

HPC BENCHMARK GAME

Status Update and Language Extensions

Max Lübke¹, Adrian Nirk¹, Stefan Petri²,
Bettina Schnor¹

¹University of Potsdam, Institute of Computer Science

²Potsdam Institute for Climate Impact Research

September 17, 2026



INTRODUCTION

Follow-up to PARS 2024 and PECS 2025 edition of the HPC Benchmark Game

- **PECS 2025 (Workshop on Performance and Energy Efficiency in Concurrent and Distributed Systems) Key Findings:**
 - Compared **C, C++, Fortran, Julia** across 4 benchmarks (CA, TUG, NAS FT, SRAD).
 - **Energy ~ Runtime:** Confirmed strong correlation between execution time and energy.
 - **Compiler Impact:** Compiler selection had a larger impact than programming language choice.
 - **JIT Language Julia:** Not competitive with native AOT.
 - **Over-threading:** Be aware of performance degradation.
- **Status Update & 2026 Extensions:**
 - **Updated Compiler Versions:** Intel 2025.0.0, GCC 15.2.0, AOCC 4.2.0, Julia 1.12.2
 - **Language Extension:** Added **Python / Numba** (JIT compilation) and **Rust**.
 - **Deeper Analysis:** Compiler version comparison (GCC 14 vs. 15) and extended over-threading evaluation.
 - **(Code Optimization:** Uncovered memory bottlenecks, evaluated Julia threading (Polyester.jl).)

SELECTED REFERENCE BENCHMARKS

Initially One Reference Benchmark per Language Domain:

Lang.	Benchmark	Domain / Origin	Berkeley Dwarfs ^{⟨1⟩}
C	CA	Conway's Game of Life	Structured Grids
C++	TUG	Diffusion solver	Structured Grids & Dense Linear Algebra
Fortran	NAS FT	NPB Suite	Spectral Methods
Julia	SRAD	Rodinia Benchmark Suite	Structured Grids

→ All benchmarks were re-implemented in every target language.

^{⟨1⟩}As defined by Asanovic et al. [“The Landscape of Parallel Computing Research: A View from Berkeley”](#), 2006

RE-IMPLEMENTATION DETAILS

Methodology & Measurement:

- **Idiomatic Code:** Native re-implementation in each target language.
- **Reproducibility:** Standardized pseudo-random number generator with fixed seed.
- **Correctness:** Verified across all languages, independent of thread count.
- **Measurement:** Time & energy measured via **EMA**^a → OS timestamps and RAPL interface.

^a<https://github.com/PERFACCT/EMA>

Parallelism & Language Settings:

- **Parallelization:** OpenMP (C, C++, Fortran); Julia FLoops.
- **Data Structures:** Plain C-style arrays (C), Eigen library (C++), ...
- **Flags:** Bounds checking disabled for C++ (Eigen) and Julia.

RESEARCH QUESTIONS & SCOPE

Languages & Research Questions:

Target Languages: C, C++, Fortran, Julia, Python/Numba

1. **JIT vs. AOT:** Performance & energy of JIT compilers vs. native AOT compilation.
2. **Compiler Impact:** Impact of compiler choice on energy efficiency.
3. **Threading & HW:** Scaling and the effect of over-threading.

Out of Scope (Simplifications):

- **DRAM Power:** CPU represents $\approx 89\%$ of energy (Pereira et al.).
- **Frequency Governor:** Used default OS scaling governor.
- **Vector Instruction Set:** Fixed per run (AVX2).
- **Problem Size:** Fixed workload per benchmark.

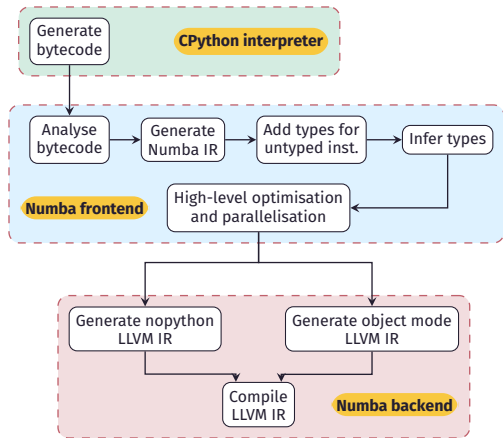
PYTHON / NUMBA

Just-in-Time Compilation & Workflow

Motivation:

- Pure Python is interpreted, and **slow**.
- **Pereira et al. (2021)**: Interpreted Python consumes **75.9× more energy** and takes **71.9× more time** than C.
- **Numba**: JIT compiler using the **LLVM** compiler infrastructure.
- Integrates well with the NumPy ecosystem.

Compilation Flow:



PYTHON / NUMBA

Compilation Modes & Parallelization

Key Features:

- **nopython Mode (@njit):**
 - Compiles Python code to machine code without CPython interpreter.
 - Mandatory to achieve AOT compilation performance.
- **Shared-Memory Parallelism:**
 - Automatically releases Global Interpreter Lock (GIL).
 - Parallel loop execution via `parallel=True` and `prange`.
 - Uses thread pool for parallel execution.

Example: Parallel Stencil Loop

```
1 from numba import njit, prange
2
3 @njit(parallel=True)
4 def simulate_loop_explicit(fro, to):
5     # Parallel loop across thread pool
6     for y in prange(1, YSIZE):
7         for x in prange(1, XSIZE):
8             to[x, y] = anneal( ... )
```

TESTBED

Hardware and Software Environment

Hardware (HPC2024 Cluster @ PIK^a):

- **CPUs:** 2× AMD EPYC 9554 “Genoa” (128 cores total) → zen4
- **Memory:** 768 GB DDR5 RAM (6 GB per core)
- **Cache:** 32 MB L3 cache shared across 8 cores

Methodology:

- Full single-node execution (128 threads)
- All measurements repeated 5 times

^aPotsdam Institute for Climate Impact Research

Compilers & Runtimes:

• Native Compilers (avx2/03):

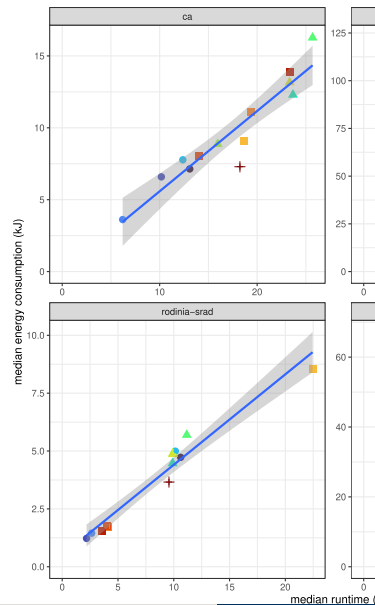
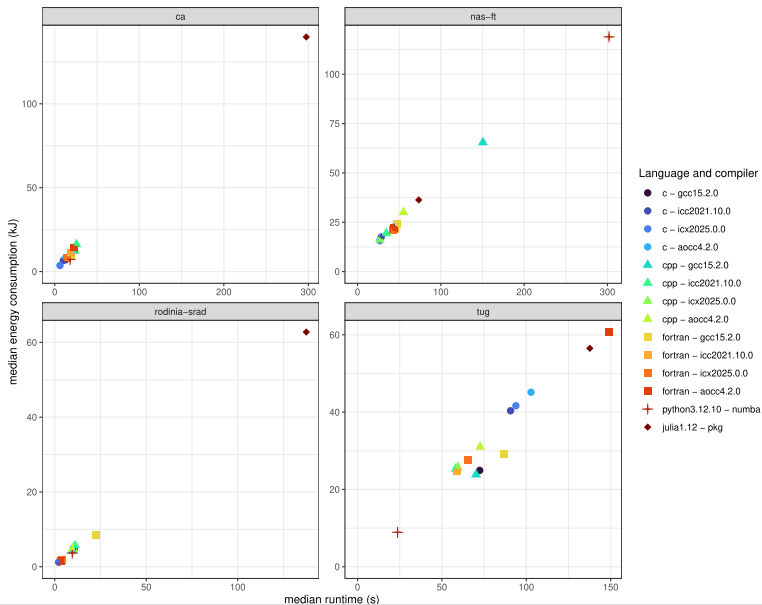
- GCC 15.2.0, AOCC 4.2.0
- Intel Classic 2023.2.0 (icc)
- Intel oneAPI 2025.0.0 (icx)

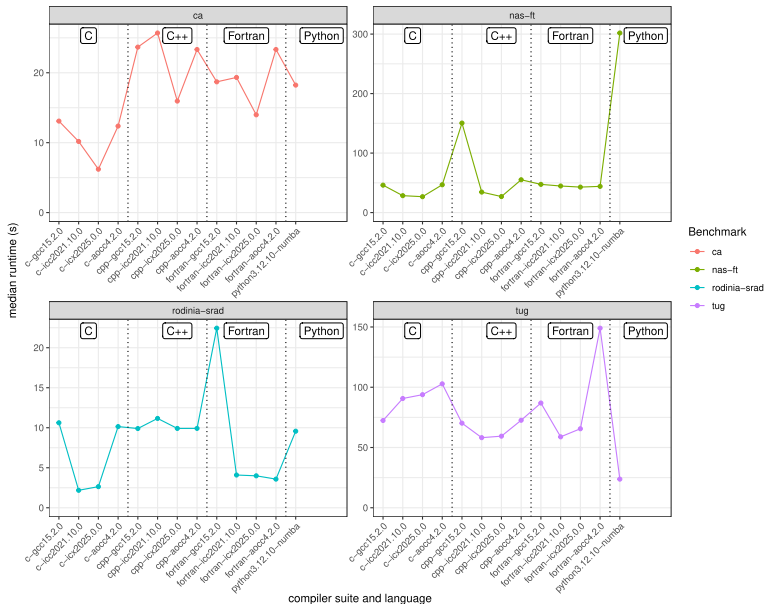
• Python / Numba:

- Python 3.12.10, Numba 0.62.1, NumPy 2.3.0

• Julia Environment:

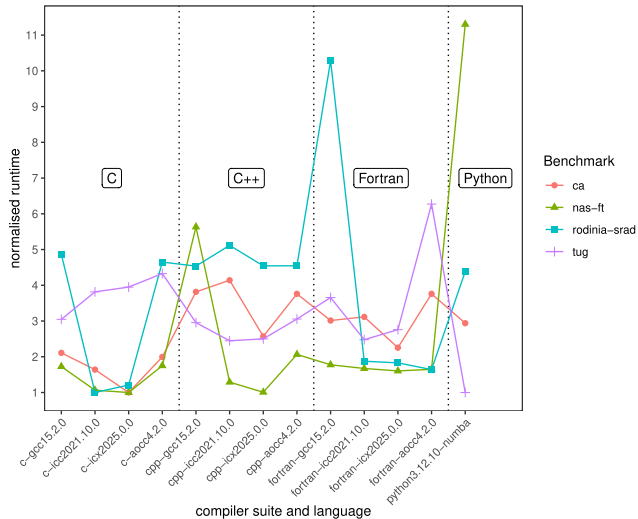
- Julia 1.12.2 (-O3 -C znver4 --check-bounds=no)





Benchmark

- ca
- nas-ft
- rodinia-srad
- tug



Normalized Runtime: Comparison of Compiler Suite and Language

GLOBAL RESULTS: 2026 CROSS-LANGUAGE EVALUATION

Geometric Mean across all Benchmarks (Normalized Runtime & Energy)

Normalized Runtime		Normalized Energy	
C / icx	1.48	C / icx	1.53
C / icc	1.61	C / icc	1.74
Fortran / ifx	2.07	Fortran / ifx	1.90
Fortran / ifort	2.22	Fortran / ifort	2.02
C++ / icx	2.33	C++ / icx	2.32
C / gcc	2.71	C / gcc	2.38
Fortran / aocc	2.83	Python / Numba	2.60
C++ / icc	2.86	Fortran / aocc	2.61
C / aocc	2.89	C / aocc	2.88
Fortran / gcc	3.07	C++ / icc	2.93
C++ / aocc	3.22	C++ / aocc	3.14
Python / Numba	3.47	C++ / gcc	3.43
C++ / gcc	4.12	Fortran / gcc	3.77
Julia / pkg	14.81	Julia / pkg	13.07

COMPARATIVE ANALYSIS: 2025 vs. 2026 RESULTS

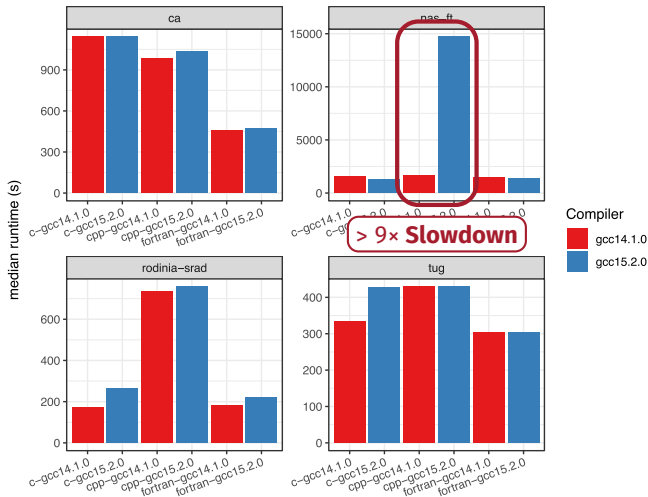
Compiler / Toolchain	2025 Norm. RT	2026 Norm. RT	Trend	Key Insight
Intel (icx, icc, ifx)	1.20 – 1.89	1.48 – 2.89	→	Dominant leader across C, Fortran, C++
AOCC (AMD / LLVM)	2.26 – 2.45	2.83 – 3.22	→	Mid-tier; trails Intel compilers on AMD Genoa
Julia	13.32 – 13.74	14.81	→	No improvement at all
Python / Numba (JIT)	–	3.47	NEW	JIT competitiveness with native code; wins TUG
GCC (GCC 14 vs. GCC 15)	2.11 – 2.86	2.71 – 4.12	↓↓	Performance regression in GCC 15 vs. 14

Takeaways:

- **Python/Numba JIT:** Able to bridge the gap between JIT and AOT performance.
- **↓↓ GCC 15 Regression:** Noticeable slowdown across C (2.21 → 2.71), Fortran (2.86 → 3.07) and C++ (2.11 → 4.12).
- **Julia:** Still far from AOT performance.

GCC 14 vs. GCC 15

Evaluating Performance Degradation Across Compiler Versions

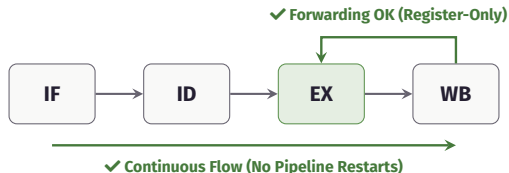


GCC 14 vs. GCC 15: FT BENCHMARK CASE STUDY

Store-to-Load Forwarding Stalls

Zen 4 Hardware Counters (perf / LIKWID)

Metric	GCC 14	GCC 15	Delta
Retired FLOP	468.1 B	468.1 B	Identical
Runtime	88.37 s	742.56 s	8.4× slower
CPI	7.25	54.65	7.5× higher
Mem. Stalls	9.4%	67.3%	Load stalls
Pipeline Restarts	10.3 M	54.0 M	5.2× resyncs



Code-Gen: Register vs. Stack-Packing

- GCC 14 (Scalar, No Stack Spills):

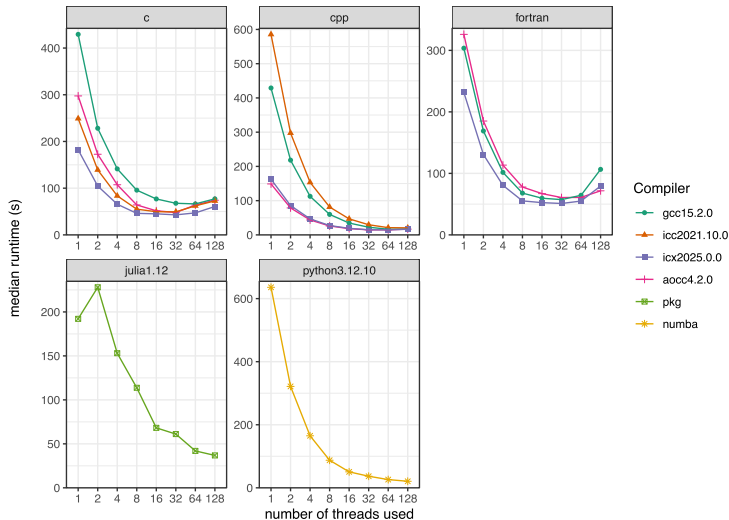
```
1 ; GCC 14: Register-Only Scalar Math
2 vmovsd xmm4, [r14] ; Load x11.real
3 vaddsd xmm7, xmm1, xmm4 ; xmm7 = x21 + x11
4 vmovsd [rbx], xmm7 ; Store (0 stack spills)
```

- GCC 15 (Vectorized Stack-Packing):

```
1 ; GCC 15: Vectorized Stack-Packing
2 vmovsd [rsp+480], xmm7 ; Store real (8B)
3 vmovsd [rsp+488], xmm0 ; Store imag (8B)
4 vmovapd xmm0, [rsp+480] ; 16B Load -> STLF FAIL!
```

→ 16-byte load overlapping two 8-byte stores triggers severe STLF pipeline stalls.

OVERTHREADING: PERFORMANCE SCALING FOR TUG



Takeaways:

- cpp/aocc, 128 threads: **16.7** seconds
- 32 threads: **14.4** seconds
→ **≈ 14% less runtime**
and 22% less energy
- **Python / Numba** and **Julia** scale up to **128 threads**.
- Julia exhibits erratic scaling behavior as the thread count increases.

RUST: IMPLEMENTATION & EVALUATION SETUP

Student Projects, Multi-Threading with Rayon & Methodology

Context:

- Student implementations from summer terms projects on “Secure and Efficient Programming with Rust”
- **Student Experience Levels:**
 - **Beginner Student:** Cellular Automaton (CA), Rodinia SRAD.
 - **Two Advanced Students:** TUG.
- **Goal:** Port benchmarks directly to Rust.

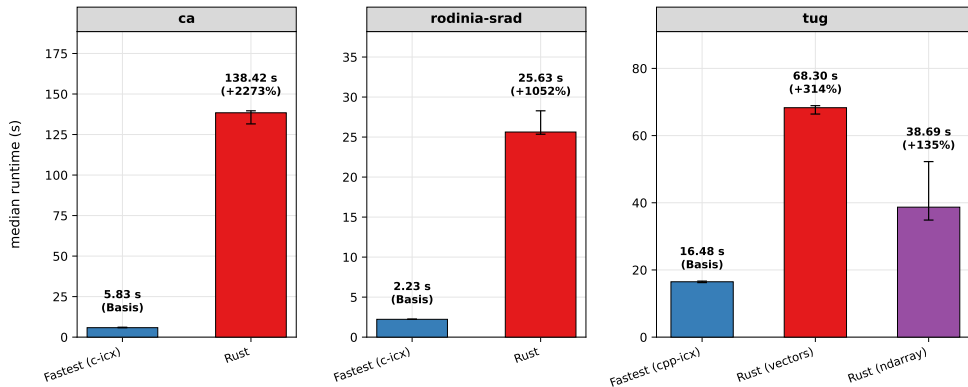
Rules & Toolchain:

- **Multi-Threading:**
 - Parallelized via **Rayon**.
 - Evaluated at **128 threads** on PIK cluster (AMD Zen 4).
- **Library Policy:**
 - External crates allowed to ease the life of the student (e.g. ndarray).
 - Algorithmic behavior of baseline must remain unchanged.
- **Compiler:** `rustc 1.98` with `x86_64_v3 LLVM` backend optimizations.

RUST VS. FASTEST BASELINE

Absolute Runtime Across Benchmarks (128 Threads)

Rust vs. Fastest Baseline: Absolute Runtime (128 Threads)



Rust vs. Fastest Baseline: Absolute Runtime (128 Threads)

FIRST TAKEAWAYS: RUST IN HPC

Performance & Productivity

- Straightforward ports close to C/C++ suffer slowdowns (CA: 23.7×, SRAD: 11.5×).
- Optimized solution achieved -25% runtime reduction in TUG vs. fastest C++ baseline.
- Requires extensive `unsafe` code (raw pointers, unchecked indexing)
- **Student Conclusion:**
“It is preferable to apply low-level optimizations in C/C++ than in Rust.”

THANK YOU FOR YOUR ATTENTION!

Questions & Discussion

Takeaways:

- **Compiler Choice > Language Choice.**
- Users should be aware of possible **Over-threading.**
- **JIT Competitiveness of Numba:** 3.47 Normalized Runtime, 2.6 Normalized Energy.
- **GCC 15 Regression.**
- HPCBG results give **guidelines** for application developers to save runtime and energy.

Project Page



<https://hpcbg.org/>

TUG CASE STUDY: OPTIMIZATIONS UNCOVERED BY NUMBA

Numba Re-Implementation as Trigger:

- Python/Numba was **faster** than original C++ TUG → prompted bottleneck analysis.
- OpenMP fork-join overhead suspected, but disproved via perf (only $n - 1$ clone3 syscalls).

Original C++ (Inside Loop)

```
1 for (iterations) {  
2   RowMajMat<T> aX = grid.getAlphaX(); // copy  
3  
4   for (x-sweep) { /* compute */  
5  
6     aX.transposeInPlace();  
7  
8     for (y-sweep) { /* compute */  
9  
10  } // end loop over iterations
```

3.55B cache misses (31.8 %)

Optimized C++ (Hoisted)

```
1 const RowMajMat<T> &aX = grid.getAlphaX();  
2 const RowMajMat<T> aX_t = aX.transpose();  
3  
4 for (iterations) {  
5   for (x-sweep) { /* compute */ }  
6   for (y-sweep) { /* compute */ }  
7 } // end loop over iterations
```

1.14B cache misses (22.2 %)

JULIA

Threading Optimization: FLoops.jl vs. Polyester.jl

Threading Benchmark (CA Stencil):

Threading Strategy	Runtime
Polyester.jl (@batch)	17.77 μ s
Base threads (static)	362.67 μ s
FLoops.jl (@floop)	573.87 μ s
Base threads (dynamic)	30.43 ms

- **Polyester.jl**: Low-overhead loop parallelization (~ 32 $\times$ faster than FLoops).
- **ThreadPinning.jl**: Pins threads to physical CPU cores.

Example: Parallel Loop with @batch

```
1 function simulate!(ca::CellularAutomaton)
2     # Polyester.jl: @batch replaces @floop
3     @batch for y = 2:size(ca.from, 2)-1
4         for x = 2:size(ca.from, 1)-1
5             ca.to[x, y] = step(ca.from, y, x)
6         end
7     end
8 end
```

OVERTHREADING: OPTIMAL THREAD COUNTS FOR TUG

Optimal Thread Count per Language & Toolchain

Program / Toolchain	Opt. Threads	Runtime (s)
c-aocc4.2.0	32	47.99
c-gcc15.2.0	64	66.32
c-icc2021.10.0	32	49.32
c-icx2025.0.0	32	42.87
cpp-aocc4.2.0	32	14.64
cpp-gcc15.2.0	64	16.55
cpp-icc2021.10.0	128	20.56
cpp-icx2025.0.0	64	14.65
fortran-aocc4.2.0	32	60.74
fortran-gcc15.2.0	32	57.37
fortran-icx2025.0.0	32	51.07
julia1.12-pkg	128	36.84
python3.12-numba	128	20.73

Key Findings:

- **Native Compilers:**

- Peak at **32 threads** (matching 1 NUMA domain) for most toolchains.
- Overthreading beyond 32/64 increases runtime/energy.

- **JIT Runtimes:**

- **Python / Numba** and **Julia** scale up to **128 threads**.
- Julia exhibits erratic scaling behavior as the thread count increases.

OPTIMIZATION APPROACHES ACROSS BENCHMARKS

Systematic Performance Refinements

- **TUG**

- Hoisting matrix acquisition and transpose out of the iteration loop → Reduces 3.55B to 1.14B cache misses.

- **Cellular Automaton (CA):**

- Hoisting OpenMP parallel region out of iteration loop.
- Fixing memory race conditions in C boundary handling functions.

- **NAS Parallel Benchmarks FT:**

- Replacing slow copy loops with efficient NumPy array slicing in Python.
- Enabling JIT compilation caching (cache=True) → saving ~10 s startup overhead.

- **Rodinia SRAD:**

- Merging consecutive OpenMP parallel loops.

- **Julia Threading:**

- Migration from `FLoops.jl` to `Polyester.jl`.

OPTIMIZATION IMPACT

Quantifying Performance Gains

