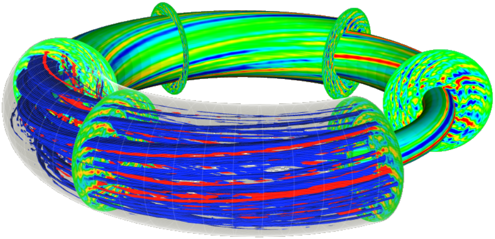
Evaluating kernels on Xeon Phi to accelerate Gysela application

J. Bigot, M. Haefele, G. Latu

CEA/DSM/IRFM & Maison de la Simulation

and coworkers:
T. Cartier-Michaud, G. Dif-Pradalier, C. Ehrlacher, D. Estève, X. Garbet, P. Ghendrih, V. Grandgirard, C. Norscini, C. Passeron, F. Rozar, Y. Sarazin
& **INTEL Exascale lab** & **IPP+HLST Garching**



DE LA RECHERCHE À L'INDUSTRIE

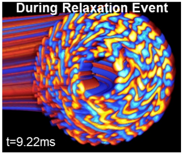
www.cea.fr

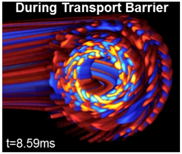
Outline

- Introduction to Gysela code
- Porting on Phi
  - Initial Setup
  - Micro-benchmarks
  - Interpolation kernels
  - 4D advections in Gys-protoapp

GYSELA physics, in short

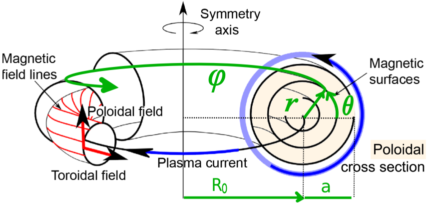
- Main features in GYSELA (Gyrokinetic Semi-Lagragian code):
  - Modelling of Tokamak plasma (targeting ITER)
  - Describing turbulence and transport (ITG instabilities)
    turbulence *governs/limits plasma performance*
  - Main equations: Vlasov 5D, Poisson 3D (quasineutrality)
    gyrokinetic setting (5D = 3D space + 2D velocity)
  - Heat & vorticity sources
    (mimics heating system)
  - Collisional operator
  - Modelling fast particles
  - Adiabatic electron response





- Main unknown: $\bar{f}^n(r,\theta,\varphi,v_{\|},\mu)$

Input : *Physics parameters*, $\bar{f}^0$
Output : *Diagnostics*



for time step $n \geq 0$ **do**
  Integrals: $\mathcal{N}_i^n(r,\theta,\varphi) = \int\int \bar{f}^n \, B(r,\theta)\, \mathcal{J}(k_\perp \rho_C)\, dv_{\|} d\mu$;
  Push fields (Poisson Eq.): $\mathcal{N}_i^n(r,\theta,\varphi) \rightarrow \Phi^n(r,\theta,\varphi)$;
  Diagnostics for time step n;
  Push particles (Vlasov Eq. + other terms): $\Phi^n(r,\theta,\varphi), \bar{f}^n \rightarrow \bar{f}^{n+1}$;

Algorithm 1: Overall simplified Gysela algorithm

- **Fortran 90** code, hybrid **MPI+OpenMP**

- *Simplified* view of gyrokinetic Vlasov equation (dir. splitting):

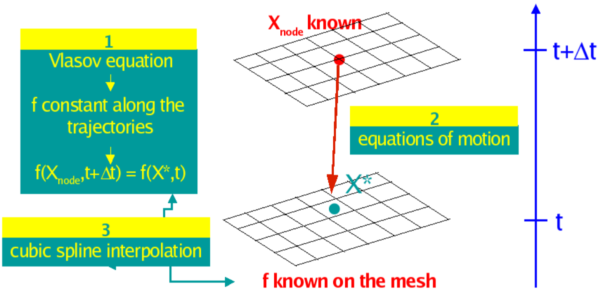
$$\frac{\partial \bar{f}}{\partial t} + \frac{dr}{dt}\frac{\partial \bar{f}}{\partial r} + \frac{d\theta}{dt}\frac{\partial \bar{f}}{\partial \theta} + \frac{d\varphi}{dt}\frac{\partial \bar{f}}{\partial \varphi} + \frac{dv_{\|}}{dt}\frac{\partial \bar{f}}{\partial v_{\|}} = 0 \text{ (collisionless)}$$

- Solved through advections, **Semi-Lagrangian** scheme:

$$\partial_t \bar{f} + v_{\|}\partial_{\varphi}\bar{f} = 0 \;\; (\hat{\varphi} \;\; \text{operator})$$
$$\partial_t \bar{f} + \dot{v}_{\|}\partial_{v_{\|}}\bar{f} = 0 \;\; (\hat{v}_{\|} \;\; \text{operator})$$
$$\partial_t \bar{f} + \overrightarrow{v_{GC}} \cdot \overrightarrow{\nabla}_{\perp}\bar{f} = 0 \;\; (r\hat{\theta} \;\; \text{operator})$$

- Vlasov solver (**explicit** scheme) is composed of:
  - Successive directional splittings (advection steps)
  - Main cost of the application: interpolations (cubic splines)

Backward semi-Lagrangian scheme advections



- $\bar{f}$ conserved along characteristics
- Find the origin of the characteristics ending at the grid points (spatial grid)
- Interpolate value at origin $X^\star$ from known grid values: Cubic spline interpolation

3 steps for one advection:

- compute splines coefficients,
- compute feet (equations of motion),
- interpolate values.

for time step $n \geq 0$ **do**

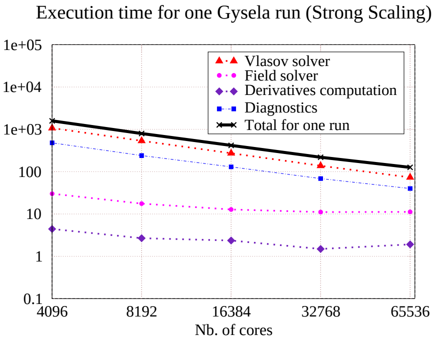
Integrals, Poisson, Diagnostics

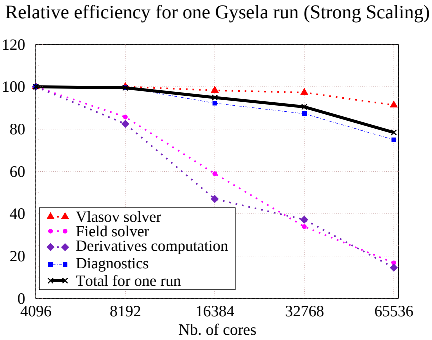
$$\text{Vlasov}\begin{cases}
\text{1D Advection in } v_{\parallel} & (\forall(\mu,r,\theta)=[local],\forall(\varphi,v_{\parallel})=[*]);\\
\text{1D Advection in } \varphi & (\forall(\mu,r,\theta)=[local],\forall(\varphi,v_{\parallel})=[*]);\\
\text{Transposition of } f; & \\
\text{2D Advection in } (r,\theta) & (\forall(\mu,\varphi,v_{\parallel})=[local],\forall(r,\theta)=[*]);\\
\text{Transposition of } f; & \\
\text{1D Advection in } \varphi & (\forall(\mu,r,\theta)=[local],\forall(\varphi,v_{\parallel})=[*]);\\
\text{1D Advection in } v_{\parallel} & (\forall(\mu,r,\theta)=[local],\forall(\varphi,v_{\parallel})=[*]);
\end{cases}$$

Algorithm 2: Parallel algo.: 2 domain decompositions

- no CFL for advections,
  comm. for transposition: $\Theta(N_r\, N_\theta\, N_\varphi\, N_{v_\parallel}\, N_\mu)$

$N_r = 512, \quad N_\theta = 512, \quad N_\varphi = 128, \quad N_{v_\parallel} = 128, \quad N_\mu = 32$ (main unknown $\bar{f}_n$ = 1 TB)

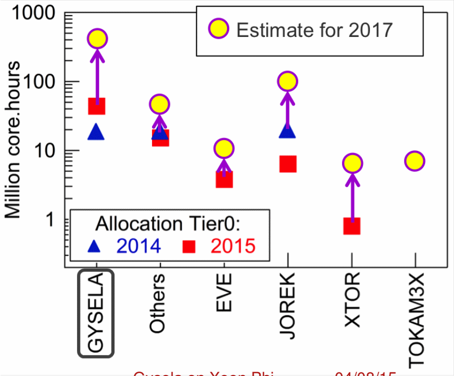




- Good result:
  78% relative efficiency on 64k cores
  (91% in Vlasov part)

Increased computer power requirements

- Fusion applications (and our institute CEA/DSM/IRFM) requires **computing power** for forthcoming years
- Supercomputers tends to provide more and more accelerators
  $\longrightarrow$ candidates for next generation of parallel architectures
  $\longrightarrow$ INTEL **Xeon Phi** and **GPGPUs** (AMD + Nvidia)



Outline

- Introduction to Gysela code
- Porting on Phi
  - Initial Setup
  - Micro-benchmarks
  - Interpolation kernels
  - 4D advections in Gys-protoapp

Outline

- Introduction to Gysela code
- Porting on Phi
  - Initial Setup
  - Micro-benchmarks
  - Interpolation kernels
  - 4D advections in Gys-protoapp

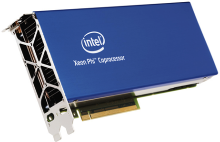
Introducing INTEL Xeon Phi

Testbed: Helios machine (Fusion community, Japan)

Processor	Intel Xeon Sandy Bridge E5	Intel Xeon Phi 5110P
Clock frequency	2.1 - 2.8 GHz	1.05 - 1.238 GHz
Number of cores	8	60
Available memory	32 GB	8 GB
Peak performance (double precision)	173 GFlops/s	1011 GFlops/s
Sustainable memory bandwidth	40 GB/s	160 GB/s
Instruction execution model	out of order	in order
Simultaneous Multi Threading	2-way	4-way
Instruction set	x86-64 + *256bits-AVX*	x86-64 + *512bits-SSE*

- 2 programming models for Xeon Phi:
  - *offload* mode:
    - Phi as an accelerator
    - `#pragma` based
  - *native* mode:
    - Phi as a linux node
    - classical MPI + OpenMP

- Testbed (Helios machine - Fusion community, Japan)
  - Xeon Phi copro (5110P), 60 cores, 8GB mem., clock 1.05 Ghz
  - Sandy B. node (E5-2680), 2×8 cores, 64GB mem., clock 2.7 Ghz
- Initial assumptions on Xeon Phi
  - Easy to port code (x86 arch.)
  - Support OpenMP/MPI paradigm
  - How to get good performance ?

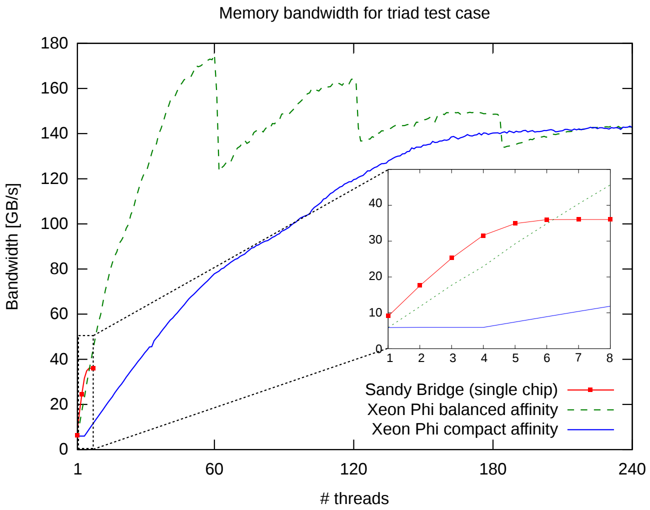


- Raw performance (×3 CPU peak, ×2 mem. BW)
  - SB: CPU peak 342 GFLOPS, mem bandwidth 70 GB/s (Stream triad)
  - Phi: CPU peak 1011 GFLOPS, mem bandwidth 130 GB/s (Stream triad)
- Approach in 4 steps:
  1. Direct port of a subset of Gysela: poor performance ☹
  2. Memory and MPI benchmarks: inhomogeneous perf.
  3. Fallback: tune interpol. kernels (needed in Gysela), no MPI
  4. Try to put back a performant interpol. kernel into Gysela

Outline

- Introduction to Gysela code
- Porting on Phi
  - Initial Setup
  - Micro-benchmarks
  - Interpolation kernels
  - 4D advections in Gys-protoapp

Memory Bandwidth



Memory Bandwidth + Latency, Network

- Bandwidth on Xeon Phi
  - Up to 175 GB/s on the Xeon Phi with one thread/core
  - But 144 GB/s with 4 threads/core
  - With 2 or 3 threads/core, thread *affinity/pinning* does matter
  - x4 in mem. bandwidth compared to 1-socket S. Bridge
  - x2 in mem. bandwidth compared to 2-socket S. Bridge
- Latency Xeon Phi versus Sandy Bridge
  - Similar L1 latencies
  - x4-x20 increase otherwise on Xeon Phi (L2, L3, memory)

    $\rightarrow$ Cache reuse implementations have to target L1, L2

    $\rightarrow$ Requires more efforts from the developer
- Network performance (MPI communications)
  - Bandwidth decreased with Xeon Phi vs Sandy B.
  - Latency increased with Xeon Phi vs Sandy B.

Outline

- Introduction to Gysela code
- Porting on Phi
  - Initial Setup
  - Micro-benchmarks
  - Interpolation kernels
  - 4D advections in Gys-protoapp

Parallelizing simplified 1D interpolator (1)

- Parallelization strategy (no MPI) :
  - **native** mode choosen
    - because offload is slower (our tests on several configs)
    - avoid overhead due to Host-to-Phi data transfers (offload)
  - outer loops: **OpenMP**
  - inner block: loop vectorization through **SIMD directives**
- Code example: 1D advec - lagrange order 3 - on 4D data

```
#pragma omp parallel for collapse(3)
 for (x1=0; x1<Nx1; x1++) {
  for (x2=0; x2<Nx2; x2++) {
   for (x3=0; x3<Nx3; x3++) {
#pragma vector nontemporal (f1)
#pragma vector always
    for (x4=0; x4<Nx4; x4++) {
     access_f(f1,x4,x3,x2,x1) = // OUTPUT data f1
       coef1 * access_f(f0,x4-1,x3,x2,x1) + // INPUT data f0
       coef2 * access_f(f0,x4  ,x3,x2,x1) +
       coef3 * access_f(f0,x4+1,x3,x2,x1) +
       coef4 * access_f(f0,x4+2,x3,x2,x1);
    } } } }
```

- Vectorization through 512-bit MIC intrinsics

  only C language, Fortran is not accessible

- Code example: 1D advec - lagrange order 3 - on 4D data

```
#pragma omp parallel for collapse(3)
 for (x1=0; x1<Nx1; x1++) {
  for (x2=0; x2<Nx2; x2++) {
   for (x3=0; x3<Nx3; x3++) {
    for (x4=0; x4<Nx4; x4+=8) {
      ptread = &(acces_f(f0,x4,x3,x2,x1));
      // read input data
      tmpr2 = _mm512_load_pd  (ptread);
      tmpr1 = _mm512_loadunpacklo_pd(tmpr1, ptread-1);
      tmpr1 = _mm512_loadunpackhi_pd(tmpr1, ptread-1+8);
      tmpr3 = _mm512_loadunpacklo_pd(tmpr3, ptread+1);
      tmpr3 = _mm512_loadunpackhi_pd(tmpr3, ptread+1+8);
      tmpr4 = _mm512_loadunpacklo_pd(tmpr4, ptread+2);
      tmpr4 = _mm512_loadunpackhi_pd(tmpr4, ptread+2+8);
      // 1+2+2+2=7 flop per loop iteration
      tmpw = _mm512_mul_pd(tmpr1,coeff1);
      tmpw = _mm512_fmadd_pd(tmpr2,coeff2,tmpw);
      tmpw = _mm512_fmadd_pd(tmpr3,coeff3,tmpw);
      tmpw = _mm512_fmadd_pd(tmpr4,coeff4,tmpw);
      // write output data
     _mm512_store_pd (&(access_f(f1,x4,x3,x2,x1)), tmpw);
    } } } }
```

- Code example: 2D advec - lagrange order 3 - on 4D data

```
#pragma omp parallel for collapse(3)
 for (x1=0; x1<Nx1; x1++) {
  for (x2=0; x2<Nx2; x2++) {
   for (x3=0; x3<Nx3; x3++) {
#pragma vector nontemporal (f1)
#pragma vector always
    for (x4=0; x4<Nx4; x4++) {
     access_f(f1,x4,x3,x2,x1) =
       coefb1 * (coefa1 * access_f(f0,x4-1,x3-1,x2,x1) +
                 coefa2 * access_f(f0,x4  ,x3-1,x2,x1) +
                 coefa3 * access_f(f0,x4+1,x3-1,x2,x1) +
                 coefa4 * access_f(f0,x4+2,x3-1,x2,x1)  ) +
       coefb2 * (coefa1 * access_f(f0,x4-1,x3  ,x2,x1) +
                 coefa2 * access_f(f0,x4  ,x3  ,x2,x1) +
                 coefa3 * access_f(f0,x4+1,x3  ,x2,x1) +
                 coefa4 * access_f(f0,x4+2,x3  ,x2,x1)  )  +
       coefb3 * (coefa1 * access_f(f0,x4-1,x3+1,x2,x1) +
                 coefa2 * access_f(f0,x4  ,x3+1,x2,x1) +
                 coefa3 * access_f(f0,x4+1,x3+1,x2,x1) +
                 coefa4 * access_f(f0,x4+2,x3+1,x2,x1)  )  +
       coefb4 * (coefa1 * access_f(f0,x4-1,x3+2,x2,x1) +
                 coefa2 * access_f(f0,x4  ,x3+2,x2,x1) +
                 coefa3 * access_f(f0,x4+1,x3+2,x2,x1) +
                 coefa4 * access_f(f0,x4+2,x3+2,x2,x1)  )
       } } } }
```

Phi versus Sandy Bridge
Performance of interpolation kernels (1)

Mem bound

1. 1D advection (constant/small displacement) 1D interp lagrange 3
   - **Phi** perf: 46 GFLOPS (5% peek), BW: 106 GB/s (81% stream)
   - **SB** perf: 25 GFLOPS (7% peek), BW: 57 GB/s (81% stream)
2. 2D advection (constant/small displacement) 2D interp lagrange 3
   - **Phi** perf: 250 GFLOPS (25% peek), BW: 111 GB/s (85% stream)
   - **SB** perf: 134 GFLOPS (39% peek), BW: 59 GB/s (84% stream)

A factor $\times 2$ is obtained on Phi compared to one full SB node
match expected behaviour ☺

Performance on Phi is varying much (10% is common) ☹
with domain size, and from one run to the other

Phi versus Sandy Bridge
Performance of interpolation kernels (2)

Compute Bound

1. 3D advection (constant displacement) 3D interp lagrange 3, 4D data
   - **Phi** perf: 228 GFLOPS (23% peek), BW: 25 GB/s (19% stream)
   - **SB** perf: 156 GFLOPS (46% peek), BW: 17 GB/s (25% stream)
2. 4D advection (constant displacement) 4D interp lagrange 3, 4D data
   - **Phi** perf: 160 GFLOPS (16% peek), BW: 4.3 GB/s (3.3% stream)
   - **SB** perf: 145 GFLOPS (42% peek), BW: 3.9 GB/s (5.6% stream)

- **Hard/long** to get good perf. on complex kernels on Phi ☹
- **3D** stencil easier to optimize than **4D** stencil (complex memory pattern)
- Speedup up to ×2 in best cases (Phi versus one SB node) ☺
- Small modifications OR changing compiler version

→ bad vectorization by the compiler on Phi → slowdown by ×4 ☹

- Prefetch (load data in advance) accelerates computation
  → especially on memory-bound kernels
  save 20% exec time on 1d/2d kernels
- Cache blocking (loop tiling) is crucial
  → especially on compute-bound kernels
  → save exec time on 3d kernels (50% reduction on exec. time)
- Tune aligned data, avoid cache trashing
  → save 20% exec time on 1d/2d kernels
- Comparing similar **C** and **Fortran** kernels
  → not clear tendency
  give better or worse exec. time depending on the kernel

Some problems & recipes (2)

- Internal compiler optim. impact perf.
  (much more on Phi than on SB) ☹
  → compiler does not give comprehensive feedbacks
  → looking at **generated assembly** code is painful but helpful
  → splitting the body of loop into multiple loops lead to
  effective speedups

- Writing “assembly” version may speedup computation
  (C code only)
  → 512-bit intrinsics help especially on compute-bound kernels ☺

- Phi works well with 170 up to 240 *well-pinned* threads
  versus 16 threads for SB ☹

Outline

- Introduction to Gysela code
- Porting on Phi
  - Initial Setup
  - Micro-benchmarks
  - Interpolation kernels
  - 4D advections in Gys-protoapp

Designing a simplified-smaller Gysela

- Goal: feasibility of porting Gysela on Phi &
  rough estimate of the performance
- Approach: design a simplified version of Gysela, named **Gys-protoapp**

Gys-protoapp code

- Remove non-essential parts of the Gysela code
  - diagnostics, alternative implementations, collisions, sources (keep the smallest set of numerical kernels Vlasov+Poisson)
  - 50k loc (Gysela) $\longrightarrow$ 14k loc (proto-app)
- Remove a lot of MPI communication schemes
  - Restrict a single μ value (4D problem instead of 5D)
  - Single node execution (works with `mpirun -np 1`)
  - A simulation $N_r = 128, N_\theta = 256, N_\varphi = 32, N_{v_\parallel} = 64$: 10 hours on one SB node
- Add a new Vlasov solver (4D advection algorithm)
  - Computation intensive kernel, well-suited for Xeon Phi

4D advection replaces splitting scheme

- Usual Vlasov solver uses directional splitting (*i.e.* 1D and 2D advection operators - mem. bound):
  $$(\hat{v}_{\parallel}/2,\ \hat{\varphi}/2,\ \widehat{r\theta},\ \hat{\varphi}/2,\ \hat{v}_{\parallel}/2)$$

- Design a new 4D advection approach (compute bound):

$\eta(r=*, \theta=*, \varphi=*, v_{\parallel}=*) \leftarrow$ compute spline coeff. from the 4D function $f^n(r=*, \theta=*, \varphi=*, v_{\parallel}=*)$;

for All grid points $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})$ **do**

$(r_i, \theta_j, \varphi_k, v_{\parallel\, l})^\star \leftarrow$ foot of characteristic that ends at $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})$;

$f^{n+1}(r_i, \theta_j, \varphi_k, v_{\parallel\, l}) \leftarrow$ interpolate f^n at location $(r_i, \theta_j, \varphi_k, v_{\parallel\, l})^\star$ using η ;

Algorithm 3: 4D semi-Lagrangian scheme

4D advection properties

- Three main kernels (rough profiling given):
  - 4D spline interpolator (**51 %** of exec. time)
  - feet of characteristics (**36 %**)
  - spline coeff computation (**10 %**)

- 4D interpolator - vectorization opportunities:
  - 4D tensor product with stencil of size 4,
    per grid point: 595 FLOP, read 1 float, write 1 float
  - very high computational intensity

- Parallelization/Optimization strategy (no MPI) :
  - Phi native mode, OpenMP
  - Cache friendly: loop blocking (2 levels in $v_{\|}$ and φ)
  - Reuse feet stored into L2 cache (temporal locality)
- Code structure of *spline 4D advection* :

```
do ith_blk=0, nb_blk_th  ! loop blocking in theta
 do ivpar=0, Nvpar
   call feet_computations_with_openmp(...)
!$OMP PARALLEL DO COLLAPSE(2)
    do iphi=0, iNphi
       do ith=ith_blk*th_bsize,(ith_blk+1)*th_bsize-1
          call interpolations_vectorized_kernel(...);
       end do
    end do
 end do
end do
```

Parallelizing spline 4D advection - SIMD part

```
#define R_BSIZE 8
subroutine interpolations_vectorized_kernel(..., spline coeff.)
 do ir_outer=0, Nr, R_BSIZE
    ! retrieve grid cell containing the foot, compute spline basis
!dir$ simd
   do ir_inner=0,R_BSIZE-1
      ir=ir_outer+ir_inner
      r_foot=...; th_foot=...; vpar_foot=...; phi_foot=...;
      ir_star       = map_on_grid(r_foot)
      ith_star      = map_on_grid(th_foot)
      ivpar_star    = map_on_grid(vpar_foot)
      iphi_star     = map_on_grid(phi_foot)
      sbasis(1:16)  = compute_spline_basis(*_star,*_foot)
   end do
   ! interpolate in combining spline basis and spline coeff.
   psum(0:R_BSIZE-1) = 0.
   do <nest_of_four_loops>
!dir$ simd
     do ir_inner=0,R_BSIZE-1
         coeff = load spline coeff. located at *_star (with unit stride)
         psum(ir_inner) = psum(ir_inner) + coeff(...) * sbasis(...)
     end do
   end do
   f1(ir_outer:ir_outer+R_BSIZE-1,ith,iphi,ivpar)=psum(0:R_BSIZE-1)
 end do
end subroutine interpolations_vectorized_kernel
```

Gys-protoapp advection kernel

- 4D advection (**variable** displacement) 4D **cubic spline**, 4D data

$$N_r = 128, N_\theta = 128, N_\varphi = 128, N_{v_\parallel} = 64$$

  - **Phi** perf: 80 GFLOPS (7% peek), BW: 2.7 GB/s (2.% stream)
  - **SB** perf: 33 GFLOPS (9% peek), BW: 1.1 GB/s (1.6% stream)

- Variable displacements $\rightarrow$ unpredictable mem. access (prefetch pb)

- Reduced performance compare to previous kernels

$\rightarrow$ variable displacements: costs induced by **integer computations**, memory indirections

$\rightarrow$ memory accesses cannot always be well aligned

- Sensivity to intel compiler version

$\rightarrow$ SIMD instructions employed and optimizations performed are varying

- Quite a long way to get this optimized version ...

Gys-protoapp benchmark

Put back the 4D kernel in Gys-protoapp on Xeon Phi:

- From first port on Phi, to optim. version, factor ×14 on exec. time ☺

  $$N_r = 128, N_\theta = 128, N_\varphi = 32, N_{v_\parallel} = 64$$

  → First port (one call to Vlasov solver): 45 s
  → Optim. version (one call to Vlasov solver): 3.2 s

- Execution time: ×2 larger on Phi than on SB (16 cores) ☹
  → Amdhal's law: others computations should be optimized also ...
  1) computation of the feet characteristics
  2) spline coeff. computations

- Optimizations was useful for running on SB ☺
  → Overall execution time: reduced by 30% up to 45% on typical cases
  → vectorization directives have some interesting collateral effect
  → Tuned 4D advection is competitive compared to classical Strang splitting

Conclusion

- Achieving good performance on Phi:
  - not impossible ☺, but harder than on Sandy Bridge
  - successful on simple interpolation kernel ☺
  - needs: vectorization, fine grain parallelism, cache, prefetch
  - interact with the compiler (look at the generated assembly code)
  - easier if *only* one small kernel needs to be optimized
- Gys-protoapp (reduced Gysela application):
  - Xeon Phi still 2× slower than Sandy Bridge (16 cores) ☹
  - Sandy Bridge perf. of Gys-protoapp improved (30%-45%) ☺

Paper in CEMRACS'14 proceedings, accessible at

`http://arxiv.org/abs/1503.04645`

Acknowledgments: T. Guillet (Intel Exascale lab) for fruitful discussions,
IFERC-CSC, Rokkasho, Japan (`Helios` supercomputer),
RZG, Garching, Germany (`Mick` machine)

Input : $\bar{f}^{\star}(r,\theta,\varphi,v_{\|},\mu)$
Output : $\bar{f}^{\diamond}(r,\theta,\varphi,v_{\|},\mu)$

for $\underline{\mu}$ **do in parallel MPI**
  for $\underline{r}$ **do in parallel MPI**
    for $\underline{\theta}$ **do in parallel MPI**
      for $\underline{\theta}$ **do in parallel OpenMP**
        for $\underline{v_{\|}}$ **do**
          $\Delta\varphi \leftarrow v_{\|}\,\Delta t$
          Compute spline representation of $\bar{f}^{\star}(r,\theta,\varphi{=}*,v_{\|},\mu)$
          for $\underline{\varphi}$ **do**
            $\bar{f}^{\diamond}(r,\theta,\varphi,v_{\|},\mu) =$
            $spline\_interpolate(\bar{f}^{\star}(r,\theta,\varphi-\Delta\varphi,v_{\|},\mu))$

Algorithm 4: Advection in variable φ on $\bar{f}^{\star}$

MPI communication performance

Ping pong benchmark from the Intel MPI Benchmark (IMB)

Host0	CPU1	0.69		
	MIC0	4.90	2.73	
	MIC1	4.31	7.56	3.12
Host1	CPU1	**2.20**		
	MIC0	4.71	**9.04**	
	MIC1	4.66	**7.93**	6.92
		CPU1	MIC0	MIC1
		Host0		

Latencies (μs)

Host0	CPU1	5029		
	MIC0	**456**	2016	
	MIC1	1609	**416**	2004
Host1	CPU1	**5729**		
	MIC0	418	**273**	
	MIC1	1608	418	969
		CPU1	MIC0	MIC1
		Host0		

Bandwidth (MB/s)

- Similar results on supermic (LRZ, Garching, Germany), Eurora (Cineca, Italy), Robin (Bull R&D, Grenoble, France)
- In green: Typical performance for Infiniband
- In red: Low and non homogeneous network performance