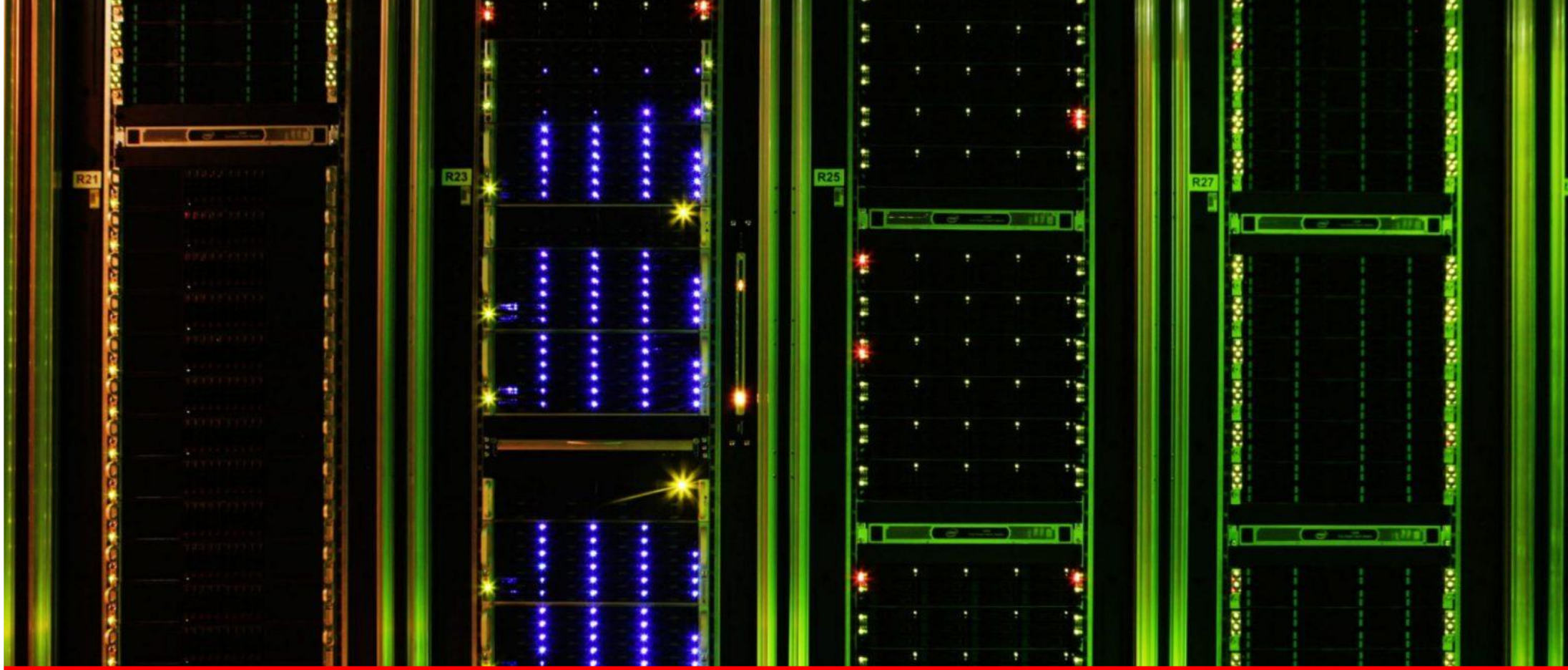


Optimizing the Vlasov Static Kernel in GENE-X

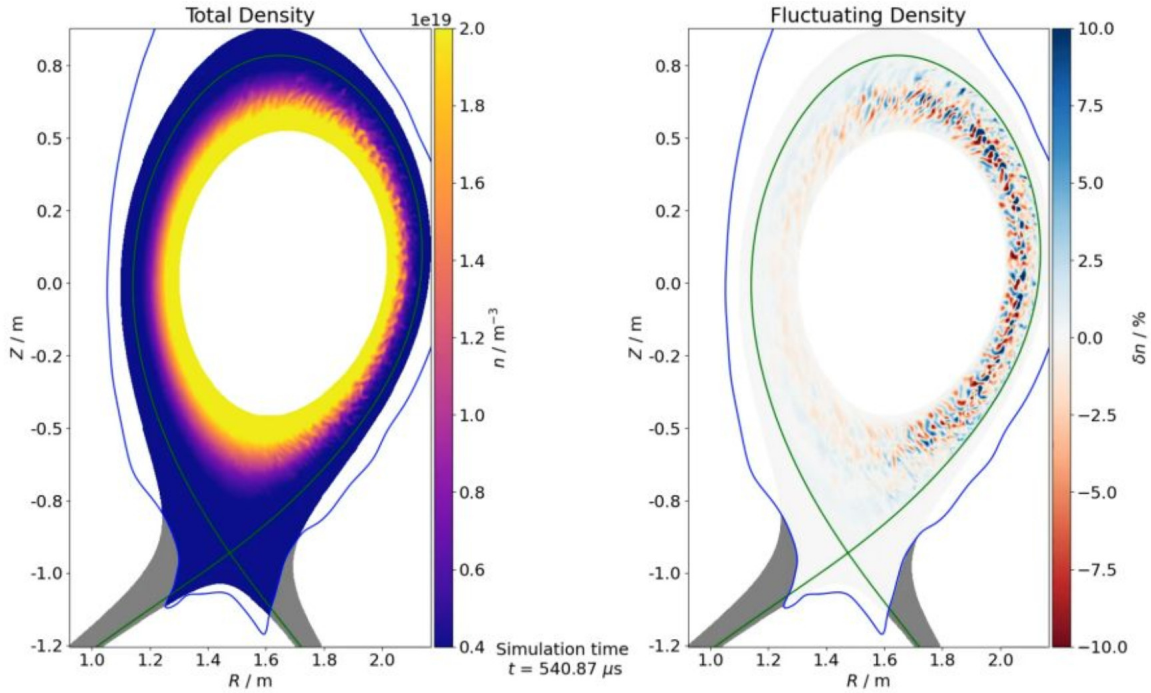
Piyush Panchal (SCITAS, EPFL)

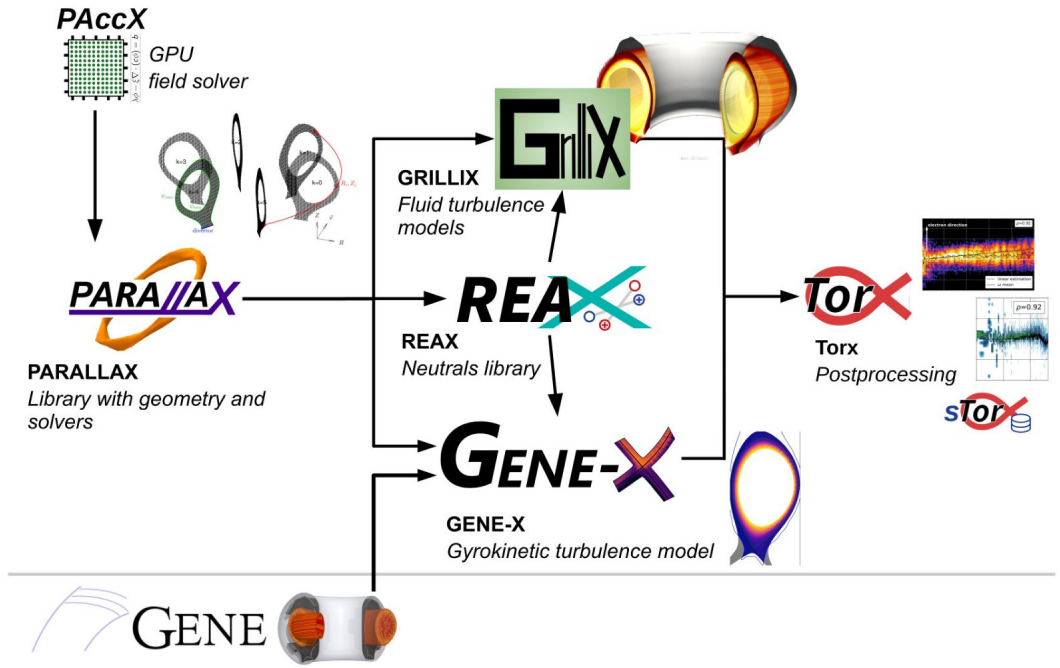
EUROfusion HPC ACH Meeting
Tue. 27 October 2026, Garching

GENE-X Overview

Gyrokinetic turbulence simulations of the edge and SOL region with X-points

- Full-f (5D) evolution via gyrokinetic Vlasov–Maxwell system, Eulerian grid
- Fortran 2008, MPI + OpenMP; C++ layer with GPU support via OpenMP/OpenACC offload
- Strong scaling with 93% efficiency up to 512 nodes (~20K cores), good weak scaling





Source: P. Ulbl et al. (2024), “Towards first-principles simulations of the L- to H-mode transition with the global gyrokinetic turbulence code GENE-X”, TSVV4/1 Workshop 2024.

The Vlasov Static Kernel

The gyrokinetic Vlasov–Maxwell system evolves the distribution function *f* as

$$\frac{\partial f}{\partial t} = \boxed{\left(\frac{\partial f}{\partial t}\right)^*} + \frac{q}{mc}\frac{\partial A_\parallel}{\partial t}\frac{\partial f}{\partial v_\parallel} + C(f)$$

static: this kernel **dynamic** **collisions**

op_rhs_vlasov_eq_static computes (∂f/∂t)*

- At every point (i, k, l, m, n) of the 5D grid
- **Parallel streaming**: derivative along the field line, from planes k±1, k±2
- **Perpendicular drifts**: Poisson bracket in the RZ plane
- **Mirror force**: derivative in v∥
- **Hyperdiffusion** for numerical stability

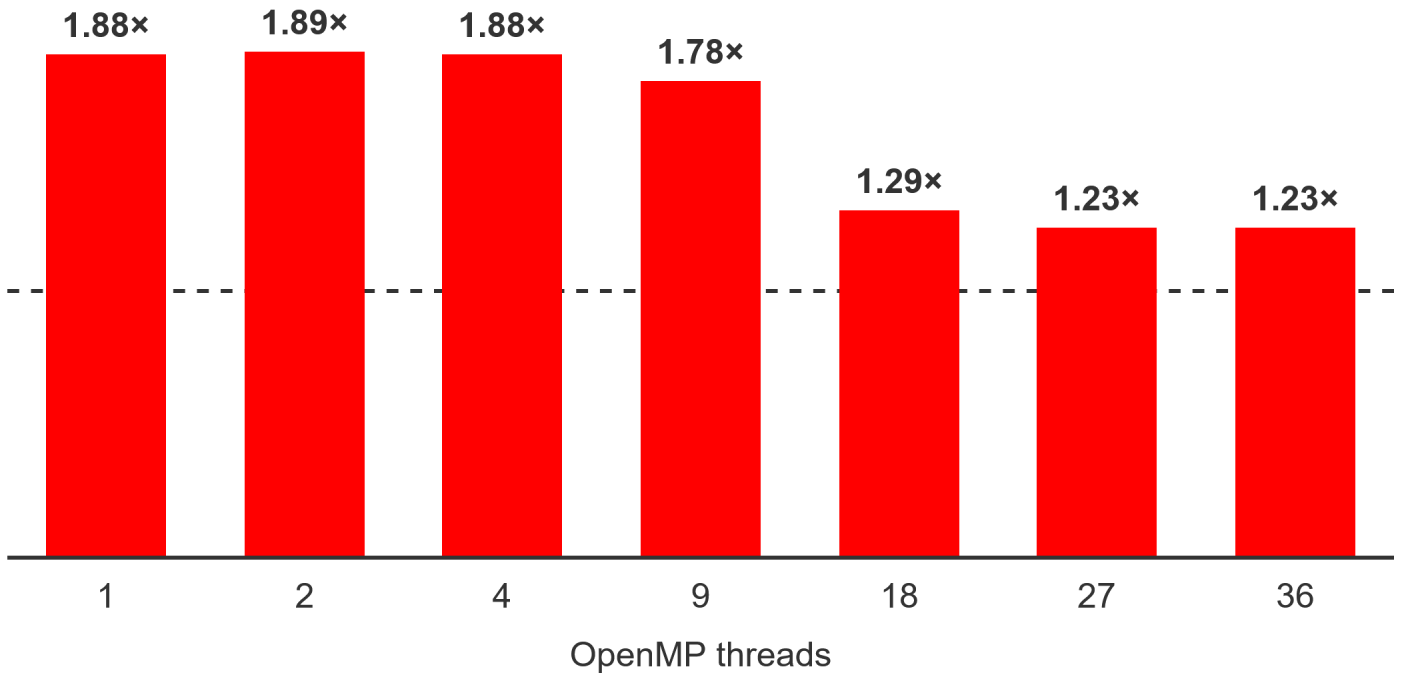
"Static": uses the fields at the current time only. The ∂A∥/∂t term and collisions are separate operators.

Two implementations: Fortran (CPU) and C++ (OpenACC, OpenMP offload, OpenMP on CPU)

Michels et al., Phys. Plasmas 29, 032307 (2022); Frei et al., Comput. Phys. Commun. (2025) 109817.

Motivation: C++ was faster on CPU

- R2: 267,288 RZ × 12 φ × 10 v∥ × 4 μ × 2 species = 256.6 M points
- 1 MPI rank on one socket of Jed (Intel Xeon 8360Y, 36 cores)



Time per call, R2

Threads	Fortran	C++
1	43.98 s	23.42 s
18	2.58 s	1.99 s
36	1.57 s	1.27 s

- 1–9 threads: C++ **1.8–1.9×** faster
- 18–36 threads: C++ **1.2–1.3×** faster

The C++ CPU path beat the Fortran kernel at every thread count

Speed-up = Fortran time / C++ time; dashed = equal. benchmark-operators, ifx 2025.3, original code at commit ac265f43.

C++ vs Fortran Across Problem Sizes

- Speed-up of C++ over Fortran (Fortran time / C++ time)
- Every case: 10 v∥ × 4 μ × 2 species

Case	RZ × φ	Points	1 thr	9 thr	18 thr	36 thr
R0	17,688 × 12	17.0 M	1.96×	1.75×	1.32×	1.48×
R1	68,120 × 12	65.4 M	1.94×	1.75×	1.28×	1.24×
R2	267,288 × 12	256.6 M	1.88×	1.78×	1.29×	1.23×
R3	1,058,840 × 12	1,016.5 M	2.00×	1.89×	1.48×	1.21×
P1	17,688 × 24	34.0 M	1.96×	1.75×	1.25×	1.31×
P2	17,688 × 48	67.9 M	1.90×	1.75×	1.21×	1.25×
P3	17,688 × 96	135.8 M	1.87×	1.75×	1.25×	1.20×
C1	267,288 × 48	1,026.4 M	1.87×	1.75×	1.32×	1.21×

On one socket, C++ was faster than Fortran at every problem size and thread count. Goal: make the Fortran CPU kernel at least as fast.

Method

Setup

- **Test case R2**: 267,288 RZ points × 12 φ × 10 v∥ × 4 μ × 2 species = **256.6 M** points
- **Jed**: 2 × Xeon Platinum 8360Y (Ice Lake), 36 cores per socket
- One MPI rank, pure OpenMP
- **36 threads** (one socket, primary) and **18 threads**
- Compiler: ifx 2025

Measurement

- **Metric**: kernel time per call from benchmark-operators
- Counters: VTune hpc-performance and memory-access
- Every change checked at both thread counts

Group the CSR projections by plane

- Order the 20 field-line interpolations by map (planes k−2, k−1, k+1, k+2) instead of by field
- Makes the compiler **vectorize the dot products**

`r_s = map_s(k)%i(i):map_s(k)%i(i + 1) - 1  (CSR row i of map_s(k), s ∈ {mm, m, p, pp})`

Before: grouped by field

```
f_mm   = dot_product(map_mm(k)%val(r_mm), f_in(map_mm(k)%j(r_mm), k - 2, l, m, n))
f_m    = dot_product(map_m(k)%val(r_m),  f_in(map_m(k)%j(r_m), k - 1, l, m, n))
f_p    = dot_product(map_p(k)%val(r_p),  f_in(map_p(k)%j(r_p), k + 1, l, m, n))
f_pp   = dot_product(map_pp(k)%val(r_pp), f_in(map_pp(k)%j(r_pp), k + 2, l, m, n))
phi_mm = dot_product(map_mm(k)%val(r_mm), phi_in(map_mm(k)%j(r_mm), k - 2))
...
```

After: grouped by map

```
f_mm     = dot_product(map_mm(k)%val(r_mm), f_in(map_mm(k)%j(r_mm), k - 2, l, m, n))
phi_mm   = dot_product(map_mm(k)%val(r_mm), phi_in(map_mm(k)%j(r_mm), k - 2))
A_par_mm = dot_product(map_mm(k)%val(r_mm), A_par_in(map_mm(k)%j(r_mm), k - 2))
B_par_mm = dot_product(map_mm(k)%val(r_mm), B_par_in(map_mm(k)%j(r_mm), k - 2))
H2_mm    = dot_product(map_mm(k)%val(r_mm), this%H2(map_mm(k)%j(r_mm), k - 2))
f_m      = dot_product(map_m(k)%val(r_m),  f_in(map_m(k)%j(r_m), k - 1, l, m, n))
...
```

36 threads · this step −28.1%



18 threads · this step −26.5%



Precompute field derivatives in 2D

Before: (i, k)-only derivatives inside the 5D loop

```
do (i, k, l, m, n)   ! 5D loop
  ...
  ! phi_mm, phi_m, phi_p, phi_pp depend only on (i, k)
  dphidy = c_mm(i,k)*phi_mm + c_m(i,k)*phi_m + c_o(i,k)*phi_in(i,k) + c_p(i,k)*phi_p + c_pp(i,k)*phi_pp
  ! same for dphidx, dphidz and for A_par, B_par, H2
  grad_H_y = charges(n) * dphidy + masses(n) * dH2dy + ...
  ...
enddo
```

After: computed once per (i, k), read in the 5D loop

```
do (i, k)   ! new 2D loop
  ! dphidy, dphidx, dphidz of phi, A_par, B_par, H2: same code as above
  precomp_xy(1, i, k) = dphidy;  precomp_xy(4, i, k) = dH2dy   ! all 12 stored in precomp_xy(1:12, i, k)
enddo
do (i, k, l, m, n)   ! 5D loop
  ...
  grad_H_y = charges(n) * precomp_xy(1, i, k) + masses(n) * precomp_xy(4, i, k) + ...
  ...
enddo
```

36 threads · this step −13.4%



18 threads · this step −9.9%



Consume each CSR gather at once

Use each interpolated plane value right after its dot_product,
instead of computing all four first. Same arithmetic, same order.

Before

```
vals(1) = dot_product(map_mm(k)%val(...), f_in(...))   ! f_mm
vals(2) = dot_product(map_m(k)%val(...), f_in(...))    ! f_m
...
hyp_y =        -this%prefac_hyp_y_mm(i, k) * vals(1)
dfdy  =         this%prefac_cd_y_mm(i, k)  * vals(1)
hyp_y = hyp_y - this%prefac_hyp_y_m(i, k)  * vals(2)
```

After

```
vals(1) = dot_product(map_mm(k)%val(...), f_in(...))   ! f_mm
hyp_y =        -this%prefac_hyp_y_mm(i, k) * vals(1)
dfdy  =         this%prefac_cd_y_mm(i, k)  * vals(1)

vals(2) = dot_product(map_m(k)%val(...), f_in(...))    ! f_m
hyp_y = hyp_y - this%prefac_hyp_y_m(i, k)  * vals(2)
dfdy  = dfdy  + this%prefac_cd_y_m(i, k)   * vals(2)
...
```

36 threads · this step −3.9%



18 threads · this step +0.3%



One running accumulator, one store

Add each term to accf where it is produced; store f_out once.
bps moved to the top of the body; one divide for all advection terms.

Before (inside the 5D loop)

```
f_out(i, k, l, m, n) = (bs_adv + bx_adv + vp_adv) / bps &
                     + hyp_xz + hyp_vp + hyp_y + dif_xz
f_out(i, k, l, m, n) = f_out(i, k, l, m, n) * is_compute(i, k)
```

After (inside the 5D loop)

```
bps = absB(i, k) &
    + this%prefac_bps(n) * vp(l) * curl_normb_y(i, k)
accf  =        -this%prefac_hyp_y_mm(i, k) * vals(1)
...
accf = accf + dif_xz
accf = accf + hyp_xz
...
adv = adv + bs_adv
accf = accf + (adv + vp_adv) / bps
f_out(i, k, l, m, n) = accf * is_compute(i, k)
```

Stores −40%

2.33e11 → 1.40e11, 36 threads (VTune)

Spills 56 → 33

Reloads 456 → 404 (ifx opt-report)

36 threads · this step −2.4%



18 threads · this step −3.3%



Block i, Split into Two Passes

- Loop over blocks of i (BLOCK_I = 256); pass 1: unit-stride work under !$omp simd, pass 2: gathers
- Packed DP FP: **20.0% → 34.8%** at 36 threads

Before

```
ntotal = nn * nm * nl * nk * ni
!$omp do schedule(static)
do itotal = 1, ntotal
  ...   ! decode i, k, l, m, n
  bps = absB(i, k) + ...
  ...   ! unit-stride work and gathers mixed
  accf = accf + (adv + vp_adv) / bps
  f_out(i, k, l, m, n) = accf * is_compute(i, k)
enddo
```

After

```
ntotal = nn * nm * nl * nk * nblk
!$omp do schedule(static)
do itotal = 1, ntotal
  ...   ! decode ib, k, l, m, n
  !$omp simd private(i, dfdvp, grad_H_x, ..., vp_adv)
  do it = 0, nb - 1   ! pass 1: unit stride
    bps_b(it) = absB(i, k) + ...
    vpadv_b(it) = vp_adv
  enddo
  do it = 0, nb - 1   ! pass 2: gathers
    accf = accf + (adv + vpadv_b(it)) / bps_b(it)
    f_out(i, k, l, m, n) = accf * is_compute(i, k)
  enddo
enddo
```

36 threads · this step −3.6%



18 threads · this step −12.9%



Transpose precomp_xy

Store the 12 precomputed (i, k) derivatives component-last, so pass 1 reads each one with unit stride instead of stride 12.

Before

```
allocate(this%precomp_xy(12, lb_stripped(1):ub_stripped(1), &
                            lb_stripped(2):ub_stripped(2)))
this%precomp_xy(1:4, i, k) = acc
... + this%charges(n) * this%precomp_xy(5, i, k) &
```

After

```
allocate(this%precomp_xy(lb_stripped(1):ub_stripped(1), &
                         lb_stripped(2):ub_stripped(2), 12))
this%precomp_xy(i, k, 1) = acc(1)
...
this%precomp_xy(i, k, 4) = acc(4)
... + this%charges(n) * this%precomp_xy(i, k, 5) &
```

Gathers in pass 1: 12 → 0

ifx -qopt-report=5, pass-1 simd loop

36 threads · this step −9.5%



18 threads · this step −5.4%



Make l the inner serial loop

- Remove l (v∥) from the collapsed OpenMP index; loop over l inside each i block
- Memory traffic: **1162.5 → 261.2 GB (−78%)** at 36 threads

Before

```
ntotal = nn * nm * nl * nk * nblk
!$omp do
do itotal = 1, ntotal   ! decode ib, k, l, m, n
  do it = 0, nb - 1      ! pass 1
    dfdvp = ... f_in(i, k, l-2:l+2, m, n) ...
    hyp_vp = ... f_in(i, k, l-2:l+2, m, n) ...
  enddo
  ...                    ! pass 2
enddo
! l and l+1 are nk * nblk iterations apart:
! planes l±1, l±2 are loaded again for each l
```

After

```
ntotal = nn * nm * nk * nblk
!$omp do
do itotal = 1, ntotal   ! decode ib, k, m, n
  do l = lb_stripped(3), ub_stripped(3)
    do it = 0, nb - 1    ! pass 1
      dfdvp = ... f_in(i, k, l-2:l+2, m, n) ...
      hyp_vp = ... f_in(i, k, l-2:l+2, m, n) ...
    enddo
    ...                  ! pass 2
  enddo
enddo
! consecutive l in one iteration:
! planes l±1, l±2 reused from the previous l
```

36 threads · this step −36.5%



493.5 ms · 3.18×

1274.6 ms

18 threads · this step −34.8%



838.0 ms · 3.08×

1992.7 ms

Hoist grad_H out of the l loop

grad_H_x/y/z do not depend on l: compute them once per strip in their own simd loop, and read them back from strip arrays.

After

```
!$omp simd private(i, grad_H_x, grad_H_y, grad_H_z)
do it = 0, nb - 1
    i = i0 + it
    grad_H_x = this%temp_scalings(n) * mu(m) * dabsBdx(i, k) &
             + this%charges(n) * this%precomp_xy(i, k, 5) &
             + ...
    gHx_b(it) = grad_H_x
    ...
enddo
do l = lb_stripped(3), ub_stripped(3)
    !$omp simd private(i, dfdvp, ...)
    do it = 0, nb - 1
        grad_H_x = gHx_b(it)
        ...
```

DP FLOPs 2294 → 2142 G

−6.6%, 36 threads, whole run (VTune DP GFLOPS × elapsed)





Cut pass 2 at the last gather

Pass 2a keeps only work that needs an indirect index (CSR, stencils, Arakawa).
Pass 2b is unit stride and vectorizes. BLOCK_I 256 → 128 in the same build.

After

```
do it = 0, nb - 1              ! pass 2a: gathers, scalar
    ...
    pb0_b(it) = this%prefac_arakawa * arakawa_stencil(zeta, psi)
    accf_b(it) = accf
    dfdx_b(it) = dfdx
    ...
enddo
!$omp simd private(i, pb, adv, bs_adv, accf, ...)
do it = 0, nb - 1              ! pass 2b: unit stride
    pb = pb0_b(it)
    ...
    accf = accf_b(it) + (adv + vpadv_b(it)) / bps_b(it)
    f_out(i, k, l, m, n) = accf * is_compute(i, k)
enddo
```

Packed DP FP 32.7% → 49.6%

36 threads (VTune hpc-performance)

36 threads · this step −15.4%



18 threads · this step −14.8%



Explicit-Shape Field Arrays

- Pointers: each field has its own runtime bounds, so its index is recomputed per array
- Explicit-shape dummies with one shared set of bounds: **one index serves phi, A_par, B_par, h2**

Before

```
real(kind=GP), contiguous, pointer, dimension(:,:) :: phi_in, &
                                        A_par_in, B_par_in
phi_in ⇒ da_phi_in%get_readonly_pointer()
...
+ this%masses(n) * this%H2(ngb_m, k)
```

After

```
module subroutine apply_sg_cpu(this, lb1, ub1, lb2, ub2, &
        da_f_in, phi_in, A_par_in, B_par_in, h2, da_f_out)
real(kind=GP), dimension(lb1:ub1, lb2:ub2), intent(in) :: &
    phi_in, A_par_in, B_par_in
real(kind=GP), dimension(lb1:ub1, lb2:ub2) :: h2
...
+ this%masses(n) * h2(ngb_m, k)
```

Kernel instructions −14.3%

3.73e12 → 3.20e12, 36 threads (VTune)

Affected region

```
! inside the 5D loop
do oz = -1, 1; do ox = -1, 1
  ngb_m = neighbors(ox, oz, i, k)
  zeta(ox, oz) = &
      ... * phi_in  (ngb_m, k) &
    + ... * B_par_in(ngb_m, k) &
    - ... * A_par_in(ngb_m, k) &
    + ... * h2      (ngb_m, k)
  psi(ox, oz) = f_in(ngb_m, k, l, m, n)
enddo; enddo
```

36 threads · this step −9.0%

Fortran		**368.1 ms** · 4.26×
C++		1274.6 ms

18 threads · this step −8.5%

Fortran		**622.6 ms** · 4.14×
C++		1992.7 ms

Gather the zeta stencil once per strip

Only the A_par term of zeta depends on l: gather the 3×3 neighbourhood once per strip, point-contiguous. Step 1: pass 2a gathers only, prefactors move to pass 2b.

Before: in pass 2a, for every l

```
zeta(ox, oz) = &
    + this%charges(n)            * phi_in(ngb_m, k) &
    + this%prefac_Bpar(n) * mu(m) * B_par_in(ngb_m, k) &
    - this%prefac_flutter_xyz(n) * vp(l) * A_par_in(ngb_m, k) &
    + this%masses(n)             * h2(ngb_m, k)
```

After: above the l loop, then in pass 2a

```
do it = 0, nb - 1;  do oz = -1, 1;  do ox = -1, 1
    zeta0_b(ox, oz, it) = &
        + this%charges(n)            * phi_in(ngb_m, k) &
        + ...
    apar_b(ox, oz, it) = A_par_in(ngb_m, k)
...
zeta(ox, oz) = zeta0_b(ox, oz, it) &
    - this%prefac_flutter_xyz(n) * vp(l) * apar_b(ox, oz, it)
```

DP flops/point 270 → 226

−16%, 36 threads (VTune DP GFLOPS × time / points)

36 threads · this step −9.2%



18 threads · this step −10.0%



Keep the zeta hoist loop scalar

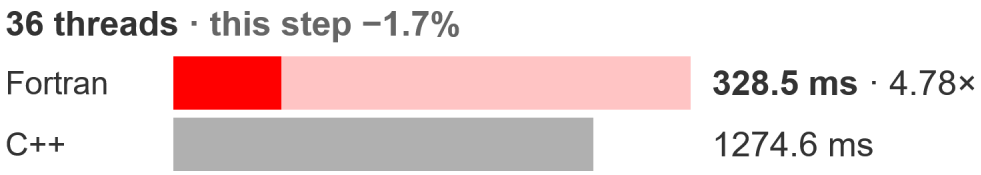
!DIR$ NOUNROLL on the hoist's 3×3 loops: the compiler stops emitting vector gathers there and uses scalar loads.

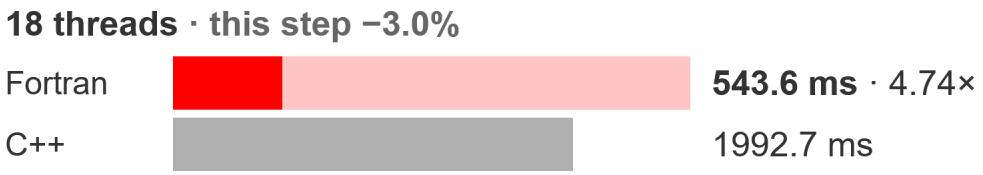
After

```
do it = 0, nb - 1
    i = i0 + it
    !DIR$ NOUNROLL
    do oz = -1, 1
    !DIR$ NOUNROLL
    do ox = -1, 1
        ngb_m = neighbors(ox, oz, i, k)
        zeta0_b(ox, oz, it) = ...
        apar_b(ox, oz, it) = A_par_in(ngb_m, k)
    enddo
    enddo
enddo
```

Gathers in the kernel routine: 20 → 16

ifx assembly of the hoist loop





From 1569 ms to 329 ms

Step (cumulative)	36 threads, ms	18 threads, ms
Original Fortran	1569.3	2577.4
1 · Group the CSR projections by plane	1128.6	1895.2
2 · Precompute field derivatives in 2D	961.8	1660.8
3 · Consume each CSR gather at once	913.9	1612.2
4 · One running accumulator, one store	891.9	1559.7
5 · Block i, split into two passes	859.6	1357.8
6 · Transpose precomp_xy	777.8	1284.9
7 · Make l the inner serial loop	493.5	838.0
8 · Hoist grad_H out of the l loop	478.2	798.5
9 · Cut pass 2 at the last gather	404.5	680.4
10 · Explicit-shape field arrays	368.1	622.6
11 · Gather the zeta stencil once per strip	334.2	560.5
12 · Keep the zeta hoist loop scalar	**328.5**	**543.6**

Confirmed best

4.78×

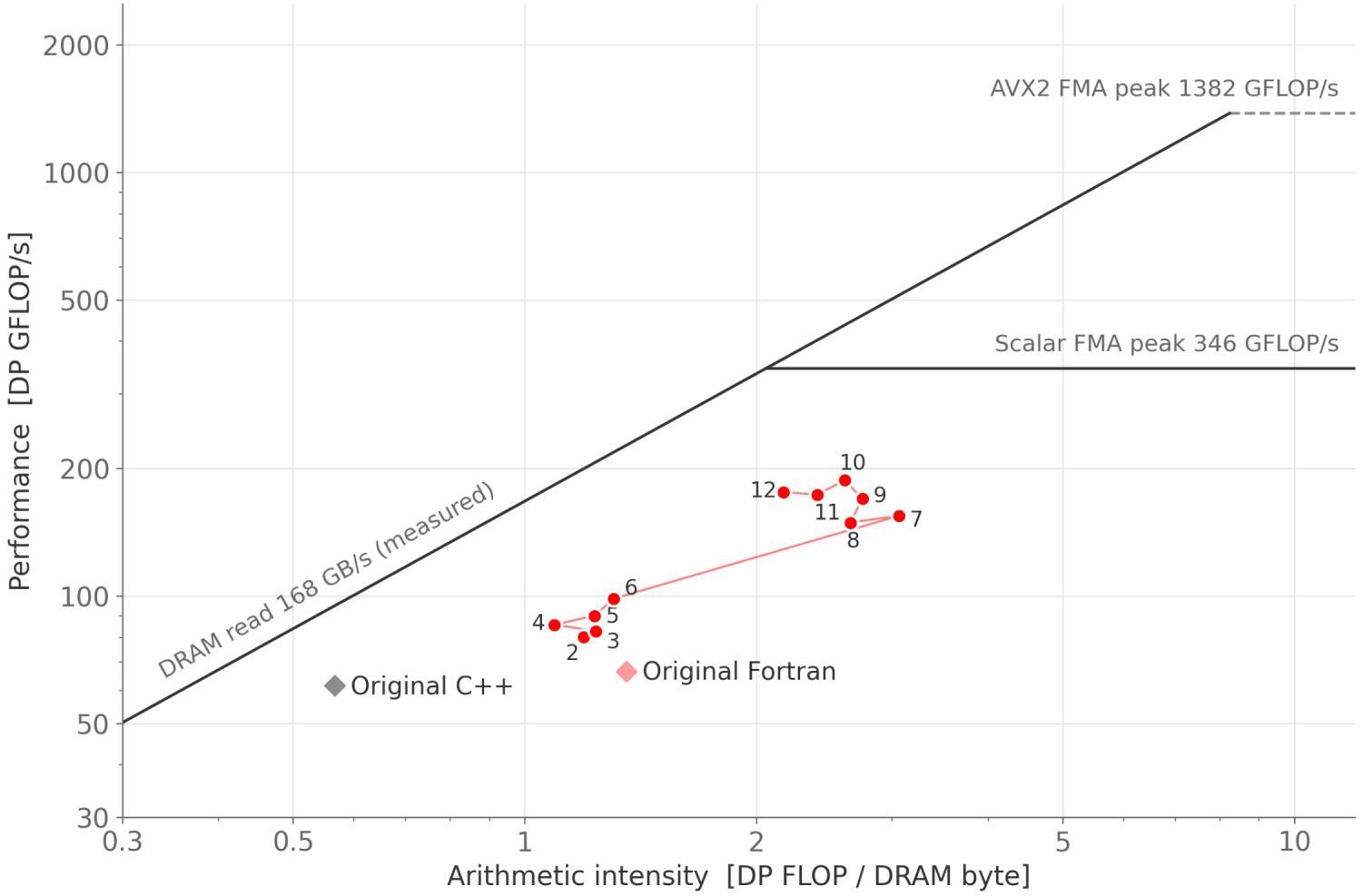
36 threads vs original Fortran
1569.3 → 328.5 ms
3.88× vs original C++

4.74×

18 threads vs original Fortran
2577.4 → 543.6 ms
3.67× vs original C++

Best build: BLOCK_I = 128.

Kernel Roofline



Kernel only: R2, 36 threads, socket 0.
Numbers = steps of the summary table

Step 7: DRAM 59 → 25 GB/call
AI 1.3 → 3.1 FLOP/B

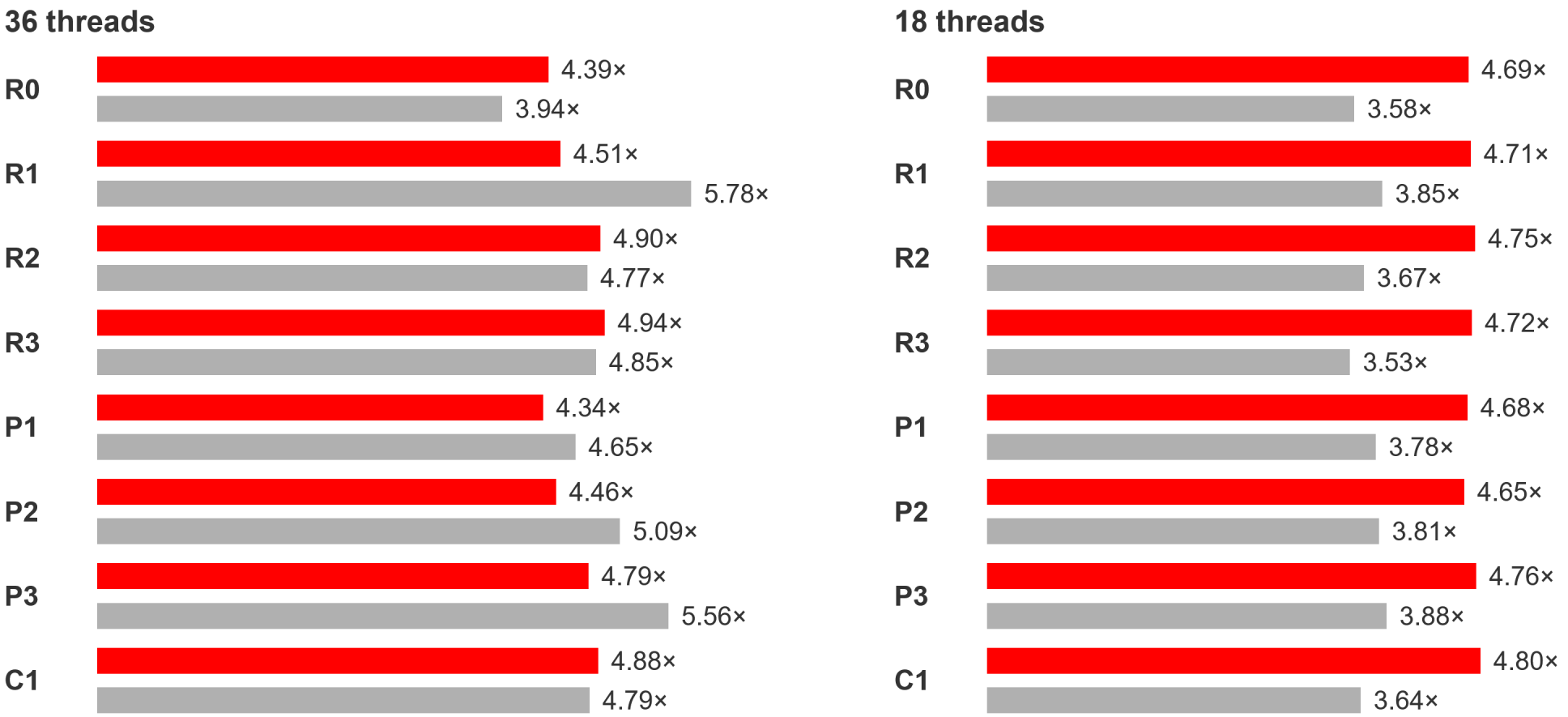
Steps 9–12: 69 → 58 GFLOP/call
fewer FLOPs, so GFLOP/s stays flat while time drops

Best: 176 GFLOP/s, 81 GB/s
51% of scalar peak, 48% of read bandwidth

VTune hpc-performance: DP FP counters and DRAM CAS counts in the kernel's time window. Step 1: no VTune run.

Across Problem Sizes: Jed, ifx

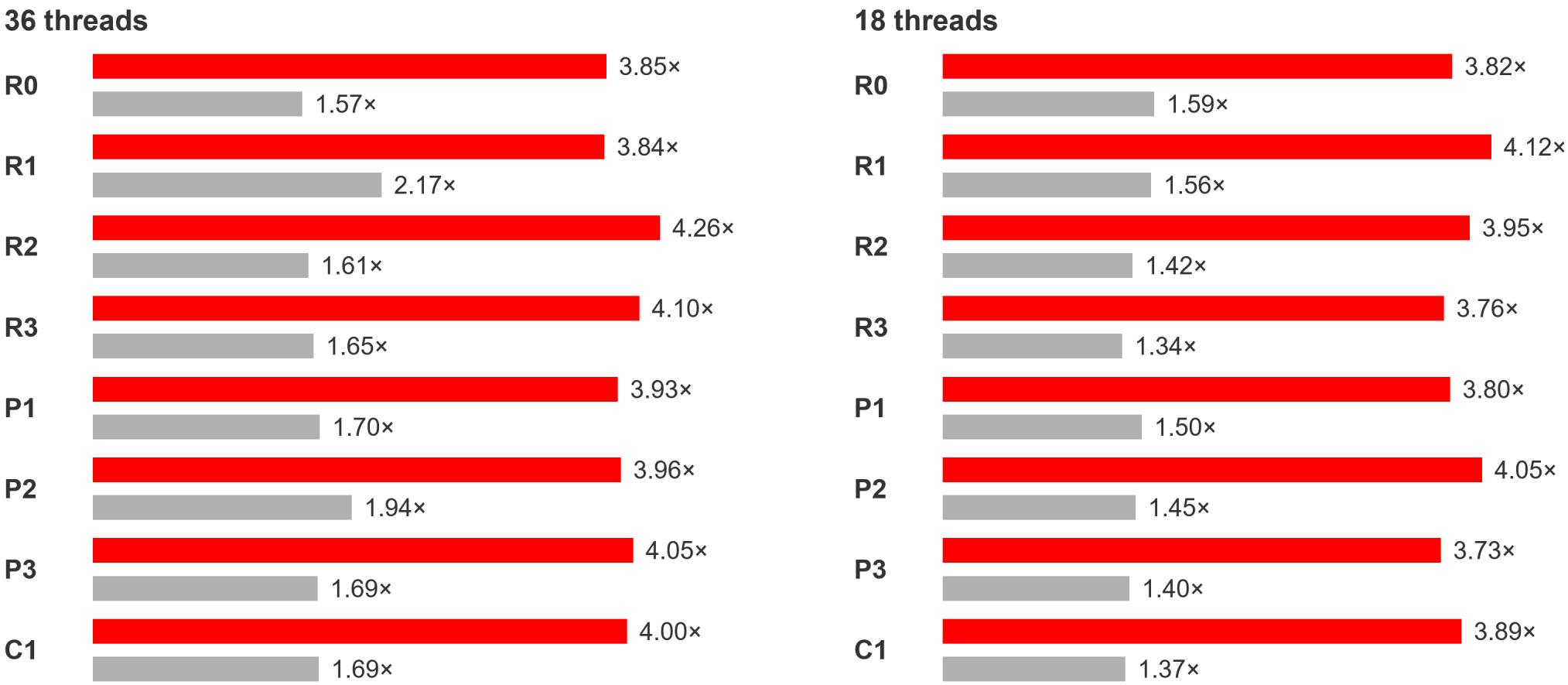
Jed, Intel Xeon 8360Y (Ice Lake), ifx 2025.3. Speed-up of the optimized kernel per problem size: red = vs original Fortran, grey = vs original C++ (original time / optimized time).



4.3–4.9× vs original Fortran, 3.5–5.8× vs original C++, in every case. One socket, memory bound to it.

Across Problem Sizes: Jed, gcc

Jed, Intel Xeon 8360Y (Ice Lake), gfortran/g++ 13.2.0. Speed-up of the optimized kernel per problem size: red = vs original Fortran, grey = vs original C++ (original time / optimized time).

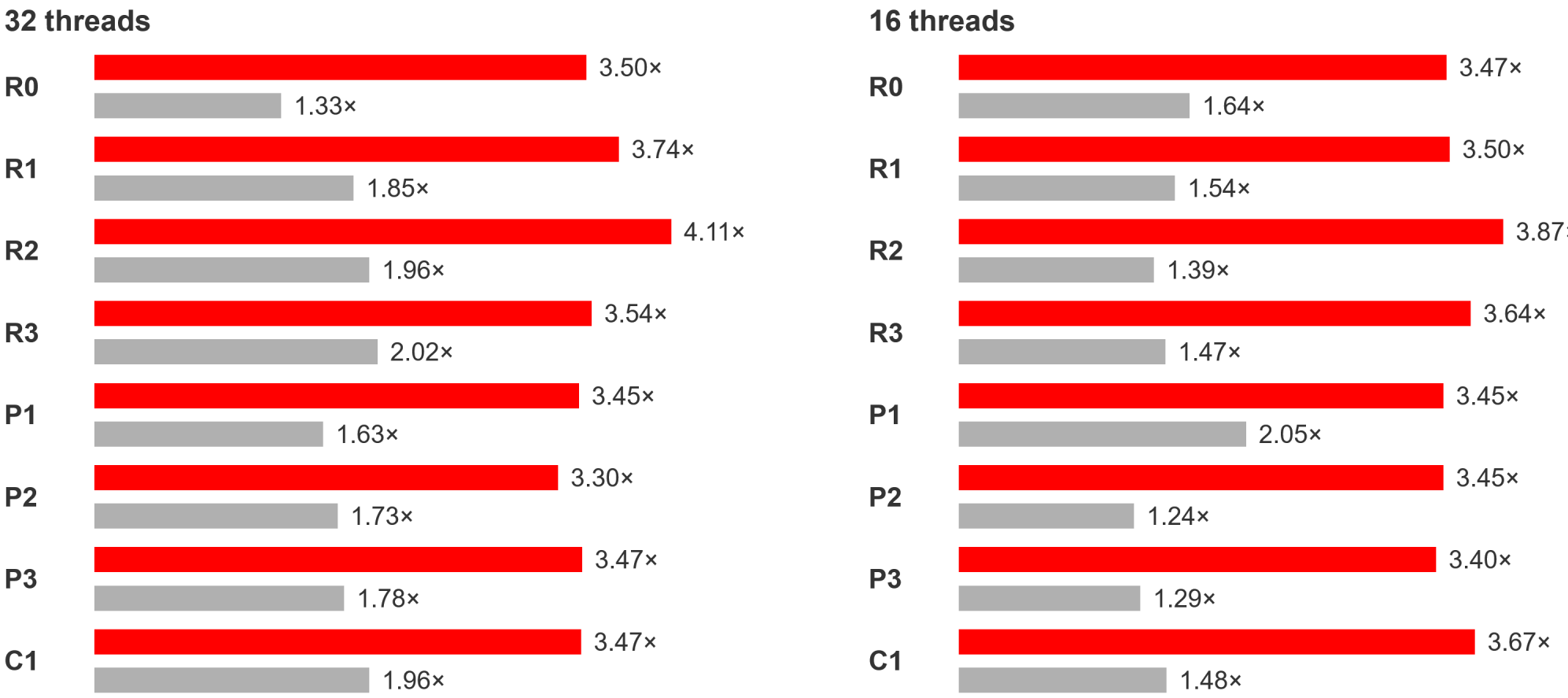


3.7–4.3× vs original Fortran, 1.3–2.2× vs original C++, in every case. One socket, memory bound to it.

gcc time / ifx time: original C++ 0.92–1.29×, original Fortran 2.32–2.63×, optimized 2.76–3.02×

Across Problem Sizes: Kuma, gcc

Kuma, AMD EPYC 9334 (Genoa), gfortran/g++ 13.2.0. Speed-up of the optimized kernel per problem size: red = vs original Fortran, grey = vs original C++ (original time / optimized time).



3.3–4.1× vs original Fortran, 1.2–2.0× vs original C++, in every case. One socket, memory bound to it.

Summary

12 optimizations, one at a time

- Each step measured on R2 against the original Fortran and C++
- Jed, ifx, R2: 1569 → 329 ms at 36 threads, 2577 → 544 ms at 18
- 4.8× vs original Fortran, 3.7–3.9× vs original C++

Intel Xeon 8360Y (Jed), 8 problem sizes

- ifx: 4.3–4.9× vs Fortran, 3.5–5.8× vs C++
- gfortran: 3.7–4.3× vs Fortran, 1.3–2.2× vs C++
- 18 and 36 threads, one socket

AMD EPYC 9334 (Kuma), 8 problem sizes

- gfortran: 3.3–4.1× vs Fortran, 1.2–2.0× vs C++
- 16 and 32 threads, one socket

BLOCK_I

- Fixed at compile time to 128, chosen on Jed
- Ideal value probably depends on the architecture (cache sizes); not tuned on AMD

Thank you for your attention

Questions?

Next Steps

Threads

- 72 threads (both sockets) for all problem sizes

Compilers

- Same problem-size test with gfortran
- LLVM Flang

Other CPUs

- All results: Intel Ice Lake (Jed)
- AMD EPYC and Arm (NVIDIA Grace)

Open items

- BLOCK_I = 256: 313.2 / 501.8 ms (−4.7% / −7.7%)
- Full GENE-X run with the final kernel

The Gyrokinetic Vlasov Equation

*op_rhs_vlasov_eq_static computes the time-independent part (∂f/∂t)**

$$(\partial f/\partial t)^* = -(1/B^*_\parallel)\,[\, v_\parallel\, \mathbf{B}^*\cdot\nabla f + (c/q)\,\mathbf{b}\times\nabla H\cdot\nabla f - (1/m)\,\mathbf{B}^*\cdot\nabla H\,\partial f/\partial v_\parallel\,] + D(f)$$

$\mathbf{B}^* = \mathbf{B} + (m\, v_\parallel\, c/q)\,\nabla\times\mathbf{b} + \nabla A_{1\parallel}\times\mathbf{b}, \quad B^*_\parallel = \mathbf{b}\cdot\mathbf{B}^*$

$H = \mu(B + B_{1\parallel}) + q\varphi_1 - (mc^2/2B^2)|\nabla_\perp\varphi_1|^2$ (last term: H_2, precomputed on the 2D plane)

Streaming

Field-line derivative from planes k±1, k±2 via CSR interpolation maps

`bs_adv`

Drifts

Arakawa bracket on a 3×3 RZ stencil (flutter folded in)

`bx_adv`

Mirror force

5-point $v_\parallel$ stencil times ∇H along **B***

`vp_adv`

Hyperdiffusion

4th-order terms in RZ, along the field line and in $v_\parallel$

`hyp_xz, hyp_y, hyp_vp`

Michels et al., Phys. Plasmas 29, 032307 (2022); Frei et al., Comput. Phys. Commun. (2025) 109817. Normalisation factors omitted.

Pack plane values and accumulators

Collect the five fields of one plane in vals(:, p) and fold them into six y-derivative accumulators acc(1:6) right away.

Before

```
dfdy = this%prefac_cd_y_mm(i, k) * f_mm &
     + this%prefac_cd_y_m(i, k)  * f_m &
     + this%prefac_cd_y_o(i, k)  * f_in(i, k, l, m, n) &
     + this%prefac_cd_y_p(i, k)  * f_p &
     + this%prefac_cd_y_pp(i, k) * f_pp
! ... the same for phi, A_par, B_par, H2, hyp_y
```

After

```
vals(1, 1) = dot_product(map_mm(k)%val(...), f_in(...))
...
vals(5, 1) = dot_product(map_mm(k)%val(...), this%H2(...))
acc(1)   = -this%prefac_hyp_y_mm(i, k) * vals(1, 1)
acc(2:6) =  this%prefac_cd_y_mm(i, k)  * vals(1:5, 1)
! ... planes k-1, k, k+1, k+2 add into acc
```

