Acceleration of stencil-based fusion kernels

Y. ASAHI[1], G. Latu[1], T. Ina[2], Y. Idomura[2],
V. Grandgirard[1], X. Garbet[1]
CEA[1]、Japan Atomic Energy Agency[2]

energie atomique • energies alternatives

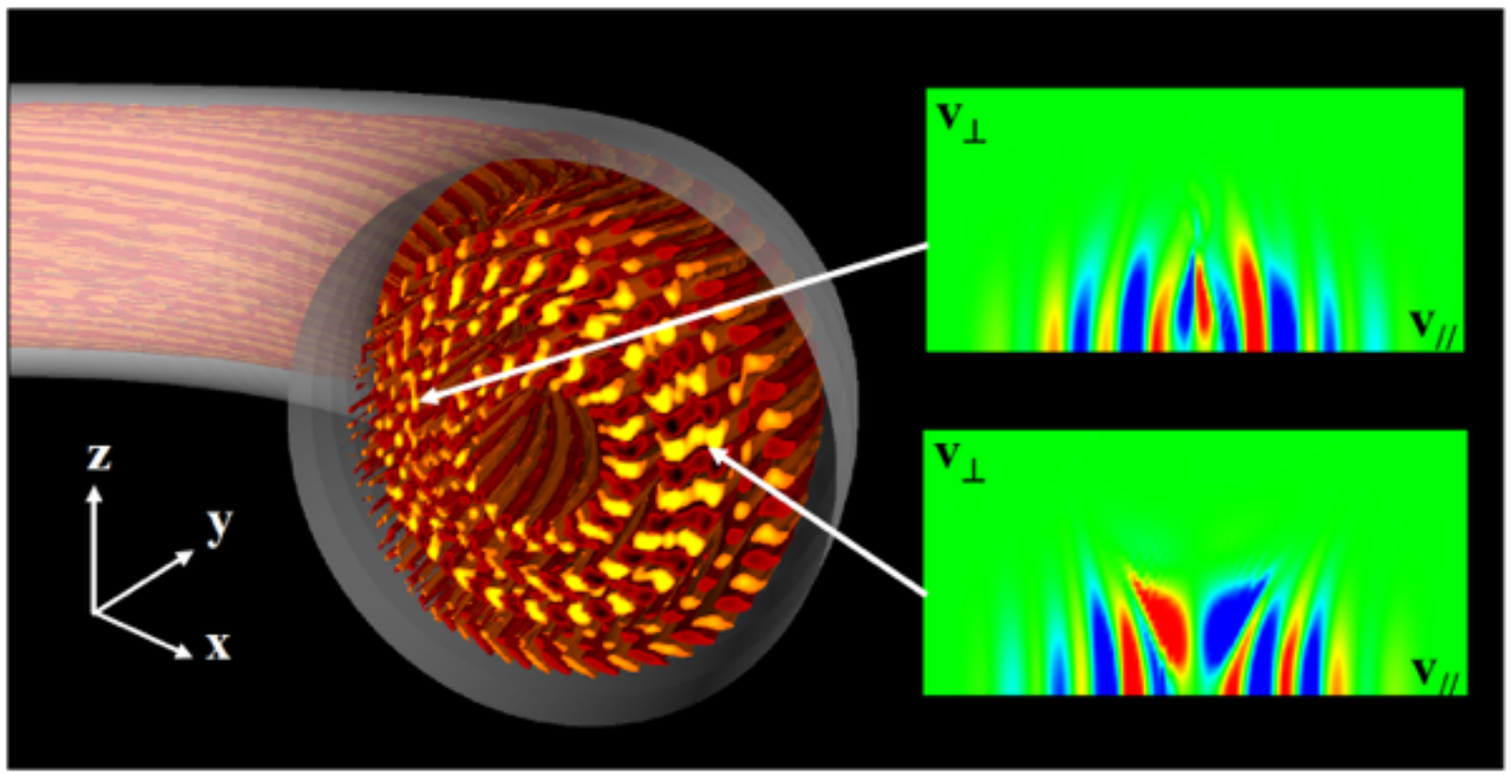
Outline

- Introduction
  - Demands for exa-scale supercomputer
  - Semi-Lagrangian and Finite-Difference kernels from GYSELA and GT5D
- Optimization strategies
  - Semi-Lagrangian kernel on Xeon Phi and GPGPU
  - Finite-Difference kernel on Xeon Phi and GPGPU
- Summary
  - Acceleration ratio of kernels
  - Summary for optimization strategies on Xeon Phi and GPGPU

Outline

- Introduction
  - Demands for exa-scale supercomputer
  - Semi-Lagrangian and Finite-Difference kernels from GYSELA and GT5D
- Optimization strategies
  - Semi-Lagrangian kernel on Xeon Phi and GPGPU
  - Finite-Difference kernel on Xeon Phi and GPGPU
- Summary
  - Acceleration ratio of kernels
  - Summary for optimization strategies on Xeon Phi and GPGPU

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)]

- The 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)
- Concerning the dynamics of **kinetic electrons**, more computational resource is needed
  - ⟶ **Accelerators** are key ingredients to satisfy huge computational demands at **reasonable energy consumption**

GYSELA and GT5D

GYSELA [1] and GT5D [2] computes **4D convection operator** in the Vlasov equation with different numerical schemes

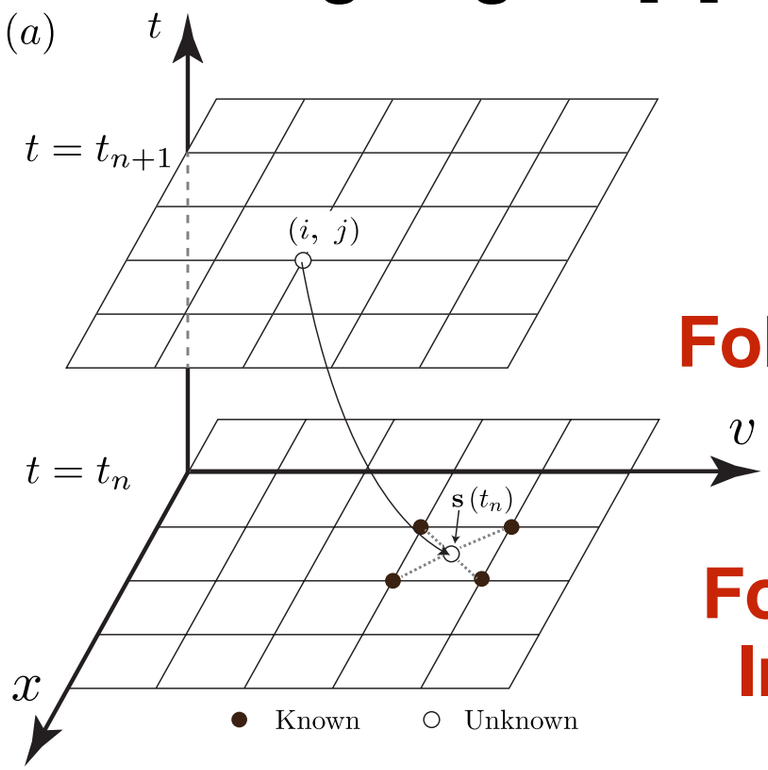
$$\frac{\partial f}{\partial t} = \mathcal{G}(f) + \mathcal{C}(f)$$

$$\mathcal{G}(f) = -\mathbf{U}_1 \cdot \frac{\partial f}{\partial \mathbf{R}} - U_2 \frac{\partial f}{\partial v_{\|}}$$

4D Op.

- GYSELA computes 4D convection operator (which is split into 1D+1D +2D parts) with **Semi-Lagrangian** method.
- GT5D kernel computes the 4D convection operator with a 4th order **finite difference** (Morinishi scheme, 17-stencils).

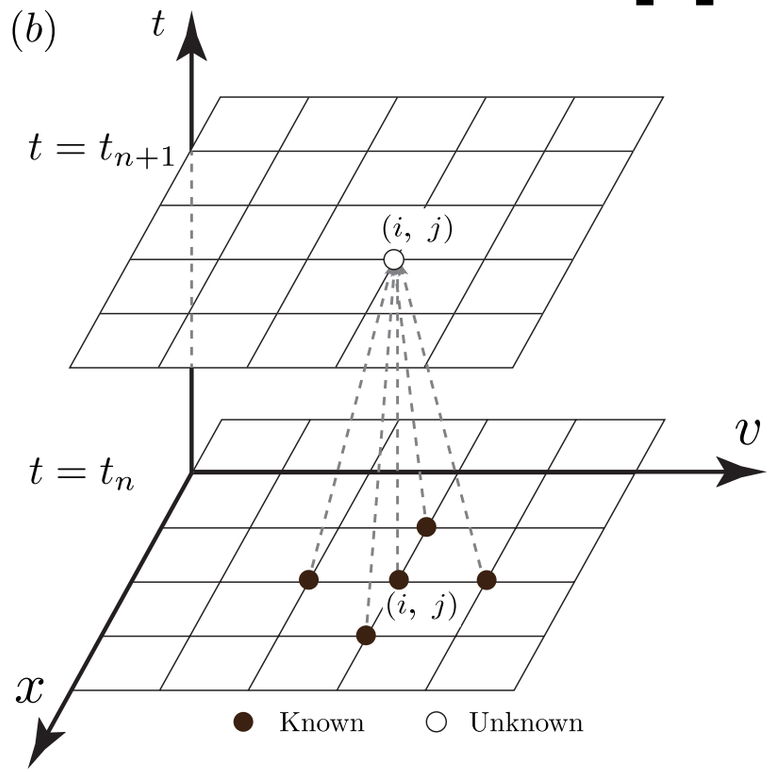
Semi-Lagrangian [3]



Follow back in time

Footpoint Interpolate

Finite Difference [3]



Acceleration of stencil kernels

Difficulty of the acceleration of 5D stencil kernels

- Vectorisation (SIMD instructions)
- load balancing within a plenty of cores
- **Complex memory access** patterns (affinity to architecture)

Establish the optimisation strategies of high dimensional stencil kernels on accelerators

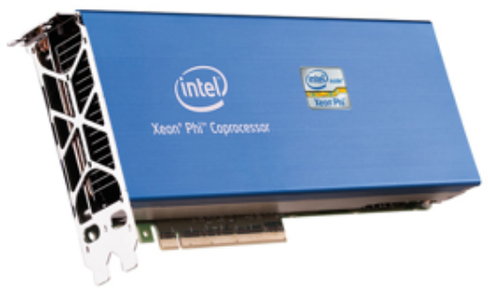
- Employing the 4D fusion kernels with different numerical schemes: **Semi-Lagrangian** and **Finite-Difference**
- Investigate the architecture affinity to the complex memory access patterns: **Indirect-access** and **strided-access**

Outline

- Introduction
  - Demands for exa-scale supercomputer
  - Semi-Lagrangian and Finite-Difference kernels from GYSELA and GT5D
- Optimization strategies
  - Semi-Lagrangian kernel on Xeon Phi and GPGPU
  - Finite-Difference kernel on Xeon Phi and GPGPU
- Summary
  - Acceleration ratio of kernels
  - Summary for optimization strategies on Xeon Phi and GPGPU

Optimization of Semi-Lagrangian kernel

Xeon Phi



	time [s]	Speed up	GFlops	GB/s
Original	7.557	-	155.2	56.96
AoSoA layout	6.853	1.10	171.2	62.81
Dynamic schedule	6.575	1.15	178.4	65.41

- Changing array style from Array of Structure (AoS) to **Array of Structure of Array (AoSoA)** for SIMD load
- Dynamic scheduling to improve load balance

GPGPU

	time [s]	Speed up	GFlops	GB/s
Original	5.883	-	199.4	71.02
Coalescing access to feet array	5.654	1.04	207.5	73.73
Texture cache utilization	3.135	1.88	374.2	130.63

- Changing array style from Array of Structure (AoS) to **Array of Structure of Array (AoSoA)** for coalescing
- **Texture cache** usage to reduce indirect access cost

Semi-Lagrangian kernel

```
1  #define VSIZE 16
2  #pragma omp for collapse(2), schedule(static, 1)
3    for(int l = 0; l < Nvpar+1; l++){
4      for(int k = 0; k < Nphi; k++){
5        for(int j = 0; j < Ntheta; j++){
6          for(int ii = 0; ii < Nr; ii += VSIZE){
7  #pragma simd
8            for(int i = 0; i < VSIZE; i++){
9              // Load the coordinate of foot points in each direction
10             theta_star[i] = feet[l][k][j][2*(ii + i)];
11             phi_star[i] = feet[l][k][j][2*(ii + i) + 1];
12           }
13 #pragma ivdep
14           for(int i = 0; i < VSIZE; i++){
15             // Corresponding array index in each direction
16             theta_pos[i] = int(theta_star[i] * thetadim_invh);
17             phi_pos[i] = int(phi_star[i] * phidim_invh);
18             // Compute spline bases
19             splinex_basis(theta_star, theta_pos, theta_base);
20             splinex_basis(phi_star, phi_pos, phi_base);
21           }
22           for(int i = 0; i < VSIZE; i++){
23             // Load spline coefficients (indirect access)
24             vec_scoef[0][i] = scoef[l][phi_pos[i]-1][theta_pos[i]-1];
25             vec_scoef[1][i] = scoef[l][phi_pos[i]-1][theta_pos[i]+0];
26             vec_scoef[2][i] = scoef[l][phi_pos[i]-1][theta_pos[i]+1];
27             vec_scoef[3][i] = scoef[l][phi_pos[i]-1][theta_pos[i]+2];
28             vec_scoef[4][i] = scoef[l][phi_pos[i]+0][theta_pos[i]-1];
29             ...
30             vec_scoef[14][i] = scoef[l][phi_pos[i]+2][theta_pos[i]+1];
31             vec_scoef[15][i] = scoef[l][phi_pos[i]+2][theta_pos[i]+2];
32           }
33 #pragma simd
34           for(int i = 0; i < VSIZE; i++){
35             // Cubic spline interpolation
36             spline_interpolation(fval,vec_scoef,theta_base,phi_base);
37           }
38         }
39       }
40     }
41   }
```

2D interpolation in θ、ϕ directions

Problem size (128, 72, 52, 201)

← 1. Load from 4D Foot point (AoS)

← 2. Compute nearest index

← 3. Compute 2D Spline basis

← **4. Load 2D Spline coefficient (indirect access)**

Non-continuous access

← 5. Spline interpolation (2D)

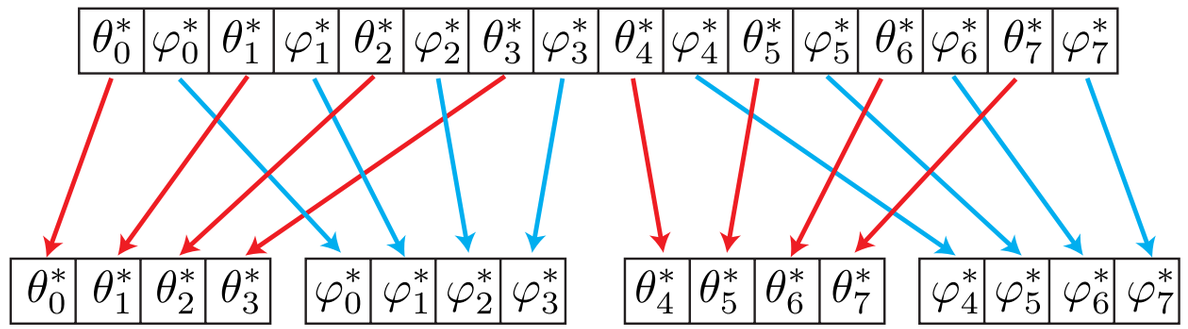
AoSoA layout

Original

```
do mir = 0, VSIZE-1
  theta_star(local_i) = feet(2*(global_i), j, k)
  phi_star(local_i) = feet(2*(global_i)+1, j, k)
enddo
```

VSIZE = **8**
(SIMD width)

(a)



foot point in θ

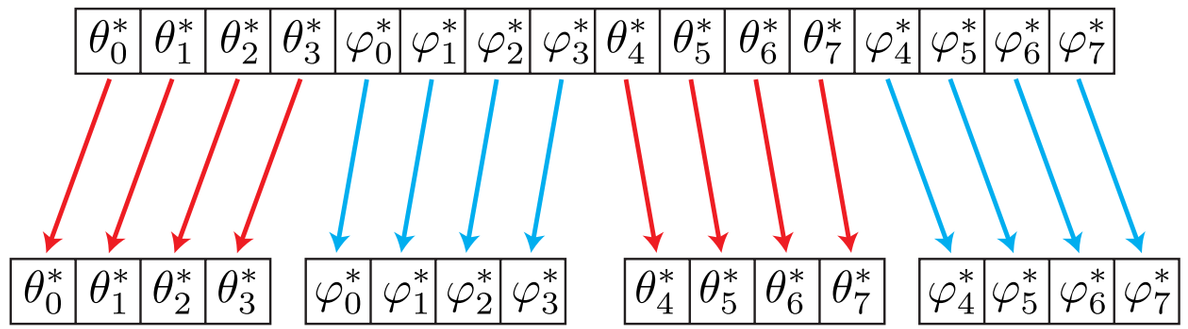
foot point in φ

stride access

Optimized

```
do mir = 0, VSIZE-1
  theta_star(local_i) = feet(2*global_i, j, k)
  phi_star(local_i) = feet(2*global_i+VSIZE, j, k)
enddo
```

(b)

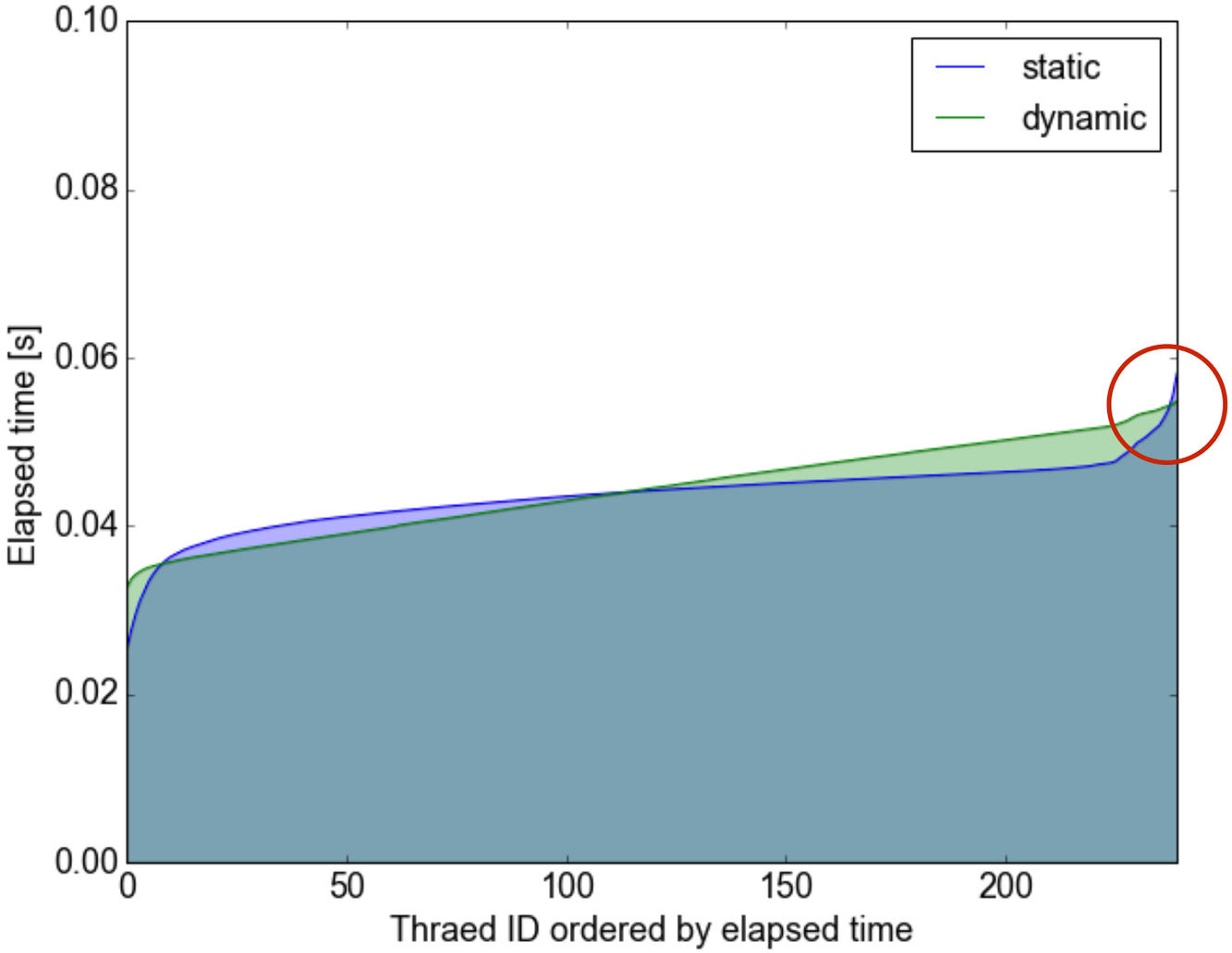


sequential access

64 Byte SIMD load

- Applying Array of Structure of Array (**AoSoA**) layout to **SIMD** load operation

Load Imbalance on Phi

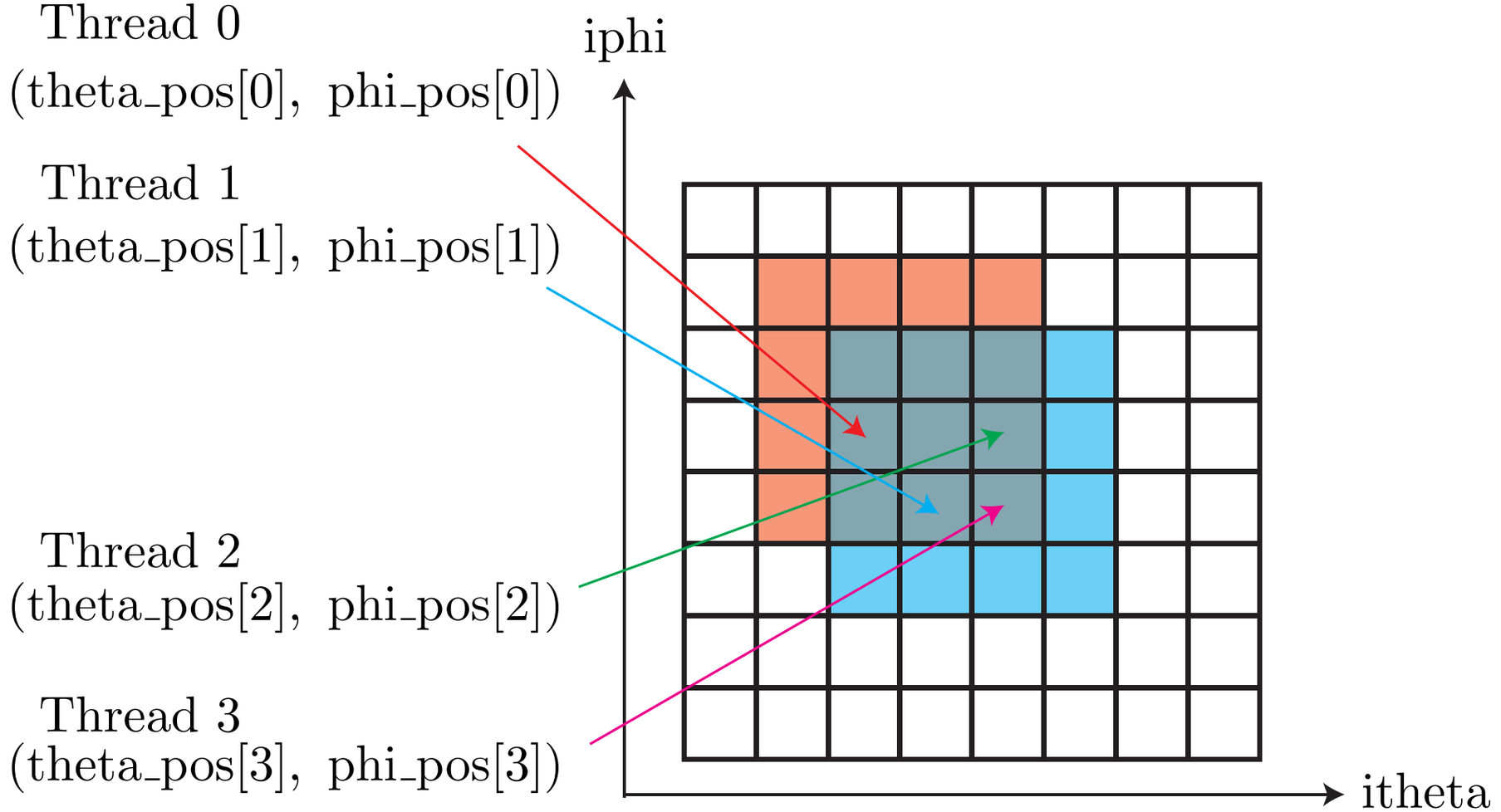


static
Average [ms]: 43.5337826708
Standard deviation [ms] 4.04344345115
Max [ms]: **58.161645**
Min [ms]: 25.18641

dynamic
Average [ms]: 44.3104860667
Standard deviation [ms] 5.33978190512
Max [ms]: **54.844884**
Min [ms]: 32.590335

- Elapsed time of a single iteration by each thread (x 100 times)
- Each result sorted by elapsed time and averaged within 100 results
- The maximum elapsed time is reduced by **6%**, which is roughly the same as **5%** performance improvement by changing the schedule

Texture memory usage



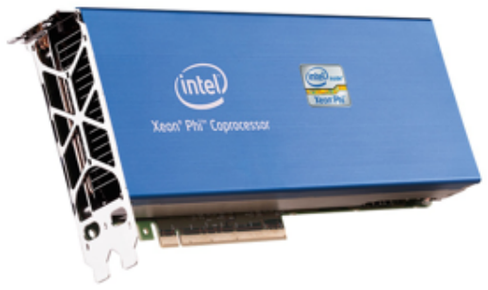
- Each thread accesses different foot points and thus the access pattern is basically unpredictable (**in-direct access**)
- Due to the physical features, there is a spatial locality
  →A thread accesses a memory address close to the addresses accessed by threads in the vicinity (similar memory access pattern to **texture mapping**).
- **Texture cache** usage to reduce the penalty from in-direct accesses

Outline

- Introduction
  - Demands for exa-scale supercomputer
  - Semi-Lagrangian and Finite-Difference kernels from GYSELA and GT5D
- Optimization strategies
  - Semi-Lagrangian kernel on Xeon Phi and GPGPU
  - Finite-Difference kernel on Xeon Phi and GPGPU
- Summary
  - Acceleration ratio of kernels
  - Summary for optimization strategies on Xeon Phi and GPGPU

Optimization of Finite-difference kernel

Xeon Phi



	time [s]	Speed up	GFlops	GB/s
Original	0.429	-	30.7	58.21
Thread mapping	0.331	1.27	39.8	66.61

- Explicit thread mapping to improve the **cache locality in space and time**

GPGPU

	time [s]	Speed up	GFlops	GB/s
Original	0.747	-	16.7	34.6
Thread mapping	0.160	4.67	78.0	161.9
Cyclic register usage	0.131	5.7	95.2	162.8
Manual unrolling	0.115	6.50	108.5	153.9

- Appropriate thread mapping to avoid **warp divergence**
- Effective **register** usage to reduce the total amount of memory accesses

Finite Difference kernel

Code 11: Pseudo Finite-Difference kernel on Xeon Phi

```
#pragma omp parallel for collapse(2)
  for(int i = 0; i < nx; i++){
    for(int j = 0; j < ny; j++){
      for(int l = 0; l < nv; l++){
        // Calculate coefficients used in the following 17 stencil computation
        compute_coeffs(vx, vy, vz, vv, coef);
      }
      for(int k = 0; k < nz; k++){
        for(int l = 0; l < nv; l++){
          // 17 stencil computation
          df[i][j][k][l] =coef[ 0][l] * f[i][j][k][l]
                         + coef[ 1][l] * f[i][j][k][l-1] + coef[ 2][l] * f[i][j][k][l+1]
                         + coef[ 3][l] * f[i][j][k-1][l] + coef[ 4][l] * f[i][j][k+1][l]
                         + coef[ 5][l] * f[i][j-1][k][l] + coef[ 6][l] * f[i][j+1][k][l]
                         + coef[ 7][l] * f[i-1][j][k][l] + coef[ 8][l] * f[i+1][j][k][l]
                         + coef[ 9][l] * f[i][j][k][l-2] + coef[10][l] * f[i][j][k][l+2]
                         + coef[11][l] * f[i][j][k-2][l] + coef[12][l] * f[i][j][k+2][l]
                         + coef[13][l] * f[i][j-2][k][l] + coef[14][l] * f[i][j+2][k][l]
                         + coef[15][l] * f[i-2][j][k][l] + coef[16][l] * f[i+2][j][k][l];
        }
      }
    }
  }
```

1. loop collapse problem (128, 16, 32, 32)

2. Compute 17 coefficients no dependence in k direction (Toroidal asymmetry)

3. Finite difference(4D) Additional memory accesses in the derivative operation in the outer most (i) direction

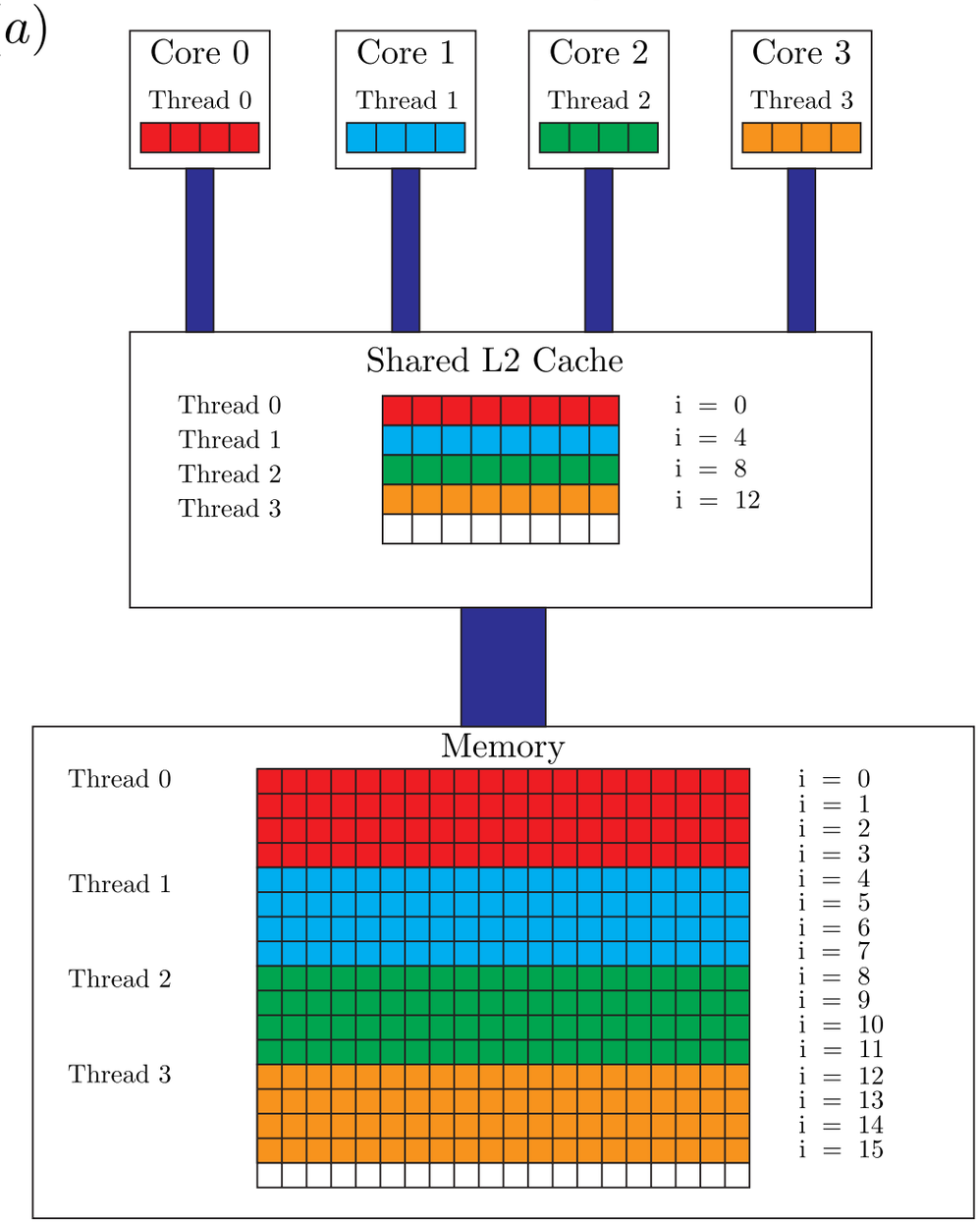
- Reduce the total amount of memory accesses by using the shared cache in conventional CPUs
- Huge latency in **remote L2 cache accessing** across cores on Xeon Phi
- The size of shared cache is not large enough for GPU

Shared cache usage in CPUs

Ex. 2D finite difference $f_{i,j}^{n+1} = C_{00} f_{i,j}^{n} + C_{01} f_{i,j-1}^{n} + C_{02} f_{i,j+1}^{n} + C_{03} f_{i-1,j}^{n} + C_{04} f_{i+1,j}^{n}.$

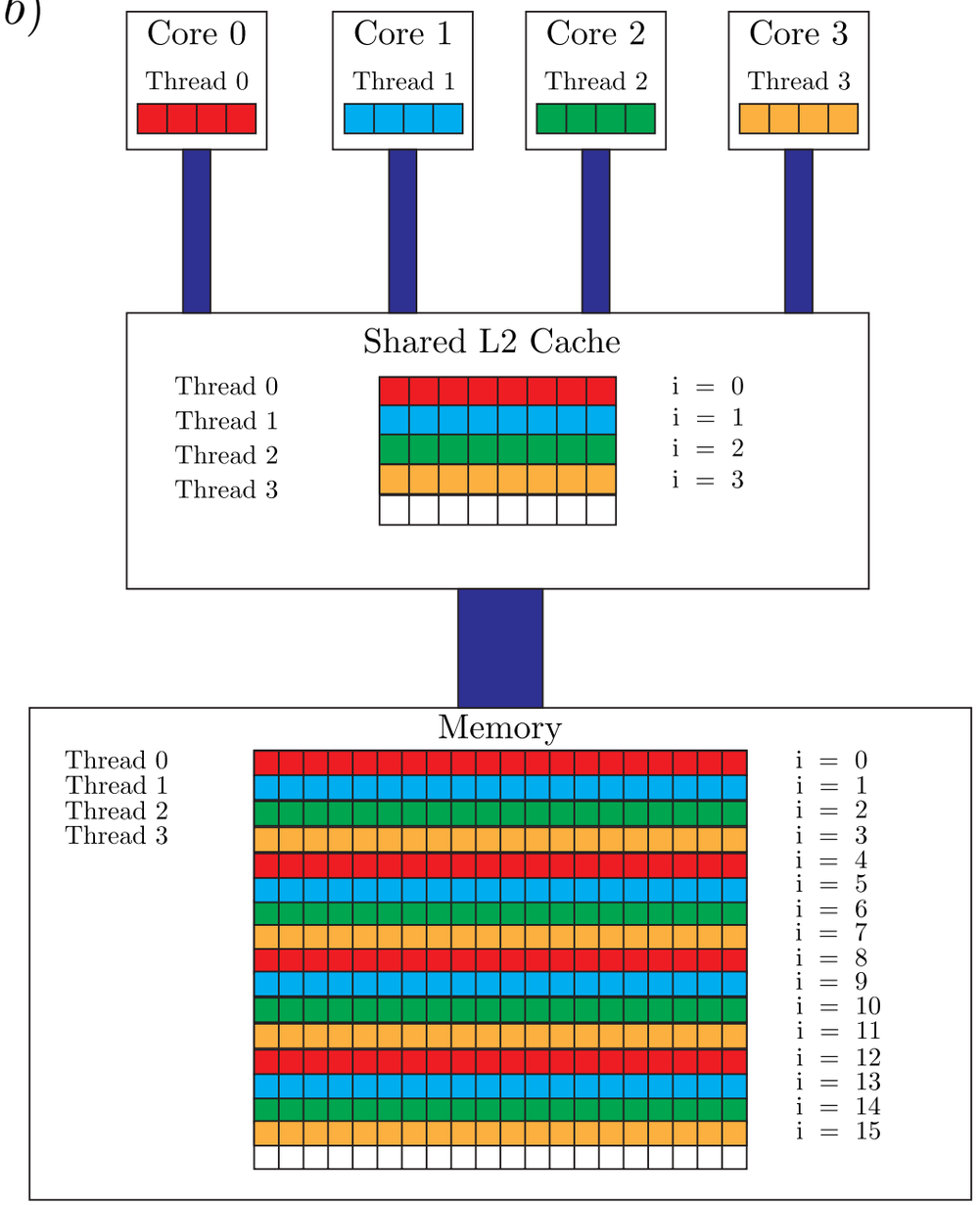
Block decomposition

(a)



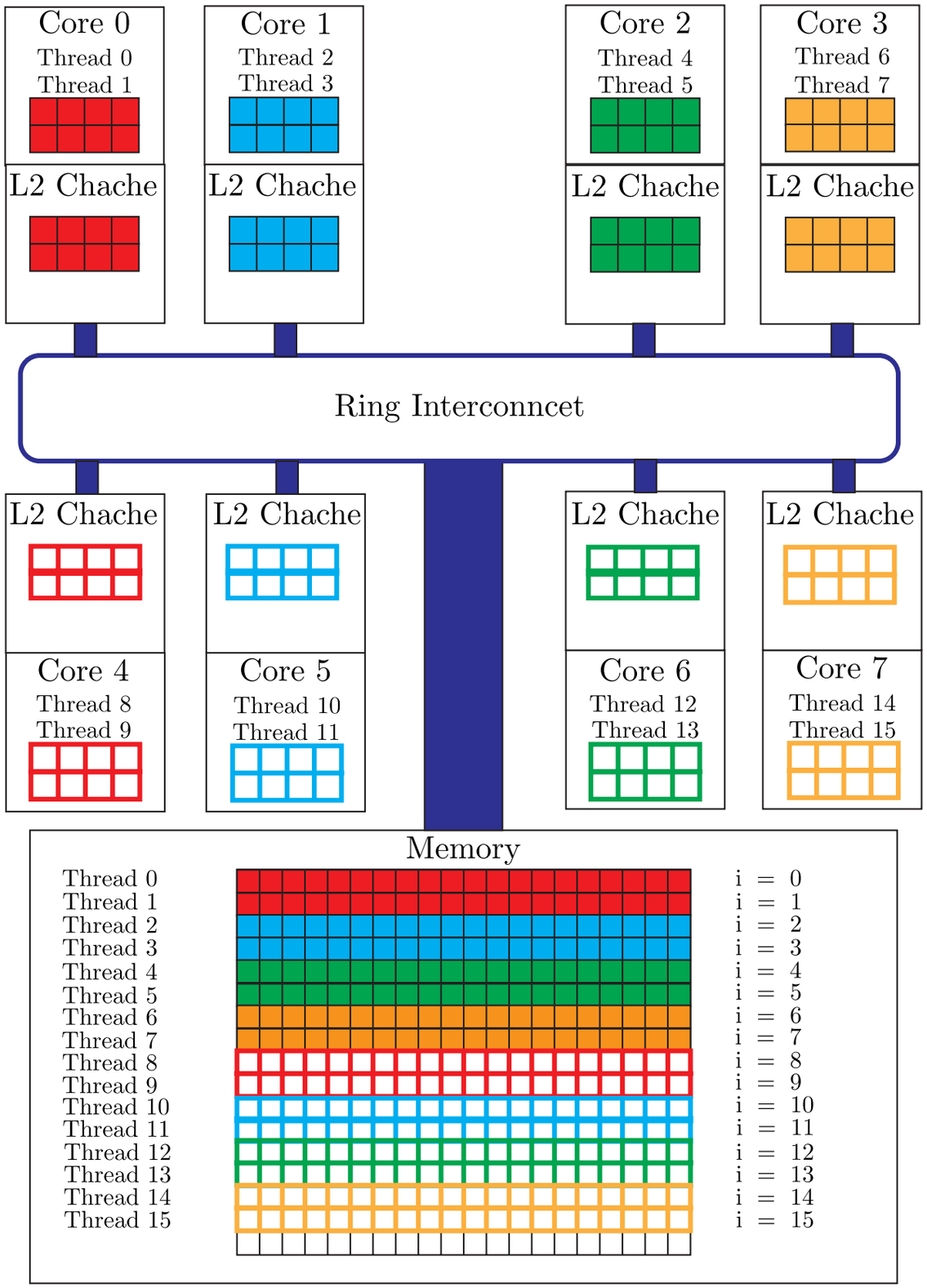
Cyclic decomposition

(b)



- In Cyclic decomposition, adjacent threads can share the data on shared cache (an adjacent thread loads data with the adjacent index)

Shared cache usage in Phi



- Shared L2 cache is distributed over cores on Xeon Phi
- Shared L2 cache accessing is still possible but it involves the cache access across the cores.
  →**remote access latency** [1]
- Efficient cache usage by changing the thread mapping pattern

[1] Jianbin Fang, Ana Lucia Varbanescu, Henk J. Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu, CoRR, 2013.

Cache locality in space and time

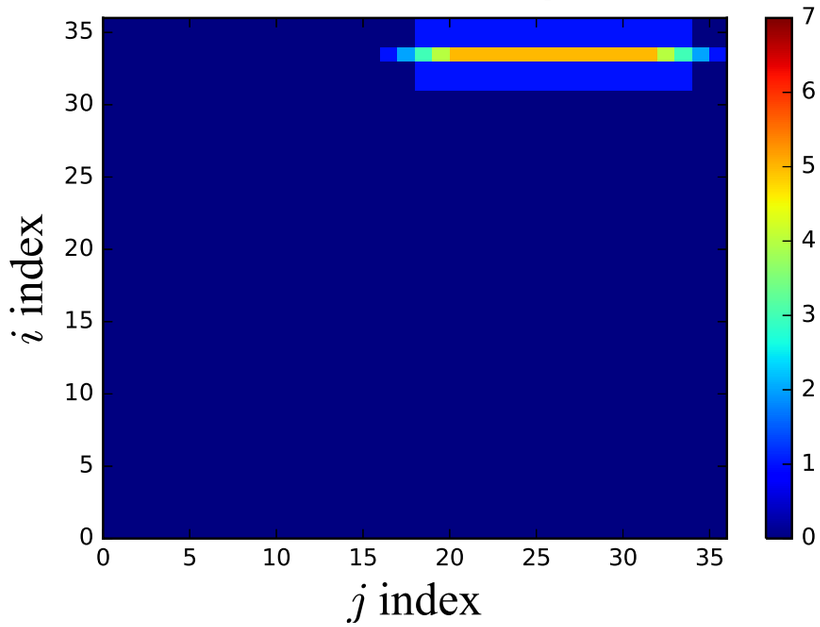
- Considering the memory access pattern to f[l][k][j][i] array
  with the task parallelisation for the two outermost directions (i and j loops)

- Counting the total amounts of memory access to the f[l][k][j][i] for each (i, j) index by a single core with **4 Hyper Threads**. Each core computes pairs of 16 different indices (in combination of i and j) in the current problem size
  → Treated by 4 different sequential steps with 4 Hyper threads

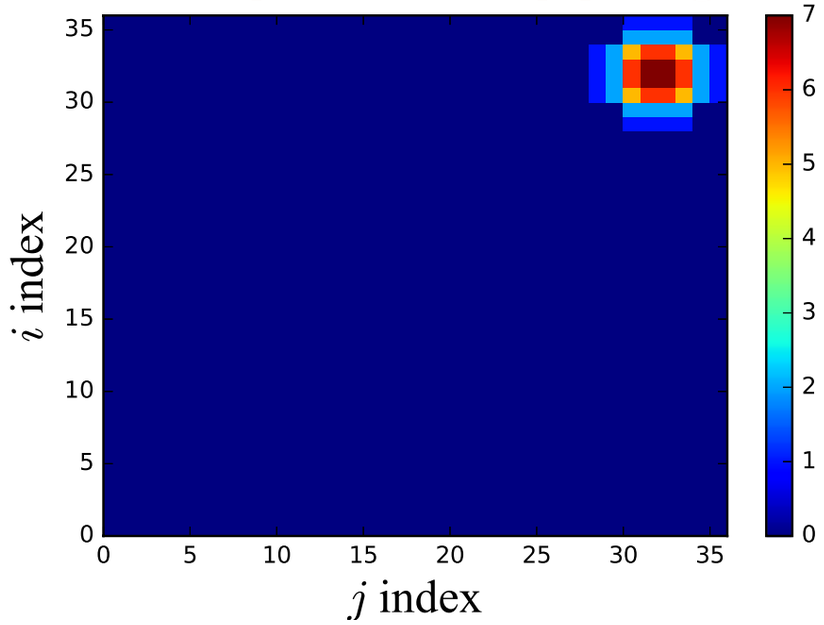
$$C_{i,j} = \sum_m C^m_{i,j} = \sum_m \sum_{tid \in \mathrm{Core}} T^m_{i,j,tid},$$

$T^m_{i,j,tid}$: the total counts of memory access to f[:][:][i][j]
by a single Hyper Thread at the m-th step

Block Decomposition



Explicit mapping



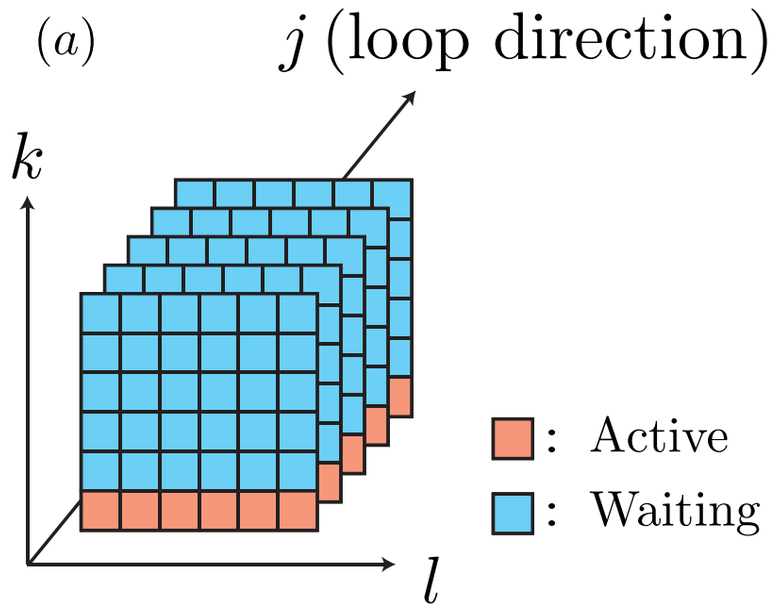
- Explicit mapping → Higher cache locality in space and time

Appropriate thread mapping (GPU)

Original kernel (threads mapped to l, k plane)

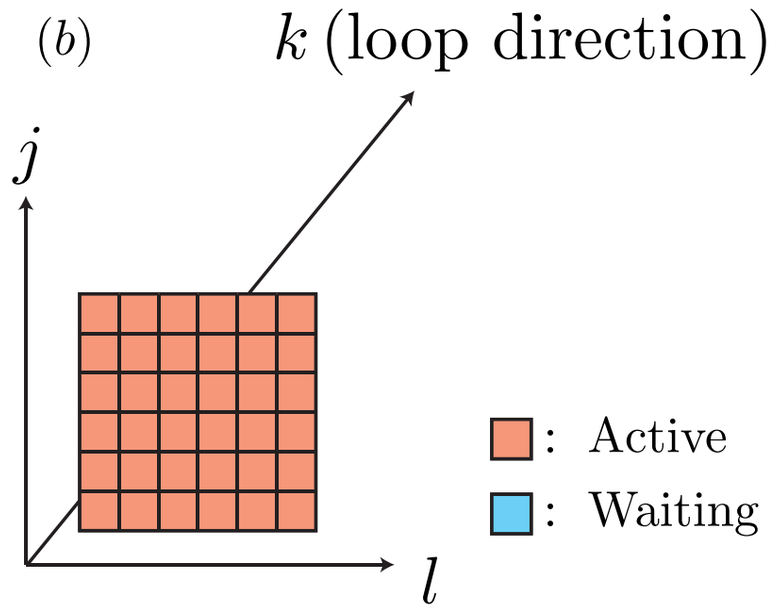
```
1   #define VSIZE 64
2   __shared__ double coef_s[17, VSIZE]; // Shared memory for coefficients
3   l = threadIdx.x+blockIdx.x*blockDim.x;
4   k = threadIdx.y+blockIdx.y*blockDim.y;
5   for(i = 0; i < nx; i++){
6     for(j = 0; j < ny; j++){
7       if(threadIdx.y == 0) compute_coeffs(vx, vy, vz, vv, coef0_s);
8       __syncthreads();
9       compute_finite_diffs(f, df, coef0_s);
10    }
11  }
```

Divergent branch



Optimized kernel (threads mapped to l, j plane)

```
1   l = threadIdx.x+blockIdx.x*blockDim.x;
2   j = threadIdx.y+blockIdx.y*blockDim.y;
3   for(i = 0; i < nx; i++){
4     compute_coeffs(vx, vy, vz, vv, coef0_r, coef1_r, coef2_r, ..., coef16_r);
5     for(k = 0; k < nz; k++){
6       compute_finite_diffs(f, df, coef0_r, coef1_r, coef2_r, ..., coef16_r);
7     }
8   }
```



- Appropriate thread mapping to avoid divergent branch (coefficient computation)

Effective register usage

- Cyclic register usage in the inner most direction

```
1  l = threadIdx.x+blockIdx.x*blockDim.x;
2  j = threadIdx.y+blockIdx.y*blockDim.y;
3  i = blockIdx.z;
4  compute_coeffs(vx, vy, vz, vv, coef0_r, coef1_r, coef2_r, ..., coef16_r);
5  k = 0;
6  fk1_r = f[i][j][k−2][l]; fk2_r = f[i][j][k−1][l];
7  fk3_r = f[i][j][k][l]; fk4_r = f[i][j][k+1][l];
8  for(k = 0; k < nz; k++){
9     fk5_r = f[i][j][k+2][l];
10    df[i][j][k][l] = coef0_r * fk3_r + coef1_r * f[i][j][k][l−1]
11    + coef2_r * f[i][j][k][l+1] + coef3_r * fk2_r + coef4_r * fk4_r
12    + coef5_r * f[i][j−1][k][l] + coef6_r * f[i][j+1][k][l]
13    + coef7_r * f[i−1][j][k][l] + coef8_r * f[i+1][j][k][l]
14    + coef9_r * f[i][j][k][l−2] + coef10_r * f[i][j][k][l+2]
15    + coef11_r * fk1_r + coef12_r * fk5_r + coef13_r * f[i][j−2][k][l]
16    + coef14_r * f[i][j+2][k][l] + coef15_r * f[i−2][j][k][l]
17    + coef16_r * f[i+2][j][k][l];
18    // Slide registers in k direction
19    fk1_r = fk2_r; fk2_r = fk3_r; fk3_r = fk4_r; fk4_r = fk5_r;
20 }
```

5 stencils in k direction are kept on registers (registers are **used cyclically**)

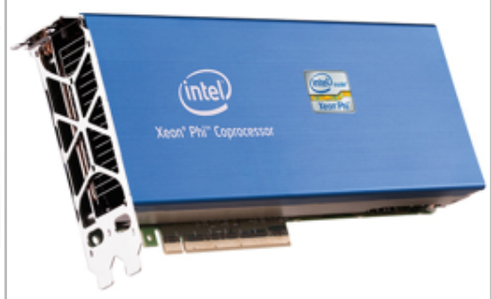
- Loop unrolling in i direction

Reduce additional memory accesses for derivative in i direction

Outline

- Introduction
  - Demands for exa-scale supercomputer
  - Semi-Lagrangian and Finite-Difference kernels from GYSELA and GT5D
- Optimization strategies
  - Semi-Lagrangian kernel on Xeon Phi and GPGPU
  - Finite-Difference kernel on Xeon Phi and GPGPU
- Summary
  - Acceleration ratio of kernels
  - Summary for optimization strategies on Xeon Phi and GPGPU

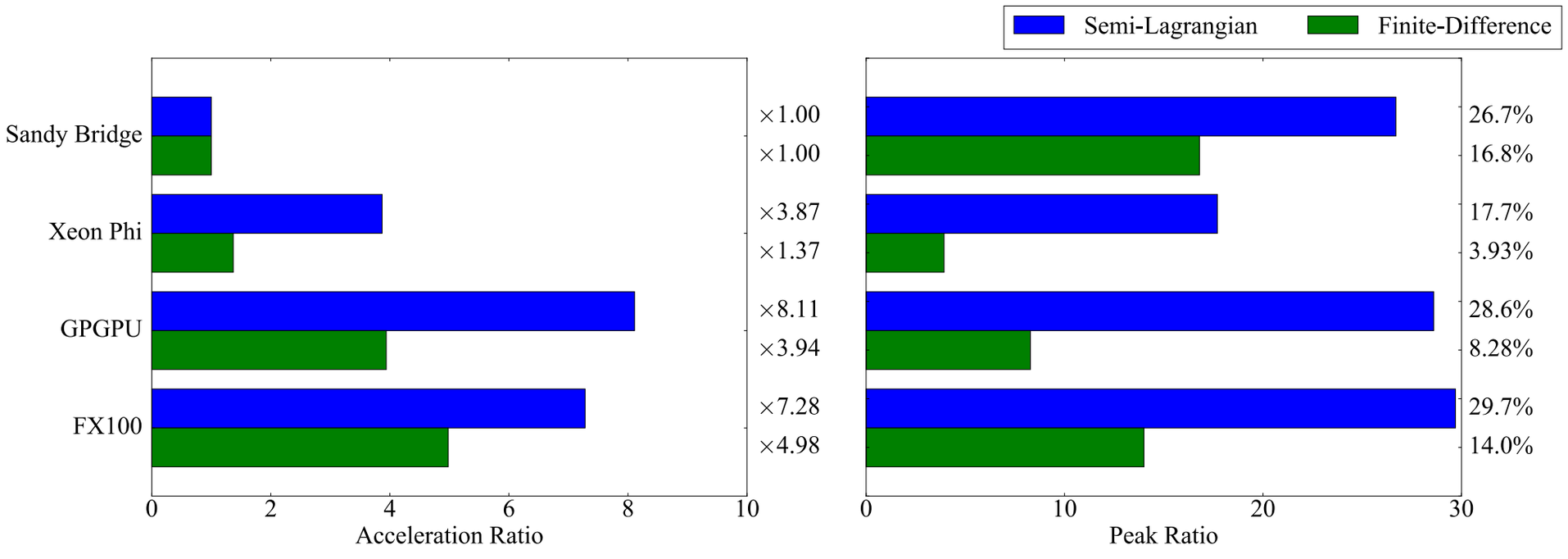
Testbed description

	Sandy Bridge	FX100	Xeon Phi	GPU
				
CPU / GPU	SandyBridge-Ep	SPARC64XIfx	Xeon Phi 5110P	Tesla K-20X
Compiler	intel compiler 13.1.3	Fujitsu compiler 2.0	intel compiler 13.1.3	pgfortran 15.7
Cores (DP)	8	32 + 2	60	896
Cache [MB]	20	24	0.5 x 60	1.5
GFlops (DP)	172.8	**1000**	**1010**	**1310**
STREAM GB/s	40	320	160	180
SIMD width	4 (AVX)	4	8	N/A
Power efficiency [MFlops/W]	562	1910	1501	2973
Flop/Byte	4.32	**3.13**	**6.31**	**7.28**

- Adding FX100 as a reference for acceleration, which has a **comparable performance** to accelerators.

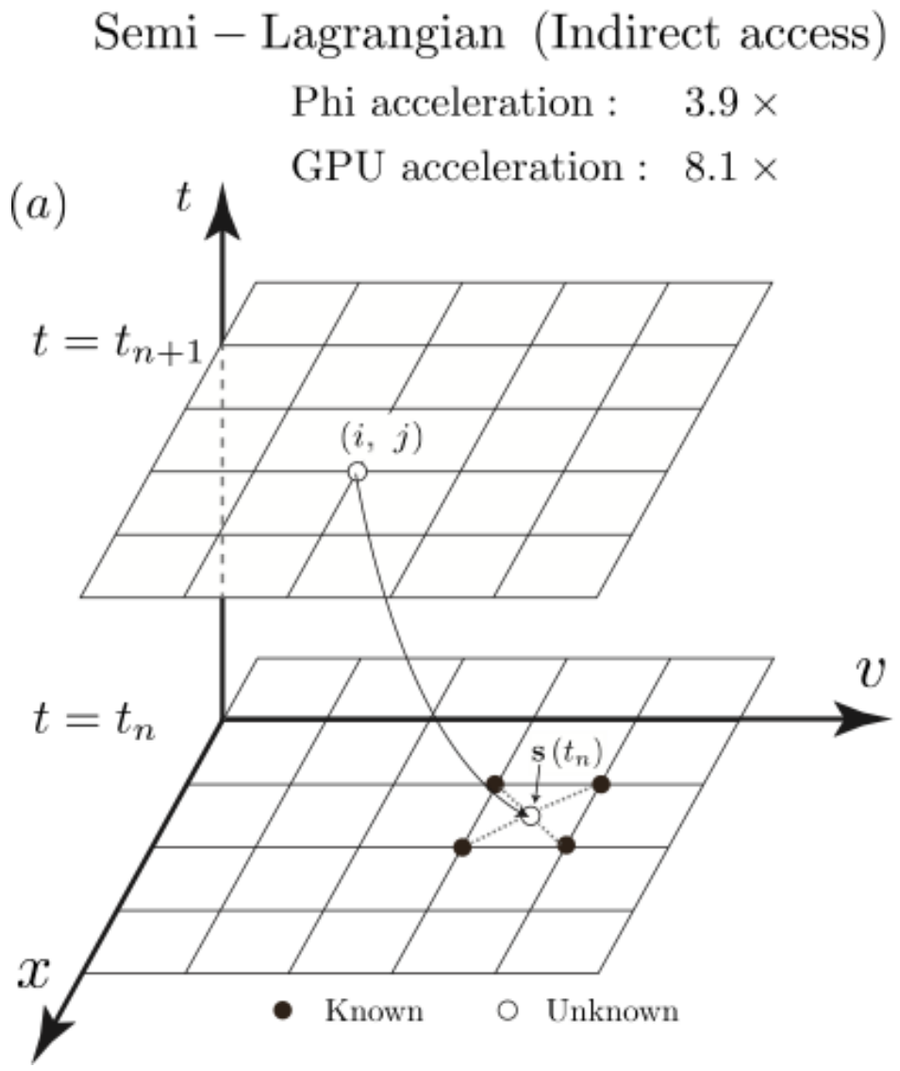
Acceleration of kernels

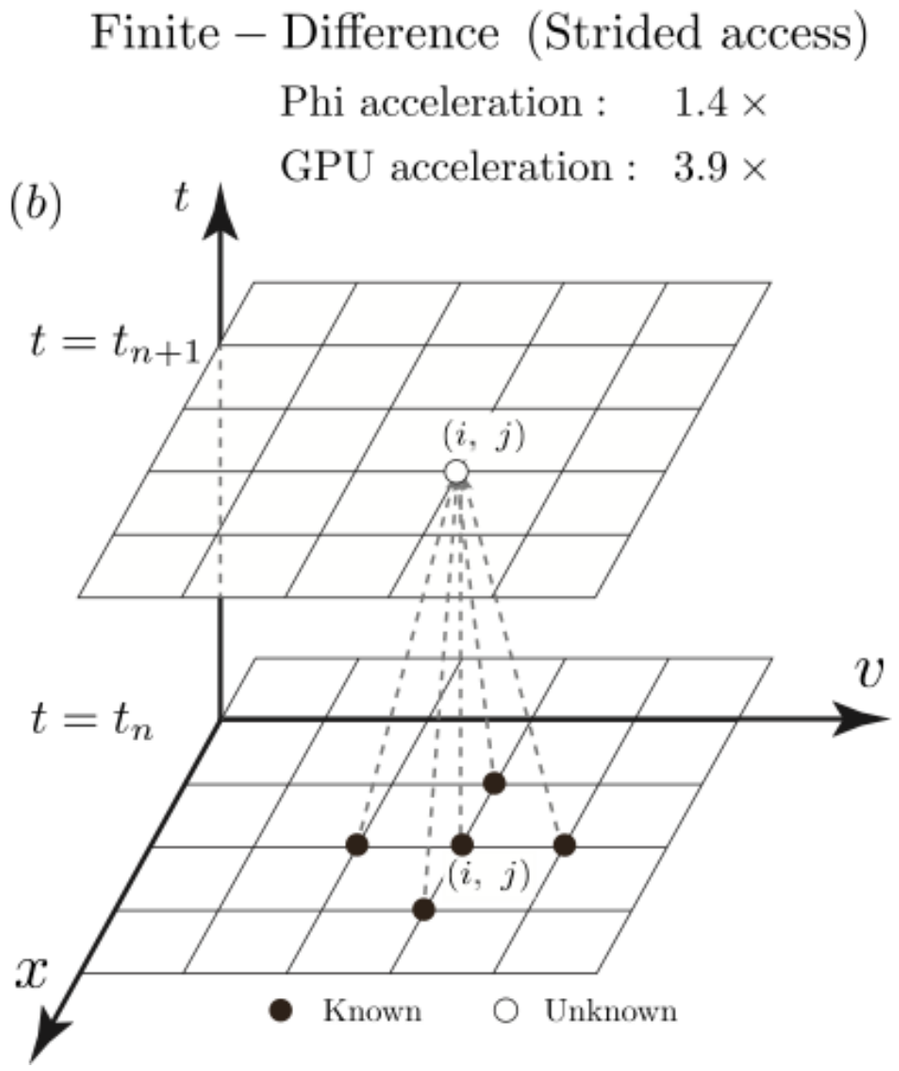
SL	Time [s]	GFlops (%)	GB/s (%)	Acceleration
Sandy	25.43	46.1 (26.7)	16.49 (41.2)	-
Xeon Phi	6.577	178.4 (17.7)	65.41 (40.9)	3.87
GPGPU	3.135	374.2 (28.6)	130.63 (72.6)	8.11
FX100	3.494	301.3 (29.7)	117.21 (36.6)	7.28
FD	Time [s]	GFlops (%)	GB/s (%)	Acceleration
Sandy	0.453	28.96 (16.8)	39.21 (98.0)	-
Xeon Phi	0.331	39.75 (3.93)	66.61 (41.6)	1.37
GPGPU	0.115	108.5 (8.28)	153.9 (85.5)	3.94
FX100	0.091	141.3 (13.97)	189.9 (59.3)	4.98



- Both kernels show good performance on GPGPU
- Low performance of SL kernel on Phi could be due to **indirect access**
- Low performance of FD kernel on Phi could be due to memory bound feature

Summary for optimization strategies





Xeon Phi

- **AoSoA** layout for SIMD load (SL)
- Improve **load balancing** by dynamic scheduling (SL)
- **High cache locality** in space and time (SL, FD)

GPGPU

- AoSoA layout for coalescing (SL)
- **Texture memory** for indirect accessing (SL)
- Effective **register** usage (FD)