Large-scale optimization on GPUs

François Pacaud
frapac.github.io

CAS, Mines Paris - PSL

Journée MesoPSL
October 28, 2026

My background

Who am I?

- Researcher in mathematical optimization
- Former postdoc at Argonne National Lab
- Now at *Centre Automatique et Systèmes* (CAS), Mines Paris-PSL

Heavy GPU user since 2020

Principal use-cases:

- GPU-accelerated mathematical optimization
- Automatic differentiation
- Sparse linear algebra

Motivation

All roads lead to Newton

Finding the roots of a nonlinear system of equations modeling a physical system:

$$F(x) = 0$$

Starting from an initial point x_0, iterate:

$$x_{k+1} = x_k - \Big(JF(x_k)\Big)^{-1} F(x_k)$$

Computational bottlenecks

1. Evaluations of $F(x_k)$ and $JF(x_k)$
2. Solution of the linear system $(JF(x_k))^{-1}F(x_k)$

Examples

- ODEs and PDEs
- Power-flow
- **Nonlinear optimization and interior-point method**

Timeframe: my experience with GPUs



2020-2022

- Part of the **Exascale** project funded by the US Department of Energy (DOE)
  - Prototyping on local NVIDIA V100s, deploying on Frontier
  - High emphasis on portability on heterogeneous hardware (Intel, AMD)
- Back then, we could rely only on `cuSPARSE` and **dense** linear solvers in `cuSOLVER`
  - Main emphasis put on iterative solvers (overlapping Schwarz, block Krylov)
  - Programming multiple GPUs in parallel using MPI (unified memory was already the key)

2023-2026

- Release of the sparse linear solver `cuDSS` in late 2023
- New generation of hardware (H100): memory is no longer a limit
- But everything is getting so expensive... and more diverse

Building extensively on the Julia ecosystem





Outstanding support for GPU programming through NVPTX

- CUDA.jl
- CUDSS.jl
- KernelAbstractions.jl

Optimization-premium

- Autodiff with ForwardDiff.jl / Enzyme.jl
- Modeling with JuMP.jl and ExaModels.jl

Personal opinion

- Valid alternative to C++/Kokkos for HPC
- More reliable than Jax (want to do sparse in Jax? :-))
- BUT: JIT is sometimes annoying, lack of support for ahead-of-time compilation

Take-away

Sparse linear algebra is key for scalability! But is memory-bound ...

Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization

WILLIAM F. TINNEY, SENIOR MEMBER, IEEE, AND JOHN W. WALKER, MEMBER, IEEE

Abstract—Matrix inversion is very inefficient for computing direct solutions of the large sparse systems of linear equations that arise in many network problems. Optimally ordered triangular factorization of sparse matrices is more efficient and offers other important computational advantages in some applications. With this method, direct solutions are computed from sparse matrix factors instead of from a full inverse matrix, thereby gaining a significant advantage in speed, computer memory requirements, and reduced round-off error. Improvements of ten to one or more in speed and problem size over present applications of the inverse can be achieved in many cases.

very inefficient. In general, the matrix of the equations formed from the given conditions of a network problem is sparse, whereas its inverse is full. By means of an appropriately ordered triangular decomposition, the inverse of a sparse matrix can be expressed as a product of sparse matrix factors, thereby gaining an advantage in computational speed, storage, and reduction of round-off error.

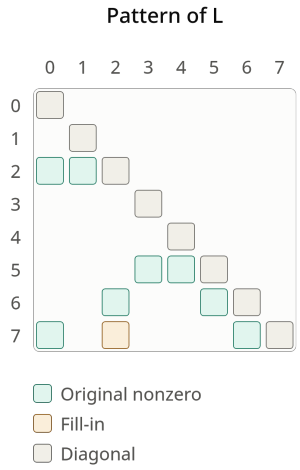
NVIDIA cuDSS Library Removes Barriers to Optimizing the US Power Grid

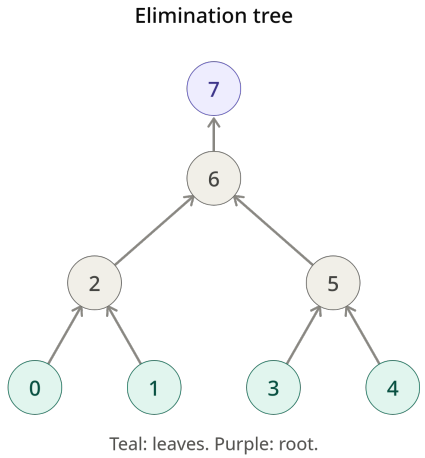


Nov 19, 2024 +15 Like Discuss (0)

By Sungho Shin, François Pacaud, Alexis Montoison, Mark Wolf and Becca Zandstein

How to solve efficiently a linear system $Ax = b$?





Workflow

1. *Symbolic factorization:* Analyze the matrix sparsity pattern and find a static ordering that reduces the fill-in
2. *Numerical factorization:* Compute inplace the nonzero values in L and U
3. *Backsolve:* Solve the system $L^{-1}b$ and $U^{-1}d$ to get solution of the linear system

$$A = LU \quad \Longrightarrow A^{-1}b = U^{-1}L^{-1}b$$

Other factorizations: Cholesky ($A = LL^\top$), signed Cholesky ($A = LDL^\top$), Bunch-Kaufman ($A = LBL^\top$)

Solving sparse linear system on the GPU

GLU/cusolverRF

Previous generation of sparse direct solvers on the GPU
Advantage: you could pass a symbolic factorization computed by KLU

cuSPARSE

Sparse routines optimized for the GPU, such as:

- **SpMV**: sparse matrix-vector multiplication
- **SpMM**: sparse matrix-matrix multiplication
- **SpSV**: triangular solve with a vector
- **SpSM**: triangular solve with a matrix

NVIDIA cuDSS

Released in November 2023

- Implement sparse LU, Cholesky and signed-Cholesky
- Compute only the symbolic factorization on the CPU
- Support **batch** solution of linear systems!
- Version 1.0 has been released **last week**

Remember the operations remain *memory-bound* (no miracle here)

Numerical results: time-to-solution

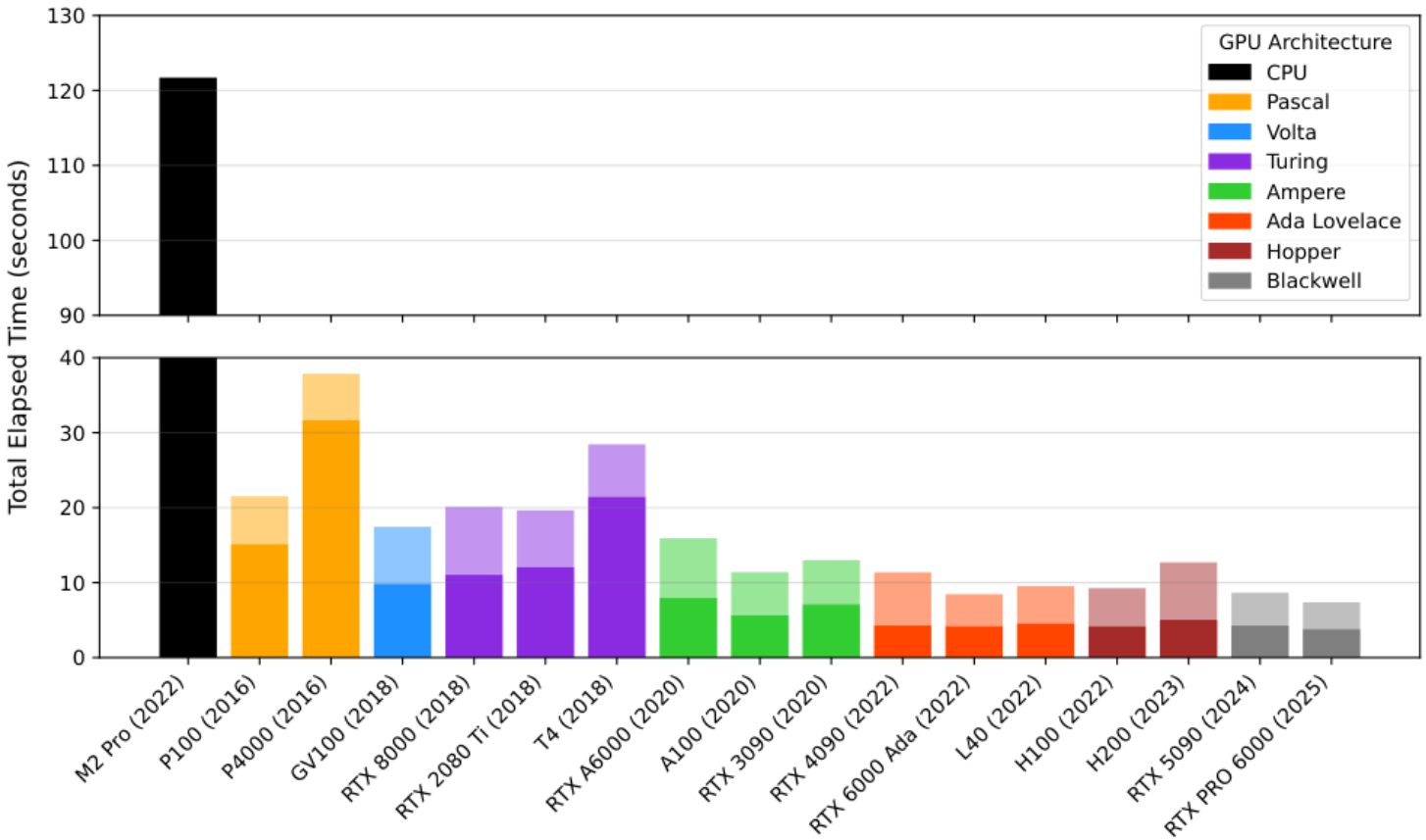


Figure: Time to optimality, here for a large-scale optimal power flow instance.

Image courtesy of Sungho Shin

Evaluating sparse derivatives on the GPU

Observation

- No, Jax is not all what you need!
- Use specialized algebraic modeling system like `ExaModels`

- Exploit repetitive patterns, e.g.:

$$\min_{x^\flat \le x \le x^\sharp} \sum_{l \in [L]} \sum_{i \in [I_l]} f^{(l)}(x; p_i^{(l)}) \qquad \textbf{(SIMD abstraction)}$$

$$\text{subject to } \left[g^{(m)}(x; q_j) \right]_{j \in [J_m]} + \sum_{n \in [N_m]} \sum_{k \in [K_n]} h^{(n)}(x; s_k^{(n)}) = 0, \quad \forall m \in [M]$$

- Specify the repeated patterns explicitly with **iterable objects**

```
constraint(c, 3 * x[i+1]^3 + 2 * sin(x[i+2]) for i = 1:N-2)
```

- **For each repeated pattern**, the derivative evaluation kernel is constructed & compiled, **executed in parallel over multiple data**

Comfortable speed-up are observed

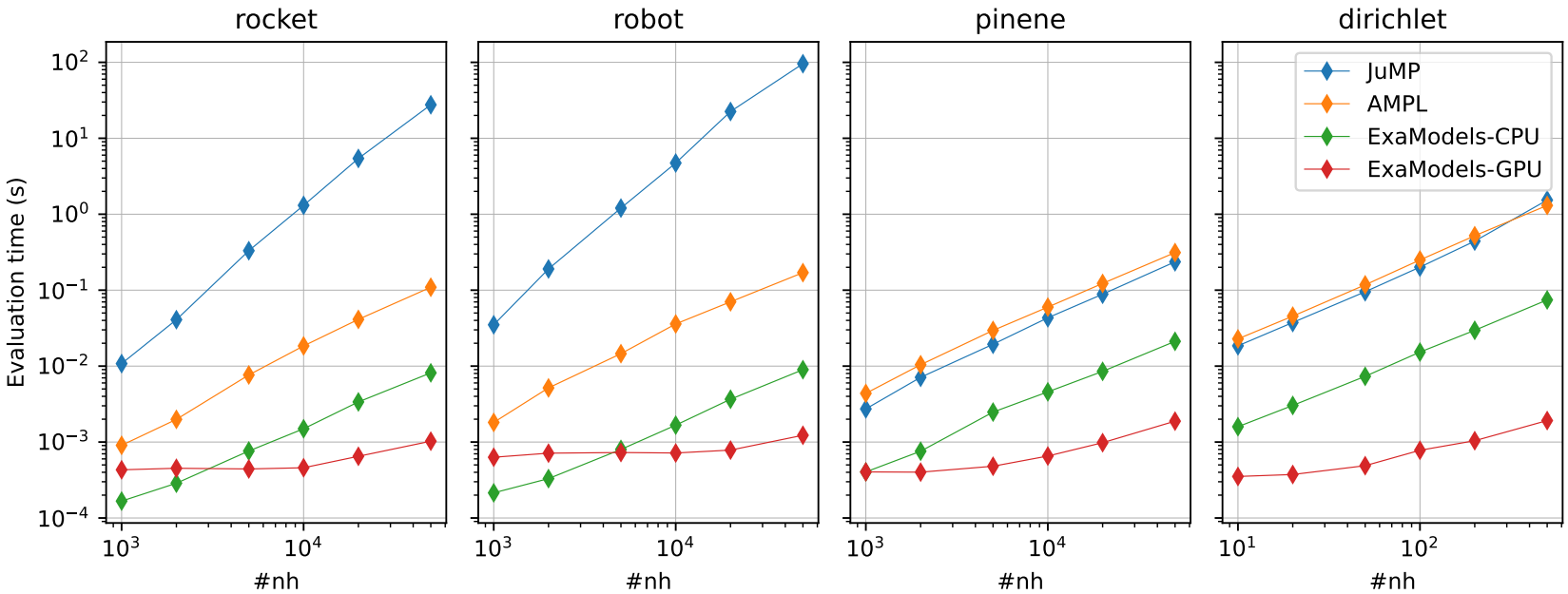


Figure: Discretized optimization problems from the COPS benchmark.

How about portability?

We should not put all our money on NVIDIA

Use a **portability layer** to avoid getting stuck in the CUDA ecosystem!
E.g. Kokkos, KernelAbstractions,...

Alternative backends

- Apple Metal
- Intel oneAPI
- AMD rocM
  - Currently working on a sparse linear solver
  - Better support for `fp64`

And we are living in a different world!

My colleague Michel Schanen managed to reimplement a **sparse** direct solver from scratch in pure Julia, using Claude Fable ... over **one** week-end

https://github.com/exanauts/SparseDirectSolver.jl/

An opportunity to specialize the code further to your application?

Conclusion

Anyone interested in a collaboration?

Looking for challenging problem in scientific computing