# ExaModels.jl: a SIMD Abstraction for GPU-Accelerated Nonlinear Programming

Pattern-aware modeling and coloring-free automatic differentiation

Sungho Shin
shin.mit.edu

Massachusetts Institute of Technology

SIAM Optimization 2026 | Edinburgh, 5 June 2026


Sungho Shin
MIT


François Pacaud
MINES-Paris


Alexis Montoison
AMD


Michel Schanen
Argonne


Mihai Anitescu
Argonne

# Who are we?

https://madsuite.org/





- Sungho Shin @ MIT (an ANL alumnus)
- François Pacaud @ MINES Paris-PSL (an ANL alumnus)
- Alexis Montoison @ AMD (an ANL alumnus)
- Michel Schanen @ Argonne National Lab
- Mihai Anitescu @ Argonne National Lab
- and friends ... Michael Saunders, Dominic Orban, Armin Nurkanović, Anton Pozharskiy, Jean-Baptiste Caillau, Michael Klamkin, ...

# What is ExaModels.jl?

- Algebraic modeling system in Julia, JuMP.jl-inspired syntax
- Builds on `NLPModels.jl`—compatible with `Ipopt`, `MadNLP`, `KNITRO`, `Uno`, ... (JSO-style solvers)
- Ships its own AD system—$f(x)$, $g(x)$, $\nabla_x f(x)$, $\nabla_x g(x)$, $\nabla^2_{xx} L(x, \lambda)$
- **Distinguishing feature:** GPU-compatible modeling—more on this shortly
- **Design philosophy:** optimized for run-time performance, compromising ease-of-use

*Example: Goddard rocket — maximize final altitude.*

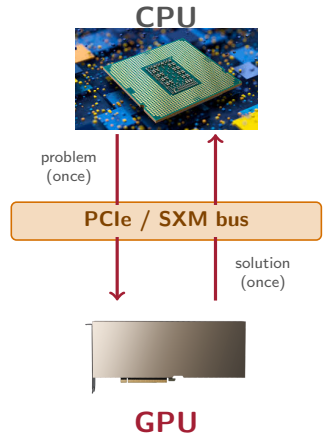
```
using ExaModels, NLPModelsIpopt

nh, h_0, v_0, m_0, g_0 = 200, 1.0, 0.0, 1.0, 1.0
T_c, h_c, v_c, m_c = 3.5, 500.0, 620.0, 0.6
c_e = 0.5*sqrt(g_0*h_0); m_f = m_c*m_0
D_c = 0.5*v_c*(m_0/g_0); T_max = T_c*m_0*g_0

core = ExaCore(minimize=false, concrete=Val(true))

@add_var(core, h, 0:nh; start = 1.0, lvar = 1.0)
@add_var(core, v, 0:nh; start = (i/nh*(1-i/nh) for i=0:nh))
@add_var(core, m, 0:nh; start = ((m_f-m_0)*i/nh+m_0 for i=0:nh),
         lvar = m_f, uvar = m_0)
@add_var(core, T, 0:nh; start = T_max/2, lvar = 0.0, uvar = T_max)
@add_var(core, step, 1; start = 1/nh, lvar = 0.0)

@add_obj(core, h[nh])

@add_con(core, c1, -h[i] + h[i-1]
         + 0.5*step[1]*(v[i]+v[i-1]) for i=1:nh)
@add_con(core, c2,
    -v[i] + v[i-1] + 0.5*step[1]*(
        (T[i] - D_c*v[i]^2*exp(-h_c*(h[i]-h_0))/h_0
         - m[i]*g_0*(h_0/h[i])^2)/m[i]
      + (T[i-1] - D_c*v[i-1]^2*exp(-h_c*(h[i-1]-h_0))/h_0
           - m[i-1]*g_0*(h_0/h[i-1])^2)/m[i-1])
    for i=1:nh)
@add_con(core, c3, -m[i] + m[i-1]
         - 0.5*step[1]*(T[i]+T[i-1])/c_e for i=1:nh)
@add_con(core, h[0] - h_0); @add_con(core, v[0] - v_0)
@add_con(core, m[0] - m_0); @add_con(core, m[nh] - m_f)

model = ExaModel(core)
result = ipopt(model)
```

# Why a GPU-compatible modeling language?

LP / QP / DCP / MIP — not strictly needed

- Canonical forms $(A, b, c, \ldots) \Rightarrow$ one-shot CPU→GPU copy of problem data
- Solver iterates entirely on the device; solution copied once back to CPU
- Transfer cost is amortized across many solver iterations
- Existing GPU-native solvers already work this way: `cuPDLP`, `CuClarabel` (LP / QP / SOCP), MIP frontends building on the same canonical-form transfer

  $\Rightarrow$ For LP / QP / DCP / MIP, a GPU-native *modeling* language adds little

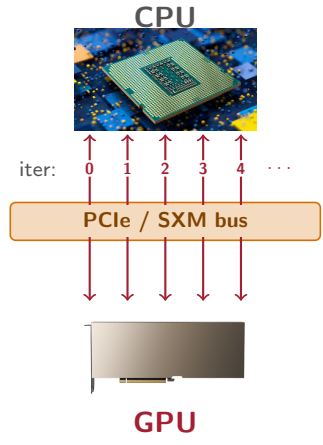


# Why a GPU-compatible modeling language?

NLP — the picture is different

- Solver queries the modeling layer every iteration via callbacks—
  $f(x)$, $g(x)$, $\nabla_x f(x)$, $\nabla_x g(x)$, $\nabla^2_{xx} L(x, \lambda)$
- Each query depends on the current iterate—no one-shot transfer
- Without GPU-native callbacks, every iteration pays a CPU↔GPU round-trip: significant per-iteration overhead, no amortization

  ⇒ NLP needs a GPU-compatible modeling language that supplies GPU-resident callbacks



# Pattern–data separation: the Goddard rocket

Goddard, 1919—maximize final altitude under aerodynamic drag and gravity, discretized by the trapezoidal rule with $T$ time steps:

$$\begin{aligned}
\max_{h_t, v_t, m_t, \tau_t, \Delta t} \; h_T \quad \text{s.t.} \quad & h_t = h_{t-1} + \tfrac{\Delta t}{2}(v_t + v_{t-1}), && t = 1, \dots, T \\
& v_t = v_{t-1} + \tfrac{\Delta t}{2}\left(\tfrac{\tau_t - D(h_t, v_t)}{m_t} - g(h_t) + \cdots\right), && t = 1, \dots, T \\
& m_t = m_{t-1} - \tfrac{\Delta t}{2c}(\tau_t + \tau_{t-1}), && t = 1, \dots, T \\
& h_0 = h^0,\ v_0 = 0,\ m_0 = m^0,\ m_T = m^f.
\end{aligned}$$

- Each dynamics constraint has the same functional form for every $t$—a single algebraic pattern evaluated over $T$ data points
- Initial/boundary conditions share one template; with the objective, four distinct patterns total—regardless of $T$

Goddard (1919). *A Method of Reaching Extreme Altitudes.* Smithsonian Miscellaneous Collections. Smithsonian Institution

# Algebraic modeling systems and the abstraction gap

An AMS bridges the user's mathematical formulation and the solver, producing the callbacks $f(x)$, $\nabla f(x)$, $g(x)$, $\nabla g(x)$, $\nabla^2_{xx} L(x, \lambda)$.

| System | Language | AD | Pattern-aware |
|---|---|---|---|
| AMPL | own | ASL (built-in) | no |
| JuMP.jl | Julia | reverse-mode on graph | no |
| CasADi | C++/Py/MATLAB | source transform | yes/no |
| Gravity | C++ | symbolic | yes |
| ExaModels.jl | Julia | reverse-mode on graph | yes |

- AMPL and JuMP.jl flatten indexed constraints into $N$ scalar nodes before AD
- Two strategies preserve indexed structure: vectorization (CasADi MX) and templatizing (Gravity, ExaModels.jl)

# This talk

*What modeling abstraction preserves the pattern structure of large-scale NLPs, and how can we harness it for efficient, GPU-compatible derivative evaluation?*

**Contribution**

SIMD abstraction of NLPs: represent a problem as a small set of algebraic patterns, each evaluated over many data points. Pattern structure is recognized at compile time by the Julia compiler, which specializes a dedicated derivative evaluator per pattern; evaluation reduces to embarrassingly parallel loops. (Per-pattern sparsity is precomputed at model construction, so no graph coloring.)

Consequence: the same model code runs on CPU and on NVIDIA, AMD, Intel, and Apple GPUs via KernelAbstractions.jl (Churavy, 2025). Up to 50× speed-up on the sparse Lagrangian Hessian (`hess_coord!`) over single-thread CPU for large COPS instances.

Churavy (2025). *KernelAbstractions.Jl*.

# The SIMD abstraction of NLPs

Represent the NLP as a small set of algebraic patterns evaluated over data iterators:

$$\min_{x^\flat \le x \le x^\sharp} \sum_{\ell \in [L]} \sum_{i \in [I_\ell]} f^{(\ell)}(x;\, p_i^{(\ell)})$$
$$\text{s.t. } g^\flat \le \left[ g^{(m)}(x;\, q_j^{(m)}) \right]_{j \in [J_m]} + \sum_{n \in [N_m]} \sum_{k \in [K_n]} h^{(n)}(x;\, s_k^{(n)}) \le g^\sharp, \quad \forall\, m \in [M]$$

# The SIMD abstraction of NLPs

Represent the NLP as a small set of algebraic patterns evaluated over data iterators:

$$\min_{x^\flat \le x \le x^\sharp} \sum_{\ell \in [L]} \sum_{i \in [I_\ell]} f^{(\ell)}(x;\, p_i^{(\ell)})$$
$$\text{s.t. } g^\flat \le \left[ g^{(m)}(x;\, q_j^{(m)}) \right]_{j \in [J_m]} + \sum_{n \in [N_m]} \sum_{k \in [K_n]} h^{(n)}(x;\, s_k^{(n)}) \le g^\sharp, \quad \forall\, m \in [M]$$

- $f^{(\ell)}$, $g^{(m)}$, $h^{(n)}$ — twice-differentiable scalar algebraic patterns

# The SIMD abstraction of NLPs

Represent the NLP as a small set of algebraic patterns evaluated over data iterators:

$$\min_{x^\flat \le x \le x^\sharp} \sum_{\ell \in [L]} \sum_{i \in [I_\ell]} f^{(\ell)}(x;\, p_i^{(\ell)})$$
$$\text{s.t. } g^\flat \le \left[ g^{(m)}(x;\, q_j^{(m)}) \right]_{j \in [J_m]} + \sum_{n \in [N_m]} \sum_{k \in [K_n]} h^{(n)}(x;\, s_k^{(n)}) \le g^\sharp, \quad \forall\, m \in [M]$$

- $f^{(\ell)}$, $g^{(m)}$, $h^{(n)}$ — twice-differentiable scalar algebraic patterns
- Each $h^{(n)}$ augmentation term is a vector with a single nonzero entry — arises in network models (e.g., arc-flow contributions to bus balance equations)

# The SIMD abstraction of NLPs

Represent the NLP as a small set of algebraic patterns evaluated over data iterators:

$$\min_{x^\flat \le x \le x^\sharp} \sum_{\ell \in [L]} \sum_{i \in [I_\ell]} f^{(\ell)}(x;\, p_i^{(\ell)})$$
$$\text{s.t. } g^\flat \le \left[ g^{(m)}(x;\, q_j^{(m)}) \right]_{j \in [J_m]} + \sum_{n \in [N_m]} \sum_{k \in [K_n]} h^{(n)}(x;\, s_k^{(n)}) \le g^\sharp, \quad \forall\, m \in [M]$$

- $f^{(\ell)}$, $g^{(m)}$, $h^{(n)}$ — twice-differentiable scalar algebraic patterns
- Each $h^{(n)}$ augmentation term is a vector with a single nonzero entry — arises in network models (e.g., arc-flow contributions to bus balance equations)
- $p_i^{(\ell)}$, $q_j^{(m)}$, $s_k^{(n)}$ — data points that parameterize each pattern instance

# The SIMD abstraction of NLPs

Represent the NLP as a small set of algebraic patterns evaluated over data iterators:

$$\min_{x^\flat \le x \le x^\sharp} \sum_{\ell \in [L]} \sum_{i \in [I_\ell]} f^{(\ell)}(x;\, p_i^{(\ell)})$$

$$\text{s.t. } g^\flat \le \left[ g^{(m)}(x;\, q_j^{(m)}) \right]_{j \in [J_m]} + \sum_{n \in [N_m]} \sum_{k \in [K_n]} h^{(n)}(x;\, s_k^{(n)}) \le g^\sharp, \quad \forall\, m \in [M]$$

- $f^{(\ell)}$, $g^{(m)}$, $h^{(n)}$ — twice-differentiable scalar algebraic patterns
- Each $h^{(n)}$ augmentation term is a vector with a single nonzero entry — arises in network models (e.g., arc-flow contributions to bus balance equations)
- $p_i^{(\ell)}$, $q_j^{(m)}$, $s_k^{(n)}$ — data points that parameterize each pattern instance
- $L$, $M$, $|N_m|$ small and fixed; $I_\ell$, $J_m$, $K_n$ scale with problem size

# User syntax: a constraint as pattern + iterator

Zoom in on the altitude dynamics from the Goddard rocket:

```
@add_con(core, c1, -h[i] + h[i-1] + 0.5*step[1]*(v[i] + v[i-1]) for i = 1:nh)
```

- The body `-h[i]+h[i-1] + 0.5*step[1]*(v[i]+v[i-1])` is the algebraic pattern—one expression tree
- The generator `for i = 1:nh` is the data iterator—`nh` data points
- ExaModels.jl compiles once per pattern: model construction, function evaluation, first + second derivatives, Jacobian / Hessian sparsity structure

# Type stability $\Rightarrow$ specialization per pattern

Dispatch on sentinel sources (`VarSource`, `DataSource`) bakes the algebraic structure into the Julia type:

```
julia> x = ExaModels.VarSource()
VarSource
  (use fulltype(node) or IOContext(io, :fulltype => true) for full type)
  x

julia> i = ExaModels.DataSource()
DataSource
  (use fulltype(node) or IOContext(io, :fulltype => true) for full type)
  i

julia> x[i-1]
Var
  (use fulltype(node) or IOContext(io, :fulltype => true) for full type)
  x[i - 1]

julia> x[i-1]^2
Node1{abs2}
  (use fulltype(node) or IOContext(io, :fulltype => true) for full type)
  x[i - 1]^2
```

# Type stability ⇒ specialization per pattern

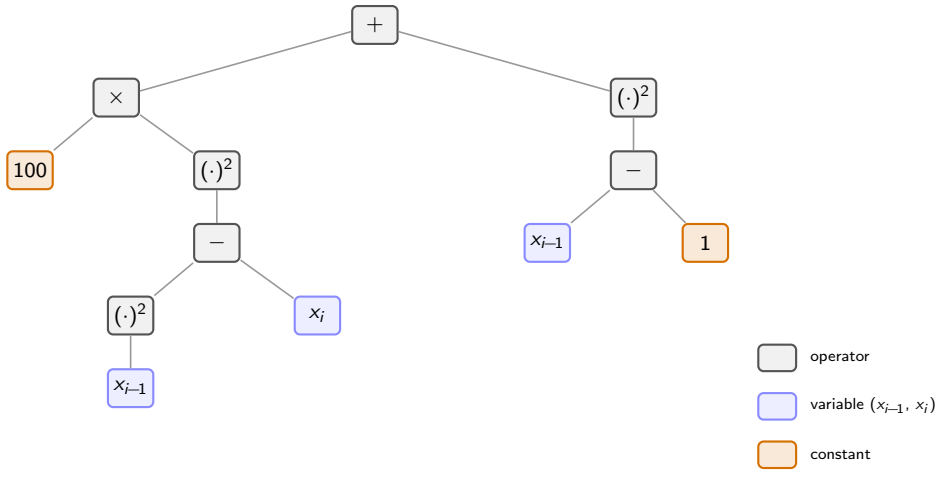
Dispatch on sentinel sources (`VarSource`, `DataSource`) bakes the algebraic structure into the Julia type:

```
julia> x = ExaModels.VarSource()
VarSource
  (use fulltype(node) or IOContext(io, :fulltype => true) for full type)
  x

julia> i = ExaModels.DataSource()
DataSource
  (use fulltype(node) or IOContext(io, :fulltype => true) for full type)
  i

julia> typeof(x[i-1])
Var{Node2{typeof(-),ExaModels.DataSource,Int64}}

julia> typeof(x[i-1]^2)
Node1{typeof(abs2),Var{Node2{typeof(-),ExaModels.DataSource,Int64}}}
```

- Each pattern ⇒ a distinct concrete type ⇒ a separate inlined, specialized kernel—no hot-path inference; exactly what GPU code generation needs

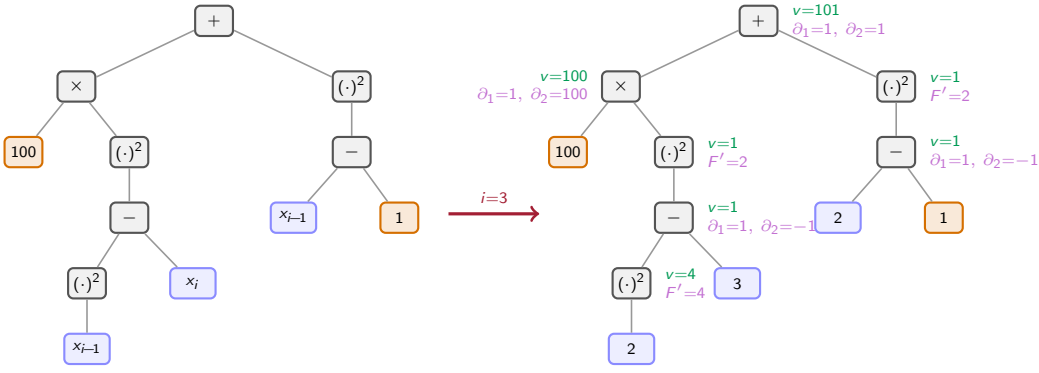
# Expression tree for the Rosenbrock pattern

$f_i(x) = 100\,(x_{i-1}^2 - x_i)^2 + (x_{i-1} - 1)^2$



# Reverse-mode AD: pattern → computation graph

$f_i(x) = 100\,(x_{i-1}^2 - x_i)^2 + (x_{i-1} - 1)^2; \quad i = 3,\ x = [1, 2, 3, 4, 5]$

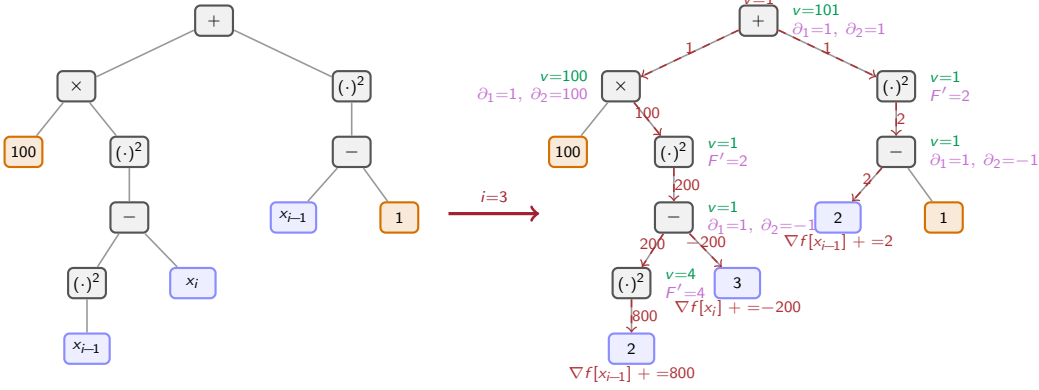



**Expression tree** (pattern) **Computation graph** (instance, $i$=3)

Same shape, every node carries a primal $v$. Root $v = f_i(x) = 101$. Forward pass fills the green annotations.

# Reverse-mode AD: pattern → computation graph

$f_i(x) = 100\,(x_{i-1}^2 - x_i)^2 + (x_{i-1} - 1)^2; \quad i = 3,\ x = [1, 2, 3, 4, 5]$




**Expression tree** (pattern) **Computation graph** (instance, $i$=3)

Reverse pass (dashed): adjoint $\bar{v}$ flows root → leaves; `Var` leaves accumulate $\nabla f_i$. Hessian: two more passes, same blueprint.

# What's new: textbook AD on a templatized tree

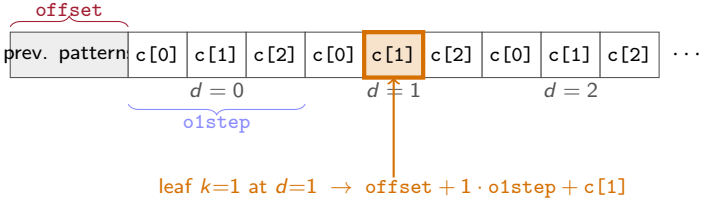
- The forward and reverse passes above are standard reverse-mode AD—no algorithmic innovation
- The one twist: the expression tree is a Julia type, fully type-stable and visible to the compiler at construction

$\Rightarrow$ each pattern compiles to a small, dedicated AD kernel that walks the tape at run time—no interpretation, no dynamic dispatch, no monolithic engine

- Each data point evaluates the same kernel independently $\Rightarrow$ embarrassingly parallel: SIMD on CPU, one thread per data point on GPU
- Second-order Hessian follows the same blueprint:
  - Diagonal pass (`hrpass`): identical structure, just propagates a second adjoint
  - Off-diagonal pass (`hdrpass`): walks two subtrees at once—slightly more bookkeeping, same parallelism story

# Per-pattern sparsity at model construction

- Parameterized sparsity pattern can be analyzed first.
- Probe pass: forward-pass the pattern with specialized $x$ and $i$ types $\rightarrow$ a fully resolved computation graph (which leaves the pattern touches, in what order)
- Compress: read off duplicate leaves from the graph (e.g., $x_i$ visited twice) and map each leaf to a unique COO slot. Stored as an `NTuple{N,Int}` type parameter
- Output layout: pattern starts at `offset`; `o1step` = slots per data point; leaf $k$ at data point $d$ writes to offset $+ d \cdot$ o1step $+$ comp$[k]$



# jac_structure! / hess_structure!

```
jac_structure!(nlp, rows, cols)    # COO (i,j) of Jacobian nonzeros
hess_structure!(nlp, rows, cols)   # COO (i,j) of Hessian nonzeros
```

- Sparsity callbacks ask for the $(i, j)$ coordinates of nonzeros, not their values
- Reuse the AD machinery verbatim: the same forward and reverse pass code that evaluates $\nabla f$ and $\nabla^2 \mathcal{L}$ also walks every $(i, j)$ pair we need—we just run the forward pass on a dummy array `x`, since only the indices matter, not the values
- Specialize only at the leaves: in the value pass, a leaf accumulates $H[i, j] \mathrel{+}= v$; in the structure pass, the leaf simply emits the tuple $(i, j)$ into the COO index buffer—one implementation, the leaf behavior chosen by multiple dispatch on the leaf type while the rest of the pass is identical

# Pattern → kernel, data point → thread

Pick the backend at construction: `ExaCore(; backend = CUDABackend())`.

**CPU:** loop over data points

```
# con: compiled pattern + its data
function _cons!(con, x, g)
  @simd for i in eachindex(con.itr)
    @inbounds g[offset0(con,i)] =
      con.f(con.itr[i], x)
  end
end
```

**GPU:** one thread per data point

```
function _cons!(backend, con, x, g)
  kerf(backend)(g, con.f, con.itr, x;
    ndrange = length(con.itr))
end

@kernel function kerf(g, @Const(f),
    @Const(itr), @Const(x))
  I = @index(Global)
  @inbounds g[offset0(f, itr, I)] =
    f(itr[I], x)
end
```

GPU portability via `KernelAbstractions.jl` (Churavy, 2025) atop `GPUCompiler.jl` (Besard et al., 2019)—same `@kernel` runs on NVIDIA, AMD, Intel, Apple, CPU.

Churavy (2025). *KernelAbstractions.jl*.
Besard et al. (2019). "Effective Extensible Programming: Unleashing Julia on GPUs". *IEEE Transactions on Parallel and Distributed Systems*

# GPU-specific implementation details

- Data-level parallelism only. Each pattern is compiled once and executed across many data points; no intra-tree parallelism.
- Jacobian / Hessian: COO out, solver does the rest. Each pattern emits $(i, j, v)$ triples into its own contiguous slice of the COO array—disjoint writes, no atomics, no on-device CSC conversion. The downstream solver (MadNLP + cuDSS / Krylov) reduces COO $\rightarrow$ CSC for the linear solve.
- Objective: buffer + reduce. Each thread writes its pattern evaluation to a buffer of length $\sum_\ell l_\ell$; the scalar is obtained by a parallel reduction. Avoids atomic adds to a single scalar.
- Gradient: sparse + compress. Multiple patterns may contribute to the same gradient entry. Each thread writes to a sparse buffer indexed by (variable, position); a follow-up `compress_to_dense` kernel reduces by target index.

# Benchmark suites and methodology

- Benchmark library
  - COPS (Dolan et al., 2001): 18 problems from optimal control, PDE-constrained, parameter estimation, mesh
  - AC OPF from PGLIB (Babaeinejadsarookolaee et al., 2021): 66 instances $\times$ 2 formulations (polar / rectangular), 3 to 78,484 buses
- Aggregation: shifted geometric mean (SGM)

$$\mathrm{SGM}(t; s) = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \ln(t_i + s)\right) - s, \qquad s = 10^{-5}\,\mathrm{s}$$

Dolan et al. (2001). *Benchmarking Optimization Software with COPS.*. Tech. rep. Argonne National Lab., IL (US)
Babaeinejadsarookolaee et al. (2021). *The Power Grid Library for Benchmarking AC Optimal Power Flow Algorithms.* arXiv: 1908.02788 [math]

# Timing GPU callbacks: btime

```
# Minimum wall time per call. Without synchronize, GPU calls return
# after dispatch (~2e-4 s) and we only time the launch, not the compute.
function btime(f, backend; seconds = 2.0)
    f(); KA.synchronize(backend)              # warm-up
    GC.gc()
    t0 = time_ns(); f(); KA.synchronize(backend)
    dt = (time_ns() - t0) / 1e9
    N = max(3, min(10_000, round(Int, seconds / max(dt, 1e-9))))
    return minimum(begin
        t = time_ns()
        f()                                   # e.g. hess_coord!(m, x, y, hv)
        KA.synchronize(backend)               # block until GPU work done
        (time_ns() - t) / 1e9
    end for _ = 1:N)
end
```

# Hardware platforms

Four platforms span the major GPU vendors. The CPU host is the same Intel Xeon for H1–H3.

| Label | CPU | GPU | RAM | VRAM | Driver | Precision |
|---|---|---|---|---|---|---|
| H1 | 2×Gold 6130 (32c) | NVIDIA Quadro GV100 | 1 TB | 32 GB | CUDA 12.0 | fp64 |
| H2 | 2×Gold 6130 (32c) | AMD Radeon VII | 256 GB | 16 GB | ROCm 6.3.4 | fp64 |
| H3 | 2×Gold 6130 (32c) | Intel Arc A770 | 256 GB | 16 GB | oneAPI 2025.1 | fp32 |
| H4 | Apple M2 Pro (12c) | Apple M2 Pro | 16 GB | shared | Metal 3 | fp32 |

# CPU AD performance: ExaModels.jl vs virtual best of AMPL / JuMP.jl

Single-threaded CPU on H1. Speedup = per-instance min(AMPL, JuMP.jl) / ExaModels.jl, aggregated by SGM ($s = 10^{-5}$ s). Size classes by max(nnzj, nnzh).

| **Suite** | **Callback** | **Small** nnz $< 10^3$ | **Medium** $10^3 \leq$ nnz $< 10^5$ | **Large** $10^5 \leq$ nnz |
|---|---|---|---|---|
| COPS | `grad!` | 1.9× | 4.5× | 8.7× |
| | `jac_coord!` | 17.4× | 5.0× | 9.4× |
| | `hess_coord!` | 4.7× | 1.6× | 5.9× |
| OPF-polar | `grad!` | 3.4× | 14.9× | 41.7× |
| | `jac_coord!` | 3.8× | 7.3× | 27.6× |
| | `hess_coord!` | 6.7× | 11.5× | 45.9× |
| OPF-rect | `grad!` | 2.8× | 6.4× | 6.5× |
| | `jac_coord!` | 7.3× | 13.1× | 36.2× |
| | `hess_coord!` | 4.5× | 7.1× | 28.6× |

**Disclaimer:** AMPL and JuMP.jl evaluate per-scalar—apples-to-oranges. AMPL `.nl` via `AmplNLWriter.jl`; JuMP.jl default AD (`SymbolicAD.jl` faster, unmeasured).

# GPU acceleration: speedup over single-thread CPU

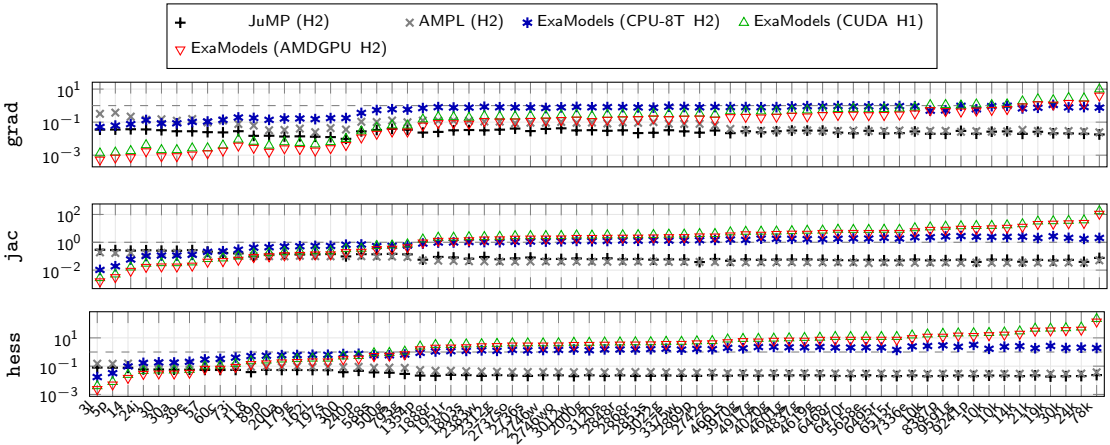
GPU speedup at Large instances (nnz $\geq 10^5$). Baseline: single-thread Xeon Gold 6130 (2017, 32c on the host, 1 thread used).

| **Suite** | **Callback** | **CPU-8T** Xeon Gold 6130 | **CUDA** GV100 | **AMDGPU** Radeon VII |
|---|---|---|---|---|
| COPS | `grad!` | 1.0× | 4.6× | 2.6× |
| | `jac_coord!` | 1.6× | 22× | 15× |
| | `hess_coord!` | 2.0× | 52× | 36× |
| OPF-polar | `grad!` | 1.0× | 0.7× | 0.3× |
| | `jac_coord!` | 2.0× | 8.2× | 5.1× |
| | `hess_coord!` | 2.2× | 12× | 7.9× |

- Speedup concentrates on Jacobian / Hessian—these scale with sparsity; `grad!` stays CPU-friendly because the OPF objective is a small sum

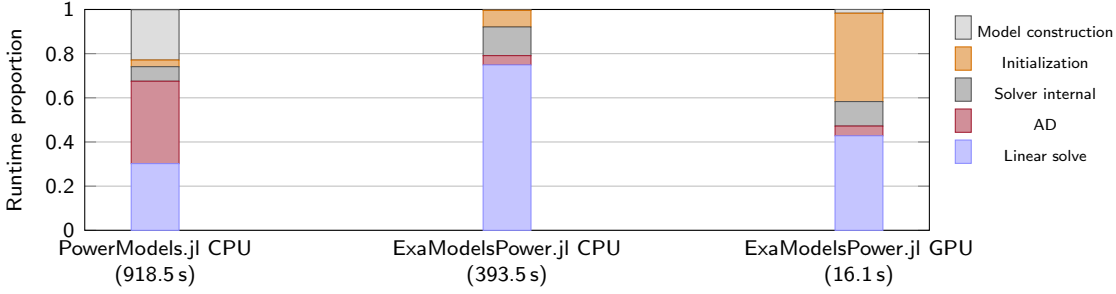
# Per-instance GPU speedup: PGLIB-OPF (polar)

Speedup vs. single-threaded CPU (H2) per instance, sorted by $\text{nnz}_J + \text{nnz}_H$; dashed line = unit speedup.



# End-to-end runtime breakdown: AC OPF (78,484 buses)

PGLIB epigrids ACOPF (polar), tol $10^{-4}$, MadNLP, on H1 (Xeon Gold 6130 + Quadro GV100). Numbers: (Johnson et al., 2025).




- PowerModels → ExaModels (CPU): AD + construction collapse ⇒ linear solve dominates
- ExaModels CPU → GPU (MadNLP.jl + cuDSS): ∼ 24× wall-time reduction; symbolic factorization (initialization) is the next bottleneck

Johnson et al. (2025). *ExaModelsPower.jl: A GPU-Compatible Modeling Library for Nonlinear Power System Optimization*. arXiv: 2510.12897 [eess]

# Beyond AD: GPU advantage carries to the solver

SGM10 (s) on 2× Xeon Gold 6130 + 2× Quadro GV100. GPU: MadIPM / MadNLP + cuDSS. CPU: Gurobi / Ipopt + MA27. ExaModels modeling backend on GPU.

| | Tol | Solver | Small $\text{nnz} < 2^{18}$ | | Medium $2^{18} \leq \text{nnz} < 2^{20}$ | | Large $2^{20} \leq \text{nnz}$ | | Total | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Solved | Time | Solved | Time | Solved | Time | Solved | Time |
| MIPLIB | $10^{-4}$ | MadIPM | 87 | 1.3013 | 56 | 5.0480 | 27 | 19.7925 | 170 | 4.5319 |
| | | Gurobi | 88 | 1.5439 | 58 | 10.4671 | 23 | 78.5783 | 169 | 9.3939 |
| | $10^{-8}$ | MadIPM | 85 | 2.8157 | 48 | 18.2642 | 25 | 33.1676 | 158 | 10.2820 |
| | | Gurobi | 88 | 1.5708 | 58 | 10.6148 | 24 | 76.3206 | 170 | 9.3826 |
| OPF | $10^{-4}$ | MadNLP | 31 | 0.4166 | 24 | 2.6380 | 11 | 3.7040 | 66 | 1.6979 |
| | | Ipopt | 31 | 0.3970 | 24 | 5.0697 | 11 | 38.5053 | 66 | 5.3817 |
| | $10^{-8}$ | MadNLP | 30 | 2.5037 | 24 | 4.6016 | 10 | 12.8040 | 64 | 4.6228 |
| | | Ipopt | 31 | 0.5100 | 24 | 5.4292 | 11 | 37.7818 | 66 | 5.5541 |
| COPS | $10^{-4}$ | MadNLP | 13 | 0.8665 | 15 | 4.8665 | 16 | 3.8194 | 44 | 3.2314 |
| | | Ipopt | 13 | 5.2315 | 15 | 15.9701 | 15 | 45.8411 | 43 | 19.2243 |
| | $10^{-8}$ | MadNLP | 13 | 0.8575 | 16 | 1.5572 | 16 | 8.3549 | 45 | 3.3797 |
| | | Ipopt | 13 | 5.9413 | 15 | 17.6758 | 15 | 40.8639 | 43 | 19.2999 |

**Disclaimer:** different solvers, algorithms, hardware—apples-to-oranges.

Montoison et al. (2025). *GPU Implementation of Second-Order Linear and Nonlinear Programming Solvers.* arXiv: 2508.16094 [math]

# Take-aways

**SIMD abstraction of NLPs**

Represent the problem as a small set of algebraic patterns evaluated over data iterators. The user pays a small expressiveness tax; in exchange the AMS exposes parallel structure to the compiler. Per-pattern sparsity is precomputed at model construction (no graph coloring).

Consequence: pattern $\rightarrow$ kernel, data point $\rightarrow$ thread. Same model code on NVIDIA, AMD, Intel, Apple, and CPU—up to 50$\times$ speed-up on the sparse Lagrangian Hessian (COPS Large, `hess_coord!`), constant in problem size.

- Extensions reuse the same AD: parametric, two-stage stochastic, batch (batch fits despite not being an original goal)
- Full paper: Shin, Pacaud, Montoison, Schanen, Anitescu (Shin et al., 2026)—*coming soon, hopefully this summer*
- `github.com/exanauts/ExaModels.jl` | `ExaModelsPower.jl`

Shin et al. (2026). "ExaModels.jl: An Algebraic Modeling System for Nonlinear Programming on GPUs". In preparation

# References


Babaeinejadsarookolaee, Sogol et al. (2021). *The Power Grid Library for Benchmarking AC Optimal Power Flow Algorithms*. arXiv: 1908.02788 [math].

Besard, Tim et al. (2019). "Effective Extensible Programming: Unleashing Julia on GPUs". *IEEE Transactions on Parallel and Distributed Systems* 30.4, pp. 827–841.

Churavy, Valentin (2025). *KernelAbstractions.Jl*.

Dolan, E. D. et al. (2001). *Benchmarking Optimization Software with COPS*. Tech. rep. ANL/MCS-TM-246. Argonne National Lab., IL (US).

Goddard, Robert H. (1919). *A Method of Reaching Extreme Altitudes*. Smithsonian Miscellaneous Collections 71(2). Smithsonian Institution.

Johnson, Sanjay et al. (2025). *ExaModelsPower.Jl: A GPU-Compatible Modeling Library for Nonlinear Power System Optimization*. arXiv: 2510.12897 [eess].

Montoison, Alexis et al. (2025). *GPU Implementation of Second-Order Linear and Nonlinear Programming Solvers*. arXiv: 2508.16094 [math].

Shin, Sungho et al. (2026). "ExaModels.jl: An Algebraic Modeling System for Nonlinear Programming on GPUs". In preparation.

# ExaModels.jl: a SIMD Abstraction for GPU-Accelerated Nonlinear Programming

Pattern-aware modeling and coloring-free automatic differentiation

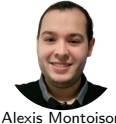
Sungho Shin
shin.mit.edu

Massachusetts Institute of Technology

SIAM Optimization 2026 | Edinburgh, 5 June 2026

Sungho Shin
MIT

François Pacaud
MINES-Paris



Alexis Montoison
AMD

Michel Schanen
Argonne

Mihai Anitescu
Argonne