Towards nonlinear programming on the GPU

JuMP-dev 2021

Exanauts

Mathematics and Computer Science Division
Argonne National Laboratory

July, 2021

Who we are?

We are a team of enthusiastic computational mathematicians at Argonne National Lab

- ✓ Mihai Anitescu (PI, ANL)
- ✓ Kibaek Kim (ANL)
- ✓ Youngdae Kim (ANL)
- ✓ Daniel Adrian Maldonado (ANL)
- ✓ Alexis Montoison (GERAD)
- ✓ Vishwas Rao (ANL)
- ✓ Michel Schanen (ANL)
- ✓ Sungho Shin (ANL)
- ✓ Anirudh Subramanyam (ANL)

ECP
EXASCALE COMPUTING PROJECT

ExaSGD project

- Optimizing Stochastic Grid Dynamics at ExaScale
- Leverage new GPU-centric HPC architectures

Nonlinear optimization: the current landscape

Nonlinear optimization problem

Let $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ be two generic nonconvex smooth functions

$$\min_{x \in \mathbb{R}^n} \; f(x) \quad \text{s.t.} \;\; g(x) \leq 0$$

Numerical optimization algorithms depend on two key routines

1. **Derivatives:** Explicit derivatives, Finite Differences, Automatic Differentiation
2. **Linear solve:** Solve KKT system to compute descent direction

$$(\nabla^2_{xx} \ell_k) d_k = -\nabla_x \ell_k$$

Nonlinear optimization: the current landscape

Nonlinear optimization problem

Let $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ be two generic nonconvex smooth functions

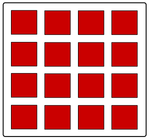
$$\min_{x \in \mathbb{R}^n} \; f(x) \quad \text{s.t.} \; g(x) \leq 0$$

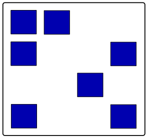
Numerical optimization algorithms depend on two key routines

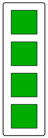
1. **Derivatives:** Explicit derivatives, Finite Differences, Automatic Differentiation
2. **Linear solve:** Solve KKT system to compute descent direction

$$(\nabla^2_{xx} \ell_k) d_k = -\nabla_x \ell_k$$

$n < 1,000$	$1,000 < n < 500,000$	$n > 500,000$
Small scale	Medium scale	Large scale


Dense LinAlg
LAPACK


Sparse Linalg
HSL MA27/MA57


Matrix-vector products

Julia's optimization ecosystem is already quite consistent...

The current ecosystem

- ✓ ForwardDiff.jl for forward-mode Automatic Differentiation
- ✓ JuMP as a nonlinear modeler
  - Expression graph for nonlinear expr
  - Custom reverse-mode AD
  - Graph coloring for sparse Hessian
- ✓ Various optimization solvers available
  - Optim.jl for dense optimization
  - Wrappers for nonlinear solvers (Ipopt, Knitro, NLOpt)

The vanguard

- ✓ New packages for reverse-mode AD
  - + Zygote.jl
  - + Enzyme.jl
  - + Diffractor.jl
- ✓ New nonlinear optimization solvers
  - + JuliaSmoothOptimizers
  - + MadNLP.jl
- + **Premium support for CUDA**

… but porting nonlinear optimization to the GPU is still non trivial

Two inconvenient truths about GPUs

- **Fact 1:** Reverse mode Automatic Differentiation is hard to parallelize
  - In Julia, excellent support of forward mode automatic differentiation on GPUs
  - But, parallelizing reverse mode is difficult for large model (race condition, storing intermediate)

- **Fact 2:** Scarce support of sparse linear algebra
  - No equivalent to MA27/MA57 on GPUs yet (Tasseff et al., 2019)
  - GPU are not efficient to factorize matrix with unstructured sparsity…

What if we exploit the structure of the problem?

Key idea

We exploit the structure of the problem to solve optimization problems on the GPU

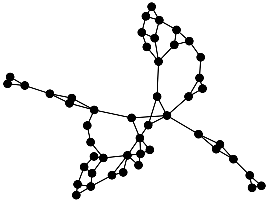
We investigate two different approaches

1. Reduced-space approach

$$\begin{aligned} \min_{x,u} \; & f(x,u) \\ \text{s.t.} \; & g(x,u) = 0 \end{aligned}$$

2. Decomposition-based approach

$$\begin{aligned} \min_{x,z} \; & f(x) + g(z) \\ \text{s.t.} \; & Fx + Gz = d \end{aligned}$$



Both examples will be tested on the classical Optimal Power Flow problem

Outline

Solving structured optimization problems on the GPU

Towards generic nonlinear programming on the GPU

Problem 1: Reduced space approach

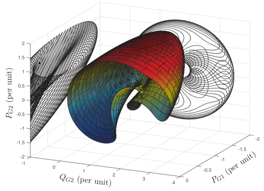


Figure: Nonlinear power flow (from Hiskens and Davy (2001))

Most real-life nonlinear problems encompasses a set of physical constraints

$$G(x, u) = 0$$

with $\boldsymbol{x}$ a state and $\boldsymbol{u}$ a control

Domain	G
Optimal control	Dynamics
PDE-constrained optimization	PDE
Optimal power flow	Power flow

Physically-constrained optimization problem

$$\min_{x,u} F(x, u)$$

$$\text{s.t. } G(x, u) = 0 \, , \quad H(x, u) \leq 0$$

Projecting the problem into the powerflow manifold

- If $\nabla_x G$ is non-singular, then Implicit Function theorem applies
- For each u, there exists a local function $\underline{x}(u)$ such that

$$G(\underline{x}(u), u) = 0$$

(numerically, the nonlinear equation is inverted with Newton-Raphson)

Reduced problem

Let $f(\boldsymbol{u}) := F(\underline{x}(\boldsymbol{u}), \boldsymbol{u})$ and $h(\boldsymbol{u}) := H(\underline{x}(\boldsymbol{u}), \boldsymbol{u})$.

$$\min_{\boldsymbol{u}} \; f(\boldsymbol{u}) \quad \text{s.t.} \quad h(\boldsymbol{u}) \leq 0$$

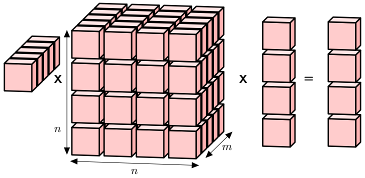
Reduced gradient

Let $F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}$ a differentiable function
Then, the function $f(\boldsymbol{u}) := F(\underline{x}(\boldsymbol{u}), \boldsymbol{u})$ is *differentiable*, and

$$\nabla f(\boldsymbol{u}) = \underbrace{\nabla_u F}_{n_u} + (\underbrace{\nabla_u G}_{n_x \times n_u})^\top \underbrace{\boldsymbol{\lambda}}_{n_x} \quad \text{with} \quad (\underbrace{\nabla_x G}_{n_x \times n_x})^\top \boldsymbol{\lambda} = - \underbrace{\nabla_x F}_{n_x}$$

The vector $\boldsymbol{\lambda} \in \mathbb{R}^{n_x}$ is the first-order adjoint

Reduced Hessian: dense, dense, dense!



Can we extract second-order information as well? Yes!

- We derive two first-order adjoints ψ and $\boldsymbol{z}$, using the *adjoint-adjoint* method (Wang et al., 1992)
- Involve only *Hessian-vector products*!
- Reduced Hessian $\nabla^2 f$ is *dense* (dimension $n_u \times n_u$)

Reduced Hessian

Let $\boldsymbol{w} \in \mathbb{R}^{n_u}$ be a vector and $\widehat{G}$ the first-order residual:

$$\widehat{G}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\lambda}) := \nabla_x F(\boldsymbol{x}, \boldsymbol{u}) + \nabla_x G(\boldsymbol{x}, \boldsymbol{u})^\top \boldsymbol{\lambda}$$

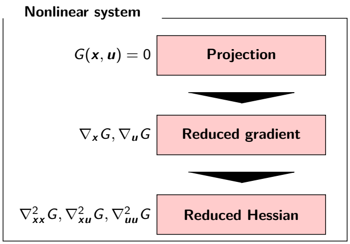
The Hessian-vector product $(\nabla^2 f)\boldsymbol{w}$ is equal to

$$(\nabla^2 f)\boldsymbol{w} = (\nabla^2_{\boldsymbol{uu}} F)\,\boldsymbol{w} + \boldsymbol{\lambda}^\top (\nabla^2_{\boldsymbol{uu}} G)\,\boldsymbol{w} + (\nabla_{\boldsymbol{u}} G)^\top \psi + (\nabla^2_{\boldsymbol{ux}} F)^\top \boldsymbol{z} + \boldsymbol{\lambda}^\top (\nabla^2_{\boldsymbol{ux}} G)^\top \boldsymbol{z}$$

with the second-order adjoints $(\boldsymbol{z}, \psi)$ defined as

$$\begin{cases} (\nabla_{\boldsymbol{x}} G) \quad \boldsymbol{z} = -(\nabla_u G)\ \boldsymbol{w} \\ (\nabla_{\boldsymbol{x}} G)^\top \psi = -(\nabla_u \widehat{G})\boldsymbol{w} - (\nabla_{\boldsymbol{x}} \widehat{G})\boldsymbol{z}\ , \end{cases}$$

Porting Automatic Differentiation to the GPU



Nonlinear system
$G(\boldsymbol{x}, \boldsymbol{u}) = 0$
Projection
$\nabla_{\boldsymbol{x}} G, \nabla_{\boldsymbol{u}} G$
Reduced gradient
$\nabla^2_{\boldsymbol{xx}} G, \nabla^2_{\boldsymbol{xu}} G, \nabla^2_{\boldsymbol{uu}} G$
Reduced Hessian

- **Projection:**
  - ✓ Solve $G(x, u) = 0$ with Newton-Raphson
    $$x_{k+1} = x_k - (\nabla_x G)^{-1} G(x_k, u)$$
  - ✓ Sparse Jacobian $\nabla_x G$ computed with forward-mode + graph coloring
- **Reduced Hessian:**
  - ✓ Forward-over-Reverse, directly on the GPU
  - ✓ No coloring involved!
  - ✓ Batch automatic differentiation for reverse mode
  - ✓ Sparse linear systems solved in batch

Automatic Differentiation backend

- **Forward-mode:** custom layer on top of ForwardDiff.jl,
  - ✓ Almost straightforward to use on the GPU, thanks to Revels et al. (2018)
  - ✓ We add support to Jacobian coloring on GPU
- **Reverse-mode:** hand-written adjoints
  - × Zygote.jl works on the GPU, but was not adapted to our use-case...
  - × Race-conditions (`getindex` becomes `setindex!` in reverse mode)
  - → hopefully, soon implemented with Enzyme.jl (Moses and Churavy, 2020)!! (works directly at the LLVM level, compatible with `KernelAbstractions.jl`)

An implementation running entirely on the GPU

- All callbacks run entirely on the GPU (using `KernelAbstractions.jl` as portability layer)
- Sparse linear systems solved either with
  - ✓ Direct LU solver (`cusolverRF`)
  - ✓ Iterative BICGSTAB (`Krylov.jl`)

ReducedSpaceEvaluator abstraction

All callbacks (objective, gradient, Hessian) abstracted in a `ReducedSpaceEvaluator`

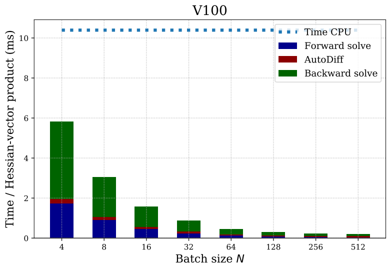
- Follow closely the interface of `NlpModels.jl`
- Interfaced with `MOI`!!

```
    opf = ReducedSpaceEvaluator("case9241pegase.m")
    u = initial(opf)   # initial point from MATPOWER
    update!(opf, u)    # Solve PF
    objective(opf, u)
    gradient(opf, u)   # Reduced gradient
    hessian(opf, u)    # Reduced Hessian
```

Results

Evaluating full reduced Hessian in parallel
(on `case9241pegase`, $n_u = 2,889$)

i) Reduced space: CPU versus GPU



ii) Reduced space versus full space

lib	device	space	time
AMPL	CPU	full-space	130ms
ExaPF	GPU	reduced-space	350ms

Table: Time to evaluate the Hessian of the Lagrangian

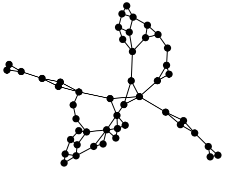
Take-away messages

- ✓ Fast Hessian evaluation using batching
- ✓ Dense Hessian easy to factorize on the GPU using CUDA/LAPACK
- × Memory heavy when using large number of batches
- × Bottleneck is currently the sparse linear algebra kernel

Next:

1. Abstracting things further to get a generic paramerized nonlinear solver in Julia
$$G(x, u) = 0$$
2. Develop optimization algorithms in the reduced space to optimize w.r.t. the control u

Problem 2: decomposition based approach



- × We saw that sparse linear algebra is a bottleneck on the GPU
- ✓ What if we use an operator-splitting method instead?

$$\min_{x,z} f(x) + g(z)$$
$$\text{s.t. } Ax + Bz = d$$

- Graph-based problems have a structure amenable to decomposition
- Allow to solve many small optimization subproblems in parallel!

ADMM-based decomposition

$$x^{k+1} \in \arg\min_x f(x) + (\lambda^k)^\top Ax + \frac{\rho}{2}\|Ax + Bz^k - d\|^2$$
$$z^{k+1} \in \arg\min_z g(z) + (\lambda^k)^\top Bz + \frac{\rho}{2}\|Ax^{k+1} + Bz - d\|^2$$
$$\lambda^{k+1} = \lambda^k + \rho(Ax^{k+1} + Bz^{k+1} - d)$$

ExaTron: a large-scale optimization solver for GPUs

1. Decompose the OPF problem component by component:
   - **Bus subproblems:** analytical expression available for solution
   - **Generator subproblems:** analytical expression available for solution
   - **Branch subproblems:** bounded (dense) nonlinear optimization problem
   - **Coupling constraints:** formulates as linear constraints $Ax + Bz = d$

$$\min_{\underline{x}_1 \le x_1 \le \overline{x}_1} f_1(x_1) \qquad \min_{\underline{x}_2 \le x_2 \le \overline{x}_2} f_2(x_2) \qquad \dots \qquad \min_{\underline{x}_N \le x_N \le \overline{x}_N} f_N(x_N)$$

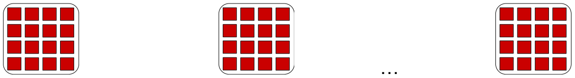


Figure: Once the problem decomposed, we get many small dense optimization subproblems to solve

2. Solve the branch subproblems in parallel, on the GPU
   - Dense optimization problems
   - First and Second-order derivatives evaluated in parallel
   - Each subproblem solved in parallel on the GPU, with a dense Tron algorithm
3. Coordination with an ADMM algorithm
   - Implement the two-level ADMM algorithm of Sun and Sun (2021)
   - Convergence guaranteed, even for nonconvex problems

→ For detailed results, see Youngdae Kim's talk at JuliaCon!

Implementation

For case9241pegase: 16,049 subproblems (all solved in parallel)

Solver	device	time
Knitro+AMPL	CPU	110s
ExaTron	GPU	10.5s

Table: Time to solve `case9241pegase` (N.B.: This comparison is given only to give an order of magnitude, as we cannot compare ExaTron's stopping criterion with Knitro's one)

ExaTron.jl

- Julia implementation, on top of CUDA.jl
- No dependency to linear algebra (only dense matrix-vector products)!
- Tested extensively on Summit in a multi-GPU setting

Outline

Solving structured optimization problems on the GPU

Towards generic nonlinear programming on the GPU

Porting the optimization solver to the GPU

Coming back to the generic nonlinear problem

$$\min_x \; f(x) \quad \text{s.t.} \;\; g(x) \leq 0$$

Promising works ahead!

- **Hiop**
  - ✓ Currently developed at LLNL
  - ✓ Implemented in C++, using RAJA/Umpire as portability layer
  - ✓ Mixed dense/sparse linear algebra for the KKT system
- **MadNLP.jl**
  - ✓ Developed by Sungho Shin (Shin et al., 2020)
    $\rightarrow$ see MadNLP lightning talk at JuMP-dev!
  - ✓ Allow to deport the factorization of the KKT system on the GPU (LapackGPU)
  - ✓ Modularity!

Open question

Suppose we have a parameterized nonlinear problem, defined as

$$\min_x \; f(x, p)$$
$$\text{s.t.} \;\; g(x, p) \leq 0$$

For parameters $p_1, \cdots, p_N$, can we solve the problem in batch on the GPU?

Porting the modeler to the GPU

Challenge

How to handle vectorized operations at the modeler level?

```
m = Model()
@vector(m, Pg, 1:ngen)  # generators's active power
@vector(m, Vm, 1:nbus)  # voltage magnitudes
@vector(m, Va, 1:nbus)  # voltage angles
@params(m, Pd, 1:nbus)  # active loads
@graph(m, G, Yre)
@spequation(m, Cg * Pg .- Pd .= vm .* sum(vm[l]
                                .* (Yre[k,l] * cos(va .- va[l])
                                 .+ Yim[k,l] * sin(va .- va[l])),
                                   for l in G)
```

→ Can we automatically generate GPU kernels?

→ Open question: how to specify the problem's structure, such as

$$G(x, u) = 0\,, \quad Fx + Gz = d$$

Towards a StructJuMP package at the MOI level?

Conclusion

Take away

- Large-scale optimization tractable on the GPU by exploiting the structure
  - ✓ Reduced space approach
  - ✓ Decomposition-based approach
- Generic nonlinear programming is still an open-problem
- We believe that we will observe tremendous changes in the following years!
  - ✓ Towards a mature reverse mode automatic differentiation in Julia
  - ✓ Efficient sparse linear algebra routines on GPU accelerators

tl;dr: we would be interested to contribute to the JuMP's NLP rewrite ;-)

Links

- Slides available at: https://frapac.github.io/pdf/jump-dev-2021.pdf
- Reduced-space code: https://github.com/exanauts/ExaPF.jl/
- ExaTron: https://github.com/exanauts/ExaTron/

References I

Hiskens, I. A. and Davy, R. J. (2001). Exploring the power flow solution space boundary. IEEE transactions on power systems, 16(3):389–395.

Moses, W. S. and Churavy, V. (2020). Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. arXiv preprint arXiv:2010.01709.

Revels, J., Besard, T., Churavy, V., De Sutter, B., and Vielma, J. P. (2018). Dynamic automatic differentiation of gpu broadcast kernels. arXiv preprint arXiv:1810.08297.

Shin, S., Coffrin, C., Sundar, K., and Zavala, V. M. (2020). Graph-based modeling and decomposition of energy infrastructures. arXiv preprint arXiv:2010.02404.

Sun, K. and Sun, X. A. (2021). A two-level admm algorithm for ac opf with convergence guarantees. IEEE Transactions on Power Systems.

Tasseff, B., Coffrin, C., Wächter, A., and Laird, C. (2019). Exploring benefits of linear solver parallelism on modern nonlinear optimization applications. arXiv preprint arXiv:1909.08104.

Wang, Z., Navon, I. M., Le Dimet, F.-X., and Zou, X. (1992). The second order adjoint analysis: theory and applications. Meteorology and atmospheric physics, 50(1):3–20.