Les Cahiers du GERAD ISSN: 0711–2440

Batched second-order adjoint sensitivity for reduced space methods

F. Pacaud, M. Schanen, D. A. Maldonado, A. Montoison, V. Churavy, J. Samaroo, M. Anitescu

G–2021–56

October 2021

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2021 – Bibliothèque et Archives Canada, 2021

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2021 – Library and Archives Canada, 2021

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Batched second-order adjoint sensitivity for reduced space methods

François Pacaud [a]

Michel Schanen [a]

Daniel Adrian Maldonado [a]

Alexis Montoison [b]

Valentin Churavy [c]

Julian Samaroo [c]

Mihai Anitescu [a]

[a] *Argonne National Laboratory, Lemont, IL 60439, United States*

[b] *Département de Mathématiques et de Génie Industriel, Polytechnique Montréal, Montréal (Qc), Canada, H3C 3A7*

[c] *Massachusetts Institute of Technology, Cambridge, MA 02139, United States*

alexis.montoison@polymtl.ca

October 2021
Les Cahiers du GERAD
G–2021–56

Abstract : This paper presents an efficient method for extracting the second-order sensitivities from a system of implicit nonlinear equations. We design a custom automatic differentiation (AutoDiff) backend that targets highly parallel graphics processing units (GPUs) by extracting the second-order information in batch. When the nonlinear equations are associated to a reduced space optimization problem, we leverage the parallel reverse-mode accumulation in a batched adjoint-adjoint algorithm to compute efficiently the reduced Hessian of the problem. We apply the method to extract the reduced Hessian associated to the balance equations of a power network, and show that a parallel GPU implementation leads to a 30 times speed-up on the largest instances, comparing to our reference CPU implementation.

Acknowledgements: This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. 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.

1 Introduction

System of nonlinear equations are ubiquitous in numerical computing. Resolving such nonlinear systems typically depends on efficient iterative algorithms, as for example Newton-Raphson. In this article, we are interested in the resolution of a *parametric* system of nonlinear equations, where the solution depends on a vector of parameters $\boldsymbol{p} \in \mathbb{R}^{n_p}$. These parametric systems are, in their abstract form, written as

$$\text{Find } \boldsymbol{x} \text{ such that } g(\boldsymbol{x}, \boldsymbol{p}) = 0 \,, \tag{1}$$

where the (smooth) nonlinear function $g : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ depends jointly on an unknown variable $\boldsymbol{x} \in \mathbb{R}^{n_x}$ and the parameters $\boldsymbol{p} \in \mathbb{R}^{n_p}$.

The solution $x(\boldsymbol{p})$ of (1) depends *implicitly* on the parameters $\boldsymbol{p}$: of particular interest are the sensitivities of the solution $x(\boldsymbol{p})$ with relation to the parameters $\boldsymbol{p}$. Indeed, these sensitivities can be embedded inside an optimization algorithm (if $\boldsymbol{p}$ is a design variable) or in an uncertainty quantification scheme (if $\boldsymbol{p}$ encodes an uncertainty). It is well known that propagating the sensitivities in an iterative algorithm is nontrivial [11]. Fortunately, there is no need to do so, as we can exploit the mathematical structure of (1) and compute directly the sensitivities of the solution $x(\boldsymbol{p})$ using the *Implicit Function Theorem*.

By repeating this process one more step, we are able to extract second-order sensitivities at the solution $x(\boldsymbol{p})$. However, this operation is computationally more demanding and involves the manipulation of third-order tensors $\nabla^2_{\boldsymbol{xx}} g, \nabla^2_{\boldsymbol{xp}} g, \nabla^2_{\boldsymbol{pp}} g$. The challenge is to avoid forming explicitly such tensors by using reverse mode accumulation of second-order information, either explicitly by using the specific structure of the problem — encoded by the function g — or by using automatic differentiation.

As illustrated in Figure 1, this paper covers the efficient computation of the second-order sensitivities of a nonlinear system (1). The sparsity structure of the problem is passed to a custom Automatic Differentiation (AutoDiff) backend that automatically generates all the intermediate sensitivities from the implementation of $g(\boldsymbol{x}, \boldsymbol{p})$. To get a tractable algorithm, we use an adjoint model implementation of the generated first-order sensitivities to avoid explicitly forming third-order derivative tensors. As an application, we compute the reduced Hessian of the nonlinear equations corresponding to the power flow balance equations of a power grid [27]. The problem has an unstructured graph structure, leading to some challenge in the automatic differentiation library, that we discuss extensively. We show that the (dense) reduced Hessian associated to the power flow equations can be computed efficiently in parallel, by using a batch of Hessian-vector products. The underlying motivation is to embed the reduction algorithm in a real-time tracking procedure [26], where the reduced Hessian updates have to be fast to track a suboptimal solution.

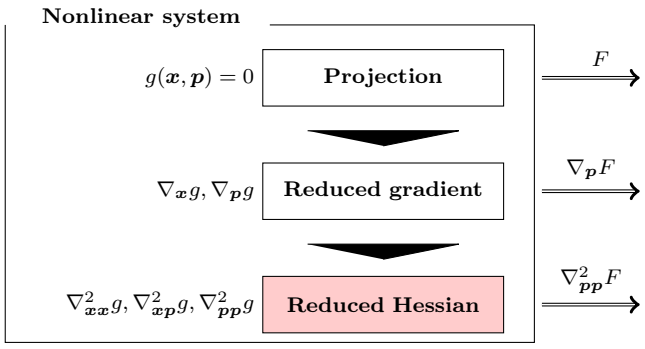

Figure 1: Reduced space algorithm. This article focuses on the last block, in red. If F is an objective function, the reduced gradient $\nabla_{\boldsymbol{p}} F$ and the reduced Hessian $\nabla^2_{\boldsymbol{pp}} F$ can be used in any nonlinear optimization algorithm.

In summary, we aim at devising a *portable*, *efficient*, and easy maintainable reduced Hessian algorithm. To this end, we leverage the expressiveness offered by the Julia programming language. Due to

the algorithm's design, the automatic differentiation backend and the reduction algorithm are transparently implemented on the GPU without any changes to the algorithm's core implementation, thus realizing a composable software design.

1.1 Contributions

Our contribution is a tractable SIMD algorithm and implementation to evaluate the reduced Hessian from a parametric system of nonlinear equations (1). This consists of three closely intertwined components. First, we implement the nonlinear function $g(\boldsymbol{x}, \boldsymbol{p})$ using the programming language Julia [5] and the portability layer `KernelAbstractions.jl` to generate abstract kernels working on various GPU architectures (CUDA, AMDGPU). Second, we develop a custom AutoDiff backend on top of the portability layer to extract automatically the first-order sensitivities $\nabla_{\boldsymbol{x}} g, \nabla_{\boldsymbol{p}} g$ and the second-order sensitivities $\nabla^2_{\boldsymbol{xx}} g, \nabla^2_{\boldsymbol{xp}} g, \nabla^2_{\boldsymbol{pp}} g$. Third, we combine these in an efficient parallel accumulation of the reduced Hessian associated to a given reduced space problem. The accumulation involves both Hessian tensor contractions and two sparse linear solves with multiple right-hand sides. Glued together, the three components give a generic code able to extract the second-order derivatives from a power grid problem, running in parallel on GPU architectures. Numerical experiments with Volta GPUs (V100) showcase the scalability of the approach, with a 30x speed-up on the largest instances, comparing to a reference CPU implementation.

2 Prior art

In this article we extract the second-order sensitivities from the system of nonlinear equations using automatic differentiation (AutoDiff). AutoDiff on Single Instruction, Multiple Data (SIMD) architectures alike the CUDA cores on GPUs is an ongoing research effort. Forward-mode AutoDiff effectively adds tangent components to the variables and preserves the computational flow. In addition, a vector mode can be applied to propagate multiple tangents or directional derivatives at once. The technique of automatically generating derivatives of function implementations has been investigated since the 1950s [3, 20].

Reverse- or adjoint-mode AutoDiff reverses the computational flow and thus incurs a lot of access restrictions on the final code. Every read of a variable becomes a write, and vice versa. This leads to application-specific solutions that exploit the structure of an underlying problem to generate efficient adjoint code [7, 12, 14]. Most prominently, the reverse mode is currently implemented as backpropagation in machine learning. Indeed, the backpropagation has a long history (e.g., [8]) with the reverse mode in AutoDiff being formalized for the first time in [17]. Because of the limited size and single access pattern of neural networks, current implementations [1, 15, 22] reach a high throughput on GPUs. For the wide field of numerical simulations, however, efficient adjoints of GPU implementations remain challenging. In this work we combine the advantages of GPU implementations of the gradient with the evaluation of Hessian-vector products first introduced in [23].

Reduced-space methods have been applied widely in uncertainty quantification and partial differential equation (PDE)-constrained optimization [6], and their applications in the optimization of power grids is known since the 1960s [10]. However, extracting the second-order sensitivities in the reduced space has been considered tedious to implement and hard to motivate on classical CPU architectures (see [16] for a recent discussion about the computation of the reduced Hessian on the CPU). To the best of our knowledge, this paper is the first to present a SIMD focused algorithm leveraging the GPU to efficiently compute the reduced Hessian of the power flow equations in polar formulation.

3 Reduced space problem

In Section 3.1 we briefly introduce the power flow nonlinear equations to motivate our application. We present in Section 3.2 the reduced space problem associated with the power flow problem, and recall

in Section 3.3 the first-order adjoint method, used to evaluate efficiently the gradient in the reduced space, and later applied to compute the adjoint of the sensitivities.

3.1 Presentation of the power flow problem

We present a brief overview of the steady-state solution of the power flow problem. The power grid can be described as a graph $\mathcal{G} = \{V, E\}$ with n_v vertices and n_e edges. The steady state of the network is described by the following nonlinear equations, holding at all nodes $i \in V$,

$$\begin{cases} P_i^{inj} = v_i \sum\limits_{j \in A(i)} v_j (g_{ij} \cos(\theta_i - \theta_j) + b_{ij} \sin(\theta_i - \theta_j)) , \\ Q_i^{inj} = v_i \sum\limits_{j \in A(i)} v_j (g_{ij} \sin(\theta_i - \theta_j) - b_{ij} \cos(\theta_i - \theta_j)) , \end{cases} \tag{2}$$

where at node i, (P_i^{inj} and Q_i^{inj}) are respectively the active and reactive power injections; v_i is the voltage magnitude; θ_i the voltage angle; and $A(i) \subset V$ is the set of adjacent nodes: for all $j \in A(i)$, there exists a line (i, j) connecting node i and node j. The values g_{ij} and b_{ij} are associated with the physical characteristics of the line (i, j). Generally, we distinguish the (PV) nodes — associated to the generators — from the (PQ) nodes comprising only loads. We note that the structure of the nonlinear equations (2) depends on the structure of the underlying graph through the adjacencies $A(\cdot)$.

We rewrite the nonlinear equations (2) in the standard form (1). At all nodes the power injection P_i^{inj} should match the net production P_i^g minus the load P_i^d:

$$g(\boldsymbol{x}, \boldsymbol{p}) = \begin{bmatrix} \boldsymbol{P}_{pv}^{inj} - \boldsymbol{P}^g + \boldsymbol{P}_{pv}^d \\ \boldsymbol{P}_{pq}^{inj} + \boldsymbol{P}_{pq}^d \\ \boldsymbol{Q}_{pq}^{inj} + \boldsymbol{Q}_{pd}^d \end{bmatrix} = 0 \, , \boldsymbol{x} = \begin{bmatrix} \boldsymbol{\theta}^{pv} \\ \boldsymbol{\theta}^{pq} \\ \boldsymbol{v}^{pq} \end{bmatrix} . \tag{3}$$

In (3), we have selected only a subset of the power flow equations (2) to ensure that the nonlinear system $g(\boldsymbol{x}, \boldsymbol{p}) = 0$ is invertible with respect to the state $\boldsymbol{x}$. The unknown variable $\boldsymbol{x}$ corresponds to the voltage angles at the PV and PQ nodes and the voltage magnitudes at the PQ nodes. However, in contrast to the variable $\boldsymbol{x}$, we have some flexibility in choosing the parameters $\boldsymbol{p}$.

In optimal power flow (OPF) applications, we are looking at minimizing a given operating cost $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}$ (associated to the active power generations $\boldsymbol{P}^g$) while satisfying the power flow equations (3). In that particular case, $\boldsymbol{p}$ is a design variable associated to the active power generations and the voltage magnitude at PV nodes: $\boldsymbol{p} = (\boldsymbol{P}^g, \boldsymbol{v}_{pv})$. We define the OPF problem as

$$\min_{\boldsymbol{x}, \boldsymbol{p}} \; f(\boldsymbol{x}, \boldsymbol{p}) \;\; \text{subject to } g(\boldsymbol{x}, \boldsymbol{p}) = 0 \, . \tag{4}$$

3.2 Projection in the reduced space

We note that in Equation (3), the functional g is continuous and that the dimension of the output space is equal to the dimension of the input variable $\boldsymbol{x}$. Thanks to the particular network structure of the problem (encoded by the adjacencies $A(\cdot)$ in (2)), the Jacobian $\nabla_{\boldsymbol{x}} g$ is sparse.

Generally, the nonlinear system (3) is solved iteratively with a Newton-Raphson algorithm. If at a fixed parameter $\boldsymbol{p}$ the Jacobian $\nabla_{\boldsymbol{x}} g$ is invertible, we compute the solution $x(\boldsymbol{p})$ starting from an initial guess $\boldsymbol{x}_0$: $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - (\nabla_{\boldsymbol{x}} g_k)^{-1} g(\boldsymbol{x}_k, \boldsymbol{p})$ for $k = 1, \cdots$. We know that if $\boldsymbol{x}_0$ is close enough to the solution, then the convergence of the algorithm is quadratic.

With the projection completed, the optimization problem (4) rewrites in the reduced space as

$$\min_{\boldsymbol{p}} \; F(\boldsymbol{p}) := f\big(x(\boldsymbol{p}), \boldsymbol{p}\big) \, , \tag{5}$$

reducing the number of optimization variables from $n_x + n_p$ to n_p, while at the same time eliminating all equality constraints in the formulation.

3.3 First-order adjoint method

With the reduced space problem (5) defined, we compute the reduced gradient $\nabla_{\boldsymbol{p}} F$ required for the reduced space optimization routine. By definition, as $x(\boldsymbol{p})$ satisfies $g(x(\boldsymbol{p}), \boldsymbol{p}) = 0$, the chain rule yields

$$\nabla_{\boldsymbol{p}} F = \nabla_{\boldsymbol{p}} f + \nabla_{\boldsymbol{x}} f \cdot \nabla_{\boldsymbol{p}} x \text{ with } \nabla_{\boldsymbol{p}} x = -\big(\nabla_{\boldsymbol{x}} g\big)^{-1} \nabla_{\boldsymbol{p}} g \, .$$

However, evaluating the full sensitivity matrix $\nabla_{\boldsymbol{p}} x$ involves the resolving of n_x linear system.

On the contrary, the *adjoint method* requires the resolving of a *single* linear system. For every dual $\boldsymbol{\lambda} \in \mathbb{R}^{n_x}$, we introduce a Lagrangian function defined as

$$\ell(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{\lambda}) := f(\boldsymbol{x}, \boldsymbol{p}) + \boldsymbol{\lambda}^\top g(\boldsymbol{x}, \boldsymbol{p}) \, . \tag{6}$$

If $\boldsymbol{x}$ satisfies $g(\boldsymbol{x}, \boldsymbol{p}) = 0$, then the Lagrangian $\ell(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{\lambda})$ does not depend on $\boldsymbol{\lambda}$ and we get $\ell(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{\lambda}) = F(\boldsymbol{p})$. By using the chain rule, the total derivative of ℓ with relation to the parameter $\boldsymbol{p}$ satisfies

$$\begin{aligned} d_p \ell &= \big(\nabla_{\boldsymbol{x}} f \cdot \nabla_{\boldsymbol{p}} x + \nabla_{\boldsymbol{p}} f\big) + \boldsymbol{\lambda}^\top \big(\nabla_{\boldsymbol{x}} g \cdot \nabla_{\boldsymbol{u}} x + \nabla_{\boldsymbol{p}} g\big) \\ &= \big(\nabla_{\boldsymbol{p}} f + \boldsymbol{\lambda}^\top \nabla_{\boldsymbol{p}} g\big) + \big(\nabla_{\boldsymbol{x}} f + \boldsymbol{\lambda}^\top \nabla_{\boldsymbol{x}} g\big) \nabla_{\boldsymbol{p}} x \, . \end{aligned}$$

We observe that by setting the first-order adjoint to $\boldsymbol{\lambda} = -(\nabla_{\boldsymbol{x}} g)^{-\top} \nabla_{\boldsymbol{x}} f^\top$, the reduced gradient $\nabla_{\boldsymbol{p}} F$ satisfies

$$\nabla_{\boldsymbol{p}} F = \nabla_{\boldsymbol{p}} \ell = \nabla_{\boldsymbol{p}} f + \boldsymbol{\lambda}^\top \nabla_{\boldsymbol{p}} g \, , \tag{7}$$

with $\boldsymbol{\lambda}$ evaluated by solving a single linear system.

4 Parallel reduction algorithm

We present in Section 4.1 the adjoint-adjoint method, and detail in Section 4.2 how to evaluate efficiently the second-order sensitivities with Autodiff. By combining together the Autodiff and the adjoint-adjoint method, we devise in Section 4.3 a parallel algorithm to compute the reduced Hessian.

4.1 Second-order adjoint over adjoint method

Among the different Hessian reduction schemes presented in [21] (direct-direct, adjoint-direct, direct-adjoint, adjoint-adjoint), the *adjoint-adjoint* method has two key advantages to evaluate the reduced Hessian on the GPU. First, it avoids forming explicitly the dense tensor $\nabla^2_{\boldsymbol{pp}} x$ and the dense matrix $\nabla_{\boldsymbol{p}} x$, leading to important memory savings on the larger cases. Second, it enables us to compute the reduced Hessian slice by slice, in an embarrassingly parallel fashion.

Conceptually, the adjoint-adjoint method extends the adjoint method (see §3.3) to compute the second-order derivatives $\nabla^2 f \in \mathbb{R}^{n_p \times n_p}$ of the objective function $f((x(\boldsymbol{p}), \boldsymbol{p})$. The adjoint-adjoint method computes the matrix $\nabla^2 f$ slice by slice, by using n_p Hessian-vector products $(\nabla^2 f)\boldsymbol{w}$ (with $\boldsymbol{w} \in \mathbb{R}^{n_p}$).

By definition of the first-order adjoint $\boldsymbol{\lambda}$, the derivative of the Lagrangian function (6) with respect to $\boldsymbol{x}$ is null:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{p}) + \boldsymbol{\lambda}^\top \nabla_{\boldsymbol{x}} g(\boldsymbol{x}, \boldsymbol{p}) = 0 \, . \tag{8}$$

Let $\hat{g}(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{\lambda}) := \nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{p}) + \boldsymbol{\lambda}^\top \nabla_{\boldsymbol{x}} g(\boldsymbol{x}, \boldsymbol{p})$. We define a new Lagrangian associated with (8) by introducing two second-order adjoints $\boldsymbol{z}, \boldsymbol{\psi} \in \mathbb{R}^{n_x}$ and a vector $\boldsymbol{w} \in \mathbb{R}^{n_p}$:

$$\begin{aligned} \hat{\ell}(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{\lambda}; \boldsymbol{z}, \boldsymbol{\psi}) := (\nabla_{\boldsymbol{p}} \ell)^\top \boldsymbol{w} + \\ \boldsymbol{z}^\top g(\boldsymbol{x}, \boldsymbol{p}) + \boldsymbol{\psi}^\top \hat{g}(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{\lambda}) \, . \end{aligned} \tag{9}$$

By computing the derivative of $\hat{\ell}$ and eliminating the terms corresponding to $\nabla_{\boldsymbol{x}}\boldsymbol{\lambda}$ and $\nabla_{\boldsymbol{p}}\boldsymbol{\lambda}$, we get the following expressions for the second-order adjoints $(\boldsymbol{z}, \boldsymbol{\psi})$:

$$\begin{cases} (\nabla_{\boldsymbol{x}} g)\boldsymbol{z} = -\big(\nabla_{\boldsymbol{p}} g\big)^\top \boldsymbol{w} \\ (\nabla_{\boldsymbol{x}} g)^\top \boldsymbol{\psi} = -(\nabla^2_{\boldsymbol{xp}} \ell)\boldsymbol{w} \ - (\nabla^2_{\boldsymbol{xx}} \ell)\boldsymbol{z} \ . \end{cases} \tag{10}$$

Then, the reduced-Hessian-vector product reduces to

$$\big(\nabla^2 f\big)\boldsymbol{w} = (\nabla^2_{\boldsymbol{pp}} \ell)\, \boldsymbol{w} + (\nabla^2_{\boldsymbol{px}} \ell)^\top \boldsymbol{z} + (\nabla_{\boldsymbol{p}} g)^\top \boldsymbol{\psi} \ . \tag{11}$$

As $\nabla^2 \ell = \nabla^2 f + \boldsymbol{\lambda}^\top \nabla^2 g$, we observe that both Equations (10) and (11) require evaluating the product of the three tensors $\nabla^2_{\boldsymbol{xx}} g$, $\nabla^2_{\boldsymbol{xp}} g$, and $\nabla^2_{\boldsymbol{pp}} g$, on the left with the adjoint $\boldsymbol{\lambda}$ and on the right with the vector $\boldsymbol{w}$. Evaluating the Hessian-vector products $(\nabla^2_{\boldsymbol{xx}} f)\boldsymbol{w}$, $(\nabla^2_{\boldsymbol{xp}} f)\boldsymbol{w}$ and $(\nabla^2_{\boldsymbol{pp}} f)\boldsymbol{w}$ is generally easier, as f is a real-valued function.

4.2 Second-order derivatives

To avoid forming the third-order tensors $\nabla^2 g$ in the reduction procedure presented previously in Section 4.1, we exploit the particular structure of Equations (10) and (11) to implement with automatic differentiation an adjoint-tangent accumulation of the derivative information. For any adjoint $\boldsymbol{\lambda} \in \mathbb{R}^{n_x}$ and vector $\boldsymbol{w} \in \mathbb{R}^{n_p}$, we build a tangent $\boldsymbol{v} = (\boldsymbol{z}, \boldsymbol{w}) \in \mathbb{R}^{n_x + n_p}$, with $\boldsymbol{z} \in \mathbb{R}^{n_x}$ solution of the first system in Equation (10). Then, the adjoint-forward accumulation evaluates a vector $\boldsymbol{y} \in \mathbb{R}^{n_x + n_p}$ as

$$\boldsymbol{y} = \begin{pmatrix} \boldsymbol{\lambda}^\top \nabla^2_{\boldsymbol{xx}} g & \boldsymbol{\lambda}^\top \nabla^2_{\boldsymbol{xp}} g \\ \boldsymbol{\lambda}^\top \nabla^2_{\boldsymbol{px}} g & \boldsymbol{\lambda}^\top \nabla^2_{\boldsymbol{pp}} g \end{pmatrix} \boldsymbol{v} \ , \tag{12}$$

(the tensor projection notation will be introduced more thoroughly in Section 4.2.3). We detail next how to compute the vector $\boldsymbol{y}$ by using forward-over-reverse AutoDiff.

4.2.1 AutoDiff

AutoDiff transforms a code that implements a multivariate vector function $\boldsymbol{y} = g(\boldsymbol{x})$, $\mathbb{R}^n \mapsto \mathbb{R}^m$ with inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$ into its differentiated implementation. We distinguish two modes of AutoDiff. Applying AutoDiff in *forward mode* generates the code for evaluating the Jacobian vector product $\boldsymbol{y}^{(1)} = \nabla g(\boldsymbol{x}) \cdot \boldsymbol{x}^{(1)}$, with the superscript $^{(1)}$ denoting first-order tangents—also known as directional derivatives. The *adjoint or reverse mode*, or backpropagation in machine learning, generates the code of the transposed Jacobian vector product $\boldsymbol{x}_{(1)} = \boldsymbol{y}_{(1)} \cdot \nabla g(\boldsymbol{x})^T$, with the subscript $_{(1)}$ denoting first-order adjoints. The adjoint mode is useful for computing gradients of scalar functions ($m = 1$) (such as Lagrangian) at a cost of $\mathcal{O}\left(cost(g)\right)$.

4.2.2 Sparse Jacobian accumulation

To extract the full Jacobian from a tangent or adjoint AutoDiff implementation, we have to let $\boldsymbol{x}^{(1)}$ and $\boldsymbol{y}_{(1)}$ go over the Cartesian basis of $\mathbb{R}^n$ and $\mathbb{R}^m$, respectively. This incurs the difference in cost for the Jacobian accumulation: $\mathcal{O}\left(n\right) \cdot cost(g)$ for the tangent Jacobian model and $\mathcal{O}\left(m\right) \cdot cost(g)$ for the adjoint Jacobian model. In our case we need the full square ($m = n$) Jacobian $\nabla_{\boldsymbol{x}} g$ of the nonlinear function (1) to run the Newton–Raphson algorithm. The tangent model is preferred whenever $m \approx n$. Indeed, the adjoint model incurs a complete reversal of the control flow and thus requires storing intermediate variables, leading to high cost in memory. Furthermore, SIMD architectures are particularly well suited for propagating the n independent tangent Jacobian vector products in parallel [24].

If n becomes larger ($\gg$1000), however, the memory requirement of all n tangents may exceed the GPU's memory. Since our Jacobian is sparse, we apply the technique of Jacobian coloring that compresses independent columns of the Jacobian and reduces the number of required *seeding* tangent vectors from n to the number of colors c (see Figure 2).

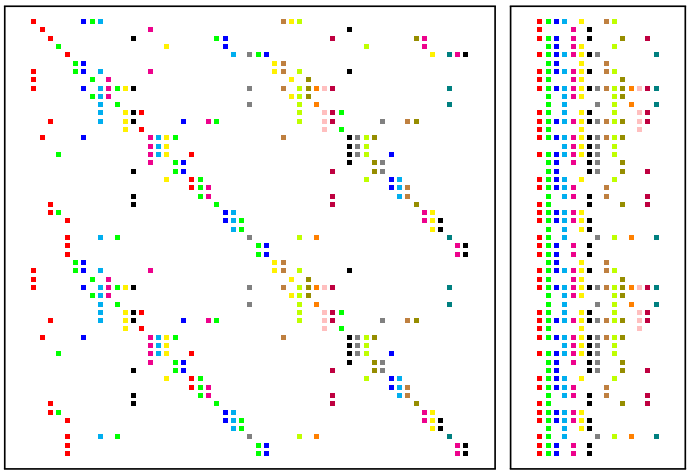

Figure 2: Jacobian compression via column coloring. On the left, the original Jacobian. On the right, the compressed Jacobian.

4.2.3 Second-order derivatives

For higher-order derivatives that involve derivative tensors (e.g., Hessian) we introduce the projection notation $< \cdots >$ introduced in [19] and illustrated in Figure 3 with $< \boldsymbol{x}_{(1)}, \nabla^2 g(\boldsymbol{x}), \boldsymbol{x}^{(1)} >$, whereby adjoints are projected from the left to the Jacobian and tangents from the right. To compute second-order derivatives and the Hessian projections in Equation (12), we use the adjoint model implementation given by

$$\boldsymbol{y} = g(\boldsymbol{x}), \boldsymbol{x}_{(1)} =< \boldsymbol{x}^{(1)}, \nabla g(\boldsymbol{x}) >= \boldsymbol{y}_{(1)} \cdot \nabla g(\boldsymbol{x})^T \,, \tag{13}$$

and we apply over it the tangent model given by

$$\boldsymbol{y} = g(\boldsymbol{x}), \quad \boldsymbol{y}^{(1)} =< \nabla g(\boldsymbol{x}), \boldsymbol{x}^{(1)} >= \nabla g(\boldsymbol{x}) \cdot \boldsymbol{x}^{(1)} \,, \tag{14}$$

yielding

$$\begin{aligned} \boldsymbol{y} &= g(\boldsymbol{x}), \\ \boldsymbol{y}^{(2)} &=< \nabla g(\boldsymbol{x}), \boldsymbol{x}^{(2)} >, \\ &\text{and} \\ \boldsymbol{x}_{(1)} &=< \boldsymbol{y}_{(1)}, \nabla g(\boldsymbol{x}) >, \\ \boldsymbol{x}^{(2)}_{(1)} &=< \boldsymbol{y}_{(1)}, \nabla^2 g(\boldsymbol{x}), \boldsymbol{x}^{(2)} > + < \boldsymbol{y}^{(2)}_{(1)}, \nabla g(\boldsymbol{x}) > \,. \end{aligned} \tag{15}$$

Notice that every variable has now a value component and three derivative components denoted by $_{(1)}$, $^{(2)}$, and $^{(2)}_{(1)}$ amounting to first-order adjoint, second-order tangent, and second-order tangent over adjoint, respectively. In Section 4.3, we compute the Hessian $\nabla^2 g \in \mathbb{R}^{m \times n \times n}$ vector product on the GPU by setting $\boldsymbol{y}^{(2)}_{(1)} = 0$ and extracting the result from $\boldsymbol{x}^{(2)}_{(1)} \in \mathbb{R}^n$.

4.3 Reduction algorithm

We are now able to write down the reduction algorithm to compute the Hessian-vector products $\nabla^2 f \cdot \boldsymbol{w}$. We first present a sequential version of the algorithm, and then detail how to design a parallel variant of the reduction algorithm.

4.3.1 Sequential algorithm

We observe that by default the Hessian reduction algorithm encompasses four sequential steps:

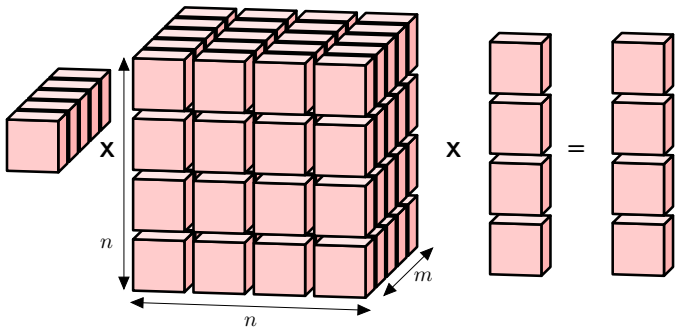

Figure 3: Hessian derivative tensor projection $< \boldsymbol{y}_{(1)}, \nabla^2 g(\boldsymbol{x}), \boldsymbol{x}^{(2)} >$. Notice that the Hessian slices along the n directions are symmetric.

Algorithm 1: Reduction algorithm.

Data: Vector $\boldsymbol{w} \in \mathbb{R}^{n_p}$
`SpMul`: $\boldsymbol{b} = (\nabla_{\boldsymbol{u}} g)\boldsymbol{w}$;
`SparseSolve`: $(\nabla_{\boldsymbol{x}} g)\boldsymbol{z} = -\boldsymbol{b}$;
`TensorProjection`: Compute $(\boldsymbol{y_x}, \boldsymbol{y_p})$ with (12) and $\boldsymbol{v} = (\boldsymbol{z}, \boldsymbol{w})$;
`SparseSolve`: $(\nabla_{\boldsymbol{x}} g)^\top \boldsymbol{\psi} = -\boldsymbol{y_x}$;
`MulAdd`: $(\nabla^2 f)\boldsymbol{w} = \boldsymbol{y_p} + (\nabla_{\boldsymbol{p}} g)^\top \boldsymbol{\psi}$;

1. `SparseSolve`: Get the second-order adjoint $\boldsymbol{z}$ by solving the first linear system in (10).
2. `TensorProjection`: Define the tangent $\boldsymbol{v} := (\boldsymbol{z}, \boldsymbol{w})$, and evaluate the second-order derivatives using (12). `TensorProjection` returns a vector $\boldsymbol{y} = (\boldsymbol{y_x}, \boldsymbol{y_p})$, with

$$\begin{cases} \boldsymbol{y_x} = < \boldsymbol{\lambda}^\top, \nabla_{\boldsymbol{xx}} g, \boldsymbol{z} > + < \boldsymbol{\lambda}^\top, \nabla_{\boldsymbol{xp}} g, \boldsymbol{w} > , \\ \boldsymbol{y_p} = < \boldsymbol{\lambda}^\top, \nabla_{\boldsymbol{px}} g, \boldsymbol{z} > + < \boldsymbol{\lambda}^\top, \nabla_{\boldsymbol{pp}} g, \boldsymbol{w} > , \end{cases} \tag{16}$$

   with "<>" denoting the derivative tensor projection introduced in Section 4.2.3 (and illustrated in Figure 3).
3. `SparseSolve`: Get the second-order adjoint $\boldsymbol{\psi}$ by solving the second linear system in Equation (10): $(\nabla_{\boldsymbol{x}} g)^\top \boldsymbol{\psi} = -\boldsymbol{y_x}$.
4. `SpMulAdd`: Compute the reduced Hessian-vector product with Equation (11).

The first `SparseSolve` differs from the second `SparseSolve` since the left-hand side is different: the first system considers the Jacobian matrix $(\nabla_{\boldsymbol{x}} g)$, whereas the second system considers its transpose $(\nabla_{\boldsymbol{x}} g)^\top$.

To compute the entire reduced Hessian $\nabla^2 f$, we have to let $\boldsymbol{w}$ go over all the Cartesian basis vectors of $\mathbb{R}^{n_p}$. The parallelization over these basis vectors is explained in the next paragraph.

4.3.2 Parallel algorithm

Instead of computing the Hessian vector products $(\nabla^2 f)\boldsymbol{w}_1, \cdots, (\nabla^2 f)\boldsymbol{w}_n$ one by one, the parallel algorithm takes as input a *batch* of N vectors $W = \left(\boldsymbol{w}_1, \cdots, \boldsymbol{w}_N\right)$ and evaluates the Hessian-vector products $\left((\nabla^2 f)\boldsymbol{w}_1, \cdots, (\nabla^2 f)\boldsymbol{w}_N\right)$ in a parallel fashion. By replacing respectively the `SparseSolve` and `TensorProjection` blocks by `BatchSparseSolve` and `BatchTensorProjection`, we get the parallel reduction algorithm presented in Algorithm 2 (and illustrated by Figure 4). On the contrary to Algorithm 1, the block `BatchSparseSolve` solves a sparse linear system with multiple right-hand-sides $B = (\nabla_{\boldsymbol{p}} g)W$, and the block `BatchTensorProjection` runs the Autodiff algorithm introduced in Section 4.2 in batch. As explained in the next section, both operations are fully amenable to the GPU.

Algorithm 2: Parallel reduction algorithm.

Data: N vectors $\boldsymbol{w}_1, \cdots, \boldsymbol{w}_N \in \mathbb{R}^{n_p}$
Build $W = (\boldsymbol{w}_1, \cdots, \boldsymbol{w}_N)$, $W \in \mathbb{R}^{n_p \times N}$;
`SpMul`: $B = \big(\nabla_{\boldsymbol{p}} g\big) W$, $B \in \mathbb{R}^{n_x \times N}$, $\nabla_{\boldsymbol{p}} g \in \mathbb{R}^{n_x \times n_p}$;
`BatchSparseSolve`: $(\nabla_{\boldsymbol{x}} g) Z = -B$;
`BatchTensorProjection`: Compute $(Y_{\boldsymbol{x}}, Y_{\boldsymbol{p}})$ with $V = (Z, W)$;
`BatchSparseSolve`: $(\nabla_{\boldsymbol{x}} g)^\top \Psi = -Y_{\boldsymbol{x}}$;
`SpMulAdd`: $(\nabla^2 f)\boldsymbol{W} = Y_{\boldsymbol{p}} + (\nabla_{\boldsymbol{p}} g)^\top \Psi$;

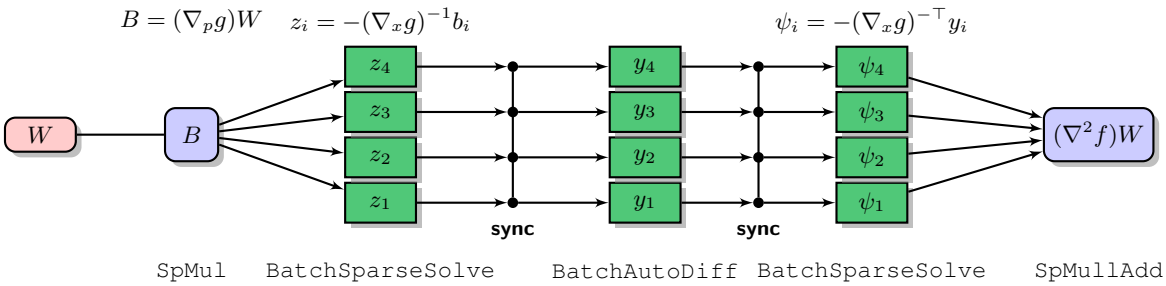

Figure 4: Parallel computation of the reduced Hessian vector products on the GPU.

5 SIMD GPU implementation

In the previous section, we have devised a parallel algorithm to compute the reduced Hessian. This algorithm involves two key ingredients, both running in parallel: `BatchSparseSolve` and `BatchTensorProjection`. We present in Section 5.1 how to implement `BatchTensorProjection` on GPU by leveraging the Julia language. Then, we focus on the parallel resolution of `BatchSparseSolve` in Section 5.2. The final implementation is presented in Section 5.3.

5.1 Batched AutoDiff

5.1.1 AutoDiff on GPU

Our implementation attempts to be architecture agnostic, and to this end we rely heavily on the just-in-time compilation capabilities of the Julia language. Julia has two key advantages for us: (i) it implements state-of-the-art automatic differentiation libraries and (ii) its multiple dispatch capability allows to write code in an architecture agnostic way. Combined together, this allows to run AutoDiff on GPU accelerators. On the architecture side we rely on the array abstraction implemented by the package `GPUArrays.jl` [4] and on the kernel abstraction layer `KernelAbstractions.jl`. The Julia community provides three GPU backends for these two packages: NVIDIA, AMD, and Intel oneAPI. Currently, `CUDA.jl` is the most mature package, and we are leveraging this infrastructure to run our code on an x64/PPC CPU and NVIDIA GPU. In the future our solution will be rolled out transparently onto AMD and Intel accelerators with minor code changes.

5.1.2 Forward evaluation of sparse Jacobians

The reduction algorithm in Section 4.3 requires (i) the Jacobian $\nabla_x g$ to form the linear system in (10) and (ii) the Hessian vector product of $\boldsymbol{\lambda}^\top \nabla^2 g$ in (16). We use the Julia package `ForwardDiff.jl` [25] to apply the first-order tangent model (14) by instantiating every variable as a dual type defined as `T1S{T,C} = ForwardDiff.Dual{T, C}}`, where `T` is the type (`double` or `float`) and `C` is the number of directions that are propagated together in parallel. This allows us to apply AutoDiff both on the CPU and on the GPU in a vectorized fashion, through a simple type change: for instance, `Array{TIS{T, C}}(undef, n)` instantiates a vector of dual numbers on the CPU, whereas `CuArray{TIS{T, C}}(undef, n)` does the same on a CUDA GPU. (Julia allows us to write code

where all the types are abstracted away). This, combined with `KernelAbstractions.jl`, allows us to write a portable residual kernel for $g(\boldsymbol{x}, \boldsymbol{p})$ that is both differentiable and architecture agnostic. By setting the number of Jacobian colors c to the parameter of type `T1S{C}` we leverage the GPUs by propagating the tangents in a SIMD way.

5.1.3 Forward-over-Reverse Hessian projections

As opposed to the forward mode, generating efficient adjoint code for GPUs is known to be hard. Indeed, adjoint automatic differentiation implies a reversal of the computational flow, and in the backward pass every read of a variable translates to a write adjoint, and vice versa. The latter is particularly complex for parallelized algorithms, especially as the automatic parallelization of algorithms is hard. For example, an embarrassingly parallel algorithm where each process reads the data of all the input space leads to a race condition in its adjoint that is challenging to address. Current state-of-the-art AutoDiff tools use specialized workarounds for certain cases. However, a generalized solution to this problem does not exist. The promising AutoDiff tool Enzyme [18] is able to differentiate CUDA kernels in Julia, but it is currently not able to digest all of our code.

To that end, we hand differentiate our GPU kernels for the Forward-over-Reverse Hessian projection. We then apply `ForwardDiff` to these adjoint kernels to extract second-order sensitivities according to the Forward-over-Reverse model. Notably, our test case (see Section 3.1) involves reversing a graph-based problem (with vertices V and edges E). The variables of the equations are defined on the vertices. To adjoin or reverse these kernels, we pre-accumulate the adjoints first on the edges and then on the nodes, thus avoiding a race condition on the nodes. This process yields a fully parallelizable adjoint kernel. Unfortunately, current AutoDiff tools are not capable of detecting such structural properties. Outside the kernels we use a tape (or stack) structure to store the values computed in the forward pass and to reuse them in the reverse (split reversal). The kernels themselves are written in joint reversal, meaning that the forward and reverse passes are implemented in one function evaluation without intermediate storage of variables in a data structure. For a more detailed introduction to writing adjoint code we recommend [13].

5.2 Batched sparse linear algebra

The block `BatchSparseSolve` presented in Section 4.3 requires the resolution of two sparse linear systems with multiple right-hand sides, as illustrated in Equation (10). This part is critical because in practice a majority of the time is spent inside the linear algebra library in the parallel reduction algorithm. To this end, we have wrapped the library `cuSOLVER_RF` in Julia to get an efficient LU solver on the GPU. For any sparse matrix $A \in \mathbb{R}^{n \times n}$, the library `cuSOLVER_RF` takes as input an LU factorization of the matrix A precomputed on the host, and transfers it to the device. `cuSOLVER_RF` has two key advantages to implement the resolution of the two linear systems in `BatchSparseSolve`. (i) If a new matrix $\tilde{A}$ needs to be factorized and has the same sparsity pattern as the original matrix A, the refactorization routine proceeds directly on the device, without data transfer between the host and the device. (allowing to match the performance of the state-of-the-art CPU sparse library UMFPACK [9]). (ii) Once the LU factorization has been computed, the forward and backward solves for different right-hand sides $\boldsymbol{b}_1, \cdots, \boldsymbol{b}_N$ can be computed in parallel, in batch mode.

5.3 Implementation of the parallel reduction

By combining the batch AutoDiff with the batch sparse linear solves of `cuSOLVER_RF`, we get a fully parallel algorithm to compute the reduced Hessian projection. We compute the reduced Hessian $\nabla^2 f \in \mathbb{R}^{n_p \times n_p}$ by blocks of N Hessian-vector products. If we have enough memory to set $N = n_p$, we can compute the full reduced Hessian in one batch reduction. Otherwise, we set $N < n_p$ and compute the full reduced Hessian in $N_b = div(n, N) + 1$ batch reductions.

Tuning the number of batch reductions N is nontrivial and depends on two considerations. How efficient is the parallel scaling when we run the two parallel blocks `BatchTensorProjection` and `BatchSparseSolve`? and Are we fitting into the device memory? This second consideration is indeed one of the bottlenecks of the algorithm. In fact, if we look more closely at the memory usage of the parallel reduced Hessian, we observe that the memory grows linearly with the number of batches N. First, in the block `BatchTensorProjection`, we need to duplicate N times the tape used in the reverse accumulation of the Hessian in Section 5.1, leading to memory increase from $\mathcal{O}(M_T)$ to $\mathcal{O}(M_T \times N)$, with M_T the memory of the tape. The principle is similar in `SparseSolve`, since the second-order adjoints $\boldsymbol{z}$ and $\boldsymbol{\psi}$ are also duplicated in batch mode, leading to a memory increase from $\mathcal{O}(2n_x)$ to $\mathcal{O}(2n_x \times N)$. This is a bottleneck on large cases when the number of variables n_x is large.

The other bottleneck arises when we combine together the blocks `BatchSparseSolve` and `BatchTensorProjection`. Indeed, `BatchTensorProjection` should wait for the first block `BatchSparseSolve` to finish its operations. The same issue arises when passing the results of `BatchTensorProjection` to the second `BatchSparseSolve` block. As illustrated by Figure 4, we need to add two explicit synchronizations in the algorithm. Allowing the algorithm to run the reduction algorithm in a purely asynchronous fashion would require a tighter integration with `cuSOLVER_RF`.

6 Numerical experiments

In this section we provide extensive benchmarking results that investigate whether the computation of the reduced Hessian $\nabla^2 f$ with Algorithm 2 is well suited for SIMD on GPU architectures. We show that our GPU implementation is 30 times faster than its sequential CPU equivalent on the largest instances and provide a path forward to further improve our implementation. Then, we illustrate that the reduced Hessian computed is effective to track a suboptimal in a real-time setting.

6.1 Experimental setup

6.1.1 Hardware

Our workstation *Moonshot* is provided by Argonne National Laboratory. All the experiments run on a NVIDIA V100 GPU (with 32GB of memory) and `CUDA 11.3`. The system is equipped with a Xeon Gold 6140, used to run the experiments on the CPU (for comparison). For the software, the workstation works with Ubuntu 18.04, and we use Julia 1.6 for our implementation. We rely on our package `KernelAbstractions.jl` and `GPUArrays.jl` to generate parallel GPU code.

All the implementation is open-sourced, and an artifact is provided to reproduce the numerical results.[1]

6.1.2 Benchmark library

The test data represents various case instances (see Table 1) in the power grid community obtained from the open-source benchmark library PGLIB [2]. The number in the case name indicates the number of buses (graph nodes) n_v in the power grid: n_x is the number of variables, while n_p is the number of parameters (which is also equal to the dimension of the reduced Hessian and the parameter space $\mathbb{R}^{n_p}$).

6.2 Numerical results

6.2.1 Benchmark reduced Hessian evaluation

For the various problems described in Table 1, we benchmarked the computation of the reduced Hessian $\nabla^2 f$ for different batch sizes N. Each batch computes N columns of the reduced Hessian (which has

[1] Available on https://github.com/exanauts/ExaPF-Opt/tree/master/papers/pp2022

a fixed size of $n_p \times n_p$). Hence, the algorithm requires $N_b = div(n_p, N) + 1$ number of batches to evaluate the full Hessian.

Table 1: Case instances obtained from PGLIB.

Case	n_v	n_e	n_x	n_p
IEEE118	118	186	181	107
IEEE300	300	411	530	137
PEGASE1354	1,354	1,991	2,447	519
PEGASE2869	2,869	4,582	5,227	1,019
PEGASE9241	9,241	16,049	17,036	2,889
GO30000	30,000	35,393	57,721	4,555

In Figure 5, we compare on various instances (see Table 2) the reference CPU implementation together with the full reduced Hessian computation $\nabla^2 f$ on the GPU (with various batch sizes N). The CPU implementation uses the sparse LU solver UMFPACK with iterative refinement disabled[2] (it yields no numerical improvement, however, considerably speeds up the computation). We scale the time taken by the algorithm on the GPU by the time taken to compute the full reduced Hessian on the CPU: a value below 1 means that the GPU is faster than the CPU. We observe that the larger the number of batches N, the faster the GPU implementation is. The speed-up is not large on small instances (≈ 2 for IEEE118 and IEEE300), but we get up to a 30 times speed-up on the largest instance GO30000, when using a large number of batches. We observe also that the computation time decreases almost linearly as we increase the number of batches, until we reach the scalability limit of the GPU (generally, when $N \geq 256 = 2^8$).

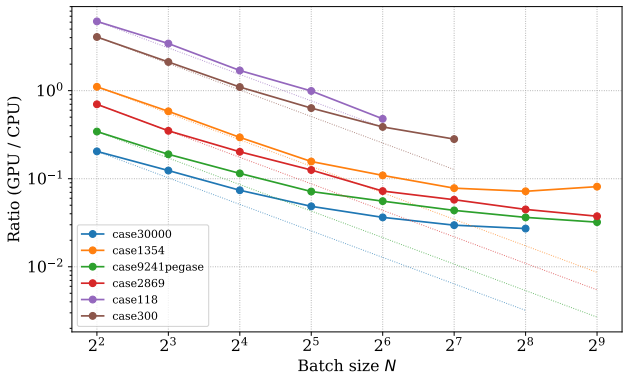

Figure 5: Parallel scaling of the total reduced Hessian accumulation $\nabla^2 f$ with batch size N: A ratio value < 1 indicates a faster runtime compared with that of UMFPACK and AutoDiff on the CPU in absolute time. The dotted lines indicate the linear scaling reference. Lower values imply a higher computational intensity.

Table 2: Size of key matrices (seed matrix W, multiple right-hand sides B, and final reduced Hessian $\nabla^2 f$) for a batch size of N. On GO30000, instantiating the three matrices $W, B, \nabla^2 f$ for $N = 256$ already takes 286MB in the GPU memory.

Cases	Dimensions		
	$W \in \mathbb{R}^{n_p \times N}$	$B \in \mathbb{R}^{n_x \times N}$	$\nabla^2 f \in \mathbb{R}^{n_p \times n_p}$
IEEE118	$107 \times N$	$181 \times N$	107×107
IEEE300	$137 \times N$	$530 \times N$	137×137
PEGASE1354	$519 \times N$	$2,447 \times N$	519×519
PEGASE2869	$1,019 \times N$	$5,227 \times N$	$1,019 \times 1,019$
PEGASE9241	$17,036 \times N$	$17,036 \times N$	$2,889 \times 2,889$
GO30000	$30,000 \times N$	$35,393 \times N$	$4,555 \times 4,555$

[2]We set the parameter `UMFPACK_IRSTEP` to 0.

Figure 6 shows the relative time spent in the linear algebra and the automatic differentiation backend. At small batch sizes N the bottleneck is the linear solver, while the derivative computation taking only a small fraction of the total runtime. As we increase N the computational intensity is higher, leading to a more efficient usage of the linear solver `cuSOLVER_RF` with multiple right-hand sides. However, the batched automatic differentiation backend leads a to a smaller speed-up, increasing the fraction of the total runtime spent in the block `BatchAutoDiff`. But even for $N = 512$, the forward and backward solves still amount to more than two-third of the total running time.

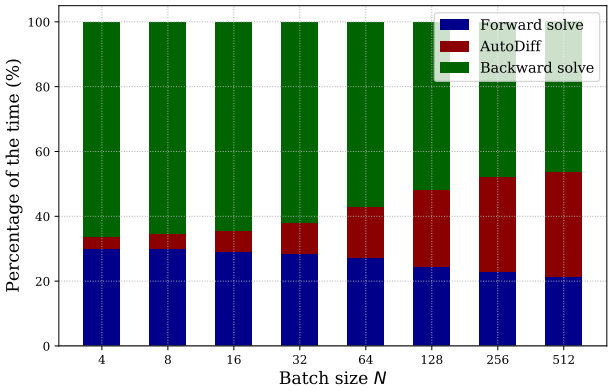

Figure 6: Decomposition of the runtime against the number of batch N, on case PEGASE 9241. With increasing batch size the derivative computation becomes more dominant. Notice the required number of batches $div(n_p, N)$ for the reduced Hessian $\nabla^2 f$ decreases with increasing N.

6.2.2 Discussion

Our analysis shows that the reduced Hessian scales with the batch size, while hitting a utilization limit for larger test cases. First, our kernels may still have potential for improvement, thus further improving utilization scaling as long as we do not hit the memory capacity limit. However, the sparsity of the power flow problems represents a worst-case problem for SIMD architectures, common in graph-structured applications. Indeed, in contrast to PDE-structured problems, graphs are difficult to handle in SIMD architectures because of their unstructured sparsity patterns.

The second bottleneck is the sparse linear algebra: As Figure 6 shows, the time spent in `cuSOLVER_RF` amounts to more than two-thirds of the total computation time on the GPU. We hope that future improvements in `cuSOLVER_RF` will help decrease that time further. An alternative is to implement multiple right-hand sides, block preconditioned, iterative linear solvers. They represent a promising portable alternative for other GPU architectures such as AMD or Intel Xe that may not provide an optimized direct linear solver like `cuSOLVER_RF`.

6.3 Real-time tracking algorithm

Finally, we illustrate the benefits of our reduced Hessian algorithm by embedding it in a real-time tracking algorithm.

Let $\boldsymbol{w}_t = (\boldsymbol{P}_t^d, \boldsymbol{Q}_t^d)$ be the loads in (3), indexed by time t and updated every minute. In that setting, the reduced space problem is parameterized by the loads $\boldsymbol{w}_t$:

$$\min_{\boldsymbol{p}_t} \; F(\boldsymbol{p}_t; \boldsymbol{w}_t) := f\big(x(\boldsymbol{p}_t), \boldsymbol{p}_t; \boldsymbol{w}_t\big) \; . \tag{17}$$

The real-time algorithm aims at tracking the optimal solutions $\boldsymbol{p}_t^\star$ associated with the sequence of problems (17), for all time t. To achieve this, we update the tracking point $\boldsymbol{p}_t$ by exploiting the reduced Hessian at every minute. The procedure is the following:

1. For new loads $\boldsymbol{w}_t = (\boldsymbol{P}_t^d, \boldsymbol{Q}_t^d)$, compute the reduced gradient $\boldsymbol{g}_t = \nabla_{\boldsymbol{p}} F(\boldsymbol{p}_t; \boldsymbol{w}_t)$ and the reduced Hessian $H_t = \nabla^2_{\boldsymbol{pp}} F(\boldsymbol{p}_t; \boldsymbol{w}_t)$ using Algorithm 2.
2. Update the tracking control $\boldsymbol{p}_t$ with $\boldsymbol{p}_{t+1} = \boldsymbol{p}_t + \boldsymbol{d}_t$, where $\boldsymbol{d}_t$ is a descent direction computed as solution of the dense linear system

$$H_t \, \boldsymbol{d}_t = -\boldsymbol{g}_t \, . \tag{18}$$

In practice, we use the dense Cholesky factorization implemented in `cuSOLVER` to solve the dense linear system (18) efficiently on the GPU.

We compare the tracking controls $\{\boldsymbol{p}_t\}_{t=1,\cdots,T}$ with the optimal solutions $\{\boldsymbol{p}_t^\star\}_{t=1,\cdots,T}$ associated to the sequence of optimization problems (17). Note that solving (17) to optimality is an expensive operation, involving calling a nonlinear optimization solver. On the contrary, the real-time tracking algorithm involves only (i) updating the gradient and the Hessian for the new loads $\boldsymbol{w}_t$ and (ii) solving the dense linear system (18).

We depict in Figure 7 the performance of the real-time tracking algorithm, by comparing with the optimal solution. We observe in the first subplot that the operating cost associated to $\{\boldsymbol{p}_t\}_t$ is close to the optimal cost associated to $\{\boldsymbol{p}_t^\star\}_t$. The second subplot depicts the evolution of the absolute difference $|\boldsymbol{p}_t - \boldsymbol{p}_t^\star|$, component by component. We observe that the difference remains tractable: the median (Quantile 50%) is almost constant, and close to 10^{-2} (which in our case is not a large deviation from the optimum) whereas the maximum difference remains below 0.5. At each time t, the real-time algorithm takes in average $0.09s$ to update $\boldsymbol{p}_t$, with

1. $0.05s$ to compute the gradient $\boldsymbol{g}_t$ and the reduced Hessian H_t (the reduced Hessian is evaluated with a number of batches $N = 256$).
2. $0.04s$ to solve the dense linear system (18) with `cuSOLVER`.

Hence, this real-time use case leverages the high parallelism of our algorithm to evaluate the reduced Hessian.

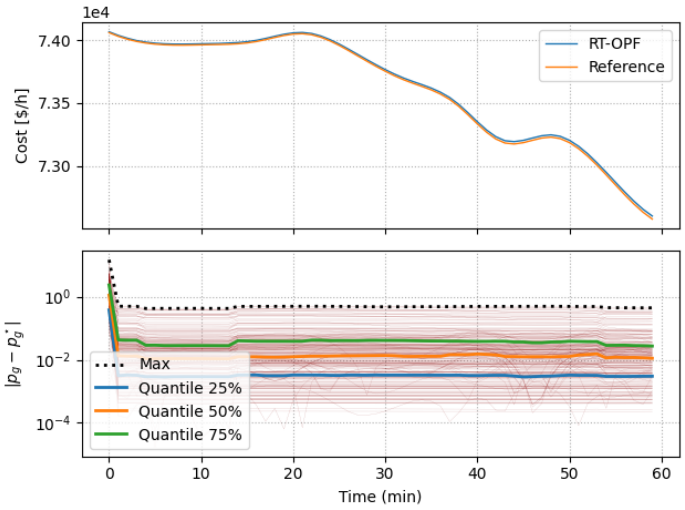

Figure 7: Performance of the real-time tracking algorithm on PEGASE1354, comparing with the optimal solutions. The real-time algorithm runs every minute, during one hour. The first plot shows the evolution of the operating cost along time, whereas the second plot shows the evolution of the absolute difference between the tracking control p_t and the optimum $p_t^\star$.

7 Conclusion

In this paper we have devised and implemented a practical batched algorithm (see Algorithm 2) to extract the second-order sensitivities from a system of nonlinear equations on SIMD architectures. Our implementation on NVIDIA GPUs leverages the programming language Julia to generate portable kernels and differentiated code. We have observed that the batch code achieves a 30x speed-up on the largest cases, compared with a CPU implementation. We have illustrated the interest of the reduced Hessian when used inside a real-time tracking algorithm.

Our solution adheres to the paradigm of differential and composable programming, leveraging the built-in metaprogramming capabilities of Julia. The method can be extended to other classes of problems (such as uncertainty quantification, optimal control, trajectory optimization, or PDE-constrained optimization). By reducing sparse problems to dense ones, we believe that this approach is promising to solve large-scale nonlinear optimization problems on GPU architectures.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] S. Babaeinejadsarookolaee, A. Birchfield, R. D. Christie, C. Coffrin, C. DeMarco, R. Diao, M. Ferris, S. Fliscounakis, S. Greene, R. Huang, et al. The power grid library for benchmarking AC optimal power flow algorithms. arXiv preprint arXiv:1908.02788, 2019.

[3] L. Beda et al. Programs for automatic differentiation for the machine besm. Precise Mechanics and Computation Techniques, Academy of Science, Moscow, 1959.

[4] T. Besard, C. Foket, and B. De Sutter. Effective extensible programming: unleashing Julia on GPUs. IEEE Transactions on Parallel and Distributed Systems, 30(4):827–841, 2018.

[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. SIAM Rev., 59(1):65–98, 2017.

[6] L. T. Biegler, O. Ghattas, M. Heinkenschloss, and B. van Bloemen Waanders. Large-scale PDE-constrained optimization: an introduction. In Large-Scale PDE-Constrained Optimization, pages 3–13. Springer, 2003.

[7] J. Blühdorn, N. R. Gauger, and M. Kabel. AutoMat – automatic differentiation for generalized standard materials on gpus. arXiv preprint arXiv:2006.04391, 2020.

[8] A. E. Bryson and W. F. Denham. A steepest-ascent method for solving optimum programming problems. Journal of Applied Mechanics, 29:247, 1962.

[9] T. A. Davis. Algorithm 832: UMFPACK v4.3–an unsymmetric-pattern multifrontal method. ACM Trans. Math. Software, 30(2):196–199, 2004.

[10] H. W. Dommel and W. F. Tinney. Optimal power flow solutions. IEEE Transactions on power apparatus and systems, (10):1866–1876, 1968.

[11] J. C. Gilbert. Automatic differentiation and iterative processes. Optimization methods and software, 1(1):13–21, 1992.

[12] M. Grabner, T. Pock, T. Gross, and B. Kainz. Automatic differentiation for GPU-accelerated 2d/3d registration. In Advances in automatic differentiation, pages 259–269. Springer, 2008.

[13] A. Griewank and A. Walther. Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM, 2008.

[14] J. C. Hückelheim, P. D. Hovland, M. M. Strout, and J.-D. Müller. Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation. Optimization Methods and Software, 33(4-6):672–693, 2018.

[15] M. Innes. Flux: Elegant machine learning with julia. Journal of Open Source Software, 3(25):602, 2018.

[16] J. Kardos, D. Kourounis, and O. Schenk. Reduced-space interior point methods in power grid problems. arXiv preprint arXiv:2001.10815, 2020.

[17] S. Linnainmaa. Taylor expansion of the accumulated rounding error. BIT Numer. Math., 16(2):146–160, 1976.

[18] W. S. Moses and V. Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In Advances in Neural Information Processing Systems, volume 33, pages 12472–12485. 2020.

[19] U. Naumann. The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation. SIAM, 2012.

[20] J. F. Nolan. Analytical differentiation on a digital computer. PhD thesis, Massachusetts Institute of Technology, 1953.

[21] D. Papadimitriou and K. Giannakoglou. Direct, adjoint and mixed approaches for the computation of hessian in airfoil design problems. International journal for numerical methods in fluids, 56(10):1929–1943, 2008.

[22] A. Paszke et al. PyTorch: An imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[23] B. A. Pearlmutter. Fast exact multiplication by the hessian. Neural computation, 6(1):147–160, 1994.

[24] J. Revels, T. Besard, V. Churavy, B. D. Sutter, and J. P. Vielma. Dynamic automatic differentiation of GPU broadcast kernels. CoRR, abs/1810.08297, 2018.

[25] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. arXiv preprint arXiv:1607.07892, 2016.

[26] Y. Tang, K. Dvijotham, and S. Low. Real-time optimal power flow. IEEE Transactions on Smart Grid, 8(6):2963–2973, 2017.

[27] W. F. Tinney and C. E. Hart. Power flow solution by Newton's method. IEEE Transactions on Power Apparatus and systems, (11):1449–1460, 1967.