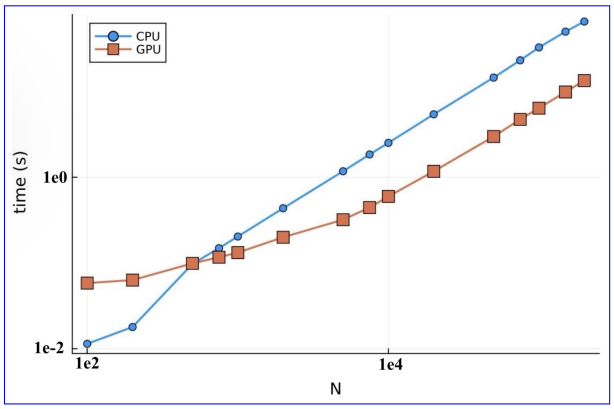


Figure 7.1: Goddard problem, A100 solve. The grid size ranges from to $N = 1e2$ to $N = 1e5$.

We note that the bulk of the computational work in our problems arises from two main components: the evaluation of sparse derivatives and the solution of the sparse KKT systems. The first component is highly GPU-friendly and does not constitute a bottleneck. The second component is more challenging due to the irregular sparsity patterns in the KKT systems and their factors, which makes memory communication the limiting factor. In the large-scale regime where the memory cache is saturated, the maximum achievable speed-up is limited to roughly 10×, corresponding to the ratio of memory bandwidth between RAM and GPU VRAM in current architectures.

8 Discussion Julia's combination of high-level abstractions, metaprogramming, and GPU compiler infrastructure makes it uniquely suited for building performant yet expressive tools for optimal control. Our results show that leveraging parallelism in both model structure and solver internals unlocks substantial speed-ups, enabling new applications in aerospace engineering, quantum control, computational biology, learning, and more.

The modular GPU stack also ensures portability across future architectures. In particular, we have added support for newly developed GPU optimization solvers, including `MadNCL.jl` [20] and `MadIPM.jl` [21], which

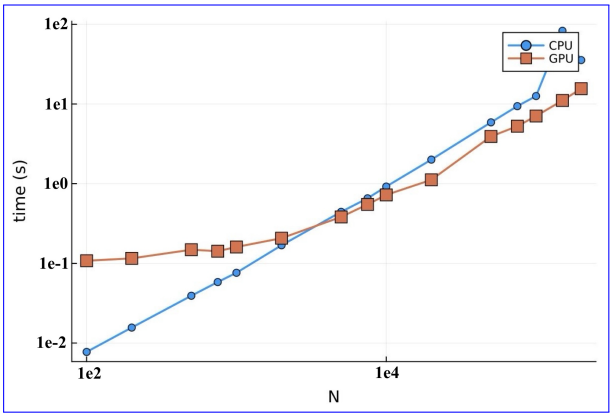


Figure 7.2: Goddard problem, H100 solve. The grid size ranges from to $N = 1e2$ to $N = 2e5$.

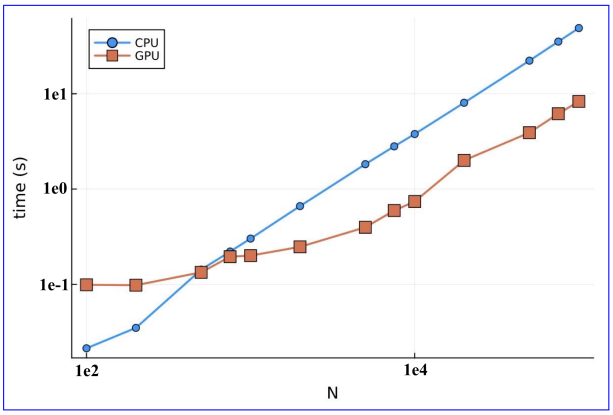


Figure 7.3: Quadrotor problem, A100 solve. The grid size ranges from to $N = 1e2$ to $N = 1e5$.

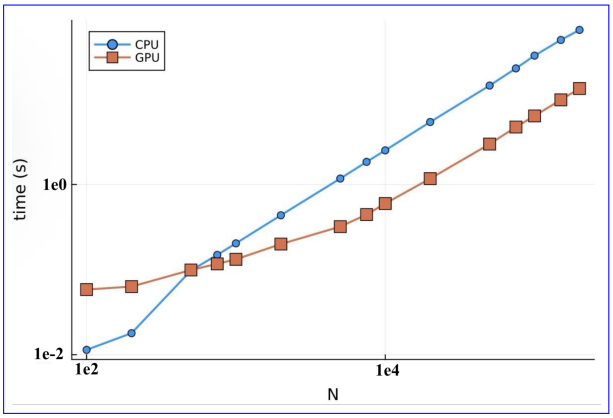


Figure 7.4: Quadrotor problem, H100 solve. The grid size ranges from to $N = 1e2$ to $N = 2e5$.

extend `MadNLP.jl` to efficiently solve LPs, convex QPs, and degenerate NLPs. This illustrates the modularity of our workflow and opens new research directions, such as batch optimization on GPUs with varying initial conditions in optimal control problems.

Future extensions include multi-GPU execution and tighter integration with differentiable programming workflows. Overall, the synergy between Julia GPU tools~~and the SIMD structure of~~, SIMD structure in direct optimal control, and the growing suite of solvers yields a powerful ~~solution~~ framework for solving challenging OCPs at scale.

References

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, *MUMPS: a general purpose distributed memory sparse solver*, in International Workshop on Applied Parallel Computing, Springer, 2000, pp. 121–130.

[2] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, *CasADi – A software framework for nonlinear optimization and optimal control*, Mathematical Programming Computation, 11 (2019), pp. 1–36, https://doi.org/10.1007/s12532-018-0139-4.

[3] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, *Rapid software prototyping for heterogeneous and distributed platforms*, Advances in Engineering Software, 132 (2019), pp. 29–46.

[4] T. Besard, C. Foket, and B. De Sutter, *Effective extensible programming: Unleashing Julia on GPUs*, IEEE Transactions on Parallel and Distributed Systems, (2018), https://doi.org/10.1109/TPDS.2018.2872064, https://arxiv.org/abs/1712.03112.

[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM review, 59 (2017), pp. 65–98.

[6] A. S. Bondarenko, D. M. Bortz, and J. Moré, *COPS: Large-scale nonlinearly constrained optimization problems*, tech. report, Argonne National Lab., IL (US), 2000.

[7] R. H. Byrd, J. Nocedal, and R. A. Waltz, *Knitro: An integrated package for nonlinear optimization*, Large-scale nonlinear optimization, (2006), pp. 35–59.

[8] J.-B. Caillau, O. Cots, J. Gergaud, and P. Martinon, *OptimalControlProblems.jl: a collection of optimal control problems with ODE's in Julia*, https://doi.org/10.5281/zenodo.17013180.

[9] J.-B. Caillau, O. Cots, J. Gergaud, P. Martinon, and S. Sed, *OptimalControl.jl: a Julia package to model and solve optimal control problems with ODE's*, https://doi.org/10.5281/zenodo.13336563.

[10] V. Churavy, *KernelAbstractions.jl*, github.com/JuliaGPU/KernelAbstractions.jl.

[11] I. Dunning, J. Huchette, and M. Lubin, *JuMP: A modeling language for mathematical optimization*, SIAM review, 59 (2017), pp. 295–320.

[12] J. Fowkes, A. Lister, A. Montoison, and D. Orban, *LibHSL: the ultimate collection for large-scale scien-tific computation*, Les Cahiers du GERAD ISSN, 711 (2024), p. 2440.

[13] B. Houska, H. J. Ferreau, and M. Diehl, *ACADO toolkit—An open-source framework for automatic control and dynamic optimization*, Optimal control applications and methods, 32 (2011), pp. 298–312.

[14] S. H. Jeon, S. Hong, H. J. Lee, C. Khazoom, and S. Kim, *CusADi: A GPU Parallelization Framework for Symbolic Expressions and Optimal Control*, 2024, https://arxiv.org/abs/2408.09662.

[15] C. Lattner and V. Adve, *LLVM: A compilation framework for lifelong program analysis & transformation*, in International symposium on code generation and optimization, 2004. CGO 2004., IEEE, 2004, pp. 75–86.

[16] A. Montoison, *CUDSS.jl: Julia interface for NVIDIA cuDSS*, github.com/exanauts/CUDSS.jl.

[17] A. Montoison, T. Migot, D. Orban, and A. S. Siqueira, *ADNLPModels.jl: Automatic differentiation models implementing the nlpmodels api*, 2021, https://doi.org/10.5281/zenodo.4605982.

[18] A. Montoison and D. Orban, *Krylov.jl: A Julia basket of hand-picked Krylov methods*, Journal of Open Source Software, 8 (2023), p. 5187, https://doi.org/10.21105/joss.05187.

[19] A. Montoison, D. Orban, and A. S. Siqueira, *NLPModelsJuMP.jl: Conversion from JuMP models to NLPModels*, 2020, https://doi.org/10.5281/zenodo.2574162.

[20] A. Montoison, F. Pacaud, M. Saunders, S. Shin, and D. Orban, *MadNCL: A GPU Implementation of Algorithm NCL for Large-Scale, Degenerate Nonlinear Programs*, arXiv preprint arXiv:2510.05885, (2025).

[21] A. Montoison, F. Pacaud, S. Shin, and M. Anitescu, *GPU Implementation of Second-Order Linear and Nonlinear Programming Solvers*, 2025, https://arxiv.org/abs/2508.16094, https://arxiv.org/abs/2508.16094.

[22] D. Orban and A. Soares Siqueira, *NLPModels.jl: Data Structures for Optimization Models*, 2023.

[23] F. Pacaud and S. Shin, *GPU-accelerated nonlinear model predictive control with ExaModels and MadNLP*, arXiv e-prints, (2024), pp. arXiv–2403.

[24] F. o. Pacaud and T. Migot, *COPSBenchmark.jl*, github.com/MadNLP/COPSBenchmark.jl.

[25] J. L. Pulsipher, W. Zhang, T. J. Hongisto, and V. M. Zavala, *A unifying modeling abstraction for infinite-dimensional optimization*, Computers and Chemical Engineering, 156 (2022), https://doi.org/doi.org/10.1016/j.compchemeng.2021.107567.

[26] O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with pardiso*, Future Generation Computer Systems, 20 (2004), pp. 475–487.

[27] S. Shin, M. Anitescu, and F. Pacaud, *Accelerating optimal power flow with GPUs: SIMD abstraction of nonlinear programs and condensed-space interior-point methods*, Electric Power Systems Research, 236 (2024), p. 110651.

[28] S. Shin, C. Coffrin, K. Sundar, and V. M. Zavala, *Graph-based modeling and decomposition of energy infrastructures*, IFAC-PapersOnLine, 54 (2021), pp. 693–698.

[29] C. Vanaret and S. Leyffer, *Unifying nonlinearly constrained nonconvex optimization*. Submitted to Mathematical Programming Computation, 2024.

[30] A. Wächter and L. T. Biegler, *On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming*, Mathematical programming, 106 (2006), pp. 25–57.

[31] T. Zhao, *MLStyle.jl*, thautwarm.github.io/MLStyle.jl.

A Supplementary material

A.1 Descriptions of the control problems used for the benchmark The complete code to reproduce the runs of Section 7 can be retrieved at the following address: https://anonymous.4open.science/r/OC-GPU-PP26

```
# Goddard problem

r0 = 1.0
v0 = 0.0
m0 = 1.0
vmax = 0.1
mf = 0.6
Cd = 310.0
Tmax = 3.5

beta = 500.0
b = 2.0

o = @def begin

    tf in R, variable
    t in [0, tf], time
    x = (r, v, m) in R^3, state
    u in R, control

    x(0) == [r0, v0, m0]
    m(tf) == mf
    0 <= u(t) <= 1
    r(t) >= r0
    0 <= v(t) <= vmax

    derivative(r)(t) == v(t)
    derivative(v)(t) == -Cd * v(t)^2 *
      exp(-beta * (r(t) - 1)) / m(t) - 1 /
      r(t)^2 + u(t) * Tmax / m(t)
    derivative(m)(t) == -b * Tmax * u(t)

    r(tf) => max

end

# Quadrotor problem

T = 1
g = 9.8
r = 0.1

o = @def begin

    t in [0, T], time
    x in R^9, state
    u in R^4, control

    x(0) == zeros(9)

    derivative(x1)(t) == x2(t)
    derivative(x2)(t) == u1(t) * cos(x7(t)) *
      sin(x8(t)) * cos(x9(t)) + u1(t) *
      sin(x7(t)) * sin(x9(t))
    derivative(x3)(t) == x4(t)
    derivative(x4)(t) == u1(t) * cos(x7(t)) *
      sin(x8(t)) * sin(x9(t)) - u1(t) *
      sin(x7(t)) * cos(x9(t))
    derivative(x5)(t) == x6(t)
    derivative(x6)(t) == u1(t) * cos(x7(t)) *
      cos(x8(t)) - g
    derivative(x7)(t) == u2(t) * cos(x7(t)) /
      cos(x8(t)) + u3(t) *
      sin(x7(t)) / cos(x8(t))
    derivative(x8)(t) ==-u2(t) * sin(x7(t)) +
      u3(t) * cos(x7(t))
    derivative(x9)(t) == u2(t) * cos(x7(t)) *
      tan(x8(t)) + u3(t) * sin(x7(t)) *
      tan(x8(t)) + u4(t)

    dt1 = sin(2pi * t / T)
    dt3 = 2sin(4pi * t / T)
    dt5 = 2t / T

    0.5integral( (x1(t) - dt1)^2 +
      (x3(t) - dt3)^2 + (x5(t) - dt5)^2 +
      x7(t)^2 + x8(t)^2 + x9(t)^2 + r *
      (u1(t)^2 + u2(t)^2 +
      u3(t)^2 + u4(t)^2) ) => min

end
```

A.2 GPU detailed configurations and results All runs performed with `OptimalControl.jl` v1.1.1, `MadNLPMumps.jl` v0.5.1 and `MadNLPGPU.jl` v0.7.7.

Configuration for the A100 runs

```
julia> CUDA.versioninfo()
CUDA runtime 12.9, artifact installation
CUDA driver 12.9
NVIDIA driver 575.57.8

CUDA libraries:
- CUBLAS: 12.9.1
- CURAND: 10.3.10
- CUFFT: 11.4.1
- CUSOLVER: 11.7.5
- CUSPARSE: 12.5.10
- CUPTI: 2025.2.1 (API 28.0.0)
- NVML: 12.0.0+575.57.8

Julia packages:
- CUDA: 5.8.2
- CUDA_Driver_jll: 0.13.1+0
- CUDA_Runtime_jll: 0.17.1+0

Toolchain:
- Julia: 1.11.6
- LLVM: 16.0.6
```

```
1 device:
  0: NVIDIA A100-PCIE-40GB
     (sm_80, 39.490 GiB / 40.000 GiB available)
```

Configuration for the H100 runs

```
julia> CUDA.versioninfo()
CUDA toolchain:
- runtime 12.9, artifact installation
- driver 580.65.6 for 13.0
- compiler 12.9

CUDA libraries:
- CUBLAS: 12.9.1
- CURAND: 10.3.10
- CUFFT: 11.4.1
- CUSOLVER: 11.7.5
- CUSPARSE: 12.5.10
- CUPTI: 2025.2.1 (API 12.9.1)
- NVML: 13.0.0+580.65.6

Julia packages:
- CUDA: 5.8.2
- CUDA_Driver_jll: 13.0.0+0
- CUDA_Compiler_jll: 0.2.0+2
- CUDA_Runtime_jll: 0.19.0+0

Toolchain:
- Julia: 1.11.6
- LLVM: 16.0.6

Preferences:
- CUDA_Runtime_jll.version: 12.9

1 device:
  0: NVIDIA H100 80GB HBM3
     (sm_90, 79.175 GiB / 79.647 GiB available)
```

Table A.1: Goddard problem, A100 run

Grid size (N)	CPU time (s)	GPU time (s)
100	1.46e-2	0.158
200	2.93e-2	0.174
500	7.16e-2	0.186
750	1.02e-1	0.202
1,000	1.34e-1	0.211
2,000	2.86e-1	0.254
5,000	8.61e-1	0.489
7,500	1.27	0.762
10,000	1.72	0.988
20,000	3.59	1.54
50,000	1.01e1	4.33
75,000	1.59e1	6.59
100,000	2.15e1	8.75

Table A.2: Quadrotor problem, A100 run

Grid size (N)	CPU time (s)	GPU time (s)
100	2.135e-2	0.099
200	3.498e-2	0.098
500	1.427e-1	0.134
750	2.210e-1	0.195
1,000	3.023e-1	0.200
2,000	6.624e-1	0.248
5,000	1.823	0.397
7,500	2.804	0.595
10,000	3.777	0.740
20,000	8.044	1.999
50,000	2.223e1	3.904
75,000	3.522e1	6.173
100,000	4.882e1	8.304

Table A.3: Goddard problem, H100 run

Grid size (N)	CPU time (s)	GPU time (s)
100	7.738e-3	0.108
200	1.564e-2	0.116
500	3.924e-2	0.148
750	5.826e-2	0.142
1,000	7.619e-2	0.161
2,000	1.687e-1	0.207
5,000	4.433e-1	0.382
7,500	6.548e-1	0.549
10,000	9.205e-1	0.723
20,000	2.004	1.118
50,000	5.884	3.915
75,000	9.390	5.262
100,000	1.258e1	7.065
150,000	8.293e1	11.069
200,000	3.556e1	15.575

Table A.4: Quadrotor problem, H100 run

Grid size (N)	CPU time (s)	GPU time (s)
100	1.141e-2	0.058
200	1.790e-2	0.063
500	9.748e-2	0.099
750	1.488e-1	0.117
1,000	2.034e-1	0.132
2,000	4.350e-1	0.199
5,000	1.174	0.320
7,500	1.842	0.443
10,000	2.516	0.597
20,000	5.420	1.171
50,000	1.453e1	2.983
75,000	2.311e1	4.728
100,000	3.267e1	6.396
150,000	4.986e1	9.873
200,000	6.546e1	13.413