Streamlining nonlinear programming on GPUs: towards a vectorized modeler?

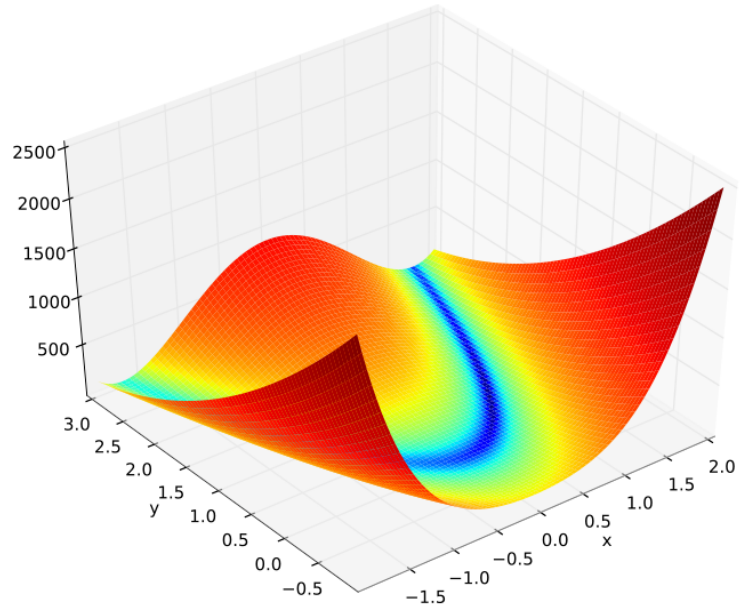
JuMP-dev 2022

François Pacaud Michel Schanen

Mathematics and Computer Science Division (MCS)
Argonne National Laboratory

July, 2022

Of Julia, automatic differentation and nonlinear programming



$$\min_{x \in \mathbb{R}^n} \; f(x)$$

$$\text{subject to} \quad g(x) \leq 0$$

Requirements

- Fast evaluations of gradient ∇f and (sparse) Jacobian ∇g
- Fast evaluation of (sparse) Hessian

$$W = \nabla^2 f(x) + \sum_i y_i \nabla^2 g(x)$$

- Automatic differentation is a good match for nonlinear programming (Griewank!)
- Mature packages:
  AMPL, GAMS, JuMP.jl

Benchmark: the optimal power flow problem

Why is the NLP community obsessed with OPF?

- Problem is (super) sparse, large-scale
  - + Require second-order derivatives and coloring
- Established benchmarks available (MATPOWER, pglib-opf)
  - + Recent ARPAE-GO competition
- Good proxy problem for various problems in engineering

Recent initiative to benchmark various AD backends: `rosetta-opf`

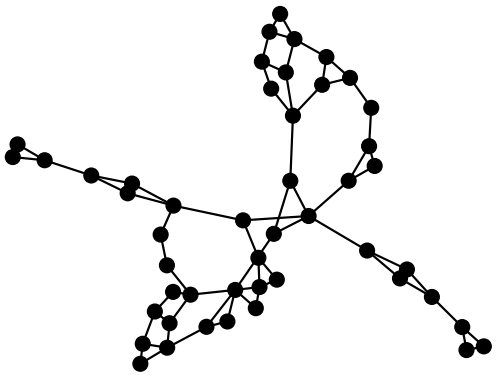


Figure: OPF has a nice graph structure $G = (\mathcal{V}, \mathcal{E})$!

Simplified OPF (polar form)

$$\min_{v,\theta,p_g,q_g} \sum_g c^g(p_g)$$

$$\text{subject to} \quad p_g^i - p_d^i = \sum_{(i,j)\in\mathcal{E}} p_{ij}(v,\theta) \quad \forall i \in \mathcal{V}$$

$$q_g^i - q_d^i = \sum_{(i,j)\in\mathcal{E}} q_{ij}(v,\theta) \quad \forall i \in \mathcal{V}$$

Implementing the OPF in JuMP (1)

Looking at `rosetta-opf`'s code:

1. Step 1: Define the $4n_\ell$ nonlinear branch flows, for each branch:

```
# From side of the branch flow
JuMP.@NLconstraint(model, p_fr ==  (g+g_fr)/ttm*vm_fr^2 + (-g*tr+b*ti)/ttm*(vm_fr*vm_to*cos(va_fr-va_to)) + (-b*tr-g*ti)/ttm*(vm_fr*vm_to*sin(va_fr-va_to)) )
JuMP.@NLconstraint(model, q_fr == -(b+b_fr)/ttm*vm_fr^2 - (-b*tr-g*ti)/ttm*(vm_fr*vm_to*cos(va_fr-va_to)) + (-g*tr+b*ti)/ttm*(vm_fr*vm_to*sin(va_fr-va_to)) )

# To side of the branch flow
JuMP.@NLconstraint(model, p_to ==  (g+g_to)*vm_to^2 + (-g*tr-b*ti)/ttm*(vm_to*vm_fr*cos(va_to-va_fr)) + (-b*tr+g*ti)/ttm*(vm_to*vm_fr*sin(va_to-va_fr)) )
JuMP.@NLconstraint(model, q_to == -(b+b_to)*vm_to^2 - (-b*tr+g*ti)/ttm*(vm_to*vm_fr*cos(va_to-va_fr)) + (-g*tr-b*ti)/ttm*(vm_to*vm_fr*sin(va_to-va_fr)) )
```

2. Step 2: Define the $2n_b$ linear balance equations, at each bus:

```
JuMP.@constraint(model,
    sum(p[a] for a in ref[:bus_arcs][i]) ==
    sum(pg[g] for g in ref[:bus_gens][i]) -
    sum(load["pd"] for load in bus_loads) -
    sum(shunt["gs"] for shunt in bus_shunts)*vm[i]^2
)

JuMP.@constraint(model,
    sum(q[a] for a in ref[:bus_arcs][i]) ==
    sum(qg[g] for g in ref[:bus_gens][i]) -
    sum(load["qd"] for load in bus_loads) +
    sum(shunt["bs"] for shunt in bus_shunts)*vm[i]^2
)
```

Observation

By design JuMP manipulates **scalar** expressions

Implementing the OPF in JuMP (2)

As a result JuMP[1] outputs $4n_\ell$ expression trees:

```
OrderedCollections.OrderedDict{MathOptInterface.Nonlinear.ConstraintIndex, MathOptInterface.Nonlinear.Constraint} w
  ConstraintIndex(1)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(2)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(3)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(4)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(5)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(6)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(7)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(8)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(9)  => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(10) => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(11) => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(12) => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(13) => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(14) => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(15) => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
  ConstraintIndex(16) => Constraint(Expression(MathOptInterface.Nonlinear.Node[Node(NODE_CALL_MULTIVARIATE, 2, -1),
```

Observation

Parallelization can happen only at the `outer` level!

For instance, if we look at `SymbolicAD.jl`'s code:

```
Threads.@threads for offsets in oracle.backend.offsets
    for c in offsets
        func = oracle.constraints[c.index]
        @inbounds g[c.index] = _eval_function(oracle, func, x)
    end
end
```

We evaluate each constraint in parallel using *multithreading*
What if we can evaluate all the constraints **in one block** instead?

[1] with the `MOI.Nonlinear` backend

Vectorizing the optimal power flow by manipulating vector expressions

Nonlinear basis (Lee et al., 2020)

We define the nonlinear basis

$$\psi(v,\theta) = \begin{bmatrix} \psi_\ell^C(\boldsymbol{v},\boldsymbol{\theta}) \\ \psi_\ell^S(\boldsymbol{v},\boldsymbol{\theta}) \\ \psi_k(\boldsymbol{v},\boldsymbol{\theta}) \end{bmatrix} \in \mathbb{R}^{2n_\ell+n_b}$$

with

$$\begin{aligned} \psi_\ell^C(v,\theta) &= v^f v^t \cos(\theta_f - \theta_t) && \forall \ell = 1,\cdots,n_\ell \\ \psi_\ell^S(v,\theta) &= v^f v^t \sin(\theta_f - \theta_t) && \forall \ell = 1,\cdots,n_\ell \\ \psi_k(v,\theta) &= v_k^2 && \forall k = 1,\cdots,n_b \end{aligned}$$

Once the nonlinearities factorized in ψ, we recover the different expressions with sparse matrix-vector operations (`SpMV`)

Vectorized model

The power flow constraints rewrite

$$\begin{bmatrix} C_g p_g - p_d \\ C_g q_g - q_d \end{bmatrix} + \underbrace{\begin{bmatrix} -\hat{G}^c & -\hat{B}^s & -G^d \\ \hat{B}^c & -\hat{G}^s & B^d \end{bmatrix}}_{M_{eq}} \psi(v,\theta) = 0$$

Evaluating the power flow model on the GPU

```
@kernel function basis_kernel!(
    cons, @Const(vmag), @Const(vang), @Const(f), @Const(t), nlines, nbus,
)
    i, j = @index(Global, NTuple)
    shift_cons = (j-1) * (nbus + 2*nlines)
    shift_bus  = (j-1) * nbus

    @inbounds begin
        if i <= nlines
            ℓ = i
            fr_bus = f[ℓ]
            to_bus = t[ℓ]
            Δθ = vang[fr_bus + shift_bus] - vang[to_bus + shift_bus]
            cosθ = cos(Δθ)
            cons[i + shift_cons] = vmag[fr_bus + shift_bus] * vmag[to_bus + shift_bus] * cosθ
        elseif i <= 2 * nlines
            ℓ = i - nlines
            fr_bus = f[ℓ]
            to_bus = t[ℓ]
            Δθ = vang[fr_bus + shift_bus] - vang[to_bus + shift_bus]
            sinθ = sin(Δθ)
            cons[i + shift_cons] = vmag[fr_bus + shift_bus] * vmag[to_bus + shift_bus] * sinθ
        elseif i <= 2 * nlines + nbus
            b = i - 2 * nlines
            cons[i + shift_cons] = vmag[b + shift_bus] * vmag[b + shift_bus]
        end
    end
end
```

Figure: `KernelAbstractions` kernel

As a shorthand, we write $g(v, \theta) = M_{eq}\psi(v, \theta)$.

Evaluation

1. We first evaluate the basis $\psi(v, \theta)$
   - ✓ SIMD parallelism
   - ✓ Implemented with `KernelAbstractions` for portability purpose
2. Then we recover the function g with one matrix operation (`SpMV`)
   - ✓ We leverage `cusparse` to deal with the sparsity

The model runs entirely on the GPU!

Porting the forward pass on the GPU

Forward-mode automatic differentation

- ✓ We can evaluate the Jacobian-product $(\nabla_x g)d$ using `ForwardDiff.jl`
  → `ForwardDiff.jl` is GPU-compatible (Revels et al., 2018)
- ✓ Exploit sparsity and use coloring to reduce the total number of Jacobian-product
- × Yet, sparsity detection is not automatic (we use MATPOWER's expression)

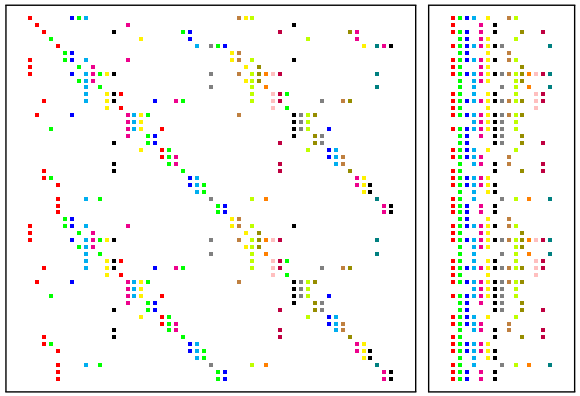


Figure: Matrix compression using coloring

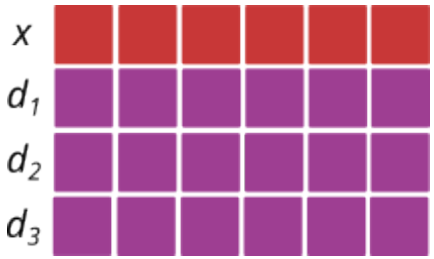


Figure: Memory representation of `ForwardDiff.Dual{Nothing, Float64, 3}` (Julia is column-oriented)

Linear algebra trick: if $d \in \mathbb{D}_p^n$ is a vector of dual numbers with p partials, evaluate $M_{eq}d$ in two steps:

1. Reinterpret d as a real-valued matrix $D \in \mathbb{R}^{(p+1)\times n}$
2. Evaluate $M_{eq}\,D$ with a `SpMM` operation (parallelize well if M_{eq} is CSR)

Porting the reverse pass on the GPU

Observations

- Implementing a parallel reverse pass is nontrial
  (every read becomes a write, leading to race condition)
- Scarce support of reverse-mode for nonlinear programming
  (we have hope in Enzyme, but not ready yet)

Our solution exploits the vectorized model:

- We have implemented manually a kernel to evaluate the adjoint $(\nabla\psi)^\top$
- Then recover the adjoint of the power-flow with linear algebra operations:

$$(\nabla g)^\top y = (\nabla\psi)^\top M_{eq}{}^\top y$$

Hessian evaluation

- We exploit the hand-coded adjoint and evaluate each Hessian-vector product with a forward-over-reverse pass *(require the adjoint to be compatible with* `ForwardDiff.jl`*)*
- Again, we use coloring and evaluate the full Hessian with one Hessian-vector product Hv
- Exploit the same linear algebra trick as in the forward pass
  (except $M_{eq}^\top Y$ is a CSC SpMM, requiring `atomic`*)*

Result #1: time to evaluate Hessian

Case	n_v	n_e	$p = n_{colors}$
IEEE118	118	186	27
IEEE300	300	411	24
PEGASE1354	1,354	1,991	28
PEGASE2869	2,869	4,582	35
PEGASE9241	9,241	16,049	**85**
ACTIVSg25K	25,000	32,230	36

Table: Benchmark cases from MATPOWER

Using JuMP's ReverseAD backend as a reference:

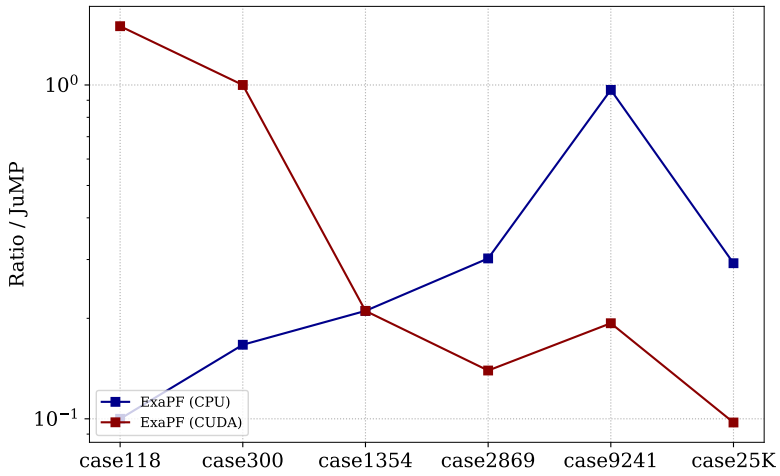


Figure: Ratio/JuMP, comparing CPU code and CUDA code (log scale)

Case	JuMP	CPU	CUDA
IEEE118	2	2	3
IEEE300	3	5	3
PEGASE1354	19	4	4
PEGASE2869	43	13	6
PEGASE9241	150	145	29
ACTIVSg25K	359	105	35

Table: Results: evaluation time in milliseconds

Result #2: time to solve OPF

	JuMP (ReverseAD)		JuMP (SymbolicAD)		Argos	
Case	#it	time (s)	#it	time(s)	#it	time (t/it)
1354pegase	41	**2.4**	41	**1.2**	40	**0.9** (0.02)
2869pegase	40	**4.2**	40	**2.4**	50	**2.0** (0.04)
9241pegase	70	**31.3**	70	**20.8**	69	**10.7** (0.16)
ACTIVSg10k	109	**36.8**	109	**22.5**	76	**7.9** (0.10)
ACTIVSg25k	102	**100.5**	102	**61.4**	86	**24.7** (0.29)
ACTIVSg70k	101	**313.0**	108	**238.8**	90	**89.8** (1.00)

Setup

- Solve OPF with MadNLP (filter line-search interior-point)
- Solve KKT system with HSL MA27

- JuMP uses formulation in `rosetta-opf`
  - ReverseAD: classical AD with custom reverse pass
  - SymbolicAD: uses `SymbolicAD.jl`
- Argos uses a custom formulation (with some variables removed)
- Comparison is only indicative!
  - Only a 2x time speed-up when using the GPU...
  - Argos is a prototype, nowhere near PowerModels' robustness

Extension to stochastic optimal power flow

Suppose we have N uncertainties

$$\xi_1, \cdots, \xi_N$$

Stochastic OPF

$$\begin{aligned} \min_{x_1,\cdots,x_N} \quad & \frac{1}{N} \sum_i f(x_i, \xi_i) \\ \text{subject to} \quad & (x_1, \cdots, x_N) \in C \\ & x_i = (v_i, \theta_i, p_g^i, q_g^i) \\ & g(x_i, \xi_i) = 0 \qquad \forall i = 1, \cdots, N \end{aligned}$$

C is the coupling set (generally porting on active power generations)

- The stochastic OPF requires to evaluate in parallel the parameterized functions

$$g(x_1, \xi_1), \cdots, g(x_N, \xi_N)$$

- Direct to parallelize on the GPU!
  (in fact that's how we have designed the KA kernel)

Evaluating a nonlinear model at Exascale

Exascale architecture

- GPU-centric: One node has 6 or 8 GPUs
- Fast communication between GPUs (unified memory)
- Setup favorable for MPI-CUDA (nicely supported by Julia)

tl;dr: Reimplementing `StructJuMP.jl` at Exascale:

- Dispatch the evaluation of the derivatives on N GPUs
- Assemble the resulting KKT system with a Schur-complement approach (ala PIPS)

Scaling results

Experiment 1

- Run Argos on one GPU, **solving the KKT system on the GPU with Schur-complement approach**
- Compare with JuMP, KKT system solved with MA27

		JuMP (ReverseAD)		JuMP (SymbolicAD)		Argos (1 GPU)	
Case	N	AD (s)	tot (s)	AD (s)	time(s)	AD (s)	tot (s)
1354pegase	10	4.6	7.6	1.2	3.0	0.2	1.9
1354pegase	30	15.2	32.1	3.2	9.4	0.5	10.3
1354pegase	60	30.4	72.1	7.5	29.6	1.0	34.0
9241pegase	10	41.6	109.2	9.0	43.4	3.8	68.2
9241pegase	30	/	/	52.7	284.2	10.8	411.9

Table: Comparing against JuMP (CPU)

Experiment 1

Dispatch the resolution on 2 GPUs

		Argos (1 GPU)		Argos (2 GPU)	
Case	N	AD (s)	time(s)	AD (s)	tot (s)
1354pegase	10	0.2	1.9	0.2	1.1
1354pegase	30	0.5	10.3	0.3	3.7
1354pegase	60	1.0	34.0	0.5	10.7
9241pegase	10	3.8	68.2	2.1	27.7
9241pegase	30	10.8	411.9	5.6	130.4

Next steps?

Take away message

It's hard to beat state-of-the-art! (even when using GPUs)

We believe that a vectorized modeler is the way to go to exploit emerging SIMD architectures (GPU, cerebras,...)

$$(v, \theta) \quad \rightsquigarrow \quad \psi(v, \theta) \quad \rightsquigarrow \quad M_{eq}\psi(v, \theta)$$

How to implement a vectorized modeler?

- The user implements custom functions, and possibly provides their adjoints
  - Integrate differentation rules with `ChainRulesCores.jl`
  - In the future, we hope we can extract the adjoint of any kernels with Enzyme
- Revisit JuMP's AST to manipulate vectorized expressions
  - Towards an `AbstractExpressionTree` in `MOI.Nonlinear`?

References I

Lee, D., Turitsyn, K., Molzahn, D. K., and Roald, L. A. (2020). Feasible path identification in optimal power flow with sequential convex restriction. IEEE Transactions on Power Systems, 35(5):3648–3659.

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.