ExaPF.jl: A Power Flow Solver for GPUs

Michel Schanen[1], Adrian Maldonado[1], François Pacaud[1], and Mihai Anitescu[1]

[1]Argonne National Laboratory

ABSTRACT

Solving optimal power flow is an important tool in the secure and cost effective operation of the transmission power grids. `ExaPF.jl` aims to implement a reduced space method for solving the optimal power flow problem (OPF) fully on GPUs. Reduced space methods enforce the constraints, represented here by the power flow's (PF) system of nonlinear equations, separately at each iteration of the optimization in the reduced space. This paper describes the API of `ExaPF.jl` for solving the power flow's nonlinear equations entirely on the GPU. This includes the computation of the derivatives using automatic differentiation, an iterative linear solver with a preconditioner, and a Newton-Raphson implementation. All of these steps allow us to run the main computational loop entirely on the GPU with no transfer from host to device.
This implementation will serve as the basis for the future OPF implementation in the reduced space.

Keywords

Julia, GPU, power flow, iterative linear solver, preconditioner, automatic differentiation

1. Statement of Need

The current state-of-the-art for solving optimal power flow is the interior-point method (IPM) in optimization implemented by the solver Ipopt [8] and is the algorithm of reference in implementations like MATPOWER[9]. However, its reliance on unstructured sparse indefinite inertia revealing direct linear solvers makes this algorithm hard to port to GPUs. 'ExaPF.jl' aims at applying a reduced gradient method to tackle this problem, which allows us to leverage iterative linear solvers for solving the linear systems arising in the PF.
Our final goal is a reduced method optimization solver that provides a flexible API for models and formulations outside of the domain of OPF.

2. Components

To make our implementation portable to CPU and GPU architectures we leverage two abstractions: arrays and kernels. Both of these abstractions are supported through the packages `CUDA.jl` [2, 1] and `KernelAbstractions.jl`

2.1 AutoDiff

Given a set of equations `F(x) = 0`, the Newton-Raphson algorithm for solving nonlinear equations (see below) requires the Jacobian `J = jacobian(x)` of `F`. At each iteration a new step `dx` is computed by solving a linear system. In our case `J` is sparse and indefinite, but invertible.

```
go = true
while(go)
  dx .= jacobian(x)\F(x)
  x  .= x .- dx
  go = norm(f(x)) < tol ? true : false
end
```

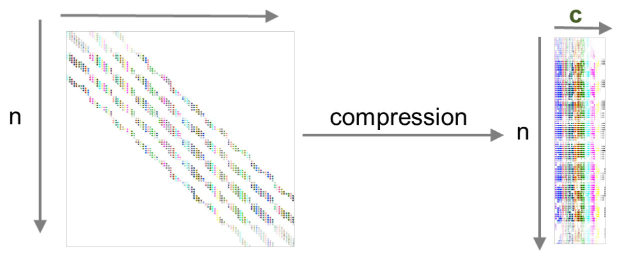

Fig. 1. Jacobian coloring

There are two modes of differentiation called *forward/tangent* or *reverse/adjoint*. The latter is known in machine learning as transposed Jacobian-vector product `adj(x,y) = J(x)'*y`. We recommend [3] for a more in-depth introduction to automatic differentiation. The computational complexity of both models favors the adjoint mode if the number of outputs of `F` is much smaller than the number of inputs `size(x) >> size(F)`, like for example the loss functions in machine learning. However, in our case `F` is a multivariate vector function from $\mathbb{R}^n$ to $\mathbb{R}^n$, where n is the number of buses.
To avoid a complexity of $\mathcal{O}(n) \cdot cost(F)$ by letting the tangent mode run over all Cartesian basis vectors of $\mathbb{R}^n$, we apply the technique of Jacobian coloring to compress the sparse Jacobian `J`. Running the tangent mode, it allows to compute columns of the Jacobian concurrently, by combining independent columns in one Jacobian-vector evaluation (see Figure 1). For sparsity detection we rely on the greedy algorithm implemented by `SparseDiffTools.jl` [4].
Given the sparsity pattern, the forward model is applied through the package `ForwardDiff.jl` [5]. With the number of Jacobian colors c we can build our dual type `t1s` with `N=c` directions:

```
t1s{N} =
ForwardDiff.Dual{Nothing,Float64, N} where N}
```

Note that a second-order type `t2s` can be created naturally by applying the same logic to `t1s`:

```
t2s{M,N} =
ForwardDiff.Dual{Nothing,t1s{N}, M} where M, N}
```

Finally, this dual type can be ported to both vector types `Vector` and `CuVector`:

```
VT = Vector{Float64}
VT = Vector{t1s{N}}}
VT = CuVector{t1s{N}}}
```

Setting `VT` to either of the three types allows us to instantiate code that has been written using the *broadcast operator* .

```
x .= a .* b
```

or accessed in kernels written with ‘KernelAbstractions.jl‘, like for example the power flow equations (here in polar form):

```
@kernel function residual_kernel!(F, v_m, v_a,
                 re_nzval, re_colptr, re_rowval,
                 im_nzval, im_colptr, im_rowval,
                 pinj, qinj, pv, pq, nbus)

    npv = size(pv, 1)
    npq = size(pq, 1)

    i = @index(Global, Linear)
    # REAL PV: 1:npv
    # REAL PQ: (npv+1:npv+npq)
    # IMAG PQ: (npv+npq+1:npv+2npq)
    fr = (i <= npv) ? pv[i] : pq[i - npv]
    F[i] -= pinj[fr]
    if i > npv
        F[i + npq] -= qinj[fr]
    end
    for c in re_colptr[fr]:re_colptr[fr+1]-1
        to = re_rowval[c]
        aij = v_a[fr] - v_a[to]
        coef_cos = v_m[fr]*v_m[to]*re_nzval[c]
        coef_sin = v_m[fr]*v_m[to]*im_nzval[c]
        cos_val = cos(aij)
        sin_val = sin(aij)
        F[i] += coef_cos * cos_val
              + coef_sin * sin_val
        if i > npv
            F[npq + i] += coef_cos * sin_val
                        - coef_sin * cos_val
        end
    end
end
```

These two abstractions are a powerful tool that allow us to implement the forward mode in vectorized form where the number of directions or tangent components of a tangent variable are the number of Jacobian colors. We illustrate this in Figure 2 with a point-wise vector product `x .* y`
This natural way of computing the compressed Jacobian yields a very high performing code that is portable to any vector architecture, given that a similar package like `CUDA.jl` exists. We note that similar packages for the Intel Compute Engine (`oneAPI.jl`) and AMD ROCm (`AMDGPU.jl`) are in development. We expect our package to be portable to AMD and Intel GPUs in the future.

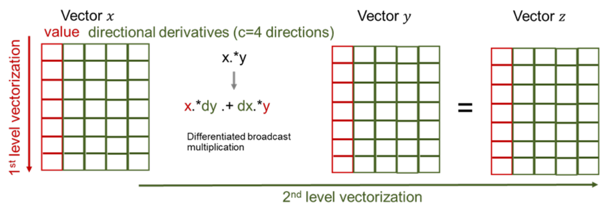

Fig. 2. SIMD AD for point-wise vector product

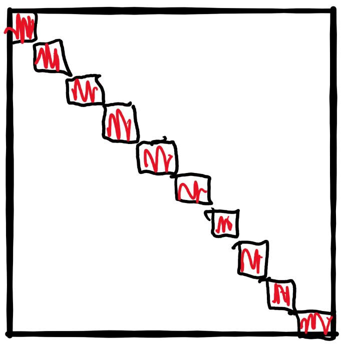

Fig. 3. Dense block Jacobi preconditioner

2.2 Linear Solver

As mentioned before, a linear solver is required to compute the Newton step in

```
dx .= jacobian(x)\F(x)
```

Our package supports the following linear solvers:

—CUSOLVER with ‘csrlsvqr‘ (GPU),

—‘Krylov.jl‘ with ‘dqgmres‘ (CPU/GPU),

—‘IterativeSolvers‘ with ‘bicgstab‘ (CPU) [6],

—UMFPACK through the default Julia ‘òperator (CPU),

—and a custom BiCGSTAB implementation [7] (CPU/GPU).

The last custom implementation was necessary as BiCGSTAB showed much better performance than GMRES and at the time of this writing both `Krylov.jl` and `IterativeSolvers.jl` did not provide an implementation that supported `CUDA.jl`.
Using the iterative solver out of the box leads to divergence and bad performance due to ill-conditioning of the Jacobian. This is a known phenomenon in power systems. That is why this package comes with a block Jacobi preconditioner that is tailored towards GPUs and is proven to work well with power flow problems.
The Jacobian is partitioned into a dense block diagonal structure, where each block is inverted to build our preconditioner `P`. For the partition we use `Metis.jl`.
Compared to incomplete Cholesky and incomplete LU this preconditioner is easily portable to the GPU if the number of blocks is high enough. `ExaPF.jl` uses the batch BLAS calls from `CUBLAS` to invert the single blocks.

```
CUDA.@sync pivot, info =
CUDA.CUBLAS.getrf_batched!(blocks, true)
```

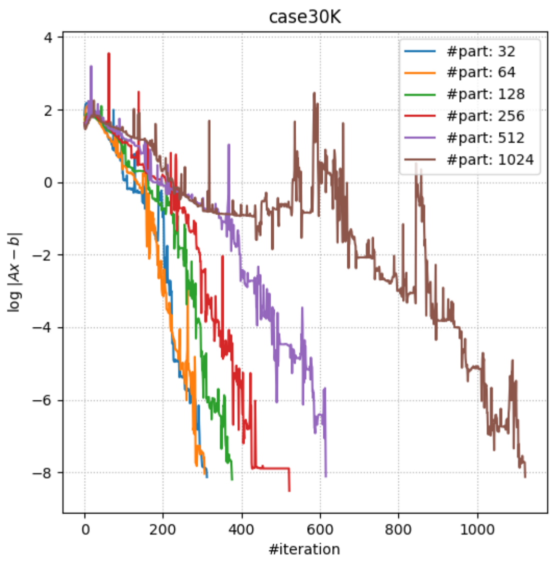

Fig. 4. Use Case

Blocks	Block Size	Time(s)	Time/It.(s)	#Iterations
32	1857	2.85e+00	9.07e-03	314
64	928	1.41e+00	4.57e-03	308
128	464	9.15e-01	2.42e-03	378
256	232	9.09e-01	1.74e-03	524
512	116	5.49e-01	8.90e-04	617
1024	58	7.50e-01	6.67e-04	1125

```
CUDA.@sync pivot, info, p.cuJs =
CUDA.CUBLAS.getri_batched(blocks, pivot)
```

Assuming that other vendors will provide such batched BLAS APIs, this code is portable to other GPU architectures.

3. Performance Example

To illustrate the use case for this solver we consider a 30,000 bus system case from the ARPA-E GO competition. We show how the convergence of the BiCGSTAB algorithm is impacted by the number of blocks in the Jacobi preconditioner. By choosing appropriately the number of blocks, we observe that the iterative solver takes 0.55s to solve the linear system on the GPU. As a comparison, LAPACK takes 0.22s to solve the system on the CPU, and CUSOLVER (with `csrlsvqr`) takes 2.70s.
This shows the number of BiCGSTAB iterations and the time needed to achieve convergence for this power system.

4. Acknowledgments

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.

5. References

[1] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, 132:29–46, 2019.

[2] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[3] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[4] C. Rackauckas. Sparsedifftools: Fast jacobian computation through sparsity exploitation and matrix coloring, 2020.

[5] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in Julia. *arXiv:1607.07892 [cs.MS]*, 2016.

[6] Gerard LG Sleijpen and Diederik R Fokkema. Bicgstab(l) for linear equations involving unsymmetric matrices with complex spectrum. *Electronic Transactions on Numerical Analysis.*, 1:11–32, 1993.

[7] H. A. van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[8] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006.

[9] H. Wang, C. E. Murillo-Sanchez, R. D. Zimmerman, and R. J. Thomas. On computational issues of market-based optimal power flow. *IEEE Transactions on Power Systems*, 22(3):1185–1193, 2007.