Accumulating knowledge for a performance portable kinetic plasma simulation code with Kokkos and directives

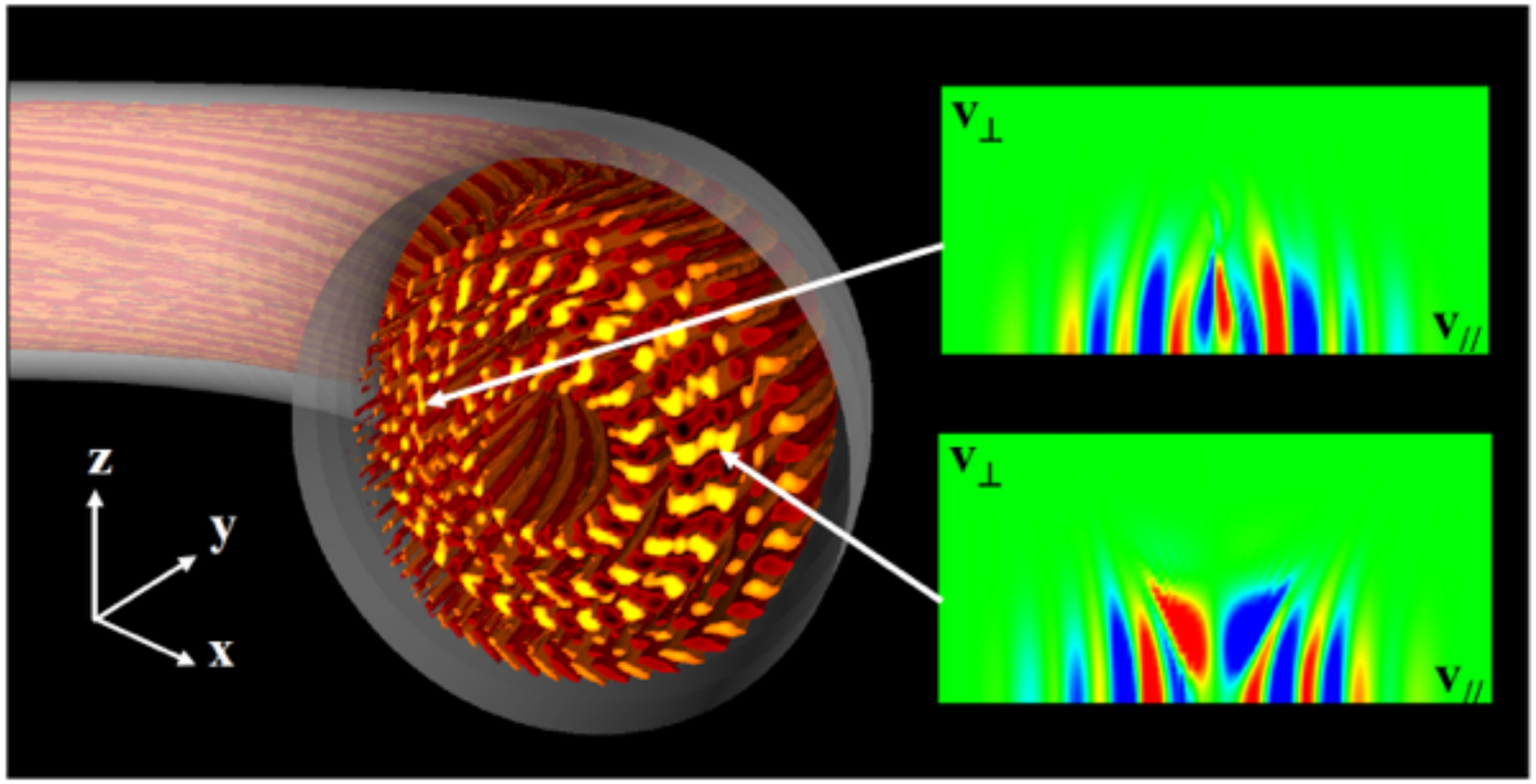
Yuuichi ASAHI[1]

G. Latu[2], V. Grandgirard[3], J. Bigot[4]

[1] JAEA, CCSE, Chiba, Japan
[2] DES/IRESNE/DEC, CEA, F-13108, St. Paul-lez-Durance cedex, France
[3] IRFM, CEA, F-13108, St. Paul-lez-Durance cedex, France
[4] Maison de la Simulation, CEA, CNRS, 91191 Gif-sur-Yvette, France

IFERC GPU workshop @ Zoom Date: 16/December/2020

Plasma turbulence simulation



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

→ **5D** stencil computations

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

First principle gyrokinetic model to predict plasma turbulence

- Confinement properties of fusion reactors (high temperature, non-Maxwellian)

Solving the machine scale problem (~m) with turbulence scale mesh (~cm)

- Degrees of freedom: $100^5 \sim 10^{10}$ Peta-scale supercomputing

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 miniapps
- 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

MPI parallelization of mini-app

- Algorithm update: Lagrange to Spline, MPI parallelization
- Optimization for OpenACC/OpenMP version with a new View class
- Optimization for Kokkos version with Layout and tile size tuning
- Performance and scalability

Summary and future work

Outline

Introduction

- Demands for MPI + 'X' for kinetic simulation codes
- Brief introduction of GYSELA code and miniapps
- 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

MPI parallelization of mini-app

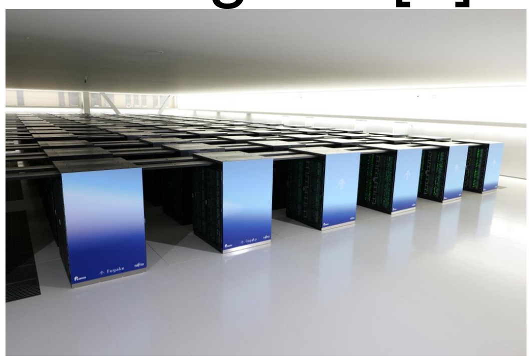
- Algorithm update: Lagrange to Spline, MPI parallelization
- Optimization for OpenACC/OpenMP version with a new View class
- Optimization for Kokkos version with Layout and tile size tuning
- Performance and scalability

Summary and future work

Demands for MPI + 'X' in our group:
readability, portability, productivity, and high performance

Portability

ARM machine Fugaku [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];
```

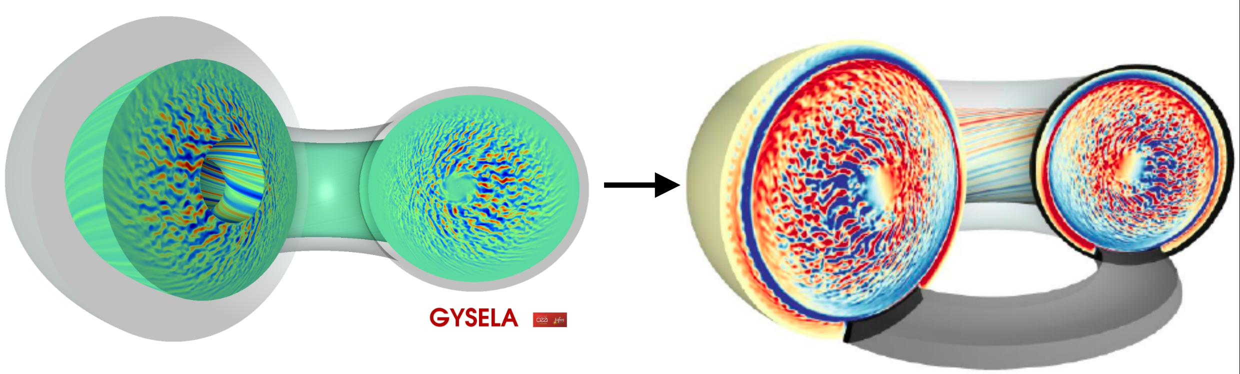
OpenACC

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

Readable for physicists

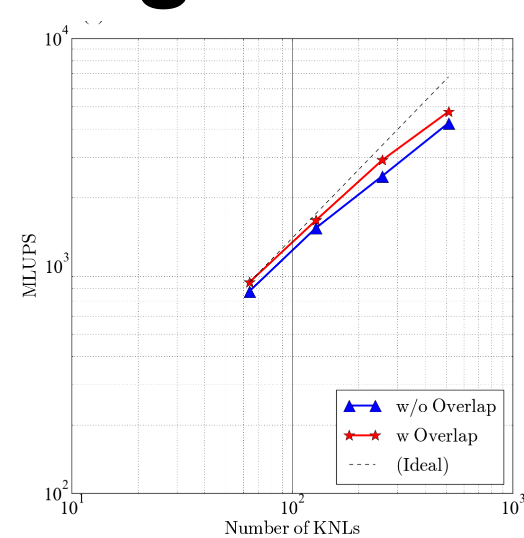
Productivity

Circular → **Limiter**



Advanced (realistic) physical model

High Performance



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

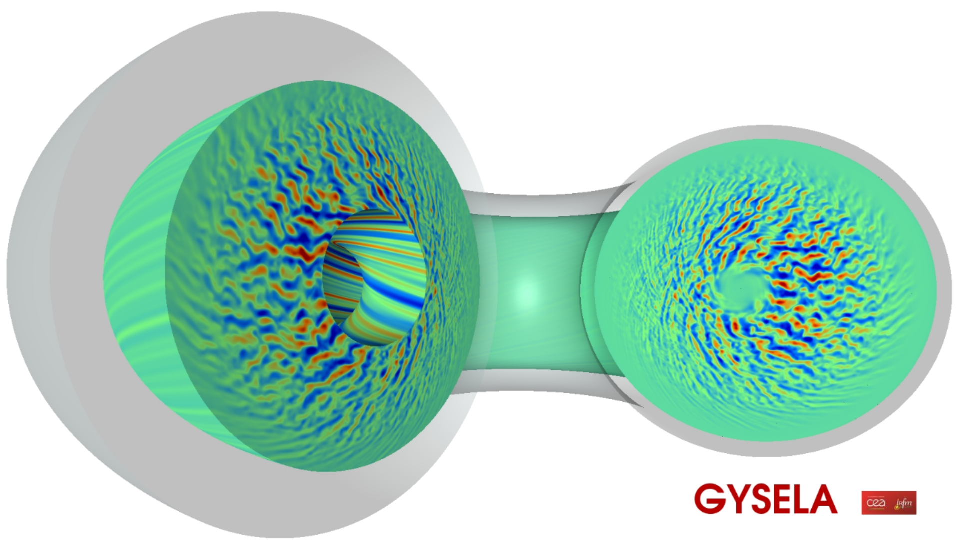
More than 100 M cpu hours/year

- Directive based approach: OpenMP, **OpenACC, OpenMP4.5**
- Higher level abstraction: **Kokkos**, RAJA, Alpaka
- Target devices: **Nvidia GPUs**, **Intel CPU**, **ARM CPU**

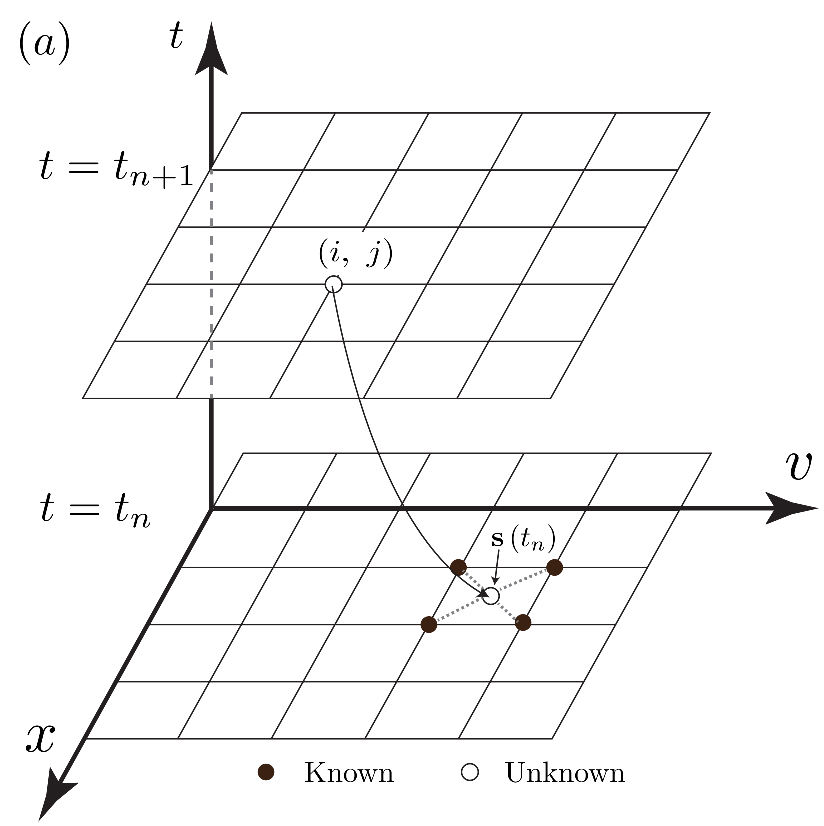
[1] https://www.r-ccs.riken.jp/en/
[2] https://www.olcf.ornl.gov/summit/

GYSELA code

Physics



Numerics



- 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

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

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

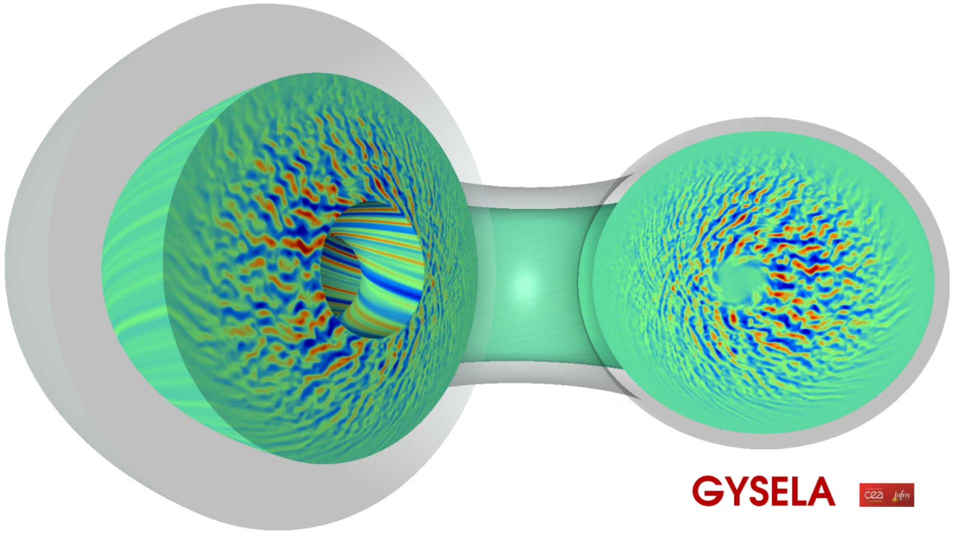
$$N_{\mathrm{MPI}} = p_r \times p_\theta \times N_\mu$$

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

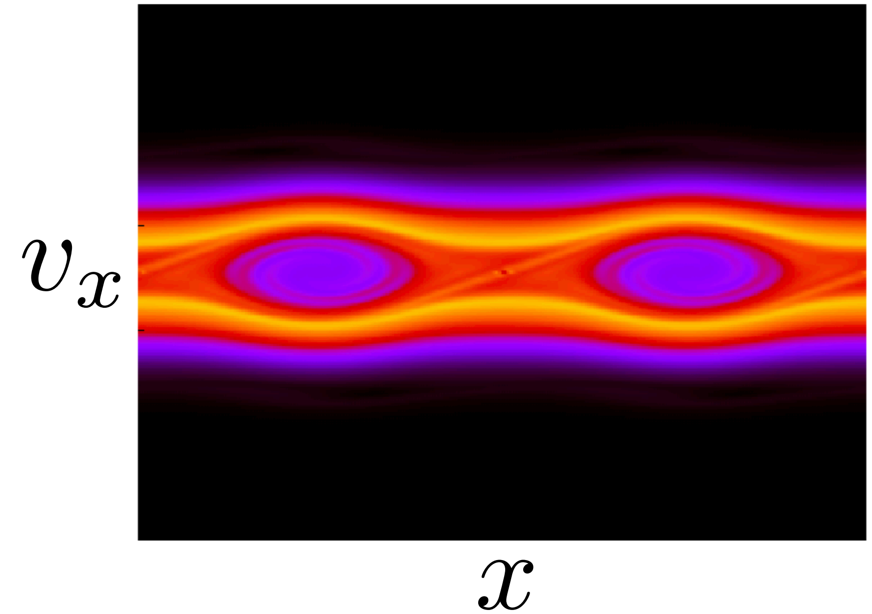
[1] V. Grandgirard, et al, CPC (2016)

Encapsulate key GYSELA features into mini-app

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



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



	GYSELA	Mini-app	Mini-app MPI
System	5D Vlasov + 3D Poisson	4D Vlasov + 2D Poisson	4D Vlasov + 2D Poisson
Geometry	Realistic tokamak geoemtry	Periodic boundary conditions	Periodic boundary conditions
Scheme	Semi-Lagrangian (Spline) + Operator splitting	Semi-Lagrangian (Lagrange) + Operator splitting	Semi-Lagrangian (Spline) without Operator splitting
MPI	Yes	**No**	**Yes**
X	OpenMP	**OpenACC/OpenMP/Kokkos**	**OpenACC/Kokkos**
Language	Fortran 90	**C++**	**C++**
Lines of	More than 50k	About 5k	About 8k

- Extract the **Semi-Lagrangian + operator splitting** strategy for Vlasov solver
- Mini-app preferable to test advanced implementations (algorithms)
- Choose OpenACC/**Kokkos** for MPI version based on our experience [1]

[1] https://github.com/yasahi-hpc/vlp4d

Testbed description

	P100	V100	Skylake	Arm (TX2)
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)

Outline

Introduction

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapps
- 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

MPI parallelization of mini-app

- Algorithm update: Lagrange to Spline, MPI parallelization
- Optimization for OpenACC/OpenMP version with a new View class
- Optimization for Kokkos version with Layout and tile size tuning
- Performance and scalability

Summary and future work

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

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)**

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

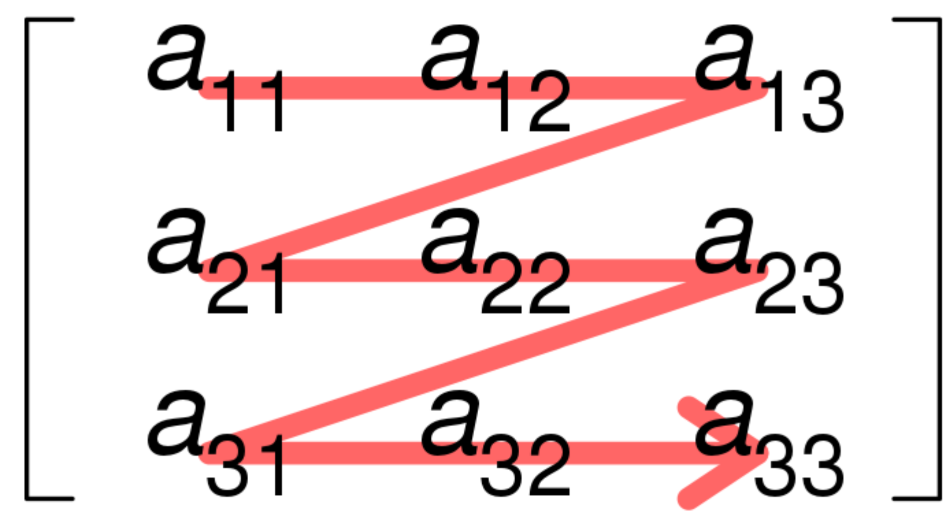
Layout Right (C style)

- **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



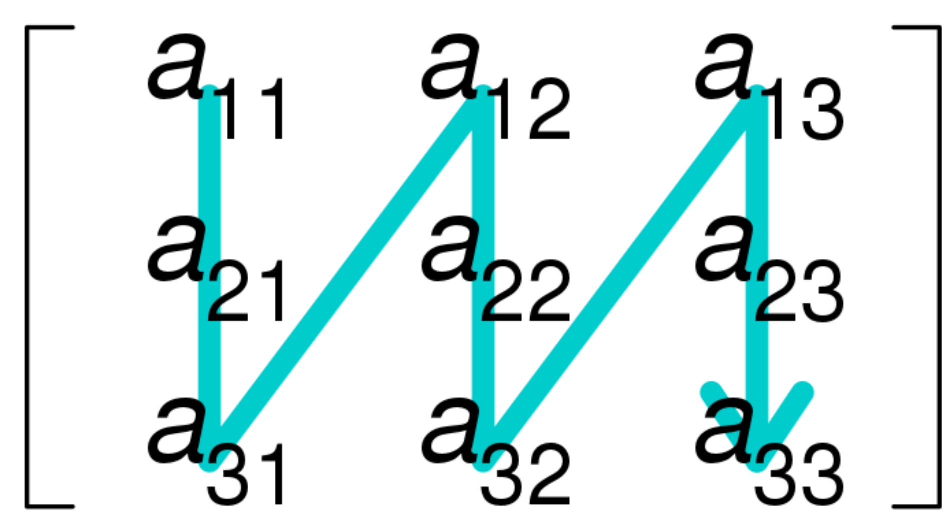
Layout Left (Fortran style)

- **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

High dimensional loop support: 3D range policy

3D indices

```
struct advect_1D_x_functor {
  …
  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;
    }
  }
}

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 tiling

OpenMP (3D Tiling)

```
#pragma omp parallel for collapse(3)
for(int ivx_tile=0; ivx_tile<nvx; ivx_tile+=TZ) {
  for(int iy_tile=0; iy_tile<ny; iy_tile+=TY) {
    for(int ix_tile=0; ix_tile<nx; ix_tile+=TX) {
      for(int ivx=ivx_tile; ivx < ivx_tile+TZ; ivx++) {
        for(int iy=iy_tile; iy < iyx_tile+TY; iy++) {
          for(int ix=ix_tile; ix < iyx_tile+TX; ix++) {
            openmp_kernel(ix, iy, ivx);
          }
        }
      }
    }
  }
}
```

CUDA (3D thread mapping)

```
grid(nx/TX, ny/TY, nvx/TZ);
block(TX, TY, TZ);
cuda_kernel<<<grid, block>>>;
```

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

:Pattern :Policy

Achieved performance

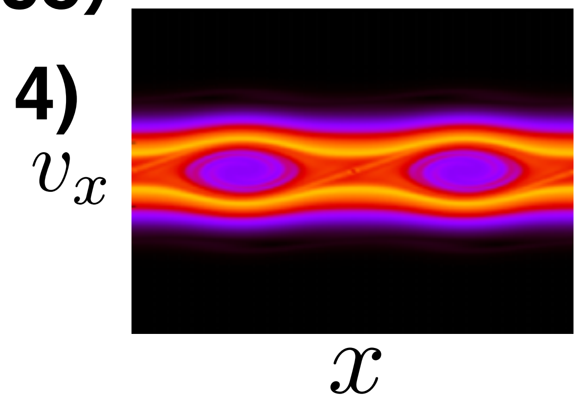
Device	Kernel	f/b	Ideal GFlops	Achieved performance			
				GFlops		GB/s (relative to STREAM %)	
Skylake (Kokkos/ OpenMP)	Advect (x)	67/16	335	271.7	41.8	64.9 (81.1%)	9.98 (12.5%)
	Advect (y)	67/16	335	63.5	291.1	15.2 (19.0%)	69.51 (86.9%)
	Advect (vx)	65/16	325	278.5	31.94	68.6 (85.7%)	7.86 (9.8%)
	Advect (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%)
TX2 (Arm) (Kokkos/ OpenMP)	Advect (x)	67/16	492.8	228.0	30.1	54.4 (45.4%)	7.20 (6.0%)
	Advect (y)	67/16	492.8	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%)
V100 (Kokkos/ OpenACC)	Advect (x)	67/16	3475.6	2701.1	1814.6	645.0 (77.8%)	433.3(52.2%)
	Advect (y)	67/16	3475.6	2205.2	1804.3	526.6 (63.4%)	430.9 (51.9%)
	Advect (vx)	65/16	3371.9	1403.7	946.1	345.5 (41.6%)	232.9 (28.1%)
	Advect (vy)	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**

Performance portable implementation with Kokkos/Directives

4D Vlasov-Poisson equation （2D space、2D velocity space）

- Vlasov solver: Semi-Lagrangian, Strang splitting (1D x 4)
- Poisson solver: 2D Fourier transform



Kokkos version of Poisson solver

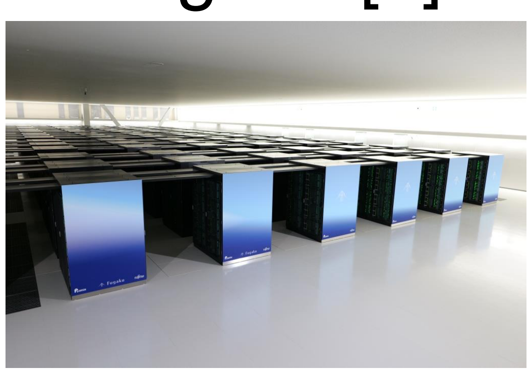
```
// Forward 2D FFT (Real to Complex)
fft_->rfft2(rho_.data(), rho_hat_.data());
Kokkos::parallel_for(nx1h, KOKKOS_LAMBDA (const int ix1) {
  for(int ix2=1; ix2<nx2h; ix2++) {
    double ky = ix2 * ky0;
    double k2 = kx * kx + ky * ky;

    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;
  }
  …
});
// Backward 2D FFT (Complex to Real)
fft_->irfft2(rho_hat.data(), rho_.data());
fft_->irfft2(ex_hat.data(),  ex_.data());
fft_->irfft2(ey_hat.data(),  ey_.data());
```

Performance against SKL (OpenMP)

	Time [s]	Speedup
Skylake (OpenMP)	278	x 1.00
Skylake (Kokkos)	192	x 1.45
TX2 (OpenMP)	589	x 0.47
TX2 (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

Single code works on CPUs/GPUs

Fugaku [1]



Summit [2]

Achievements

Good performance portability keeping readability and productivity with Kokkos (Abstraction of memory and parallel operation)

[1] https://www.r-ccs.riken.jp/en/
[2] https://www.olcf.ornl.gov/summit/
[3] Y. Asahi et al., OpenACC meeting, September, Japan
[4] Y. Asahi et al., waccpd (SC19), November, US

Outline

Introduction

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapps
- 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

MPI parallelization of mini-app

- Algorithm update: Lagrange to Spline, MPI parallelization
- Optimization for OpenACC/OpenMP version with a new View class
- Optimization for Kokkos version with Layout and tile size tuning
- Performance and scalability

Summary and future work

GYSELA mini app (One time step)

Algorithm 1 One time step

1: **Input:** f^n, **Output:** f^{n+1}
2: Halo exchange on f^n (P2P communications)
3: Compute spline coefficient along (x, y) directions: $f^n \to \eta_{\alpha,\beta}$
4: 2D advection along (x, y) directions for $\Delta t/2$
5: Velocity space integral: Compute $\rho^{n+1/2}$ (MPI_all_reduce communication)
6: Field solver: Compute $E_x^{n+1/2}$, $E_y^{n+1/2}$
7: Compute spline coefficient along (v_x, v_y) directions: $\eta_{\alpha,\beta} \to \eta_{\alpha,\beta,\gamma,\delta}$
8: 4D advection along x, y, v_x, v_y directions for Δt

Metric	Adv2D	Adv4D	Spline	Integral
Memory accesses	2 load + 2 store	2 load + 2 store	1 load + 1 store	1 load
Access pattern	Indirect access along **x, y** direction	Indirect access along **x, y, vx, vy** direction	Read/Write dependency along (vx and vy directions)	Reduction by row (along vx and vy)
Flop/Byte (f/b)	61/32	845/32	18/16	1/8

2D advection $\frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{x}} = 0$ at (v_x, v_y) fixed.

4D advection $\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} f + E(t, \mathbf{x}) \cdot \nabla_{\mathbf{v}} f = 0,$

Poisson (Integral) $\nabla_{\mathbf{x}} \cdot E(t, \mathbf{x}) = \rho(t, \mathbf{x}) - 1$ $\quad \rho(t, \mathbf{x}) = \int d\mathbf{v} f(t, \mathbf{x}, \mathbf{v})$

- Domain decomposition based on Unbalanced Recursive Balanced (URB) algorithm (P2P communication and all reduce in Poisson equation)
- Local spline is used for interpolation

OpenACC View with layout abstraction

View v0

```
float64 *dptr_in = d_in.raw(); complex64 *dptr_out = d_out.raw();
complex64 *dptr_ikx = d_ikx.raw(),*dptr_iky = d_iky.raw();
int n1 = Nx*Ny*Nz, n2 = (Nx/2+1)*Ny*Nz, n3 = (Nx/2+1)*Ny;
#pragma acc data copy(dptr_in[0:n1], dptr_out[0:n2]), copyin(dptr_ikx[0:n3], dptr_iky[0:n3])
{
  #pragma acc host_data use_device(dptr_in, dptr_out)
  fft.rfft2(dptr_in, dptr_out);

  #pragma acc parallel loop collapse(2)
  for(int iy=0; iy<Ny; iy++) {
    for(int ix=0; ix<Nx/2+1; ix++) {
      complex64 ikx = dptr_ikx[Index::coord_2D2int(ix,iy,Nx/2+1,Ny)];
      complex64 iky = dptr_iky[Index::coord_2D2int(ix,iy,Nx/2+1,Ny)];

      #pragma acc loop seq
      for(int iz=0; iz<Nz; iz++) {
        int idx = Index::coord_3D2int(ix,iy,iz,Nx/2+1,Ny,Nz);
        dptr_out[idx] = (ikx * dptr_out[idx] + iky * dptr_out[idx]) * normcoeff;
      }
    }
  }

  #pragma acc host_data use_device(dptr_in, dptr_out)
  fft.irfft2(dptr_out, dptr_in);
}
```

Cast to raw pointers, **views not accessible** in the accelerated region

View v1

```
#pragma acc data present(d_in, d_out, d_ikx, d_iky)
{
  fft.rfft2(d_in.data(), d_out.data());

  #pragma acc parallel loop collapse(2)
  for(int iy=0; iy<Ny; iy++) {
    for(int ix=0; ix<Nx/2+1; ix++) {
      complex128 ikx = d_ikx(ix, iy);
      complex128 iky = d_iky(ix, iy);

      for(int iz=0; iz<Nz; iz++) {
        d_out(ix, iy, iz) = (ikx * d_out(ix, iy, iz) + iky * d_out(ix, iy, iz)) * normcoeff;
      }
    }
  }
  fft.irfft2(d_out.data(), d_in.data());
};
```

Kokkos like accessor

Contiguous dim can be specified at compile time

host_data use_device inside the function

Testbed description

	Tsubame3	JFRS1	Flow
Processor	NVIDIA Tesla P100	Intel Xeon Gold 6148	Fujitsu A64FX
Number of nodes	540	1512	2304
Processors per Node	4 GPUs	2 CPUs	1 (4 CMGs)
Number of cores	1792 (DP)	20	48 + 4
Network architecture	Intel Omni Path 2:1	InfiniBand EDR	TofuD
Network topology	Fat-tree	Cray Dragonfly	3D mesh/torus
Network [GB/s]	12.5 x 2	12.5	40.8
	P100 (1 GPU)	Skylake (1 CPU)	A64FX (4 CMGs)
L2/L3 Cache [MB]	4	27.5	32 (8 x 4)
GFlops (DP)	**5300**	**1536**	**3379**
Peak B/W [GB/s]	732	127.97	1024
STREAM B/W [GB/s]	**540**	**80**	**800**
SIMD width	**-**	**512 bit**	**512 bit (SVE)**
B/F ratio	0.138	0.083	0.213
TDP [W]	300	145	-
Manufacturing process	16 nm	14 nm	7 nm
Compiler	cuda/10.2.89, pgi19.1	intel19.0.0.117	Fujitsu compiler 1.2.25
Compiler options	-ta=nvidia:cc60 -O3	-xCORE-AVX512 -O3	-O3 -Kfast,openmp -Krestp=all

- Flow has a quite high network bandwidth ~ 40 GB/s (bidirectional)
- Peak Gflops are high on each architecture
- The memory bandwidth of A64FX is comparable to GPUs

Outline

Introduction

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapps
- 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

MPI parallelization of mini-app

- Algorithm update: Lagrange to Spline, MPI parallelization
- Optimization for OpenACC/OpenMP version with a new View class
- Optimization for Kokkos version with Layout and tile size tuning
- Performance and scalability

Summary and future work

Optimization for directive version (on A64FX)

Original version (velocity space integral)

```
#pragma omp parallel for
for(int iy=ny_min; iy<ny_max; iy++) {
  #pragma omp simd
  for(int ix=nx_min; ix<nx_max; ix++) {
    float64 sum = 0.;
    for(int ivy=nvy_min; ivy<nvy_max; ivy++) {
      for(int ivx=nvx_min; ivx<nvx_max; ivx++) {
        sum += fn(ix, iy, ivx, ivy);
      }
    }
    ef->rho_loc_(ix, iy) = sum * dvx * dvy;
  }
}
```

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

Optimized version (strip-mining and SIMD loops)

```
#define SIMD_WIDTH 8
#pragma omp parallel for
for(int iy=ny_min; iy<ny_max; iy++) {
  for(int ix = nx_min; ix < nx_max; ix+=SIMD_WIDTH) {
    float64 sum_vec[SIMD_WIDTH];
    for(int ivec = 0; ivec < SIMD_WIDTH; ivec++) {
      sum_vec[ivec] = 0.;
    }
    for(int ivy=nvy_min; ivy<nvy_max; ivy++) {
      for(int ivx=nvx_min; ivx<nvx_max; ivx++) {
        for(int ivec = 0; ivec < SIMD_WIDTH; ivec++) {
          sum_vec[ivec] += fn(ix + ivec, iy, ivx, ivy);
        }
      }
    }
    for(int ivec = 0; ivec < SIMD_WIDTH; ivec++) {
      ef->rho_loc_(ix + ivec, iy) = sum_vec[ivec] * dvx * dvy;
    }
  }
}
```

Strip-mining "ix" loop

- **Integral (x 3.31 acceleration), Adv2D (x 1.44), Adv4D (x 1.15)**

Tile size tuning with Kokkos

Launch a 2D Advection kernel with Kokkos

```
MDPolicyType_4D advect_2d_policy4d({{nx_min,   ny_min,   nvx_min,   nvy_min}},
                                   {{nx_max+1, ny_max+1, nvx_max+1, nvy_max+1}},
                                   {{TX, TY, TVX, TVY}});
Kokkos::parallel_for("advect_2d", advect_2d_policy4d, blocked_advect_2D_xy_functor(conf, fn,
fn_tmp, dt, scatter_error));
```

Performance affected by tile size

OpenMP (4D Tiling)

```
#pragma omp parallel for collapse(4)
for(int ivy_tile=0; ivy_tile<nvx; ivy_tile+=TVY) {
  for(int ivx_tile=0; ivx_tile<nvx; ivx_tile+=TVX) {
    for(int iy_tile=0; iy_tile<ny; iy_tile+=TY) {
      for(int ix_tile=0; ix_tile<nx; ix_tile+=TX) {
        for(int ivy=ivy_tile; ivy < ivy_tile+TVY; ivy++) {
          for(int ivx=ivx_tile; ivx < ivx_tile+TVX; ivx++) {
            for(int iy=iy_tile; iy < iyx_tile+TY; iy++) {
              for(int ix=ix_tile; ix < iyx_tile+TX; ix++) {
                openmp_kernel(ix, iy, ivx, ivy);
              }
            }
          }
        }
      }
    }
  }
}
```

CUDA (4D thread mapping)

```
grid(nx*ny/TX, nvx/TY, nvy/TZ);
block(TX, TY, TZ);
cuda_kernel<<<grid, block>>>;
```

Default on CPUs: (Tx, Ty, Tvx, Tvy) = (4, 4, 4, 4)

Default on GPUs: (Tx, Ty, Tvx, Tvy) = (32, 4, 2, 1)

- Scan the elapsed time with respect to the tile size
  Best tile size is affected by **architecture**, **problem size**, etc.

- Run the kernel with the best tile size

Layout Optimization and tile size tuning

Problem size 128^4, 4 CPUs (8 MPI procs), 7 threads (on Broadwell)

Layout tuning

(Layout Left vs Layout Right)

Kernel name	Layout Left [s]	LayoutRight [s]
Advection 2D	0.327	0.532
Advection 4D	1.23	1.520
Packing	0.12	0.158
Unpacking	0.04	0.038
Integral	0.023	0.0091
Spline (x, y)	0.153	0.362
Spline (v_x, v_y)	0.304	0.169

TABLE 1 Elapsed time of each kernel on CPUs with LayoutLeft and LayoutRight.

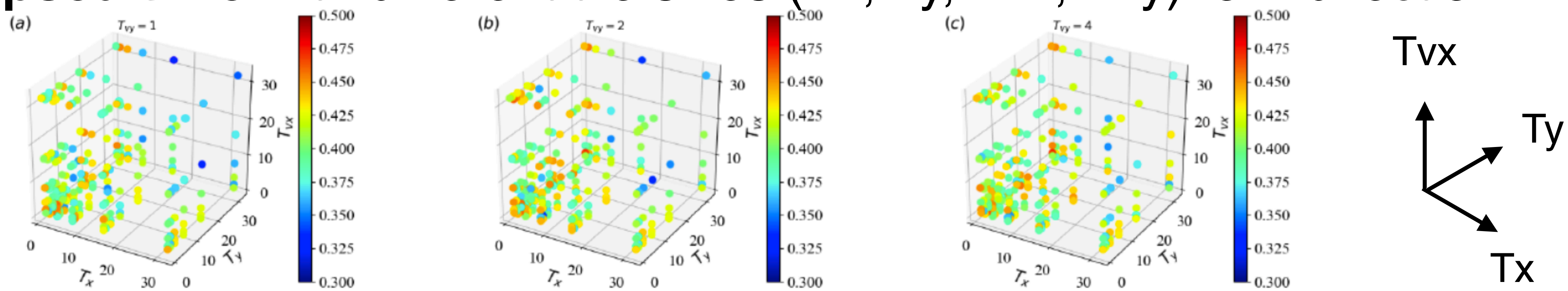
Tile size tuning

Kernel name	OpenMP [s]	OpenMP (tuned) [s]	CUDA [s]	CUDA (tuned) [s]
Advection 2D	0.327	0.297 (x1.10)	0.0107	0.0105 (x1.02)
Advection 4D	1.226	1.137 (x1.08)	0.0413	0.0389 (x1.06)
Integral	0.023	0.0165 (x1.27)	0.00113	0.00113 (x1.00)
Spline (x, y)	0.153	0.151 (x1.02)	0.035	0.029 (x1.18)
Spline (v_x, v_y)	0.304	0.292 (x1.03)	0.035	0.034 (x1.04)

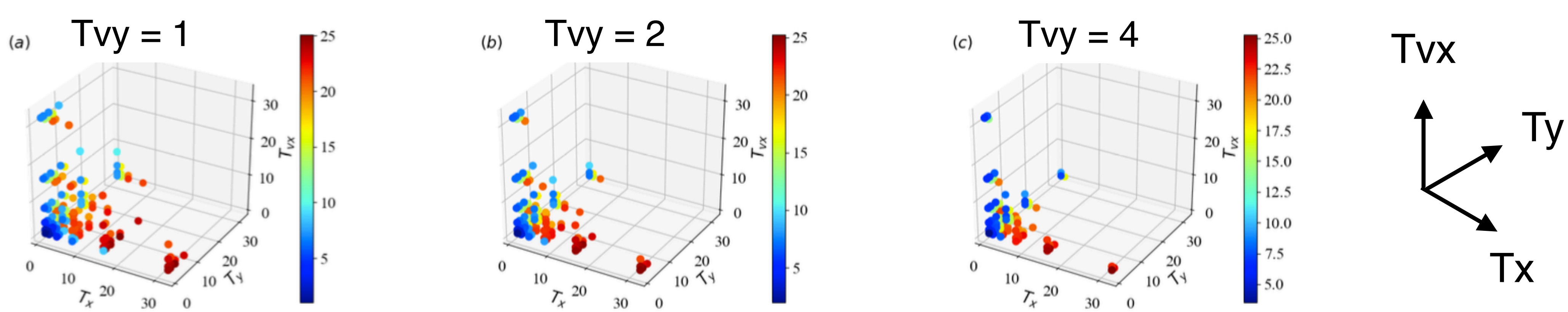
TABLE 2 Elapsed time of each kernel with a tile size tuning for Cuda and OpenMP backends.

Inverse elapsed time with different tile sizes (Tx, Ty, Tvx, Tvy) for Advection 4D kernel

On CPUs



On GPUs



- **Layout Left** is better choice to accelerate the entire mini-app
- **CPU** (resp. GPU) performance is highly (resp. less) affected by the tile size

Achieved performance

Problem size 128^4, 8 MPI processes

Skylake 4 CPUs
(1536 x 4 GFlops, 80 x 4 GBytes/s,
L3 Cache: 27.5 MB x 4)

A64FX 2 CPUs
(3379 x 2 GFlops, 800 x 2 GBytes/s,
L2 Cache: 32 MB x 2)

P100 8 GPUs
(5300 x 8 GFlops, 540 x 8 GBytes/s,
L2 Cache: 4 MB x 8)

	Kernel	f/b	Ideal Performance [GFlops]	Achieved Performance GFlops	GBytes/s
Skylake (Kokkos)	advection 2D	61/32	76.25	11.5 (15.1%)	6.02
	advection 4D	845/32	768	47.8 (6.2%)	1.81
	spline 2D	18/16	45	6.27 (13.9%)	5.57
	integral	1/8	5	1.04 (20.8%)	8.32
Skylake (OpenMP)	advection 2D	61/32	76.25	22.32 (29.3%)	11.7
	advection 4D	845/32	768	61.62 (8.02%)	2.33
	spline 2D	18/16	45	6.51 (14.5%)	5.79
	integral	1/8	5	2.80 (56.0%)	22.39
A64FX (Kokkos)	advection 2D	61/32	381.25	2.75 (0.72%)	1.44
	advection 4D	845/32	844.75	24.23 (2.87%)	0.92
	spline 2D	18/16	225	1.70 (0.76%)	1.51
	integral	1/8	25	0.50 (2%)	3.97
A64FX (OpenMP)	advection 2D	61/32	381.25	11.92 (3.12%)	6.25
	advection 4D	845/32	844.75	28.39 (3.36%)	1.08
	spline 2D	18/16	225	1.56 (0.69%)	1.39
	integral	1/8	25	2.52 (10.06%)	20.13
P100 (Kokkos)	advection 2D	61/32	1029.37	192.48 (18.7%)	100.97
	advection 4D	845/32	5300	706.8 (13.3%)	26.77
	spline 2D	18/16	607.5	20.05 (3.3%)	17.83
	integral	1/8	67.5	28.36 (42%)	226.91
P100 (OpenACC)	advection 2D	61/32	1029.37	319.19 (31%)	165.35
	advection 4D	845/32	5300	819.06 (15.4%)	31.02
	spline 2D	18/16	607.5	22.67 (3.73%)	20.16
	integral	1/8	67.5	56.46 (83.6%)	451.66

TABLE 8 Achieved performance on Skylake (half socket), A64FX (1 CMG, quarter socket) and P100. The Flop/Byte (f/b) is measured in average assuming a perfect and unlimited cache. The ideal performance is estimated by the Roofline model in Eq. (4), where the upper ceiling is given by the STREAM bandwidth in each case. The achieved GFlops to the ideal performance are presented in the parentheses.

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

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

- **Low performance on A64FX due to Smaller cache and the usage of C++**

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

Performance portability based on Roofline model

Performance portability metric [1]

$$\mathcal{P}(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

a: Application
p: Simulation setting
H: Set of platforms

Efficiency evaluated based on Roofline model

$$e_i(a,p) = \frac{P_{a,p,i}}{\min(F_i, B_i \times f_a / b_a)}.$$

$P_{a,p,i}$: Achieved GFlops on i

Performance portability on Skylake, (A64FX) and P100

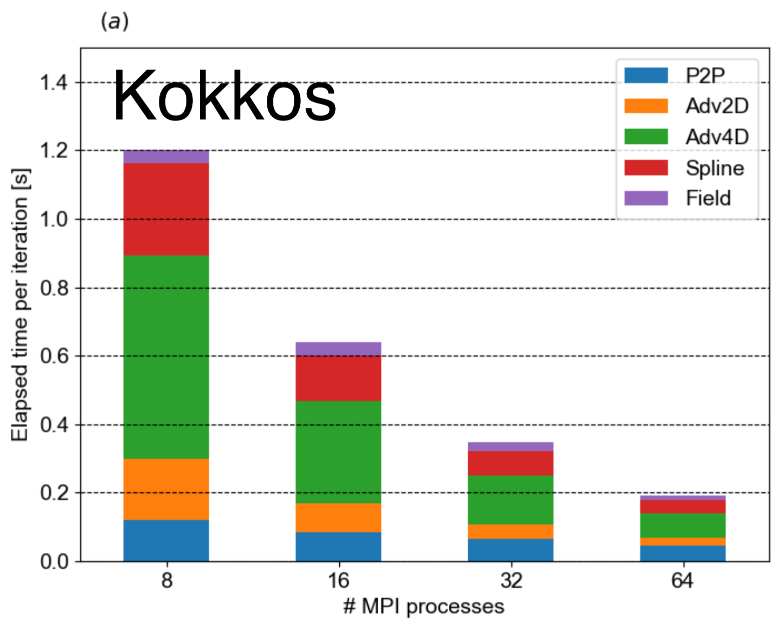
Kernel name	Directives	Kokkos
Advection 2D	7.75 (30.13)	1.99 (16.71)
Advection 4D	6.16 (10.55)	5.13 (8.46)
Spline 2D	1.68 (5.93)	1.77 (5.33)
Integral	23.2 (67.07)	5.25 (27.82)

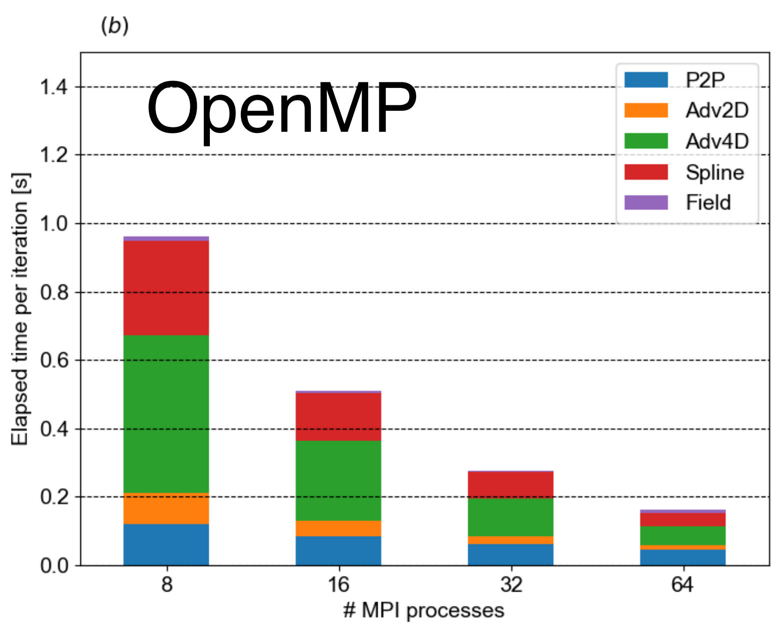
- Excluding A64FX, we get a good performance portability
- OpenMP/OpenACC version shows better performance

[1] S. Pennycook, et al, Future Generation Computer Systems, (2019).

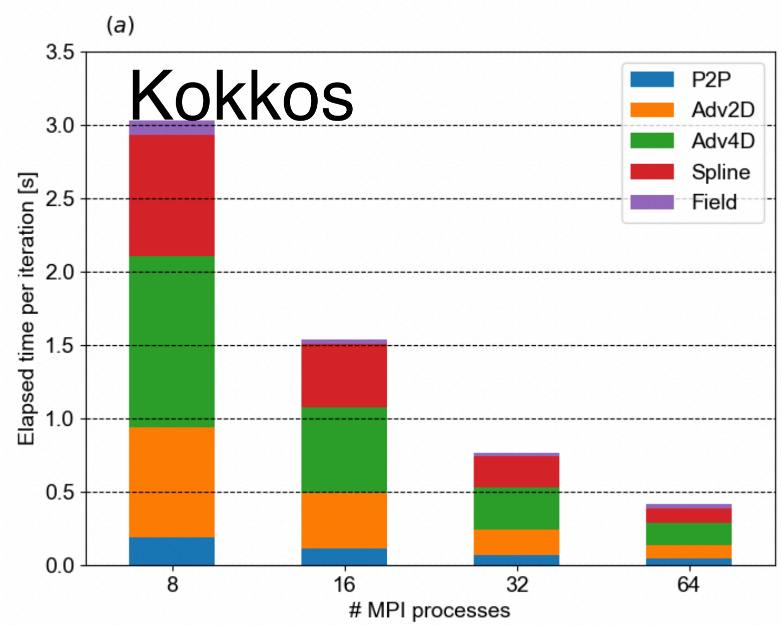
Scalability of the Mini-app

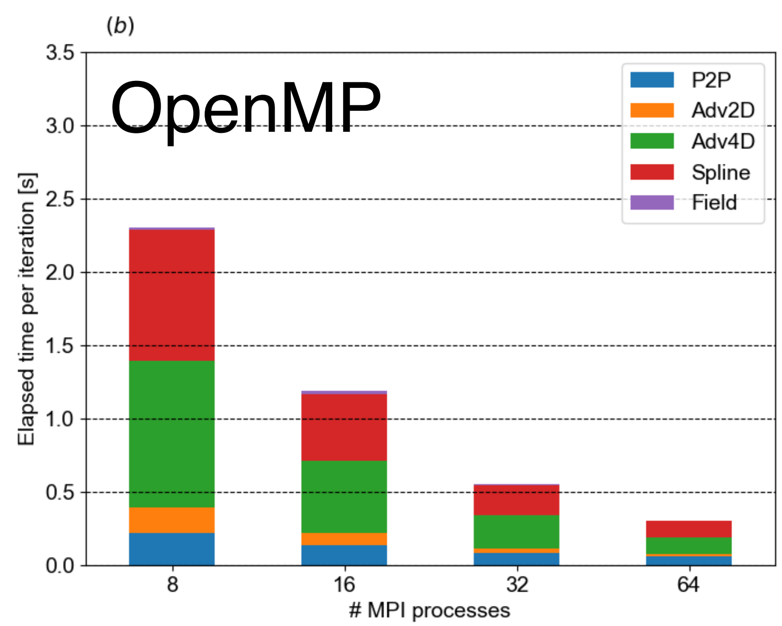
Skylake (4 to 32 CPUs)



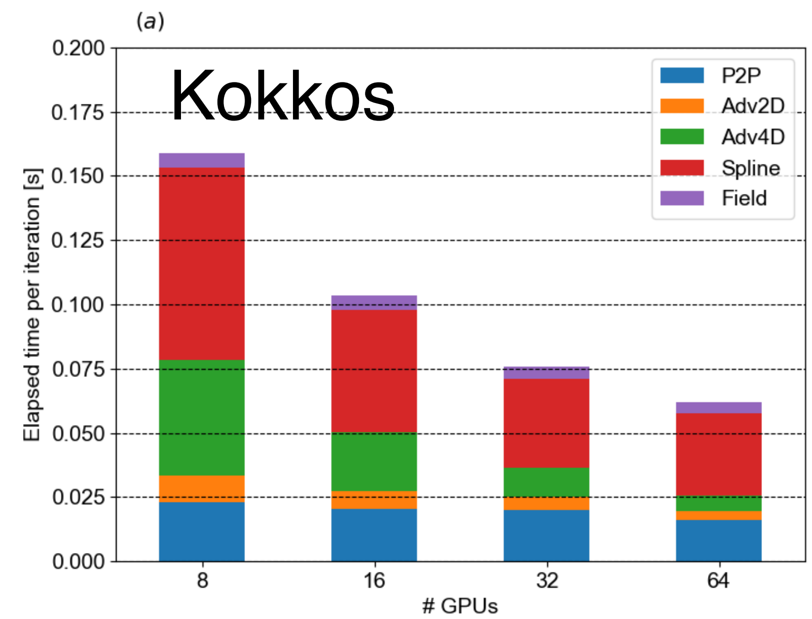


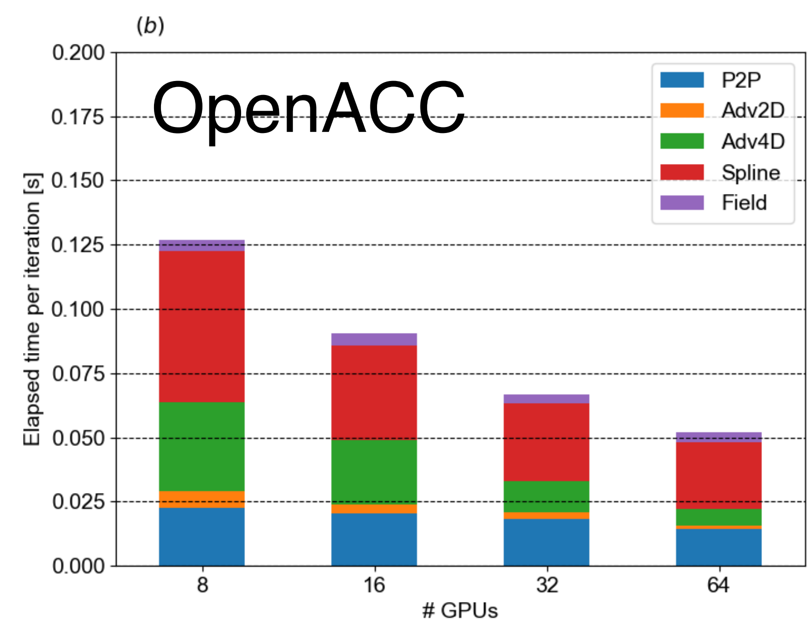
A64FX (2 to 16 CPUs)





P100 (8 to 64 GPUs)





- On P100, Spline and P2P do not scale well 2 - 16 nodes
- OpenMP/OpenACC version shows better performance

Outline

Introduction

- Demands for MPI + ‘X’ for kinetic simulation codes
- Brief introduction of GYSELA code and miniapps
- 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

MPI parallelization of mini-app

- Algorithm update: Lagrange to Spline, MPI parallelization
- Optimization for OpenACC/OpenMP version with a new View class
- Optimization for Kokkos version with Layout and tile size tuning
- Performance and scalability

Summary and future work

Summary

Directive based approach: mixed OpenACC/OpenMP

- Mixed OpenACC/OpenMP achieves high performance (marginal on A64FX)
- Suitable for **porting a large legacy code** (e.g. more than 50k LoC)
- Introducing OpenACC View improves the readability
- SIMD optimizations (like strip-mining) are critical on CPUs

Higher level abstraction: Kokkos

- Kokkos can achieve good **performance portability except for A64FX**
- Appropriate choice of an **execution policy** seems critical for CPUs
- Layout and tile size tunings improve the performance when cache matters

Future Plans

- Further optimization on A64FX (clang compiler may be helpful)