Journal of Optimization Theory and Applications
https://doi.org/10.1007/s10957-022-02129-5

Accelerating Condensed Interior-Point Methods on SIMD/GPU Architectures

François Pacaud[1] · Sungho Shin[1] · Michel Schanen[1] · Daniel Adrian Maldonado[1] · Mihai Anitescu[1]

Received: 14 March 2022 / Accepted: 22 October 2022

Abstract
The interior-point method (IPM) has become the workhorse method for nonlinear programming. The performance of IPM is directly related to the linear solver employed to factorize the Karush–Kuhn–Tucker (KKT) system at each iteration of the algorithm. When solving large-scale nonlinear problems, state-of-the art IPM solvers rely on efficient sparse linear solvers to solve the KKT system. Instead, we propose a novel reduced-space IPM algorithm that condenses the KKT system into a dense matrix whose size is proportional to the number of degrees of freedom in the problem. Depending on where the reduction occurs, we derive two variants of the reduced-space method: linearize-then-reduce and reduce-then-linearize. We adapt their workflow so that the vast majority of computations are accelerated on GPUs. We provide extensive numerical results on the optimal power flow problem, comparing our GPU-accelerated reduced-space IPM with Knitro and a hybrid full-space IPM algorithm. By evaluating the derivatives on the GPU and solving the KKT system on the CPU, the hybrid solution is already significantly faster than the CPU-only solutions. The two reduced-space algorithms go one step further by solving the KKT system entirely on the GPU. As expected, the performance of the two reduction algorithms depends critically on the number of available degrees of freedom: They underperform the full-space method when the problem has many degrees of freedom, but the two algorithms are up to three times faster than Knitro as soon as the relative number of degrees of freedom becomes smaller.

Communicated by Florian Jarre.

Mihai Anitescu dedicates this work to the 70-th birthday of Florian Potra. Florian, thank you for the great contributions to optimization in general, and interior-point methods in particular, and for initiating me and many others in them.

✉ François Pacaud
fpacaud@anl.gov

[1] Argonne National Laboratory, Lemont, USA

Published online: 13 February 2023

1 Introduction

Most optimization problems in engineering consist of minimizing a cost subject to a set of equality constraints that represent the physics of the problem. Often, only a subset of the problem variables is actionable. A particular instance of these types of problems is the optimal power flow (OPF), which is formulated as a large-scale nonlinear nonconvex optimization problem. In electrical power system markets, operators solve the OPF problem multiple times per day [15]. The OPF problem finds the optimal real power of dispatchable resources (typically power plants) subject to physical constraints (the power flow constraints or power balance) and operational constraints (e.g., voltage or line flow limits) [6]. It is challenging to solve to optimality, particularly since its solution is needed within a prescribed and fairly tight time limit. Since the 1990s, the efficient solution of OPF has relied on the interior-point method (IPM). In particular, state-of-the-art numerical tools developed for sparse IPMs can efficiently handle the sparse structure of typical OPF problems. At each iteration of the IPM algorithm, the descent direction is computed by solving a Karush–Kuhn–Tucker (KKT) system, which requires the factorization of large-scale ill-conditioned unstructured symmetric indefinite matrices [26]. For that reason, current state-of-the-art OPF solvers combine a mature IPM solver [38, 39] together with a sparse Bunch–Kaufman factorization routine [11, 32]. Along with an efficient evaluation of the derivatives, this method is able to efficiently solve OPF problems with up to 200,000 buses on modern CPU architectures [22].

1.1 Interior-Point Method on GPU Accelerators: State of the Art

Most upcoming high-performance computing (HPC) architectures are GPU-centric, and we can leverage these new parallel architectures for solving very large OPF instances. However, porting IPM to the GPU is nontrivial because GPUs are based on a different programming paradigm from that of CPUs: Instead of computing a sequence of instructions on a single input (potentially dispatched on different threads or processes), GPUs run the same instruction simultaneously on hundreds of threads (SIMD paradigm: *Single Instruction, Multiple Data*). Hence, GPUs shine when the algorithm is decomposable into simple instructions running entirely in parallel where the same instruction is executed on different data in lockstep (as is the case for most dense linear algebra). In general, not all algorithms are fully amenable to this paradigm. One notorious example is branching in the control flow: When the instructions have multiple conditions, dispatching the operations on multiple threads makes execution in lockstep impossible.

Unfortunately, factorization of an unstructured sparse indefinite matrix is one of these edge cases: Unlike dense matrices, sparse matrices have unstructured sparsity, rendering most sparse algorithms difficult to parallelize. Thus, implementing a sparse direct solver on the GPU is nontrivial, and the performance of current GPU-based sparse linear solvers lags far behind that of their CPU equivalents [36, 37]. Previous attempts to solve nonlinear problems on the GPU have circumvented this problem by relying on iterative solvers [7, 33] or on decomposition methods [23]. Here, we have

chosen instead to revisit the original reduced-space algorithm proposed in [10]: this method *condenses* the KKT system into a dense matrix, whose size is small enough to be factorized efficiently on the GPU with dense direct linear algebra.

1.2 Reduced-Space Interior-Point Method

Reduced-space algorithms have been studied for a long time. In [10], the authors introduced a reduced-space algorithm, one of the first effective methods to solve the OPF problem. The method has several first-order variants, known as the generalized reduced-gradient algorithm [1] or the gradient projection method [30], whose theoretical implications are discussed in [16]. The extension of the reduced-space method to second-order comes later [31]—as well as its application to OPF [5]—the method becoming during the 1980s a particular case of the sequential quadratic programming algorithm [9, 13, 18, 25]. However, the (dense) reduced Hessian has always been challenging to form explicitly, favoring the development of approximation algorithms for second-order derivatives, based on quasi-Newton [3] or on Hessian vector products [4]. The reduced-space method was extended to IPM in the late 1990s [8] and was already adopted to solve OPF [20]. We refer to [21] for a recent report describing the application of reduced-space IPM to OPF.

1.3 Contributions

Our reduced-space algorithm is built on this extensive previous work. In Sect. 2, we propose a tractable reduced-space IPM algorithm, allowing the OPF problem to be solved entirely on the GPU. Instead of relying on a direct sparse solver, our reduced-space IPM *condenses* the KKT system into a dense linear system whose size depends only on the number of control variables in the reduced problem (here, the number of generators). When the number of degrees of freedom is a fraction of the total number of variables, the KKT system size can be dramatically reduced. In addition, we establish a formal connection between the reduced IPM algorithm and the seminal reduced-gradient method of Dommel and Tinney [10]. Depending on whether the reduction occurs at the KKT system level or directly at the nonlinear level, we propose two different reduced-space algorithms: *Linearize-then-reduce* and *Reduce-then-linearize*. We describe a parallel implementation of the reduction algorithm in Sect. 3 and show how we can exploit efficient linear algebra kernels on the GPU to accelerate the algorithm. We discuss an application of the proposed method to the OPF problem in Sect. 4: our numerical results show that both the reduced IPM and its feasible variant can solve large-scale OPF instances—with up to 70,000 buses—entirely on the GPU. This result improves on the previous results reported in [21, 27]. As expected, the reduced-space algorithm is competitive when the problem has fewer degrees of freedom. Still, it achieves respectable performance (within a factor of 3 compared with state-of-the-art methods) even on the less favorable instances. To the best of our knowledge, this is the first time a second-order GPU-based NLP solver matches the performance of state-of-the-art CPU-based solvers on the resolution of OPF problems.

2 Reduced Interior-Point Method

In Sect. 2.1, we introduce the problem formulation under consideration. In this problem, the independent variables (the *control*, associated with the problem's *degrees* of freedom) are split from the dependent variables (the *state*). The reduction is akin to a Schur complement reduction and can occur at the linear algebra or the nonlinear levels. In Sect. 2.2, we present our first method, *linearize-then-reduce*, performing the reduction directly on the KKT system. In Sect. 2.3, we show that we can reduce equivalently the nonlinear model using the implicit function theorem, giving our second method: *reduce-then-linearize*.

2.1 Formalism

The problem we study in this section has a particular structure: The optimization variables are divided into a control $\boldsymbol{u} \in \mathbb{R}^{n_u}$ and a state $\boldsymbol{x} \in \mathbb{R}^{n_x}$. The control and the state are coupled together via a set of equality constraints:

$$g(\boldsymbol{x}, \boldsymbol{u}) = 0, \tag{1}$$

with $g : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$ (note that we choose the dimension of the output space to be equal to n_x, the dimension of the state). The function g is often related to the physical equations of the problem; and, depending on the applications, it can encode balance equations (optimal power flow, the primary object of study in this work), but also discretizations of dynamics (optimal control), or partial differential equations (PDE-constrained optimization).

We call Eq. (1) the *state equation* of the problem. Further, an objective function $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}$ and a set of generic constraints $h : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^m$ are given. The optimization problem in state-control form can be expressed as

$$\min_{\boldsymbol{x},\boldsymbol{u}} \; f(\boldsymbol{x}, \boldsymbol{u}) \quad \text{subject to} \quad \begin{cases} \boldsymbol{u} \geq 0, \quad \boldsymbol{x} \geq 0, \\ g(\boldsymbol{x}, \boldsymbol{u}) = 0, \quad h(\boldsymbol{x}, \boldsymbol{u}) \leq 0. \end{cases} \tag{2}$$

The focus of this paper is the efficient solution of (2) using the IPM. For that reason, one often prefers to introduce slack variables $\boldsymbol{s} \in \mathbb{R}^m$ for the nonlinear inequality constraints. Then, (2) can be rewritten as

$$\min_{\boldsymbol{x},\boldsymbol{u},\boldsymbol{s}} \; f(\boldsymbol{x}, \boldsymbol{u}) \quad \text{subject to} \quad \begin{cases} \boldsymbol{u} \geq 0, \quad \boldsymbol{x} \geq 0, \quad \boldsymbol{s} \geq 0 \\ g(\boldsymbol{x}, \boldsymbol{u}) = 0, \quad h(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{s} = 0. \end{cases} \tag{3}$$

The Lagrangian of the problem (3) is

$$\begin{aligned} L(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{s}; \boldsymbol{\lambda}, \boldsymbol{y}, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{z}) = f(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{\lambda}^\top g(\boldsymbol{x}, \boldsymbol{u}) \\ + \boldsymbol{y}^\top \big(h(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{s}\big) - \boldsymbol{v}^\top \boldsymbol{x} - \boldsymbol{w}^\top \boldsymbol{u} - \boldsymbol{z}^\top \boldsymbol{s}, \end{aligned} \tag{4}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^{n_x}$ is the multiplier associated with the state Eq. (1), $\boldsymbol{y} \in \mathbb{R}^m$ the multiplier associated with the inequality constraints $h(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{s} = 0$, and $\boldsymbol{v} \in \mathbb{R}^{n_x}$, $\boldsymbol{w} \in \mathbb{R}^{n_u}$, $\boldsymbol{z} \in \mathbb{R}^m$ the multipliers associated with the bound constraints.

In what follows, we assume that f, g, and h are twice continuously differentiable on their domain. Throughout the article, we denote

$$\begin{aligned}
\boldsymbol{g} &= \nabla_{(\boldsymbol{x},\boldsymbol{u})} f(\boldsymbol{x}, \boldsymbol{u}) \in \mathbb{R}^{n_x+n_u} && \text{gradient of the objective} \\
A &= \partial_{(\boldsymbol{x},\boldsymbol{u})} h(\boldsymbol{x}, \boldsymbol{u}) \in \mathbb{R}^{m\times(n_x+n_u)} && \text{Jacobian of the inequality cons.} \\
G &= \partial_{(\boldsymbol{x},\boldsymbol{u})} g(\boldsymbol{x}, \boldsymbol{u}) \in \mathbb{R}^{n_x\times(n_x+n_u)} && \text{Jacobian of the equality cons.} \\
W &= \nabla^2_{(\boldsymbol{x},\boldsymbol{u})} L(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{s}; \boldsymbol{\lambda}, \boldsymbol{y}, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{z}) && \text{Hessian of Lagrangian.}
\end{aligned}$$

We partition the first- and second-order derivatives into the blocks associated with the state $\boldsymbol{x}$ and the control $\boldsymbol{u}$; that is,

$$\boldsymbol{g} = \begin{bmatrix} \boldsymbol{g}_u \\ \boldsymbol{g}_x \end{bmatrix}, \quad \begin{aligned} A &= \begin{bmatrix} A_u & A_x \end{bmatrix} \\ G &= \begin{bmatrix} G_u & G_x \end{bmatrix} \end{aligned}, \quad \text{and} \quad W = \begin{bmatrix} W_{uu} & W_{ux} \\ W_{xu} & W_{xx} \end{bmatrix}.$$

2.2 Linearize-then-Reduce

Now, we discuss the first reduction method: *linearize-then-reduce*. This method exploits the structure of the problem (2) directly at the linear algebra level.

2.2.1 Successive Reductions of the KKT System

We first present the KKT system in an augmented form and show how we can reduce it by removing from the formulation first the equality constraints and then the inequality constraints.

KKT conditions. We introduce the diagonal matrices $X = \operatorname{diag}(\boldsymbol{x})$, $U = \operatorname{diag}(\boldsymbol{u})$, $S = \operatorname{diag}(\boldsymbol{s})$. The KKT conditions associated with the standard formulation (3) are

$$\boldsymbol{g}_u + G_u^\top \boldsymbol{\lambda} + A_u^\top \boldsymbol{y} - \boldsymbol{w} = 0, \tag{5a}$$
$$\boldsymbol{g}_x + G_x^\top \boldsymbol{\lambda} + A_x^\top \boldsymbol{y} - \boldsymbol{v} = 0, \tag{5b}$$
$$\boldsymbol{y} - \boldsymbol{z} = 0, \tag{5c}$$
$$g(\boldsymbol{x}, \boldsymbol{u}) = 0, \tag{5d}$$
$$h(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{s} = 0, \tag{5e}$$
$$X\boldsymbol{v} = 0, \ \boldsymbol{x}, \boldsymbol{v} \geq 0, \tag{5f}$$
$$U\boldsymbol{w} = 0, \ \boldsymbol{u}, \boldsymbol{w} \geq 0, \tag{5g}$$
$$S\boldsymbol{z} = 0, \ \boldsymbol{s}, \boldsymbol{z} \geq 0 \tag{5h}$$

Augmented KKT system. The interior-point method replaces the complementarity conditions by using a homotopy approach. In particular, with a fixed barrier $\mu > 0$ the complementary conditions (5f)-(5h) give $X\boldsymbol{v} = \mu \boldsymbol{e}_{n_x}$, $U\boldsymbol{w} = \mu \boldsymbol{e}_{n_u}$, $S\boldsymbol{z} = \mu \boldsymbol{e}_m$ [26].

Here, $\boldsymbol{e}_n$ is the vector of all ones of dimension n. By linearizing (5), we obtain the following (nonsymmetric) linear system, which is used for the step computation within interior-point iterations.

$$\begin{bmatrix} W_{uu} & W_{ux} & 0 & G_u^\top & A_u^\top & 0 & -I & 0 \\ W_{xu} & W_{xx} & 0 & G_x^\top & A_x^\top & -I & 0 & 0 \\ 0 & 0 & 0 & 0 & I & 0 & 0 & -I \\ G_u & G_x & 0 & 0 & 0 & 0 & 0 & 0 \\ A_u & A_x & I & 0 & 0 & 0 & 0 & 0 \\ 0 & V & 0 & 0 & 0 & X & 0 & 0 \\ W & 0 & 0 & 0 & 0 & 0 & U & 0 \\ 0 & 0 & Z & 0 & 0 & 0 & 0 & S \end{bmatrix} \begin{bmatrix} \boldsymbol{p}_u \\ \boldsymbol{p}_x \\ \boldsymbol{p}_s \\ \boldsymbol{p}_\lambda \\ \boldsymbol{p}_y \\ \boldsymbol{p}_v \\ \boldsymbol{p}_w \\ \boldsymbol{p}_z \end{bmatrix} = - \begin{bmatrix} \boldsymbol{g}_u + G_u^\top \boldsymbol{\lambda} + A_u^\top \boldsymbol{y} - \boldsymbol{w} \\ \boldsymbol{g}_x + G_x^\top \boldsymbol{\lambda} + A_x^\top \boldsymbol{y} - \boldsymbol{v} \\ \boldsymbol{y} - \boldsymbol{z} \\ g(\boldsymbol{x}, \boldsymbol{u}) \\ h(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{s} \\ X\boldsymbol{v} - \mu \boldsymbol{e}_{n_x} \\ U\boldsymbol{w} - \mu \boldsymbol{e}_{n_u} \\ S\boldsymbol{z} - \mu \boldsymbol{e}_m \end{bmatrix}. \tag{6a}$$

In IPM, it is standard to eliminate the last three block rows from (6a) (associated with $(\boldsymbol{p}_v, \boldsymbol{p}_w, \boldsymbol{p}_z)$). The elimination yields the following reduced, symmetric linear system:

$$\underbrace{\begin{bmatrix} W_{uu} + \Sigma_u & W_{ux} & 0 & G_u^\top & A_u^\top \\ W_{xu} & W_{xx} + \Sigma_x & 0 & G_x^\top & A_x^\top \\ 0 & 0 & \Sigma_s & 0 & I \\ G_u & G_x & 0 & 0 & 0 \\ A_u & A_x & I & 0 & 0 \end{bmatrix}}_{K_{aug}} \begin{bmatrix} \boldsymbol{p}_u \\ \boldsymbol{p}_x \\ \boldsymbol{p}_s \\ \boldsymbol{p}_\lambda \\ \boldsymbol{p}_y \end{bmatrix} = - \begin{bmatrix} \boldsymbol{g}_u + G_u^\top \boldsymbol{\lambda} + A_u^\top \boldsymbol{y} - \mu U^{-1} \boldsymbol{e}_{n_u} \\ \boldsymbol{g}_x + G_x^\top \boldsymbol{\lambda} + A_x^\top \boldsymbol{y} - \mu X^{-1} \boldsymbol{e}_{n_x} \\ \boldsymbol{y} - \mu S^{-1} \boldsymbol{e}_m \\ g(\boldsymbol{x}, \boldsymbol{u}) \\ h(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{s} \end{bmatrix}. \tag{6b}$$

Here, $\Sigma_u := U^{-1}W$, $\Sigma_x := X^{-1}V$, and $\Sigma_s := S^{-1}Z$. The matrix K_{aug} is sparse and symmetric indefinite and is typically factorized by a direct linear solver at each iteration of the interior-point algorithm. As discussed before, however, this form is not amenable to the GPU, motivating us to reduce further the system (6b).

Reduced KKT system. The reduction strategy adopted in the linearize-then-reduce method exploits the invertibility of G_x. In many applications, such as optimal power flow, optimal control, and PDE-constrained optimization, the state is completely determined by the control. This situation imposes the structure in the Jacobian block, and in many applications the G_x block is invertible. In the next theorem, we show how the invertibility of G_x allows the reduction in the KKT system (6b). For ease of notation, we denote by $\boldsymbol{r} := (\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4, \boldsymbol{r}_5)$ the right-hand side vector in (6b).

Theorem 2.1 *If at $(\boldsymbol{x}, \boldsymbol{u}) \in \mathbb{R}^{n_u} \times \mathbb{R}^{n_x}$ the Jacobian G_x is invertible, then we define the reduced Hessian $\hat{W}_{uu}$ and the reduced Jacobian $\hat{A}_u$ as*

$$\begin{aligned} \hat{W}_{uu} &= W_{uu} - W_{ux} G_x^{-1} G_u - G_u^\top G_x^{-\top} W_{xu} + G_u^\top G_x^{-\top} (W_{xx} + \Sigma_x) G_x^{-1} G_u, \\ \hat{A}_u &= A_u - A_x G_x^{-1} G_u. \end{aligned}$$

Then, the augmented KKT system (6b) *is equivalent to*

$$\underbrace{\begin{bmatrix} \hat{W}_{uu} + \Sigma_u & 0 & \hat{A}_u^\top \\ 0 & \Sigma_s & I \\ \hat{A}_u & I & 0 \end{bmatrix}}_{K_{red}} \begin{bmatrix} \boldsymbol{p}_u \\ \boldsymbol{p}_s \\ \boldsymbol{p}_y \end{bmatrix} = - \begin{bmatrix} \hat{\boldsymbol{r}}_1 \\ \hat{\boldsymbol{r}}_2 \\ \hat{\boldsymbol{r}}_3 \end{bmatrix}, \tag{7a}$$

where

$$\begin{cases} \hat{\boldsymbol{r}}_1 = \boldsymbol{r}_1 - G_u^\top G_x^{-\top} \boldsymbol{r}_2 - \big(W_{ux} - G_u^\top G_x^{-\top}(W_{xx} + \Sigma_x)\big) G_x^{-1} \boldsymbol{r}_4, \\ \hat{\boldsymbol{r}}_2 = \boldsymbol{r}_3, \\ \hat{\boldsymbol{r}}_3 = \boldsymbol{r}_5 - A_x G_x^{-1} \boldsymbol{r}_4. \end{cases}$$

Further, the state and adjoint descent directions can be recovered as

$$\begin{aligned} \boldsymbol{p}_x &= -G_x^{-1}\big(\boldsymbol{r}_4 + G_u \boldsymbol{p}_u\big), \\ \boldsymbol{p}_\lambda &= -G_x^{-\top}\Big(\boldsymbol{r}_2 + A_x^\top \boldsymbol{p}_y + W_{xu}\boldsymbol{p}_u + (W_{xx} + \Sigma_x)\boldsymbol{p}_x\Big). \end{aligned} \tag{7b}$$

Proof Using the fourth block of rows in (6b), we can remove the variable $\boldsymbol{p}_x$ in the system (7a) using G_x^{-1} as a pivot. We get

$$\begin{bmatrix} W_{uu} + \Sigma_u - W_{ux}G_x^{-1}G_u & 0 & G_u^\top & A_u^\top \\ W_{xu} - (W_{xx} + \Sigma_x)G_x^{-1}G_u & 0 & G_x^\top & A_x^\top \\ 0 & \Sigma_s & 0 & I \\ A_u - A_x G_x^{-1} G_u & I & 0 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{p}_u \\ \boldsymbol{p}_s \\ \boldsymbol{p}_\lambda \\ \boldsymbol{p}_y \end{bmatrix} = - \begin{bmatrix} \boldsymbol{r}_1 - W_{ux}G_x^{-1}\boldsymbol{r}_4 \\ \boldsymbol{r}_2 - (W_{xx} + \Sigma_x)G_x^{-1}\boldsymbol{r}_4 \\ \boldsymbol{r}_3 \\ \boldsymbol{r}_5 - A_x G_x^{-1}\boldsymbol{r}_4 \end{bmatrix}.$$

The descent direction w.r.t. $\boldsymbol{x}$ can be recovered with an additional linear solve: $\boldsymbol{p}_x = -G_x^{-1}\big(\boldsymbol{r}_4 + G_u \boldsymbol{p}_u\big)$. Then, we can eliminate the third block of columns (associated with $\boldsymbol{p}_\lambda$), with $G_x^{-\top}$ as a pivot. We recover the linear system in (7a). The descent direction $\boldsymbol{p}_\lambda$ now satisfies

$$\begin{aligned} \boldsymbol{p}_\lambda &= -G_x^{-\top}\Big(\boldsymbol{r}_2 - (W_{xx} + \Sigma_x)G_x^{-1}\boldsymbol{r}_4 + A_x^\top \boldsymbol{p}_y + (W_{xu} - (W_{xx} + \Sigma_x)G_x^{-1}G_u)\boldsymbol{p}_u\Big) \\ &= -G_x^{-\top}\Big(\boldsymbol{r}_2 + A_x^\top \boldsymbol{p}_y + W_{xu}\boldsymbol{p}_u + (W_{xx} + \Sigma_x)\boldsymbol{p}_x\Big) \end{aligned}$$

by rearranging the terms, thus completing the proof. □

Condensed KKT system. The left-hand side matrix K_{red} in (7a) has a size $(n_u + 2m) \times (n_u + 2m)$, which can be prohibitively large if the number of constraints m is substantial. Fortunately, (7a) can be condensed further, down to a system with size $n_u \times n_u$. This additional reduction requires the invertibility of Σ_s, which is always the case for interior-point algorithms.

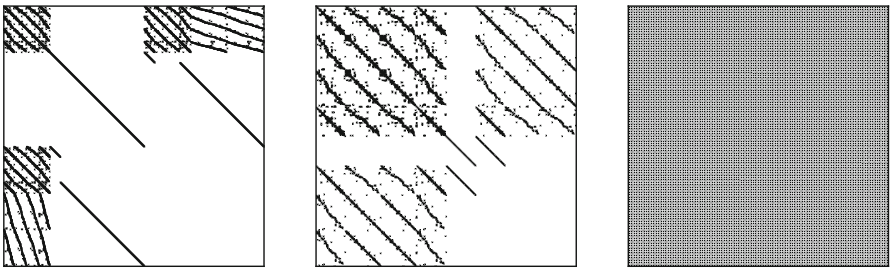


Fig. 1 Successive reductions $K_{aug} \to K_{cond} \to K_{red}$ associated with `118ieee`

Theorem 2.2 *We suppose that Σ_s is nonsingular. Then, the linear system (7a) is equivalent to*

$$\underbrace{\left(\hat{W}_{uu} + \hat{A}_u^\top \Sigma_s \hat{A}_u\right)}_{K_{cond}} \boldsymbol{p}_u = -\left(\hat{\boldsymbol{r}}_1 + \hat{A}_u^\top \Sigma_s \hat{\boldsymbol{r}}_3 - \hat{A}_u^\top \hat{\boldsymbol{r}}_2\right), \tag{8a}$$

the slack and multiplier descent directions being recovered as

$$\boldsymbol{p}_y = \Sigma_s\left(\hat{A}_u \boldsymbol{p}_u + \hat{\boldsymbol{r}}_3 - \Sigma_s^{-1}\hat{\boldsymbol{r}}_2\right), \quad \boldsymbol{p}_s = -\Sigma_s^{-1}(\boldsymbol{p}_y + \hat{\boldsymbol{r}}_2)\ . \tag{8b}$$

Proof. In (7a), we eliminate the second block of columns (associated with the slack descent $\boldsymbol{p}_s$) to get

$$\begin{bmatrix} \hat{W}_{uu} & \hat{A}_u^\top \\ \hat{A}_u & -\Sigma_s^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{p}_u \\ \boldsymbol{p}_y \end{bmatrix} = - \begin{bmatrix} \hat{\boldsymbol{r}}_1 \\ \hat{\boldsymbol{r}}_3 - \Sigma_s^{-1}\hat{\boldsymbol{r}}_2 \end{bmatrix}. \tag{9}$$

The slack descent direction is recovered as $\boldsymbol{p}_s = -\Sigma_s^{-1}(\boldsymbol{p}_y + \hat{\boldsymbol{r}}_2)$. In the last block of rows, we can reuse Σ_s^{-1} as a pivot to further simplify (9) with $\boldsymbol{p}_y = \Sigma_s\left(\hat{A}_u \boldsymbol{p}_u + \hat{\boldsymbol{r}}_3 - \Sigma_s^{-1}\hat{\boldsymbol{r}}_2\right)$:

$$\left(\hat{W}_{uu} + \hat{A}_u^\top \Sigma_s \hat{A}_u\right)\boldsymbol{p}_u = -(\hat{\boldsymbol{r}}_1 + \hat{A}_u^\top \Sigma_s \hat{\boldsymbol{r}}_3 - \hat{A}_u^\top \hat{\boldsymbol{r}}_2). \qquad \square$$

Discussion. The final matrix K_{cond} has a size $n_u \times n_u$ and is dense, meaning that it can be factorized efficiently by any LAPACK library. We note that the reduction has proceeded in two steps: first, we have reduced the system by eliminating the state variables, and then we have condensed it by eliminating the slacks, giving the order $K_{aug} \to K_{red} \to K_{cond}$. We have opted for this order to simplify the comparison with the *reduce-then-linearize* approach presented in Sect. 2.3. In practice, however, it is more convenient to first condense the linear system and then reduce it: $K_{aug} \to K_{cond} \to K_{red}$. This equivalent approach avoids the allocation of the dense reduced Jacobian $\hat{A}_u$, which has a size $m \times n_u$ and is expensive to store in memory. The reduction $K_{aug} \to K_{cond} \to K_{red}$ is illustrated in Fig. 1.

We establish the last condition to guarantee that we compute a descent direction at each iteration. For any symmetric matrix $M \in \mathbb{R}^{n\times n}$, we denote its inertia (the numbers of positive, negative, and zero eigenvalues) by $\mathrm{I}(M) = (n_+, n_-, n_0)$.

Theorem 2.3 *The step* $\boldsymbol{p}$ *in* (6b) *is a descent direction if*

- $I(K_{aug}) = (n_x + n_u + m, n_x + m, 0)$*, equivalent to*
- $I(K_{red}) = (n_u + m, m, 0)$*, equivalent to*
- $I(K_{cond}) = (n_u, 0, 0)$ *(the condensed matrix* K_{cond} *is positive definite).*

The equivalence of the three conditions can be verified via the Haynsworth inertia additivity formula.

2.2.2 Linearize-then-Reduce Algorithm (LinRed IPM)

Now that the different reductions have been introduced, we are able to present the *linearize-then-reduce* (`LinRed`) algorithm in Algorithm 1, following [8]. The **reduction** step is in itself an expensive operation, as we will explain in Sect. 3.3. The other bottlenecks are the factorization of the dense condensed matrix K_{cond} (which amounts to a Cholesky factorization if the matrix is positive definite) and the factorization of the sparse Jacobian G_x.

Data: Initial primal variables $(\boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{s}_0)$ and dual variables $(\boldsymbol{\lambda}_0, \boldsymbol{y}_0)$
for $k = 0, \ldots$ **do**
  Evaluate the derivatives W, G, A at $(\boldsymbol{x}_k, \boldsymbol{u}_k)$;
  Reduction: Condense the KKT system (6b) in K_{cond};
  Control step: Factorize K_{cond} in (8a) and solve the system to find $\boldsymbol{p}_u$;
  Dual step: $\boldsymbol{p}_y = \Sigma_s(\hat{A}_u \boldsymbol{p}_u + \hat{\boldsymbol{r}}_3 - \Sigma_s^{-1}\hat{\boldsymbol{r}}_2)$;
  Slack step: $\boldsymbol{p}_s = -\Sigma_s^{-1}(\boldsymbol{p}_y + \hat{\boldsymbol{r}}_2)$;
  State step: $\boldsymbol{p}_x = -G_x^{-1}\big(\boldsymbol{r}_4 + G_u \boldsymbol{p}_u\big)$;
  Adjoint step: $\boldsymbol{p}_\lambda = -G_x^{-\top}(\boldsymbol{r}_2 + A_x^\top \boldsymbol{p}_y + W_{xu}\boldsymbol{p}_u + (W_{xx} + \Sigma_x)\boldsymbol{p}_x)$;
  Line search: Update primal-dual direction $(\boldsymbol{u}_{k+1}, \boldsymbol{x}_{k+1}, \boldsymbol{s}_{k+1}, \boldsymbol{\lambda}_{k+1}, \boldsymbol{y}_{k+1})$ using a filter line search along the direction $(\boldsymbol{p}_u, \boldsymbol{p}_x, \boldsymbol{p}_s, \boldsymbol{p}_\lambda, \boldsymbol{p}_y)$;
end

Algorithm 1: Linearize-then-reduce algorithm

2.3 Reduce-then-Linearize

We now focus on our second reduction scheme, operating directly at the level of the nonlinear problem (2). This method can be interpreted as an interior-point alternative of the reduced-gradient algorithm of Dommel and Tinney [10] and was revisited recently in [21].

2.3.1 Nonlinear Reduction

Nonlinear projection. Instead of operating the reduction in the linearized KKT system as in Sect. 2.2, Dommel and Tinney's method uses the implicit function theorem to remove the state from the problem.

Theorem 2.4 *(Implicit function theorem). Let* $g : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$ *a continuously differentiable function, and let* $(\boldsymbol{x}, \boldsymbol{u}) \in \mathbb{R}^{n_x} \times \mathbb{R}^{n_u}$ *be such that* $g(\boldsymbol{x}, \boldsymbol{u}) = 0$*. If the Jacobian* G_x *is* invertible*, then there exist an open set* $U \subset \mathbb{R}^{n_u}$ *(with* $\boldsymbol{u} \in U$*) and a unique differentiable function* $\underline{x} : U \to \mathbb{R}^{n_x}$ *such that* $g(\underline{x}(\boldsymbol{u}), \boldsymbol{u}) = 0$ *for all* $\boldsymbol{u} \in U$*.*

The implicit function theorem gives, under certain assumptions, the existence of a local differentiable function $\underline{x} : U \to \mathbb{R}^{n_x}$ attached to a given control $\boldsymbol{u}$. If we assume that on the feasible domain $g(\boldsymbol{x}, \boldsymbol{u}) = 0$ is solvable and the Jacobian is invertible everywhere, we can derive the reduced-space problem as

$$\min_{\boldsymbol{u}} \quad f(\underline{x}(\boldsymbol{u}), \boldsymbol{u}) \quad \text{subject to} \quad \begin{cases} \boldsymbol{u} \geq 0, \quad \underline{x}(\boldsymbol{u}) \geq 0 \\ h(\underline{x}(\boldsymbol{u}), \boldsymbol{u}) \leq 0. \end{cases} \tag{10}$$

In contrast to (2), the problem (10) optimizes only with relation to the control $\boldsymbol{u}$, the state being defined *implicitly* via the local function $\underline{x}$. By definition $g(\underline{x}(\boldsymbol{u}), \boldsymbol{u}) = 0$, the state equation is automatically satisfied in the reduced space. However, the reduced-space problem is tied to the assumptions of the implicit function theorem: in some applications, finding a control $\boldsymbol{u}$ invertible w.r.t. the state Eq. (1) can be challenging.
Reduced derivatives. We define the reduced objective and the reduced constraints as

$$f_r(\boldsymbol{u}) := f(\underline{x}(\boldsymbol{u}), \boldsymbol{u}), \quad h_r(\boldsymbol{u}) := \begin{bmatrix} h(\underline{x}(\boldsymbol{u}), \boldsymbol{u}) \\ -\underline{x}(\boldsymbol{u}) \end{bmatrix}, \tag{11}$$

and we note the reduced Lagrangian $L_r(\boldsymbol{u}, \boldsymbol{s}; \boldsymbol{y}) := f_r(\boldsymbol{u}) + \boldsymbol{y}^\top (h_r(\boldsymbol{u}) + \boldsymbol{s}) - \boldsymbol{w}^\top \boldsymbol{u} - \boldsymbol{z}^\top \boldsymbol{s}$. By exploiting the implicit function theorem, we can deduce the derivatives in the reduced space.

Theorem 2.5 *(Reduced derivatives [17, Chapter 15.]) Let* $\boldsymbol{u} \in \mathbb{R}^{n_u}$ *such that the conditions of the implicit function theorem hold. Then,*

– *The functions* f_r *and* h_r *are continuously differentiable, with*

$$\nabla f_r(\boldsymbol{u}) = \boldsymbol{g}_u - G_u^\top G_x^{-\top} \boldsymbol{g}_x, \quad \hat{A}_u = \partial h_r(\boldsymbol{u}) = \begin{bmatrix} A_u - A_x G_x^{-1} G_u \\ G_x^{-1} G_u \end{bmatrix}. \tag{12a}$$

– *The Hessian of the reduced Lagrangian* L_r *satisfies*

$$\hat{W}_{uu} = W_{uu} - W_{ux} G_x^{-1} G_u - G_u^\top G_x^{-\top} W_{xu} + G_u^\top G_x^{-\top} W_{xx} G_x^{-1} G_u. \tag{12b}$$

Reduced-space KKT system. The KKT conditions associated with the reduced problem (10) are

$$\nabla_u f_r + \hat{A}_u^\top \boldsymbol{y} - \boldsymbol{w} = 0, \tag{13a}$$
$$\boldsymbol{y} - \boldsymbol{z} = 0, \tag{13b}$$
$$h_r(\boldsymbol{u}) + \boldsymbol{s} = 0, \tag{13c}$$
$$U\boldsymbol{w} = 0,\ \boldsymbol{u}, \boldsymbol{w} \geq 0, \tag{13d}$$
$$S\boldsymbol{z} = 0,\ \boldsymbol{s}, \boldsymbol{z} \geq 0, \tag{13e}$$

translating, at each iteration of the IPM algorithm, to the following augmented KKT system:

$$\begin{bmatrix} \hat{W}_{uu} + \Sigma_u & 0 & \hat{A}_u^\top \\ 0 & \Sigma_s & I \\ \hat{A}_u & I & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{p}_u \\ \boldsymbol{p}_s \\ \boldsymbol{p}_y \end{bmatrix} = - \begin{bmatrix} \nabla_u L_r - \mu U^{-1} \boldsymbol{e}_{n_u} \\ \boldsymbol{y} - \mu S^{-1} \boldsymbol{e}_m \\ h_r(\boldsymbol{u}) + \boldsymbol{s} \end{bmatrix}. \tag{14}$$

We note that we can apply Theorem 2.2 to get a condensed form of the KKT system (14). The KKT system (14) has a structure similar to that of (7a) but with minor differences. (i) The state $\boldsymbol{x}$ and the adjoint $\boldsymbol{\lambda}$ are updated independently of (14), respectively, by solving the state Eq. (1) and by solving a linear system. (ii) The bounds $\boldsymbol{x} \geq 0$ are incorporated inside the function h_r, whereas (7a) handles them explicitly in the KKT system (leading to an additional term Σ_x in the reduced Hessian $\hat{W}_{uu}$). (iii) The right-hand side in (7a) incorporates additional second-order terms that do not appear in (14). Indeed, as here with $\boldsymbol{r}_4 = g(\boldsymbol{x}, \boldsymbol{u}) = 0$, all the terms associated with $\boldsymbol{r}_4$ disappeared in the right-hand side of (14), including the second-order terms.

2.3.2 Reduce-then-Linearize Algorithm (RedLin IPM)

At each iteration k, the *reduce-then-linearize* (`RedLin`) algorithm proceeds in two steps. First, given a new control $\boldsymbol{u}_k$, the algorithm finds a state $\boldsymbol{x}_k$ satisfying $g(\boldsymbol{x}_k, \boldsymbol{u}_k) = 0$ by using a nonlinear solver. Then, once $\boldsymbol{x}_k$ has been computed, the reduced derivatives are updated, and the condensed form of the system (14) is solved to compute the next iterate. This process is summarized in Algorithm 2. We note that compared with Algorithm 1, the **state step** and the **adjoint step** are computed before solving the KKT system since we first have to solve (1). In addition, the algorithm gives no guarantee that at iteration k there exists a state $\boldsymbol{x}_k$ such that $g(\boldsymbol{x}_k, \boldsymbol{u}_k) = 0$, which can be problematic on certain applications. On the other hand, even if interrupted early, the algorithm yields a feasible state for the state Eq. (1), which may be important in real-time applications.

3 Implementation of Reduced IPM on GPU

In this section, we present a GPU implementation of the reduced-space algorithm in the programming language Julia. To avoid expensive data transfers between the host

Data: Initial primal variable $(\boldsymbol{u}_0, \boldsymbol{s}_0)$ and dual variable $(\boldsymbol{y}_0)$
for $k = 0, \ldots$ **do**
  Projection: Find $\boldsymbol{x}_k$ satisfying $g(\boldsymbol{x}_k, \boldsymbol{u}_k) = 0$;
  Adjoint step: Solve $\boldsymbol{\lambda} = -G_x^{-\top} \nabla_x L$;
  Reduction: Condense the KKT system (14) in K_{cond};
  Control step: Factorize K_{cond} and solve the system to find $\boldsymbol{p}_u$;
  Dual step: $\boldsymbol{p}_y = \Sigma_s(\hat{A}_u \boldsymbol{p}_u + \hat{\boldsymbol{r}}_3 - \Sigma_s^{-1} \hat{\boldsymbol{r}}_2)$;
  Slack step: $\boldsymbol{p}_s = -\Sigma_s^{-1}(\boldsymbol{p}_y + \hat{\boldsymbol{r}}_2)$;
  Line search: Update primal-dual direction $(\boldsymbol{u}_{k+1}, \boldsymbol{s}_{k+1}, \boldsymbol{y}_{k+1})$ using a filter line search along the direction $(\boldsymbol{p}_u, \boldsymbol{p}_s, \boldsymbol{p}_y)$;
end

Algorithm 2: Reduce-then-linearize algorithm

and the device, we have designed our implementation to run as much as possible on the GPU, comprising (i) the evaluation of the callbacks, (ii) the reduction algorithm, and (iii) the dense factorization of the condensed KKT system. In the end, only the interior-point routines (line search, second-order correction, etc.) run on the host. In particular, our method does not require transferring dense matrices between device and host, and most of the operations performed on the host are simple scalar operations.

3.1 GPU Operations

The key idea is to exploit efficient computation kernels on the GPU to implement the reduction algorithm. To the extent possible, we avoid writing custom kernels and rely instead on the BLAS and LAPACK operations, as provided by the vendor library (CUDA in our case). We list hereafter the kernels we are using. (1) `SpMV`/`SpMM` *(sparse matrix–vector product/sparse matrix–dense matrix product)*. (2) `SpSV`/`SpSM` *(sparse triangular solve)*: The routine `SpSV` solves the triangular systems $L^{-1}\boldsymbol{b}$, with L a lower-triangular matrix. The extension to multiple right-hand sides is provided by the `SpSM` kernel. (3) `SpRF` *(sparse LU refactorization)*: GPUs are notoriously inefficient at factorizing a sparse matrix M. However, if the sparse matrix M always has the same sparsity pattern, we can compute the initial factorization on the CPU and move the factors to the GPU. Then, if the nonzero coefficients of the matrix change, but its sparsity pattern stays fixed, the matrix can be refactorized entirely on the GPU by updating directly the L and U factors. In CUDA, the operations `SpMV`/`SpMM` and `SpSV`/`SpSM` are implemented in `cusparse`, whereas `SpRF` is implemented in the `cusolverRF` package.

3.2 Porting the Callbacks to the GPU

In nonlinear programming, the evaluation of the callbacks is often one of the most time-consuming parts, and the OPF problem is not immune to this issue. Most of the time, the derivatives of the OPF model are provided explicitly [40], evaluated by using automatic differentiation [12, 14] or symbolic differentiation [19]. Here, we have chosen to stick with automatic differentiation. We have streamlined on the GPU the evaluation of the

objective and of the constraints by adopting the vectorized model proposed in [24]. The nonlinear expressions are factorized in a basis vector $\psi(\boldsymbol{x}, \boldsymbol{u}) \in \mathbb{R}^{n_\psi}$, evaluated in parallel inside a single GPU kernel. Then, the objective and the constraints are recovered by composing the basis with linear transformations ($f(\boldsymbol{x}, \boldsymbol{u}) = M_f \psi(\boldsymbol{x}, \boldsymbol{u})$, $g(\boldsymbol{x}, \boldsymbol{u}) = M_g \psi(\boldsymbol{x}, \boldsymbol{u})$, $h(\boldsymbol{x}, \boldsymbol{u}) = M_h \psi(\boldsymbol{x}, \boldsymbol{u})$) all translating to `SpMV` operations on the GPU. The basis $\psi(\boldsymbol{x}, \boldsymbol{u})$ is differentiable on the GPU using `ForwardDiff.jl` [29]. In total, each iteration of the reduced IPM algorithm requires one evaluation of the objective's gradient (=one reverse pass), one evaluation of the Jacobian of the constraints (=one forward pass), and one evaluation of the Hessian of the Lagrangian (=one forward-over-reverse pass).

The OPF displays two desirable structures. (i) Its KKT system is super-sparse, rendering all the `SpMV` operations efficient (we have at most a dozen of nonzeroes on each row of the sparse matrices). (ii) The sparsity of the Jacobian G_x is fixed (it is associated with the structure of the power network), allowing for efficient refactorization with `SpRF`.

3.3 Porting the Reduction Algorithm to the GPU

Once the derivatives W and A are evaluated in the full space, it remains to build the condensed matrix K_{cond} in (8a). The algorithm has to be repeated at each iteration of the IPM algorithm, and its performance is critical. If we introduce the sparse matrix $K = W + A^\top \Sigma_s A$, the condensed matrix K_{cond} (8a) rewrites

$$K_{cond} = \hat{W}_{uu} + \hat{A}_u^\top \Sigma_s \hat{A}_u = \begin{bmatrix} I \\ -G_x^{-1} G_u \end{bmatrix}^\top \begin{bmatrix} K_{uu} & K_{ux} \\ K_{xu} & K_{xx} \end{bmatrix} \begin{bmatrix} I \\ -G_x^{-1} G_u \end{bmatrix}. \tag{15}$$

Evaluating (15) requires three different operations: (i) factorizing the Jacobian G_x (`SpRF`), (ii) triangular solves $G_x^{-1} b$ (`SpSV/SpSM`), and (iii) sparse matrix–matrix multiplications with K (`SpMM`). The order in which the operations are performed is important, affecting the complexity of the reduction algorithm.

First, we factorize G_x as $P G_x Q = L U$, with P and Q two permutation matrices and L and U being, respectively, a lower and an upper triangular matrix: using `SpRF`, can be refactorized entirely on the GPU if the sparsity pattern of G_x is the same along the iterations. Once the factorization is computed, solving the linear solve $G_x^{-1} \boldsymbol{b}$ translates to 2 `SpMV` and 2 `SpSV` routines, as $G_x^{-1} \boldsymbol{b} = Q U^{-1} L^{-1} P \boldsymbol{b}$.

Once the factorization is computed, a naive idea is to evaluate the full sensitivity matrix $S = -G_x^{-1} G_u$ explicitly, in order to reduce the total number of linear solves to n_u. However, the matrix S is dense, with size $n_x \times n_u$, and the cross-product $S^\top K_{xx} S$ requires storing another intermediate dense matrix with size $n_x \times n_u$ to evaluate the dense product $S^\top (K_{xx} S)$ (which is itself slow when n_x is large). Hence, the algorithm is not tractable on large instances.

Therefore, we avoid computing the full sensitivity matrix S and rely instead on a batched variant of the adjoint–adjoint algorithm [28]. We evaluate *explicitly* the sparse matrix $K \in \mathbb{R}^{(n_x+n_u)\times(n_x+n_u)}$ (nontrivial but doable in one sparse addition and one sparse–sparse multiplication `SpGEMM` to evaluate the cross-product $A^\top \Sigma_s A$). Then,

we evaluate K_{cond} *explicitly* slice by slice using $\text{div}(n_u, N)+1$ Hessian matrix products $K_{cond}V$, with $V \in \mathbb{R}^{n_u \times N}$ a dense matrix encoding N vectors in the Cartesian basis of $\mathbb{R}^{n_u}$. Evaluating the product $K_{cond}V$ amounts to three successive operations:

1. Solve $Z = -G_x^{-1}(G_u\, V)$. (3 `SpMM`, 2 `SpSM`)
2. Evaluate $\begin{bmatrix} H_u \\ H_x \end{bmatrix} = \begin{bmatrix} K_{uu} & K_{ux} \\ K_{xu} & K_{xx} \end{bmatrix} \begin{bmatrix} V \\ Z \end{bmatrix}$. (1 `SpMM`)
3. Solve $\Psi = G_x^{-\top} H_x$ and get $K_{cond} = H_u - G_u \Psi$. (3 `SpMM`, 2 `SpSM`)

One Hessian matrix product $K_{cond}V$ requires $2N$ linear solves (streamlined in four `SpSM` operations), giving a total of 7 `SpMM` and 4 `SpSM` operations. Evaluating $K_{cond}V$ requires $3 \times n_x \times N$ storage for the two intermediates $Z, \Psi \in \mathbb{R}^{n_x \times N}$, as well as an additional buffer to store the permuted matrix in the LU triangular solves. Using `cusparse`, we can evaluate one Hessian matrix product $K_{cond}V$ in parallel on the GPU. By repeating the operation $\text{div}(n_u, N) + 1$ times, we are able to evaluate the condensed matrix K_{cond} in order to factorize it inside a direct linear solver.

4 Numerical Results

In this section, we assess the performance of the reduced IPM algorithm on the GPU. Our algorithms have been implemented in Julia. All the benchmarks presented have been generated on our workstation, equipped with an NVIDIA V100 GPU and using `CUDA 11.4`. The open-source code is available on https://github.com/exanauts/Argos.jl/. We present in Sect. 4.1 how we have implemented *linearize-then-reduce* (`LinRed`) and *reduce-then-linearize* (`RedLin`) inside our interior-point solver MadNLP. Then, we detail in Sect. 4.2 a benchmark assessing the performance of reduced-space IPM on the GPU, by comparing its performance with that of state-of-the-art full-space IPM solvers running on the CPU.

4.1 MadNLP: A GPU-Ready Interior-Point Solver

We have implemented both the *linearize-then-reduce* (`LinRed`, Sect. 2.2) and the *reduce-then-linearize* (`RedLin`, Sect. 2.3) inside the MadNLP solver [35], a filter line-search interior-point solver written in Julia. As a reference, we benchmark our code against Knitro 13.0 [39] (using the derivatives evaluated by MATPOWER [40]) and with MadNLP solving the OPF problem in the full space.

Our implementation evaluates the derivatives on the GPU using the vectorized OPF model presented in Sect. 3.2. When solving the OPF in the full-space, MadNLP solves the original augmented system (6b) on the CPU with the state-of-the-art linear solver HSL MA27 [11], whereas the `LinRed` and `RedLin` variants solve the KKT system entirely on the GPU. Concerning scaling, MadNLP uses the same approach as Ipopt, based on the norm of the first-order derivatives [38]. The initial primal variables $(\boldsymbol{x}_0, \boldsymbol{u}_0)$ are specified inside the MATPOWER file, and the initial dual variables are set to zero: $\boldsymbol{y}_0 = 0$. The algorithm stops when the primal and dual infeasibilities are below 10^{-8}.

`LinRed` uses the reduction algorithm presented in Sect. 3.3 with a batch size $N = 256$ to evaluate the condensed matrix K_{cond}. MadNLP runs in inertia-based mode: Theorem 2.3 states that the inertia is correct if and only if the reduced matrix K_{cond} is positive definite. At each iteration, the algorithm factorizes the matrix K_{cond} with the dense Cholesky solver shipped with `cusolver`; if the factorization fails, we apply a primal-dual regularization to K_{cond} until it becomes positive definite. The hybrid *full-space* IPM and `LinRed` use both the same model and the same derivatives and are equivalent in exact arithmetic unless we run into a feasibility restoration phase. In that edge case, the algebra of `LinRed` can be adapted but no longer follows the workflow we presented above. In that circumstance, `LinRed` applies the dual regularization only on the inequality constraints, to fit the linear algebra framework we introduced in Sect. 2.2.

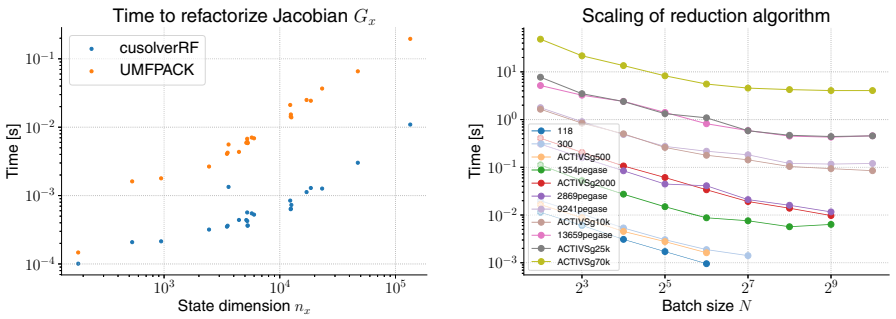
In contrast to `LinRed`, `RedLin` follows a feasible path w.r.t. the state equation: the algorithm can be stopped at any time and return a point feasible w.r.t. the state equation $g(\boldsymbol{x}, \boldsymbol{u}) = 0$ (but without any guarantee regarding the inequality constraints $h(\boldsymbol{x}, \boldsymbol{u}) \leq 0$). `RedLin` evaluates the derivatives in the reduced-space (12) and solves the state Eq. (1) (the power flow balance equations) at each iteration with a Newton–Raphson algorithm (with a tolerance of 10^{-10}). The algorithm has two drawbacks: (i) this approach requires evaluating the reduced Jacobian $\hat{A}_u$, with size $m \times n_u$, and (ii) the default scaling computed by MadNLP depends on G_x^{-1}, and is inappropriate if G_x exhibits poor conditioning. To ensure that the comparison is fair, we have modified the scaling in MadNLP so that `RedLin` uses the same scaling as `LinRed`.

4.2 Solving the OPF Problem on the GPU

We benchmark the reduced IPM algorithm on the OPF problem. We select a subset of the MATPOWER cases provided in [40], whose size varies from medium to large scale. In addition, we add three cases from the PGLIB benchmark [2] with fewer degrees of freedom.

How far can we parallelize the reduction algorithm on the GPU? We first benchmark the reduction algorithm on the different OPF instances. For `SpRF`, the reduction algorithm uses the library `cusolverRF` (with an initial factorization computed by KLU), the kernels `SpSM` and `SpMM` being provided by `cusparse`. We show in Fig. 2a that `cusolverRF` is able to refactorize efficiently the Jacobian G_x on the GPU (its sparsity pattern is constant and given by the structure of the underlying network). In Fig. 2b, we depict the performance of the reduction algorithm against the batch size N. We observe that the greater the size N, the better is the performance, until we reach the scalability limit of the GPU. For instance, on `ACTIVSg70k` we reach the limit when $N = 512$, meaning we cannot parallelize the algorithm further on the GPU. This limits the performance of the reduction since $\text{div}(11\,789, 512) + 1 = 24$ batched Hessian matrix products remain to be computed to evaluate the reduced Hessian of `ACTIVSg70k`. We note that, overall, it makes no sense to use a batch size greater than $N = 256$.

How fast can we solve the OPF on the GPU? The results of the benchmark are presented in Table 1. We note that the performance is consistent with that reported in the recent



(a) (b)

Fig. 2 Performance of the reduction algorithm: **a** time spent refactorizing the Jacobian G_x in `cusolverRF` and in `UMFPACK`, **b** performance of the reduction algorithm against the batch size N

benchmark [22]. Here, the dagger sign † indicates that the IPM algorithm runs into a feasibility restoration: this is the case both for `ACTIVSg10k` and for `ACTIVSg70k` (even Knitro struggles on these two instances, with numerous conjugate gradient iterations performed). The $*$ sign indicates that the initial point is infeasible w.r.t. the state Eq. (1), leading to a breakdown in `RedLin` (which assumes the current point is feasible at each iterate). We project the initial point back on the power flow manifold for the three `goc` cases encountering this issue.

We make the following observations. (i) *Hybrid full-space IPM* is in average 10% faster than Knitro, but only because it evaluates the derivatives on the GPU. (ii) `LinRed` is able to solve all the instances, including `ACTIVSg70k`. (iii) As expected, we get the same number of iterations between *Full-space IPM* and `LinRed` except on `13659pegase`. This case is indeed ill-conditioned, and the convergence is sensitive to the linear solver employed (even HSL MA27 and MA57 give different results on this case). (iv) Despite the good performance of the Cholesky factorization, `LinRed` is negatively impacted by the scalability of the reduction algorithm, as illustrated in Fig. 3. On the largest instances, `LinRed` beats *Full-space IPM* only on the three `goc` instances, which have fewer degrees of freedom. (v) The conditioning of the condensed matrix K_{cond} remains moderate and is not an issue for the direct Cholesky solver implemented in `cusolver`: on `case1354pegase`, it increases up to 1.6×10^{13} (compared to 2.5×10^{14} in the full-space).

We now analyze the performance of `RedLin`. (i) `RedLin` is able to solve instances with up to 25,000 buses, which, to the best of our knowledge, is a net improvement compared with previous attempts to solve the OPF in the reduced space [21, 27]. (ii) On the largest instances, `RedLin` is penalized compared with `LinRed`, since it has to deal with the reduced Jacobian $\hat{A}_u$. For that reason, `RedLin` breaks on `ACTIVSg70k` since we are running out of memory. (iii) From `case118` to `9241pegase`, `RedLin` converges in fewer iterations than does `LinRed`. (iv) On `ACTIVSg10k`, `RedLin` does not run into the feasibility restoration we encountered in `LinRed`: On this difficult instance, the convergence of `RedLin` is smoother, even if it requires more iterations. (v) The Newton–Raphson is not guaranteed to converge, but empirically we find this is not an issue.

Table 1 Benchmarking `LinRed` and `RedLin` with *Full-space IPM* on various OPF instances

Case	n_x	n_u	% DOF	Knitro		Full-space IPM		`LinRed`		`RedLin`	
				#it	Time (s)	#it	Time (s)	#it	Time (s)	#it	Time (s)
ieee118	181	107	0.37	10	**0.1**	16	**0.2**	16	**0.3**	16	**0.4**
ieee300	530	137	0.21	10	**0.1**	22	**0.3**	22	**0.4**	24	**0.6**
ACTIVSg500	943	111	0.11	20	**0.5**	24	**0.3**	24	**0.5**	22	**0.6**
1354pegase	2447	519	0.17	22	**1.2**	40	**0.9**	40	**1.2**	32	**1.3**
ACTIVSg2000	3607	783	0.18	18	**1.6**	43	**2.0**	43	**2.4**	33	**2.3**
2869pegase	5227	1019	0.16	22	**2.1**	50	**2.0**	50	**2.7**	34	**2.6**
9241pegase	17,036	2889	0.14	102	**31.7**	69	**10.7**	69	**23.7**	48	**28.3**
ACTIVSg10k	18,544	2909	0.14	130	**39.3**	76†	**7.9**	88†	**21.9**	140	**90.5**
13659pegase	23,225	8183	0.26	120	**116.0**	346	**98.3**	145	**242.7**	167	**944.0**
ACTIVSg25k	47,246	5505	0.10	47	**36.1**	86	**24.7**	86	**85.0**	52	**156.2**
ACTIVSg70k	134,104	11,789	0.08	101	**242.0**	90†	**89.8**	85†	**658.2**	∞	∞
9591goc	19,013	335	0.02	37	**22.5**	43	**11.7**	43	**7.7**	35*	**7.2**
10480goc	20,620	677	0.03	40	**25.7**	50	**14.0**	50	**11.5**	28*	**8.7**
19402goc	38,418	769	0.02	45	**66.5**	47	**30.8**	47	**19.5**	37*	**24.9**

Bold values emphasize the total time spent in the algorithm, to distinct that value from the number of iterations
We define % DOF= $n_u/(n_x + n_u)$
† The IPM algorithm has run into feasibility restoration
*The initial point is infeasible w.r.t. the power flow equations: the initial point is projected onto the power flow feasible set before running the RedLin algorithm

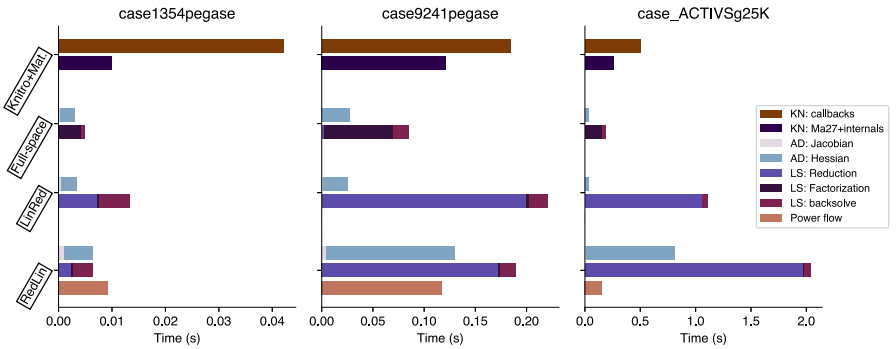


Fig. 3 Breakdown of the time spent in one iteration on a small instance (`case1354pegase`), a medium instance (`case9241pegase`) and a large instance (`ACTIVSg25k`)

How does the time for one iteration decompose? We depict in Fig. 3 the time spent in *one* IPM iteration for the four different methods, on three different cases with small, medium, and large sizes. We observe: (i) Thanks to GPU acceleration, automatic differentiation is not a bottleneck for *Full-space IPM* and `LinRed`, and the time spent in the callbacks is less than those reported for Knitro (which uses MATPOWER and explicit derivatives in the callbacks). However, `RedLin` is penalized on large-scale instances, as it has to evaluate the dense reduced Jacobian and Hessian explicitly. (ii) Both Knitro and *Full-space IPM* spend a significant amount of time on factorizing the KKT system with MA27. However, MA27 remains overall faster than the reduction algorithm used both in `LinRed` and `RedLin` (even if factorizing the condensed matrix K_{cond} with dense Cholesky on the GPU is blazingly fast). (iii) Solving the power flow in `RedLin` amounts to 26% of the total time on the small case, but this ratio decreases to 3% of the total time on the large case.

5 Conclusion

This paper has presented an efficient implementation of the IPM on GPU architectures, based on a Schur reduction in the underlying KKT system. We have derived two practical algorithms, *linearize-then-reduce* and *reduce-then-linearize*, adapted their workflows to be efficient when the vast majority of their computation is run on the GPU, and detailed their respective performance on different large-scale OPF instances. We also discussed the benefits of a hybrid full-space IPM solver—computing the derivatives on the GPU and the linear algebra on the CPU—and demonstrated that this approach generally outperforms Knitro, running exclusively on the CPU. The relative performance of the reduced-space algorithms is highly dependent on the ratio of controls with respect to the total number of variables. Their performance lags behind both Knitro and the hybrid full-space solver when the problem has many control variables (as it is the case on the MATPOWER benchmark) but is significantly ahead—up to a factor of 3—when the problem has less than 7% degrees of freedom. Moreover, the reduce-then-linearize algorithm has the added benefit of producing a feasible solution to the power flow equations at *any* iteration, which makes it a

great candidate for real-time applications. To improve the performance of reduction algorithms, we believe the most important item is to alleviate the dependence on the number of control variables. We plan to explore a way to accelerate the reduction by exploiting the exponentially decaying structure of the reduced Hessian [34].

Acknowledgements We thank Juraj Kardoš for reviving the interest in reduced-space methods, Kasia Świrydowicz for pointing us to the `cusolverRF` library and the anonymous referees for their helpful remarks. This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the US Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. The material is based upon work supported in part by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

1. Abadie, J., Carpentier, J.: Generalization of the wolfe reduced gradient method to the case of nonlinear constraints. In: Fletcher, R. (ed.) Optimization, pp. 37–47. Academic Press (1969)
2. Babaeinejadsarookolaee, S., Birchfield, A., Christie, R.D., Coffrin, C., DeMarco, C., Diao, R., Ferris, M., Fliscounakis, S., Greene, S., Huang, R., Josz, C., Korab, R., Lesieutre, B., Maeght, J., Mak, T.W.K., Molzahn, D.K. Overbye, T.J., Panciatici, P., Park, B., Snodgrass, J., Tbaileh, A., Van Hentenryck, P., Zimmerman, R.: *The power grid library for benchmarking AC optimal power flow algorithms*, arXiv preprint arXiv:1908.02788, (2019)
3. Biegler, L.T., Nocedal, J., Schmid, C.: A reduced Hessian method for large-scale constrained optimization. SIAM J. Optim. **5**, 314–347 (1995)
4. Biros, G., Ghattas, O.: Parallel Lagrange-Newton-Krylov-Schur methods for PDE-constrained optimization: Part I - The Krylov-Schur Solver. SIAM J. Sci. Comput. **27**, 687–713 (2005)
5. Burchett, R., Happ, H., Vierath, D.: Quadratically convergent optimal power flow. IEEE Trans. Power Appar. Syst. **11**, 3267–3275 (1984)
6. Cain, M.B., Oneill, R.P., Castillo, A.: History of optimal power flow and formulations. Federal Energy Regul. Comm. **1**, 1–36 (2012)
7. Cao, Y., Seth, A., Laird, C.D.: An augmented Lagrangian interior-point approach for large-scale NLP problems on graphics processing units. Comput. Chem. Eng. **85**, 76–83 (2016)
8. Cervantes, A.M., Wächter, A., Tütüncü, R.H., Biegler, L.T.: A reduced space interior point strategy for optimization of differential algebraic systems. Comput. Chem. Eng. **24**, 39–51 (2000)
9. Coleman, T.F., Conn, A.R.: Nonlinear programming via an exact penalty function: Asymptotic analysis. Math. Program. **24**, 123–136 (1982)
10. Dommel, H., Tinney, W.: Optimal power flow solutions. IEEE Trans. Power Appar. Syst. **PAS–87**, 1866–1876 (1968)
11. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Oxford University Press, Oxford (2017)
12. Dunning, I., Huchette, J., Lubin, M.: JuMP: a modeling language for mathematical optimization. SIAM Rev. **59**, 295–320 (2017)
13. Fletcher, R.: Practical Methods of Optimization. John Wiley and Sons, New Jersey (2013)
14. Fourer, R., Gay, D.M., Kernighan, B.W.: A modeling language for mathematical programming. Manage. Sci. **36**, 519–554 (1990)
15. Frank, S., Steponavice, I., Rebennack, S.: Optimal power flow: a bibliographic survey I. Energy syst. **3**, 221–258 (2012)
16. Gabay, D.: Minimizing a differentiable function over a differential manifold. J. Optim. Theory Appl. **37**, 177–219 (1982)
17. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM (2008)
18. Gurwitz, C.B., Overton, M.L.: Sequential quadratic programming methods based on approximating a projected Hessian matrix. SIAM J. Sci. Stat. Comput. **10**, 631–653 (1989)
19. Hijazi, H., Wang, G., Coffrin, C.: Gravity: a mathematical modeling language for optimization and machine learning, Machine Learning Open Source Software Workshop at NeurIPS 2018, (2018). Available at www.gravityopt.com

20. Jiang, Q., Geng, G.: A reduced-space interior point method for transient stability constrained optimal power flow. IEEE Trans. Power Syst. **25**, 1232–1240 (2010)
21. Kardos, J., Kourounis, D., Schenk, O.: Reduced-space interior point methods in power grid problems, arXiv preprint arXiv:2001.10815, (2020)
22. Kardos, J., Kourounis, D., Schenk, O., Zimmerman, R.: Complete results for a numerical evaluation of interior point solvers for large-scale optimal power flow problems, arXiv preprint arXiv:1807.03964, (2018)
23. Kim, Y., Pacaud, F., Kim, K., Anitescu, M.: Leveraging GPU batching for scalable nonlinear programming through massive Lagrangian decomposition, arXiv preprint arXiv:2106.14995, (2021)
24. Lee, D., Turitsyn, K., Molzahn, D.K., Roald, L.A.: Feasible path identification in optimal power flow with sequential convex restriction. IEEE Trans. Power Syst. **35**, 3648–3659 (2020)
25. Nocedal, J., Overton, M.L.: Projected Hessian updating algorithms for nonlinearly constrained optimization. SIAM J. Numer. Anal. **22**, 821–850 (1985)
26. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer series in operations research, 2nd edn. Springer, New York (2006)
27. Pacaud, F., Maldonado, D.A., Shin, S., Schanen, M., Anitescu, M.: A feasible reduced space method for real-time optimal power flow. Electric Power Syst. Res. **212**, 108268 (2022)
28. Pacaud, F., Schanen, M., Maldonado, D.A., Montoison, A., Churavy, V., Samaroo, J., Anitescu, M.: Batched second-order adjoint sensitivity for reduced space methods, In Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing, SIAM, pp. 60–71 (2022)
29. Revels, J., Lubin, M., Papamarkou, T.: *Forward-mode automatic differentiation in Julia*, arXiv preprint arXiv:1607.07892, (2016)
30. Rosen, J.B.: The gradient projection method for nonlinear programming: part II - nonlinear constraints. J. Soc. Ind. Appl. Math. **9**, 514–532 (1961)
31. Sargent, R.W.H.: Reduced-gradient and projection methods for nonlinear programming. In: Gill, P.E., Murray, W. (eds.) Numerical Methods for Constrained Optimization, pp. 149–175. Academic Press, London (1974)
32. Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with PARDISO. Futur. Gener. Comput. Syst. **20**, 475–487 (2004)
33. Schubiger, M., Banjac, G., Lygeros, J.: GPU acceleration of ADMM for large-scale quadratic programming. J. Parallel Distrib. Comput. **144**, 55–67 (2020)
34. Shin, S., Anitescu, M., Zavala, V.M.: Exponential decay of sensitivity in graph-structured nonlinear programs. SIAM J. Optim. **32**, 1156–1183 (2022)
35. Shin, S., Coffrin, C., Sundar, K., Zavala, V.M.: Graph-based modeling and decomposition of energy infrastructures. IFAC-PapersOnLine **54**, 693–698 (2021)
36. Świrydowicz, K., Darve, E., Jones, W., Maack, J., Regev, S., Saunders, M.A., Thomas, S.J., Peleš, S.: Linear solvers for power grid optimization problems: a review of GPU-accelerated linear solvers, *Parallel Comput.*, p. 102870 (2021)
37. Tasseff, B., Coffrin, C., Wächter, A., Laird, C.: Exploring benefits of linear solver parallelism on modern nonlinear optimization applications, arXiv preprint arXiv:1909.08104, (2019)
38. Wächter, A., Biegler, L.T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math. Program. **106**, 25–57 (2006)
39. Waltz, R.A., Morales, J.L., Nocedal, J., Orban, D.: An interior algorithm for nonlinear optimization that combines line search and trust region steps. Math. Program. **107**, 391–408 (2006)
40. Zimmerman, R.D., Murillo-Sánchez, C.E., Thomas, R.J.: MATPOWER: steady-state operations, planning, and analysis tools for power systems research and education. IEEE Trans. Power Syst. **26**, 12–19 (2010)