Accelerating plasma simulation codes with portable frameworks: OpenACC and Kokkos

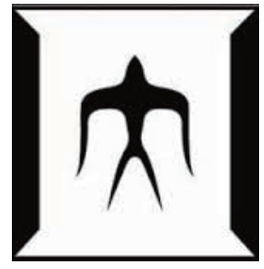
Yuuichi ASAHI[1]

G. Latu[2], **V. Grandgirard**[2], **J. Bigot**[3]

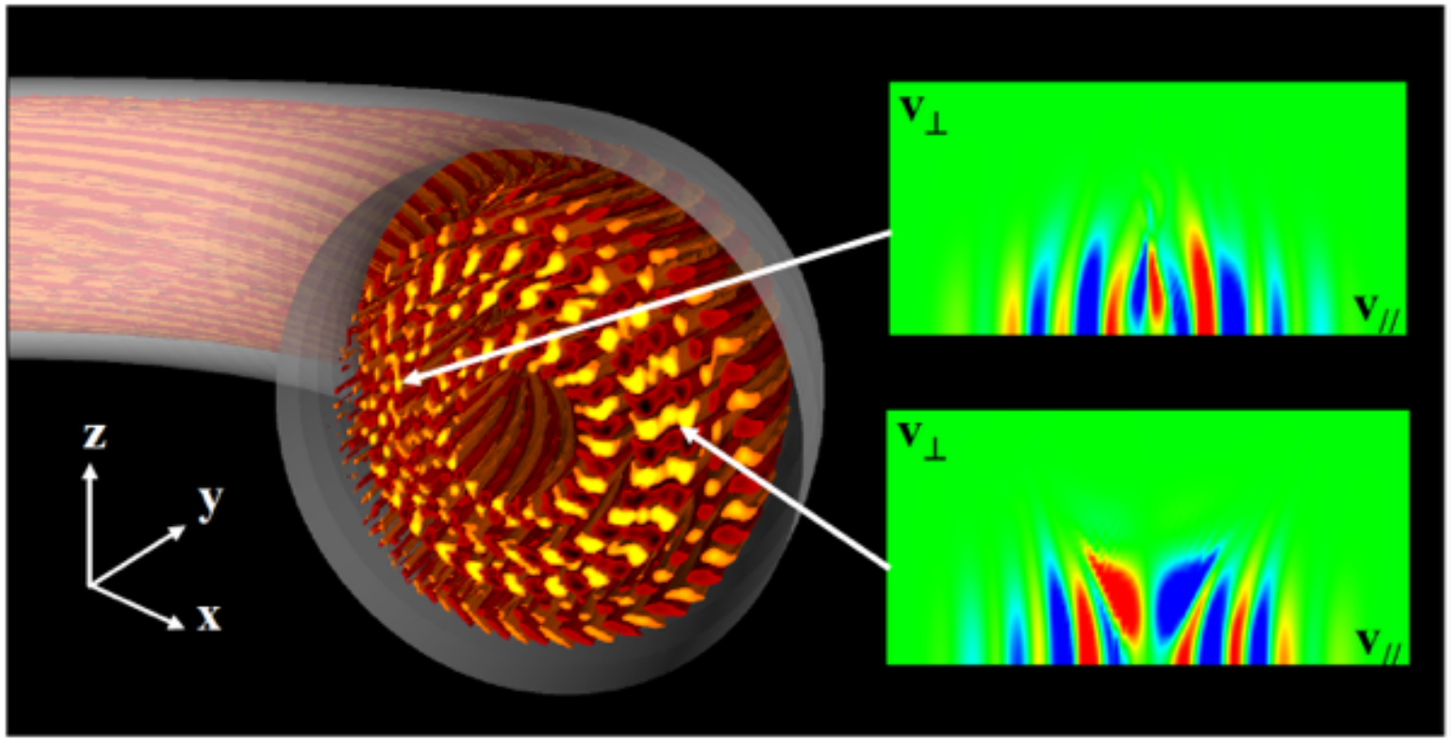
[1] QST, Rokkasho, Aomori, Japan
[2] IRFM, CEA, F-13108, St. Paul-lez-Durance cedex, France
[3] Maison de la Simulation, CEA, CNRS, 91191 Gif-sur-Yvette, France

Date: 2/September/2019

OpenACC Annual Meeting 2019, Kobe, Japan



Plasma turbulence simulation



Each grid point has structure in real space (x, y, z) and velocity space ($v_{\parallel}$, $v_{\perp}$)

⟶ **5D** stencil computations

[Idomura et al., Comput. Phys. Commun (2008); Nuclear Fusion (2009)]

- Fusion plasma performance is dominated by plasma turbulence
- First principle full-f 5D gyrokinetic model is employed for plasma turbulence simulation
  - ⟶ Peta-scale machine required due to huge computational cost (even for single-scale simulation: MPI + **OpenMP** approach)
- Concerning the dynamics of **kinetic electrons**, **complicated geometry** more computational resource is needed
  - ⟶ **Accelerators** are key ingredients to satisfy huge computational demands at **reasonable energy consumption:** MPI + **'X'**

Outline

Introduction: Demands of acceleration in GK codes

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapp
- Aim and setting of this research

Kokkos and OpenACC/OpenMP versions of mini-app

- Higher level abstraction in kokkos: memory and operation
- Mixed OpenACC/OpenMP implementation

Performance measurement and optimization

- Performance improvement with 3D Range policy in Kokkos
- Detailed analysis of kernels based on Roofline model
- Readability, Performance portability, Productivity in each implementation

Summary and future work

Outline

Introduction: Demands of acceleration in GK codes

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapp
- Aim and setting of this research

Kokkos and OpenACC/OpenMP versions of mini-app

- Higher level abstraction in kokkos: memory and operation
- Mixed OpenACC/OpenMP implementation

Performance measurement and optimization

- Performance improvement with 3D Range policy in Kokkos
- Detailed analysis of kernels based on Roofline model
- Readability, Performance portability, Productivity in each implementation

Summary and future work

Demands for MPI + 'X' in our group

Portability

ARM machine
Apollo [1]

GPU machine
SUMMIT [2]

Exa machine may be very **divergent**

Readability

OpenMP

```
#pragma omp parallel for
for(int i=0; i<n; i++)
  a[i] = b[i] + scalar * c[i];
```

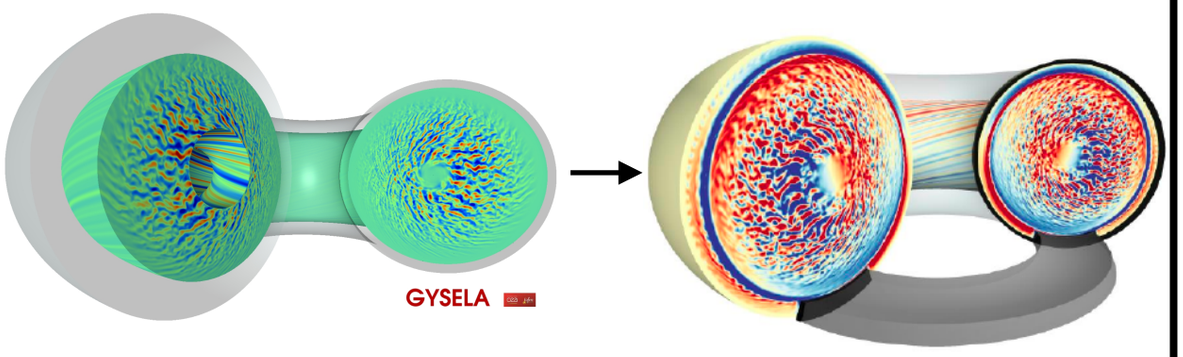
OpenACC

```
#pragma acc parallel loop
for(int i=0; i<n; i++)
  a[i] = b[i] + scalar * c[i];
```

Readable for physicists

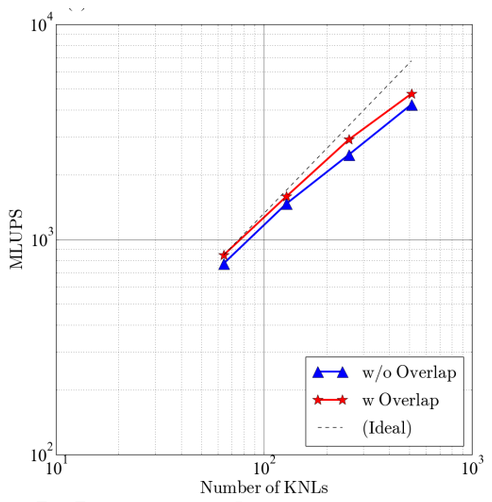
Productivity

Circular → **Limiter**



Advanced (realistic) physical model

High Performance



Strong scaling
of GYSELA
up to 512 KNLs
(MPI+OpenMP)

More than 100 M cpu hours/year

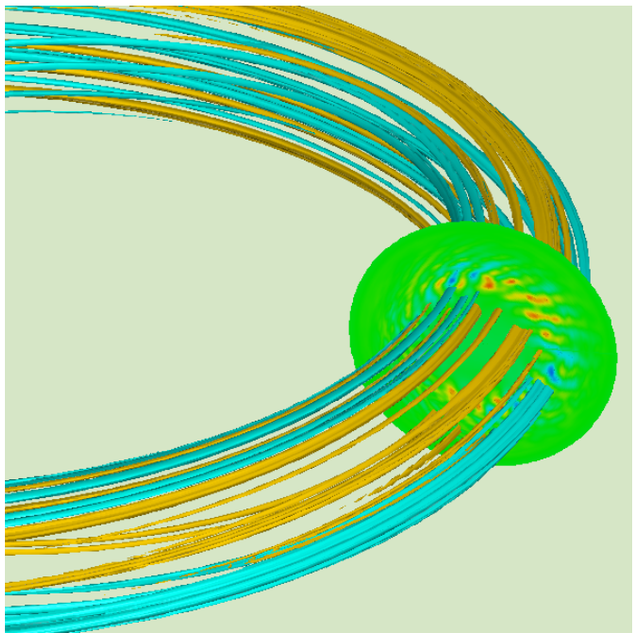
We need a **readable, portable,** and **high performance code** that is **easy to upgrade**!

[1] https://www.extremetech.com/computing/271759-worlds-largest-arm-based-supercomputer-launched-as-exascale-heats-up
[2] https://www.olcf.ornl.gov/summit/

Fluid vs Kinetic simulations

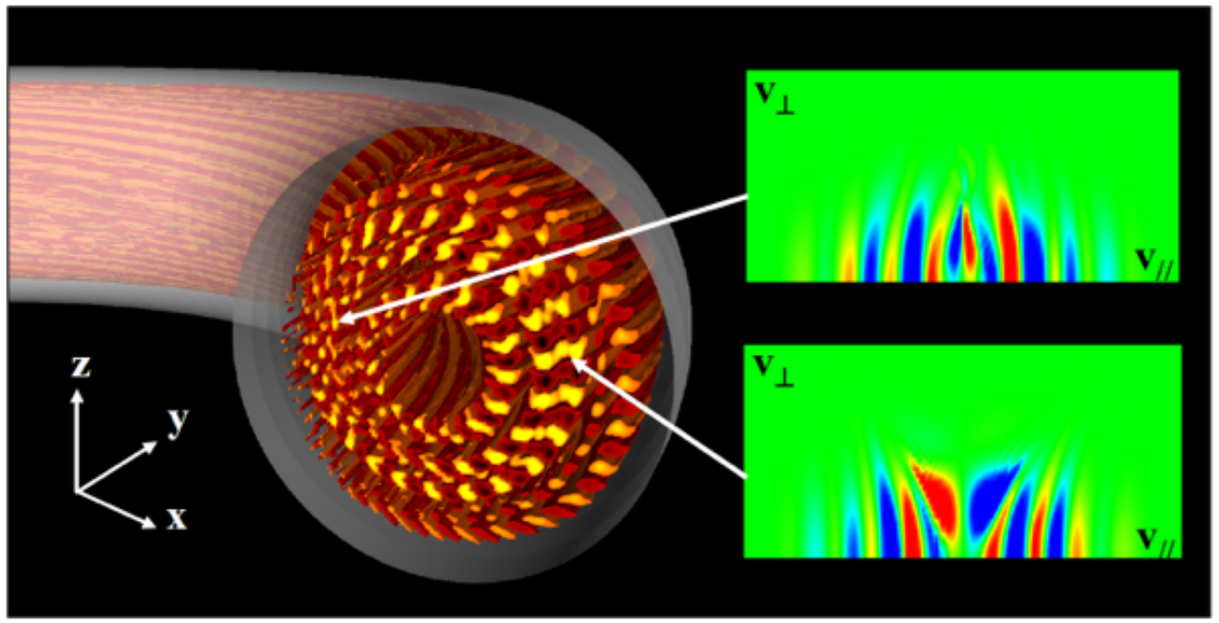
	Fluid	Kinetic
Resolution	~ 1000 grid points/	~ 100 grid points/dimension
Dimension	2D or 3D	**4D or 5D**
Number of	$10^9 \sim 10^{10}$	$10^8 \sim 10^{10}$

Fluid (3D space)



Ishizawa et al.

Kinetic (3D space + 2D vspace)

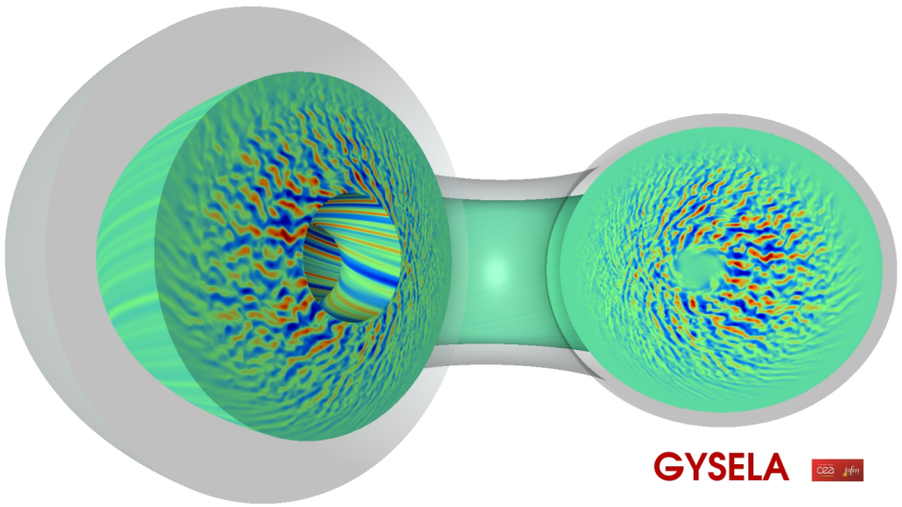


Idomura et al.

- **Highly nested** loop (4 or 5 dimensional)
- Relatively **low loop counts** in each dimension
- Many combinations of parallelization: collapse, SIMD, unroll

GYSELA code

Physics



- Modeling ITG turbulence in Tokamak
- Solving **5D** Vlasov + 3D Poisson eqs.

Gyrokinetic equation: Solve f

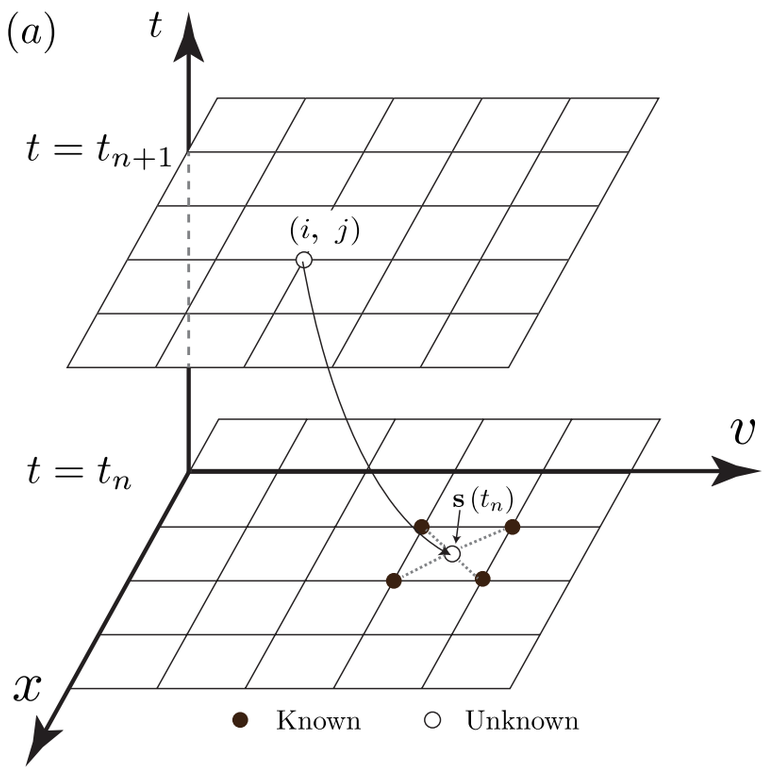
$$\partial_t f - [H, f] = C + S + K$$

C **: collision** S **: source** K **: sink**

Poisson equation: Solve electric field

$$-\nabla_\perp \cdot (P_1 \nabla_\perp \phi) + P_2 (\phi - \langle \phi \rangle) = \rho[f]$$

Numerics



- **Semi-Lagrangian** scheme to solve Vlasov eq.
- Interpolation of footpoints: Spline/Lagrange
- Parallelisation: MPI + OpenMP
- 3D domain decomposition by MPI

$$N_{\mathrm{MPI}} = p_r \times p_\theta \times N_\mu$$

- Good scalability up to 450 kcores
- More than 50k lines in Fortran 90

[1] V. Grandgirard, et al, CPC (2016)

Aim: explore performance portable implementation with miniapp

Requirements

- Productivity: Easy to modify and maintenance
- Readability: Easy to read for developers from many different fields
- Portability: A single code runs on **many different devices**
- High performance: Good performance on a given device

Possible approaches

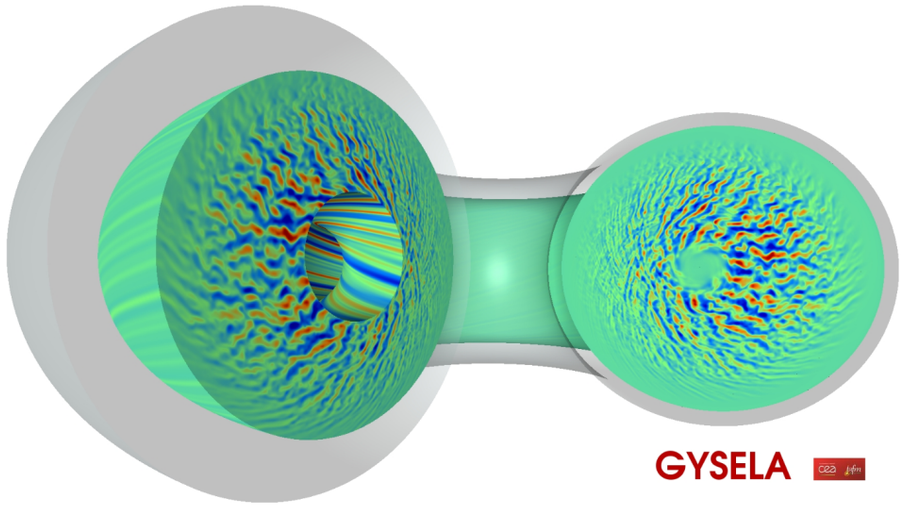
- Directive based approach: OpenMP, **OpenACC**
- Higher level abstraction: **Kokkos**, RAJA, Alpaka

Methodology

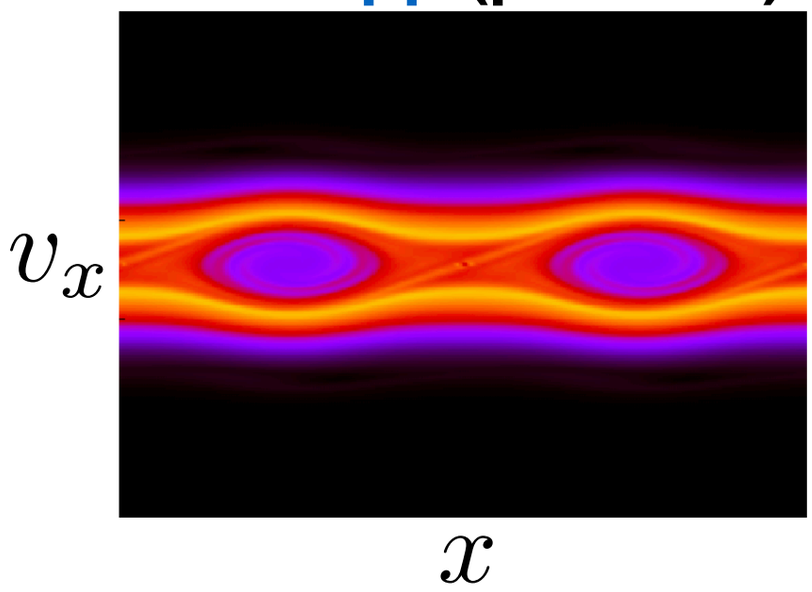
- Directive based and abstraction based implementation of miniapp
- Explore performance portable implementation over different devices: **Nvidia GPU**, **Intel CPU**, **ARM CPU**

Encapsulate key GYSELA features into mini-app

GYSELA (3D torus) $(r, \theta, \phi, v_{\|}, \mu)$



Mini-app (periodic) (x, y, v_x, v_y)



	GYSELA	Mini-app
System	5D Vlasov + 3D Poisson	4D Vlasov + 2D Poisson
Geometry	Realistic tokamak geoemtry	Periodic boundary conditions
Scheme	Semi-Lagrangian + Operator splitting (2D + 1D + 1D)	Semi-Lagrangian + Operator splitting (1D + 1D + 1D + 1D)
MPI	Yes	**No**
X	OpenMP	**OpenACC/OpenMP/Kokkos**
Language	Fortran 90	**C++**
Lines of codes	More than 50k	About 5k

- Extract the **Semi-Lagrangian + operator splitting** strategy for Vlasov solver
- Geometry and boundary conditions are simplified

Testbed description

	P100	SKL	ARM
Processor	NVIDIA Tesla P100 (Pascal)	Intel Xeon Gold 6148 (Skylake)	Marvell Thunder X2 (ARMv8)
Number of cores	1792 (DP)	20	32
L2 Cache [MB]	4	45	32
GFlops (DP)	**5300**	**1536**	**563.2**
Peak B/W [GB/s]	732	127.97	170.69
STREAM B/W [GB/s]	**540**	**80**	**120**
SIMD width	**-**	**512 bit**	**128 bit**
B/F ratio	0.138	0.083	0.30
TDP [W]	300	145	180
Compiler	cuda/8.0.61, pgi19.1	intel19.0.0.117	armclang++ 19.2.0
Compiler options	-ta=nvidia:cc60 -O3	-xCORE-AVX512 -O3	-std=C++11 -O3

- Relatively low B/F ratio, suitable for compute intense kernels
- Huge **diversity** in terms of L2 Cache, number of cores, B/W, GFLops
- Different compilers, careful compiler option settings needed for porting

Kernel description

Metric	Advect (x)	Advect (y)	Advect (vx)	Advect (vx)	Integral
Memory accesses	1 load + 1 store	1 load + 1 store	1 load + 1 store	1 load + 1 store	1 load
Access pattern	Indirect access along x direction	Indirect access along y direction	Indirect access along vx direction	Indirect access along vy direction	Reduction by row (along vx and vy)
Flop/Byte (f/b)	67/16	67/16	65/16	65/16	1/8

4D advection with Strang splitting [1]

$$\frac{\partial f}{\partial t} + v_x \frac{\partial f}{\partial x} = 0 \text{ at } (y, v_x, v_y) \text{ fixed}$$

$$\frac{\partial f}{\partial t} + v_y \frac{\partial f}{\partial y} = 0 \text{ at } (x, v_x, v_y) \text{ fixed}$$

$$\frac{\partial f}{\partial t} + E_x \frac{\partial f}{\partial v_x} = 0 \text{ at } (x, y, v_y) \text{ fixed}$$

$$\frac{\partial f}{\partial t} + E_y \frac{\partial f}{\partial v_y} = 0 \text{ at } (x, y, v_x) \text{ fixed}$$

Velocity space integral (4D to 2D) appeared in Poisson equation

$$\rho(t, \mathbf{x}) = \int d\mathbf{v} f(t, \mathbf{x}, \mathbf{v})$$

[1] G. Strang, et al, SIAM Journal on Numerical analysis (1968)

- More than **95%** of the costs are coming from these 5 kernels
- Advection kernels are almost identical but the performance is quite different particularly on CPUs due to **cache** and **vectorization** effects
- Integral kernel **reduces** a **4D** array into a **2D** array (reduction by row)

Baseline OpenMP implementation

```
#pragma omp for schedule(static) collapse(2)
for(int ivy = 0; ivy < nvy; ++ivy) {
  for(int ivx = 0; ivx < nvx; ++ivx) {
    const float64 vx = vx_min + ivx * dvx;
    const float64 depx = dt * vx;
    for(int iy = 0; iy < ny; ++iy) {
      for(int ix = 0; ix < nx; ++ix) {
        const float64 x = x_min + ix * dx;
        const float64 xstar = x_min + fmod(Lx + x - depx - x_min, Lx);
        int ipos1 = floor((xstar - x_min) * inv_dx);
        const float64 d_prev1 = LAG_OFFSET
                              + inv_dx * (xstar - (x_min + ipos1 * dx));
        ipos1 -= LAG_OFFSET;
        float64 coef[LAG_PTS];
        lag_basis(d_prev1, coef);
        float64 ftmp = 0.;
        for(int k = 0; k <= LAG_ORDER; k++)
          ftmp += coef[k] * fn[ivy][ivx][iy][(nx + ipos1 + k) % nx];
        fnp1[ivy][ivx][iy][ix] = ftmp;
      }
    }
  }
}
```

Langrange interpolation with degree of 5

load: fn, load/store: fnp1 $f/b = 67\text{flop}/16\text{bytes}$

- Relatively high compute intensity: $f/b \sim 4$
- OpenMP parallelization applied to the outermost loops (collapsed by 2)
- Bottlenecked with **indirect memory accesses**: load from fn

Outline

Introduction: Demands of acceleration in GK codes

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapp
- Aim and setting of this research

Kokkos and OpenACC/OpenMP versions of mini-app

- Higher level abstraction in kokkos: memory and operation
- Mixed OpenACC/OpenMP implementation

Performance measurement and optimization

- Performance improvement with 3D Range policy in Kokkos
- Detailed analysis of kernels based on Roofline model
- Readability, Performance portability, Productivity in each implementation

Summary and future work

Kokkos introduction: abstraction

Execution patterns: Types of parallel operations

`Kokkos::parallel_for`
`Kokkos::parallel_reduce`
`Kokkos::parallel_scan`

Execution space: Where the operations performed

`GPUs or CPUs`

Execution policy: How the operation is performed

`RangePolicy, TeamPolicy`

Example: parallel reduction (operation defined by user)

```
struct squaresum {
  // Specify the type of the reduction value with a "value_type"
  // typedef.  In this case, the reduction value has type int.
  typedef int value_type;

  KOKKOS_INLINE_FUNCTION
  void operator () (const int i, int& lsum) const {
    lsum += i*i; // compute the sum of squares
  }
};

Kokkos::parallel_reduce (n, squaresum (), sum);
```

From tutorial

Abstract memory management: view

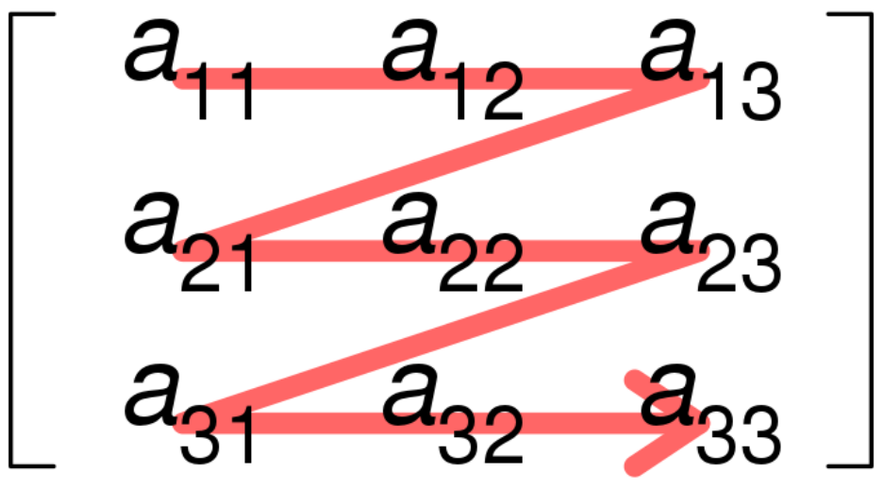
Layout Right (C style)

- **Default style for OpenMP background**

```
#pragma omp parallel for
for(int i=0; i<3; i++) {
  for(int j=0; j<3; j++) {
    a(i,j) = ...
  }
}
```

Contiguous along “j” (SIMD)

Row-major order



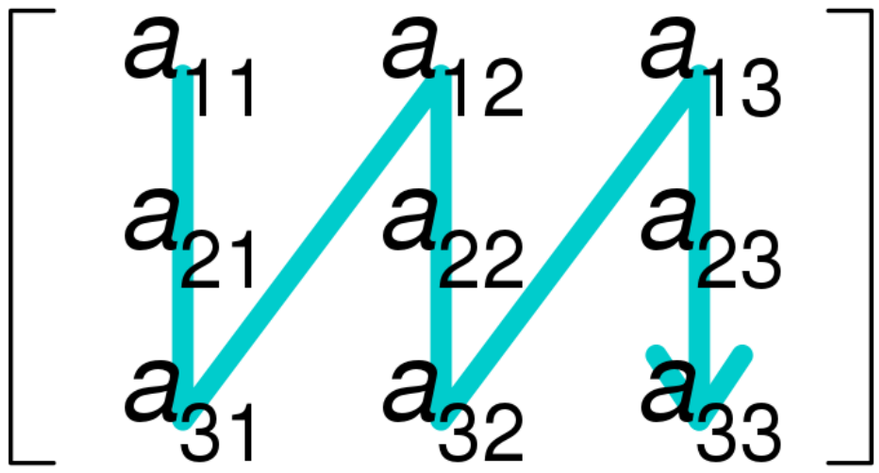
Layout Left (Fortran style)

- **Default style for CUDA background**

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
for(int j=0; j<3; j++) {
  a(i,j) = ...
}
```

Contiguous along “i” (coalesced)

Column-major order



https://en.wikipedia.org/wiki/Row-_and_column-major_order

Kokkos 2D view: `a(i,j)`

Outermost independent loop preferable for OpenMP

Innermost independent loop preferable for CUDA

Kokkos implementation: 1D range

```
struct advect_1D_x_functor {
  Config* conf_;
  view_4d fn_, fnp1_;
  ...

  advect_1D_x_functor(Config*conf, const view_4d fn, view_4d fnp1, double dt)
    : conf_(conf), fn_(fn), fnp1_(fnp1), dt_(dt) {
    const Domain *dom = &(conf_->dom_);
    nx_ = dom->nx_;
    ...
  }

  KOKKOS_INLINE_FUNCTION
  void operator()( const int &i ) const {
    int4 idx_4D = Index::int2coord_4D(i, nx_, ny_, nvx_, nvy_);
    int ix = idx_4D.x, iy = idx_4D.y, ivx = idx_4D.z, ivy = idx_4D.w;
    // Compute Lagrange bases
    ...

    float64 ftmp = 0.;
    for(int k=0; k<=LAG_ORDER; k++) {
      int idx_ipos1 = (nx_ + ipos1 + k) % nx_;
      ftmp += coef[k] * fn_(idx_ipos1, iy, ivx, ivy);
    }
    fnp1_(ix, iy, ivx, ivy) = ftmp;
  }
}

Kokkos::parallel_for(nx*ny*nvx*nvy, advect_1D_x_functor(conf, fn, fnp1, dt));
```

Flatten 1D loop (manual unpacking)

- **Parallel execution of 1D advection with 1D range policy**

:Pattern **:Policy**

OpenACC implementation

```
float64 *dptr_fn   = fn.raw(); // Raw pointer to the 4D view fn
float64 *dptr_fnp1 = fnp1.raw();

const int n = nx * ny * nvx * nvy;
#pragma acc data present(dptr_fn[0:n],dptr_fnp1[0:n])
{
  #pragma acc parallel loop collapse(3)
  for(int ivy = 0; ivy < nvy; ivy++) {
    for(int ivx = 0; ivx < nvx; ivx++) {
      for(int iy = 0; iy < ny; iy++) {
        #pragma acc loop vector independent
        for(int ix = 0; ix < nx; ix++) {
          // Compute Lagrange bases
          ...
          float64 ftmp = 0.;
          for(int k=0; k<=LAG_ORDER; k++) {
            int idx_ipos1 = (nx + ipos1 + k) % nx;
            int idx = idx_ipos1 + iy*nx + ivx*nx*ny + ivy*nx*ny*nvx;
            ftmp += coef[k] * dptr_fn[idx];
          }
          int idx = ix + iy*nx + ivx*nx*ny + ivy*nx*ny*nvx;
          dptr_fnp1[idx] = ftmp;
        }
      }
    }
  }
}
```

- **Loops collapsed by 3 and vectorized (innermost)**
- **Using 1D flatten index and raw pointer (avoid using in-house data structure)**

Mixed OpenACC/OpenMP implementation

```
#if defined( ENABLE_OPENACC )
  #pragma acc data present(dptr_rho,dptr_ex,dptr_ey,...)
  {
#endif
    #if defined( ENABLE_OPENACC )
      #pragma acc host_data use_device(dptr_rho, dptr_rho_hat)
    #endif
    fft_->rfft2(dptr_rho, dptr_rho_hat);
    #if defined( ENABLE_OPENACC )
      #pragma acc parallel loop
    #else
      #pragma omp for schedule(static)
    #endif
    for(int ix1=0; ix1<nx1h; ix1++) {
      int idx = ix1; float64 kx = ix1 * kx0;
      dptr_ex_hat[idx]  = -kx * I * dptr_rho_hat[idx] * dptr_filter[ix1] / (nx1*nx2);
      /* Similar computations ... */
    }
    #if defined( ENABLE_OPENACC )
      #pragma acc host_data use_device(dptr_rho,...)
      {
    #endif
        fft_->irfft2(dptr_rho_hat, dptr_rho);
        // Inverse FFTs for dptr_Ex_hat and dptr_Ey_hat
    #if defined( ENABLE_OPENACC )
      }
    #endif
#if defined( ENABLE_OPENACC )
  }
#endif
```

- **Macro heavy implementation or code duplications**
- **Macro free implementation is also difficult to follow**

Outline

Introduction: Demands of acceleration in GK codes

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapp
- Aim and setting of this research

Kokkos and OpenACC/OpenMP versions of mini-app

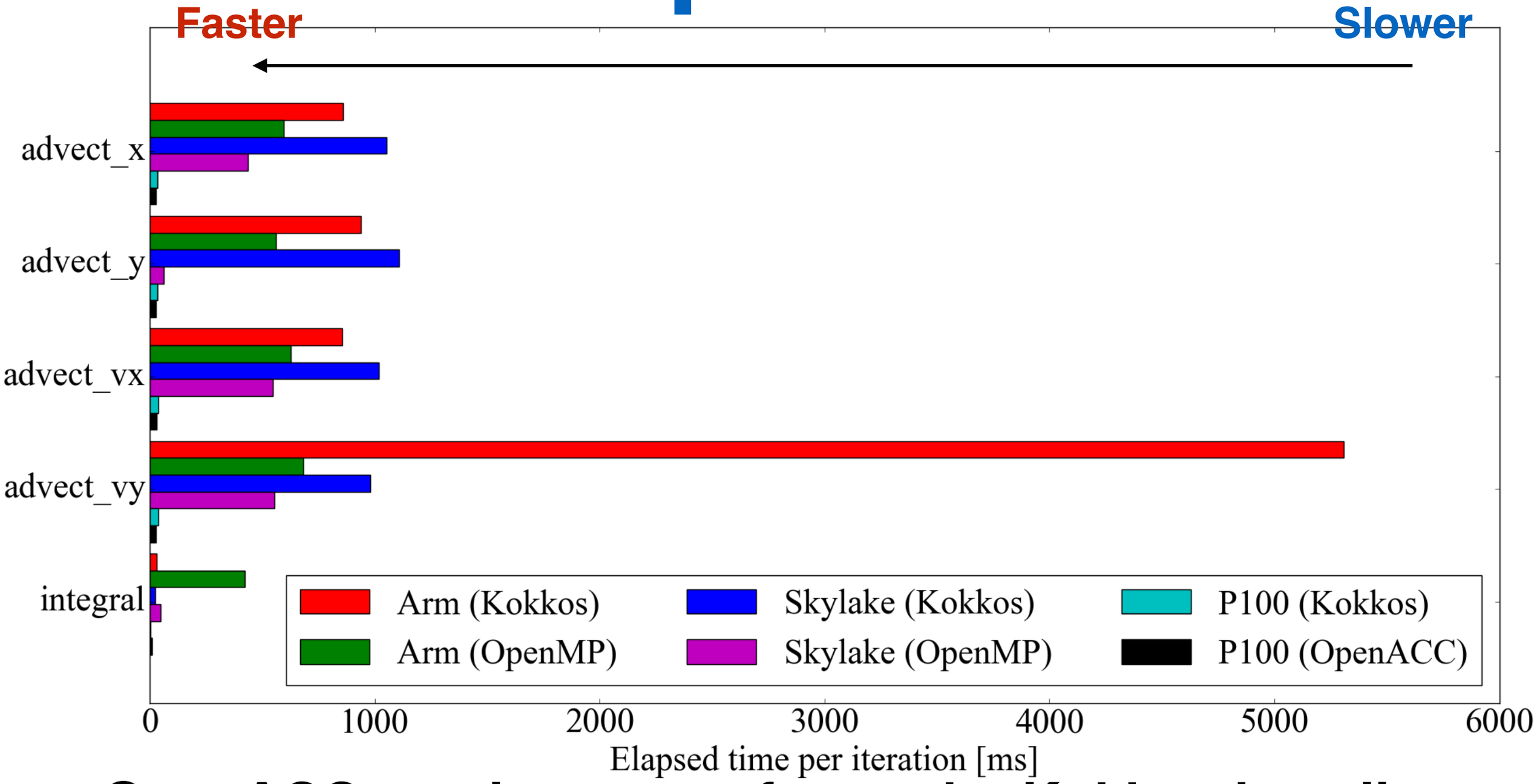
- Higher level abstraction in kokkos: memory and operation
- Mixed OpenACC/OpenMP implementation

Performance measurement and optimization

- Performance improvement with 3D Range policy in Kokkos
- Detailed analysis of kernels based on Roofline model
- Readability, Performance portability, Productivity in each implementation

Summary and future work

Baseline performance



- **OpenACC version outperforms the Kokkos baseline**
- **Unvectorized advection (vy) in Arm (kokkos) slowest**
- **Kokkos Skylake/Arm performance unsatisfactory**

High dimensional loop support: 3D range policy

```
struct advect_1D_x_functor {
  Config* conf_;
  view_4d fn_, fnp1_;
  ...

  advect_1D_x_functor(Config*conf, const view_4d fn, view_4d fnp1, double dt)
    : conf_(conf), fn_(fn), fnp1_(fnp1), dt_(dt) {
    ...
  }

  KOKKOS_INLINE_FUNCTION
  void operator()(const int ix, const int iy, const int ivx)  const {
    // Compute Lagrange bases
    ...
    for(int ivy=0; ivy<nvy; ivy++) {
      float64 ftmp = 0.;
      for(int k=0; k<=LAG_ORDER; k++) {
        int idx_ipos1 = (nx_ + ipos1 + k) % nx_;
        ftmp += coef[k] * fn_(idx_ipos1, iy, ivx, ivy);
      }
      fnp1_(ix, iy, ivx, ivy) = ftmp;
    }
  }
}

typedef typename Kokkos::Experimental::MDRangePolicy< Kokkos::Experimental::Rank<
3, Kokkos::Experimental::Iterate::Default, Kokkos::Experimental::Iterate::Default>
> MDPolicyType_3D;

MDPolicyType_3D mdpolicy_3d( {{0,0,0}}, {{nx,ny,nvx}}, {{TX,TY,TZ}} );
Kokkos::parallel_for( mdpolicy_3d, advect_1D_x_functor(conf, fn, fnp1, dt) );
```

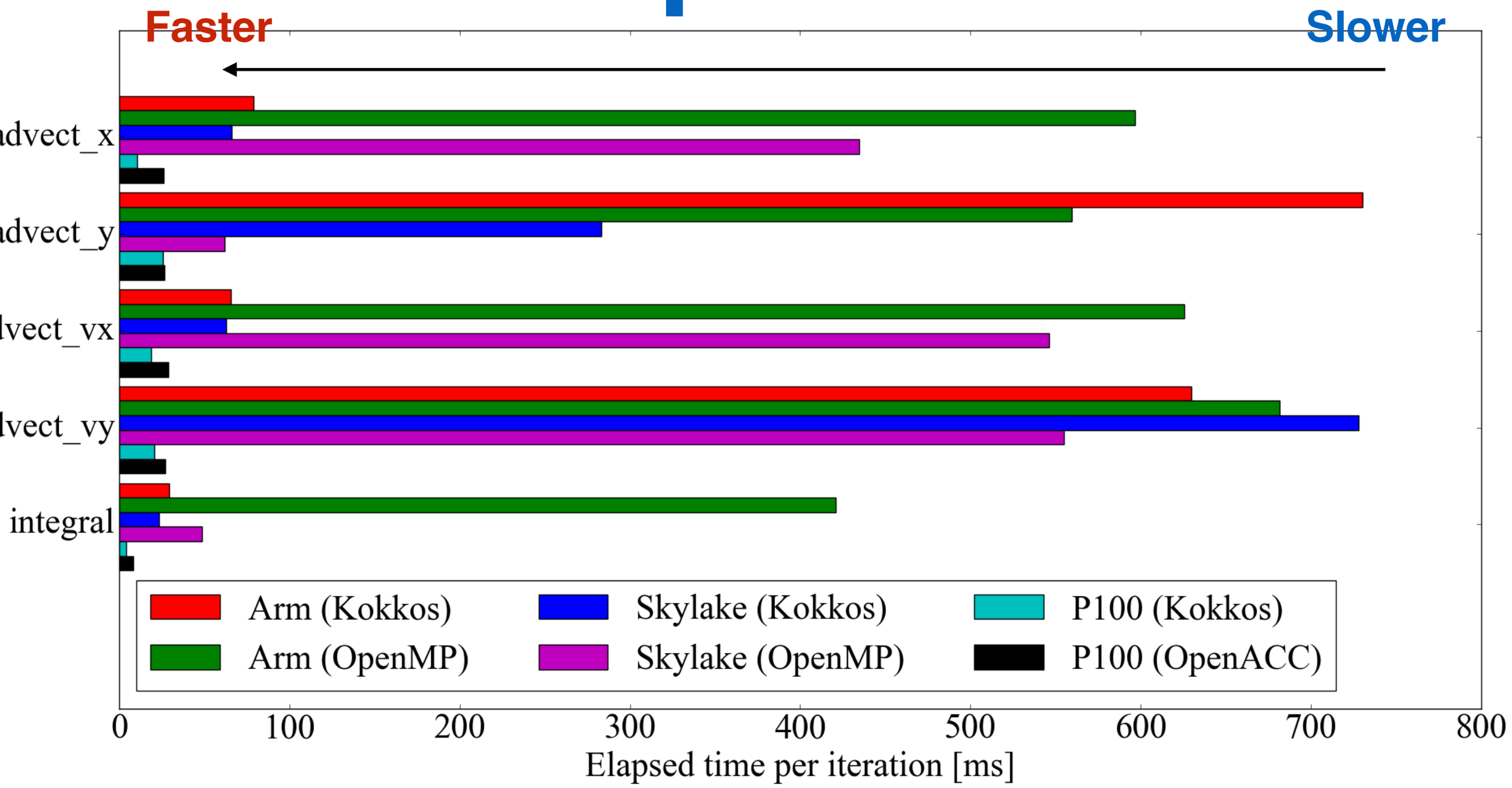
3D indices

3D tiling

- **3D policy facilitates SIMD on CPUs and cache on GPUs**

:Pattern **:Policy**

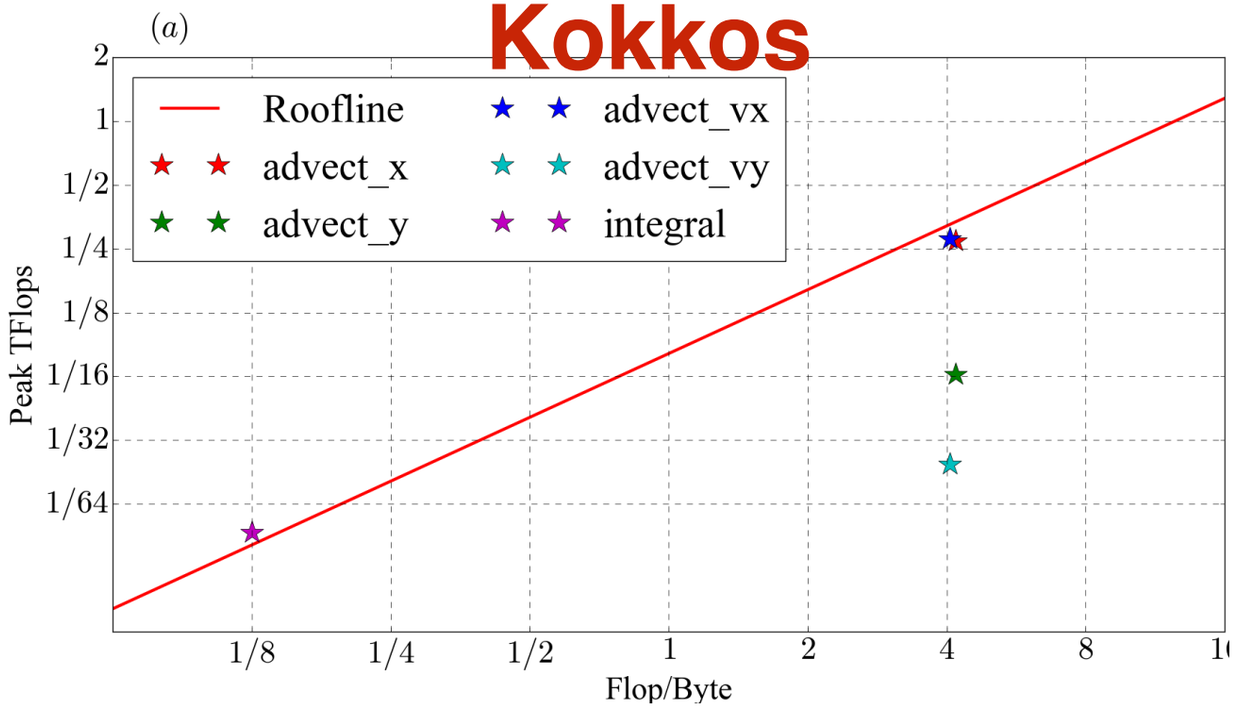
Achieved performance

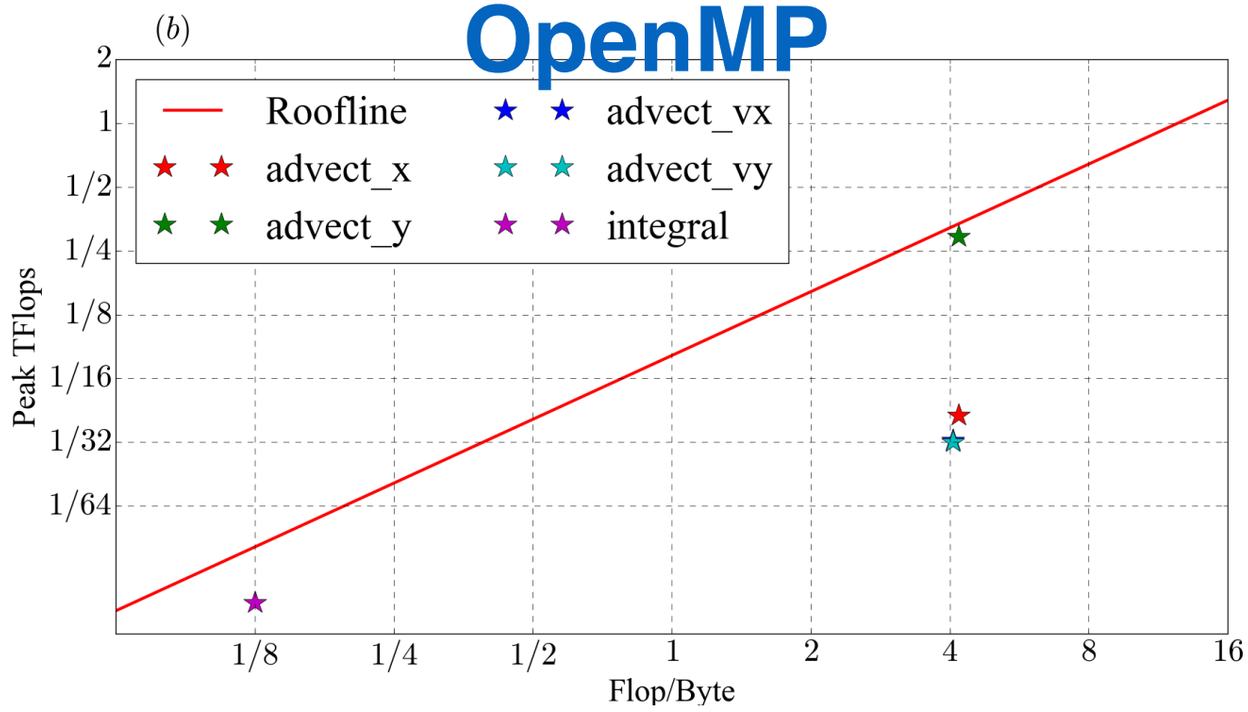


- **In our high dimensional loop with low loop counts 3D MD range policy (3D tiling) improves performance (worse performance reported for 3D MHD code in [1])**

[1] Grete, et al, CoRR (2019)

Kokkos vs OpenMP (Skylake)





- **Performance evaluated based upon Roofline model [1]**

$$\text{Attainable GFlops/s} = \min(F, B \times f/b)$$

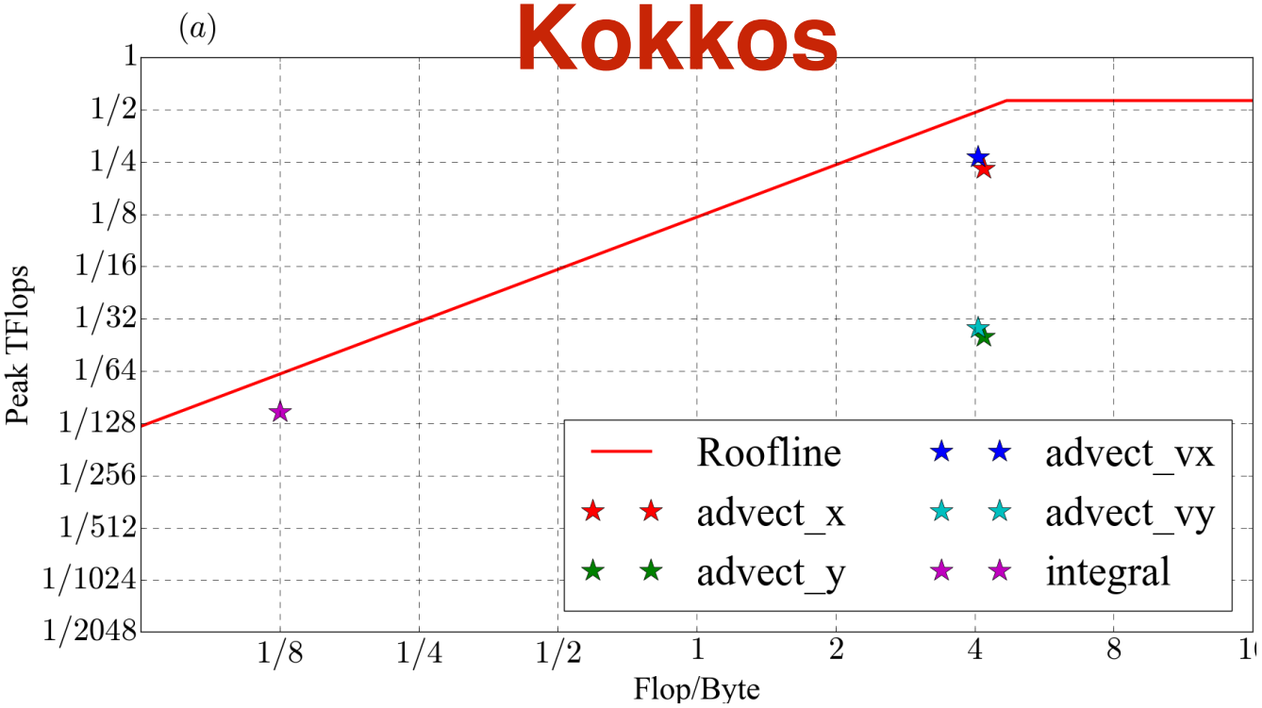
- **Good performance with advection (x/vx) for Kokkos, advection (y) for OpenMP (cache + vectorization)**

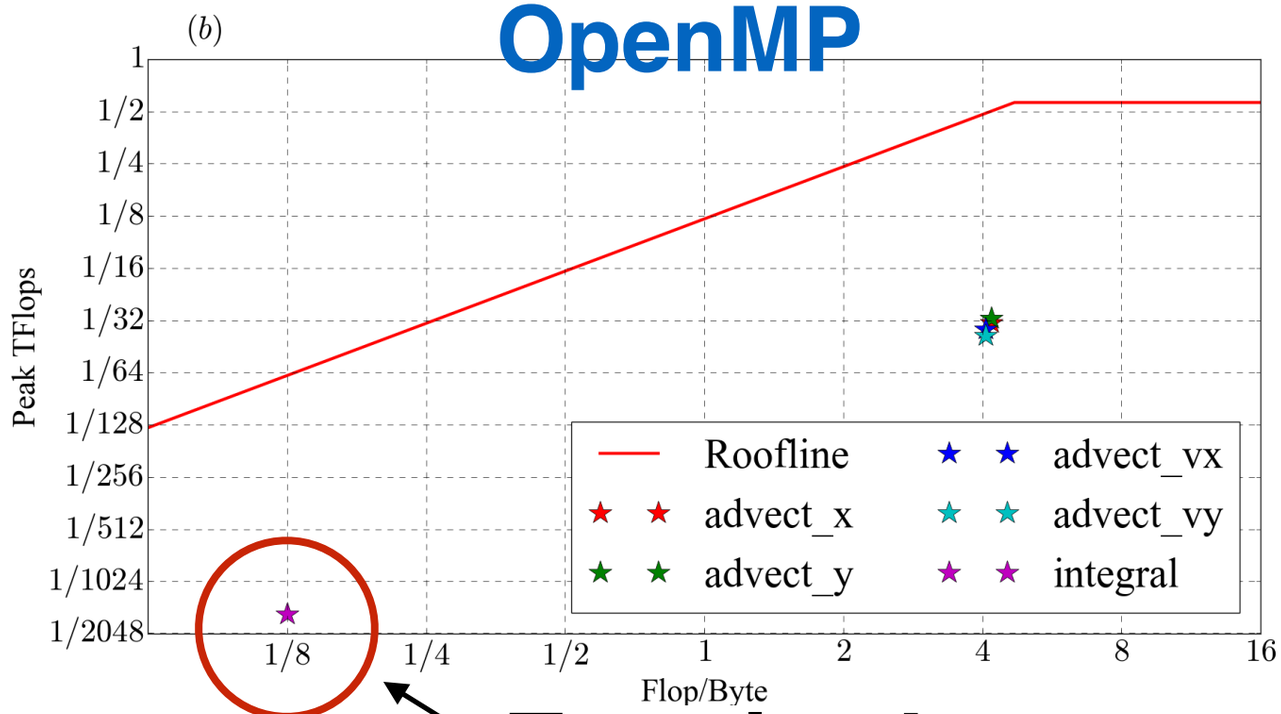
  ⟶ **Second innermost direction is the best**

- **OpenMP atomic operation for reducing 4D to 2D array harms the performance (integral)**

[1] S. Williams, et al," Commun. ACM, (2009).

Kokkos vs OpenMP (Arm)





- **Good performance with advection (x/vx) for Kokkos, but poor performance with advection (y) for OpenMP**

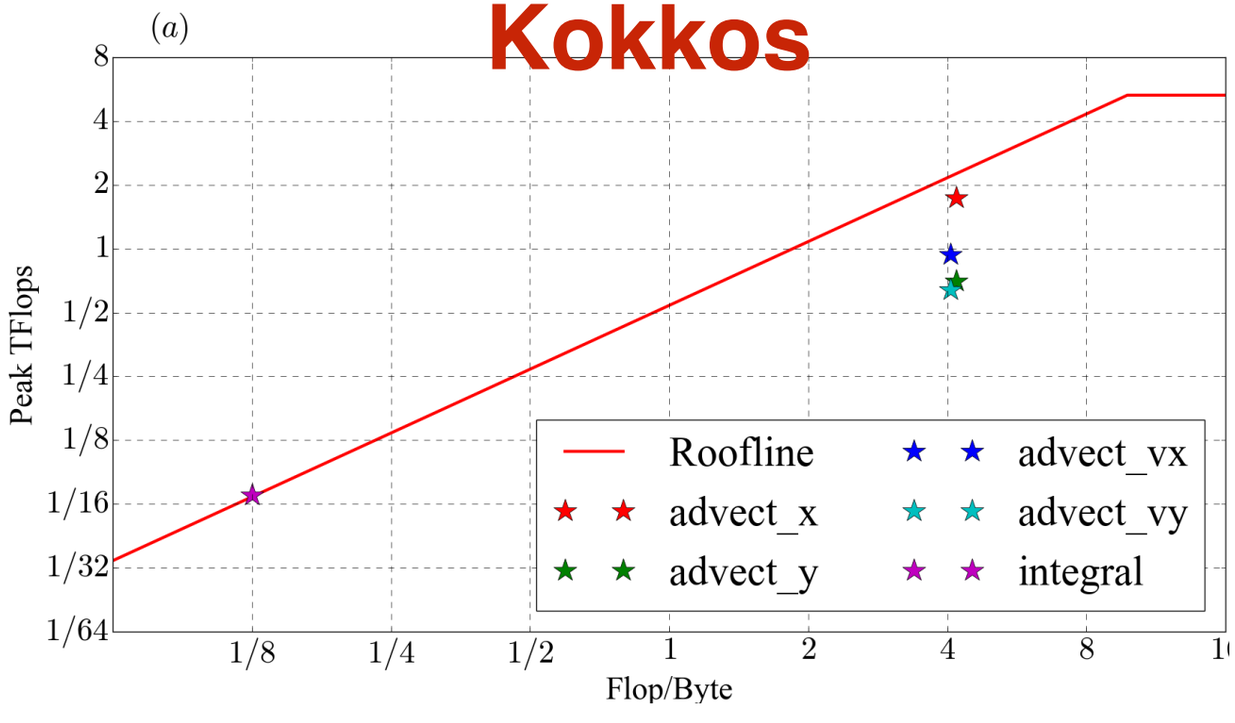
  ⟶ **Difference in cache behavior**

- **OpenMP atomic operation for reducing 4D to 2D array shows terrible performance (integral)**

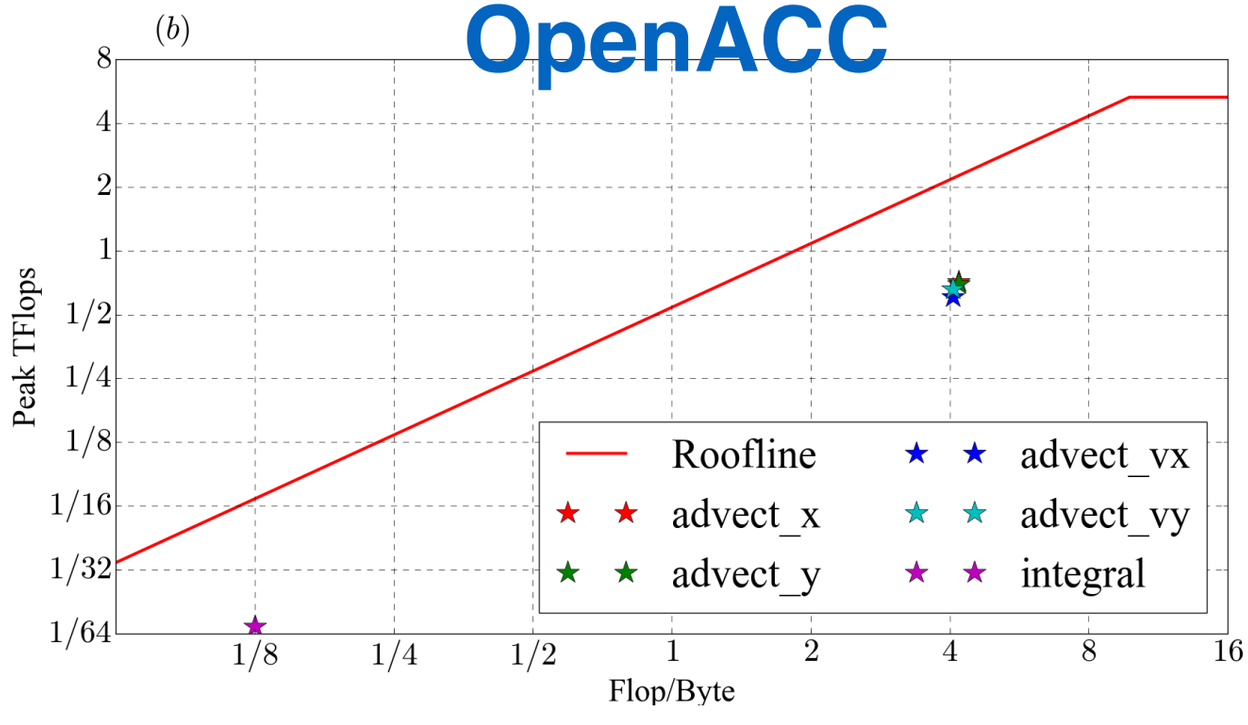
  ⟶ **Alternative implementation needed**

Kokkos vs OpenACC (P100)

Kokkos



OpenACC



nvprof metrics	Advect (x)	Advect (y)	Advect (vx)	Advect (vy)
Hit ratio for global loads in l1/tex cache [%]	96.67/96.46	61.93/0.0	44.86/0.37	42.14/2.46
Achieved occupancy	0.28/0.55	0.29/0.55	0.28/0.55	0.28/0.54
l2_write_throughput [GB/s]	162.6/90.1	57.0/88.3	108.9/78.2	51.72/85.3
l2_read_throughput [GB/s]	162.6/79.9	130.1/466.9	385.2/479.7	241.5/526.2

- **Higher L2 throughput in Kokkos version for advection along x direction (innermost)**
- **Kokkos uses L1 cache more likely than OpenACC**

Achieved performance

Device	Kernel	f/b	Ideal Gflops	Achieved performance	
				GFlops	GB/s (relative to STREAM %)
Skylake (Kokkos/OpenMP)	Advect (x)	67/16	335	271.7/41.8	64.9 (81.1%)/9.98 (12.5%)
	Advect (y)	67/16	335	63.5/291.1	15.2 (19.0%)/69.51 (86.9%)
	Advect (vx)	65/16	325	278.5/31.94	68.6 (85.7%)/7.86 (9.8%)
	Advect (vy)	65/16	325	24/31.5	5.9 (7.4%)/7.74 (9.6%)
	Integral	1/8	10	11.4/5.5	91.6 (114 %)/43.7 (54.7%)
Arm (Kokkos/OpenMP)	Advect (x)	67/16	502.5	228.0/30.1	54.4 (45.4%)/7.20 (6.0%)
	Advect (y)	67/16	502.5	24.6/32.1	5.88 (4.9%)/6.40 (6.4%)
	Advect (vx)	65/16	487.5	266.6/27.9	65.6 (54.9%)/6.86 (5.7%)
	Advect (vy)	65/16	487.5	27.7/25.6	6.82 (5.7%)/6.30 (5.3%)
	Integral	1/8	15	9.1/0.63	72.8 (60.7%)/5.06 (4.2%)
P100 (Kokkos/OpenACC)	Advect (x)	67/16	2261.3	1739.9/710.8	415.0 (76.9%)/169.8 (31.4%)
	Advect (y)	67/16	2261.3	704.4/695.6	168.2 (31.1%)/166.1 (30.8%)
	Advect (vx)	65/16	2193.8	935.7/605.2	230.3 (42.7%)/149.0 (27.6%)
	Advect (vy)	65/16	2193.8	638.6/657.5	157.2 (29.1%)/161.8 (30.0%)
	Integral	1/8	67.5	68.8/16.9	550.0 (101.9%)/134.9 (25.0%)

- **Some kernels achieved almost ideal performance**

Readability, Portability, Productivity

	OpenACC/ OpenMP	Kokkos
Readability	Medium	High
Portability	High	High
Performance	High	High
Productivity	Medium	Low

	Time [s]	Speedup
Skylake (OpenMP)	278	1.0
Skylake (Kokkos)	192	1.45
Arm (OpenMP)	589	0.47
Arm (Kokkos)	335	0.83
P100 (OpenACC)	21.5	12.95
P100 (Kokkos)	15.6	17.83

Speedup relative to SKL (OpenMP)

- **Readability**

  OpenACC/OpenMP: multiple macros harm readability

  Kokkos: easy to read, but hard to understand what is exactly done

- **Productivity/Portability**

  OpenACC/OpenMP: reasonable solution to port large Fortran codes

  Kokkos: large porting costs for Fortran code (less costly for C++), maintenance costs may be suppressed

Outline

Introduction: Demands of acceleration in GK codes

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapp
- Aim and setting of this research

Kokkos and OpenACC/OpenMP versions of mini-app

- Higher level abstraction in kokkos: memory and operation
- Mixed OpenACC/OpenMP implementation

Performance measurement and optimization

- Performance improvement with 3D Range policy in Kokkos
- Detailed analysis of kernels based on Roofline model
- Readability, Performance portability, Productivity in each implementation

Summary and future work

Summary and future works

Directive based approach: mixed OpenACC/OpenMP

- Mixed OpenACC/OpenMP achieves high performance except for ARM
- Suitable for **porting a large legacy code** (e.g. more than 50k LoCs)
- Mixed approach harms the readability due to **multiple macros**
- Insufficient performance in some kernels due to lack of memory abstraction

Higher level abstraction: Kokkos

- Kokkos can achieve **performance portability over several devices**
- Appropriate choice of a **range policy** seems critical for CPUs
- Kokkos requires **large initial investments** but may suppress maintenance costs due to a **good readability** and **abstraction**

Future tasks

- MPI parallelization of mini-app and test scalability

[Y. Asahi et al., submitted to waccpd SC19)]

Backup slides

Remarks for OpenACC implementation

Using external library (e.g. cufft) from accelerated region

```
void Efield::solve_poisson_fftw(float64 xmax, float64 ymax)
```

```
88    // Use local pointer to avoid issues with "use_device" [See Known limitations 3.5]
89    float64 *dptr_rho = rho_.raw(), *dptr_ex = ex_.raw();
90    float64 *dptr_ey = ey_.raw(), *dptr_filter = filter_.raw();
91    complex64 *dptr_rho_hat = rho_hat_.raw(), *dptr_ex_hat = ex_hat_.raw(), *dptr_ey_hat = ey_hat_.raw();
92    #pragma acc data present(dptr_rho,dptr_ex,dptr_ey,dptr_rho_hat,dptr_ex_hat,dptr_ey_hat,dptr_filter)
93    {
94      // Forward 2D FFT (Real to Complex)
95      #pragma acc host_data use_device(dptr_rho, dptr_rho_hat)
96      fft_->fft2(dptr_rho, dptr_rho_hat);
97    }
```

- **Issue with “use_device" for class members**
  “use_device” does not work for class members
  [See Known limitations 3.5] https://www.pgroup.com/resources/docs/19.1/x86/openacc-gs/index.htm

- **Issue with cuda streams**
  cuda stream used for cufft (default stream) and openacc is not identical
  Set OpenACC stream on cufft

```
121       // Force cuFFT on OpenACC stream
    https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/openacc/5-openacc-interoperability.pdf?__blob=publicationFile
122       cudaStream_t accStream = (cudaStream_t) acc_get_cuda_stream(acc_async_sync);
123       cufftSetStream(forward_plan_, accStream);
124       cufftSetStream(backward_plan_, accStream);
```

Using C++ class with OpenACC

- **Shallow and deep copies needed for class members**

```
45    #pragma acc enter data copyin(this) // shallow copy
46    #pragma acc enter data create(dptr_rho[0:n1],dptr_ex[0:n1],dptr_ey[0:n1],dptr_phi[0:n1]) // deep copy data members
47    #pragma acc enter data create(dptr_rho_hat[0:n2],dptr_ex_hat[0:n2],dptr_ey_hat[0:n2],dptr_filter[0:nx1h]) // deep copy data members
48    #pragma acc update device(dptr_filter[0:nx1h]) // update filter
```

Macro free OpenACC + OpenMP implementation

```
#pragma acc data present(dptr_fn[0:n2],dptr_rho[0:n1],dptr_ex[0:n1],dptr_ey[0:n1],dptr_phi[0:n1])
{
  #pragma acc parallel loop collapse(4)
  #pragma omp for schedule(static)
  for(int ivy=0; ivy<nvy; ivy++) {
    for(int ivx=0; ivx<nvx; ivx++) {
      for(int iy=0; iy<ny; iy++) {
        for(int ix=0; ix<nx; ix++) {
          int idx_src = Index::coord_4D2int(ix,iy,ivx,ivy,nx,ny,nvx,nvy);
          int idx_dst = Index::coord_2D2int(ix,iy,nx,ny);
          #pragma acc atomic update
          #pragma omp atomic
          dptr_rho[idx_dst] += dptr_fn[idx_src];
        }
      }
    }
  }
}
```

- **The code works with naively inserting directives**
- **It seems less clear how the parallelization performed with each directive**

Kokkos implementation when calling libraries

```
fft_->rfft2(rho_.ptr_on_device(), rho_hat_.ptr_on_device());

complex_view_2d ex_hat = ex_hat_, ey_hat = ey_hat_, rho_hat = rho_hat_;
view_1d         filter = filter_;
float64 normcoeff = 1./(nx*ny);
Kokkos::parallel_for(nx1h, KOKKOS_LAMBDA (const int ix1) {
  float64 kx = ix1 * kx0;
  {
    int ix2 = 0;
    ex_hat(ix1, ix2) = -kx * I * rho_hat(ix1, ix2) * filter(ix1) * normcoeff;
    ey_hat(ix1, ix2) = 0.;
    rho_hat(ix1, ix2) = rho_hat(ix1, ix2) * filter(ix1) * normcoeff;
  }
  for(int ix2=1; ix2<nx2h; ix2++) {
    float64 ky = ix2 * ky0; float64 k2 = kx * kx + ky * ky;
    ex_hat(ix1, ix2) = -(kx/k2) * I * rho_hat(ix1, ix2) * normcoeff;
    ey_hat(ix1, ix2) = -(ky/k2) * I * rho_hat(ix1, ix2) * normcoeff;
    rho_hat(ix1, ix2) = rho_hat(ix1, ix2) / k2 * normcoeff;
  }
  for(int ix2=nx2h; ix2<nx2; ix2++) {
    float64 ky = (ix2-nx2) * ky0; float64 k2 = kx*kx + ky*ky;
    ex_hat(ix1, ix2) = -(kx/k2) * I * rho_hat(ix1, ix2) * normcoeff;
    ey_hat(ix1, ix2) = -(ky/k2) * I * rho_hat(ix1, ix2) * normcoeff;
    rho_hat(ix1, ix2) = rho_hat(ix1, ix2) / k2 * normcoeff;
  }
});

fft_->irfft2(rho_hat.ptr_on_device(), rho_.ptr_on_device());
fft_->irfft2(ex_hat.ptr_on_device(),  ex_.ptr_on_device());
fft_->irfft2(ey_hat.ptr_on_device(),  ey_.ptr_on_device());
```

- **FFT class wraps 2D FFT based on cufft or fftw with OpenMP parallelization**

Remarks for Kokkos implementation

Kokkos view: multidimensional array with execution space

```
#include <Kokkos_Core.hpp>
#include <Kokkos_Complex.hpp>
typedef double float64;
typedef Kokkos::complex<double> complex64;
typedef Kokkos::DefaultExecutionSpace execution_space;
typedef Kokkos::View<float64*, execution_space> view_1d;
typedef Kokkos::View<float64**, execution_space> view_2d;
typedef Kokkos::View<float64****, execution_space> view_4d;
typedef Kokkos::View<complex64*, execution_space> complex_view_1d;
typedef Kokkos::View<complex64**, execution_space> complex_view_2d;
```

- **Complex data type support (works with reduction)**

KOKKOS_LAMABDA with C++ class

```
complex_view_2d ex_hat = ex_hat_, ey_hat = ey_hat_, rho_hat = rho_hat_;
view_1d         filter = filter_;
Kokkos::parallel_for(nx1h, KOKKOS_LAMBDA (const int ix1) {
  float64 kx = ix1 * kx0;
  {
    int ix2 = 0;
    ex_hat(ix1, ix2) = -kx * I * rho_hat(ix1, ix2) * filter(ix1) * normcoeff;
    ...
  }
```

- **Shallow copy to capture class members (nvcc 8.0)**