Performance portable implementation of a kinetic plasma simulation mini-app

Yuuichi ASAHI[1]

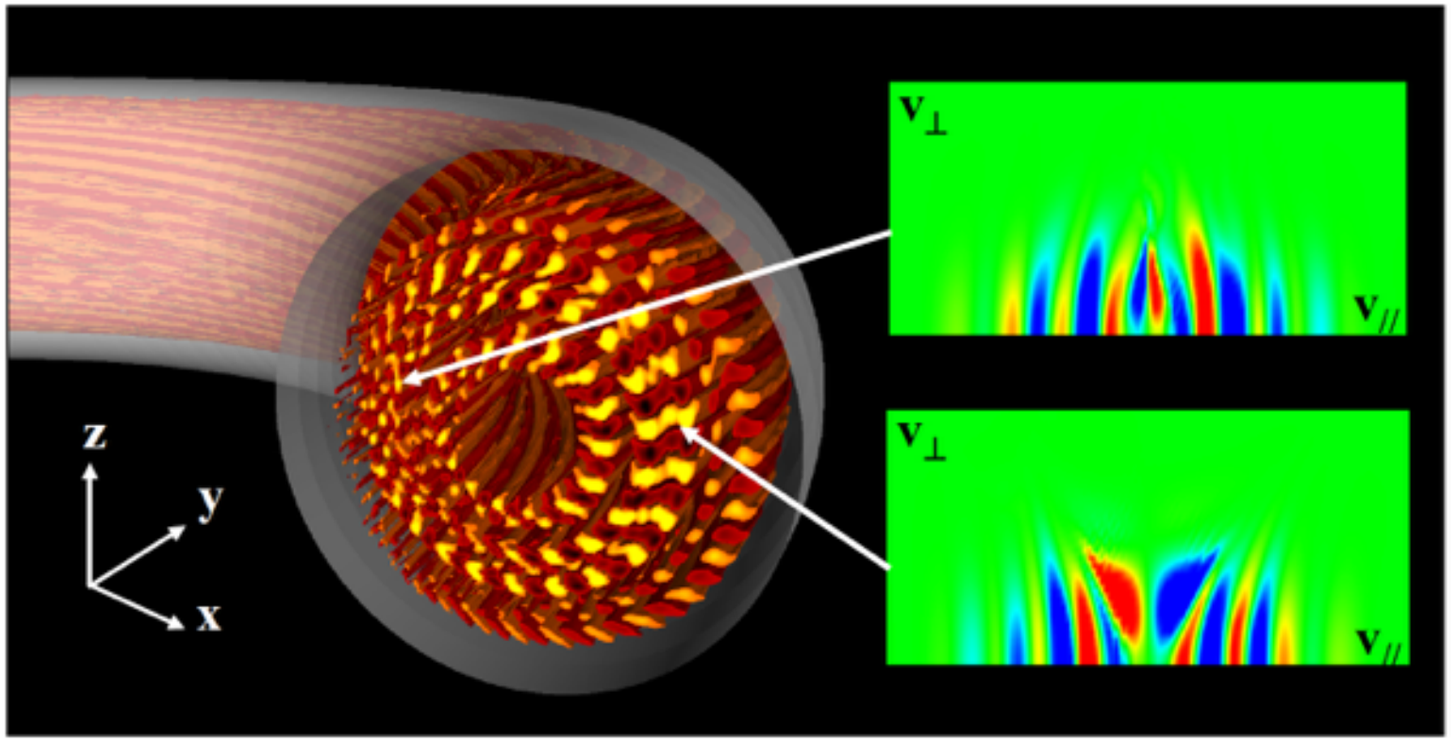
G. Latu[2], **V. Grandgirard**[2], **J. Bigot**[3]

[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: 18/November/2019

Sixth Workshop on Accelerator Programming Using Directives (WACCPD), Denver, US

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,** even more computational resource is needed
  - ⟶ **Accelerators** are key ingredients to satisfy huge computational demands at **reasonable energy consumption:** MPI + **'X'**

Outline

Introduction

- 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 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];
```

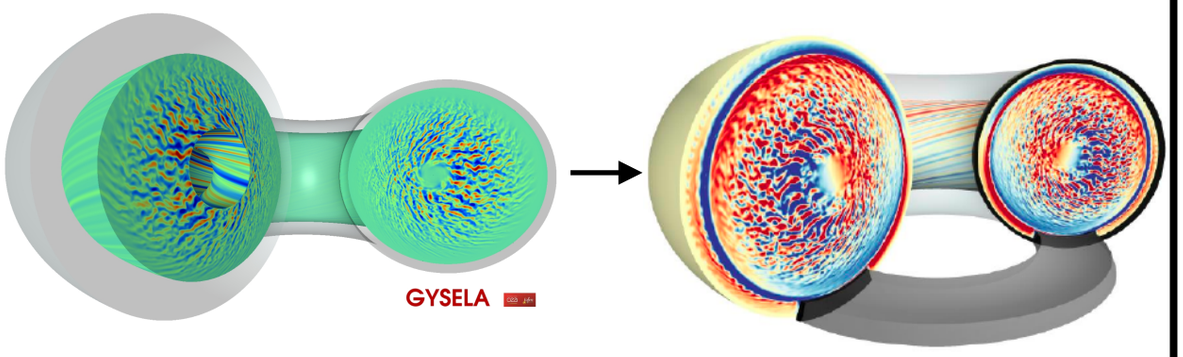
OpenACC

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

Readable for physicists

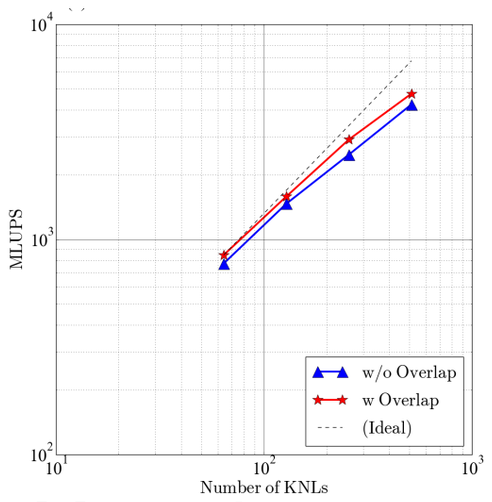
Productivity

Circular → **Limiter**



Advanced (realistic) physical model

High Performance



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

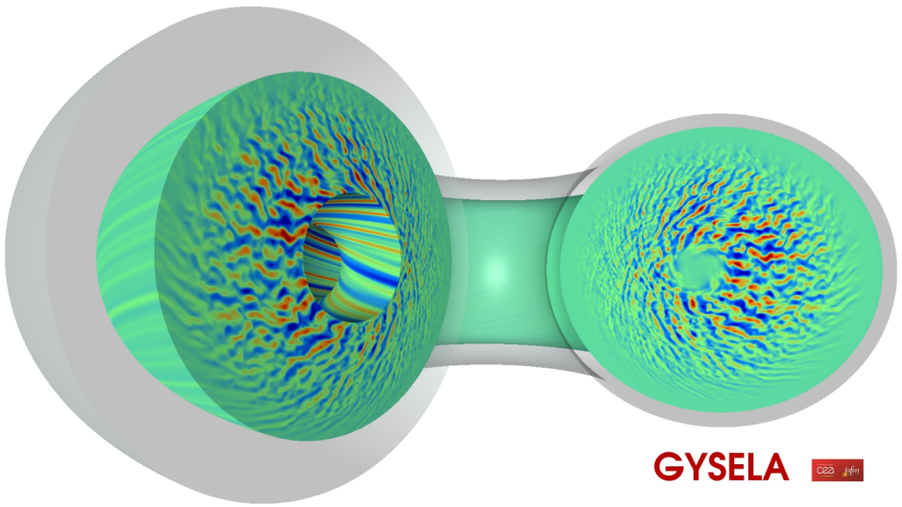
More than 100 M cpu hours/year

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/

GYSELA code

Physics



- Modeling Ion temperature gradient (ITG) turbulence in Tokamak
- Solving **5D** Vlasov + 3D Poisson eqs.

Gyrokinetic equation: Solve f

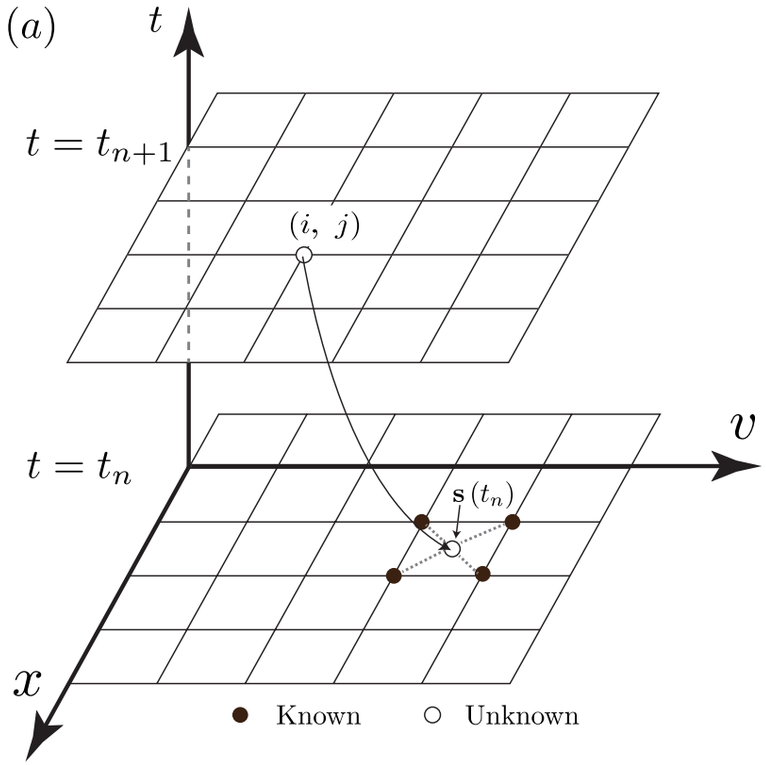
$$\partial_t f - [H, f] = C + S + K$$

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 the mini-app

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

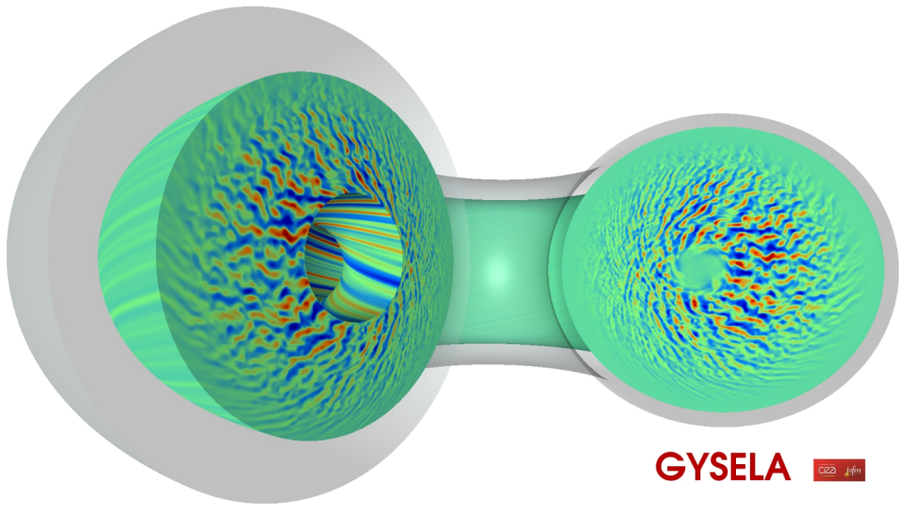
- Directive based approach: OpenMP, **OpenACC, OpenMP4.5**
- Higher level abstraction: **Kokkos**, RAJA, Alpaka

Methodology

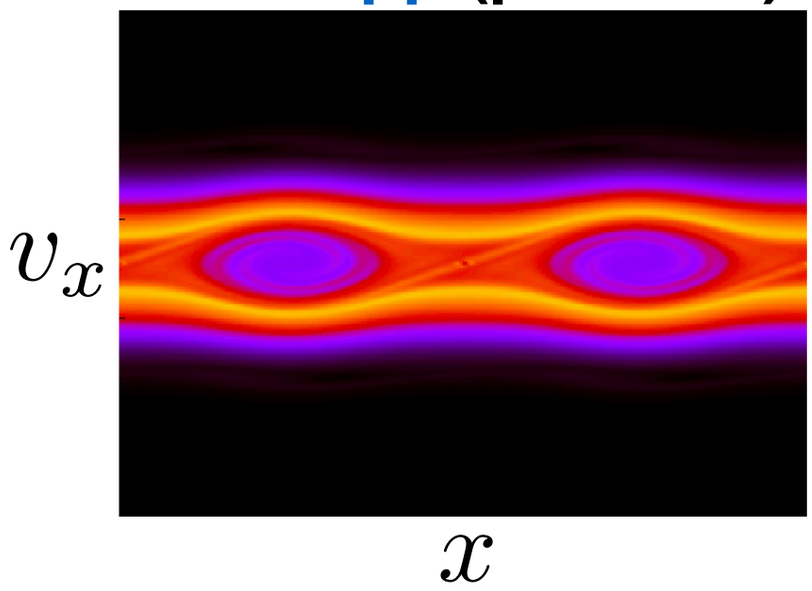
- Directive-based and abstraction-based implementation of mini-app
  Mixed OpenMP/OpenACC and Kokkos (minimize code duplication)
- Explore performance portable implementation over different devices:
  Nvidia GPUs, **Intel CPU**, **ARM CPU**

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	V100	Skylake	Arm
Processor	NVIDIA Tesla P100 (Pascal)	NVIDIA Tesla V100 (Volta)	Intel Xeon Gold 6148 (Skylake)	Marvell Thunder X2 (ARMv8)
Number of cores	1792 (DP)	2560 (DP)	20	32
L2/L3 Cache [MB]	4	6	27.5	32
GFlops (DP)	**5300**	**7800**	**1536**	**512**
Peak B/W [GB/s]	732	900	127.97	170.6
STREAM B/W [GB/s]	**540**	**830**	**80**	**120**
SIMD width	-	-	**512 bit**	**128 bit**
B/F ratio	0.138	0.115	0.083	0.332
TDP [W]	300	300	145	180
Manufacturing process	16 nm	12 nm	14 nm	16 nm
Year	2016	2017	2017	2018
Compiler	cuda/8.0.61, pgi19.1	cuda/10.1.168, pgi19.1	intel19.0.0.117	armclang 19.2.0
Compiler options	-ta=nvidia:cc60 -O3	-ta=nvidia:cc70 -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 (vy)	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**	Indirect access along **y**	Indirect access along **vx**	Indirect access along **vy**	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;
      }
    }
  }
}
```

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

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 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

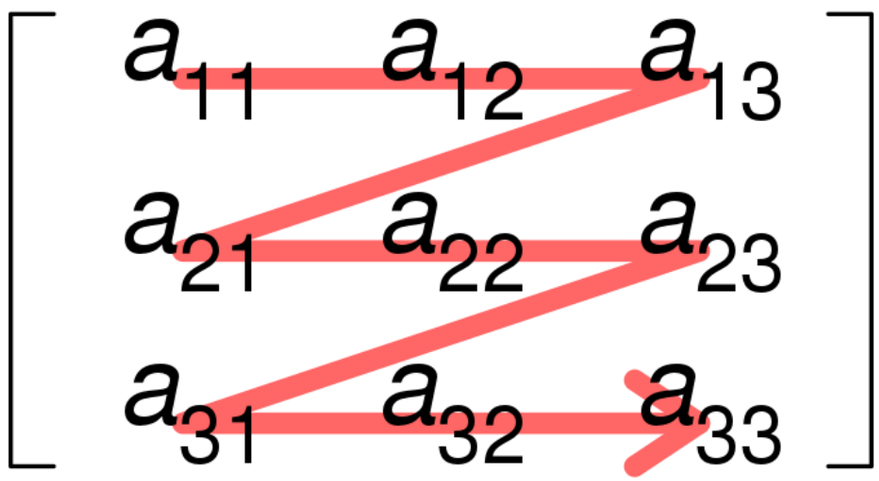
Layout Right (C style)

- **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



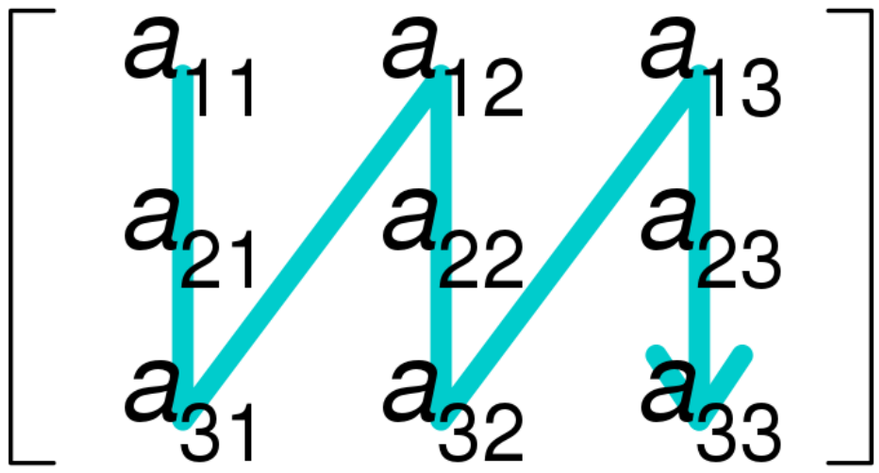
Layout Left (Fortran style)

- **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));
```

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

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;
        }
      }
    }
  }
}
```

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

- **Loops collapsed by 3 and vectorized (innermost)**
- **Using 1D flatten index and raw pointer (avoid using in-house data structure, i.e. simplified version of view)**

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
```

Loop parallelization

Wrapper to library (fftw, cufft)

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

Macro free OpenACC + OpenMP implementation (4D to 2D reduction, reduction by row)

```
const int n1 = nx * ny * nvx * nvy; const int n2 = nx * ny;

#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];
        }
      }
    }
  }
}
```

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

- The code works with naively inserting directives
- It seems less clear how the parallelization performed with each directive
- Due to the absence of memory abstraction, memory access patterns could be inappropriate for CPUs or GPUs.
- It may be worth considering to change the loop order.

Outline

Introduction

- 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

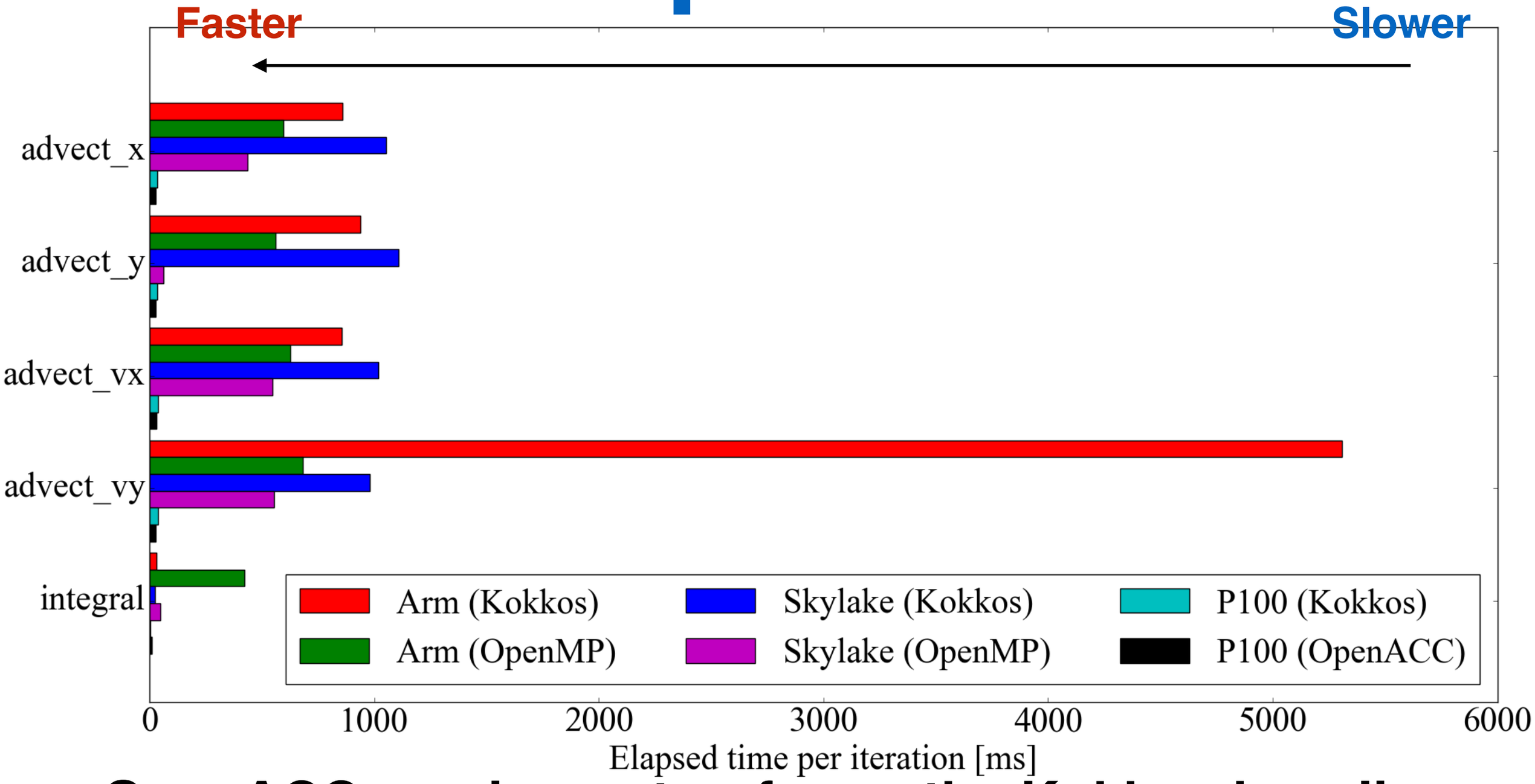
- Higher level abstraction in kokkos: memory and operation
- Mixed OpenACC/OpenMP implementation

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**

$(x, y, v_x, v_y) = 128^4$

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) );
```

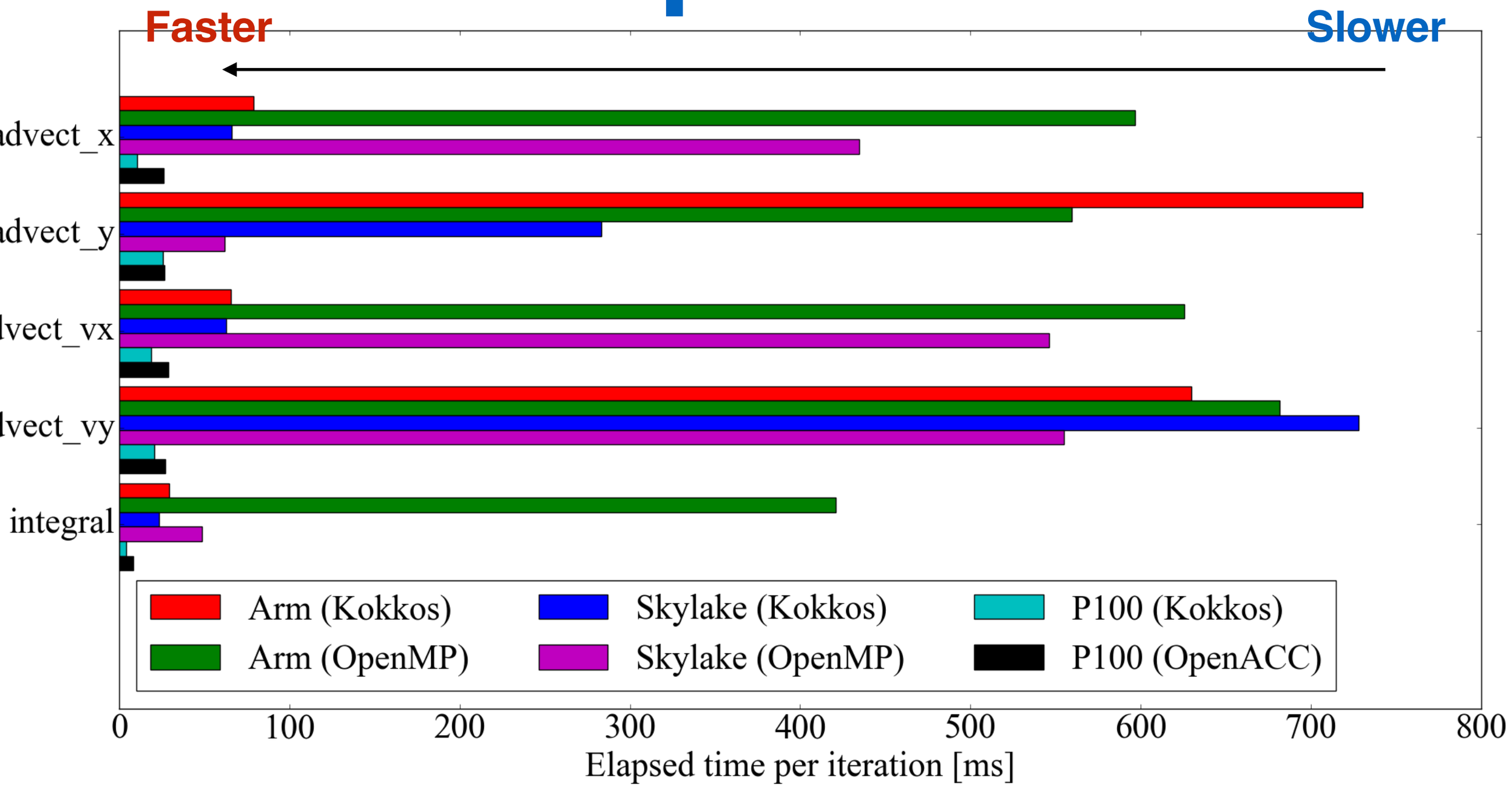
3D indices

3D tiling

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

:Pattern **:Policy**

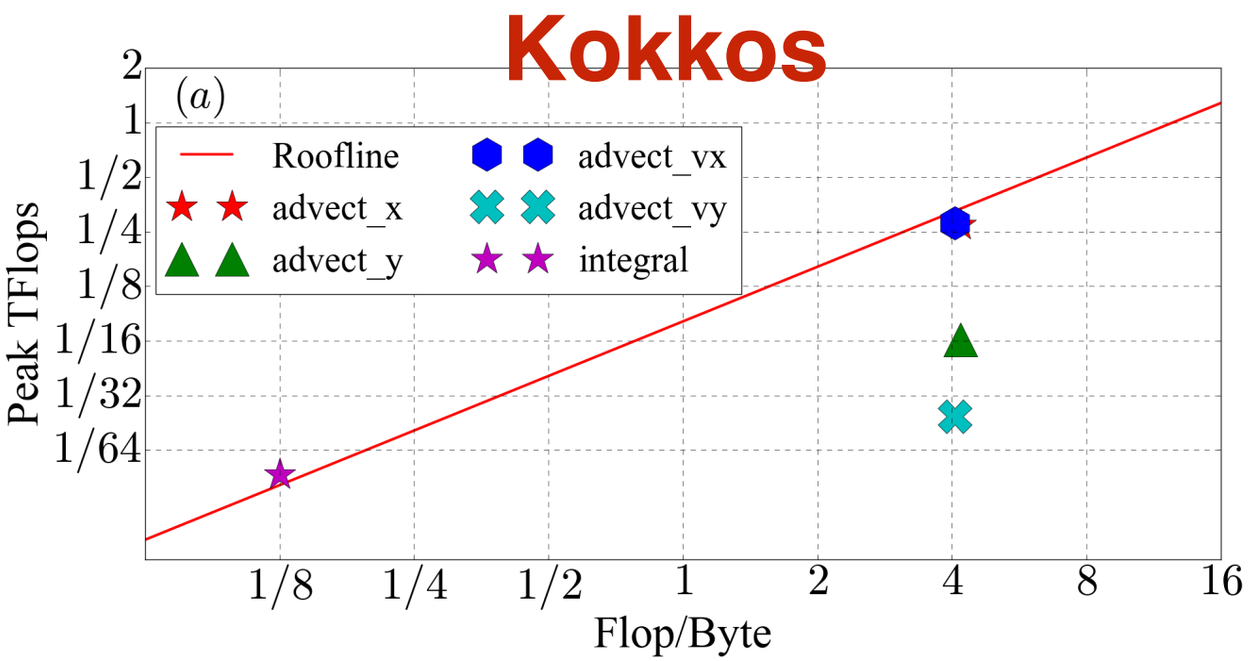
Achieved performance

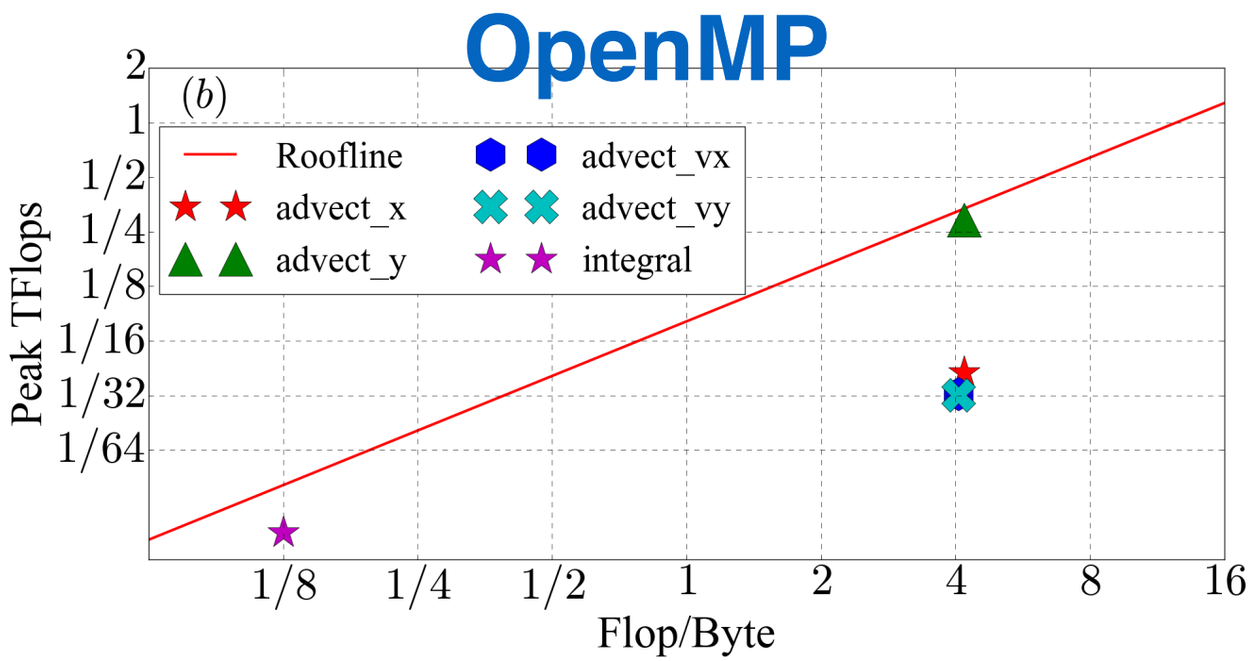


- **In our high dimensional loop with low loop counts 3D MD range policy (3D tiling) improves performance (worse performance reported for 3D MHD code in [1])**

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

Kokkos vs OpenMP (Skylake)





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

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

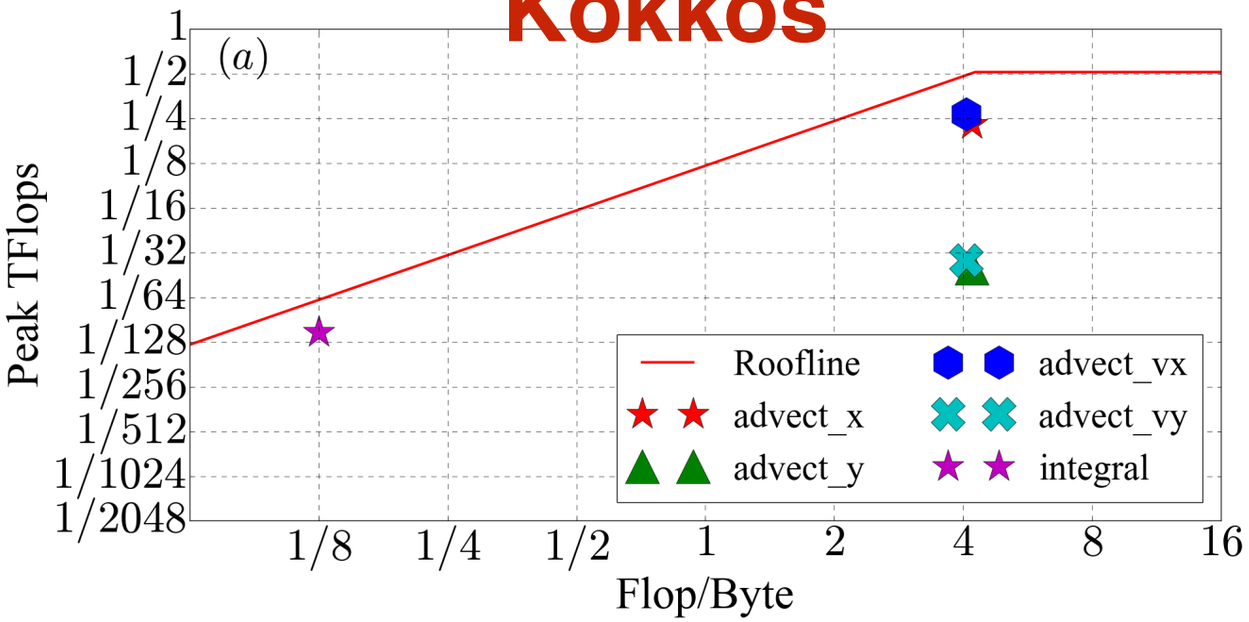
- **Good performance with advection (x/vx) for Kokkos, advection (y) for OpenMP (cache + vectorization)**

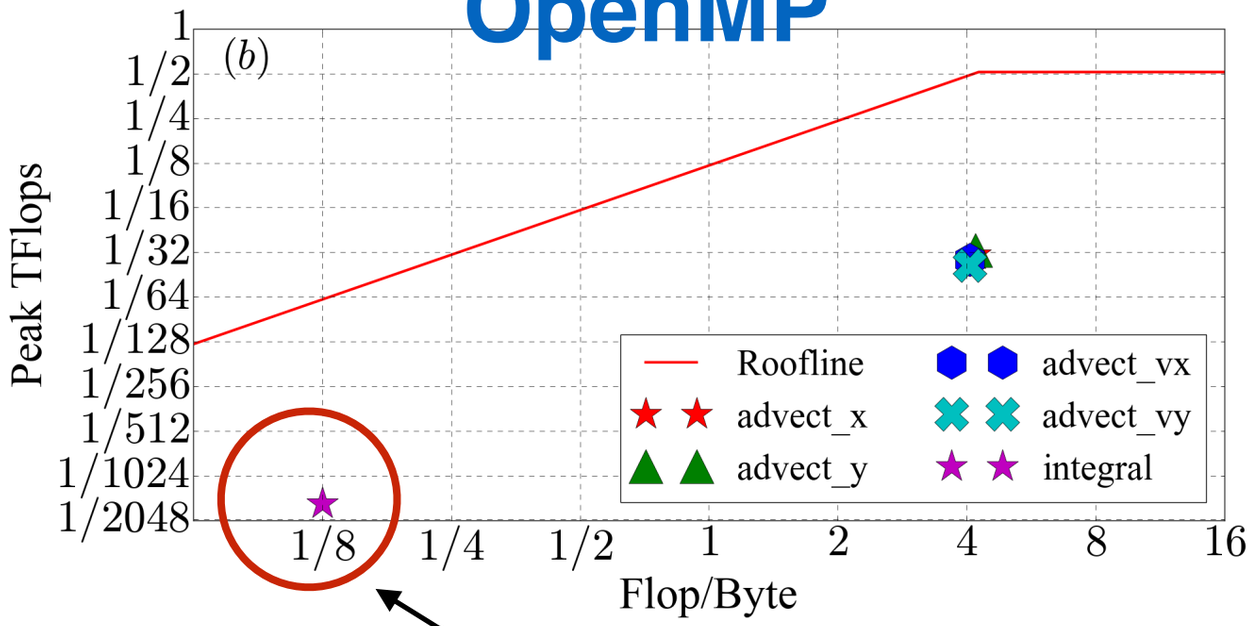
  ⟶ **Second innermost direction is the best**

- **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

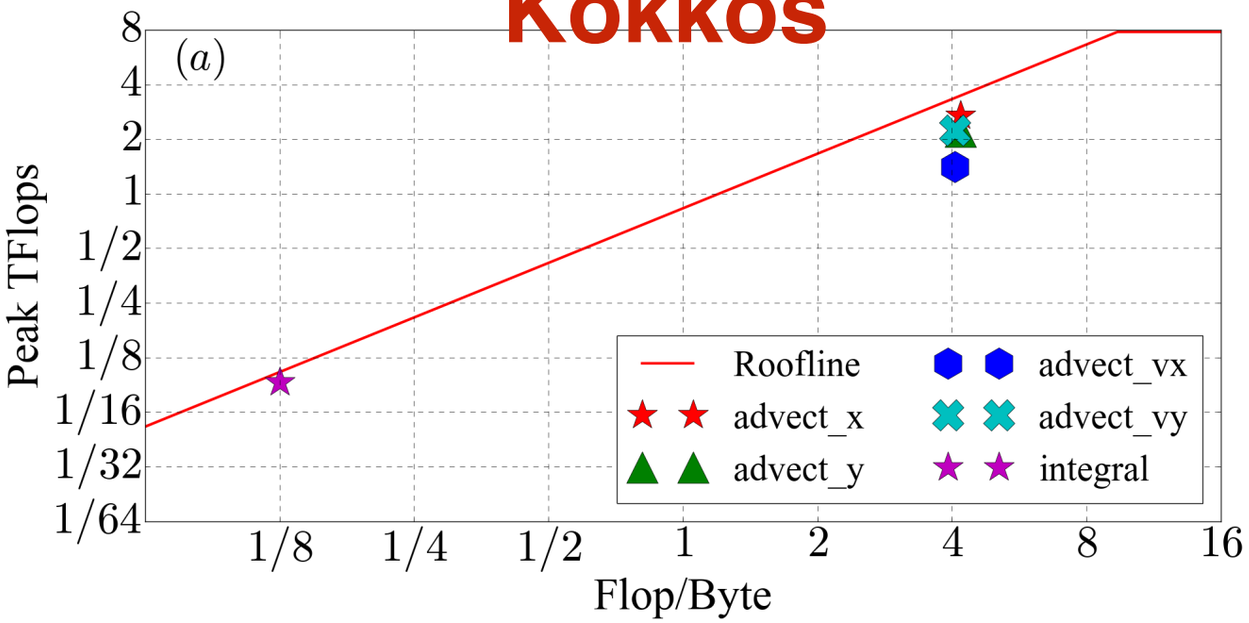
  ⟶ Difference in cache behavior

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

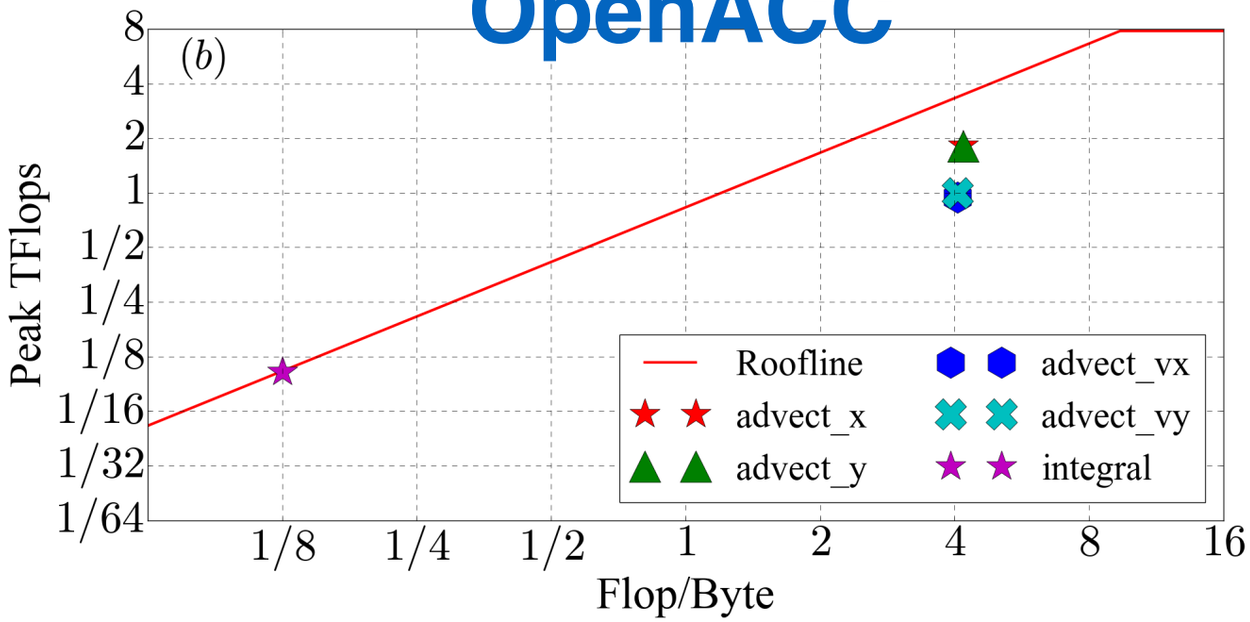
  ⟶ Alternative implementation needed

Kokkos vs OpenACC (V100)

Kokkos



OpenACC



nvprof metrics	Advect (x)	Advect (y)	Advect (vx)	Advect (vy)
Achieved GFlops	2701/1815	2205/1804	1404/946	2239/1001
Hit ratio for global loads in l1/tex cache [%]	70.35/73.81	53.63/0.06	34.87/4.76	70.69/3.48
Achieved occupancy	0.37/0.55	0.49/0.55	0.37/0.54	0.49/0.55
l2_write_throughput [GB/s]	299.9/183.5	221.7/182.5	163.32/108.8	226.4/115.3
l2_read_throughput [GB/s]	372.9/183.6	497.5/1094	582.5/720.64	319.2/795.3

- **Higher L1/L2 throughput in Kokkos version**
- **Roughly the same trend as P100, L2/L3 cache improved**

Improvement of V100 w.r.t P100

	P100	V100	Improvement
L2/L3 Cache [MB]	4	6	x 1.50
GFlops (DP)	5300	7800	x 1.47
STREAM B/W (Peak B/W) [GB/s]	540 (732)	830 (900)	x 1.54
Advections (x, y, vx, vy) in **OpenACC** [GFlops]	710.8, 695.6, 605.2, 657.5	1814.6, 1804.3, 946.1, 1001.2	x 2.55, **x 2.59** x 1.56, x 1.52
Reductions in **OpenACC** [GFlops]	16.9	102.5	**x 6.07**
Advections (x, y, vx, vy) in **kokkos** [GFlops]	1739.9, 704.4, 935.7, 638.6	2701.1, 2205.2, 1403.7, 2239.3	x 1.55, **x 3.13** x 1.50, **x 3.51**
Reductions in **kokkos** [GFlops]	68.8	90.9	x 1.32

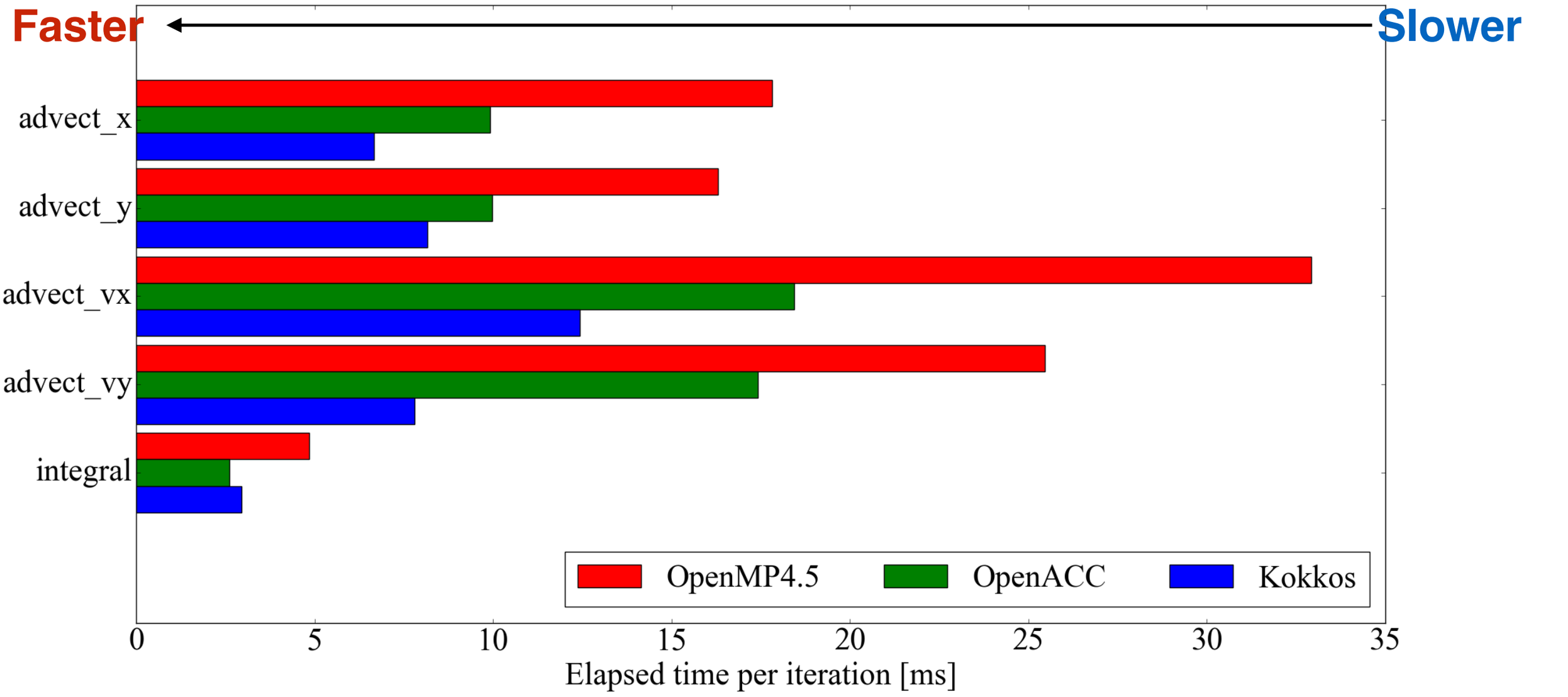
```
#pragma acc parallel loop collapse(4)
for(int ivy=0; ivy<nvy; ivy++) {
  for(int ivx=0; ivx<nvx; ivx++) {
    for(int iy=0; iy<ny; iy++) {
      for(int ix=0; ix<nx; ix++) {
        int idx_src = Index::coord_4D2int(ix,iy,ivx,ivy,nx,ny,nvx,nvy);
        int idx_dst = Index::coord_2D2int(ix,iy,nx,ny);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

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

- **Improved OpenACC performance of atomic op. on V100**

OpenMP4.5 experience (preliminary)

	Compiler	Compiler flags	STREAM TRIAD	Mini-app
OpenMP4.5	xl/16.1.1-3	-qsmp=omp -qoffload -Xptxas -v	**540 GB/s**	**16.9 [s]**
OpenACC	pgi19.1	-acc -ta=nvidia:cc70	830 GB/s	10 [s]
Kokkos	cuda/10.1.168	-Xcudafe -arch=sm_70	844 GB/s	6.8 [s]



- **Performance less impressive, but fantastic portability**

Achieved performance

Device	Kernel	f/b	Ideal GFlops	Achieved performance			
				GFlops		GB/s (relative to STREAM %)	
Skylake (Kokkos/ OpenMP)	Advect	67/16	335	271.7	41.8	64.9 (81.1%)	9.98 (12.5%)
	Advect	67/16	335	63.5	291.1	15.2 (19.0%)	69.51 (86.9%)
	Advect	65/16	325	278.5	31.94	68.6 (85.7%)	7.86 (9.8%)
	Advect	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	67/16	492.8	228.0	30.1	54.4 (45.4%)	7.20 (6.0%)
	Advect	67/16	492.8	24.6	32.1	5.88 (4.9%)	6.40 (6.4%)
	Advect	65/16	487.5	266.6	27.9	65.6 (54.9%)	6.86 (5.7%)
	Advect	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	67/16	2261.3	1739.9	710.8	415.0 (76.9%)	169.8 (31.4%)
	Advect	67/16	2261.3	704.4	695.6	168.2 (31.1%)	166.1 (30.8%)
	Advect	65/16	2193.8	935.7	605.2	230.3 (42.7%)	149.0 (27.6%)
	Advect	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%)
V100 (Kokkos/ OpenACC)	Advect	67/16	3475.6	2701.1	1814.6	645.0 (77.8%)	433.3(52.2%)
	Advect	67/16	3475.6	2205.2	1804.3	526.6 (63.4%)	430.9 (51.9%)
	Advect	65/16	3371.9	1403.7	946.1	345.5 (41.6%)	232.9 (28.1%)
	Advect	65/16	3371.9	2239.3	1001.2	551.2 (66.4%)	246.4 (29.7%)
	Integral	1/8	103.8	90.9	102.5	727.6 (87.7%)	820.0 (98.8%)

- **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	x 1.00
Skylake (Kokkos)	192	x 1.45
Arm (OpenMP)	589	x 0.47
Arm (Kokkos)	335	x 0.83
P100 (OpenACC)	21.5	x 12.9
P100 (Kokkos)	15.6	x 17.8
V100 (OpenMP4.5)	16.9	x 16.4
V100 (OpenACC)	10.0	x 27.8
V100 (Kokkos)	6.79	x 40.9

Speedup relative to SKL (OpenMP)

- **Readability**

  Directives: multiple macros harm readability

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

- **Productivity/Portability**

  Directives: 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 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 LoC)
- 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 an **execution 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