Les Cahiers du GERAD ISSN: 0711–2440

Julia as a portable high-level language for numerical solvers of power flow equations on GPU architectures

M. Schanen, D. A. Maldonado, F. Pacaud, A. Montoison, M. Anitescu, K. Kim, Y. Kim, V. Rao, A. Subramanyam

G–2020–74

December 2020

Julia as a portable high-level language for numerical solvers of power flow equations on GPU architectures

Michel Schanen [a]

Daniel Adrian Maldonado [a]

François Pacaud [a]

Alexis Montoison [b]

Mihai Anitescu [a]

Kibaek Kim [a]

Youngdae Kim [a]

Vishwas Rao [a]

Anirudh Subramanyam [a]

[a] *Argonne National Laboratory, 9700 S. Cass Ave, Illinois, USA*

[b] *GERAD & Department of Mathematics and Industrial Engineering, Polytechnique Montréal, Montréal (Québec), Canada, H3C 3A7*

mschanen@anl.gov

December 2020
Les Cahiers du GERAD
G–2020–74

Abstract: We present `ExaPF.jl`, a solver for power flow on GPUs, entirely written in Julia. It implements a highly parallel Newton-Raphson solver for nonlinear equations. We exploit Julia packages for kernel and array abstractions at the modeling level, and generate efficient codes at runtime for both CPU and NVIDIA GPU using Julia's inherent metaprogramming capabilities. In the future, this infrastructure will allow us to leverage this machinery for AMD and Intel GPUs by targeting the ROCm and oneAPI frameworks. The composable design of the Julia language allows us to apply automatic differentiation that alleviates the user from providing derivatives. We also detail a GPU implementation of the iterative solver BiCGSTAB to solve efficiently the linear systems arising in the Newton-Raphson algorithm, and show how to improve its performance with a block-Jacobi preconditioner tailored to the batched matrix inversion capabilities of modern GPUs. The Newton-Raphson algorithm will eventually serve as the foundation of a reduced-space optimization method that will run entirely on the GPU.

Keywords: Power flow, GPU, optimization, Julia, framework, automatic differentiation, iterative methods

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. The research of A. Montoison is supported by an excellence scholarship of the IVADO institute.

1 Introduction

Julia [1] is a new programming language that leverages modern programming design for scientific computing. It is both an interpreted and just-in-time compiled language that allows users to write high performing code while having accessible syntax similar to Matlab. Its strong metaprogramming capabilities allows the developer to generate and transform code at runtime, making it highly flexible. The functional design of the language enables domain scientists to write composable, modular, and maintainable code.

All this is tied to the efficient compiler back-end LLVM [2], known to generate highly efficient machine code for C/C++ on a variety of architectures. With portability solved for general purpose computing platforms, the rise of GPU architectures in high-performance computing shows the limit of this solution. This specialized hardware comes in combination with a custom programming model. In scientific computing—where portability is key—this requires well-designed portability layers through another layer of metaprogramming models (e.g. Kokkos [3], RAJA [4]), thus substantially increasing the complexity of a code base.

With the metaprogramming capabilities of Julia no such programming models are needed, since code transformation is done through language supported *macros*, moving the transformation of the original code from compile time to runtime. This leads to a better separation between the applied transformation logic and the original code. In Listing 1 we show a naive implementation of a dispatch macro that generates code either for the CPU or CUDA depending on what the global variable `target` is set to. The expression `expr` contains the actual function call. This macro either generates the code that dispatches the function using CUDA or just calls the function using the CPU. An expression is stored as an abstract syntax tree and enables the programmer to directly access the Intermediate Representation (IR). This is a strong tool that allows, for example, a language intrinsic implementation of automatic differentiation (AutoDiff) and hardware specific code. We show how these two totally independent code transformations are easily combined in Julia, a work that would otherwise be difficult to achieve in custom programming models.

Listing 1: Macro example in Julia

```
macro dispatch(threads, blocks, expr)
ex = nothing
if target == "cuda"
ex = quote
@cuda $threads $blocks $expr
end
end
if target == "cpu"
ex = quote
$expr
end
end
return esc(ex)
end
```

In this paper we develop an efficient power flow solver that targets general purpose architectures as well as GPUs. It uses algorithms from AutoDiff and iterative linear solvers and combines them in a modular way while separating hardware and algorithm as much as possible. This implementation serves as a mini-app and as the foundation for our future work on an optimal power flow solver for GPUs.

In Section 2 we give an overview of Julia's portable abstractions for GPUs. Section 3 describes how we leveraged Julia's composable design to implement a power flow solver with a modeling layer, differential programming capabilities, and a portable algebra all targeting GPUs. Each component is detailed in Section 4. The results in Section 5 give an overview on current GPU hardware using relevant power grid cases. We then conclude with a brief path forward in Section 6 for using this machinery in solving optimal power flow on GPU architectures.

2 GPU programming in Julia

The GPU API design of Julia follows a classical pattern of language design and their compilers (see Figure 1). On the one end are abstractions connecting to the algorithms and on the other end the generation of hardware specific code. The Julia programmer can either choose to write *abstracted kernels* or follow an *array abstraction* that connects to the abstract array type of Julia. The latter option has the advantage of requiring no code changes at all and opening all the tools written for array types; vectors, matrices, linear algebra. It relies solely on the broadcasting operator '.' for generating kernels at the assignment level (see Listing 2).

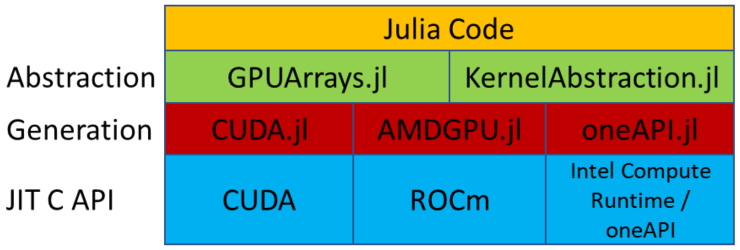


Figure 1: Julia GPU API - The abstraction is either through the abstract type of Julia language arrays or through a kernel abstraction. The generation and compilation of GPU code happens through metaprogramming in Julia that generates calls to the C APIs which initiates a just-in-time compilation of the object code at runtime

Listing 2: Array abstraction in Julia

```julia
V=Vector # CPU vector
V=CuVector  # GPU CUDA vector
V=ROCVector # GPU ROCm vector
V=oneArray  #  GPU oneAPI vector
V=CuVector{Float64} # GPU vector
D=V{Dual{Float64}} # GPU dual vector
da, db, dc=D(ones(Float64,n) # GPU transfer
function incrmul(a::AbstractArray,
b::AbstractArray)
c::AbstractArray)
c .+= a .* b
end
incrmul(a,b,c) # JIT instantiation
```

This allows linear algebra kernels to be written generically for both CPU and GPU. For example `Krylov.jl` [5] implements various iterative solvers that use exactly the same code for CPUs and for GPUs. In addition it allows the application of AutoDiff right through the iterative solver on a GPU by a simple type change in the code similar to Listing 2.

When algorithms leave the area of linear algebra and vector calculus, more complex kernels might be necessary, which leads us to the other method of abstraction. This is achieved through a hardware generalization that defines a common *kernel abstracted* API across various GPU types. This has more similarity to traditional portability layers like RAJA or Kokkos. However, it allows to be encapsulated in codes with the array abstraction described above, making switching from one abstraction to the other seamless. We will go into more detail when we cover the modeling of the power flow in Section 4.1.

The generation layer (see Figure 1) connects to the just-in-time interface of the respective vendor provided GPU API (CUDA for NVIDIA, ROCm for AMD, and oneAPI for Intel). Without going into the details, CUDA is currently by far the most developed API for scientific computing. This is also true on the Julia side. A lot of companies and customers of Julia Computing, Inc. rely on NVIDIA GPU support in Julia. This makes the package `CUDA.jl` a Tier 1 supported package by Julia Computing, Inc. and NVIDIA a sponsor of the language Julia. ROCm and oneAPI support in Julia is currently in early development with a targeted goal of using the same programming models implemented by `GPUArrays.jl` and `KernelAbstraction.jl`.

This document focuses on GPUs and thus on programming for single nodes. However, it should be noted that Julia has support for distributed parallelism through MPI and its own distributed abstraction. With MPI being the de facto standard for distributed scientific computing, `MPI.jl` provides a solid MPI API for Julia, alleviating the user through its multiple dispatch from dealing with tedious low level data structure types. Its support for serialization even allows the transparent communication of functions if the involved processes use the same Julia system image. `MPI.jl` provides support for CUDA enabled MPI libraries allowing direct communication between GPUs connected via NVLink. Nonetheless, it would also be worthwhile to investigate Julia's own distributed API that allows for much leaner code avoiding the Fortran and C focused MPI interface. In particular, we aim at applying distributed computing for solving multiperiod security constrained optimal power flow.

3 Fast and portable power flow solver: `ExaPF.jl`

`ExaPF.jl`[1] leverages the GPU capabilities of Julia by implementing a power flow solver that runs fully on the GPU without any host-device transfer. We model mathematically the electrical network as an undirected graph $\mathcal{G} = (\mathcal{B}, \mathcal{E})$, where we denote by $\mathcal{B}$ the set of buses and $\mathcal{E}$ the lines connecting the buses together. The topology of the network is usually given by a node admittance matrix $\boldsymbol{Y} \in \mathbb{C}^{n_b \times n_b}$, with $n_b = |\mathcal{B}|$ the number of buses in the network. The real and imaginary components of the matrix $\boldsymbol{Y}$ are usually split apart $\boldsymbol{Y} = \boldsymbol{Y}^{re} + j\boldsymbol{Y}^{im}$, with $j = \sqrt{-1}$. For a bus $b \in \mathcal{B}$, we denote by V_b its voltage magnitude, θ_b its voltage angle, p_b its active power and q_b its reactive power which are physical quantities that describe the steady state of the power network. Let $\boldsymbol{V} = (V_b)_{b\in\mathcal{B}}$ and $\boldsymbol{\theta} = (\theta_b)_{b\in\mathcal{B}}$

For each bus b, the network must satisfy energy conservation laws that are defined by the Kirchoff law: these are the power balance equations. Kirchoff law states that the sum of injected power (p_k, q_k) into bus $k \in \mathcal{B}$ must be equal to the sum of extracted power, establishing a set of *power balance* equations:

$$\begin{aligned} p_k &= V_k \sum_{l\in\mathcal{B}} V_l (Y_{kl}^{re} \cos\theta_{kl} + Y_{kl}^{im} \sin\theta_{kl}) \ , \\ q_k &= V_k \sum_{l\in\mathcal{B}} V_l (Y_{kl}^{re} \sin\theta_{kl} - Y_{kl}^{im} \cos\theta_{kl}) \ , \end{aligned} \tag{1}$$

where Y_{kl}^{re} and Y_{kl}^{im} are the elements of matrices $\boldsymbol{Y}^{re}$ and $\boldsymbol{Y}^{im}$, respectively, and $\theta_{kl} := \theta_k - \theta_l$. We write the power Equations (1) in an abstract formalism by introducing a state variable $\boldsymbol{x} = (\boldsymbol{V}, \boldsymbol{\theta}) \in \mathbb{R}^{2n_b}$, a vector $\boldsymbol{p} \in \mathbb{R}^{n_p}$ of n_p parameters, and a residual functional $f : \mathbb{R}^{2n_b} \to \mathbb{R}^{n_b}$ encoding the Equations (1) in the compact form: $f(\boldsymbol{x}, \boldsymbol{p}) = 0$.

In power system analysis, the Newton-Raphson algorithm [6] is a standard algorithm to solve the set of non-linear equations $f(\boldsymbol{x}, \boldsymbol{p}) = 0$. Starting from an initial guess $\boldsymbol{x}_0$, the algorithm proceeds to the following iterations, till a convergence criteria is reached:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \nabla f(\boldsymbol{x}_i, \boldsymbol{p})^{-1} f(\boldsymbol{x}_i, \boldsymbol{p}) \ , \quad i \geq 0$$

Step by step, the algorithm writes out for each iteration i:

1. Evaluate $\boldsymbol{f}_i = f(\boldsymbol{x}_i, \boldsymbol{p})$
2. Compute $\boldsymbol{J}_i = \nabla f(\boldsymbol{x}_i, \boldsymbol{p})$
3. Solve $\boldsymbol{J}_i \Delta\boldsymbol{x}_i = -\boldsymbol{f}_i$
4. Update $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \Delta\boldsymbol{x}_i$

In our endeavor to perform this computations fully leveraging the GPU, we first come to the problem of solving the linear system of Step 3 that involves the Jacobian of the residual. The latter can be large

[1] https://github.com/exanauts/ExaPF.jl

($\sim 100,000$ entries) and cannot fit in a single GPU block in dense format. It is generally very sparse but unstructured, due to the topology of the power network. Because of the difficulties of solving large and sparse systems on GPUs with direct solvers, we employ an iterative solver. Iterative solvers for square systems such as GMRES [7] can offer good convergence but often require direct preconditioners such as incomplete LU (ILU), which can be difficult to implement on a GPU architecture [8] . In its standard formulation, GMRES requires significant amounts of storage, so that the restarted variant GMRES(m) or the limited-memory version DQGMRES are typically used instead for large systems. Another solver designed for square systems that has low memory requirements is BiCGSTAB [9], and we have found that BiCGSTAB [9] with a block-Jacobi preconditioner offers good performance on GPU architectures.

With this in mind, all aforementioned steps are entirely implemented on the GPU. The algorithm proceeds as follows: step 1 is implemented using an abstract GPU kernel via `KernelAbstraction.jl` (see Listing 3). In step 2 the Jacobian $\boldsymbol{J}_i$ is generated once per run using AutoDiff and evaluated directly on the GPU at each iteration. The AutoDiff package `ForwardDiff.jl` [10] is transparently applied to that kernel to generate $\boldsymbol{J}_i$ in step 2.

Listing 3: Kernel implementation using `KernelAbstraction.jl`

```julia
@kernel function residual_kernel!(F, ...)
i = @index(Global, Linear)
fr = (i <= npv) ? pv[i] : pq[i-npv]
F[i] -= pinj[fr]
if i>npv
F[i + npq] -= qinj[fr]
end
for c in colptr[fr]:colptr[fr+1]-1
to  = ybus_re_rowval[c]
aij = v_a[fr]-v_a[to]
coef_cos = v_m[fr]*v_m[to]*ybus_re_nzval[c]
coef_sin = v_m[fr]*v_m[to]*ybus_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
```

In step 3, we note that the sparsity pattern of the Jacobian matrix $\boldsymbol{J}_i$ does not change from one iteration to the next. Thus, we instantiate the block-Jacobi preconditioner at the first iteration by partitioning the initial Jacobian matrix $\boldsymbol{J}_0$. Then, we update the preconditioner $\boldsymbol{P}_i$ at each iteration with a three steps procedure:

1. Extract the Jacobi blocks from the sparse CSR matrix $\boldsymbol{J}_i$ and store them in dense format blocks.
2. Apply *batch inversion* (e.g. CUBLAS) on the dense blocks to get an approximation to $\boldsymbol{J}_i^{-1}$.
3. Move the inverted blocks to the sparse CSR matrix $\boldsymbol{P}_i$.

The batch inversion has to be supported by the BLAS library provided by the GPU vendor. This is not a standard BLAS call and we admit that its support is not guaranteed. However, its implementation can also be mapped to single dispatching BLAS kernels for matrix inversion.

We solve the linear system $\boldsymbol{J}_i \Delta \boldsymbol{x}_i = -\boldsymbol{f}_i$ using the implementation of BiCGSTAB available as part of the `Krylov.jl` collection of Krylov methods [5]. BiCGSTAB only requires matrix-vector products ($\boldsymbol{u} \leftarrow A\boldsymbol{v}$)

and vector operations ($\|\boldsymbol{v}\|$, $\boldsymbol{u}^{\boldsymbol{T}}\boldsymbol{v}$, $\boldsymbol{v} \leftarrow \alpha\boldsymbol{u} + \beta\boldsymbol{v}$), which are easily parallelizable and make the algorithm suitable for GPU. The BiCGSTAB implementation is generic so as to take advantage of the multiple dispatch and broadcast features of Julia. It allows the implementation to be specialized automatically by the compiler for both CPU and GPU usages.

The power flow solver `ExaPF.jl` depends on three key algorithmic domains for the modeling of f, the computation of the Jacobian J and the implementation of linear solvers. We will go over these three areas in the next section.

4 Portable and composable algorithm design

In our portable design we distinguish between three essential ingredients: *modeling*, *differentiation*, and *linear algebra*. Our goal is to have these three components abstracted in a portable way and apply this design pattern for nonlinear equations in the future (such as nonlinear programming in optimization). We will emphasize the benefits of using a composable and differentiable language like Julia that allows these three components to be completely independent and not domain or application specific.

4.1 Modeling

```
cost = sum(c2 .+ c3 .* pg .+ c4 .* pg.^2)
```

Figure 2: Array abstraction using `GPUArrays.jl`

```
@oneapi residual_kernel(F,...)
@cuda residual_kernel(F,...)
@roc residual_kernel(F,...)
```

Figure 3: Kernel abstraction using `KernelAbstractions.jl`

The modeling in `ExaPF.jl` leverages the `GPUArrays.jl` and `KernelAbstraction.jl` abstractions presented in Section 1. In addition to this hardware/software abstraction, we have separated the mathematical and physical abstractions. The power flow kernel, which evaluates the residual function of the Newton Rhapson problem, is implemented using the physical quantities.

Using the `@kernel` macro provided by `KernelAbstraction.jl` the function is dispatched on the GPU architecture chosen by the user (`@cuda`, `@oneapi`, `@roc`). Note that `F` here is the output of the residual function f mentioned before. This is all the modeling code a user has to provide. In the future we aim at making this more amenable to the user by generating the code according to the domain specific language specified in the `JuMP.jl` package [11]. However, we identified three constructs that `JuMP.jl` is currently missing and we are working towards addressing these issues. We will go through these proposed changes by showing a proposed model written in the extended language (see Listing 4)

Listing 4: Proposed DSL extension

```
m = Model()
@vector(m, P, 1:nbus); @vector(m, Vm, 1:nbus)
@vector(m, Q, 1:nbus); @vector(m, Va, 1:nbus)
@graph(m, G, Yre)
@spequation(m, P .= vm
.* sum(vm[l]
.* (Yre[k,l]
.* cos(va[k] .- va[l])
.+ Yim[k,l]
.* sin(va[k] .- va[l])
),
for k,l=neighbor(k) in G
)
)
@spequation(m, Q .= vm
.* sum(vm[l]
.* (Yre[k,l]
.* sin(va[k] .- va[l])
.- Yim[k,l]
.* cos(va[k] .- va[l])
),
for k,l=neighbor(k) in G
)
)
@synchronize(m)
```

1. Currently, in `JuMP.jl`, variables are the first-class object. For GPUs, models have to be written with vectors (`@vector`) as a first class object to apply the broadcast operator as shown in Section 1 to seamlessly generate SIMD kernels for the GPU.
2. To allow the expression of sparsity, an adjacency graph `G` is defined through the macro `@graph` passed to a sparse equation macro `@spequation`. This eventually allows loops over sparse matrix entries in CSR or CSC format in a kernel.
3. And last, since kernels are evaluated asynchronously, we want to convey that notion to the model and explicitly give the user the ability to define synchronization points via the `@synchronize` macro.

These amendments to the language are currently discussed in our community and have not yet been implemented in our package. This feature would allow the model to be easily defined outside of the package `ExaPF.jl`.

4.2 Differentiation

Writing the differentiated model by hand is known to be a tedious process prone to errors and very hard to debug. Since this software is meant as a research tool we do not want to burden the user with differentiating the model and implementing $J(\boldsymbol{x})$ and only requiring the primal/non differentiated function f. To this end, we designed the package to make full use of AutoDiff [12] throughout all architectures. AutoDiff is a technology that transforms code of a function implementation $\boldsymbol{y} = f(\boldsymbol{x})$, with $\boldsymbol{x} \in \mathbb{R}^n, y \in \mathbb{R}^n$ algorithmically to compute the *tangent model* calculating $\boldsymbol{y} = f(\boldsymbol{x})$ and $\dot{\boldsymbol{y}} = J(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}$, where $\dot{\boldsymbol{y}}$ and $\dot{\boldsymbol{x}}$ are called the tangents, directional derivatives, or *duals*. Note, that the Jacobian J is not generated directly. In order to compute the full Jacobian, one has to call the tangent-model n times over the n Cartesian basis vectors of $\mathbb{R}^n$. This yields a computational cost of $\mathcal{O}(n \cdot cost(f))$, with $cost(f)$ being the cost of the original function evaluation.

The composability of the Julia language has shown early on that code transformations like AutoDiff can be seamlessly integrated in a modular workflow. In stark contrast to tools like Tensorflow [13], AutoDiff in Julia does not rely on a domain specific language (DSL)[14], since all statements in the modeling frameworks and hardware abstractions eventually end up as Julia code or Julia IR. The only restriction the hardware abstractions for the GPU have, is that the data structures have to be of binary types. This is true for all primitive types and types composed of primitive types. However, it is not true for example if functions or references are part of a type. Fortunately, the AutoDiff package `ForwardDiff.jl` generates a dual or derivative type that has the `isbits` property. This allows us to apply AutoDiff seamlessly for GPUs just as for CPUs. Each dual type is composed of its value and the directional derivative values.

In Figure 4 we show an example of a point-wise multiplication of two vectors. The red color is the original vector that is then generated to a GPU kernel using the broadcast multiplication. The dual type adds c directions to each vector entry. Again, the broadcast operator is seamlessly applied to the now twice vectorized object. This is also reflected in our runtime results (see Figure 5), where we see a superlinear speedup by adding directions c with an optimum reached around 32 and 64 directions for an entire power flow evaluation.

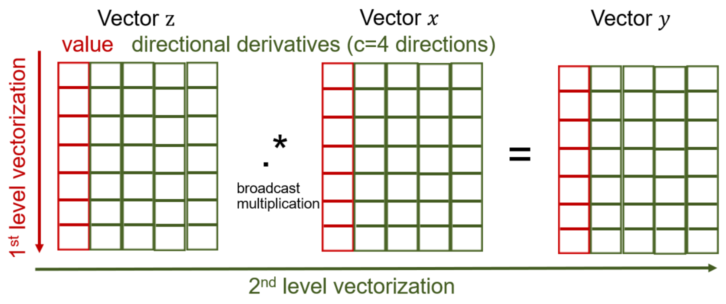


Figure 4: Point-wise multiplication transformed to a dual type

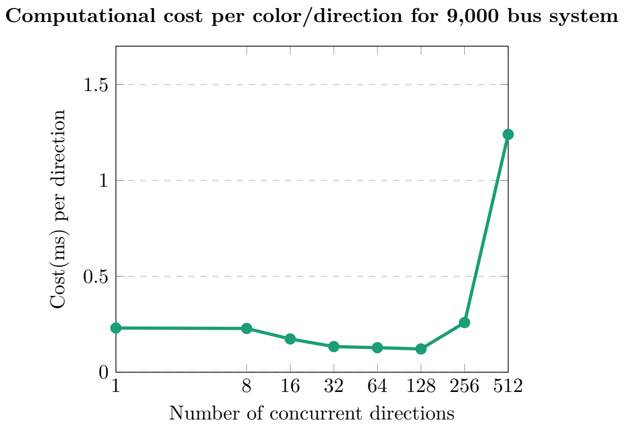
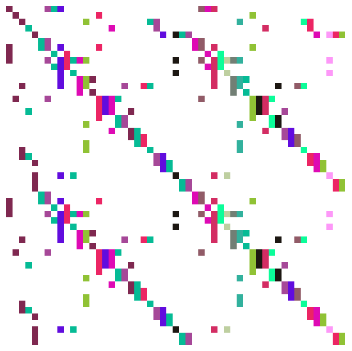


Figure 5: Optimal number of concurrent directions in a power flow model evaluation with 9,241 buses on the PEGASE case (see Section 5). The number of actual Jacobian colors is 28. To compute the full Jacobian, a directional derivative component for each color is required.

However, our Jacobian has a dimension that is on the scale of $\mathcal{O}$(number of buses) which would amount to a large number of directions c and exhausts the memory of the GPU. That is why we apply a technique called Jacobian coloring (see Figure 6). As the sparsity structure stays the same across the computation we do this once in the setup stage on the CPU because this algorithm is not amenable to the GPU. This allows us to compute the full Jacobian in one evaluation by computing several independent directions in parallel while exploiting the aforementioned SIMD operations. The output is a compressed Jacobian that is directly used to build the preconditioner described in the following section. Another kernel is used to extract the sparse Jacobian into CSR or CSC format for the iterative linear solver.

4.3 Linear algebra

We have extensively explored the best iterative solver setup for our GPU implementation, consisting of choosing a preconditioner and an iterative linear solver algorithm. For the preconditioner the natural choice for a GPU is to have independent tasks that can be executed by small kernels. We therefore picked a block-Jacobi preconditioner usually used in distributed parallelism for PDEs. We partition the Jacobian using METIS. The procedure splits the Jacobian into diagonal blocks with a minimal cut (see Figure 7). These blocks are built in single dense format directly from the compressed Jacobian. We then apply a batched BLAS call to invert these single dense blocks (see Figure 8). This nonstandard batched BLAS call is present in


(a) Jacobian stored in CSR or CSC: A valid coloring has no color appearing twice in a row

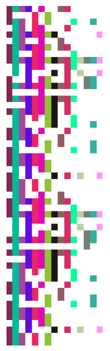

(b) Compressed Jacobian in dense storage: White color entries are unused space

Figure 6: Jacobian coloring on the IEEE-30 bus case compressing the matrix from 53×53 to 15×53

CUBLAS. However, there is no guarantee this feature will be available in ROCm and oneAPI. A workaround would be to asynchronously dispatch single dense BLAS calls for each block. After this batch inversion is done, the inverted blocks are put into a matrix P stored in sparse CSR or CSC format via another GPU kernel. Note that this technique of having a lot of single blocks to invert (up to $1,000$) is not amenable to the CPU, thus a one-on-one comparison between CPU and GPU is not reasonable here. We acknowledge that for the CPU the user should continue using standard direct sparse solvers.

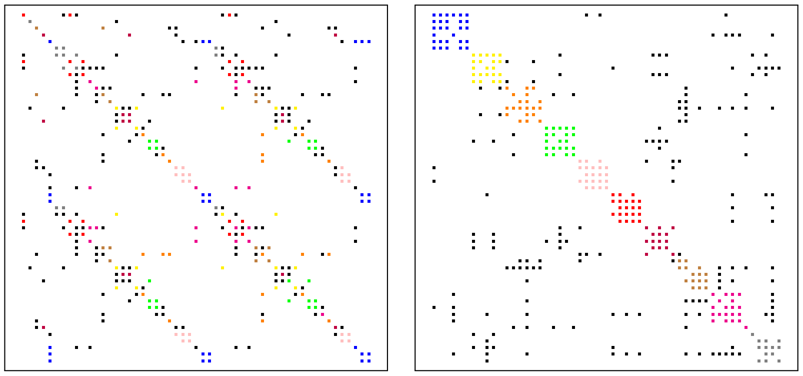


Figure 7: Jacobian partitioning on the IEEE-30 bus case. The figures show the Jacobian with 10 partitions (left) and the Jacobian rearranged by cluster (right). Coefficients with identical color mean that the associated rows and columns share the same partition.

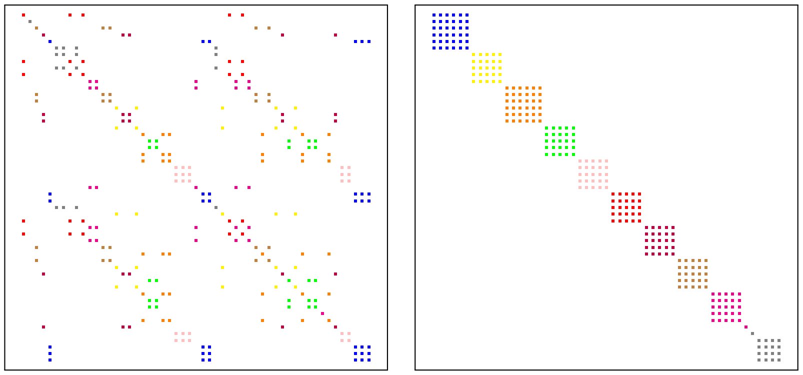


Figure 8: The figures show the block-Jacobi preconditioner computed using batch inversion on diagonal blocks of the restructured Jacobian (right) and the rearranged one that will be used for $P_i * v$ products (left).

Our Jacobian is square and unsymmetric. Thus, we compared two algorithms: GMRES and BiCGSTAB. BiCGSTAB performs much better than GMRES on our problem with the block-Jacobi preconditioner (see Figure 9). Although BiCGSTAB and GMRES are matrix-free methods, we don't use a linear operator that models Jacobian-vector products because neither linear solver converges without the preconditioner and the

Jacobian $\boldsymbol{J}_i$ must be explicitly built to compute $\boldsymbol{P}_i$. However, a linear operator that models $\boldsymbol{P}_i * \boldsymbol{v}$ can be used to take advantage of dense matrix-vector products with each inverted block instead of storing all of them in a sparse matrix and using a sparse matrix-vector routine. An ILU preconditioner can also benefit from this asset with an efficient linear operator that performs the forward and backsolves. We leave the investigation and integration of such extensions to future work.

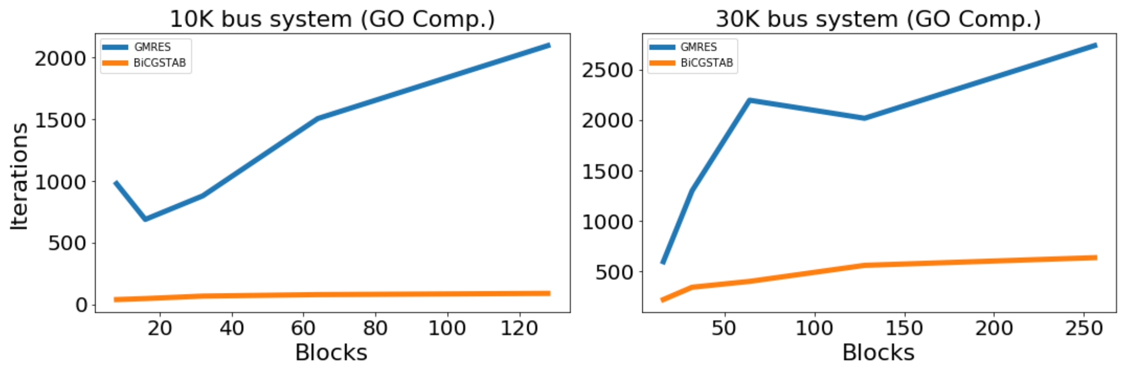


Figure 9: Comparison of GMRES and BiCGSTAB on two cases from the Grid Optimization competition using reference implementations on CPU

Three linear solvers are available inside our package, two direct methods developed specifically for CPU or GPU architectures and the iterative method BiCGSTAB, that relies on Julia features to be suitable on both architectures. We present the results of each linear solver in the next section.

5 Results

All experiments were conducted on our workstation at Argonne and a node of the Summit supercomputer at the Oak Ridge Leadership Computing Facility. Both systems are equipped with NVIDIA Quadro GV100 GPUs with both cards only differing in the amount of RAM (16GB vs 32GB). Our workstation uses the 32GB variant combined with a dual socket Intel Xeon Gold 6140 CPU @ 2.30GHz and 512 GB of RAM. Summit uses the PowerPC architecture which is supported by Julia since version 1.3. On Summit we build Julia directly from source and use the CUDA libraries installed on the system. Since `ExaPF.jl` runs nearly entirely on the GPU we observed that the runtime differences between Summit and our workstation are negligible for our results, which is attributed to the benchmarked parts running entirely on the GPU. To get an overview of the performance we use a variety of network data (see Table 1): the IEEE 300 bus case, the European Pan European Grid Advanced Simulation and state Estimation (PEGASE) 9,241 bus case, and three grids of various sizes (10,000, and 30,000 buses) from the Trial 3 dataset of the ARPA-E Grid Optimization (GO) competition. Note that the largest transmission network in the U.S. is the Eastern Interconnect with a size of around 70,000 buses.

Table 1: Overview of tested cases: Case name, number of buses, generators

Case	Buses	Generators
IEEE300	300	70
PEGASE	9,241	1445
GO1	10,000	2,089
GO2	30,000	3,526

The size of the Jacobian is proportional to the number of buses. From Section 3 we have the following three major computational steps: compute the Jacobian J through AutoDiff, compute the preconditioner P, and solve the linear system $J\Delta x = -f$. In addition, the structure of the network influences the number of Jacobian colors, which itself influences the performance of AutoDiff.

We compare the performance between using a direct linear solver on the CPU, a direct linear solver on the GPU, and the iterative linear solver BiCGSTAB with a block-Jacobi preconditioner. The direct linear solver is implemented through the backslash operator \ in Julia which eventually calls UMFPACK. This solution is roughly the same algorithm as in MATPOWER and achieves similar performance. For the direct solver on the GPU we chose the vendor provided `csrlsvqr` of the CUSOLVE library, which use a sparse QR factorization. To speed up the convergence of BiCGSTAB and to maximize performance (see Section 4.2), we keep the block-Jacobi block sizes at around 64×64. This leads to a large number of blocks (up to 902 for the GO2 case). It is impossible to invert that many blocks efficiently on the CPU. The CPU should clearly shine with the traditional sparse solver. The choice of the solvers is preceded by an extensive exploration of the linear solvers available on GPUs for our class of problems. This work is included in an article submitted to this same issue of Parallel Computing [15].

We distinguish between the *setup stage* and the *solve stage*. The setup stage runs on the CPU and includes reading the power system data, coloring the Jacobian, the block-Jacobi partitioning, compilation, and general object factory assembly. These steps have to be only executed once per case and could in theory be largely reused across runs. We plan on deploying such features in the future. The solving stage is the loop over the Newton-Raphson iterations elaborated in Section 3. This loop constitutes our benchmarked section and requires no allocations and no host-device transfer.

The total runtime results in Figure 10 show that the sparse solver UMFPACK on the CPU provides the fastest time to solution on all four cases. The CUSOLVE solver shows better results than BiCGSTAB on IEEE300 and PEGASE. However, on the cases GO1 and GO2 we see CUSOLVE faltering and BiCGSTAB being the second fastest solution. To understand this better we look at the performance details of BiCGSTAB in Table 2 and realize that Newton-Raphson iterations are all around 4-6 iterations, with an absolute tolerance for the Krylov solver set to $\epsilon = 10^{-6}$. However, BiCGSTAB struggles with both the IEEE300 and PEGASE cases with an associated number of Krylov iterations being around a quarter of the matrix dimension. The cases GO1 and GO2 show a much better Krylov solver convergence. This is due to the condition of the matrix, which eventually is based on the structure and physics of the network. IEEE300 and PEGASE are artificially created networks, which are known to be harder problem instances. GO1 and GO2 could be based on more realistic instances. However we have no confirmation that such is the case. Tuning the number of blocks and the tolerances could lead to better performance, but we wanted to have the same parameters for all four cases and a realistic use case scenario of our software, by a heuristic default choice of tolerances and number of Jacobi blocks in the preconditioner.

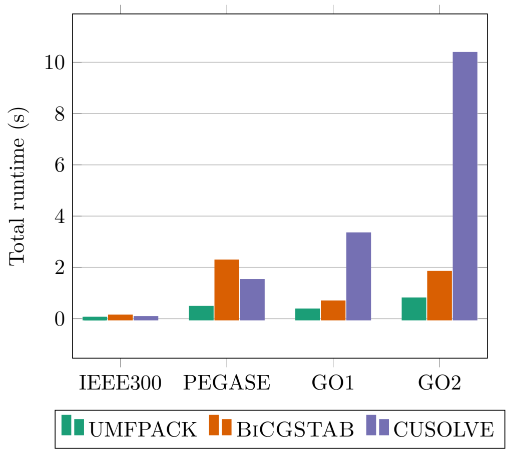


Figure 10: Total runtime with various linear solvers

Last, we want to take a look at the relative performance (see Figure 11) of AutoDiff, the block-Jacobi preconditioner, and BiCGSTAB. AutoDiff is known to lead to performance overheads due to the additional complexity of the code transformation. Added to this, knowing that AutoDiff on the GPU is a bottleneck in machine learning applications, we were surprised that our careful implementation resulted in a stellar performance for the Jacobian computation on the GPU. AutoDiff takes below 10% of the runtime in all four cases. The preconditioner — which relies on the batch inversion of the blocks — comes in with below 20% of the total runtime. This leaves us with the BiCGSTAB taking around 80% of the runtime.

Table 2: Performance details - Jacobian colors, and dimension of the $n \times n$ Jacobian matrix, Newton-Raphson iterations, and Krylov iterations

Case	Colors	Dim.	Blocks	N-R	Krylov
IEEE300	8	530	9	5	192
PEGASE	28	17,036	267	6	4666
GO1	14	19,068	298	4	1292
GO2	20	57,721	902	4	2153

We expect further performance gains by improving the preconditioner (e.g. additive-Schwarz). In summary, we are confident that our solution of using iterative solvers on the GPU for complex system problems is the right way forward. Different architectures require different solutions, and sparse direct solvers do not seem a good fit for GPUs. Moreover, we are confirming that Julia technologies, like AutoDiff, are efficiently portable to scientific applications on GPUs without relying on complex frameworks like Tensorflow or PyTorch.

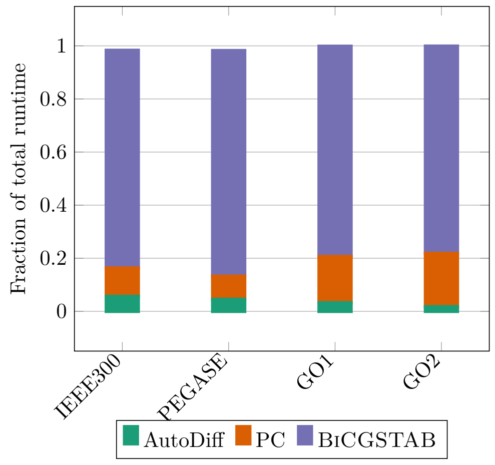


Figure 11: Fraction of runtime for the three most costly computations: BiCGSTAB, preconditioner (PC), and Jacobian computation through AutoDiff. The bulk of the time is spent in BiCGSTAB performing matrix-vector products

6 Extension to optimal power flow: Reduced methods

Now that we have described everything to port the resolution of the power flow equations on the GPU, the attentive reader is in right to wonder on how to apply this to optimal power flow. For the sake of clarification, we propose hereafter an application to the resolution of optimal power flow, directly in the reduced space induced by the power flow equations

$$f(\boldsymbol{x}, \boldsymbol{p}) = 0 \; .$$

We will not cover the numerical optimization details and focus on how our GPU implementation of the power flow allows us to solve optimal power flow entirely on the GPU.

6.1 Optimal power flow problem

We assume now a control subset of the parameters vector $\boldsymbol{p}$, which we partition into a vector of control $\boldsymbol{u}$ and a vector of fixed parameter $\boldsymbol{p}_f$ such that $\boldsymbol{p} = (\boldsymbol{u}, \boldsymbol{p}_f)$. In optimal power flow, the control variables $\boldsymbol{u}$ are usually the active generated power at the generators in the power grid. Then, the job of the optimizer is to find the optimal control $\boldsymbol{u}^\sharp$ such that a cost functional $c(\boldsymbol{x}, \boldsymbol{u})$ is minimized while satisfying a set of inequality constraints $h(\boldsymbol{x}, \boldsymbol{u}) \leq 0$. The optimization problem is given by

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{u}} \quad & c(\boldsymbol{x}, \boldsymbol{u}) \\ \text{s.t.} \quad & f(\boldsymbol{x}, \boldsymbol{u}) = 0 \\ & h(\boldsymbol{x}, \boldsymbol{u}) \leq 0 \; , \end{aligned} \tag{2}$$

(we have omitted the vector of fixed parameter $\boldsymbol{p}_f$ for the sake of conciseness).

6.2 Reduced-space algorithm

In (2), we are optimizing with relation to the state $\boldsymbol{x}$ and the control $\boldsymbol{u}$. However, we know that the state depends on the control $\boldsymbol{u}$ via the implicit equation $f(\boldsymbol{x}, \boldsymbol{u}) = 0$. Rephrasing it, for each control $\boldsymbol{u}$, we ought to find a state $\boldsymbol{x} = x(\boldsymbol{u})$ such that $f\big(x(\boldsymbol{u}), \boldsymbol{u}\big) = 0$. The solution $x(\boldsymbol{u})$ is defined implicitly as a solution of the Newton-Raphson algorithm described in Section 3. Therefore, the reduced-space problem writes out as simply as

$$\begin{aligned} \min_{\boldsymbol{u}} \;\; & c(x(\boldsymbol{u}), \boldsymbol{u}) \\ \text{s.t.} \;\; & h(x(\boldsymbol{u}), \boldsymbol{u}) \leq 0 \,. \end{aligned} \tag{3}$$

The formulation (3) comes with two advantages.

1. The dimension of the problem is reduced from $n_x + n_u$ to n_u, allowing to decrease the memory usage of the resolution algorithm.
2. The equality equations $f(\boldsymbol{x}, \boldsymbol{u}) = 0$ are satisfied *implicitly*, and do not appear in the reduced-space problem. Thus, the resulting non-linear problem encompasses only *inequality* constraints.

The beauty of reduced-space method is that the implicit function theorem (under some regularity conditions) allows us to derive the gradient of the functions f and h *directly* in the reduced-space. For instance, if we consider the objective c, we introduce the Lagrangian functional associated to the equality constraints $f(\boldsymbol{x}, \boldsymbol{u}) = 0$ as

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{u}, \lambda) = c(\boldsymbol{x}, \boldsymbol{u}) + \lambda^\top f(\boldsymbol{x}, \boldsymbol{u}) \,. \tag{4}$$

If $\boldsymbol{x} = x(\boldsymbol{u})$ and satisfies the power flow equation, the Lagrangian is equal to $\mathcal{L}(\boldsymbol{x}, \boldsymbol{u}, \lambda) = c(\boldsymbol{x}, \boldsymbol{u})$ and its value is independent from the adjoint λ. By the chain rule, the gradient of $\mathcal{L}\big(x(\boldsymbol{u}), \boldsymbol{u}, \lambda\big)$ with relation to $\boldsymbol{u}$ is then given by

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{u}} &= \frac{\partial c}{\partial \boldsymbol{u}} + \frac{\partial c}{\partial \boldsymbol{x}} \frac{d\boldsymbol{x}}{d\boldsymbol{u}} + \lambda^\top \Big(\frac{\partial f}{\partial \boldsymbol{u}} + \frac{\partial f}{\partial \boldsymbol{x}} \frac{d\boldsymbol{x}}{d\boldsymbol{u}} \Big) \\ &= \frac{\partial c}{\partial \boldsymbol{u}} + \lambda^\top \frac{\partial f}{\partial \boldsymbol{u}} + \Big(\frac{\partial c}{\partial \boldsymbol{x}} + \lambda^\top \frac{\partial f}{\partial \boldsymbol{x}} \Big) \frac{d\boldsymbol{x}}{d\boldsymbol{u}} \,. \end{aligned} \tag{5}$$

By choosing the Lagrangian multipliers or adjoints as a solution of the linear system

$$\frac{\partial f}{\partial \boldsymbol{x}}^\top \lambda = -\frac{\partial c}{\partial \boldsymbol{x}} \,, \tag{6}$$

we compute the *reduced gradient* as

$$\nabla_u c(\boldsymbol{u}) = \frac{\partial c}{\partial \boldsymbol{u}} + \frac{\partial f}{\partial \boldsymbol{u}}^\top \lambda \,. \tag{7}$$

Fortunately, the gradient $\dfrac{\partial f}{\partial \boldsymbol{x}}$ has already been computed in the solution of the powerflow as the Jacobian $\boldsymbol{J}_i = \nabla f(\boldsymbol{x})$, which we acquired using AutoDiff. Moreover, we can use the same linear solver BiCGSTAB on the GPU to solve the linear system (6). AutoDiff may also be applied to compute ∇c, however, due to performance reasons and it only being a quadratic function, we implemented the gradient by hand. In the future, we envision to apply adjoint AutoDiff tools like `Zygote.jl` [16] to generate that implementation.

The selection of the step update $\boldsymbol{u}_{k+1}$ is subject to current research. However, all these steps are trivial to implement on the GPU using the same `GPUArrays.jl` abstraction. The optimal power flow implementation is only a thin additional layer on top of our implementation for the power flow.

7 Conclusion

We have described a portable design of a nonlinear equations solver in Julia, extensible to nonlinear programming. It serves both as a mini-app for optimal power flow and constitutes an engineering application in itself. It includes the basic issues that also arise in optimization: Newton-Raphson, linear solver, AutoDiff, and modeling.

ExaPF.jl runs on systems from a laptop to supercomputers like Summit without any code change. Julia completely relieves the user from the compilation step and tedious building of portability frameworks conventionally used in scientific computing. With incoming support for Intel and AMD GPUs in KernelAbstraction.jl and GPUArrays.jl we will be able to target the upcoming supercomputers Aurora and Frontier. This seamless support for GPU accelerators is extensible to other accelerators like FPGAs, TPMs, as long as they provide accelerated vector operations, crucial for scientific codes.

Our implementation adheres to the principle of *universal differential programming*, allowing it to be readily encapsulated in a machine learning framework like Flux.jl [17]. As opposed to Tensorflow, no code has to be ported to a domain specific implementation and it can be transparently included in Flux.jl models.

We expect that every type of accelerator comes with its strength and weaknesses that will shape the algorithm that fits the architecture best. We have shown that CPUs are a natural fit for sparse algebra with their deep pipelines and large caches, whereas GPUs fit very well with the matrix-vector operations in iterative linear solvers. We also apply the block-Jacobi algorithm with a very large number of blocks. That is a domain in which it was rarely used on the CPU, but seems to be promising on the GPU. With the advent of GPUs as the main source of performance on upcoming systems, we expect a new trend of algorithms used for nonlinear programming in complex system problems.

References

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

[2] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE, 2004, pp. 75–86.

[3] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202–3216.

[4] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, T. R. Scogland, RAJA: Portable performance for large-scale scientific applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019, pp. 71–81.

[5] A. Montoison, D. Orban, contributors, Krylov.jl: A Julia basket of hand-picked Krylov methods, https://github.com/JuliaSmoothOptimizers/Krylov.jl (June 2020).

[6] T. J. Ypma, Historical development of the Newton–Raphson method, SIAM review 37 (4) (1995) 531–551.

[7] Y. Saad, M. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal on Scientific and Statistical Computing 7 (3) (1986) 856–869.

[8] A. Flueck, H.-D. Chiang, Solving the nonlinear power flow equations with an inexact newton method using GMRES, IEEE Transactions on Power Systems 13 (2) (1998) 267–273.

[9] 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) (1992) 631–644.

[10] J. Revels, M. Lubin, T. Papamarkou, Forward-mode automatic differentiation in Julia, arXiv:1607.07892 [cs.MS] (2016). URL https://arxiv.org/abs/1607.07892

[11] I. Dunning, J. Huchette, M. Lubin, JuMP: A modeling language for mathematical optimization, SIAM Review 59 (2) (2017) 295–320.

[12] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd Edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A system for large-scale machine learning, in: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), 2016, pp. 265–283.

[14] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, V. Dixit, Diffeqflux.jl - A Julia library for neural differential equations, CoRR abs/1902.02376 (2019). arXiv:1902.02376. URL https://arxiv.org/abs/1902.02376

[15] K. Świrydowicz, S. Thomas, J. Maack, S. Peles, G. Kestor, J. Li, Linear solvers for power grid optimization problems: a review of gpu-accelerated solvers, Parallel ComputingSubmitted (2020).

[16] M. Innes, A. Edelman, K. Fischer, C. Rackauckus, E. Saba, V. B. Shah, W. Tebbutt, Zygote: A differentiable programming system to bridge machine learning and scientific computing, arXiv preprint arXiv:1907.07587 (2019) 140. URL https://arxiv.org/abs/1907.07587

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