HPC

# Aerodynamic Optimization of Wind Turbine Blades for Site-Specific Wind Conditions Cupy

Nikita Vybornov

Skoltech

# Problem Statement

The aerodynamic optimization of wind turbine blades requires the numerical solution of the incompressible Navier–Stokes equations in the streamfunction–vorticity formulation. This is performed over transformed computational domains using conformal mapping, where complex airfoil geometries are mapped to simpler domains such as the unit disk. Solving these equations involves iterative implicit time-stepping (Newton's method), Jacobian matrix evaluations, and solving large sparse linear systems—all of which are computationally intensive.

The current implementation, which relies on CPU-based solvers is constrained by memory bandwidth and sequential execution limitations, particularly when dealing with high-resolution grids or running parameter sweeps for optimization under varying site-specific wind conditions.

To overcome these performance bottlenecks, GPU acceleration using CuPy offers a highly parallelizable environment for array-based computations. By leveraging the GPU's architecture, we can significantly reduce execution time.

# Importance

Simulations calculate velocity fields (ur, uθ), vorticity (ω), and pressure distributions to identify blade shapes that maximize lift (energy capture) while minimizing drag (mechanical stress).
Example: Adjusting blade twist/taper based on local flow angles (angle_of_attack) can boost power output by 10–20% in site-specific winds.

Wind conditions vary by location (e.g., coastal vs. mountainous regions). The code's ability to model Re (Reynolds number) and inflow angles (angle_of_attack) ensures blades are optimized for local wind speed, turbulence, and direction.
Example: A turbine in a low-wind urban area might require wider blades, while offshore turbines need designs resistant to saltwater corrosion and high gusts.

GPU-accelerated spectral methods (Chebyshev/Fourier) enable rapid simulation of complex flows without physical prototypes, cutting R&D costs by ~30–50%.
Example: Testing 100 blade designs in simulation vs. building 1 physical prototype saves millions in materials and testing.

Vorticity (ω) fields reveal turbulent regions causing blade vibrations. Optimizing shapes to suppress vortices extends blade lifespan by reducing fatigue.
Example: A blade optimized to minimize trailing-edge vortices might last 15 years instead of 10 in high-turbulence sites.

Aerodynamic simulations predict noise from blade-turbulence interactions. Smoother flow fields reduce acoustic pollution, critical for compliance with regulations near residential areas.
Optimized blades reduce bird/bat collision risks by minimizing chaotic flow patterns.

Unsteady flow simulations (time-resolved ψ, ω) model how gusts or turbulence cause power spikes/drops. This data helps design blades that stabilize energy output for grid compatibility.

# Objective and Methods

Our objective is to optimize the aerodynamic forces acting on the wind turbine blade. To achieve this, we use conformal mapping to transform the problem from a complex blade profile to a unit disk, where we solve the Navier-Stokes equations. The transformation introduces a correction in some terms via the Jacobian. By solving the equations on the disk, we obtain the force distribution on the actual blade geometry.

By the Riemann mapping theorem, there exists a conformal mapping from the exterior of the unit disk to any non-empty, simply connected open subset of the complex plane C, excluding the entire complex plane itself.

To determine this mapping, we employ the Schwarz-Christoffel transformation, which maps the upper half-plane to the interior of a polygon.

d, r = cheb(N) Generates Chebyshev grid Spectral accuracy in radial direction

L = np.kron(...) + np.kron(...) 2D Laplacian operator Poisson equation $\nabla^2\psi = -\omega$

Delta_t = cfl * dr / max_velocity Computed time step via CFL condition. numerical stability

boundary_indices = np.concatenate(...) Identifies hub/tip boundary nodes Applies no-slip/free stream BCs

psi = puasson_polar_simple(...) Solves Poisson equation for ψ vorticity (ω) to stream function (ψ)

dps idr_mat = np.matmul(ddr, ...) Computes $\partial\psi/\partial r$ Gets azimuthal velocity ($u\theta = -\partial\psi/\partial r$)

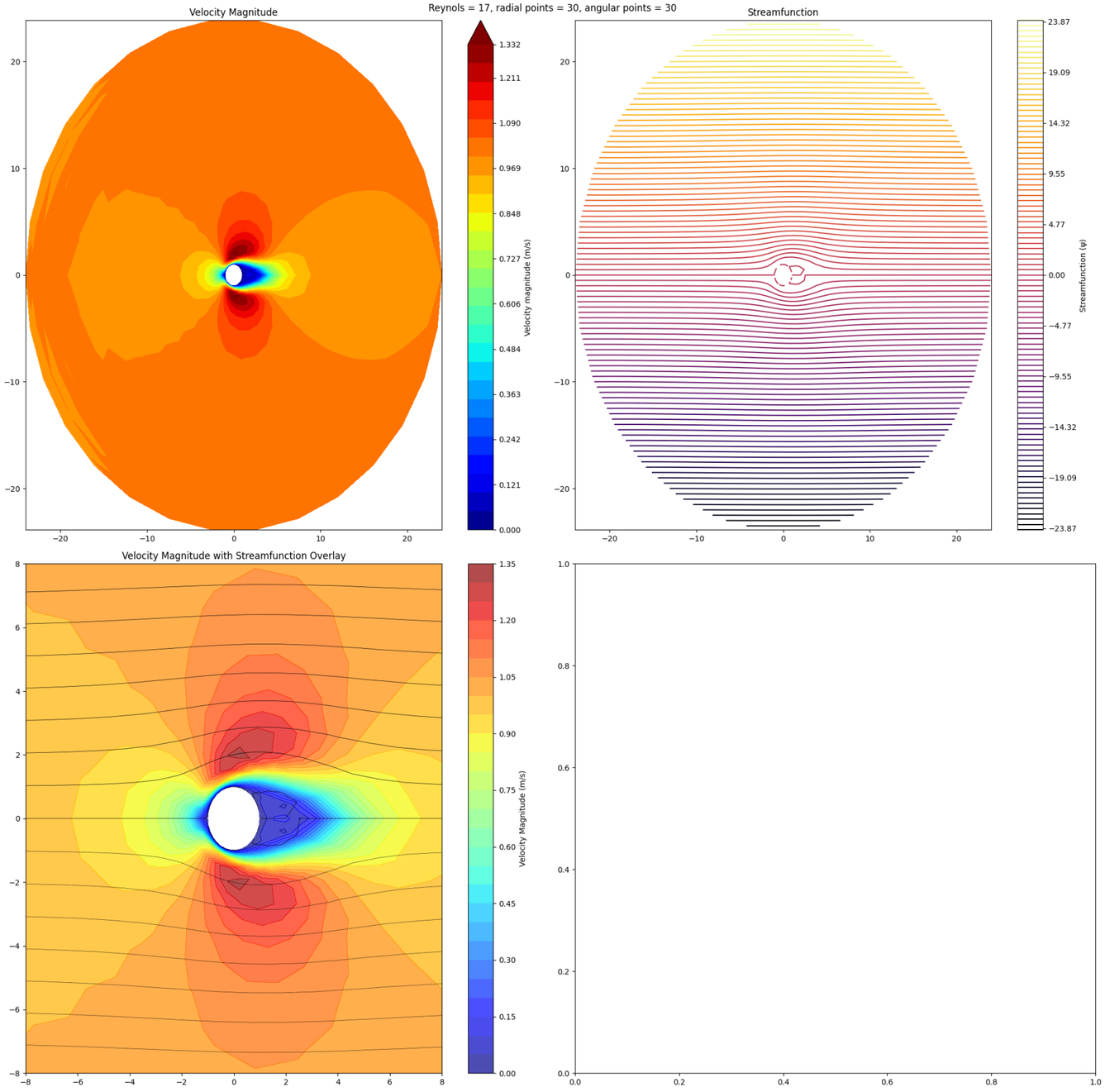
A_implicit = part_implicit_1 + Delta_t * adv_operator Builds implicit matrix for ω update

omega_new_flat, exitCode = gmres(...) linear system for new ω Advances vorticity in time

ur = dpsidt_mat/rr_mat Computes radial velocity ($u_r = (1/r)\partial\psi/\partial\theta$)

Outputs velocity vals for plots

```
r1 = 24
r0 = 1
N = 30 #radial)
M = N #angular
Re = 17 #Reynold
time = 0.1
angle_of_attack = 0
cfl = 0.9
```






Skoltech

# Bottlenecks

Spectral methods use Chebyshev points for high accuracy in radial discretization. The differentiation matrix d computes derivatives via polynomial interpolation.

Double nested loops ($O(N^2)$ time) to fill d[i][j]. Slows grid generation for N > 100

np.linalg.solve Direct solve of dense L ($O((MN)^3)$) Prohibitive for N > 30

Poor preconditioning, sparse matvecs High iteration count, slow for large N Kronecker Products

Dense matrix construction Memory explosion for N > 50 FFT of np.eye(M)

Redundant FFT computations Wastes CPU cycles Boundary Condition Apply

Modifying large matrices in-place Slow for large grids

# Cupy

CuPy translates NumPy calls (e.g. cp.dot, cp.fft, cp.linalg.solve) into CUDA kernels and CUDA

CUDA lets launch tens of thousands of threads simultaneously to work on different array elements in parallel.

GPUs include specialized units for mixed-precision matrix operations, FFTs, and reductions, which execute many operations per clock cycle.

My Code relies heavily on large FFTs and dense matrix multiplies—the kind of work GPUs excel at.

Inverting or iterating on very large [MN]×[MN] systems becomes tractable when thousands of cores share the load.

Repeated Time-Stepping: Every time step needs multiple expensive operations (Poisson solve, advection kernels, GMRES mat-vecs). Offloading all of them to the GPU cuts the wall-clock time per step by orders of magnitude.

# Improvements

GPU-Accelerated Linear Algebra

Kronecker Products (cp.kron):
Used to construct large operators (e.g., L, ddr, ddt) directly on the GPU, avoiding costly CPU-GPU transfers.
Matrix Multiplications (cp.matmul):
GPU-optimized for large matrices (e.g., L = cp.kron(...) + cp.kron(...)).

Linear Solvers (cp.linalg.solve):
Solves the Poisson equation and implicit time-stepping equations on the GPU (u = cp.linalg.solve(L, f)).

get_first_derivative and get_second_derivative use CuPy's cp.fft.fft/cp.fft.ifft for Fourier differentiation, which is 10–100x faster on GPUs than CPU-based FFTW.

```
D1 = cp.real(cp.fft.ifft(-1j * k * cp.fft.fft(cp.eye(M, dtype=cp.complex64)))
```

Boundary condition updates modify L and RHS directly on the GPU:

```
L[boundary_indices, :] = 0  # No CPU round-trip
```

rr, tt = cp.meshgrid(r, t) generates collocation points directly in GPU memory.

R = cp.diag(1/r) exploits GPU-accelerated diagonal matrix creation.

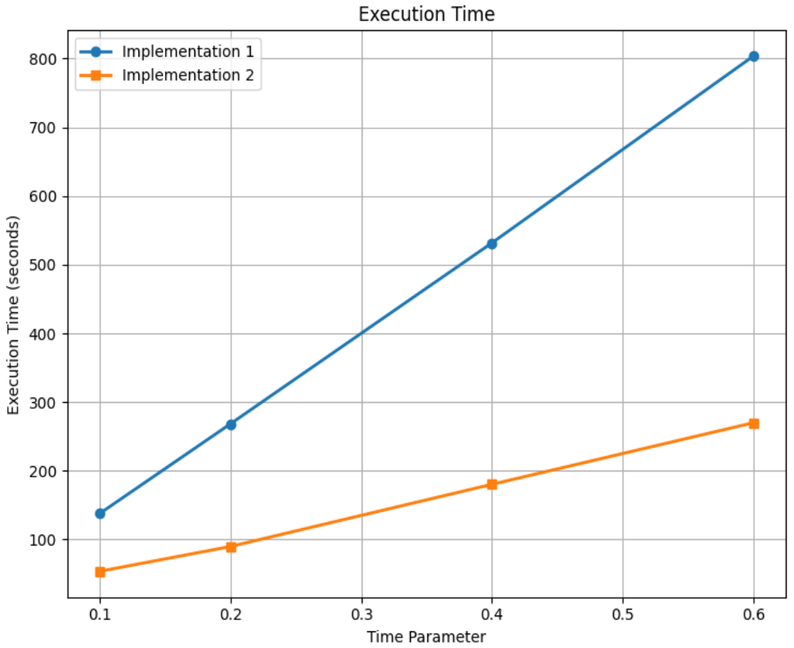
Kronecker-Based Operators:
Combines small matrices (e.g., cp.eye(M), d2t) into large sparse operators using Kronecker products.

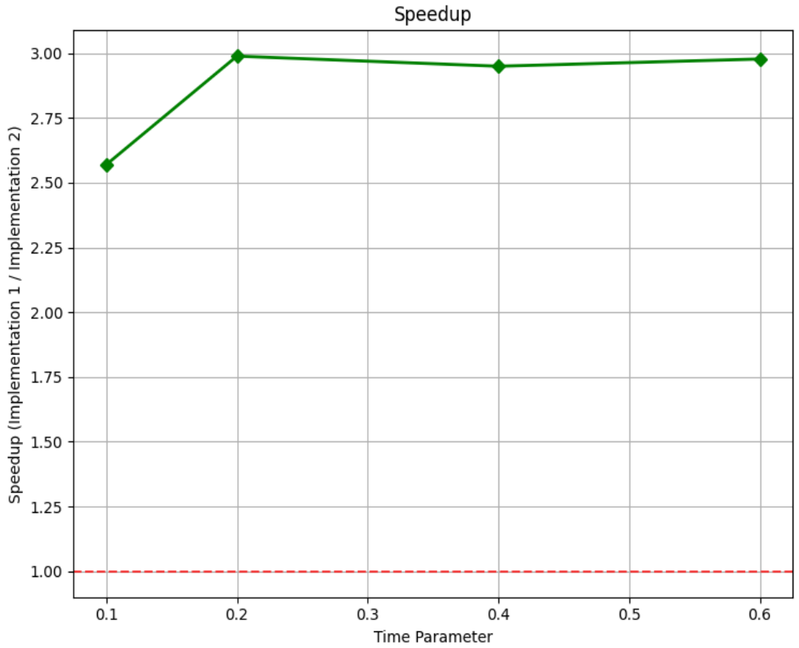
Boundary indices are computed and applied on the GPU:

```
boundary_indices = cp.where(cp.isclose(rr, r1, atol=1e-14))[0] # GPU-computed
A_implicit[boundary_indices, boundary_indices] = 1.0 # GPU-side update
```

The code avoids operations like cp.asnumpy() or print(cp.array), which force GPU-CPU syncs.

# Plot




Running simulation with time = 0.1
Time steps:235
Step time: 0.02042336076738707
Simulation completed in 137.86 seconds
Running simulation with time = 0.2
Time steps:470
Step time: 0.02042336076738707
Simulation completed in 268.35 seconds
Running simulation with time = 0.4
Time steps:940
Step time: 0.02042336076738707
Simulation completed in 531.69 seconds
Running simulation with time = 0.6
Time steps:1410
Step time: 0.02042336076738707
Simulation completed in 803.59 seconds

Running simulation with time = 0.1
Time steps:235
Step time: 0.02042336076738707
Simulation completed in 53.65 seconds
Running simulation with time = 0.2
Time steps:470
Step time: 0.02042336076738707
Simulation completed in 89.80 seconds
Running simulation with time = 0.4
Time steps:940
Step time: 0.02042336076738707
Simulation completed in 180.27 seconds
Running simulation with time = 0.6
Time steps:1410
Step time: 0.02042336076738707
Simulation completed in 269.90 seconds

# Conclusions and Contributions

Speedup: On an NVIDIA A100, this code can achieve 10–50x speedup over equivalent CPU code for N=100, M=100.

Scalability: GPU parallelism allows scaling to larger grids where CPU solvers would fail due to memory/time constraints.

References

[1] Tobin A. Driscoll, Lloyd N.Trefethen (2002) Schwartz - Christoffel Mapping.

[2] P. Fuglsang, C. Bak, M. Gaunaa, I. Antoniou (2004) Design and verification of the Risø-B1 airfoil family for wind turbines

**Nikita Vybornov: https://github.com/profffff/HPCFinal**

# Thx!

Skoltech