thrive SIGGRAPH2019
LOS ANGELES · 28 JULY – 1 AUGUST

On Hybrid Lagrangian-Eulerian Simulation Methods

Practical Notes and High-Performance Aspects

http://mpm.graphics

Yuanming Hu[1]   Xinxin Zhang[2]   Ming Gao[2]   Chenfanfu Jiang[3]

[1]MIT CSAIL   [2]Tencent   [3]University of Pennsylvania

Speakers

Yuanming Hu

PhD student

TAICHI
COMPUTER GRAPHICS R&D INFRASTRUCTURE

Dr. Xinxin Zhang

Senior Graphics RnD

UBC

AICFVE
未来影像高精尖创新中心
Advanced Innovation Center For Future Visual Entertainment

Tencent

Dr. Ming Gao

Senior RnD researcher

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Penn
UNIVERSITY of PENNSYLVANIA

Tencent

Dr. Chenfanfu Jiang

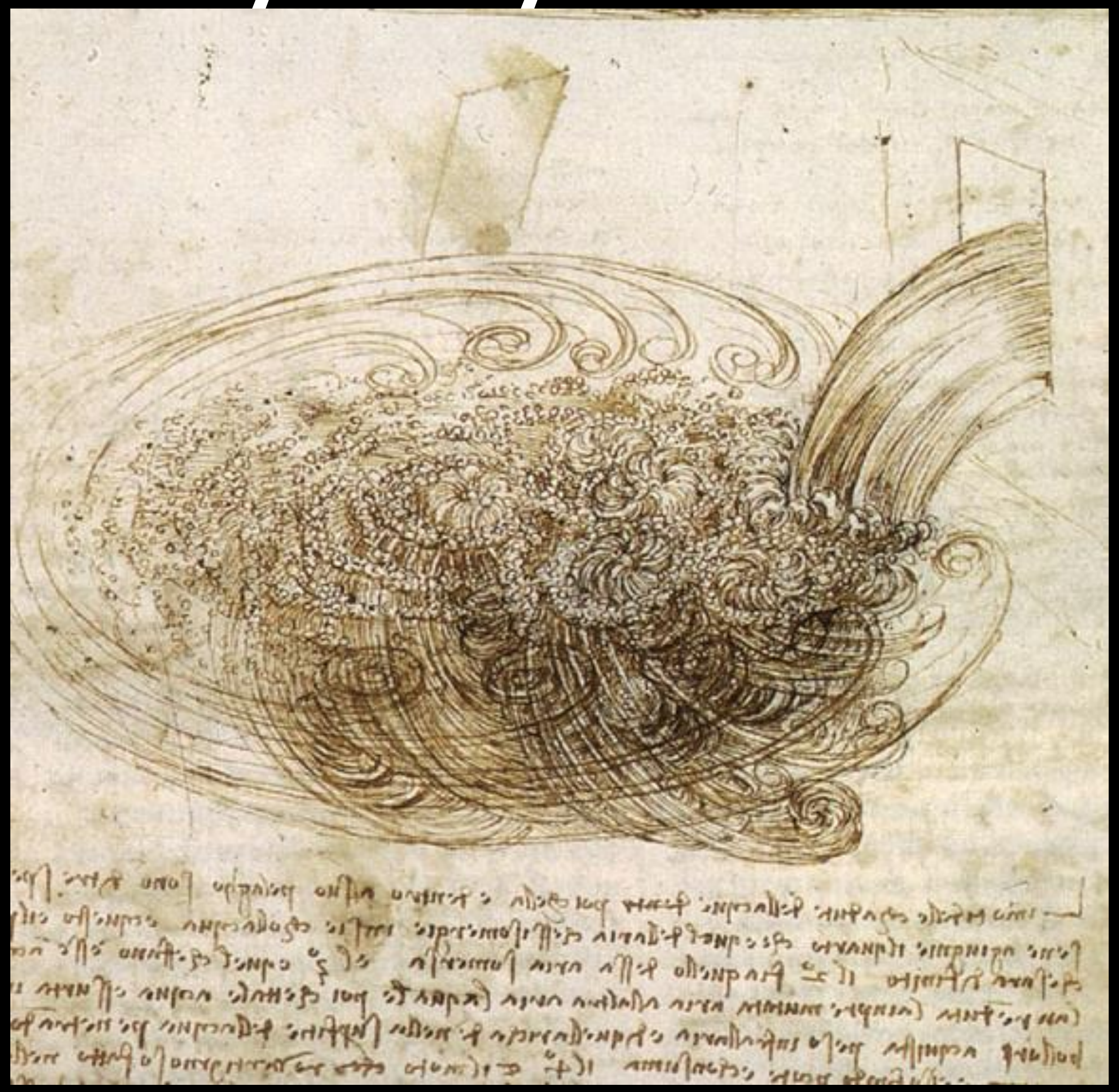
Assistant professor

THE UNIVERSITY OF CALIFORNIA
LET THERE BE LIGHT
UCLA

Penn
UNIVERSITY of PENNSYLVANIA

Particle-Particle Particle-Mesh Method for fast N-Body dynamics in Eulerian-Lagrangian Computations

N-Body Dynamics



$$u_i = \sum_{j=1, j\neq i}^{N} \frac{v_j \omega_j \times (x_i - x_j)}{4\pi \left\| x_i - x_j \right\|_2^3}$$

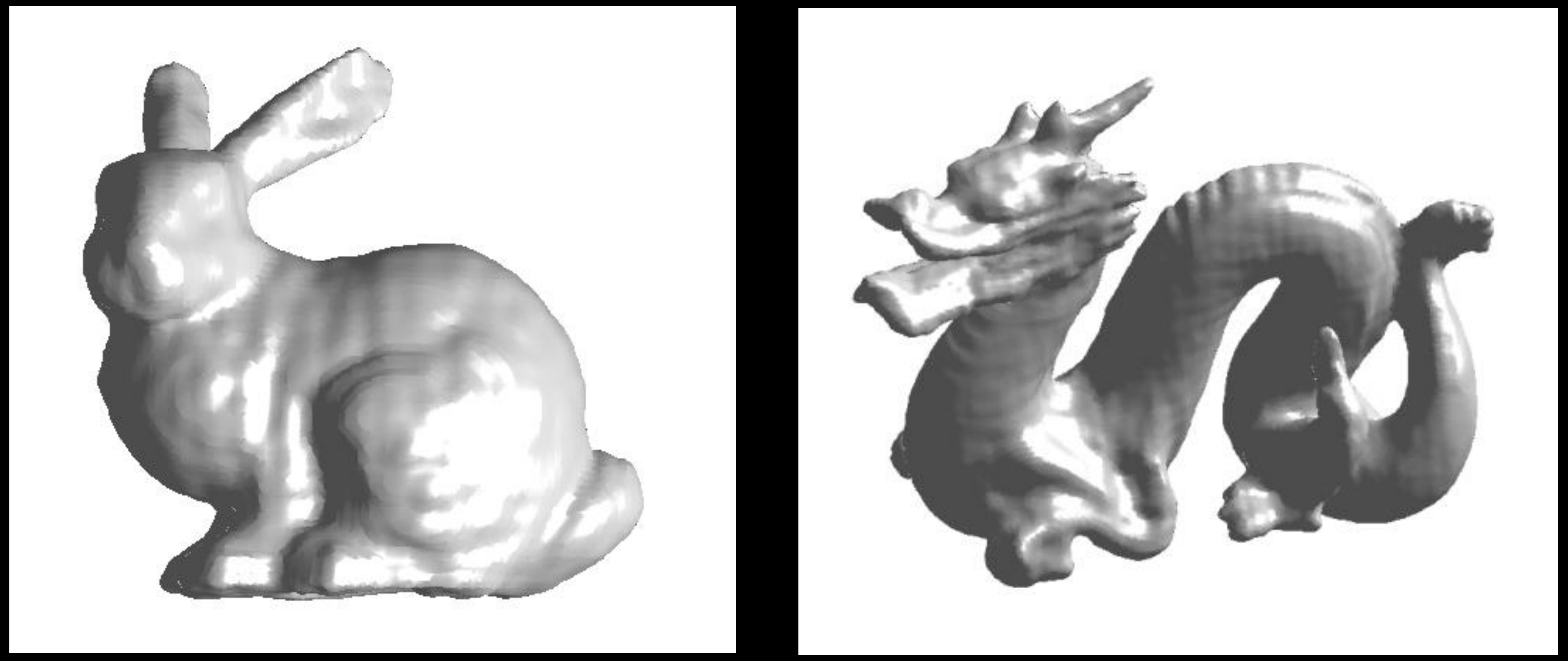
$$f_i = -\epsilon \sum_{j=1, j\neq i}^{N} \frac{\rho_j v_j \, (x_i - x_j)}{4\pi \left\| x_i - x_j \right\|_2^3}$$

N-Body Dynamics

$$u_i = \sum_{j=1, j \neq i}^{N} \frac{v_j \omega_j \times (x_i - x_j)}{4\pi \left\| x_i - x_j \right\|_2^3}$$

Given N particles and M evaluation position, direct computation requires O(NM) time!

$$f_i = -\epsilon \sum_{j=1, j \neq i}^{N} \frac{\rho_j v_j \, (x_i - x_j)}{4\pi \left\| x_i - x_j \right\|_2^3}$$



Surface reconstruction.

On the Accurate Large-scale Simulation of Ferrofluids. Huang et. al. SIGGRAPH 2019

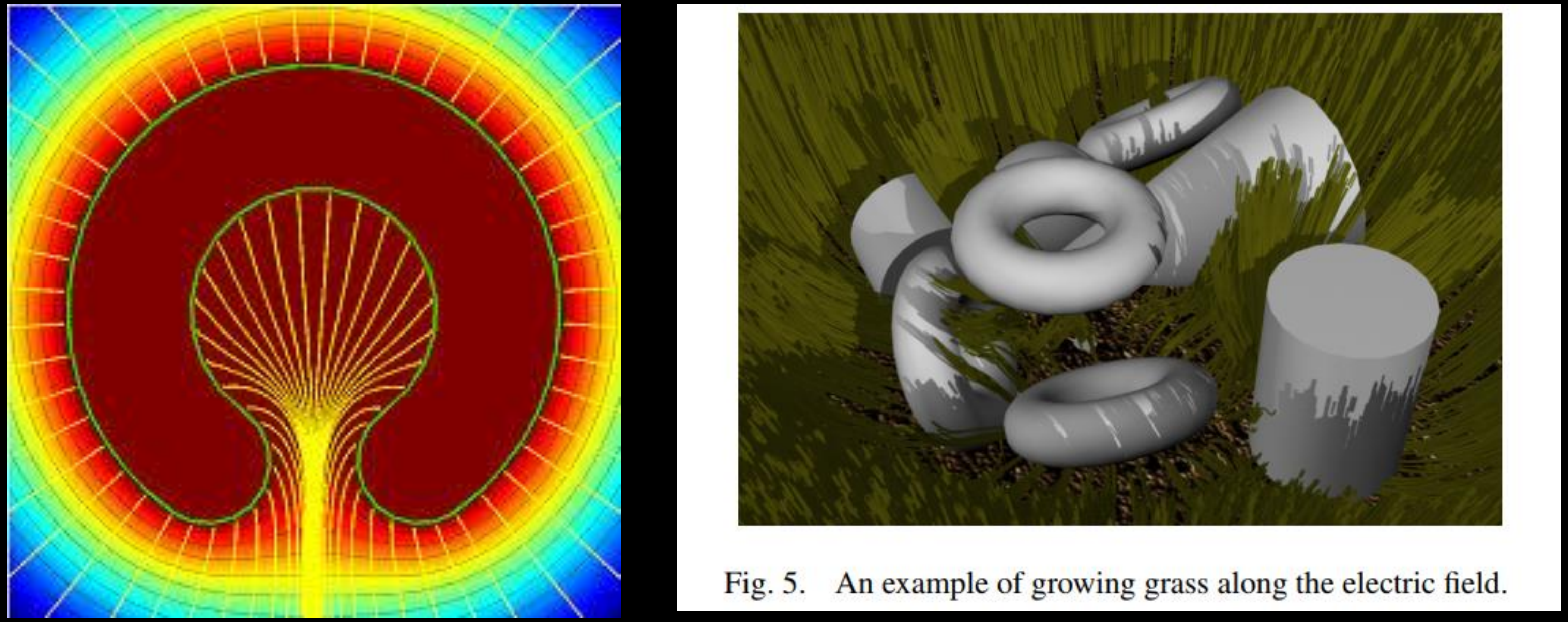
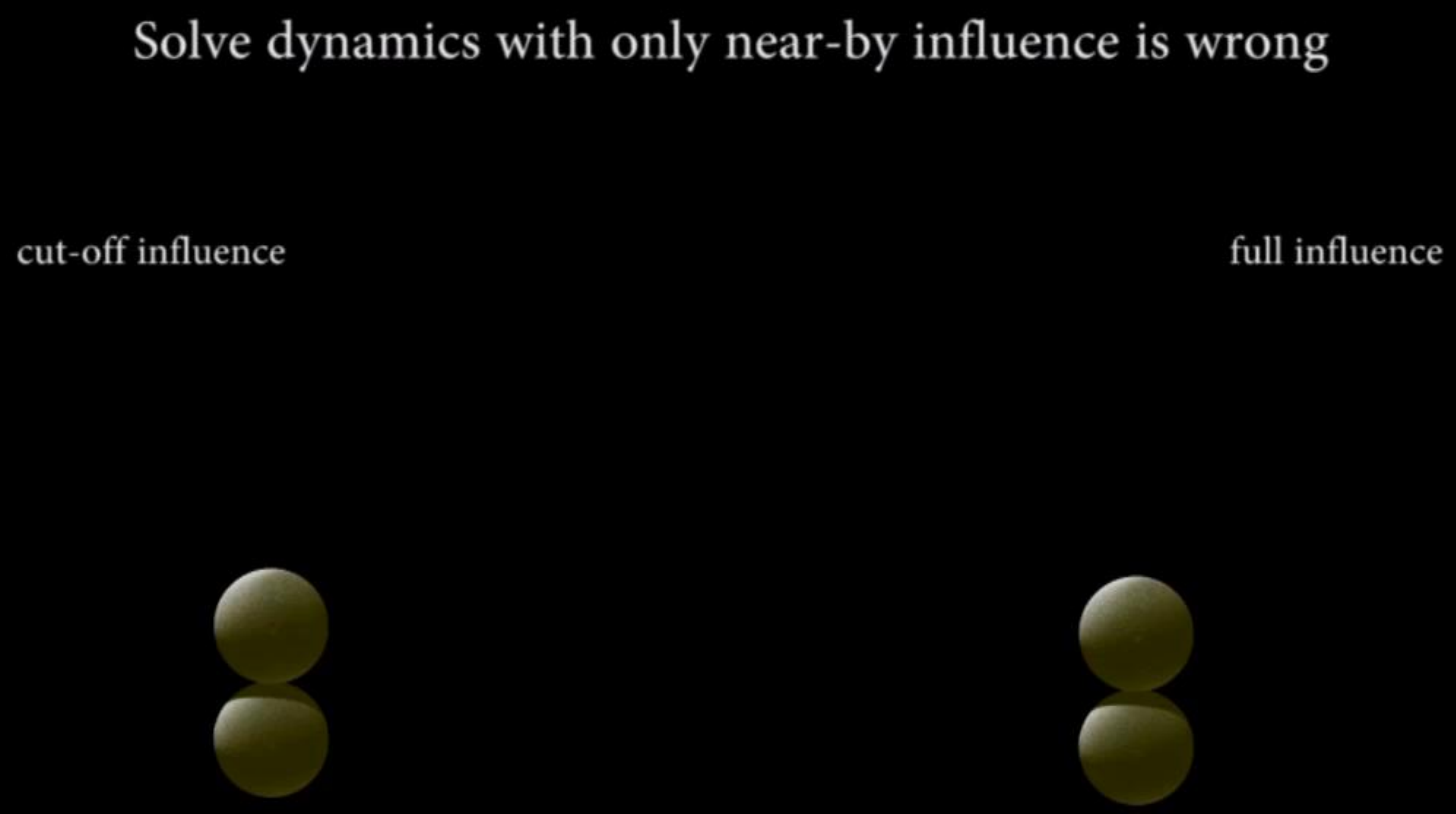


Fig. 5. An example of growing grass along the electric field.

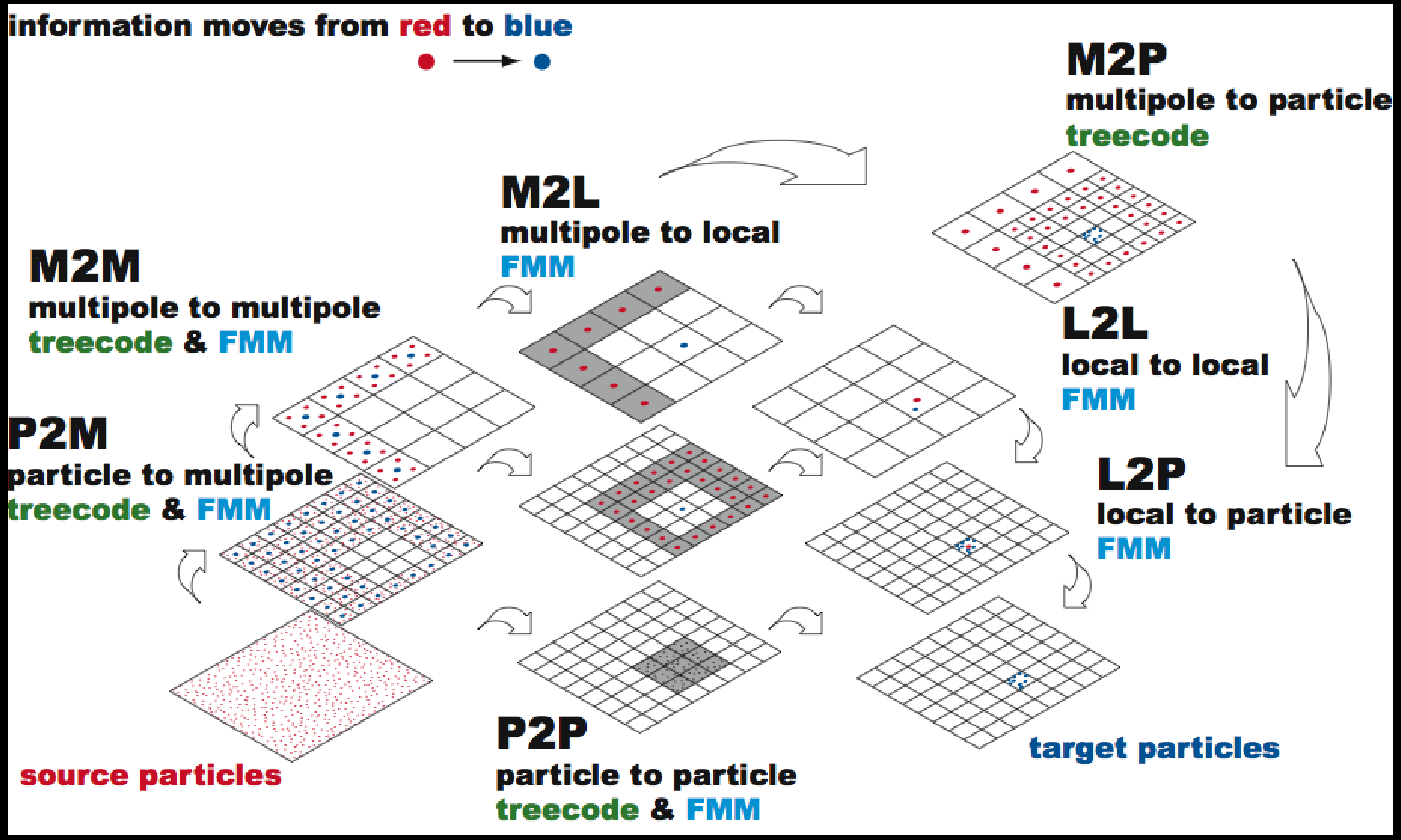
Harmonic Parameterization by Electrostatics. Wang et. al. ACM TOG

Fast N-body Summation is Non-Trivial



Fast N-body Summations

- Solutions have been widely discovered to reduce this computation bottleneck.

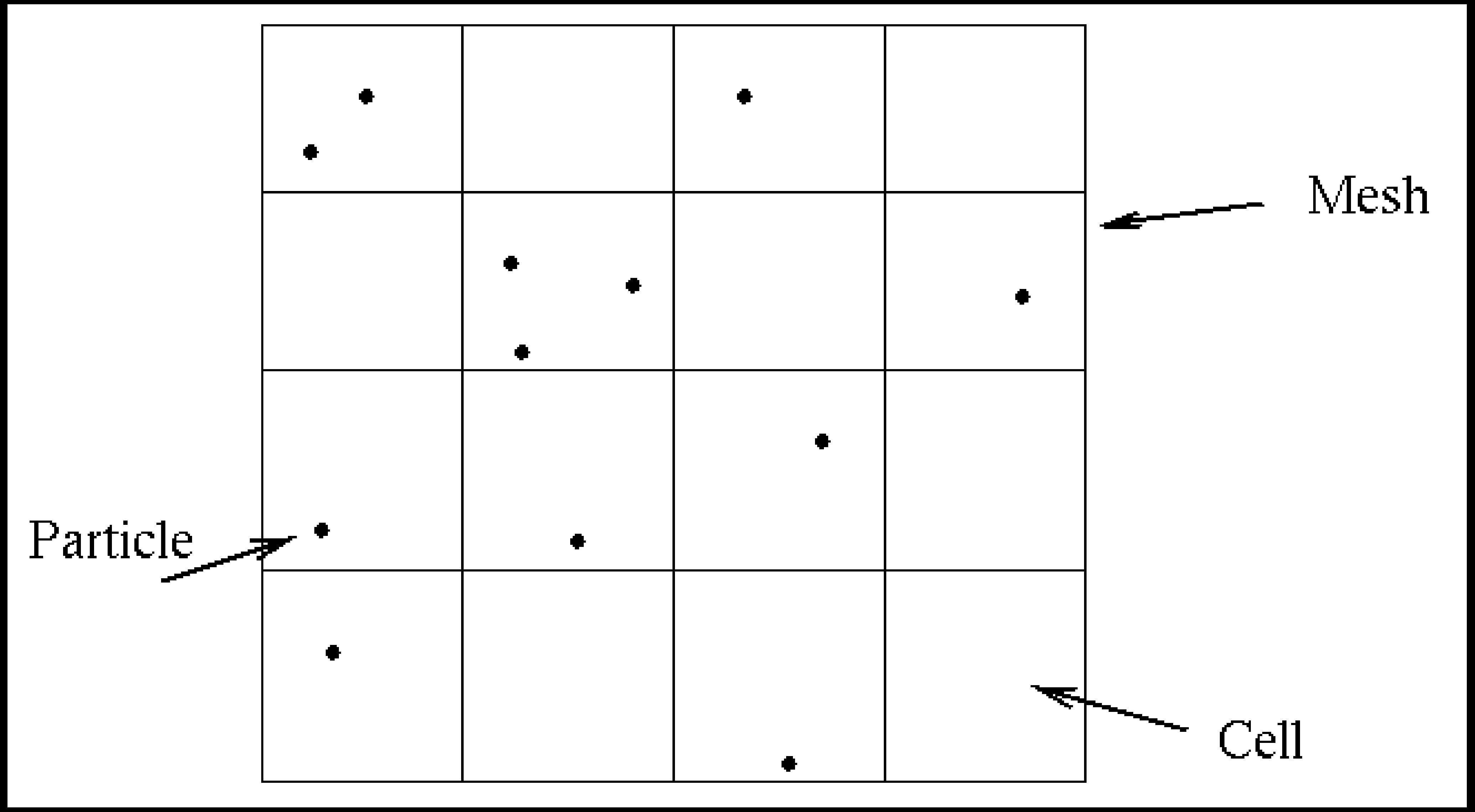


O(N) Fast Multipole Method(Rokhlin & Greengard)

Accuracy	Efficiency	Code Complexity
Excellent	Good	Fair

Fast N-body Summations

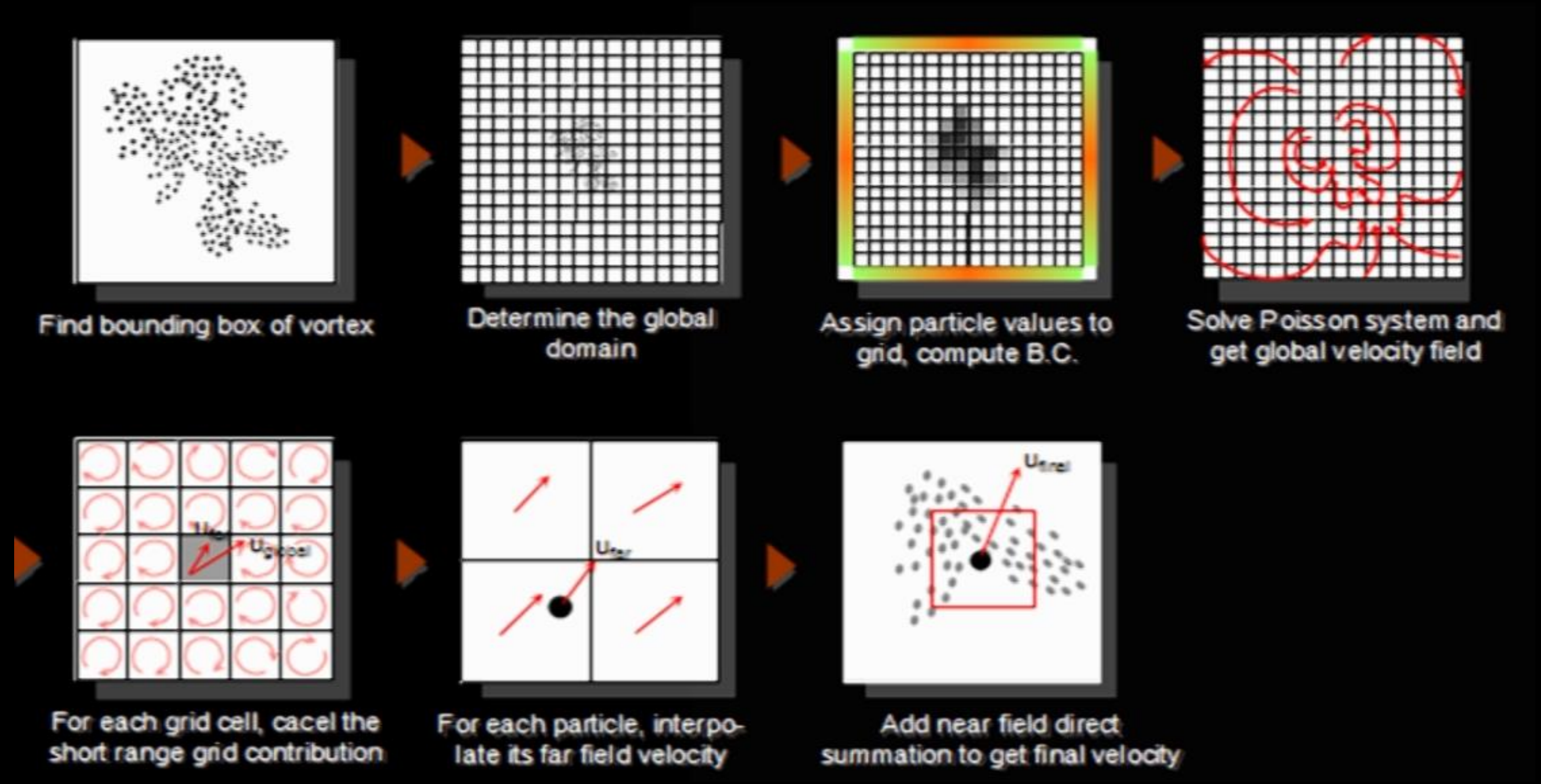
- Solutions have been widely discovered to reduce this computation bottleneck.



O(G) Particle Mesh Methods, Vortex-In-Cell.

Accuracy	Efficiency	Code Complexity
Fair	Excellent	Excellent

Fast N-body Summations: Particle-Particle, Particle-Mesh(PPPM)



Accuracy	Efficiency	Code Complexity
Good	Excellent	Excellent

PPPM: Key idea

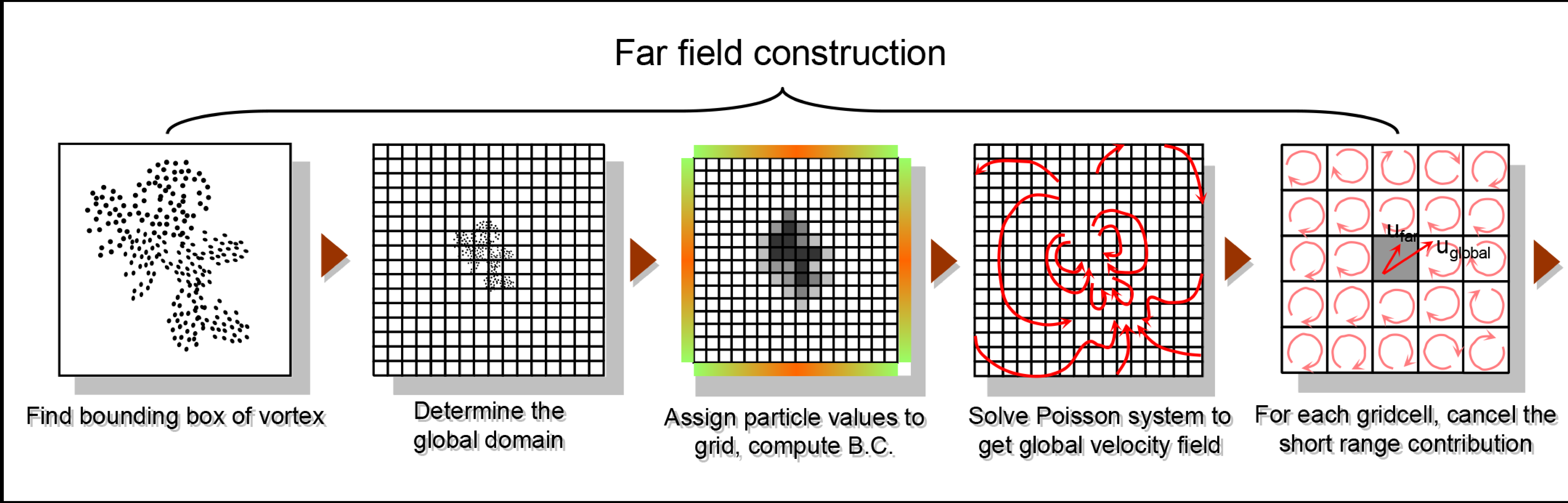
$$u_i = \sum_{j=1, j \neq i}^{N} \frac{v_j \omega_j \times (x_i - x_j)}{4\pi \|x_i - x_j\|_2^3} \quad \Longleftrightarrow \quad \begin{aligned} \nabla^2 \psi &= -\omega \\ u &= \nabla \times \psi \end{aligned}$$

$$f_i = -\epsilon \sum_{j=1, j \neq i}^{N} \frac{\rho_j v_j (x_i - x_j)}{4\pi \|x_i - x_j\|_2^3} \quad \Longleftrightarrow \quad \begin{aligned} \nabla^2 \phi &= \epsilon \rho \\ f &= \nabla \phi \end{aligned}$$

Direct summation for the turbulent part ⟺ Poisson's Equation for the smooth part

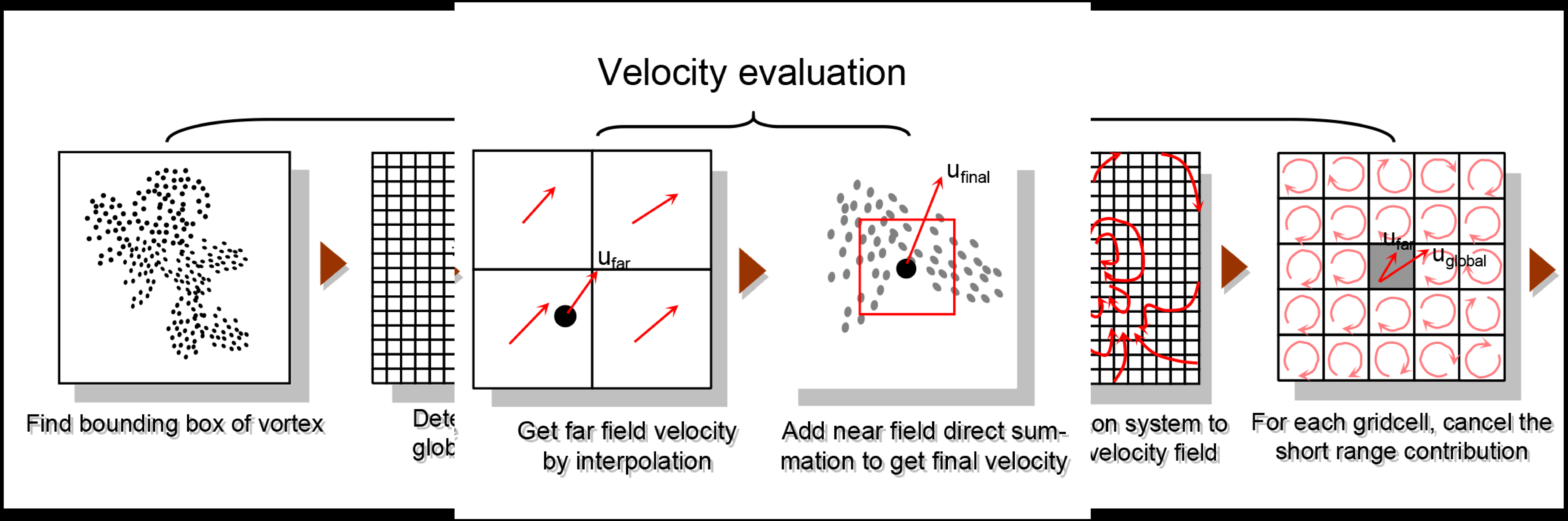
PPPM

- Fast solution uses near-far decomposition to get acceleration. Can we do similar thing on a particle-mesh setup?



PPPM

- Fast solution uses near-far decomposition to get acceleration. Can we do similar thing on a particle-mesh setup?



PPPM

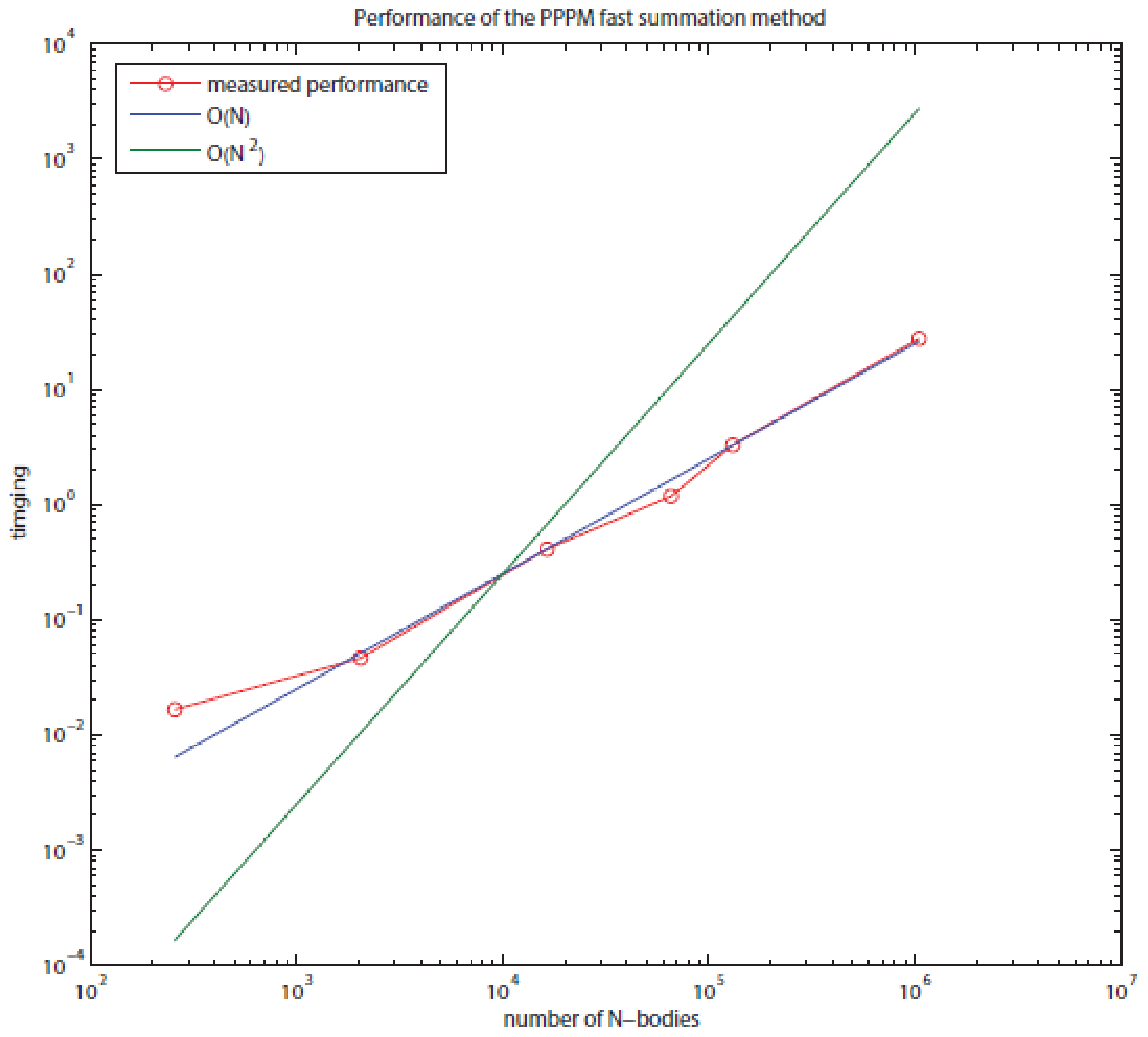


Figure 4: *Performance of the PPPM fast summation. Computation time grows linearly with the number of computational elements.*

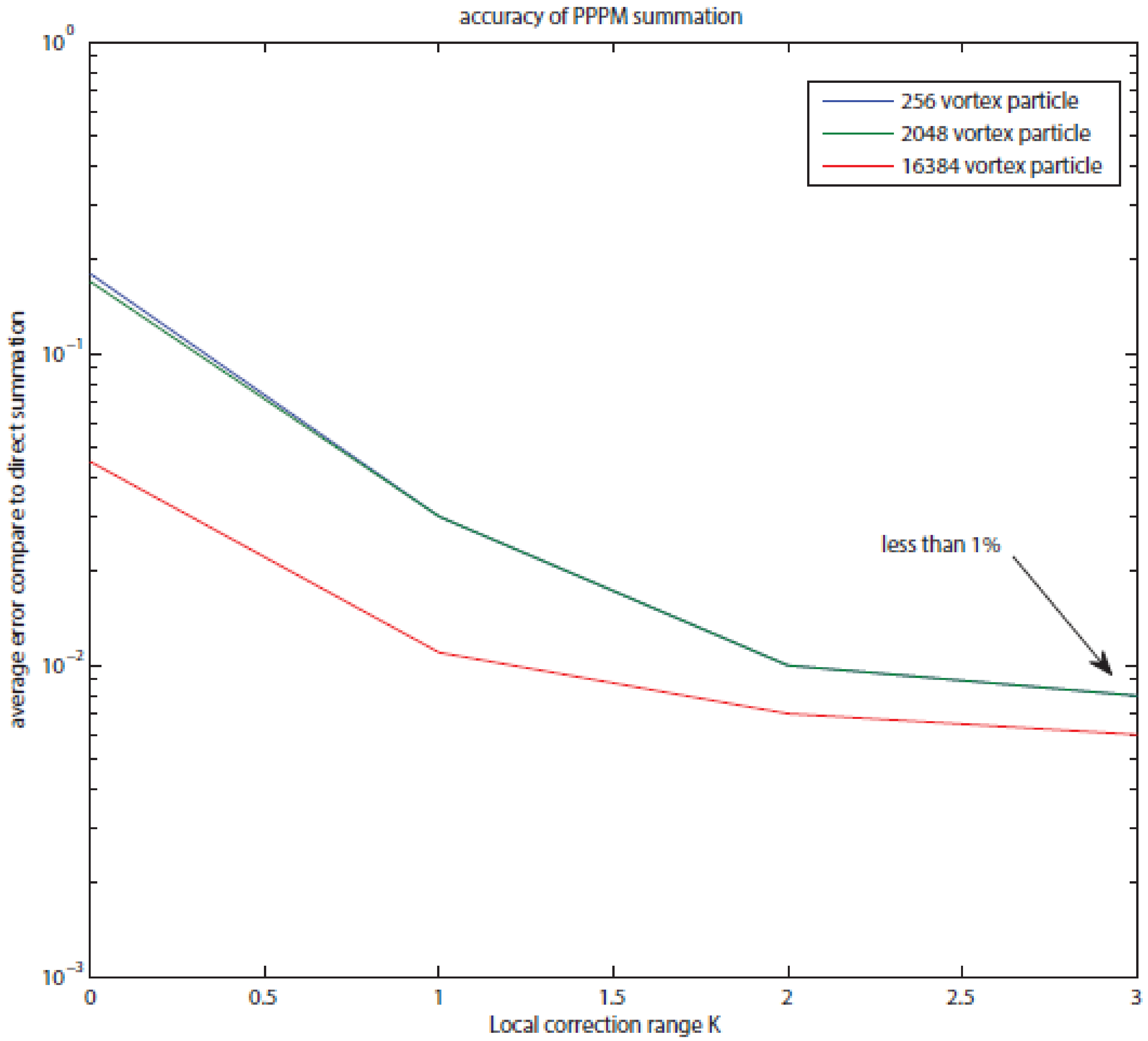


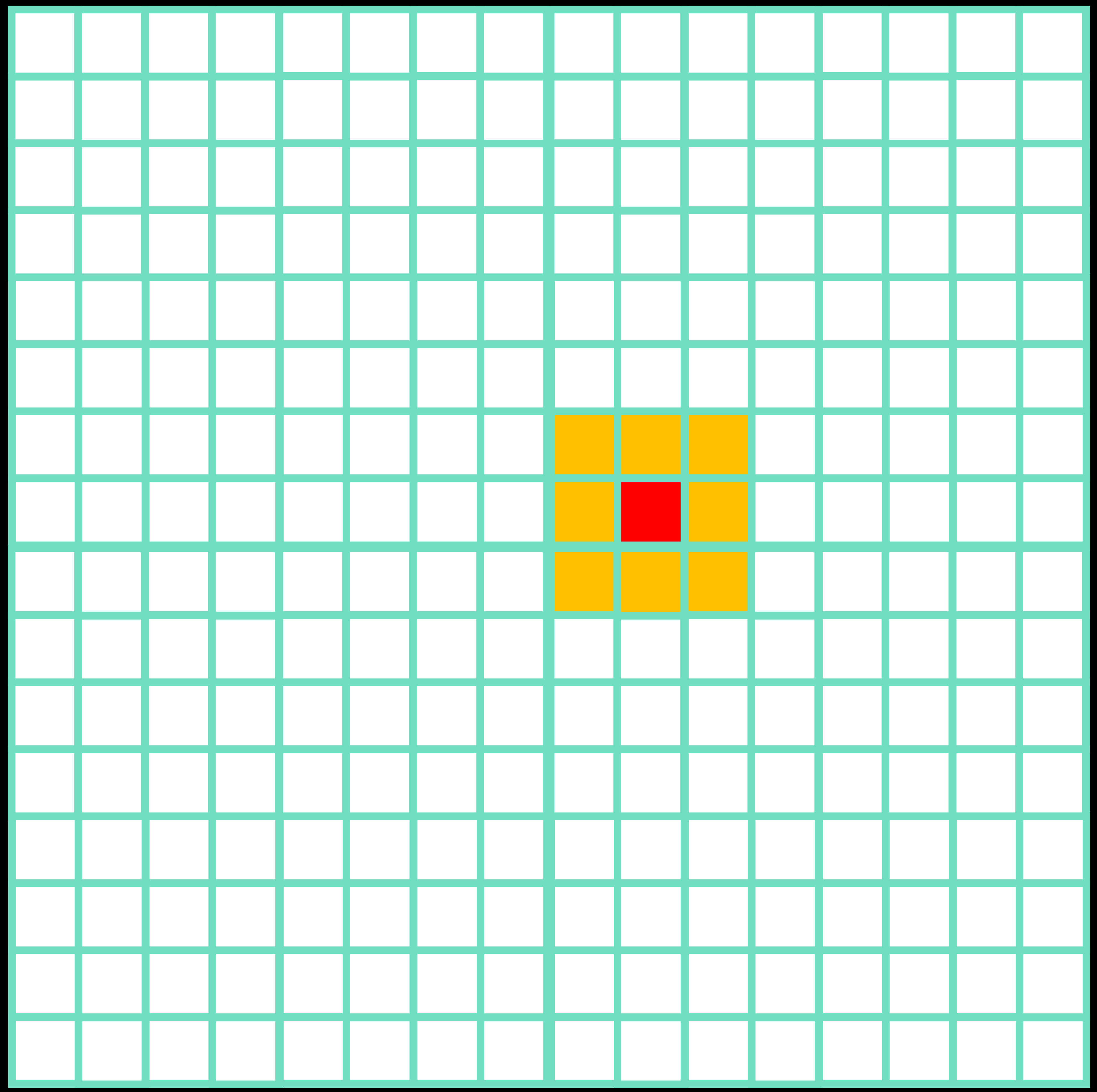
Figure 5: *Accuracy statistics of the PPPM fast summation.*

Local Correction

- In 3D, for a correction window of size K in each dimension, a local matrix of size $K^3 \times K^3$ can be precomputed to cancel the local influence from grid.

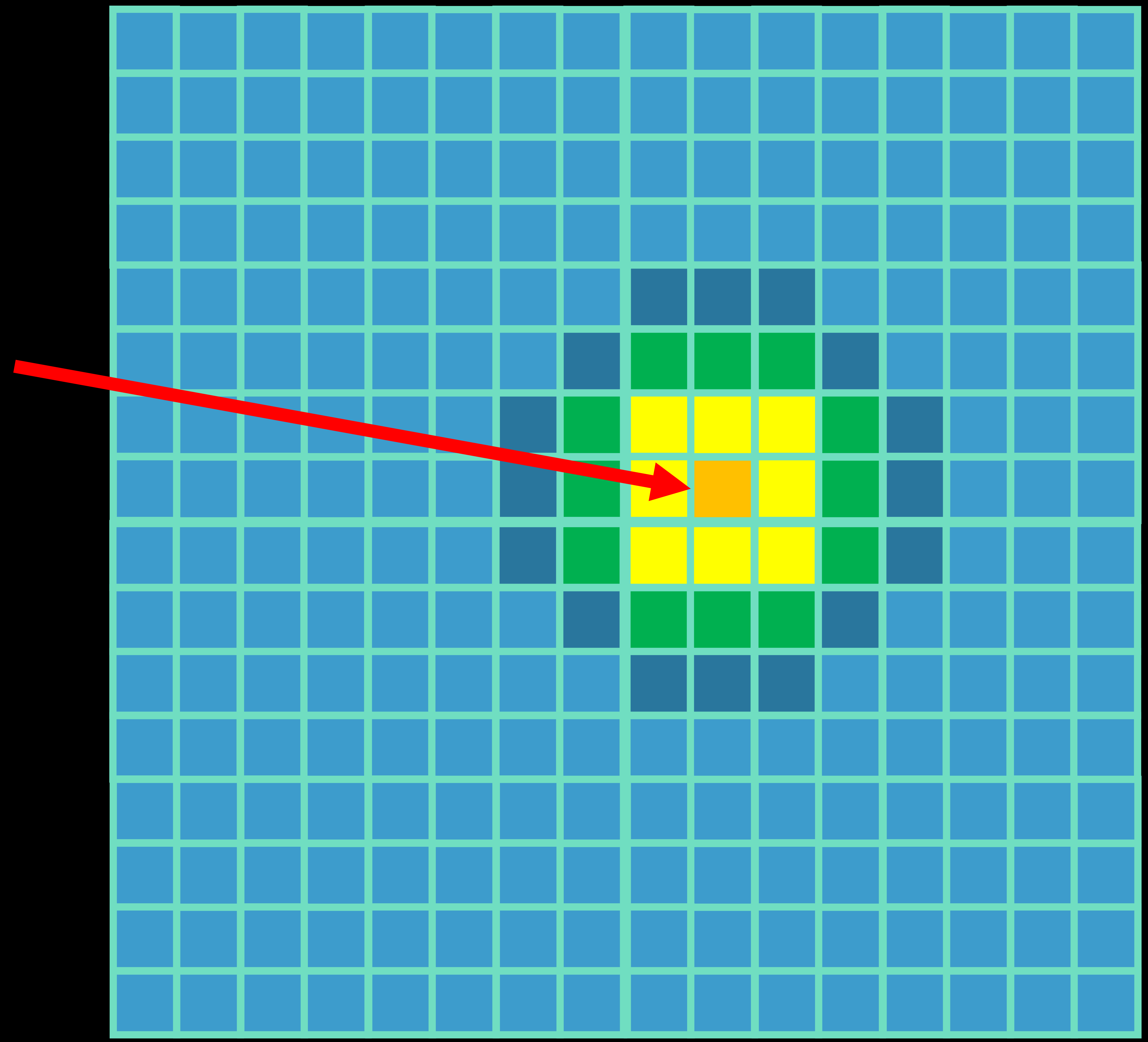
- T(N) = O(c K^6 N)

Local Correction



Local Correction

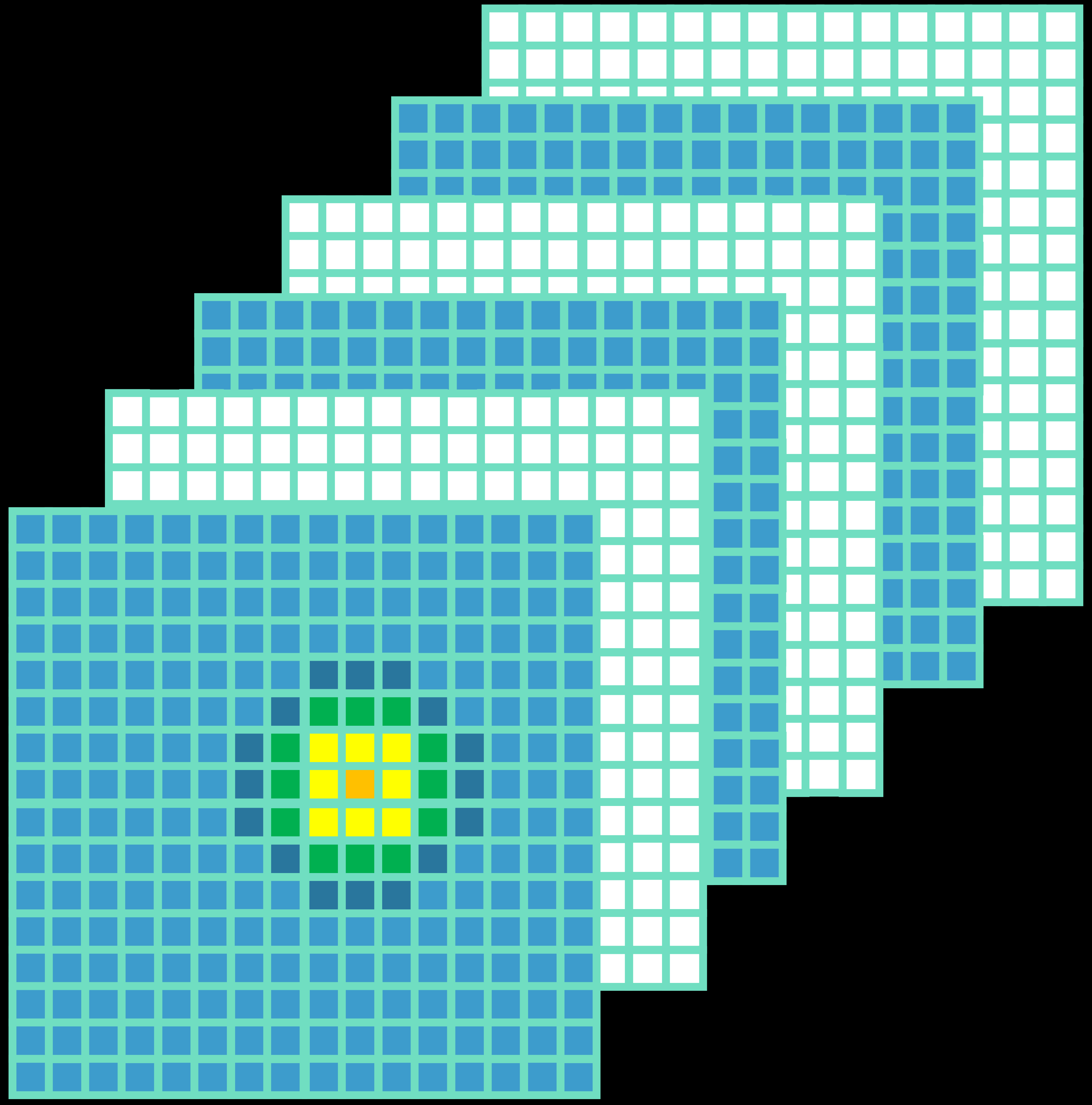
The influence made by neighbor cells.



Local Correction

- *The matrix inverse reveals how the center cell's value depends linearly on its neighbors(including itself).*

$$s_c = \sum_{j \in \eta} a_j r_j$$



One interesting finding

$$a_j \approx G_{cj} = \frac{1}{4\pi \left\| x_c - x_j \right\|_2}$$

Even Simpler PPPM

- Decompose the velocity field as

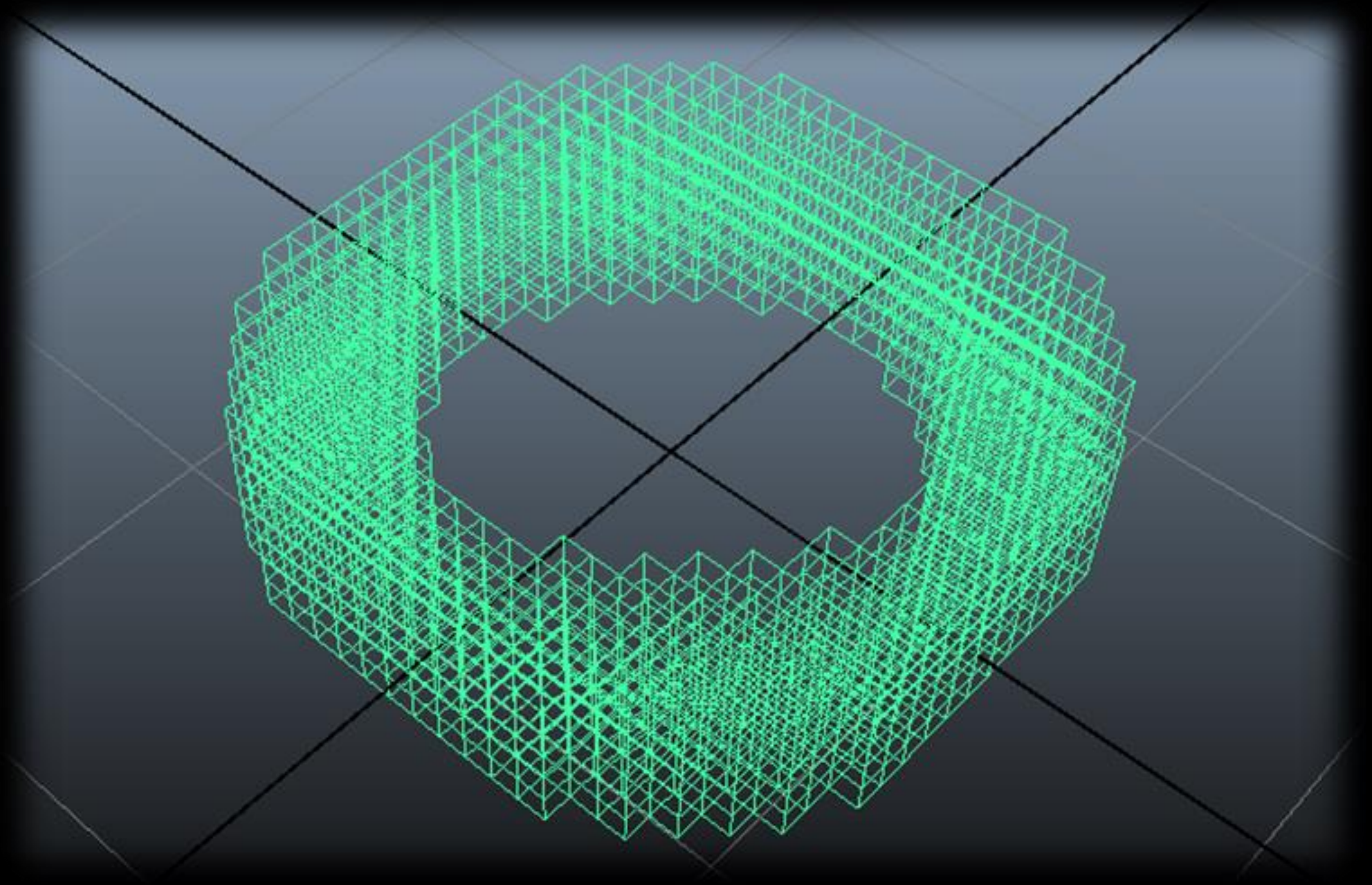
$$u = u_{smooth} + u_{turbulent}$$

- Where

$$u_{smooth} = Interpolate(PM\ Solution)$$
$$u_{turbulent} = NearFieldSummation(\omega_j - VIC)$$

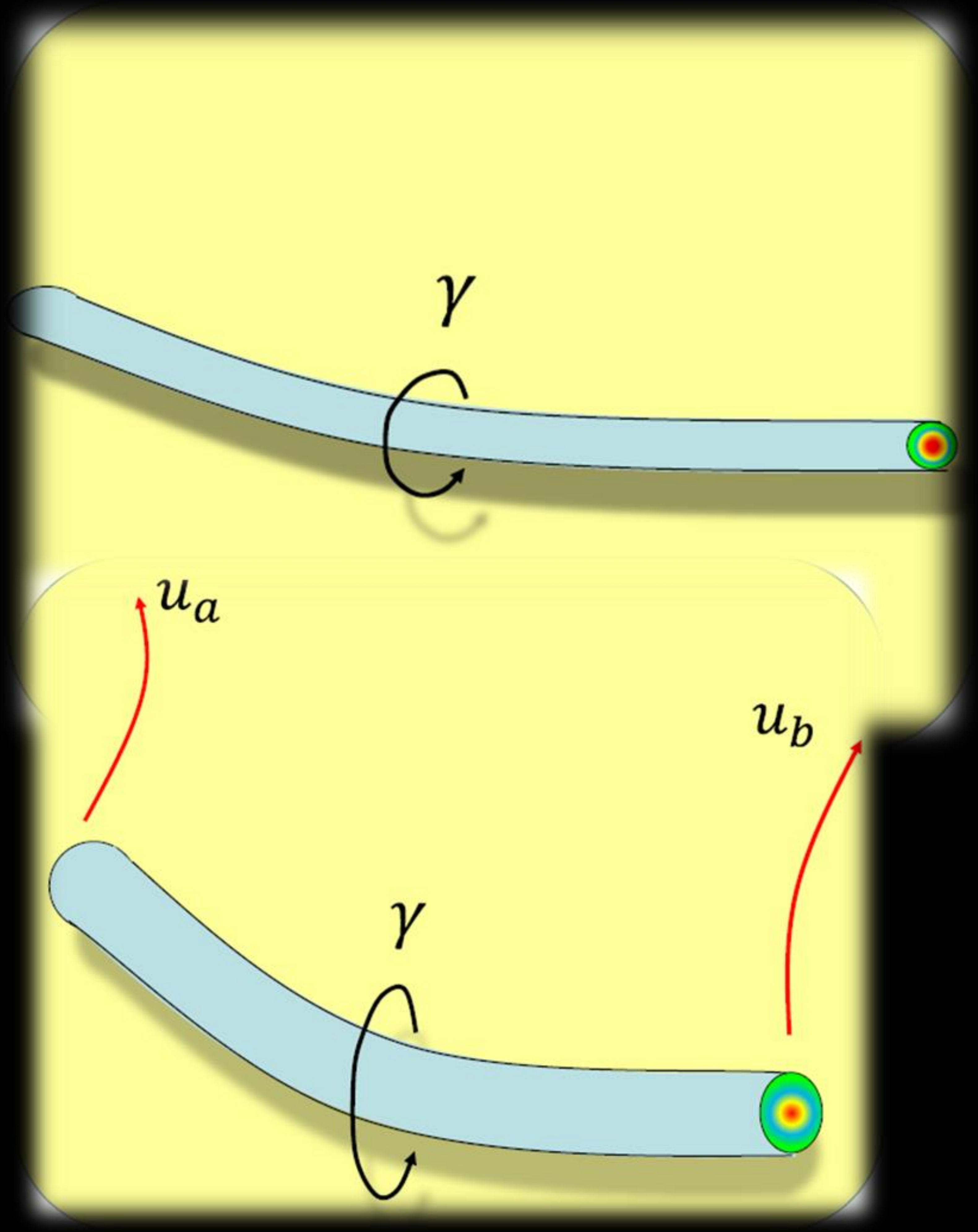
Higher Order PPPM

- Compute Velocity Field not on Particles, but on SpGrid Nodes



- Allows tracking particle trajectories with higher order schemes, for example RK3.

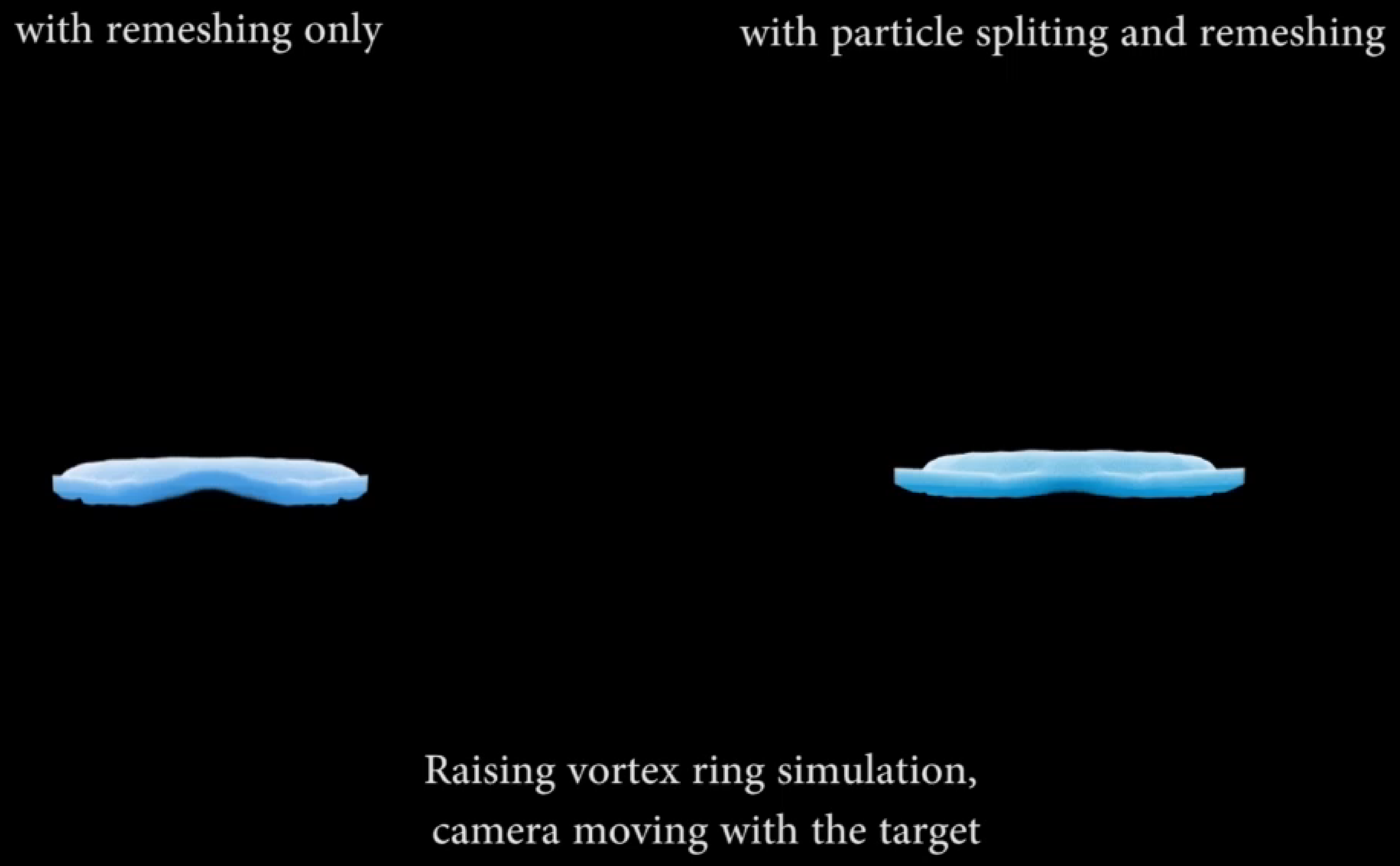
Stable Vortex Segments



$$\left|\omega_i^{n+1}\right|_2 = |\omega_i^n|_2 L_i^{n+1} / L_i^n$$

$$|\omega_i^n|_2 = \frac{\left|\omega_i^0\right|_2 L_i^n}{L_i^0} = \Gamma^0 L_i^n$$

Better than remeshing



Hybrid vortex-Eulerian simulation

Continuum Mechanics and MPM

— Chenfanfu Jiang

Constitutive modeling: basic example

Constitutive modeling: basic example

- **Keywords: constitutive relationship/behavior, strain stress curve, …**

Constitutive modeling: basic example

- **Keywords: constitutive relationship/behavior, strain stress curve, …**
- **General definition: local mechanical <u>response</u> under local <u>kinematics</u>**

Constitutive modeling: basic example

- **Keywords: constitutive relationship/behavior, strain stress curve, …**
- **General definition: local mechanical <u>response</u> under local <u>kinematics</u>**
- **Example: Linear elasticity equilibrium (elastostatics)**

Kinematics describes strain and deformation

infinitesimal strain (Cauchy strain): $\epsilon = \frac{1}{2}(\mathbf{F} + \mathbf{F}^T) - \mathbf{I}$

Response comes from the constitutive model, relates to kinematics

linear elasticity: $\Psi(\mathbf{F}) = \mu\epsilon : \epsilon + \frac{\lambda}{2}\mathrm{tr}(\epsilon)^2$ $\quad \mathbf{P} = 2\mu\epsilon + \lambda\mathrm{tr}(\epsilon)\mathbf{I}$

Governing equations are the the PDEs to be solved

force balance (equilibrium): $\nabla^{\mathbf{X}} \cdot \mathbf{P} + \mathbf{f}^{\mathrm{ext}} = \mathbf{0}$

Constitutive modeling: basic example

- **Keywords: constitutive relationship/behavior, strain stress curve, …**
- **General definition: local mechanical <u>response</u> under local <u>kinematics</u>**
- **Example: Linear elasticity equilibrium (elastostatics)**

Kinematics describes strain and deformation

infinitesimal strain (Cauchy strain): $\epsilon = \frac{1}{2}(\mathbf{F} + \mathbf{F}^T) - \mathbf{I}$

Response comes from the constitutive model, relates to kinematics

linear elasticity: $\Psi(\mathbf{F}) = \mu\epsilon : \epsilon + \frac{\lambda}{2}\mathrm{tr}(\epsilon)^2$ $\quad \mathbf{P} = 2\mu\epsilon + \lambda\mathrm{tr}(\epsilon)\mathbf{I}$

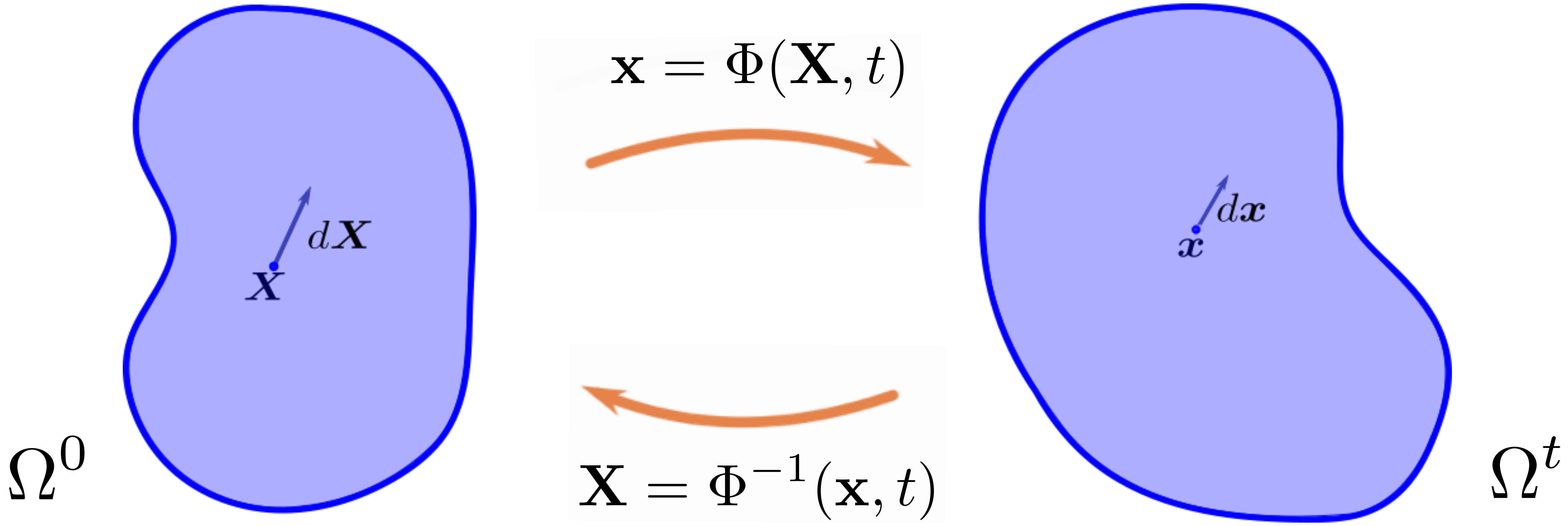
Governing equations are the the PDEs to be solved

force balance (equilibrium): $\nabla^{\mathbf{X}} \cdot \mathbf{P} + \mathbf{f}^{\mathrm{ext}} = \mathbf{0}$

A core model in topology optimization

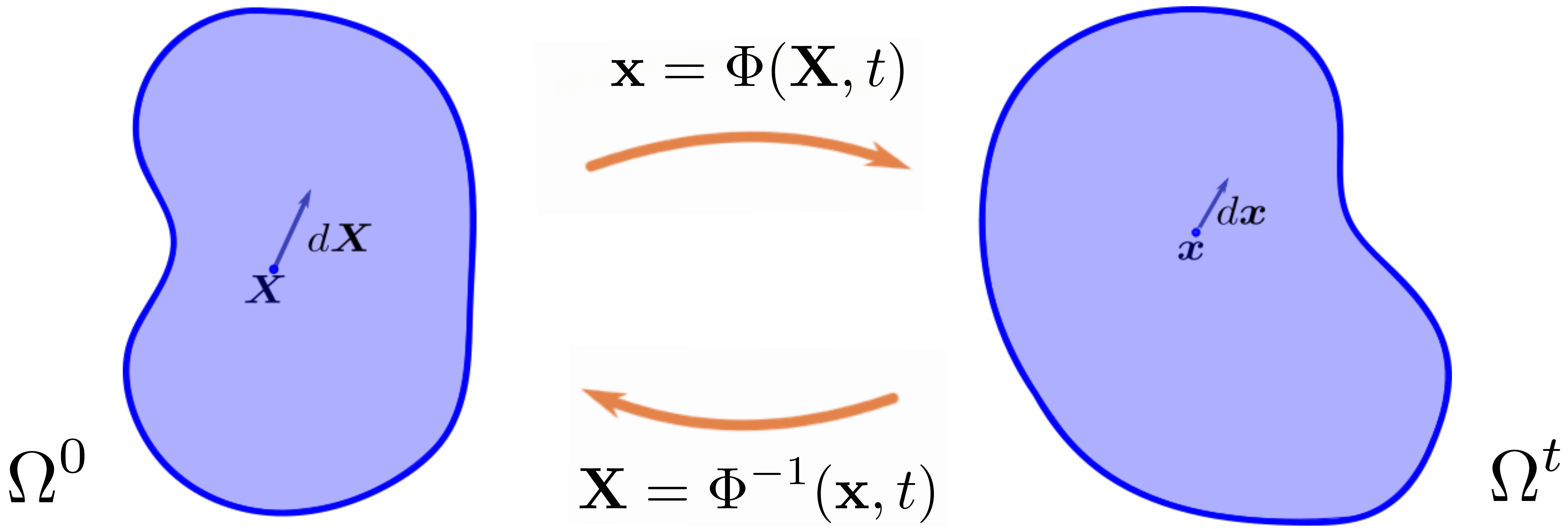
Continuum assumption of material kinematics

Continuum assumption



Continuum assumption of material kinematics

Continuum assumption



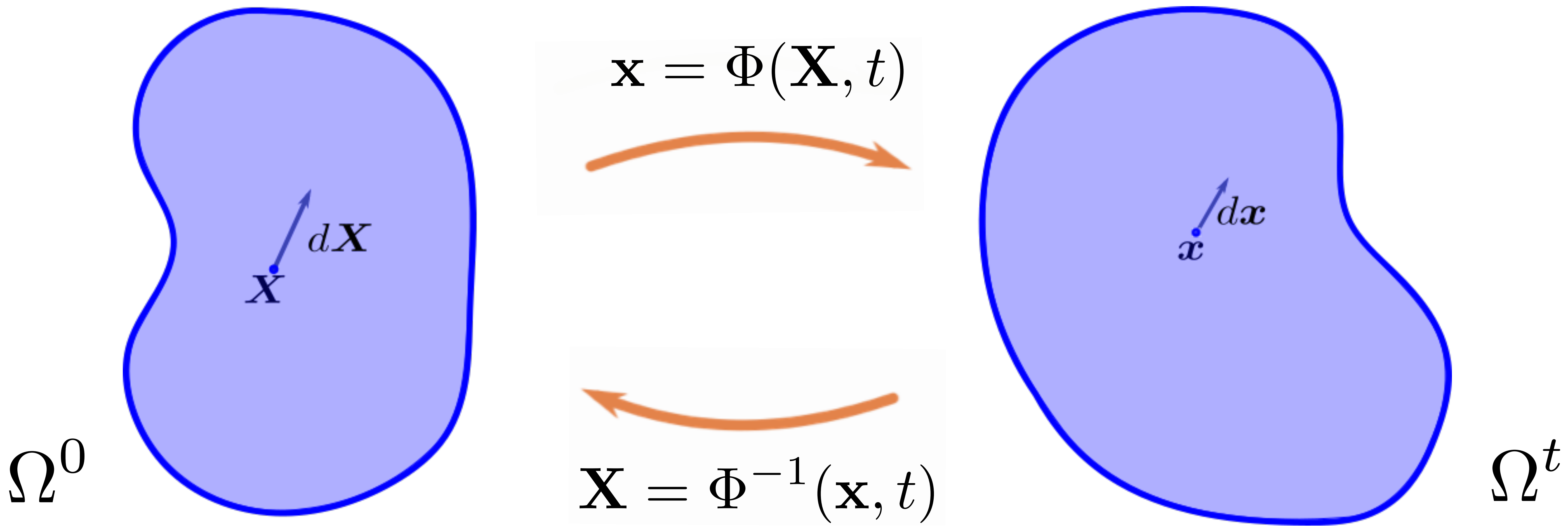
Lagrangian kinematics

$$\mathbf{V}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial t}$$

$$\mathbf{A}(\mathbf{X}, t) = \frac{\partial^2 \Phi}{\partial t^2}$$

Continuum assumption of material kinematics

Continuum assumption



Lagrangian kinematics

$$\mathbf{V}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial t}$$

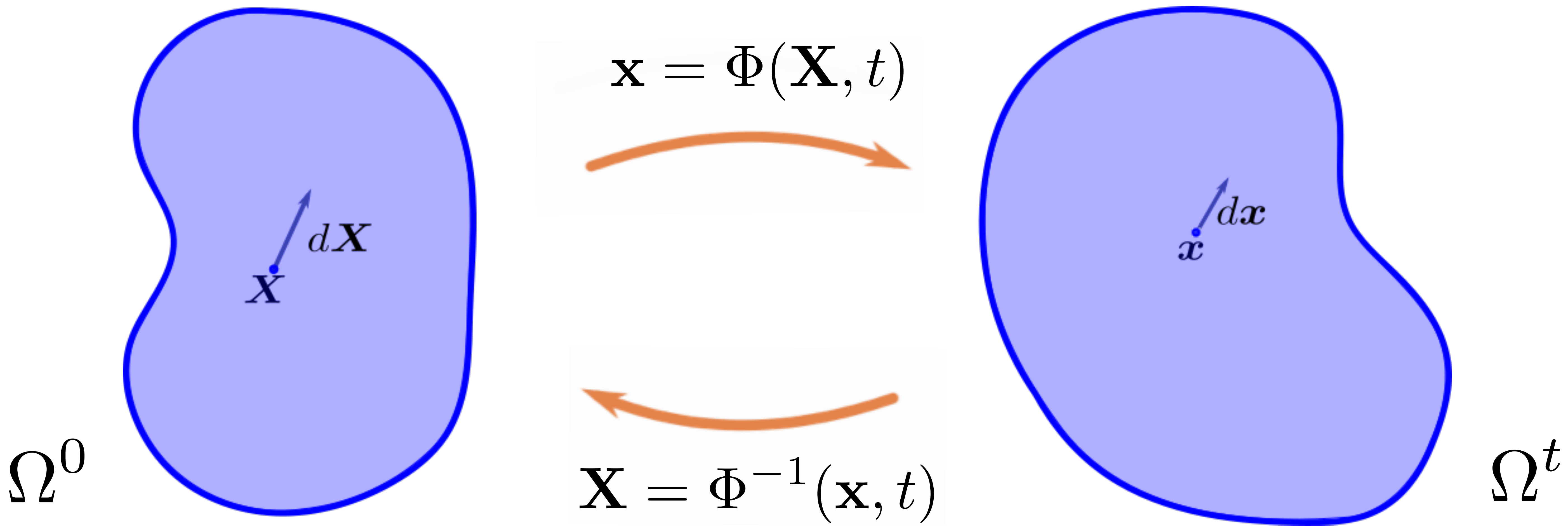
$$\mathbf{A}(\mathbf{X}, t) = \frac{\partial^2 \Phi}{\partial t^2}$$

Eulerian velocity

$$\mathbf{v}(\mathbf{x}, t) = \mathbf{V}(\Phi^{-1}(\mathbf{x}, t), t)$$

Continuum assumption of material kinematics

Continuum assumption



Lagrangian kinematics

$$\mathbf{V}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial t}$$

$$\mathbf{A}(\mathbf{X}, t) = \frac{\partial^2 \Phi}{\partial t^2}$$

Deformation gradient

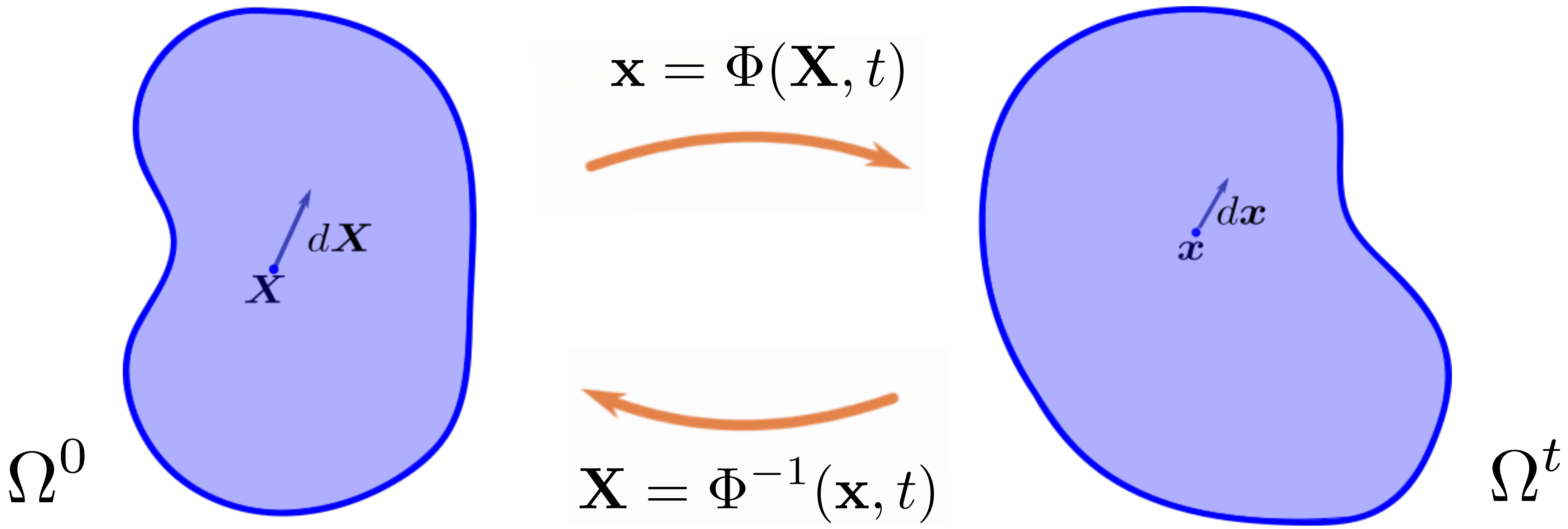
$$\mathbf{F}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial \mathbf{X}}$$

Eulerian velocity

$$\mathbf{v}(\mathbf{x}, t) = \mathbf{V}(\Phi^{-1}(\mathbf{x}, t), t)$$

Continuum assumption of material kinematics

Continuum assumption



Lagrangian kinematics

$$\mathbf{V}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial t}$$

$$\mathbf{A}(\mathbf{X}, t) = \frac{\partial^2 \Phi}{\partial t^2}$$

Deformation gradient

$$\mathbf{F}(\mathbf{X}, t) = \frac{\partial \Phi}{\partial \mathbf{X}}$$

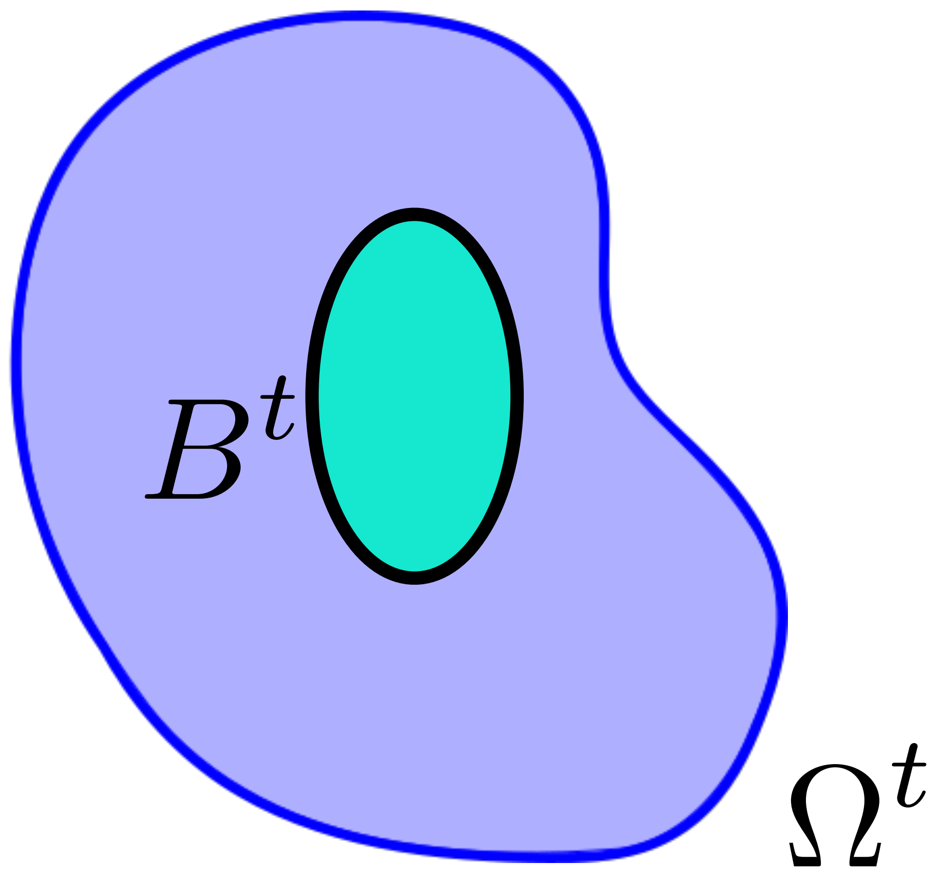
Eulerian velocity

$$\mathbf{v}(\mathbf{x}, t) = \mathbf{V}(\Phi^{-1}(\mathbf{x}, t), t)$$

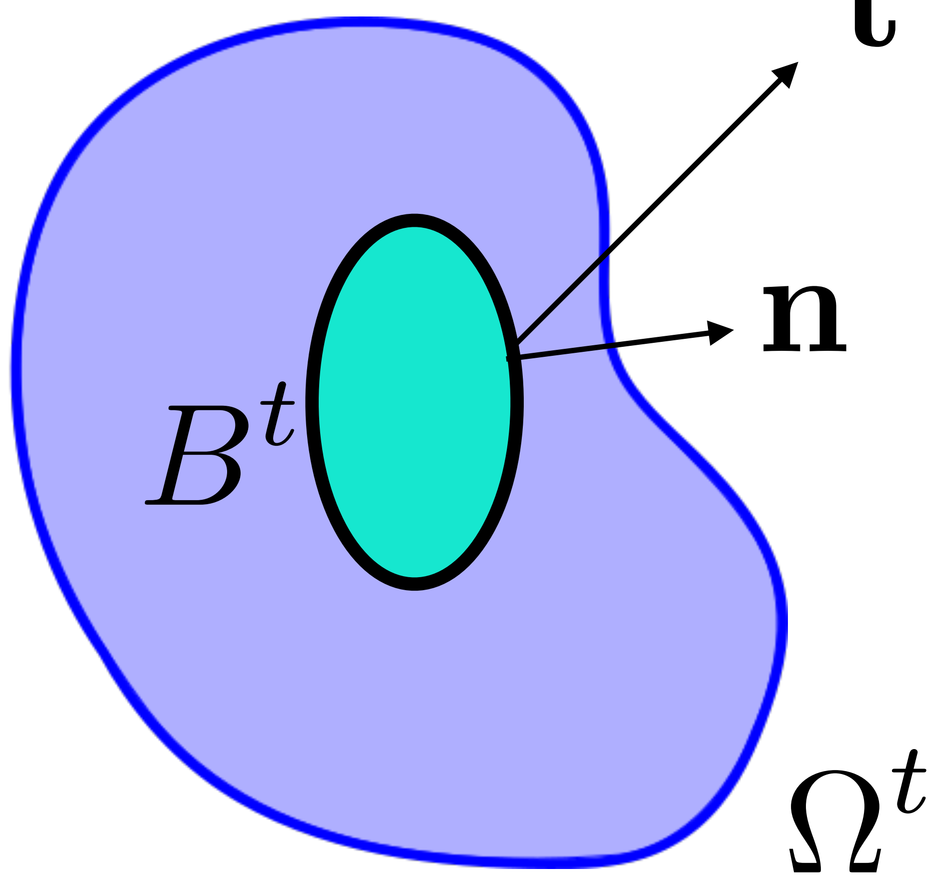
Volume change

$$J(\mathbf{X}, t) = \det(\mathbf{F}(\mathbf{X}, t))$$

Stress and dynamics



Stress and dynamics

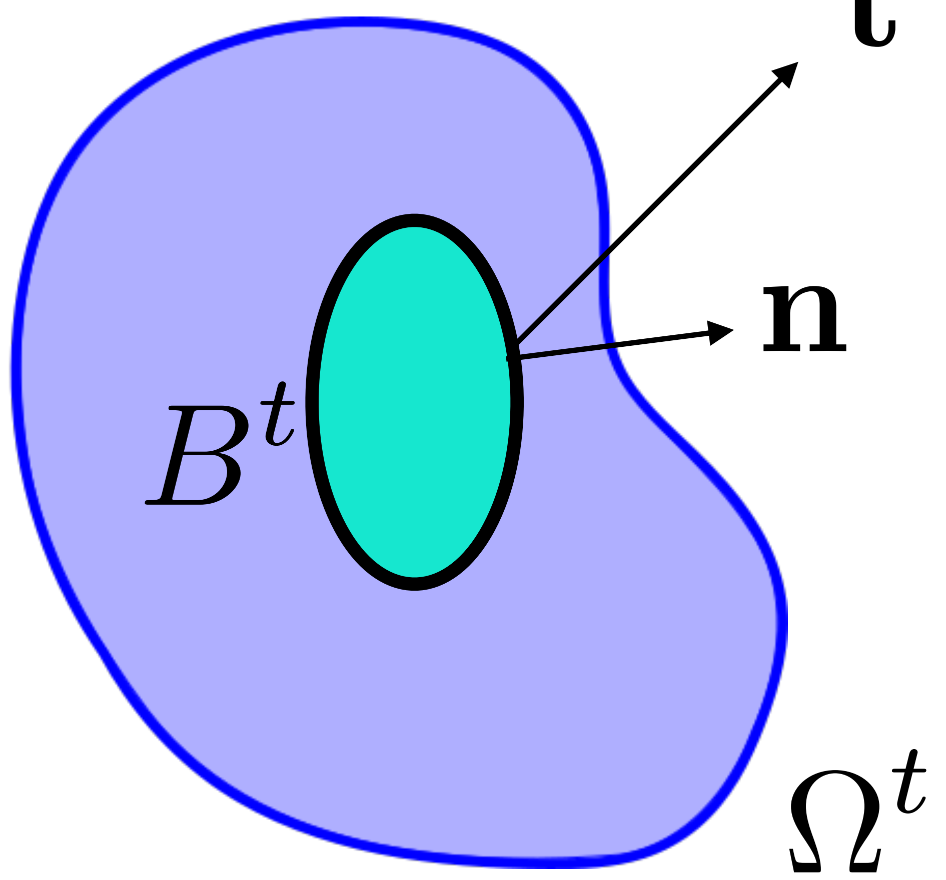


$\mathbf{t} = \sigma \mathbf{n}$ traction: force per unit area

$\mathbf{n}$ outward pointing normal

σ Cauchy stress tensor

Stress and dynamics



$\mathbf{t} = \sigma \mathbf{n}$ traction: force per unit area

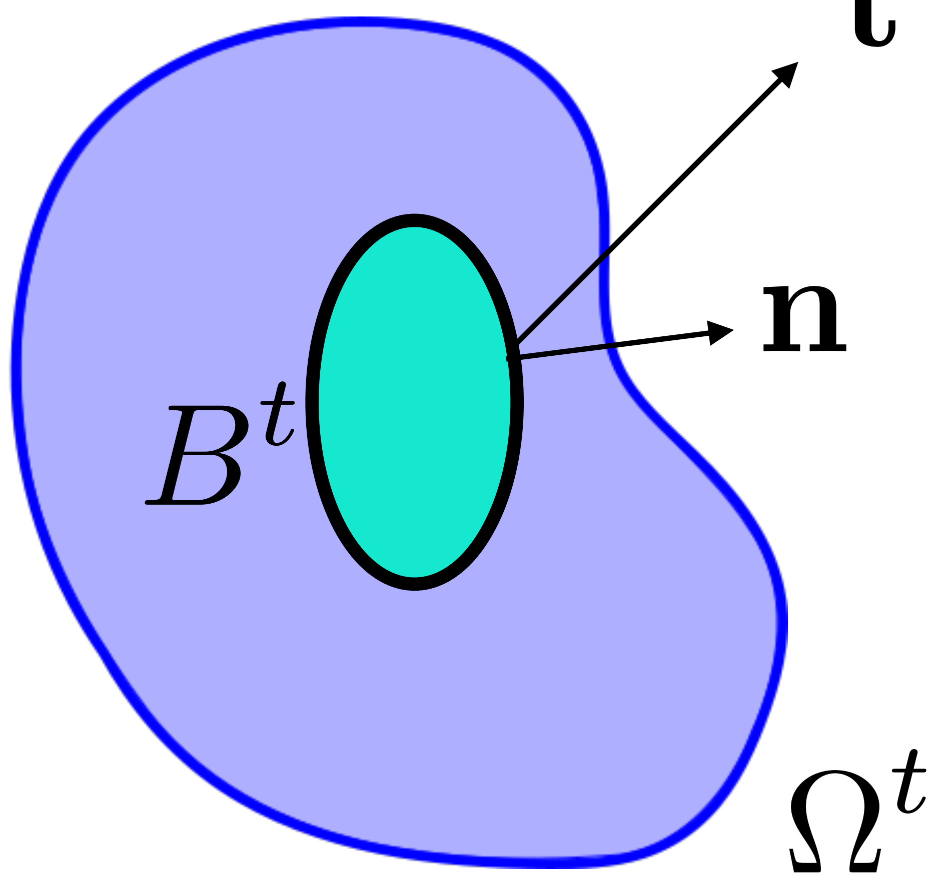
$\mathbf{n}$ outward pointing normal

σ Cauchy stress tensor

Net force due to contact with exterior

$$\mathbf{f}_{B^t} = \int_{\partial B^t} \mathbf{t} \, ds(\mathbf{x}) = \int_{\partial B^t} \sigma \mathbf{n} \, ds(\mathbf{x}) = \int_{B^t} \nabla^{\mathbf{x}} \cdot \sigma \, d\mathbf{x}$$

Stress and dynamics



$\mathbf{t} = \sigma\mathbf{n}$ traction: force per unit area

$\mathbf{n}$ outward pointing normal

σ Cauchy stress tensor

Net force due to contact with exterior

$$\mathbf{f}_{B^t} = \int_{\partial B^t} \mathbf{t} ds(\mathbf{x}) = \int_{\partial B^t} \sigma \mathbf{n} ds(\mathbf{x}) = \int_{B^t} \nabla^{\mathbf{x}} \cdot \sigma d\mathbf{x}$$

Conservation of momentum

$$\rho \frac{D\mathbf{v}}{Dt} = \nabla \cdot \boldsymbol{\sigma} + \rho \mathbf{g}.$$

MPM and MLS-MPM from the weak form

Strong form

$$\rho \frac{D\mathbf{v}}{Dt} = \nabla \cdot \boldsymbol{\sigma} + \rho \mathbf{g} \quad \text{(conservation of momentum)}$$

Weak form

$$\frac{1}{\Delta t} \int_{\Omega^{t^n}} \rho(\mathbf{x}, t^n) \left(\hat{v}_\alpha^{n+1}(\mathbf{x}) - v_\alpha^n(\mathbf{x}) \right) q_\alpha(\mathbf{x}, t^n) d\mathbf{x}$$

$$= \int_{\partial\Omega^{t^n}} q_\alpha(\mathbf{x}, t^n) \mathcal{T}_\alpha(\mathbf{x}, t^n) ds - \int_{\Omega^{t^n}} q_{\alpha,\beta}(\mathbf{x}, t^n) \sigma_{\alpha\beta}(\mathbf{x}, t^n) d\mathbf{x}$$

B-Spline MPM kernels

$$q_\alpha(\mathbf{x}, t^n) = N_i(\mathbf{x}) q_{i\alpha}^n$$

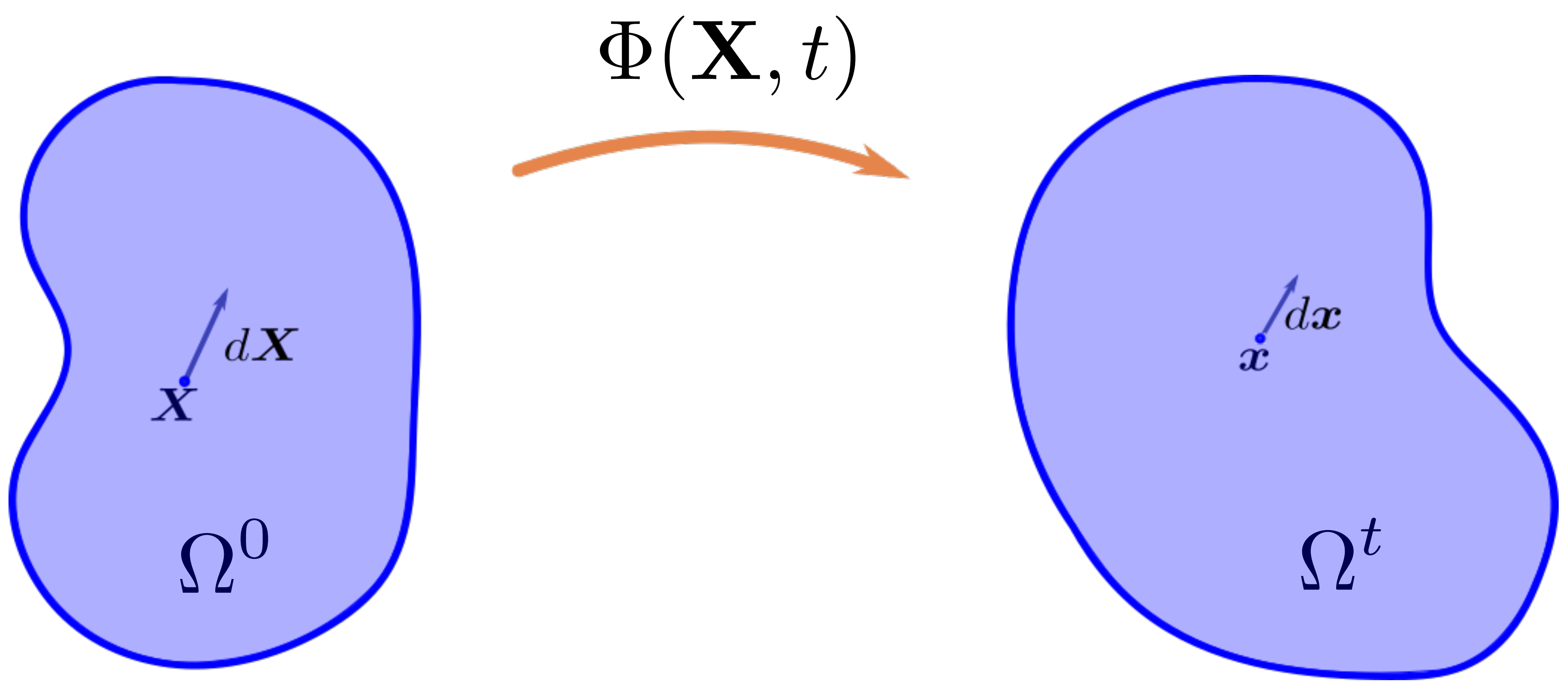
MLS-MPM kernels

$$q_\alpha(\mathbf{x}, t^n) = \mathbf{P}^T(\mathbf{x} - \mathbf{x}_p^n) \mathbf{M}^{-1}(\mathbf{x}_p^n) \xi_{\hat{i}}(\mathbf{x}_p^n) \mathbf{P}(\mathbf{x}_{\hat{i}} - \mathbf{x}_p^n) \delta_{\alpha\hat{\alpha}}$$

Lagrangian Kinematics of Continuum

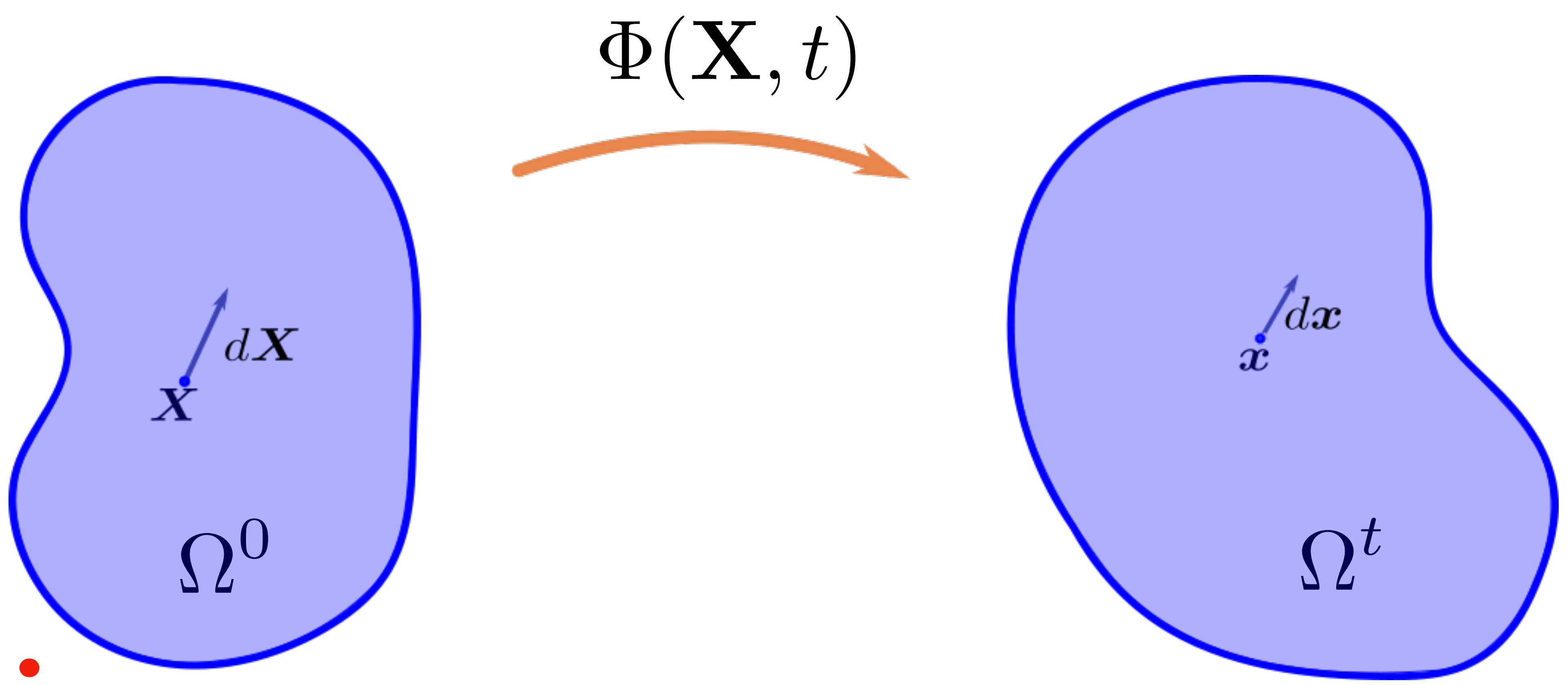
Lagrangian Kinematics of Continuum

Full Lagrangian kinematics

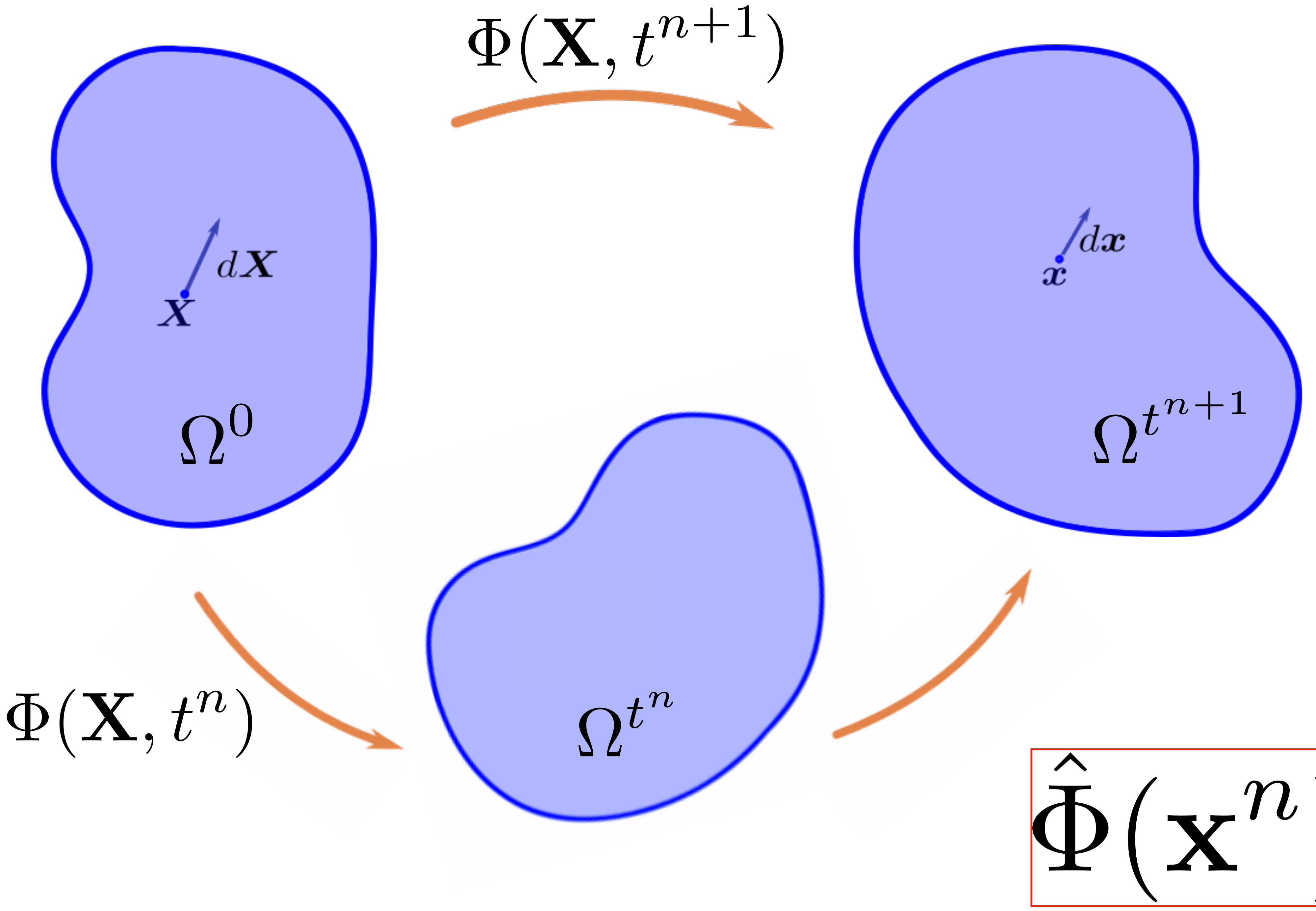


Lagrangian Kinematics of Continuum

Full Lagrangian kinematics



Updated Lagrangian kinematics



The deformation flows through updating the current configuration.

$$\hat{\Phi}(\mathbf{x}^n) = \Phi(\Phi^{-1}(\mathbf{x}^n, t^n), t^{n+1})$$

Evolve the deformation

$$\boldsymbol{F}_p^{n+1} = (\boldsymbol{I} + \Delta t \nabla \boldsymbol{v}_p^{n+1})\boldsymbol{F}_p^n$$

Evolve the deformation

$$\boldsymbol{F}_p^{n+1} = (\boldsymbol{I} + \Delta t \nabla \boldsymbol{v}_p^{n+1})\boldsymbol{F}_p^n$$

F is updated using velocity gradient

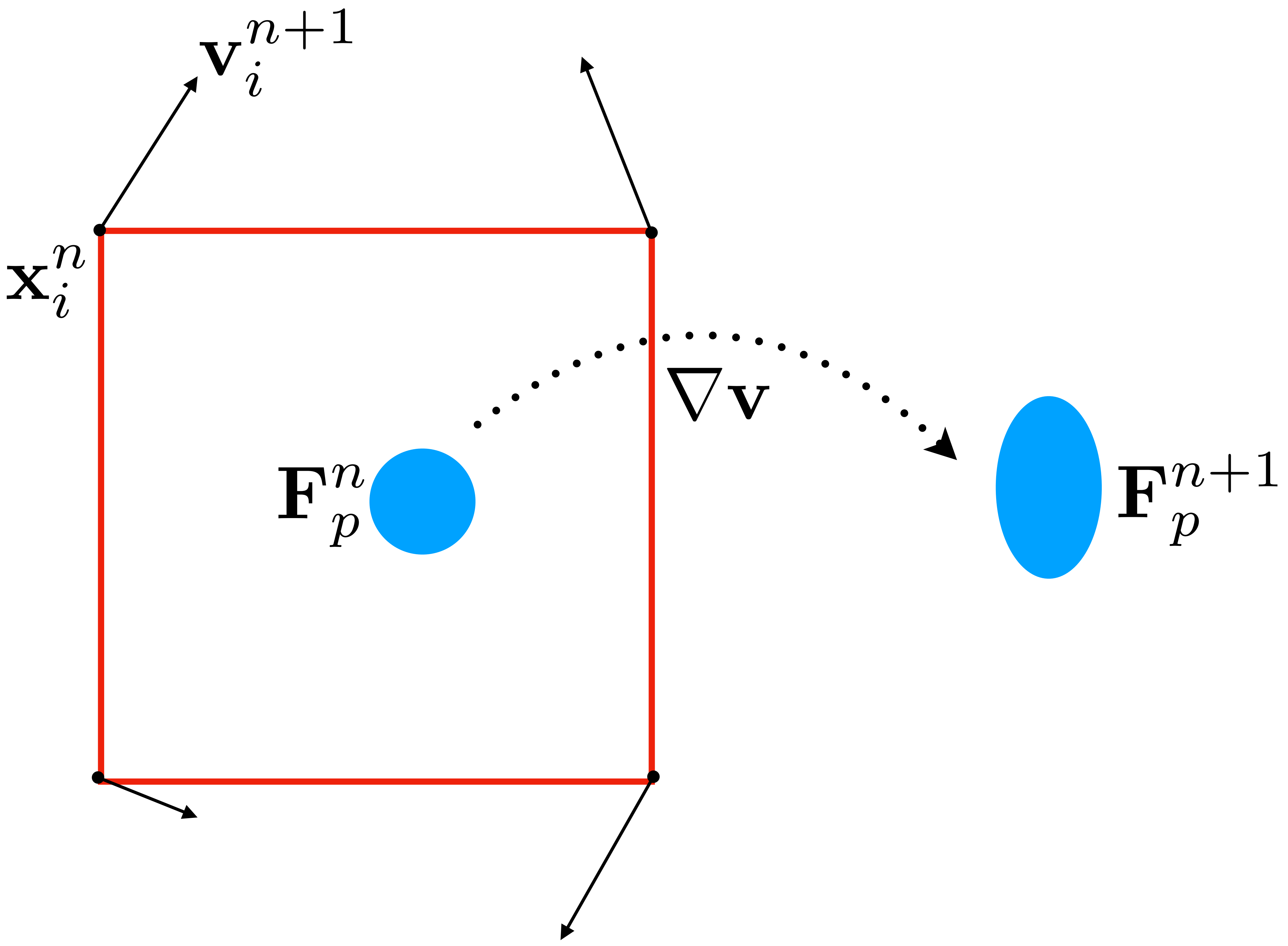
$$\hat{\Phi}(\mathbf{x}_p^n) = \Phi(\Phi^{-1}(\mathbf{x}_p^n, t^n), t^{n+1})$$

$$\frac{\partial \hat{\Phi}}{\partial \mathbf{x}_p^n} = \frac{\partial \Phi}{\partial \mathbf{X}}(\mathbf{X}, t^{n+1}) \left(\frac{\partial \Phi}{\partial \mathbf{X}}(\mathbf{X}, t^n) \right)^{-1}$$

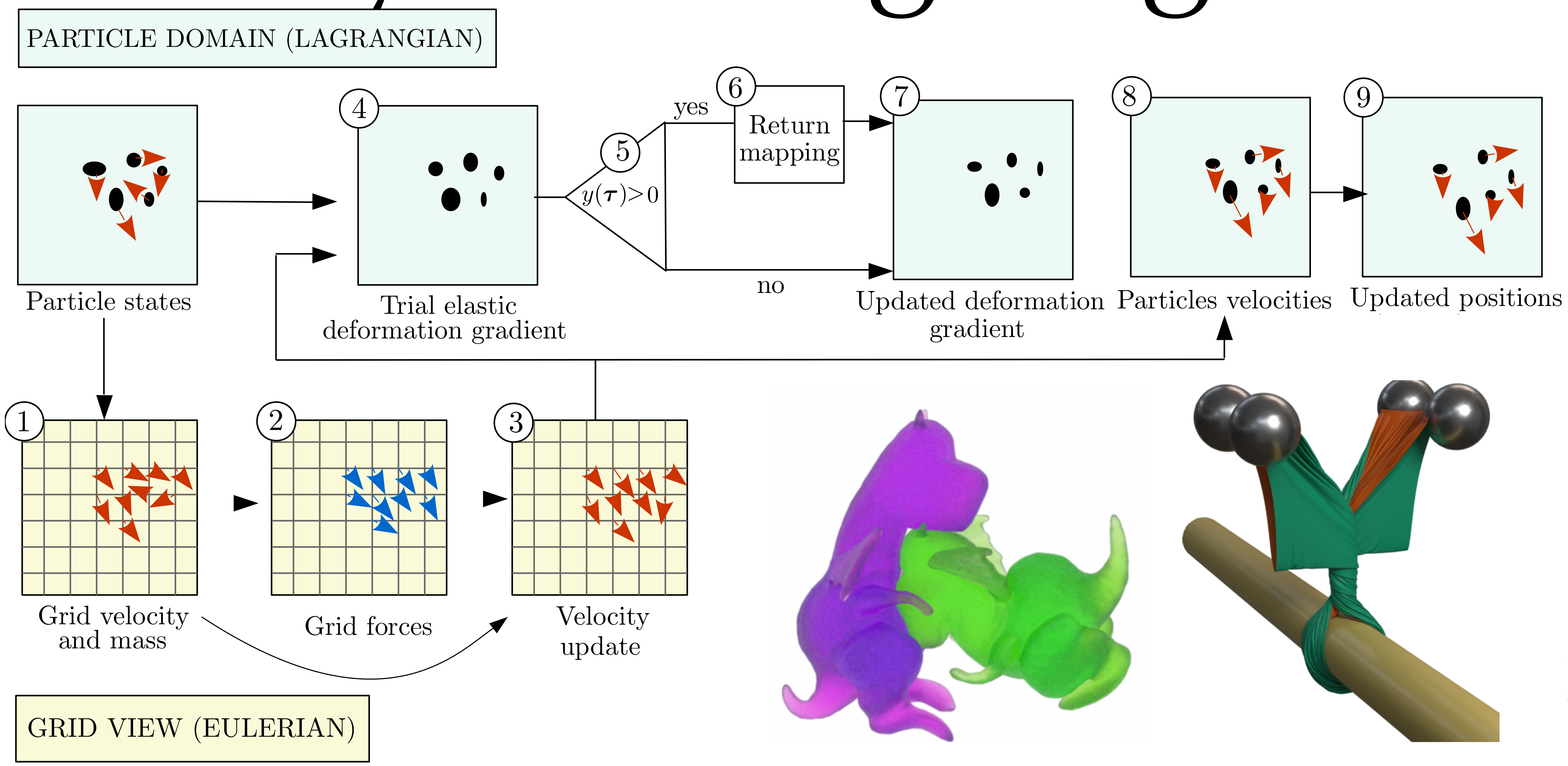
$$\hat{\mathbf{f}} = \mathbf{F}^{n+1}(\mathbf{F}^n)^{-1}$$

$$\hat{\Phi} = \mathbf{x}_p^n + \Delta t \sum_i \mathbf{v}_i^{n+1} w_{ip}^n$$

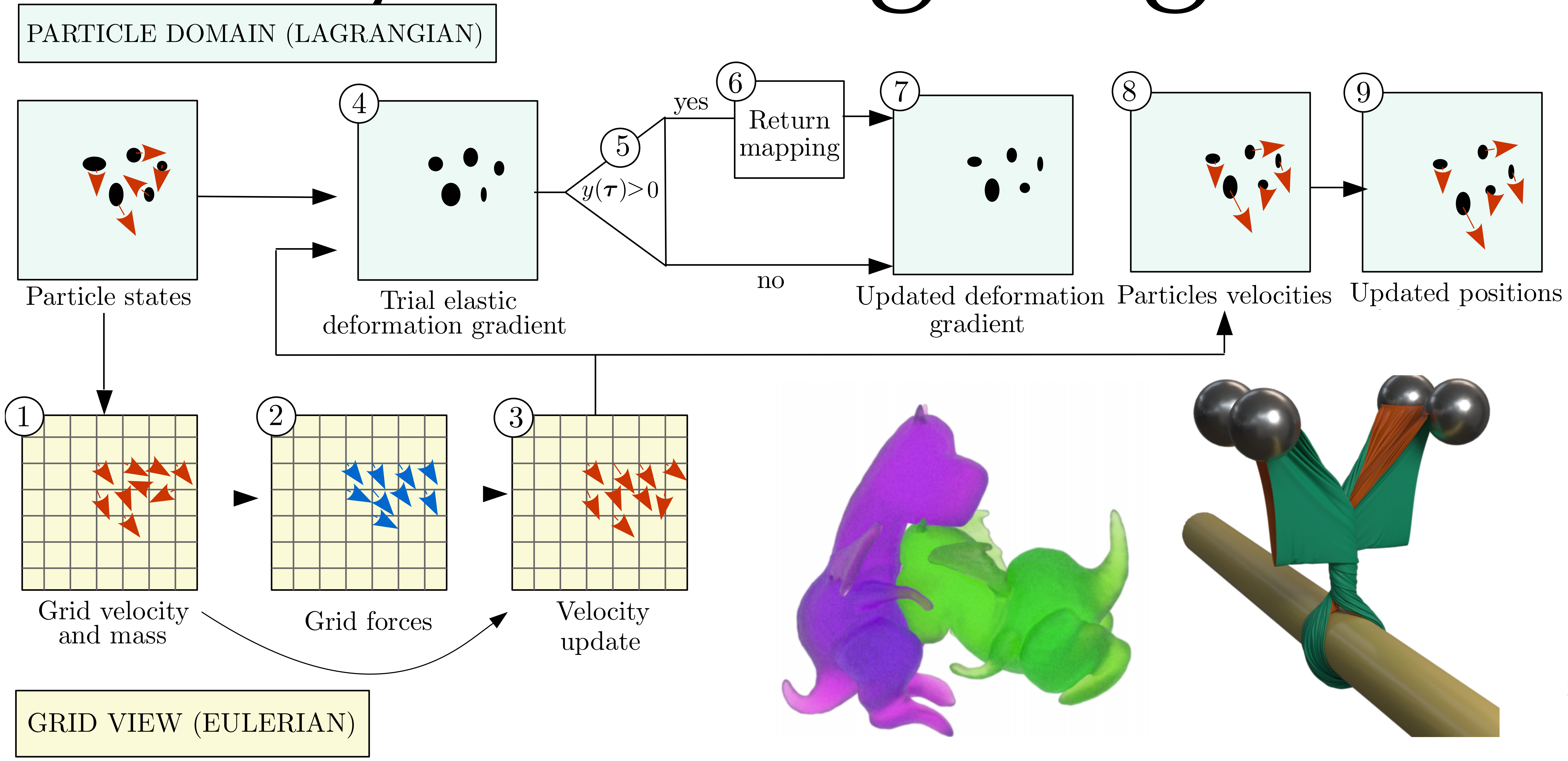
$$\hat{\mathbf{f}} = \mathbf{I} + \Delta t \sum_i \mathbf{v}_i^{n+1} \nabla w_{ip}^n$$



MPM is hybrid Lagrangian/Eulerian

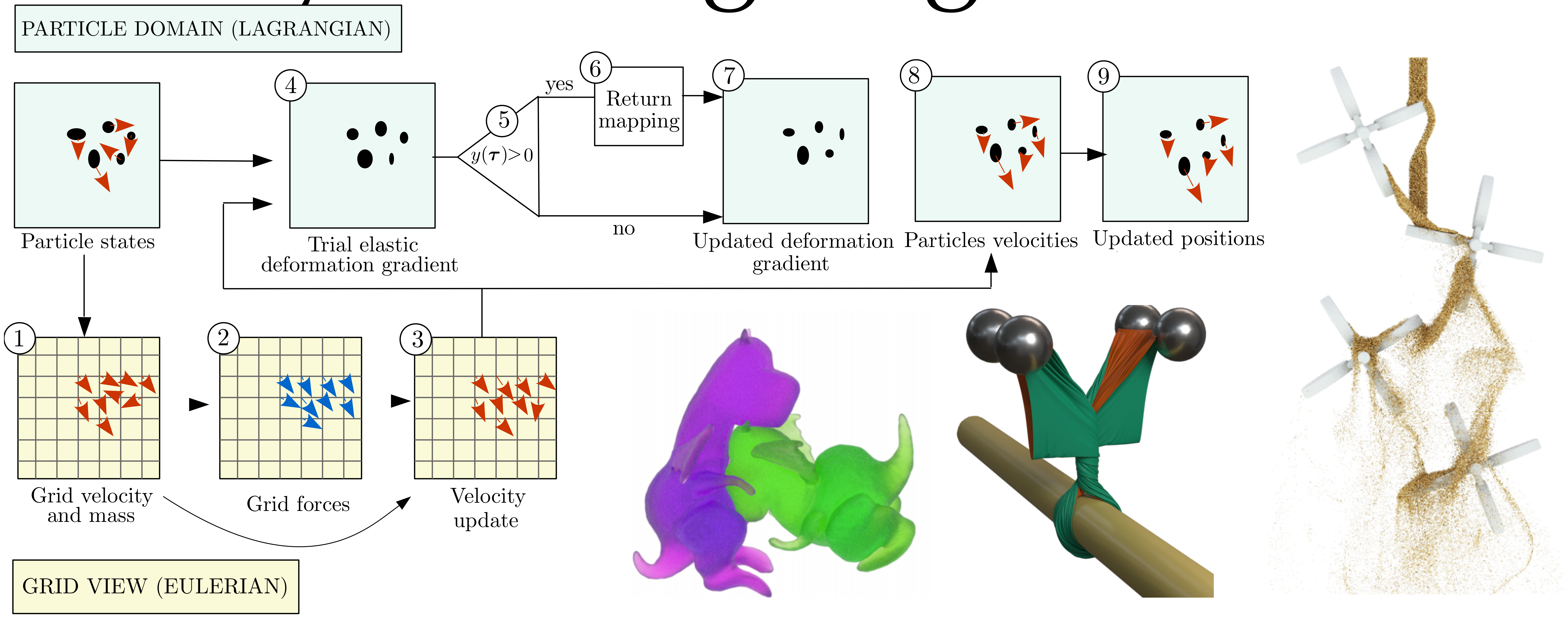


MPM is hybrid Lagrangian/Eulerian



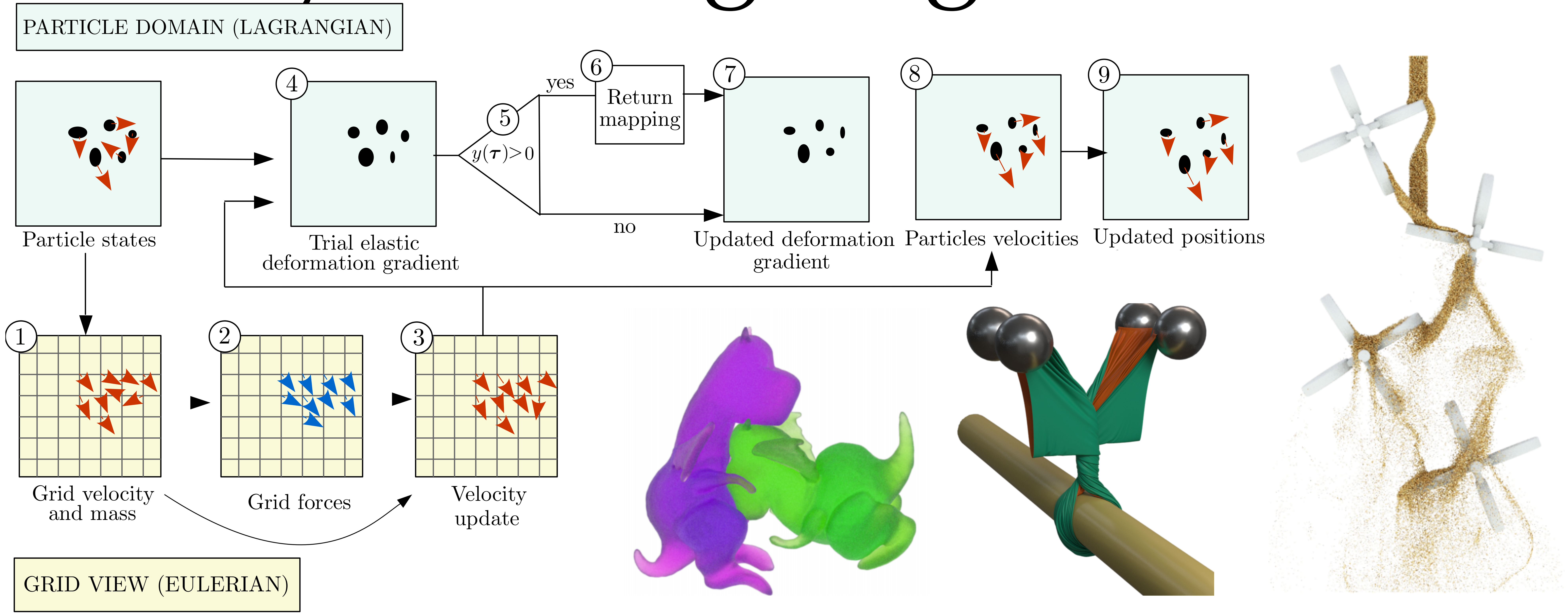
✦ **Grid handles: the Galerkin DOFs, discretization, function space, ….**

MPM is hybrid Lagrangian/Eulerian



- **Grid handles: the Galerkin DOFs, discretization, function space, ….**
- **The transfer/embedding handles: coupling, quadrature rule, non-penetration, topology change, …**

MPM is hybrid Lagrangian/Eulerian



- ✦ **Grid handles: the Galerkin DOFs, discretization, function space, ….**
- ✦ **The transfer/embedding handles: coupling, quadrature rule, non-penetration, topology change, …**
- ✦ **Each individual particle handles: constitutive modeling - the physics**

Elasticity, hyperelasticity

- **Hyperelasticity is convenient**
  - Capturing real materials
  - Artistic design
  - Anisotropy, damping …

$$\Psi(\mathbf{F}) = \frac{\mu}{2}\left(\mathrm{tr}(\mathbf{F}\mathbf{F}^T) - 3\right) - \mu\log(J) + \frac{\lambda}{2}\log^2(J)$$

$$\Psi(\mathbf{F}) = \mu\|\mathbf{F} - \mathbf{R}\|_F^2 + \frac{\lambda}{2}(\det(\mathbf{F}) - 1)^2$$

$$\Psi(\mathbf{F}) = \mu\mathrm{tr}((\log\mathbf{\Sigma})^2) + \frac{\lambda}{2}(\mathrm{tr}(\log\mathbf{\Sigma}))^2$$

$$\Psi(\mathbf{F}) = f(\sigma_1) + f(\sigma_2) + f(\sigma_3) + g(\sigma_1\sigma_2) + g(\sigma_2\sigma_3) + g(\sigma_3\sigma_1) + h(\sigma_1\sigma_2\sigma_3)$$

Elasticity, hyperelasticity

- **Hyperelasticity is convenient**
  - Capturing real materials
  - Artistic design
  - Anisotropy, damping …

$$\Psi(\mathbf{F}) = \frac{\mu}{2}\left(\mathrm{tr}(\mathbf{F}\mathbf{F}^T) - 3\right) - \mu\log(J) + \frac{\lambda}{2}\log^2(J)$$

$$\Psi(\mathbf{F}) = \mu\|\mathbf{F} - \mathbf{R}\|_F^2 + \frac{\lambda}{2}(\det(\mathbf{F}) - 1)^2$$

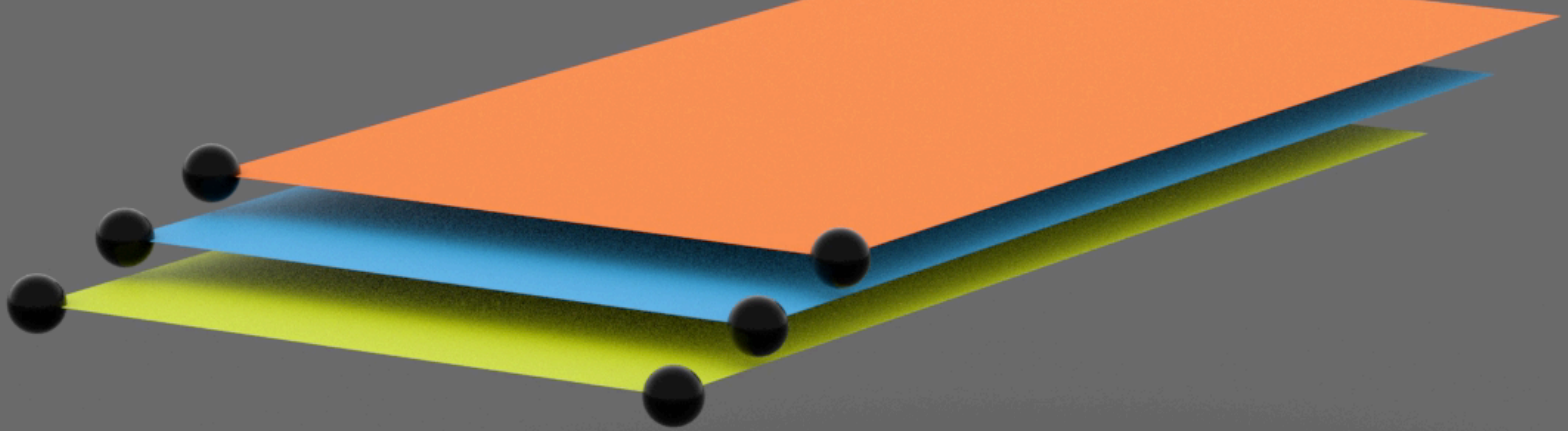
$$\Psi(\mathbf{F}) = \mu\mathrm{tr}((\log\mathbf{\Sigma})^2) + \frac{\lambda}{2}(\mathrm{tr}(\log\mathbf{\Sigma}))^2$$

$$\Psi(\mathbf{F}) = f(\sigma_1) + f(\sigma_2) + f(\sigma_3) + g(\sigma_1\sigma_2) + g(\sigma_2\sigma_3) + g(\sigma_3\sigma_1) + h(\sigma_1\sigma_2\sigma_3)$$

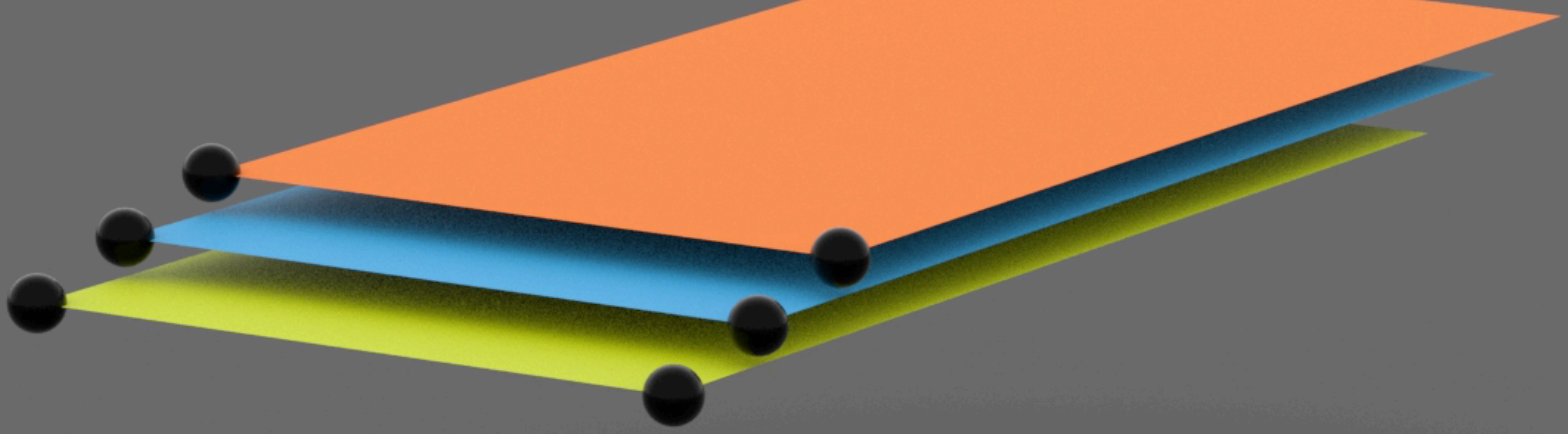
Inelasticity


fluid
sediment


fluid
sediment



[Jiang17] **Anisotropic Elastoplasticity for Cloth, Knit and Hair Frictional Contact**, Chenfanfu Jiang, Theodore Gast, Joseph Teran, SIGGRAPH 2017



[Jiang17] **Anisotropic Elastoplasticity for Cloth, Knit and Hair Frictional Contact**, Chenfanfu Jiang, Theodore Gast, Joseph Teran, SIGGRAPH 2017



MPM is in Bifrost[Maya] starting yesterday

TRY IT NOW!

Autodesk Maya @AdskMaya · 23h

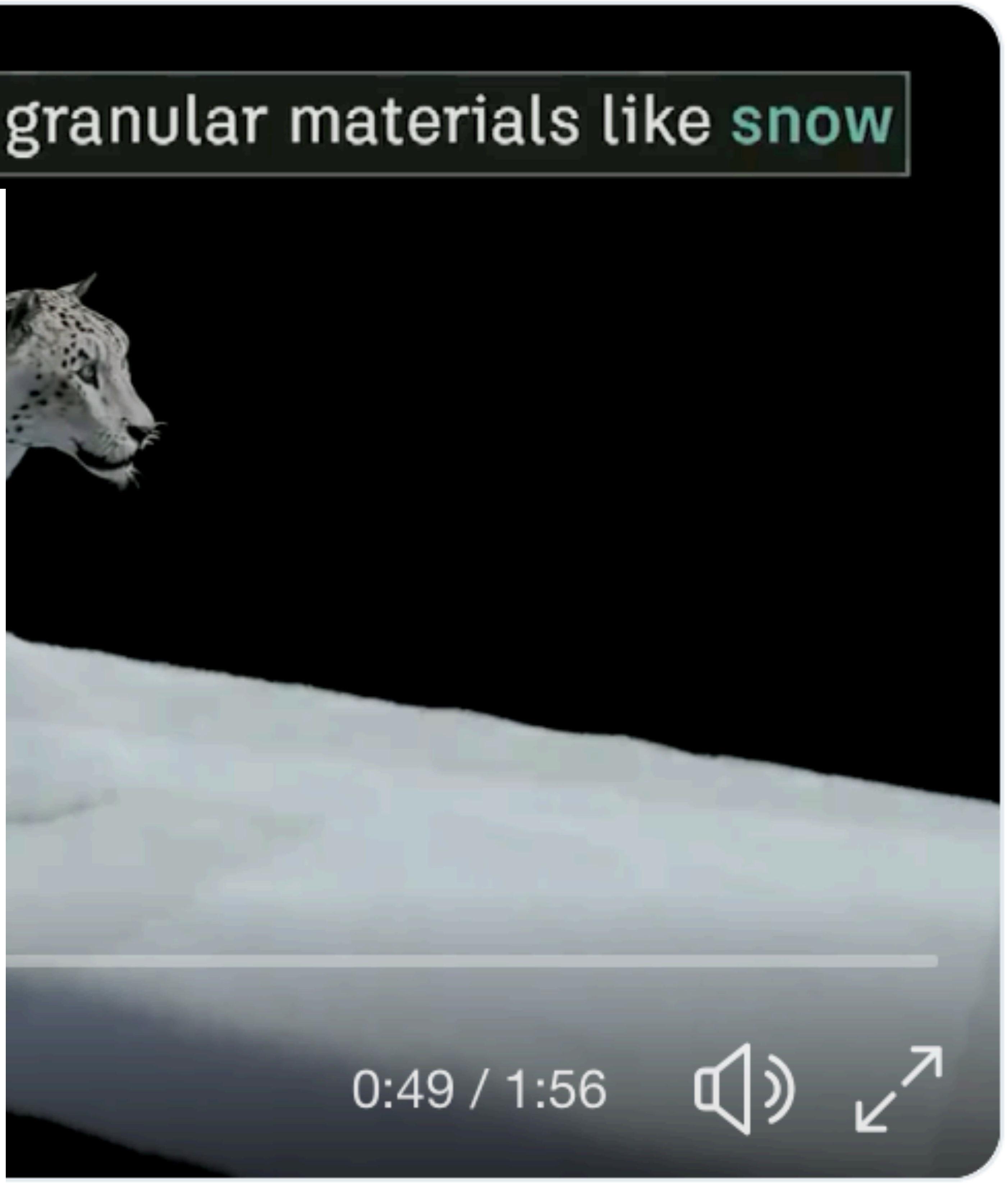
Coming July 31 –#Bifrost for Maya empowers #3D artists and TDs to create blockbuster-worthy effects using a new visual programming environment. Learn about the powerful dynamic solvers, ready-to-use graphs, and more here: bit.ly/bifrost-maya

MPM is in Bifrost[Maya] starting yesterday

TRY IT NOW!

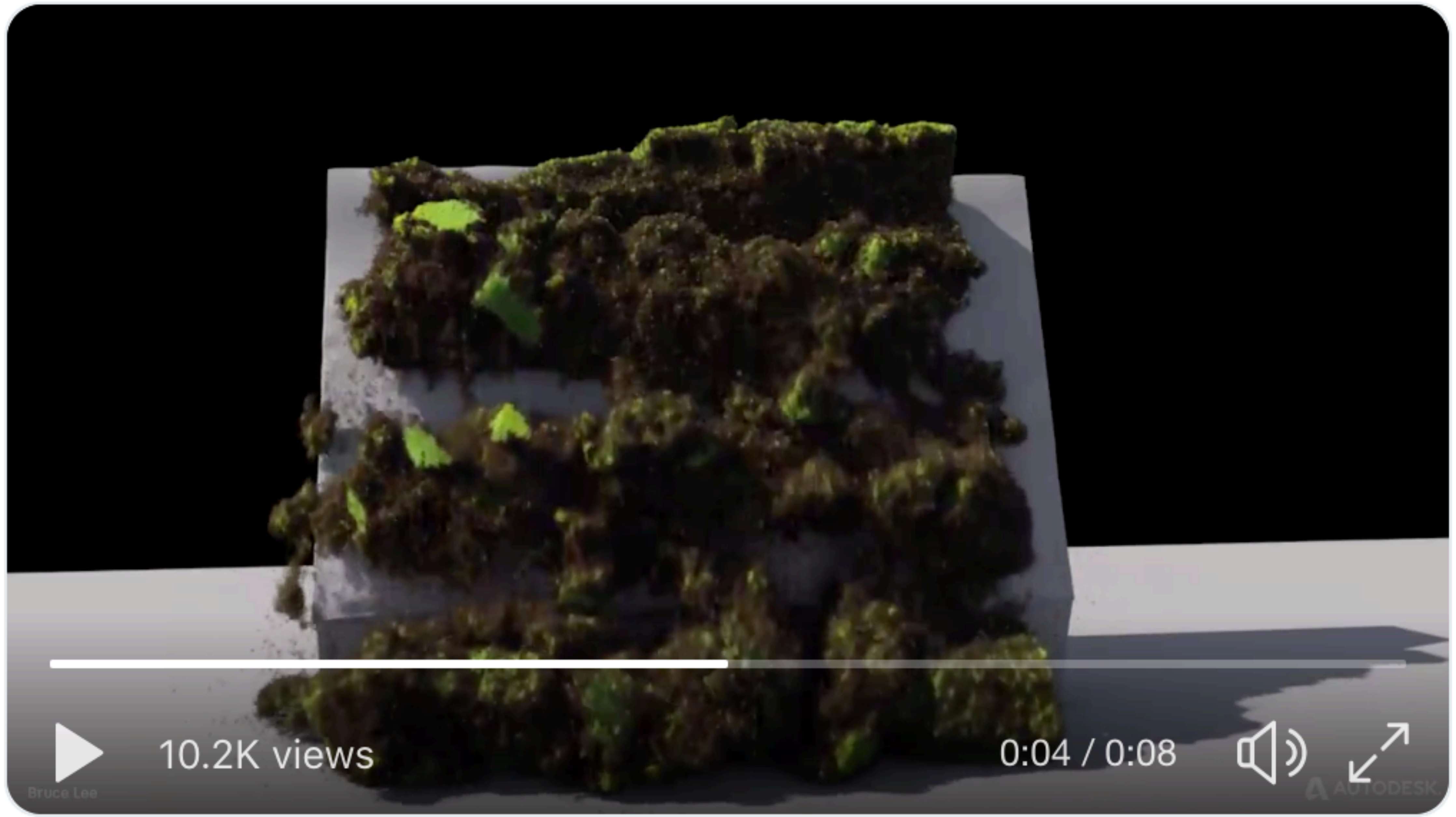
Autodesk Maya @AdskMaya · 23h

Coming July 31 –#Bifrost for Maya empowers #3D artists and TDs to create blockbuster-worthy effects using a new visual programming environment. Learn about the powerful dynamic solvers, ready-to-use graphs, and more here: bit.ly/bifrost-maya



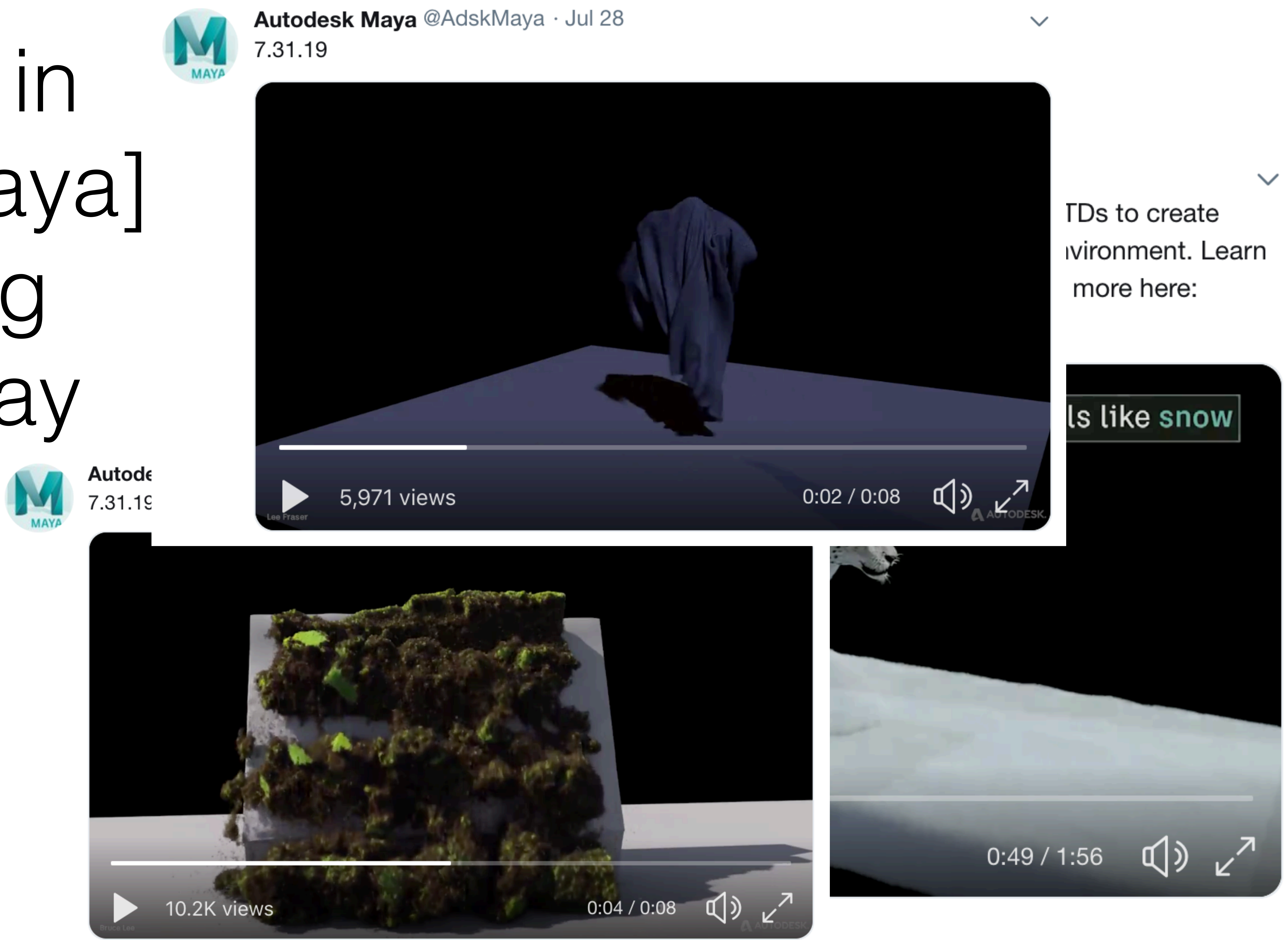
Autodesk Maya @AdskMaya · Jul 23

7.31.19



MPM is in Bifrost[Maya] starting yesterday

TRY IT NOW!

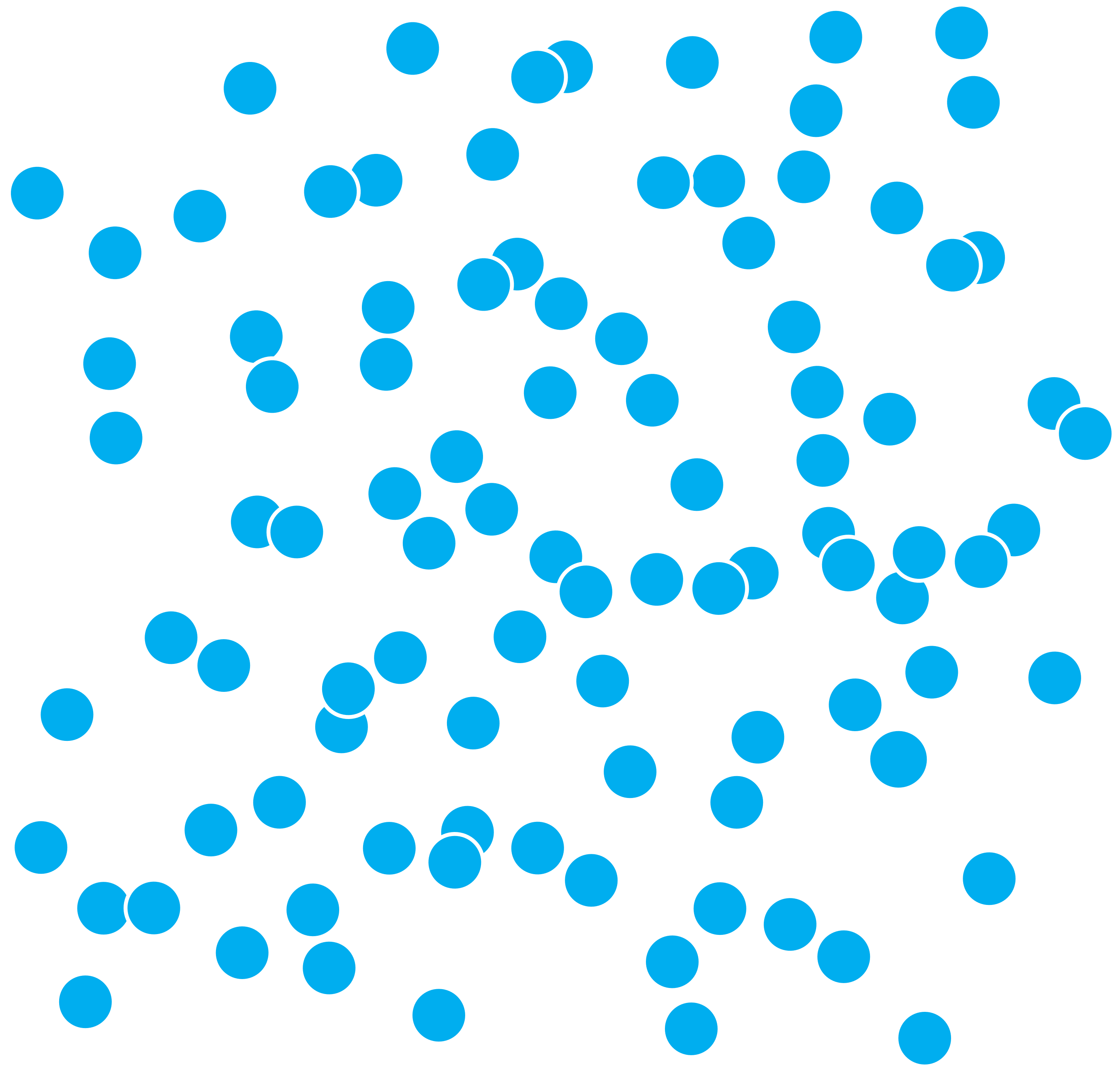


MPM Implementation & Performance Considerations

Yuanming Hu
MIT CSAIL

The Material Point Method (MPM)

The Material Point Method (MPM)



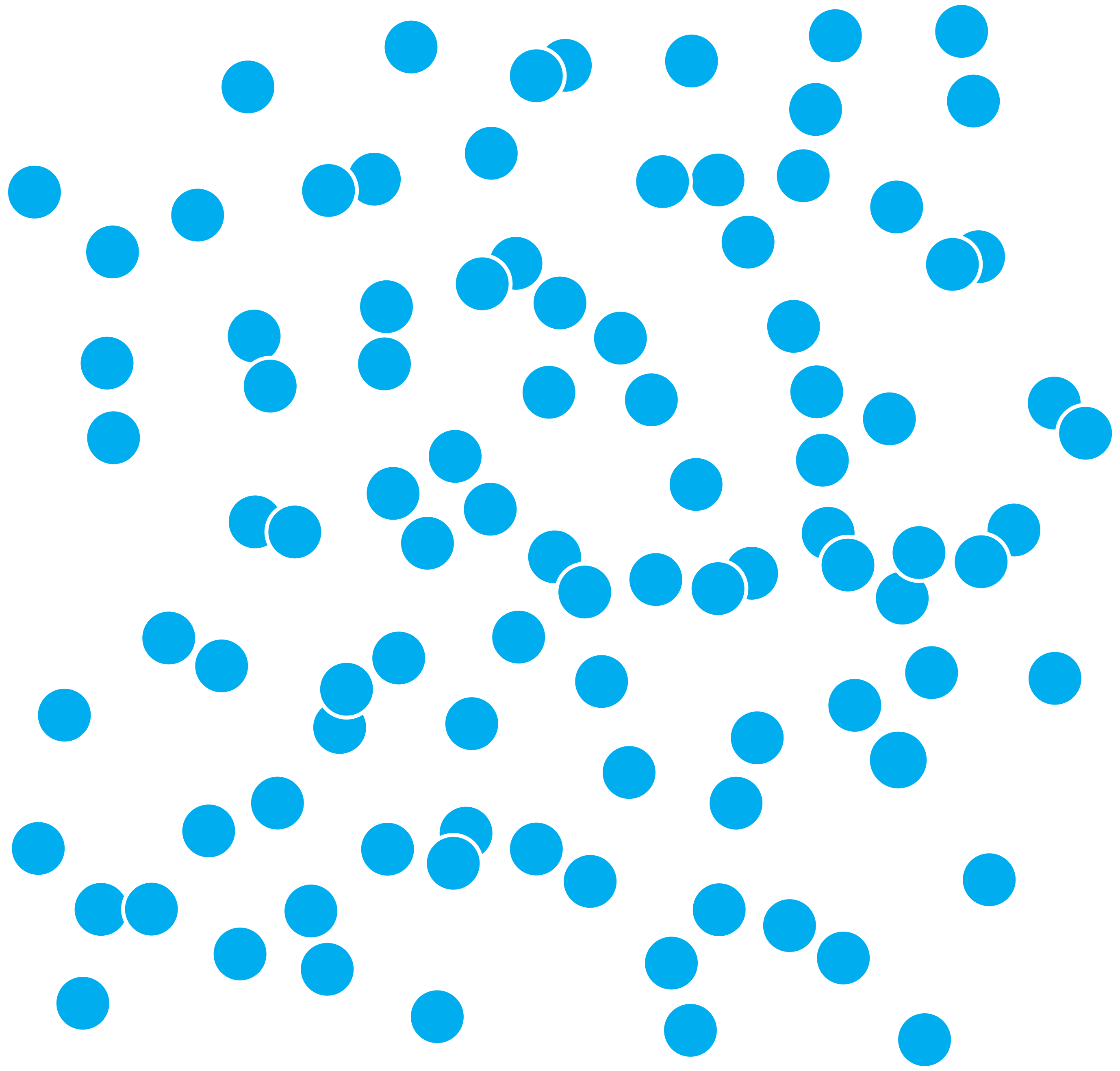
Particles (Constitutive models)

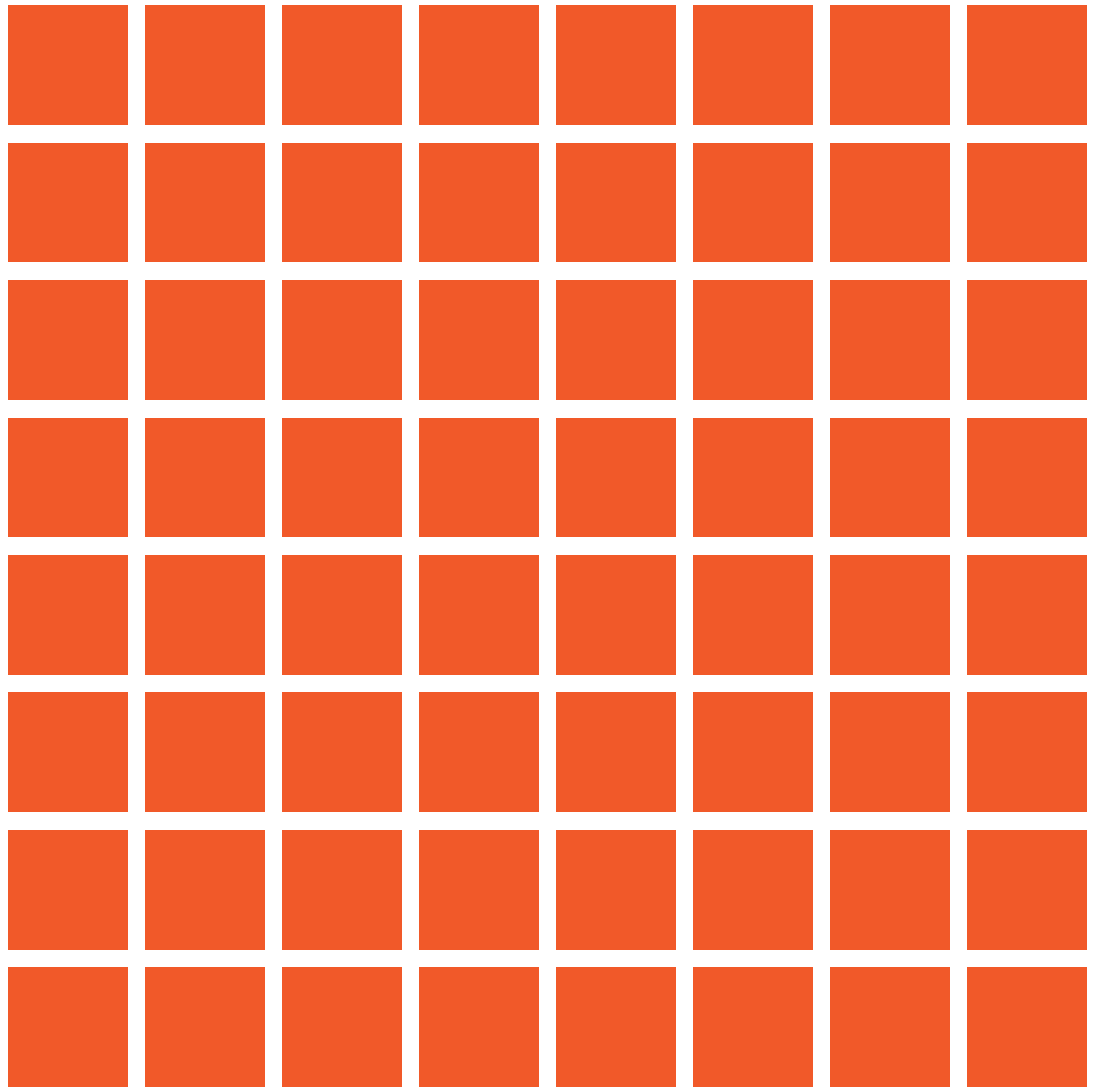
Snow [Stomakhin et al. 2013],

Foam [Ram et al. 2015, Yue et al. 2015]

Sand [Klar et al. 2015, Pradhana et al 2017]

The Material Point Method (MPM)

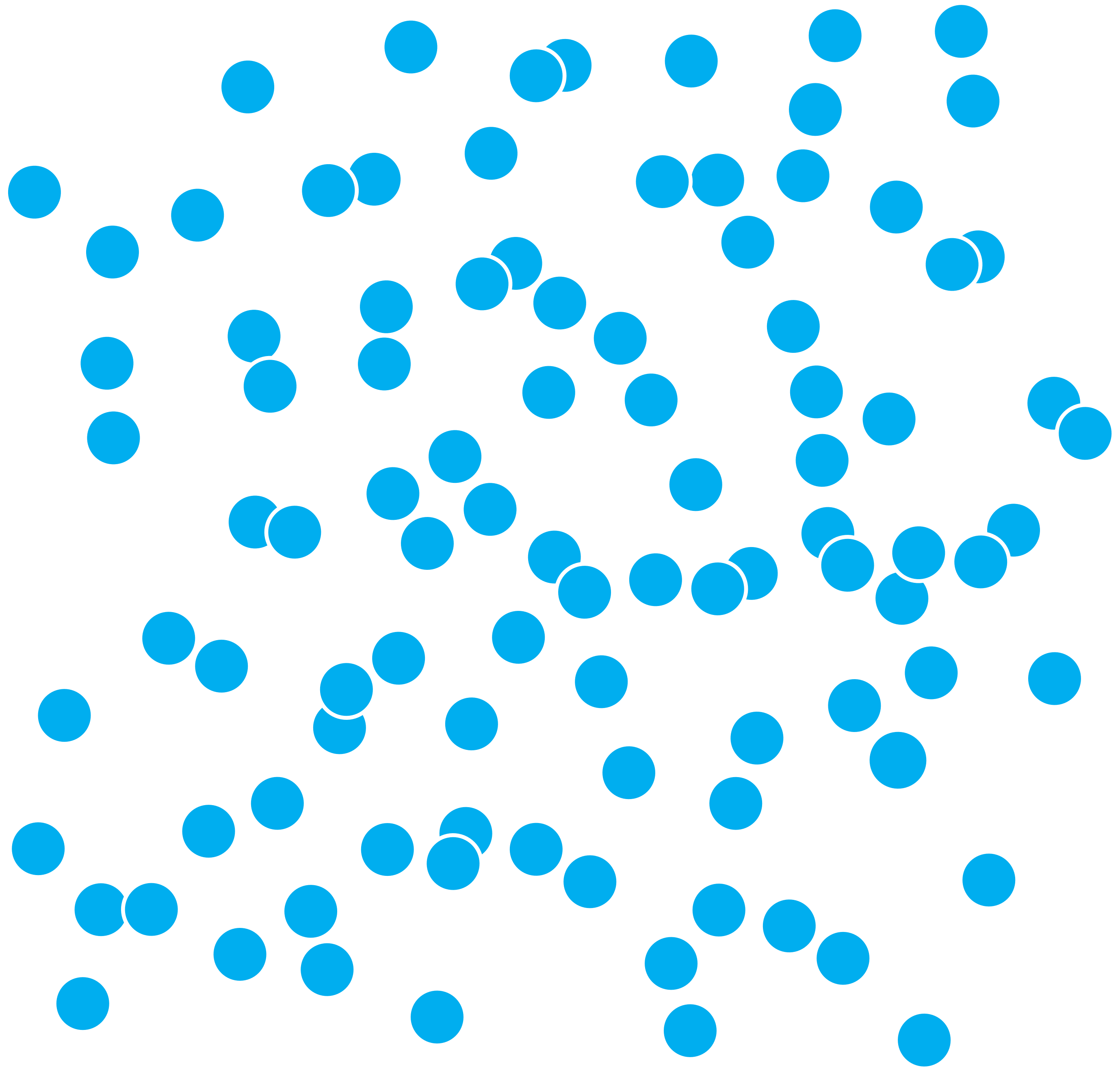




Particles (Constitutive models)
Snow [Stomakhin et al. 2013],
Foam [Ram et al. 2015, Yue et al. 2015]
Sand [Klar et al. 2015, Pradhana et al 2017]

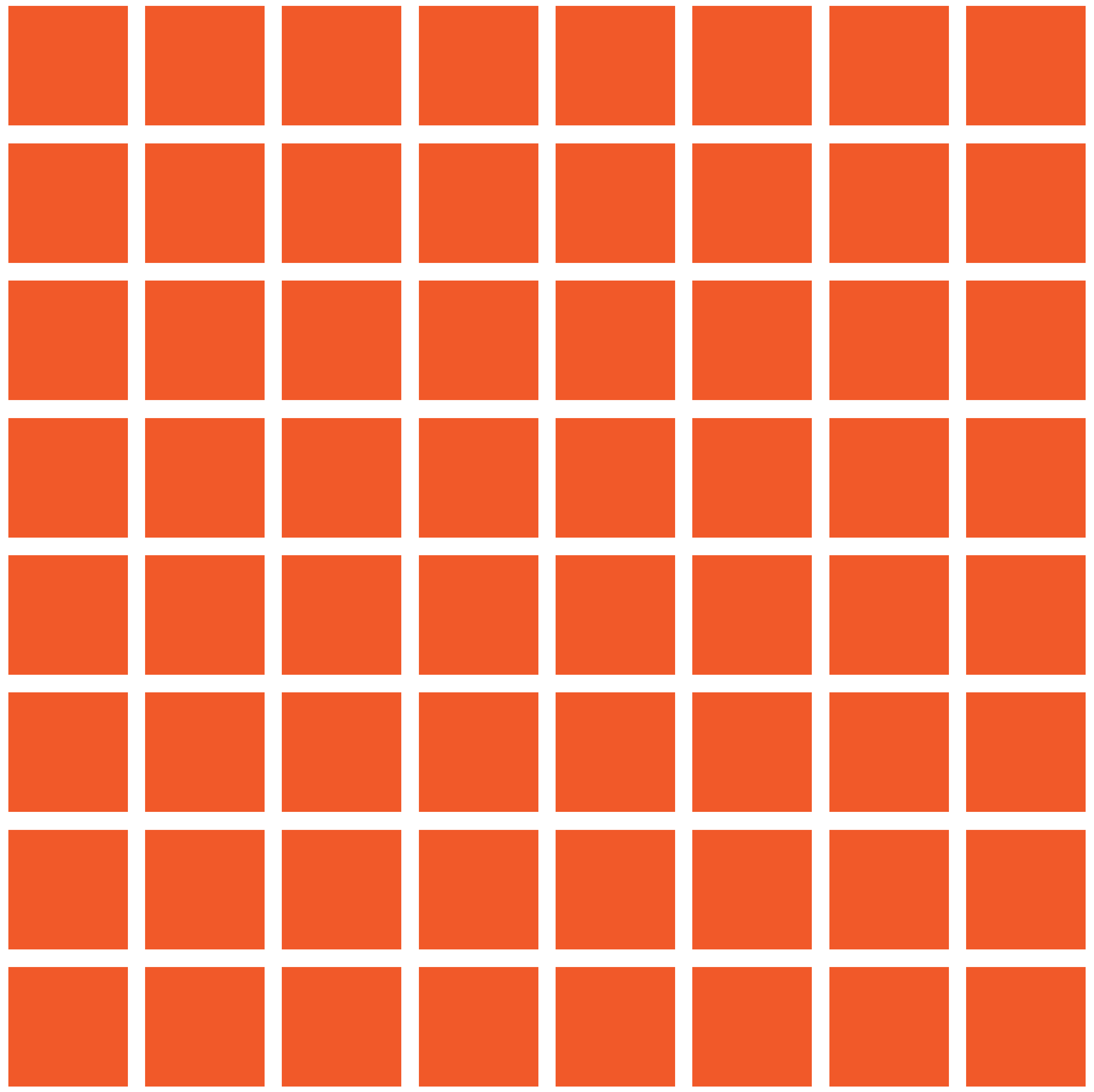
Grid
SPGrid [Setaluri et al. 2014],
OpenVDB [Museth 2013]
Multiple Grids
[Pradhana et al. 2015]

The Material Point Method (MPM)



Particle to Grid (P2G)

Grid to Particle (G2P)



Transfer (Particle in Cell, PIC)

Affine PIC, APIC [Jiang et al. 2016]

Polynomial PIC, PolyPIC [Fu et al. 2017]

High-performance GIMP [Gao et al. 2017]

Moving Least Squares [Hu et al. 2018]

Compatible PIC [Hu et al. 2018]

...

Particles (Constitutive models)

Snow [Stomakhin et al. 2013],

Foam [Ram et al. 2015, Yue et al. 2015]

Sand [Klar et al. 2015, Pradhana et al 2017]

Grid

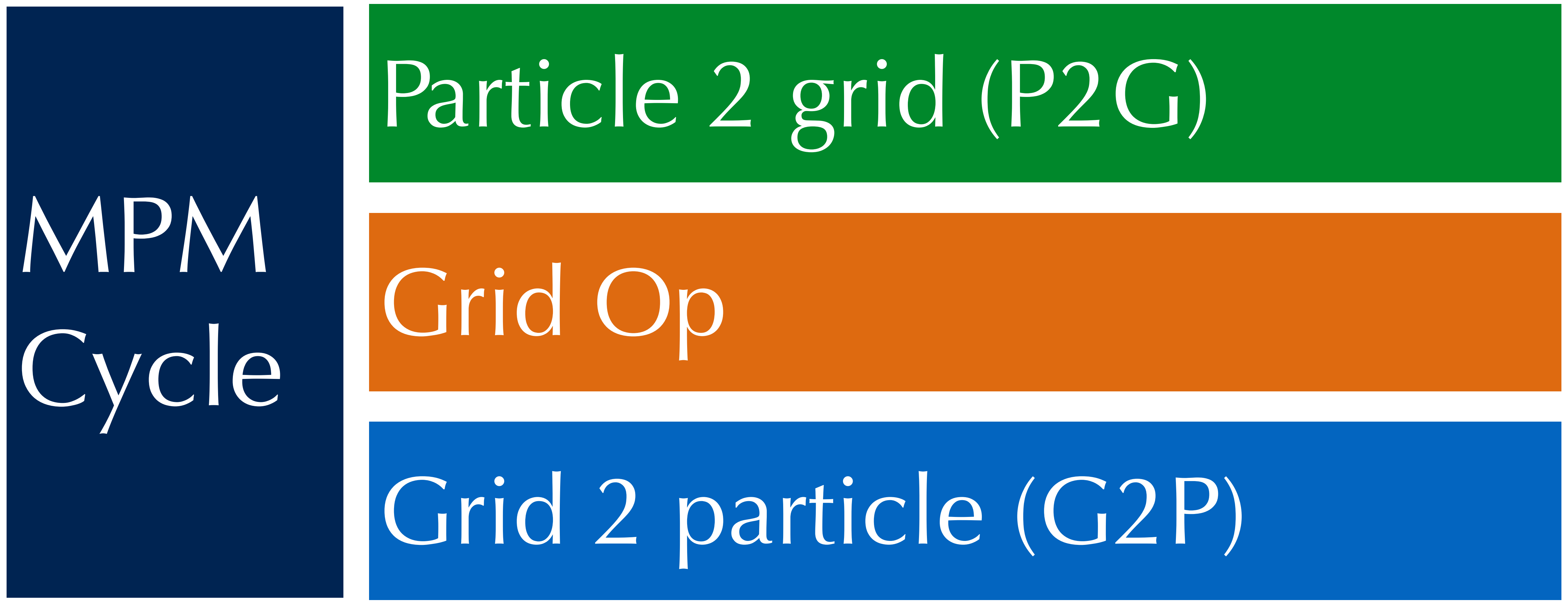
SPGrid [Setaluri et al. 2014],

OpenVDB [Museth 2013]

Multiple Grids

[Pradhana et al. 2015]

The MPM Simulation Cycle



The MPM Simulation Cycle

MPM Cycle

- Particle 2 grid (P2G)
- Grid Op
- Grid 2 particle (G2P)

For each particle:
Update particle
Scatter mass & momentum to nearby 3x3x3 nodes

The MPM Simulation Cycle

MPM Cycle

- Particle 2 grid (P2G)
- Grid Op
- Grid 2 particle (G2P)

Particle 2 grid (P2G) →

For each particle:
  Update particle
  Scatter mass & momentum to nearby 3x3x3 nodes

Grid Op →

For each grid node:
  divide momentum by mass to get velocity;
  apply gravity.

The MPM Simulation Cycle

The MPM Simulation Cycle

MPM Cycle

Particle 2 grid (P2G)

Grid Op

Grid 2 particle (G2P)

For each particle:
Update particle
Scatter mass & momentum to nearby 3x3x3 nodes

For each particle:
Gather velocity info from 3x3x3 nodes;
Update particle.

```cpp
void advance(real dt) {
  std::memset(grid, 0, sizeof(grid));                              // Reset grid
  for (auto &p : particles) {                                             // P2G
    Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
    Vec fx = p.x * inv_dx - base_coord.cast<real>();
    // Quadratic kernels  [http://mpm.graphics   Eqn. 123, with x=fx, fx-1,fx-2]
    Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
             Vec(0.5) * sqr(fx - Vec(0.5))};
    auto e = std::exp(hardening * (1.0_f - p.Jp)), mu=mu_0*e, lambda=lambda_0*e;
    real J = determinant(p.F);         //                          Current volume
    Mat r, s; polar_decomp(p.F, r, s); //Polar decomp. for fixed corotated model
    auto stress =                           // Cauchy stress times dt and inv_dx
        -4*inv_dx*inv_dx*dt*vol*(2*mu*(p.F-r) * transposed(p.F)+lambda*(J-1)*J);
    auto affine = stress+particle_mass*p.C;
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) { // Scatter to grid
        auto dpos = (Vec(i, j) - fx) * dx;
        Vector3 mv(p.v * particle_mass, particle_mass); //translational momentum
        grid[base_coord.x + i][base_coord.y + j] +=
            w[i].x*w[j].y * (mv + Vector3(affine*dpos, 0));
    }
  }
  for(int i = 0; i <= n; i++) for(int j = 0; j <= n; j++) { //For all grid nodes
    auto &g = grid[i][j];
    if (g[2] > 0) {                                // No need for epsilon here
      g /= g[2];                                   //        Normalize by mass
      g += dt * Vector3(0, -200, 0);               //                  Gravity
      real boundary=0.05,x=(real)i/n,y=real(j)/n; //boundary thick.,node coord
      if (x < boundary||x > 1-boundary||y > 1-boundary) g=Vector3(0); //Sticky
      if (y < boundary) g[1] = std::max(0.0_f, g[1]);             //"Separate"
    }
  }
  for (auto &p : particles) {                                // Grid to particle
    Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
    Vec fx = p.x * inv_dx - base_coord.cast<real>();
    Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
             Vec(0.5) * sqr(fx - Vec(0.5))};
    p.C = Mat(0); p.v = Vec(0);
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) {
        auto dpos = (Vec(i, j) - fx),
            grid_v = Vec(grid[base_coord.x + i][base_coord.y + j]);
        auto weight = w[i].x * w[j].y;
        p.v += weight * grid_v;                                      // Velocity
        p.C += 4 * inv_dx * Mat::outer_product(weight * grid_v, dpos); // APIC C
    }
    p.x += dt * p.v;                                                // Advection
    auto F = (Mat(1) + dt * p.C) * p.F;                      // MLS-MPM F-update
    Mat svd_u, sig, svd_v; svd(F, svd_u, sig, svd_v);
    for (int i = 0; i < 2 * int(plastic); i++)                // Snow Plasticity
      sig[i][i] = clamp(sig[i][i], 1.0_f - 2.5e-2_f, 1.0_f + 7.5e-3_f);
    real oldJ = determinant(F); F = svd_u * sig * transposed(svd_v);
    real Jp_new = clamp(p.Jp * oldJ / determinant(F), 0.6_f, 20.0_f);
    p.Jp = Jp_new; p.F = F;
  }
}
```

P2G

Grid Op

G2P

88 Lines of C++

https://github.com/yuanming-hu/taichi_mpm

- 88-Line MLS-MPM (cross-platform)
- High-Performance 3D MLS-MPM+CPIC Solver on CPUs
- MLS-MPM Tetris Game
- **niall**'s MLS-MPM implementation and tutorial in ***Unity***
- **Roberto Toro** made ***mls-mpm.js*** that runs in your browser
- **David Medina** contributed ***mls-mpm88-explained.cpp***

main simulation loop:

→ **void advance(float dt) [54 lines]**

Demos!

Particle 2 Grid

```
for (auto &p : particles) {                                           // P2G
  Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
  Vec fx = p.x * inv_dx - base_coord.cast<real>();
  // Quadratic kernels  [http://mpm.graphics   Eqn. 123, with x=fx, fx-1,fx-2]
  Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
           Vec(0.5) * sqr(fx - Vec(0.5))};
  auto e = std::exp(hardening * (1.0_f - p.Jp)), mu=mu_0*e, lambda=lambda_0*e;
  real J = determinant(p.F);         //                         Current volume
  Mat r, s; polar_decomp(p.F, r, s); //Polar decomp. for fixed corotated model
  auto stress =                           // Cauchy stress times dt and inv_dx
      -4*inv_dx*inv_dx*dt*vol*(2*mu*(p.F-r) * transposed(p.F)+lambda*(J-1)*J);
  auto affine = stress+particle_mass*p.C;
  for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) { // Scatter to grid
      auto dpos = (Vec(i, j) - fx) * dx;
      Vector3 mv(p.v * particle_mass, particle_mass); //translational momentum
      grid[base_coord.x + i][base_coord.y + j] +=
          w[i].x*w[j].y * (mv + Vector3(affine*dpos, 0));
    }
}
```

Particle 2 Grid

```
for (auto &p : particles) {                                              // P2G
  Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
  Vec fx = p.x * inv_dx - base_coord.cast<real>();
  // Quadratic kernels  [http://mpm.graphics   Eqn. 123, with x=fx, fx-1,fx-2]
  Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
           Vec(0.5) * sqr(fx - Vec(0.5))};
  auto e = std::exp(hardening * (1.0_f - p.Jp)), mu=mu_0*e, lambda=lambda_0*e;
  real J = determinant(p.F);         //                         Current volume
  Mat r, s; polar_decomp(p.F, r, s); //Polar decomp. for fixed corotated model
  auto stress =                           // Cauchy stress times dt and inv_dx
      -4*inv_dx*inv_dx*dt*vol*(2*mu*(p.F-r) * transposed(p.F)+lambda*(J-1)*J);
  auto affine = stress+particle_mass*p.C;
  for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) { // Scatter to grid
      auto dpos = (Vec(i, j) - fx) * dx;
      Vector3 mv(p.v * particle_mass, particle_mass); //translational momentum
      grid[base_coord.x + i][base_coord.y + j] +=
          w[i].x*w[j].y * (mv + Vector3(affine*dpos, 0));
    }
}
```

Particle Processing

Particle 2 Grid

```
for (auto &p : particles) {                                              // P2G
  Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
  Vec fx = p.x * inv_dx - base_coord.cast<real>();
  // Quadratic kernels  [http://mpm.graphics   Eqn. 123, with x=fx, fx-1,fx-2]
  Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
           Vec(0.5) * sqr(fx - Vec(0.5))};
  auto e = std::exp(hardening * (1.0_f - p.Jp)), mu=mu_0*e, lambda=lambda_0*e;
  real J = determinant(p.F);         //                       Current volume
  Mat r, s; polar_decomp(p.F, r, s); //Polar decomp. for fixed corotated model
  auto stress =                           // Cauchy stress times dt and inv_dx
      -4*inv_dx*inv_dx*dt*vol*(2*mu*(p.F-r) * transposed(p.F)+lambda*(J-1)*J);
  auto affine = stress+particle_mass*p.C;
  for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) { // Scatter to grid
      auto dpos = (Vec(i, j) - fx) * dx;
      Vector3 mv(p.v * particle_mass, particle_mass); //translational momentum
      grid[base_coord.x + i][base_coord.y + j] +=
          w[i].x*w[j].y * (mv + Vector3(affine*dpos, 0));
  }
}
```

Particle Processing

Grid Rasterization
(3^3=27 nodes in 3D!)

Grid 2 Particle

```
for (auto &p : particles) {                                  // Grid to particle
  Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
  Vec fx = p.x * inv_dx - base_coord.cast<real>();
  Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
           Vec(0.5) * sqr(fx - Vec(0.5))};
  p.C = Mat(0); p.v = Vec(0);
  for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) {
      auto dpos = (Vec(i, j) - fx),
          grid_v = Vec(grid[base_coord.x + i][base_coord.y + j]);
      auto weight = w[i].x * w[j].y;
      p.v += weight * grid_v;                                        // Velocity
      p.C += 4 * inv_dx * Mat::outer_product(weight * grid_v, dpos); // APIC C
    }
  p.x += dt * p.v;                                                  // Advection
  auto F = (Mat(1) + dt * p.C) * p.F;                       // MLS-MPM F-update
  Mat svd_u, sig, svd_v; svd(F, svd_u, sig, svd_v);
  for (int i = 0; i < 2 * int(plastic); i++)                 // Snow Plasticity
    sig[i][i] = clamp(sig[i][i], 1.0_f - 2.5e-2_f, 1.0_f + 7.5e-3_f);
  real oldJ = determinant(F); F = svd_u * sig * transposed(svd_v);
  real Jp_new = clamp(p.Jp * oldJ / determinant(F), 0.6_f, 20.0_f);
  p.Jp = Jp_new; p.F = F;
}
```

Grid 2 Particle

```
for (auto &p : particles) {                              // Grid to particle
  Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
  Vec fx = p.x * inv_dx - base_coord.cast<real>();
  Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
           Vec(0.5) * sqr(fx - Vec(0.5))};
  p.C = Mat(0); p.v = Vec(0);
  for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) {
      auto dpos = (Vec(i, j) - fx),
          grid_v = Vec(grid[base_coord.x + i][base_coord.y + j]);
      auto weight = w[i].x * w[j].y;
      p.v += weight * grid_v;                                    // Velocity
      p.C += 4 * inv_dx * Mat::outer_product(weight * grid_v, dpos); // APIC C
    }
  p.x += dt * p.v;                                              // Advection
  auto F = (Mat(1) + dt * p.C) * p.F;                      // MLS-MPM F-update
  Mat svd_u, sig, svd_v; svd(F, svd_u, sig, svd_v);
  for (int i = 0; i < 2 * int(plastic); i++)              // Snow Plasticity
    sig[i][i] = clamp(sig[i][i], 1.0_f - 2.5e-2_f, 1.0_f + 7.5e-3_f);
  real oldJ = determinant(F); F = svd_u * sig * transposed(svd_v);
  real Jp_new = clamp(p.Jp * oldJ / determinant(F), 0.6_f, 20.0_f);
  p.Jp = Jp_new; p.F = F;
}
```

Particle Processing

Grid 2 Particle

```
for (auto &p : particles) {                                  // Grid to particle
  Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
  Vec fx = p.x * inv_dx - base_coord.cast<real>();
  Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
           Vec(0.5) * sqr(fx - Vec(0.5))};
  p.C = Mat(0); p.v = Vec(0);
  for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) {
      auto dpos = (Vec(i, j) - fx),
          grid_v = Vec(grid[base_coord.x + i][base_coord.y + j]);
      auto weight = w[i].x * w[j].y;
      p.v += weight * grid_v;                                        // Velocity
      p.C += 4 * inv_dx * Mat::outer_product(weight * grid_v, dpos); // APIC C
    }
  p.x += dt * p.v;                                                  // Advection
  auto F = (Mat(1) + dt * p.C) * p.F;                          // MLS-MPM F-update
  Mat svd_u, sig, svd_v; svd(F, svd_u, sig, svd_v);
  for (int i = 0; i < 2 * int(plastic); i++)                    // Snow Plasticity
    sig[i][i] = clamp(sig[i][i], 1.0_f - 2.5e-2_f, 1.0_f + 7.5e-3_f);
  real oldJ = determinant(F); F = svd_u * sig * transposed(svd_v);
  real Jp_new = clamp(p.Jp * oldJ / determinant(F), 0.6_f, 20.0_f);
  p.Jp = Jp_new; p.F = F;
}
```

Particle Processing

Grid Gathering
(3^3=27 nodes in 3D!)

Grid 2 Particle

```
for (auto &p : particles) {                                  // Grid to particle
  Vector2i base_coord=(p.x*inv_dx-Vec(0.5_f)).cast<int>();//element-wise floor
  Vec fx = p.x * inv_dx - base_coord.cast<real>();
  Vec w[3]{Vec(0.5) * sqr(Vec(1.5) - fx), Vec(0.75) - sqr(fx - Vec(1.0)),
           Vec(0.5) * sqr(fx - Vec(0.5))};
  p.C = Mat(0); p.v = Vec(0);
  for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) {
    auto dpos = (Vec(i, j) - fx),
        grid_v = Vec(grid[base_coord.x + i][base_coord.y + j]);
    auto weight = w[i].x * w[j].y;
    p.v += weight * grid_v;                                          // Velocity
    p.C += 4 * inv_dx * Mat::outer_product(weight * grid_v, dpos); // APIC C
  }
  p.x += dt * p.v;                                                  // Advection
  auto F = (Mat(1) + dt * p.C) * p.F;                        // MLS-MPM F-update
  Mat svd_u, sig, svd_v; svd(F, svd_u, sig, svd_v);
  for (int i = 0; i < 2 * int(plastic); i++)                  // Snow Plasticity
    sig[i][i] = clamp(sig[i][i], 1.0_f - 2.5e-2_f, 1.0_f + 7.5e-3_f);
  real oldJ = determinant(F); F = svd_u * sig * transposed(svd_v);
  real Jp_new = clamp(p.Jp * oldJ / determinant(F), 0.6_f, 20.0_f);
  p.Jp = Jp_new; p.F = F;
}
```

Particle Processing

Grid Gathering
(3^3=27 nodes in 3D!)

Particle Processing

Key Computational Patterns:

a) Streaming Particle Data

b) Particle-Grid Interaction

How much do implementation practices affect performance?

40% ?

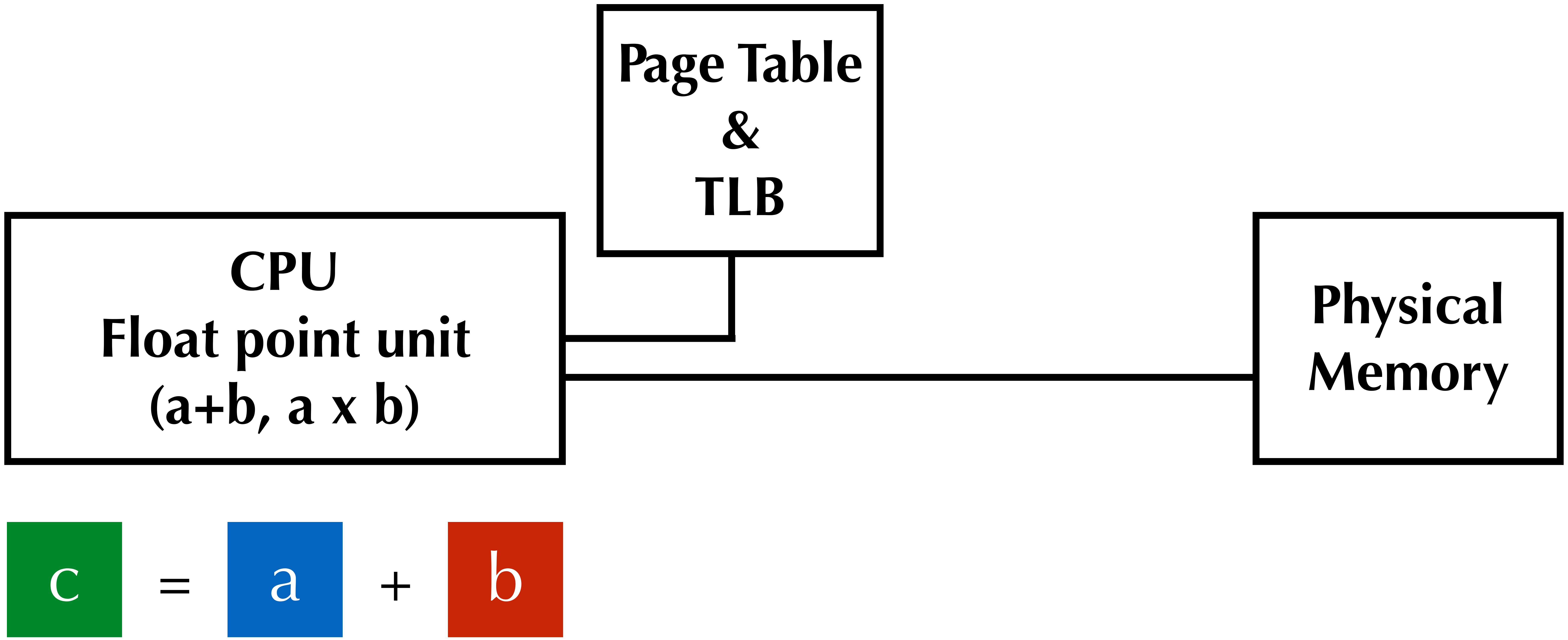
How much do implementation practices affect performance?

10x higher performance!

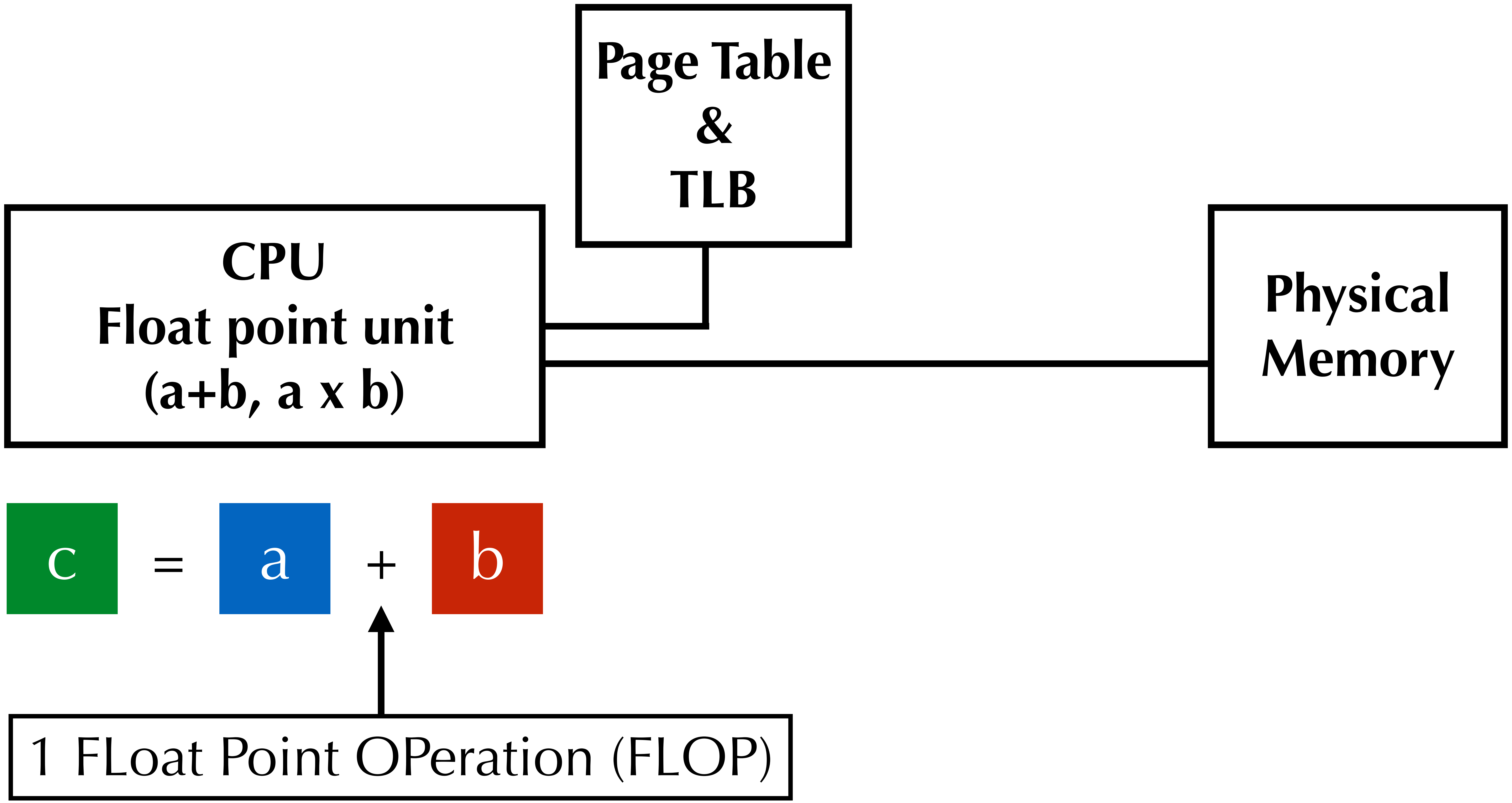
without multithreading

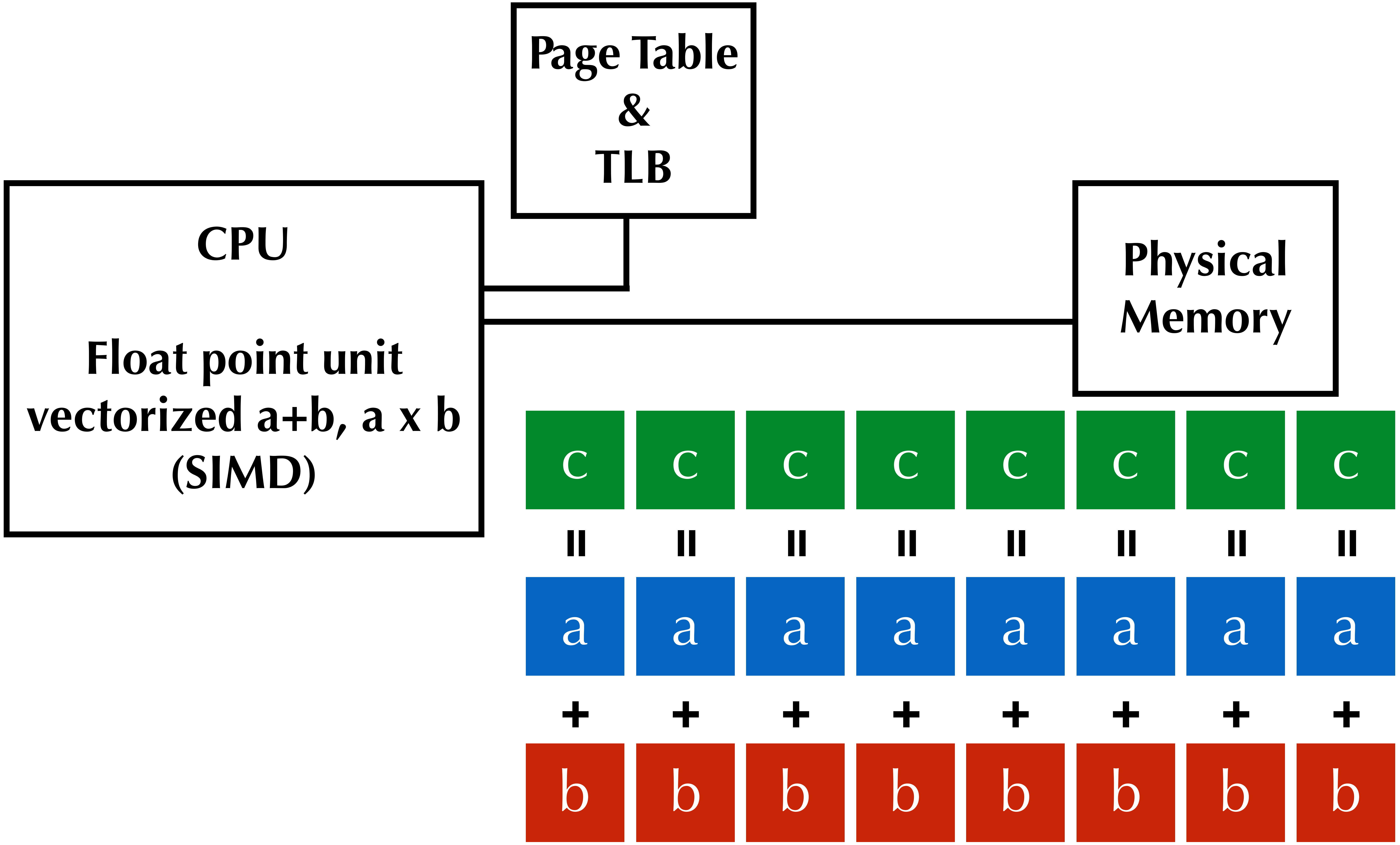
- Instruction-level parallelism
- Vectorization
- High-quality instruction generation
- Lock-free parallelization
- Bandwidth-saving data structures
- …

Recap:
Modern Computer Architecture

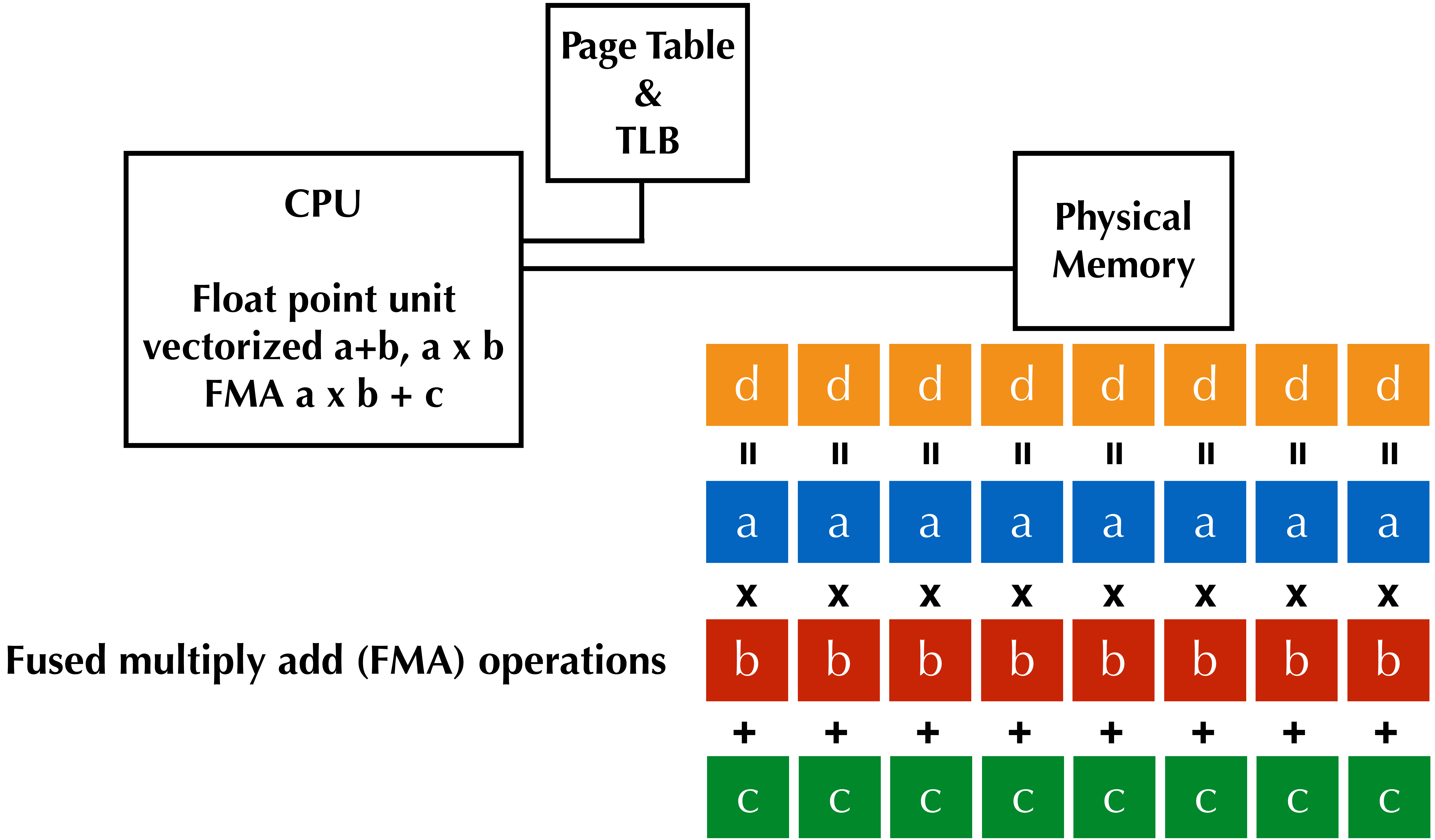


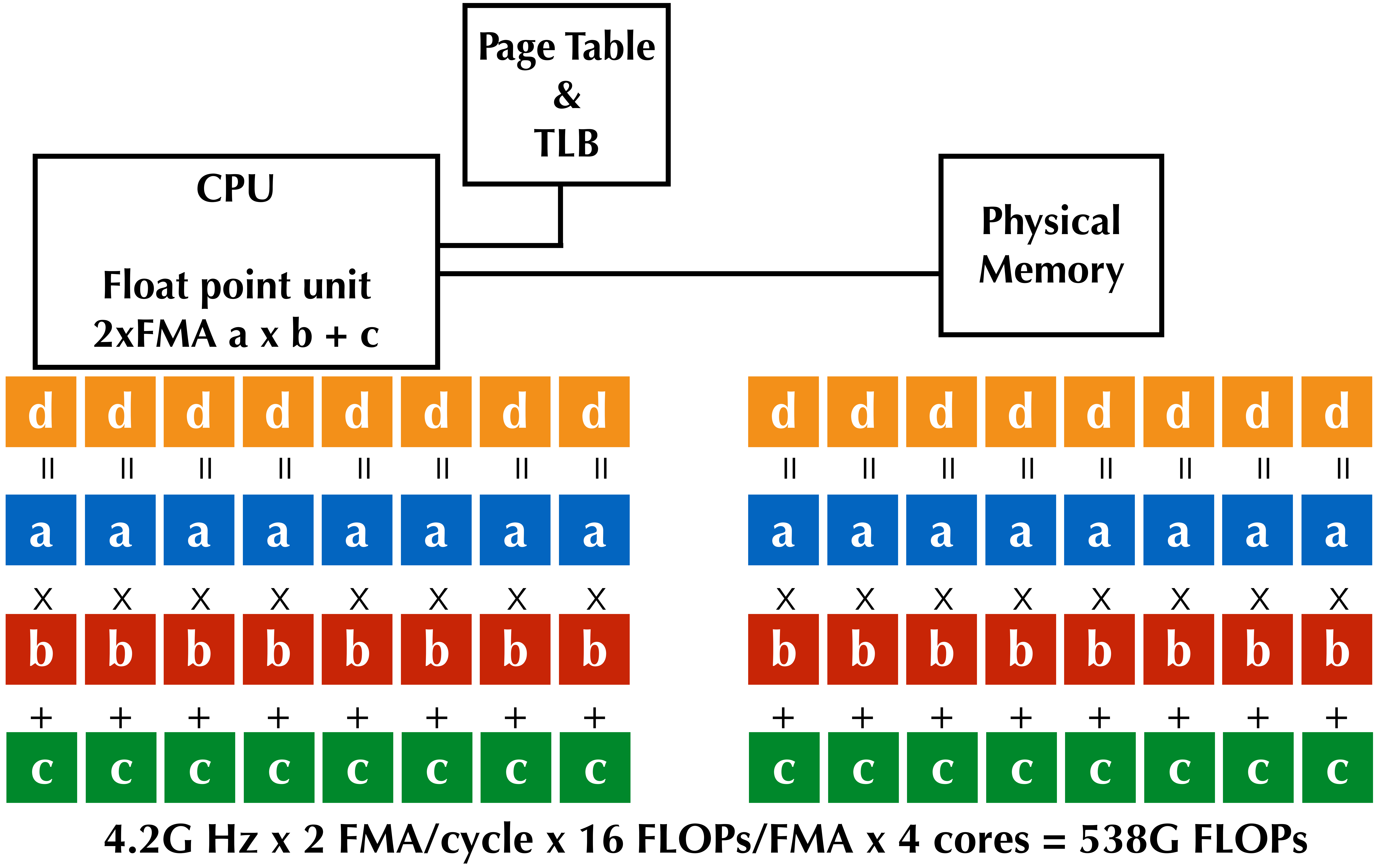
Recap:
Modern Computer Architecture

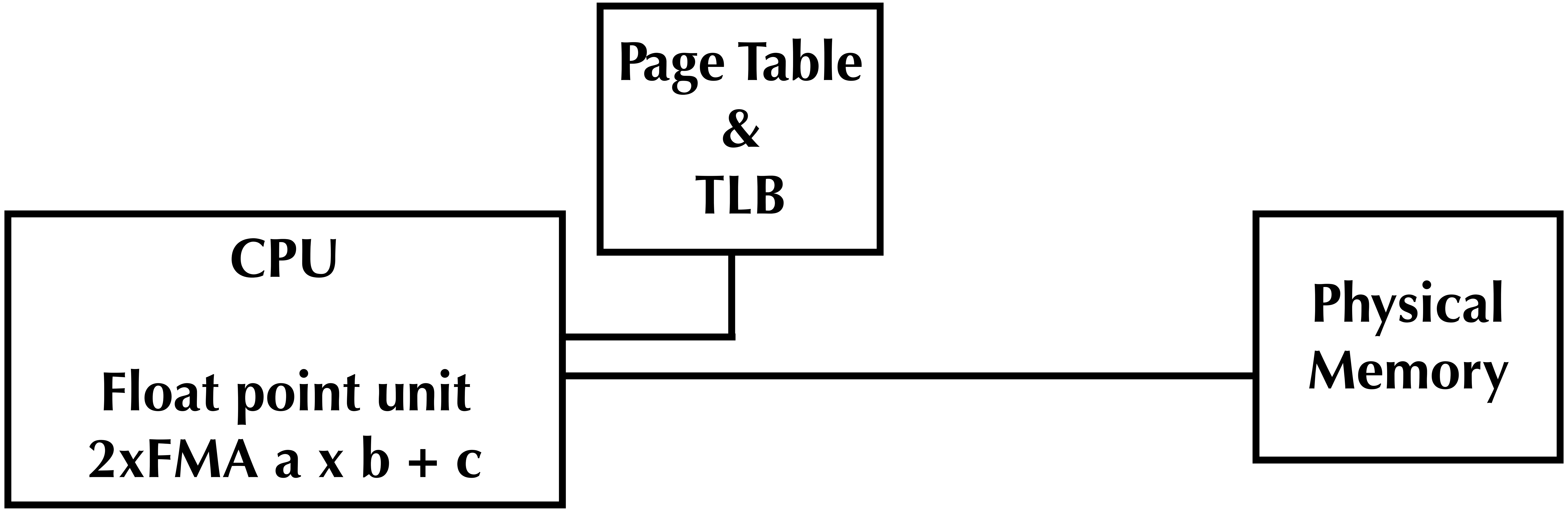


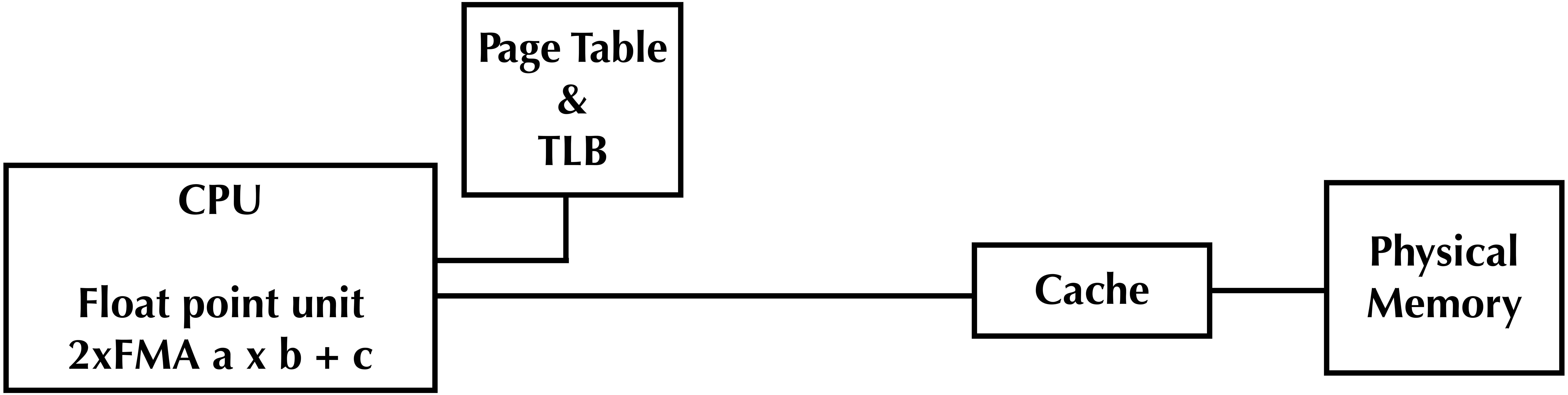


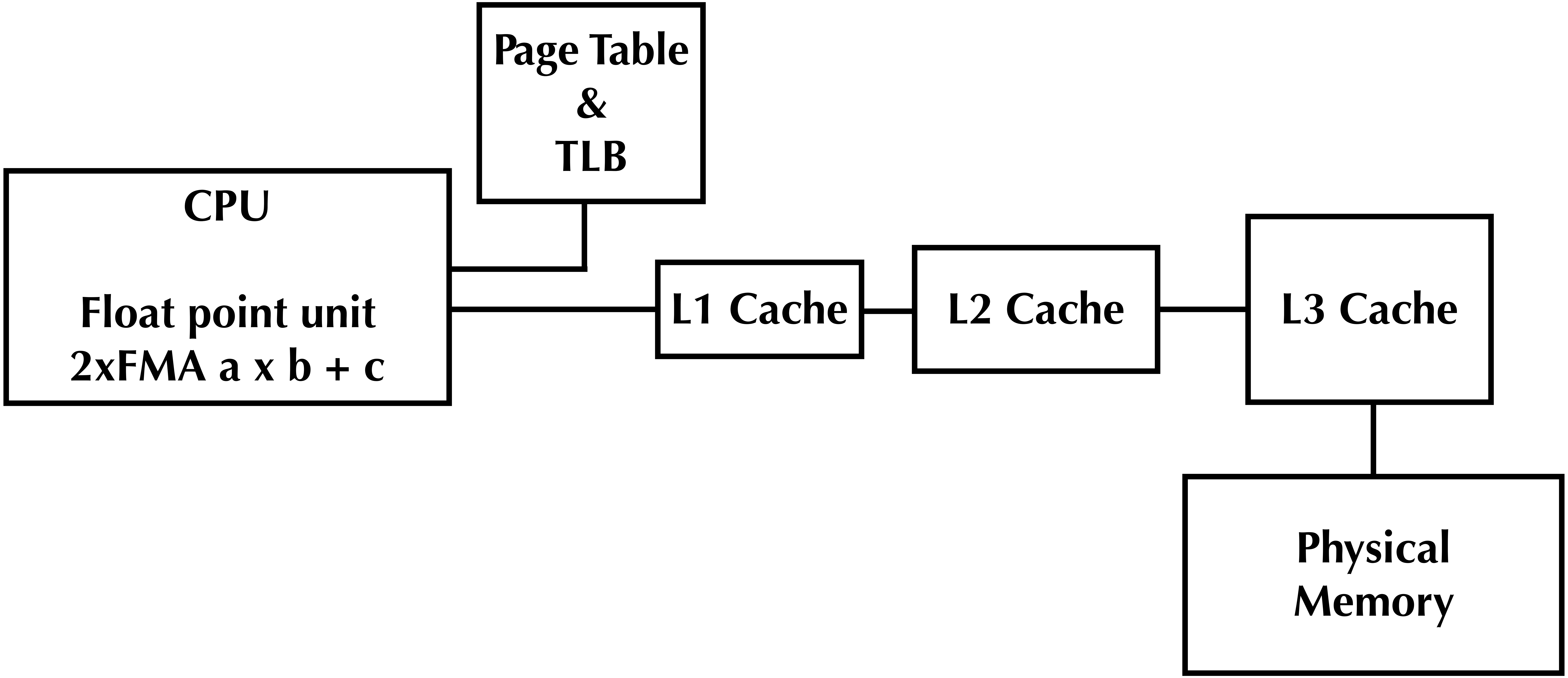
AVX2: 8 single precision/4 double precision float point number operations in a row
AVX512: 16 single precision/8 double precision float point number operations in a row


Page Table
&
TLB
CPU
Float point unit
vectorized a+b, a x b
FMA a x b + c
Physical
Memory
d d d d d d d d
= = = = = = = =
a a a a a a a a
x x x x x x x x
b b b b b b b b
+ + + + + + + +
c c c c c c c c
Fused multiply add (FMA) operations


Page Table
&
TLB
CPU
Float point unit
2xFMA a x b + c
Physical
Memory
4.2G Hz x 2 FMA/cycle x 16 FLOPs/FMA x 4 cores = 538G FLOPs


Page Table
&
TLB
CPU
Float point unit
2xFMA a x b + c
Physical
Memory


Page Table
&
TLB
CPU
Float point unit
2xFMA a x b + c
Cache
Physical
Memory


Page Table
&
TLB
CPU
Float point unit
2xFMA a x b + c
L1 Cache
L2 Cache
L3 Cache
Physical
Memory

(Part of) the Memory Hierarchy

Main Memory
35.8 GB/s
256 cyc latency

L3 cache 2M/core
134.4 GB/s
42 cyc latency

L2 cache 256KB
268.8 GB/s
12 cyc latency

L2 Unified TLB (STLB)
4 KB/2MB pages - 1536 entries
1G pages - 16 entries

L1 data cache 32KB
403.2 GB/s
4 cyc latency

L1 Data TLB
4 KB pages - 64 entries
2/4 MB pages - 32 entries
1G pages - 4 entries

CPU core

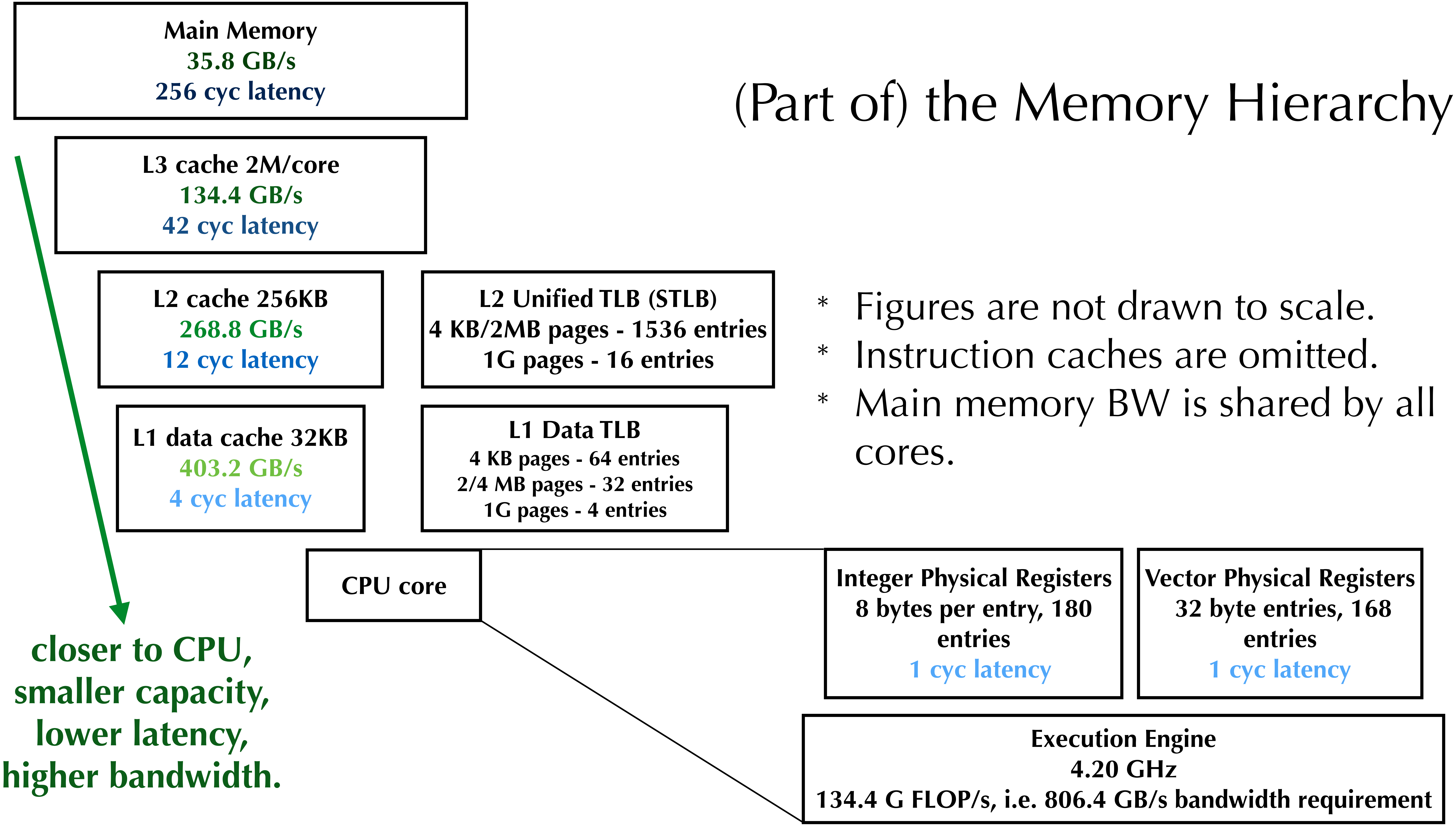
Integer Physical Registers
8 bytes per entry, 180 entries
1 cyc latency

Vector Physical Registers
32 byte entries, 168 entries
1 cyc latency

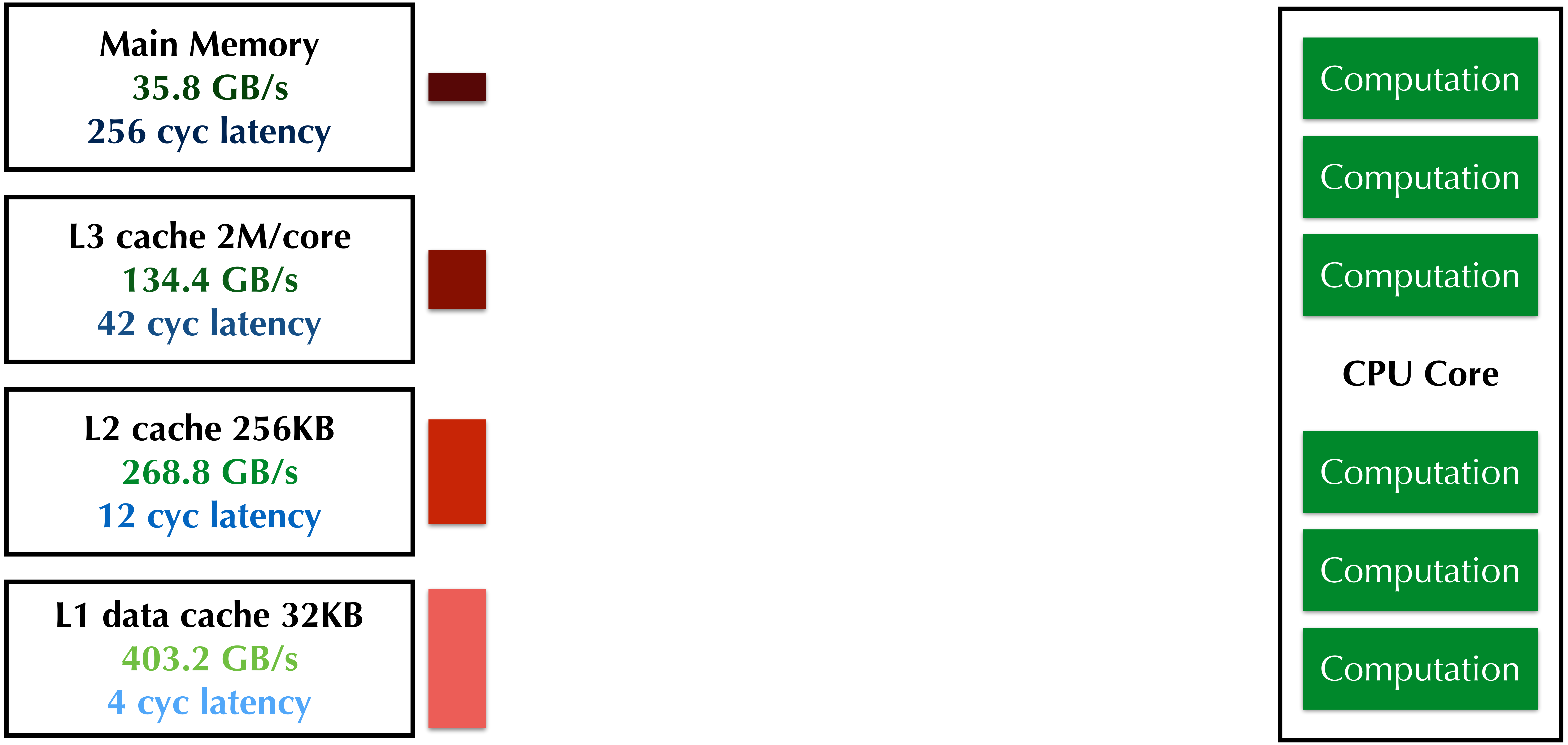
Execution Engine
4.20 GHz
134.4 G FLOP/s, i.e. 806.4 GB/s bandwidth requirement

* Figures are not drawn to scale.
* Instruction caches are omitted.
* Main memory BW is shared by all cores.

(Part of) the Memory Hierarchy



- Figures are not drawn to scale.
- Instruction caches are omitted.
- Main memory BW is shared by all cores.



* Caches are not drawn to scale.
* Data collected from the Intel Skylake architecture, single core.
* There can be multiple data transfers happening simultaneously.
* Access to slower memory is invoked by faster memory cache miss.

* Caches are not drawn to scale.
* Data collected from the Intel Skylake architecture, single core.
* There can be multiple data transfers happening simultaneously.
* Access to slower memory is invoked by faster memory cache miss.

Locality

- **Spatial locality**: try to access spatially neighboring data in main memory
  - Higher cacheline utilization
  - Fewer Cache/TLB misses
  - Better hardware prefetching on CPUs
- **Temporal locality**: reuse the data as much as you can
  - Higher cache-hit rates
  - Lower main memory bandwidth pressure
- Shrink the working set, so that data resides in lower-level (higher throughput, lower latency) memory

Main Memory
35.8 GB/s in total, **9 GB/s per core**
256 cyc latency

Execution Engine
4.20 GHz
134.4 G FLOP/s, **i.e. 806.4 GB/s bandwidth requirement**

Main Memory

35.8 GB/s in total, **9 GB/s per core**

256 cyc latency

Without data reuse (temporal locality),
Processors need 100x higher bandwidth than
what they actually have!

Execution Engine

4.20 GHz

134.4 G FLOP/s, **i.e. 806.4 GB/s bandwidth requirement**

The era of slow memory…

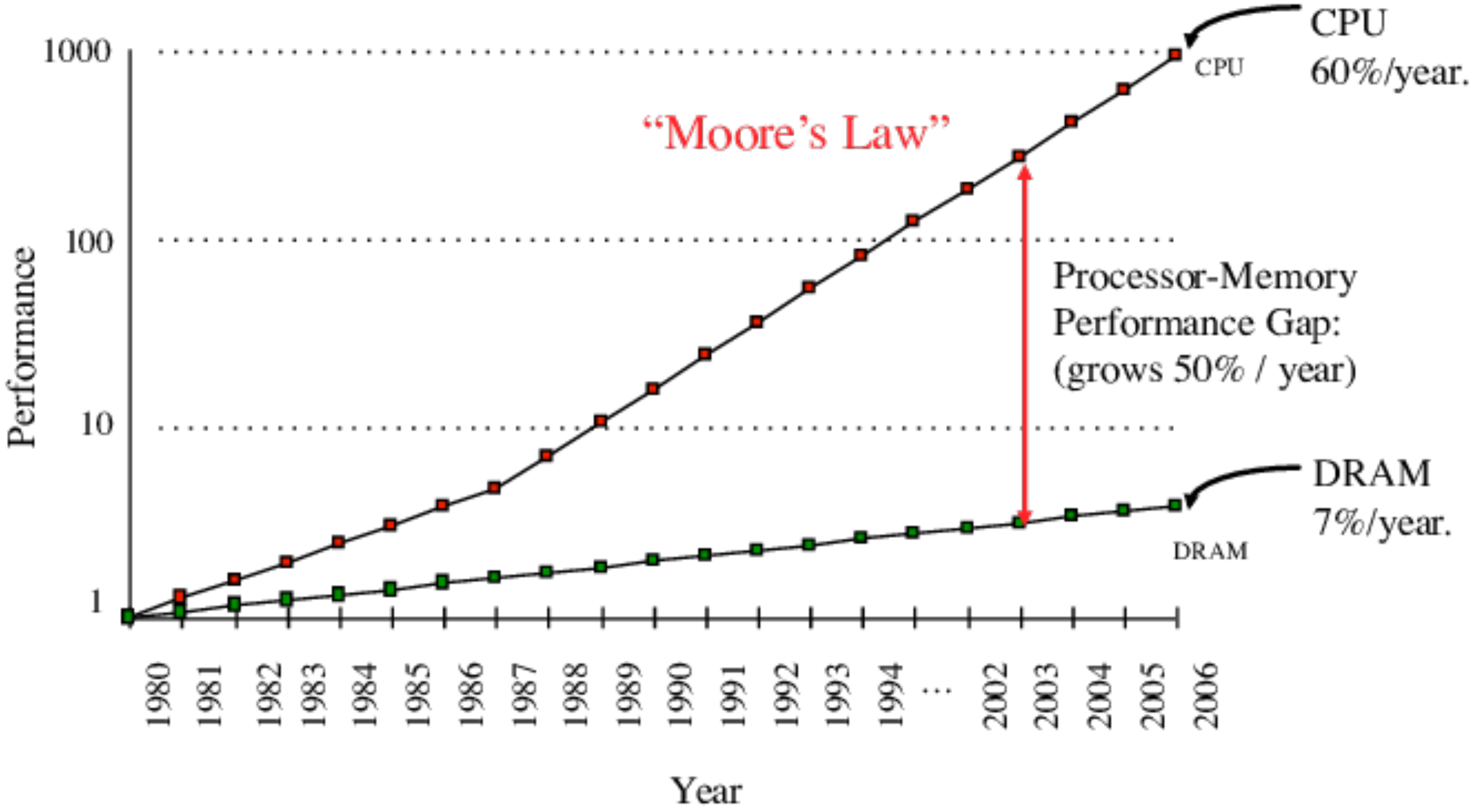
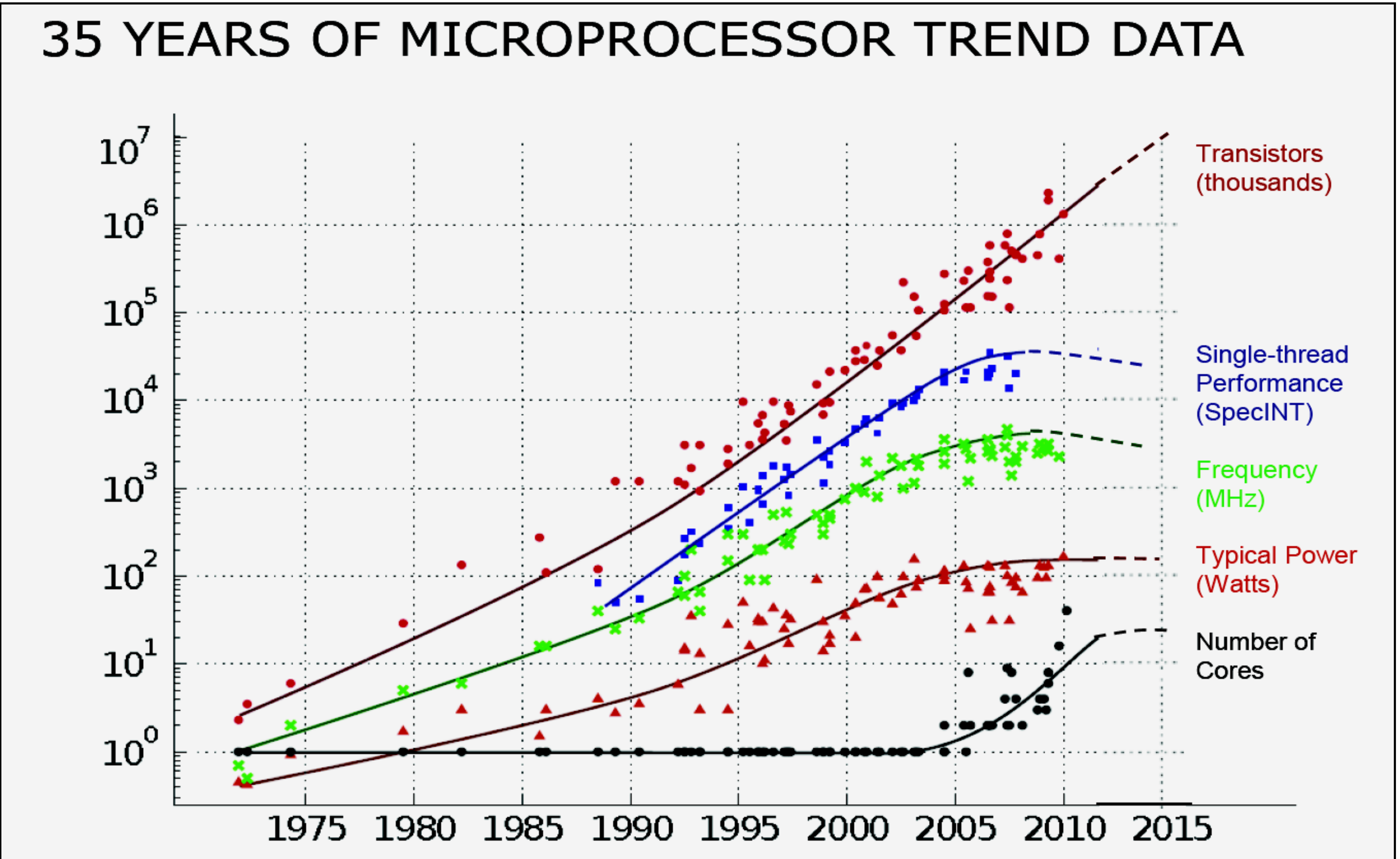


Figure from *High Performance by Exploiting Information Locality through Reverse Computing*, Bahi & Eisenbeis

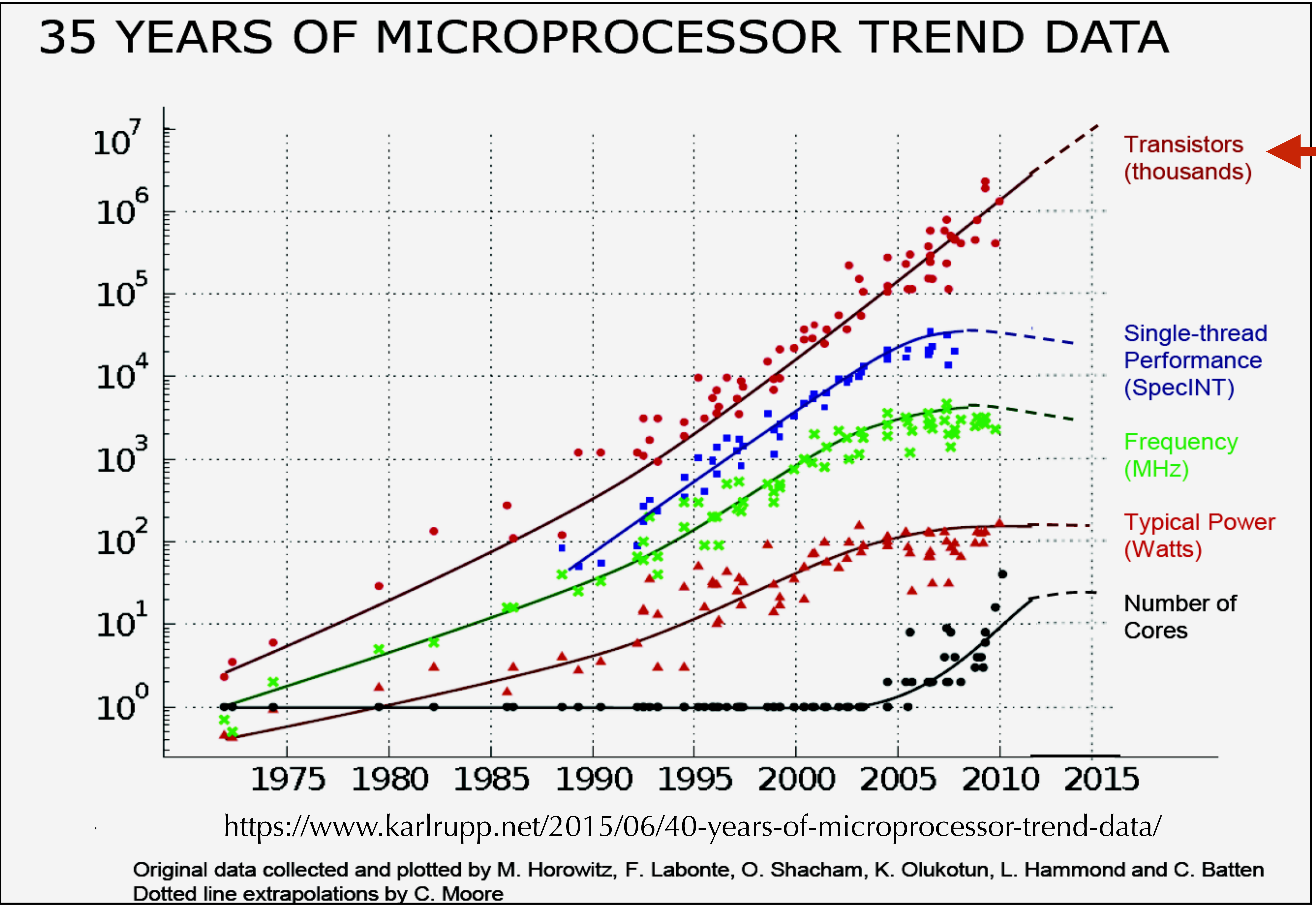
...and parallelism.



https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

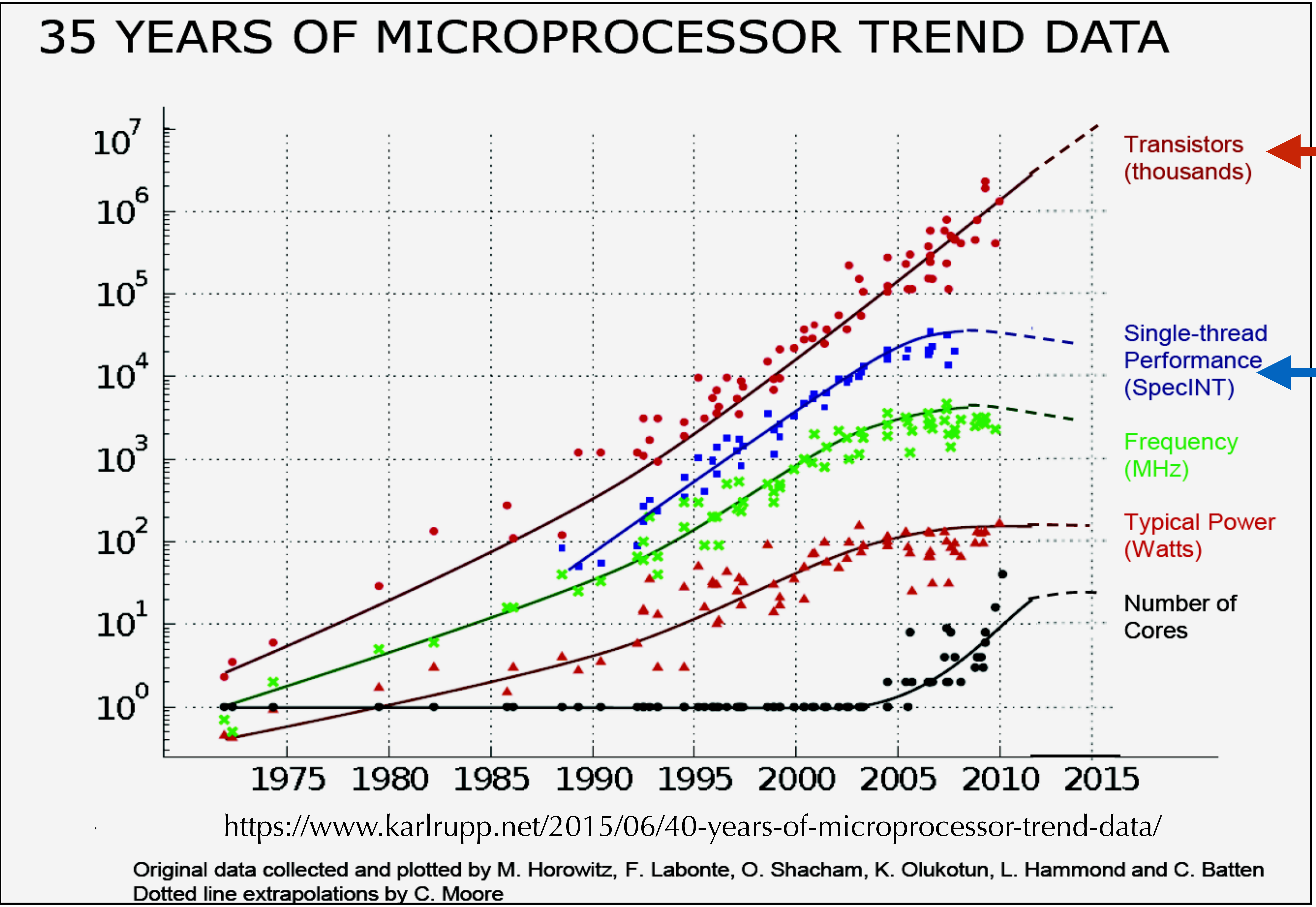
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

…and parallelism.



#transistors keeps growing

…and parallelism.

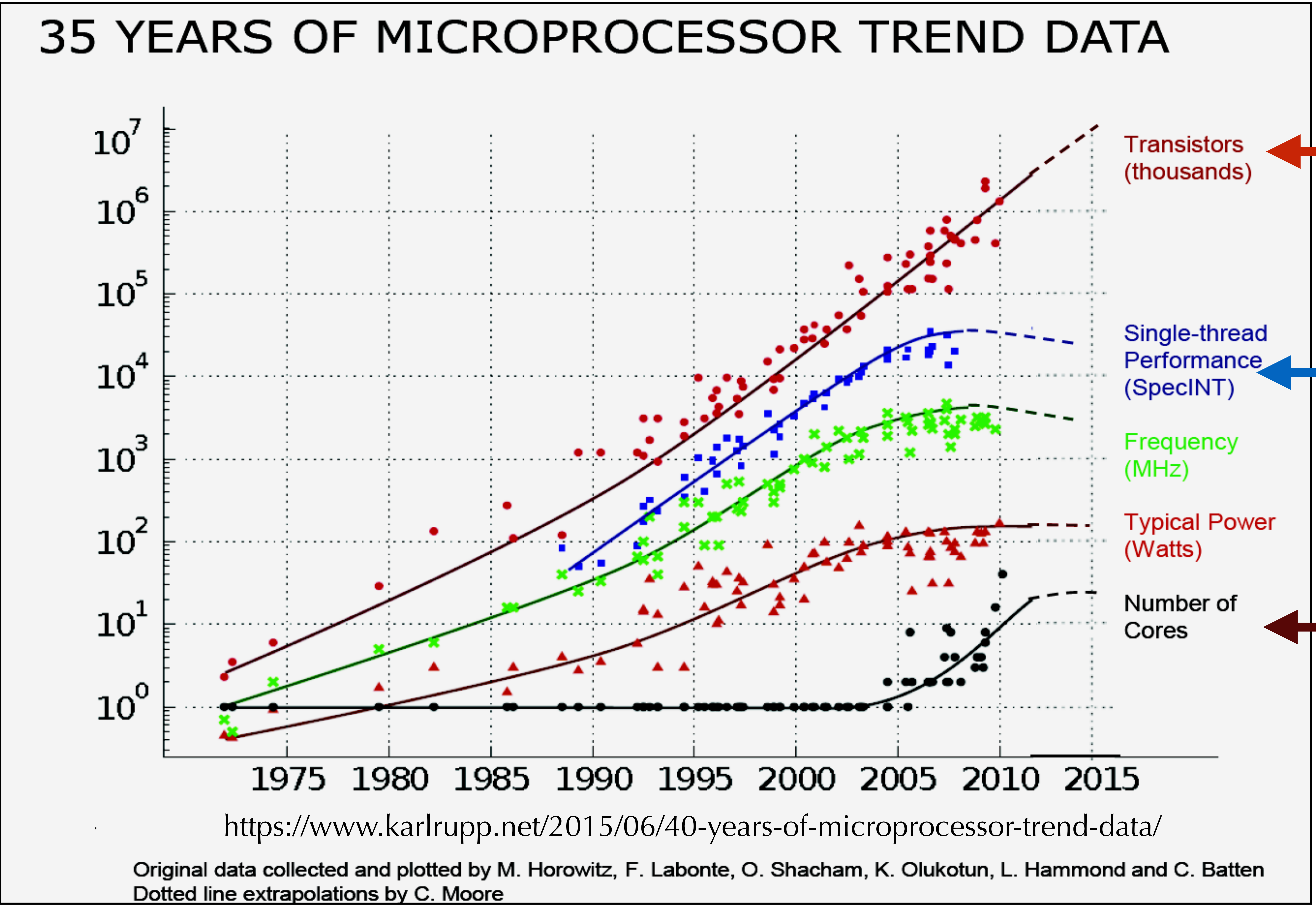


https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

#transistors keeps growing

Single-core performance stops growing

…and parallelism.

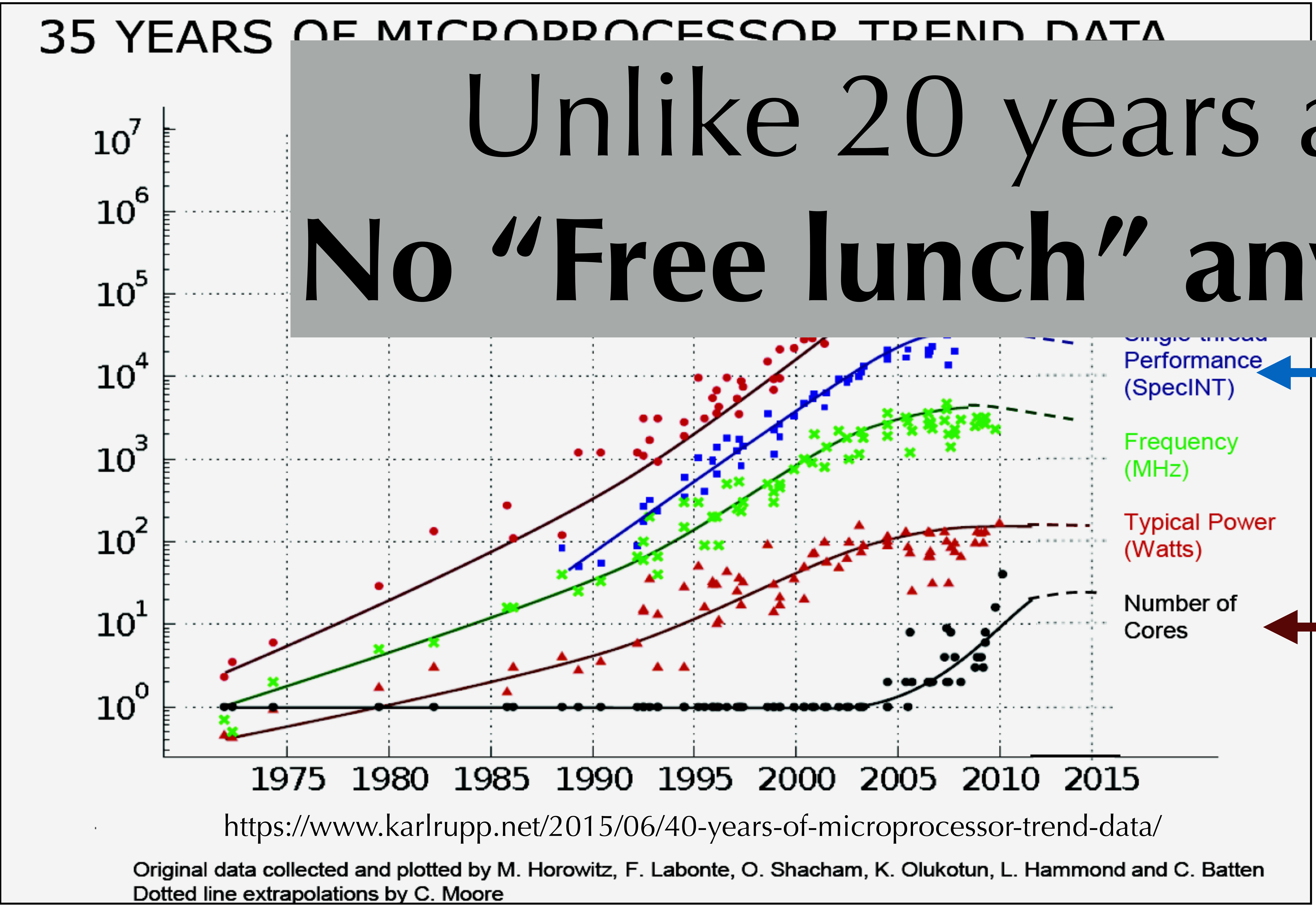


#transistors keeps growing

Single-core performance stops growing

Instead of "better" cores, we have "more" cores (with wider SIMD)

…and parallelism.



Unlike 20 years ago,

No "Free lunch" anymore!

…ps growing

Single-core performance stops growing

Instead of "better" cores, we have "more" cores (with wider SIMD)

Takeaways:

1. **[Architecture-aware programming]** will become increasingly important in the future since processors and memory are stopping getting faster.

2. Processors are more capable than you thought.
[Parallelism: Computation is cheap]

3. Memory bandwidth is a more scarce resource nowadays.
[Locality: Communication is expensive]

Takeaways:

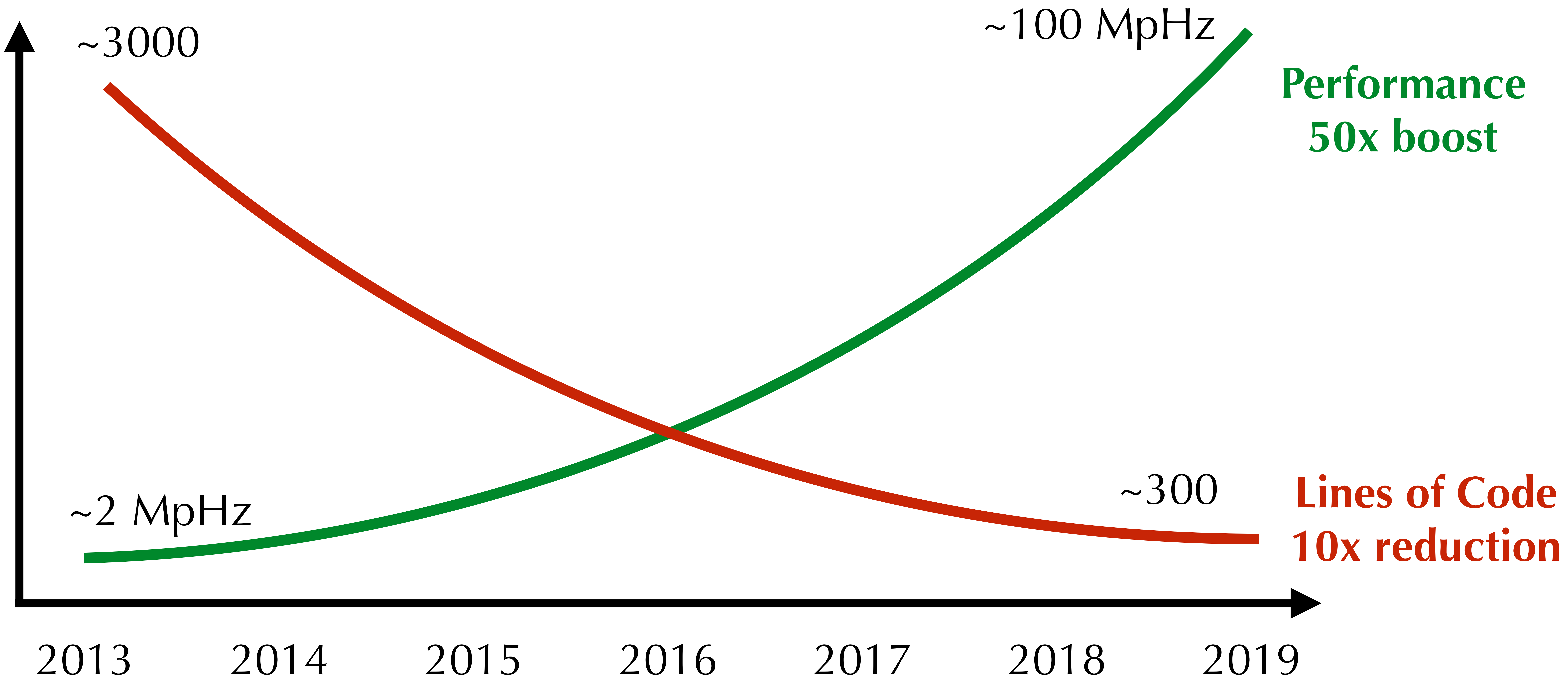
1. **[Architecture-aware programming]** will become increasingly important in the future since processors and memory are stopping getting faster.

2. Processors are more capable than you thought.
[Parallelism: Computation is cheap]

3. Memory bandwidth is a more scarce resource nowadays.
[Locality: Communication is expensive]

Same rules apply to GPUs

The "Moore's Law" of MPM



MpHz: Million particle x (Explicit Substep) Hertz

The Design Space for High Performance

- **Data structures**
  - For particles:
    - array of structure (AOS)/ structure of arrays (SOA), sorting/reordering
  - For grid:
    - Dense/sparse (VDB/SPGrid), dynamic/fixed hierarchy, leaf block data layout, Z-indexing…
- **Parallelization:**
  - what does each vector lane/thread/core/warp/block/streaming multiprocessor do?
  - How to avoid data race and cacheline sharing?

Organizing Particle Data

AOS v.s. SOA Layout

- **Array of Structures (AOS)**

```
struct Particle {float x, y, z};
Particle particles[8192];
```

- **Structure of Arrays (SOA)**

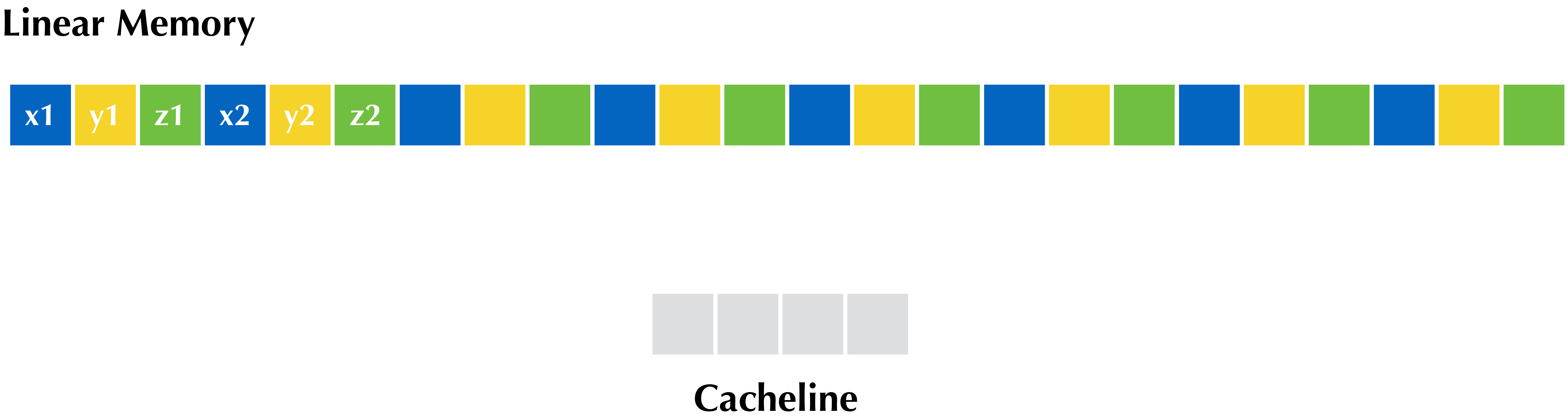
```
struct particles {
   float particle_x[8192];
   float particle_y[8192];
   float particle_z[8192];
};
```

Array of Structures

Linear Memory

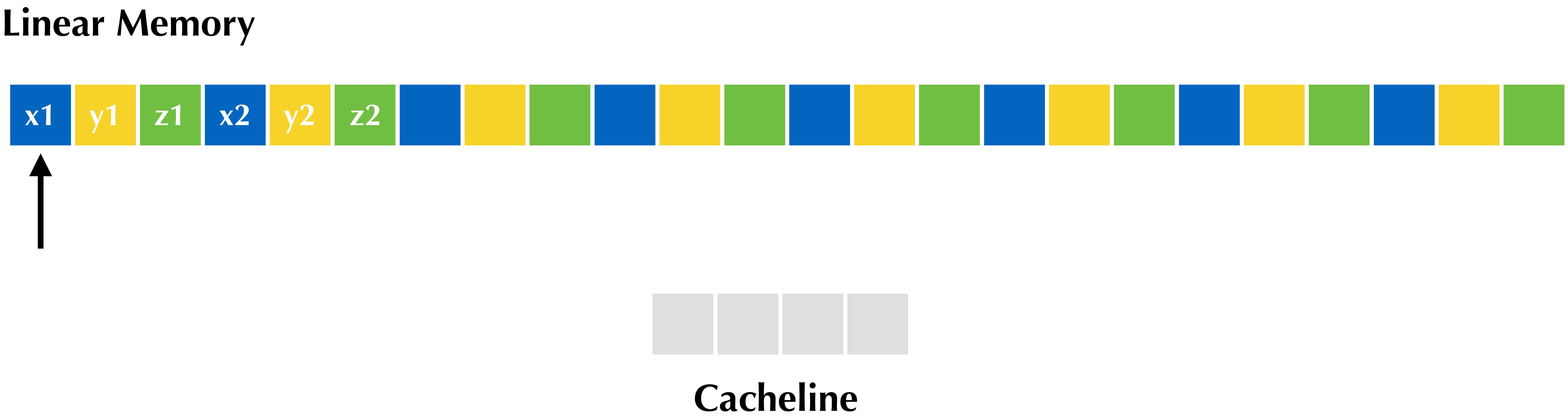
Each particle's fields are continuous in memory

Array of Structures: Sequential Access



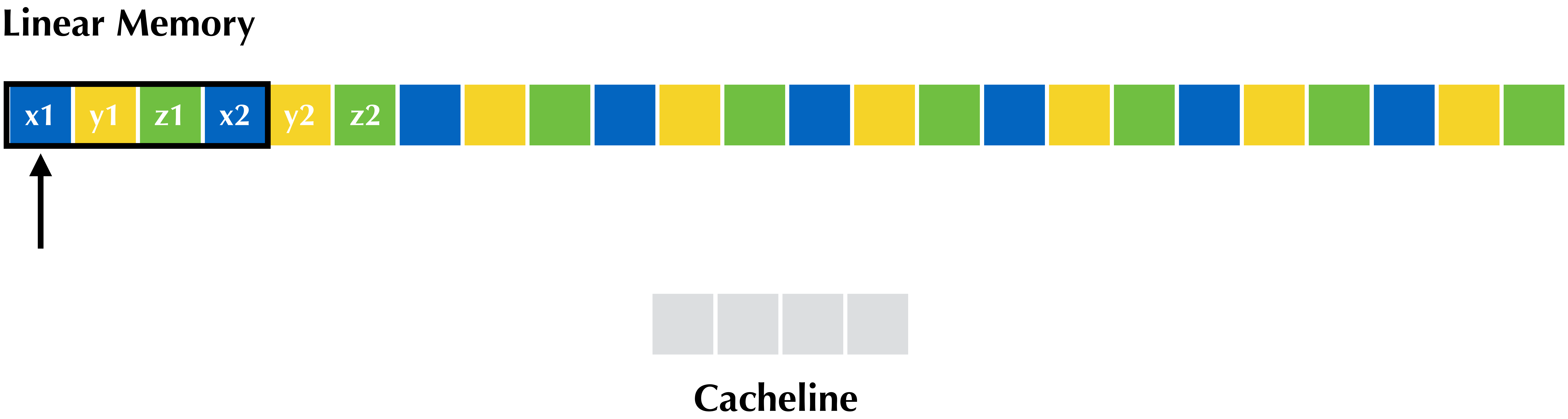
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



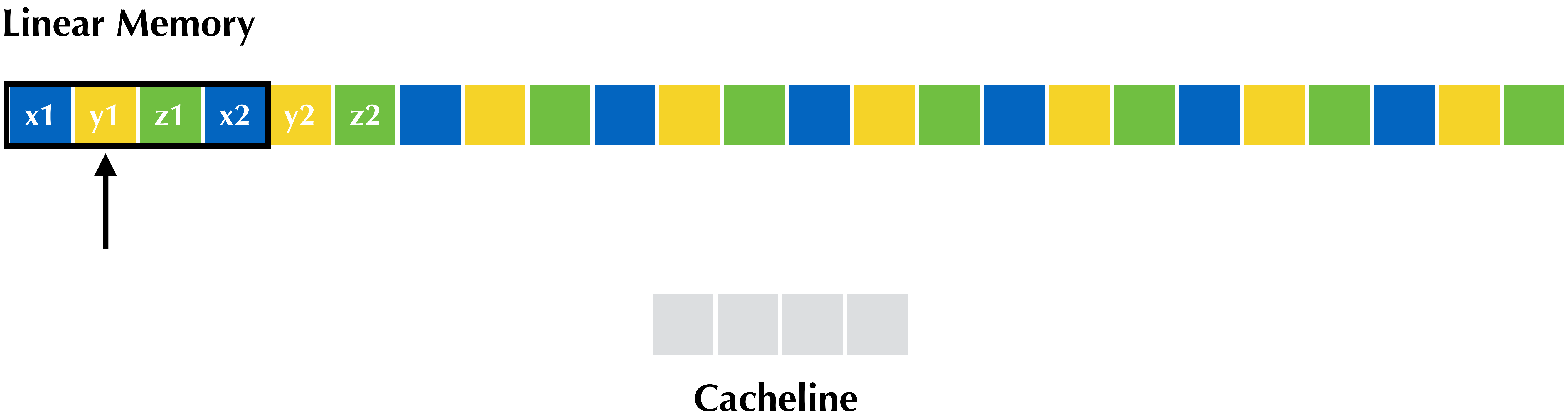
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



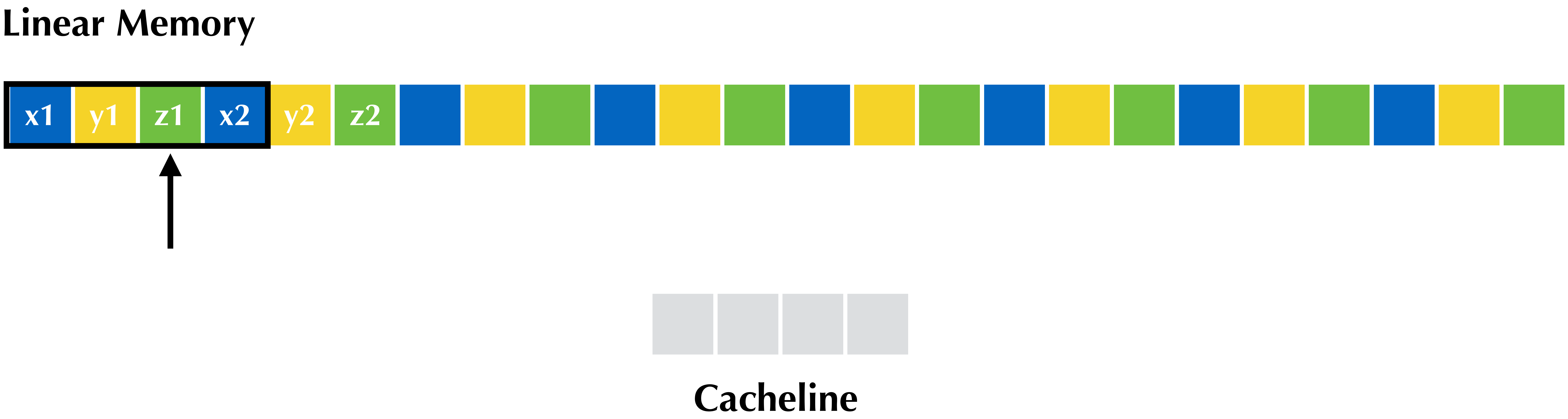
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



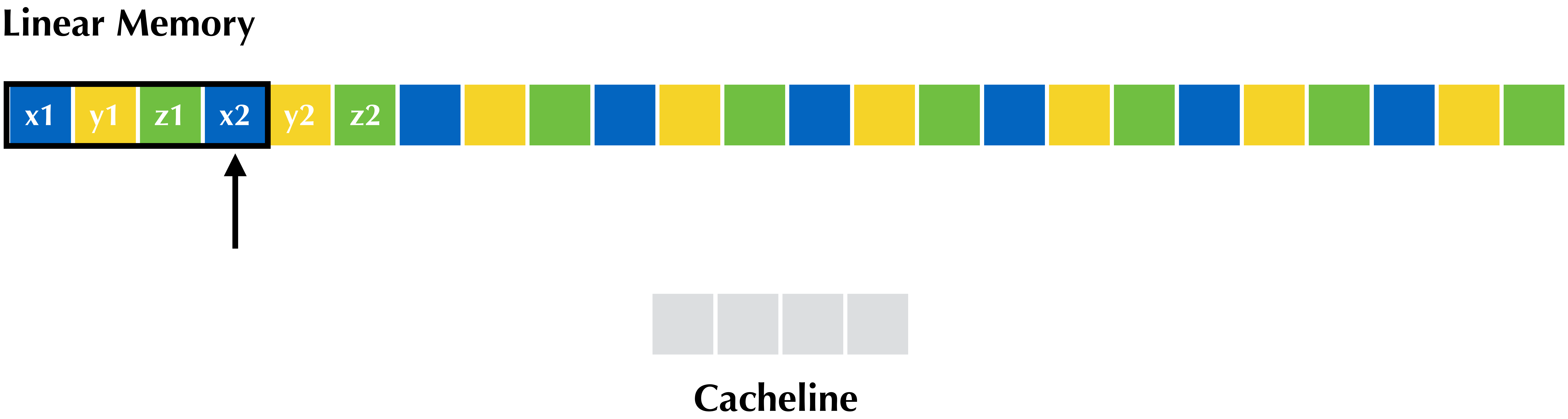
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



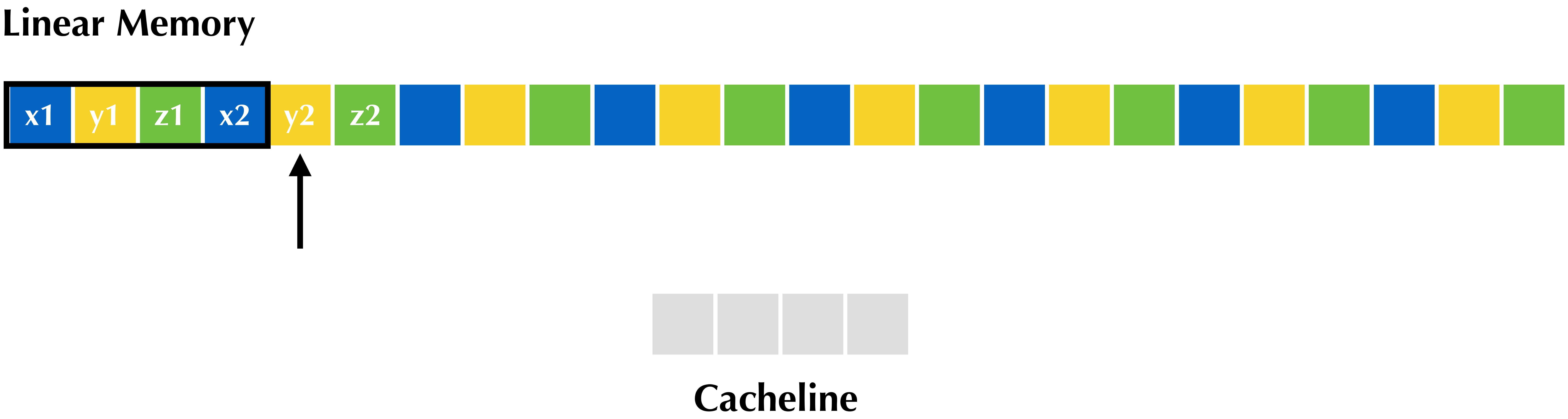
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



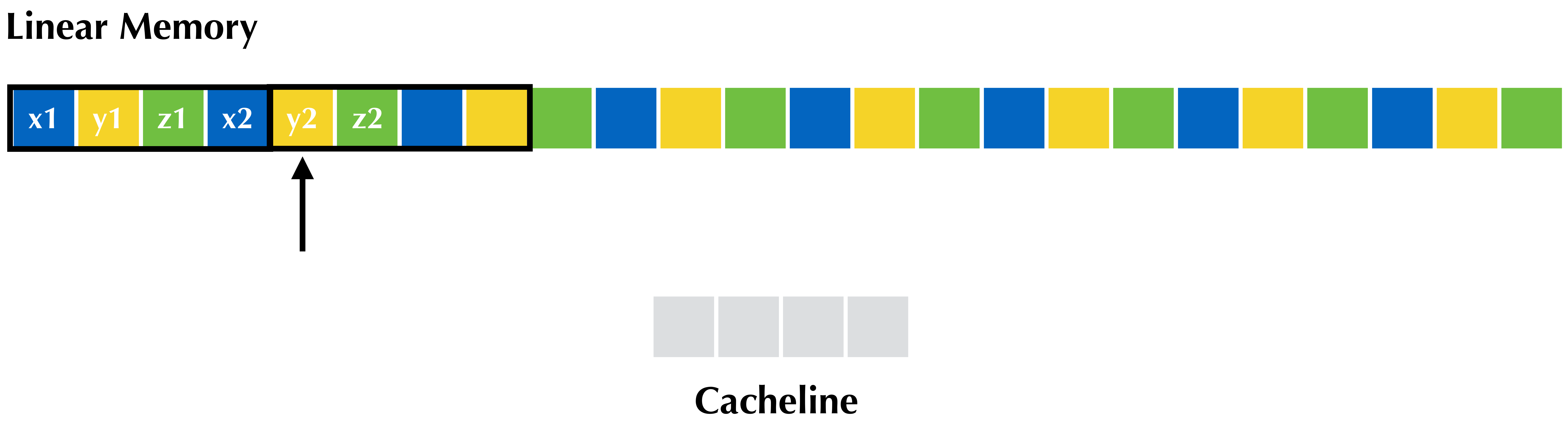
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



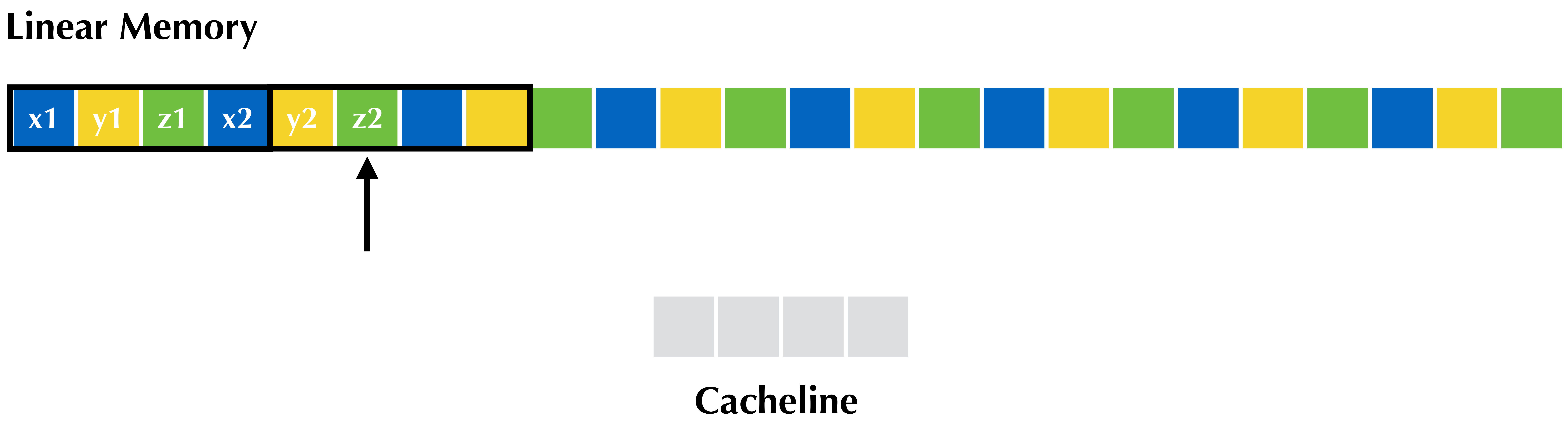
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



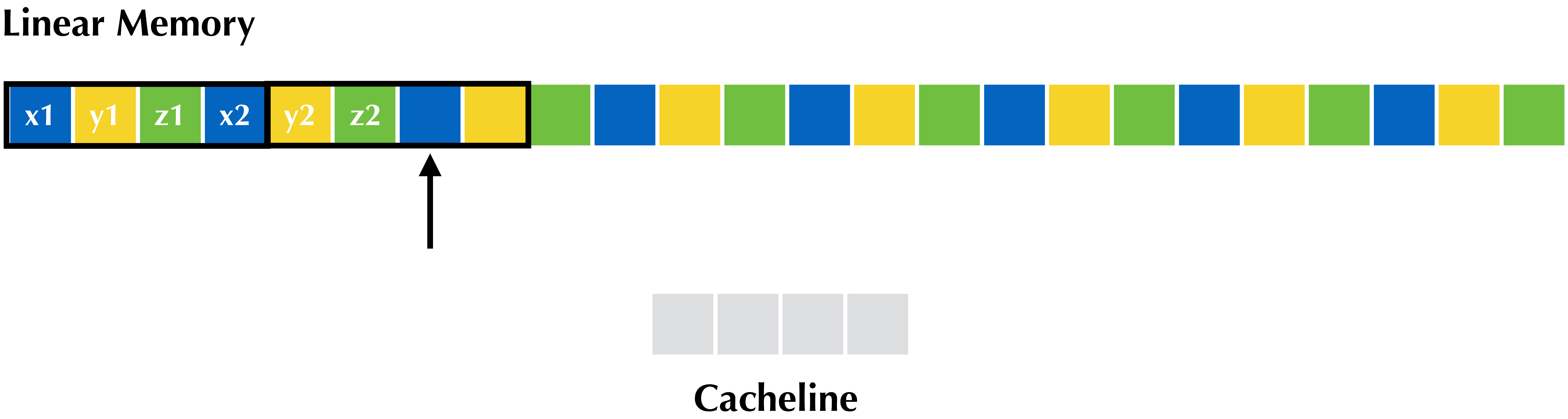
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



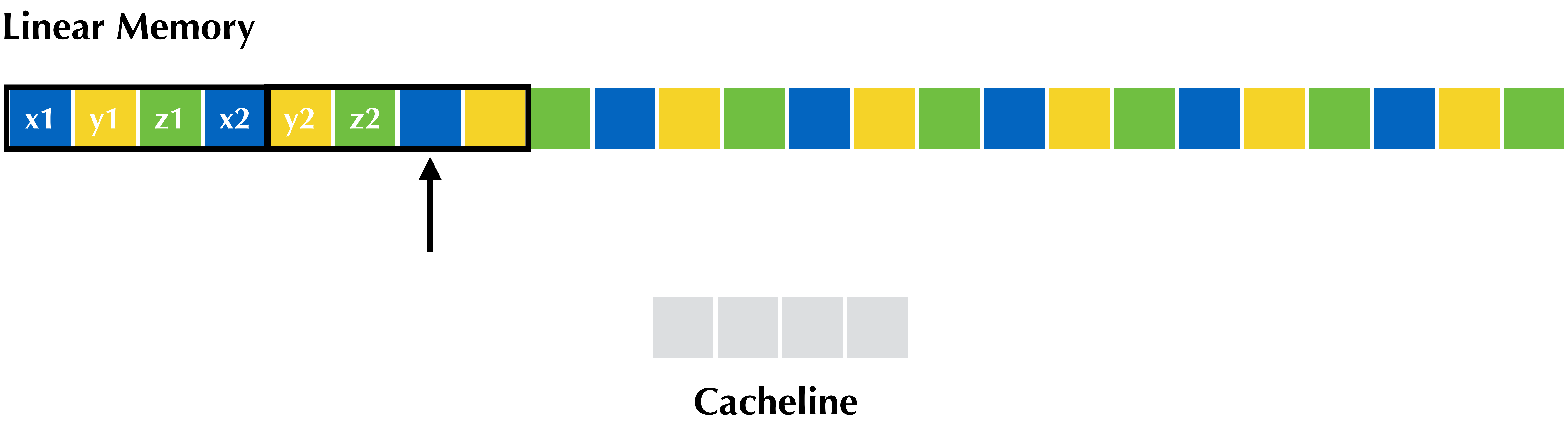
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Array of Structures: Sequential Access



Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

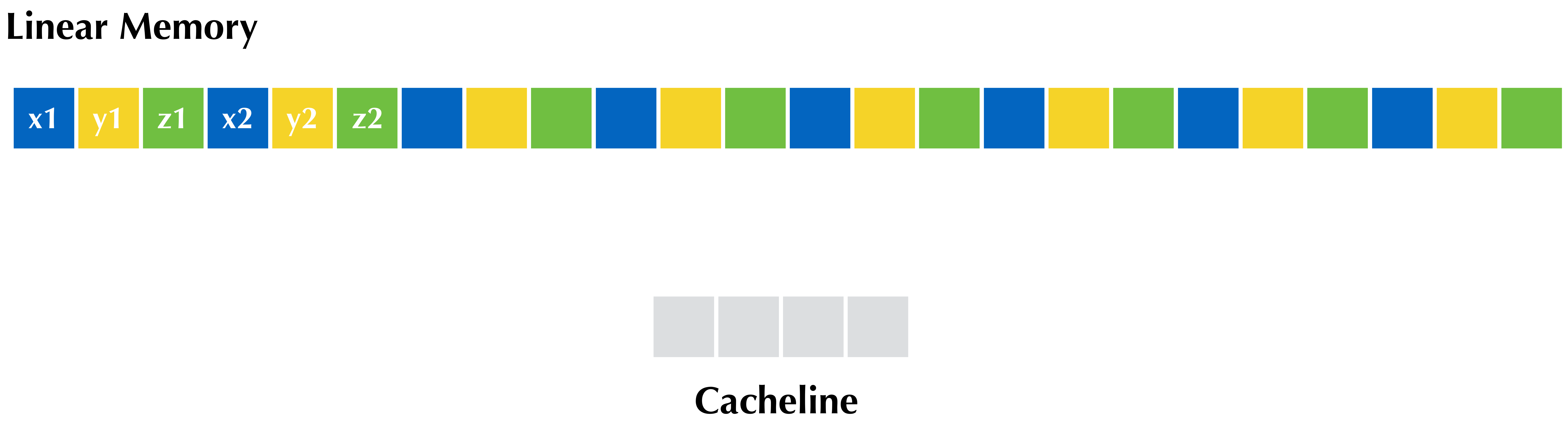
Array of Structures: Sequential Access



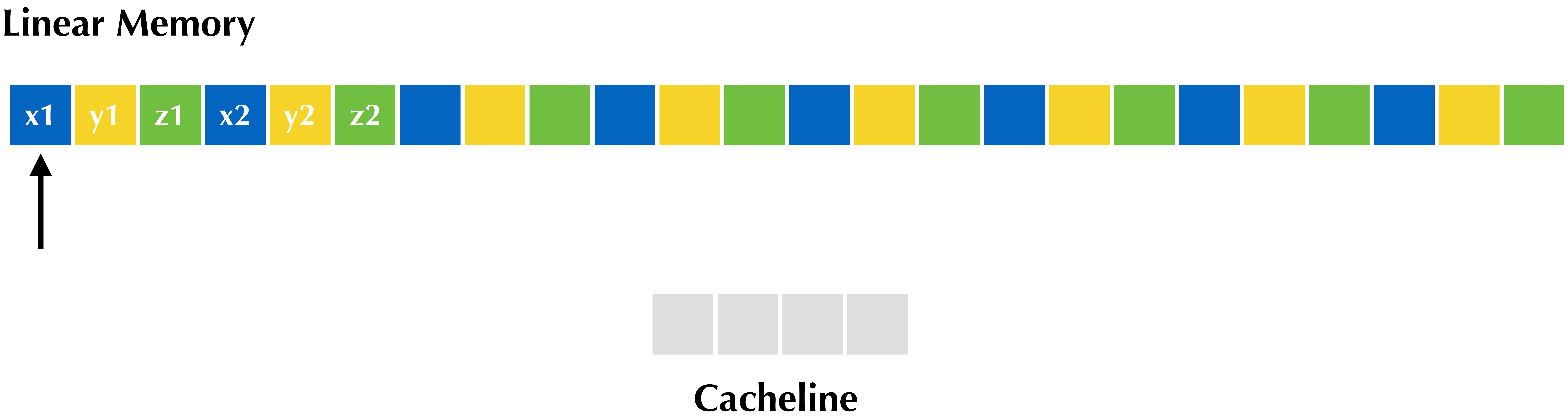
Cacheline size: x86_64: 64B; NVIDIA GPU: 128B

Cacheline Utilization: 100%

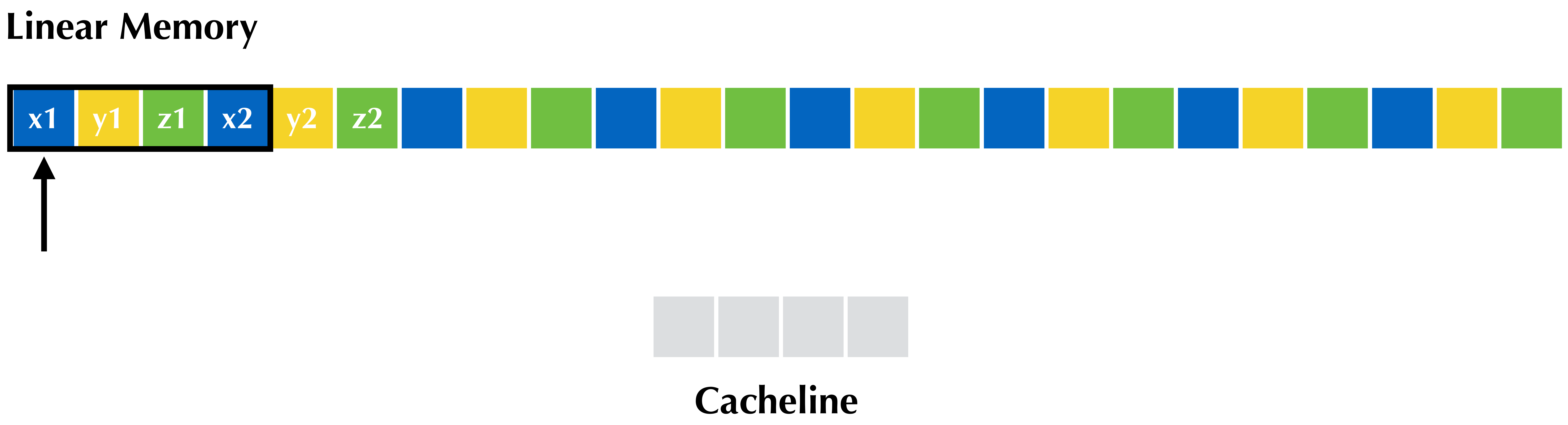
Array of Structures: Random Access



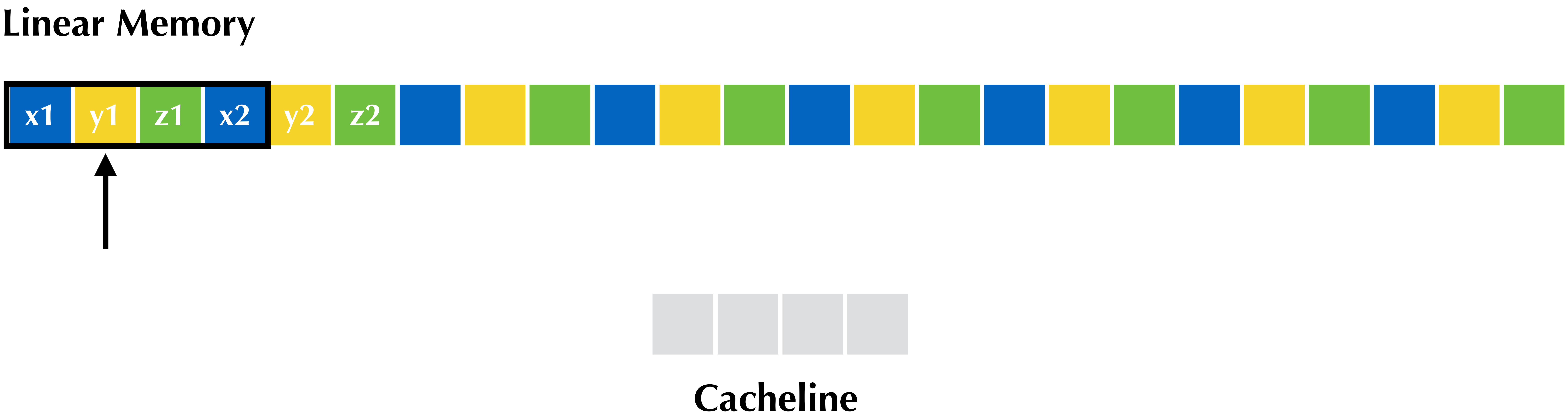
Array of Structures: Random Access



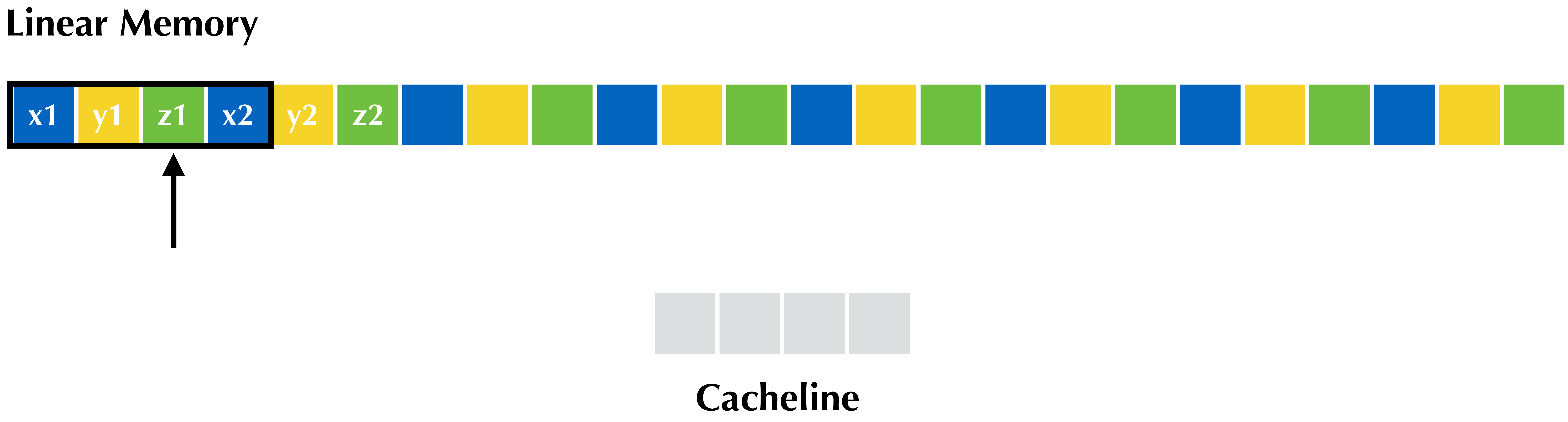
Array of Structures: Random Access



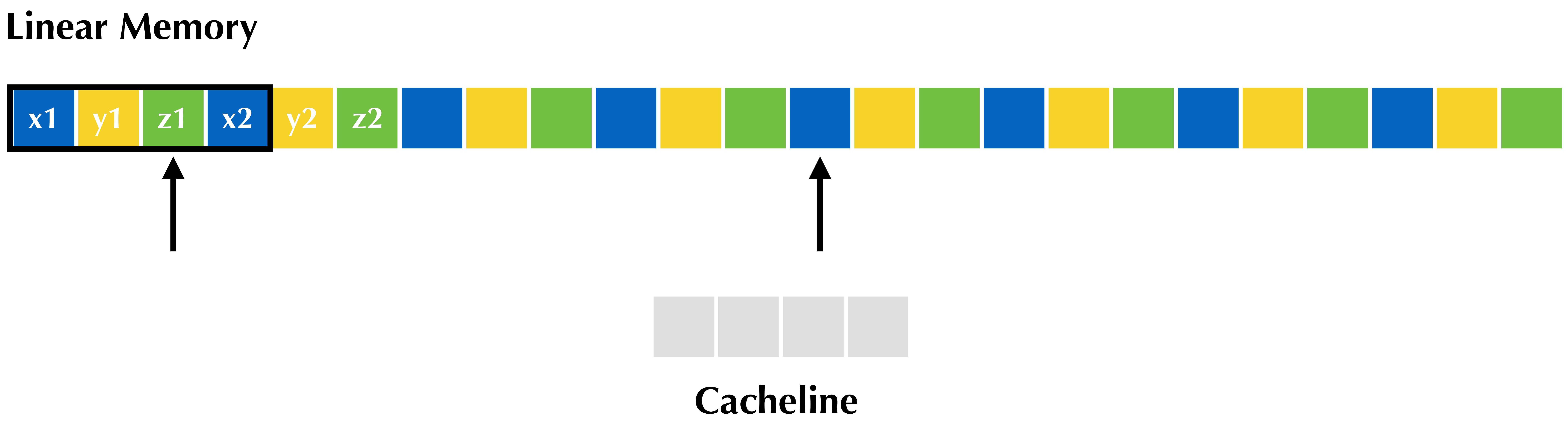
Array of Structures: Random Access



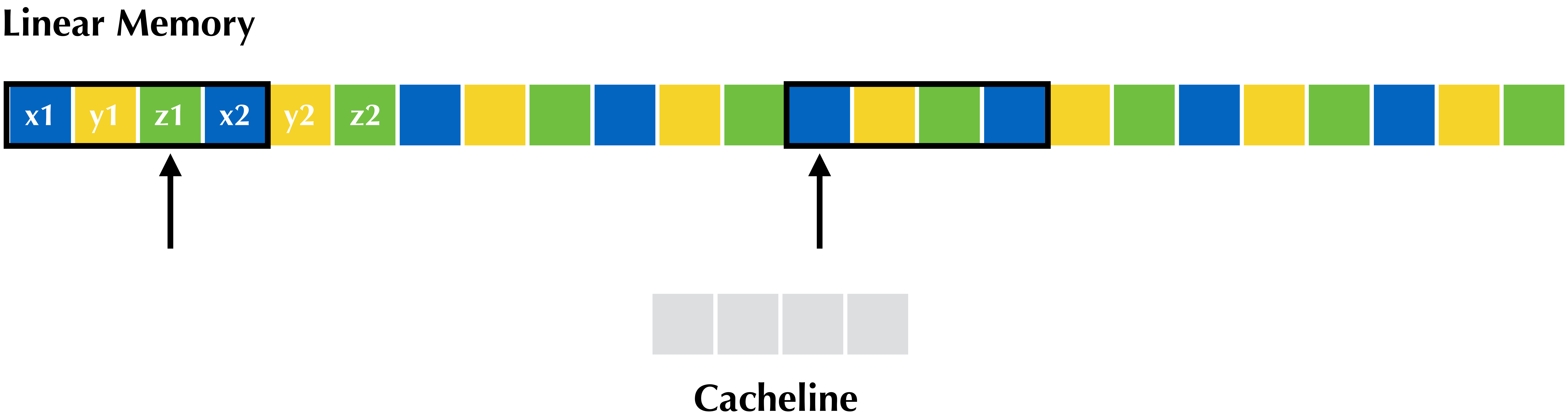
Array of Structures: Random Access



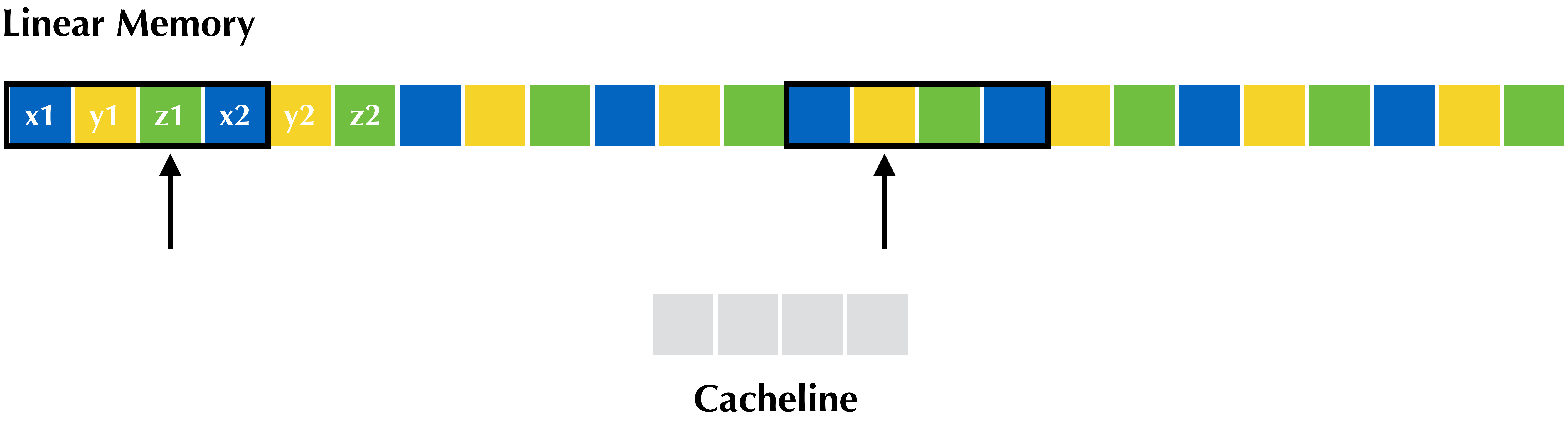
Array of Structures: Random Access



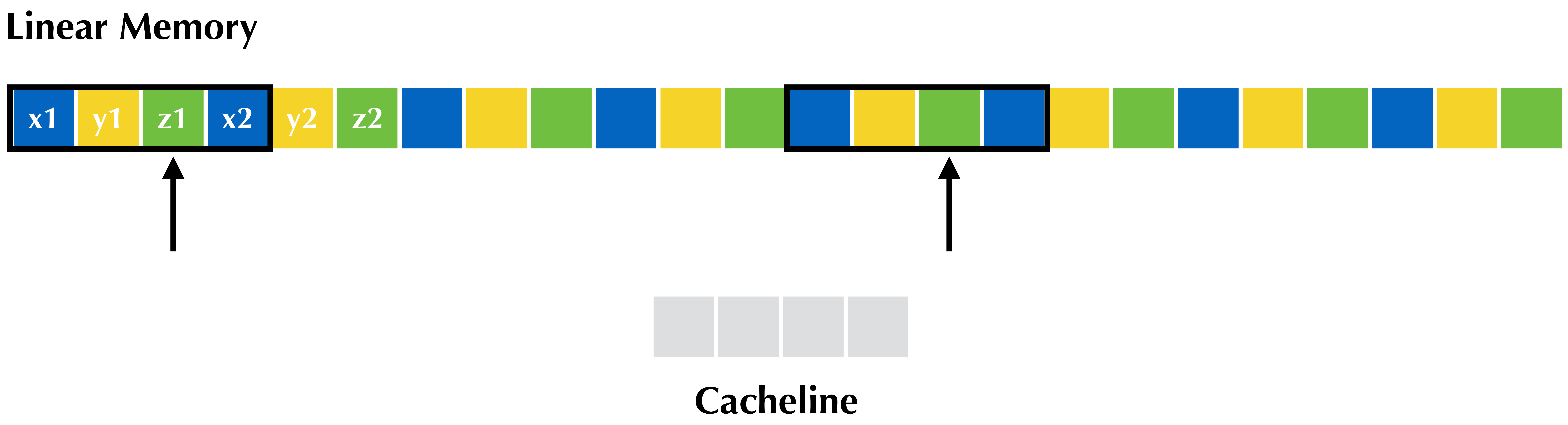
Array of Structures: Random Access



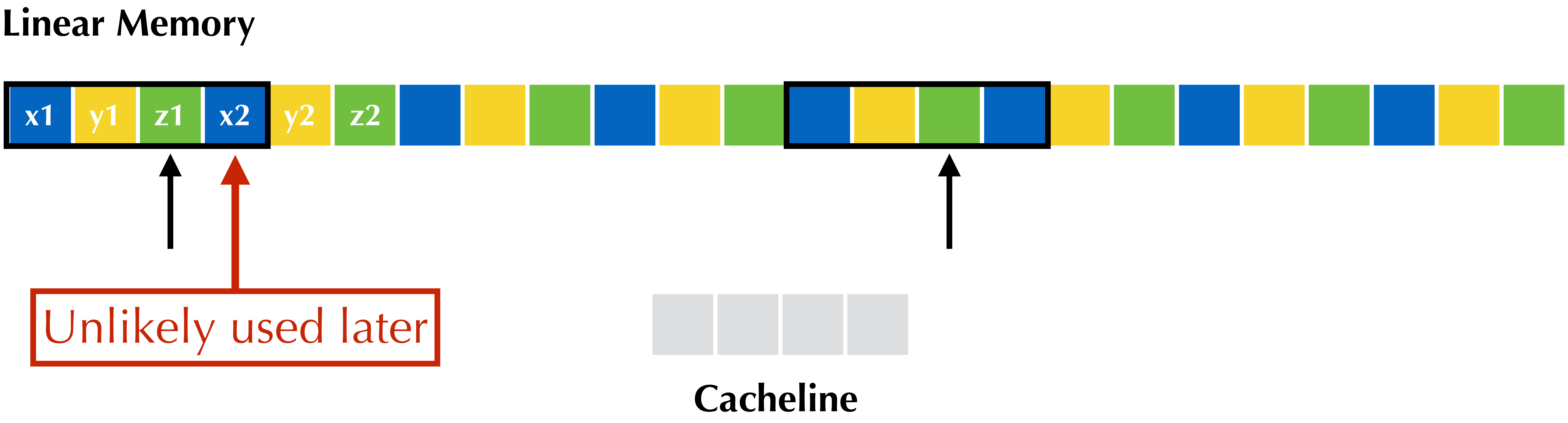
Array of Structures: Random Access



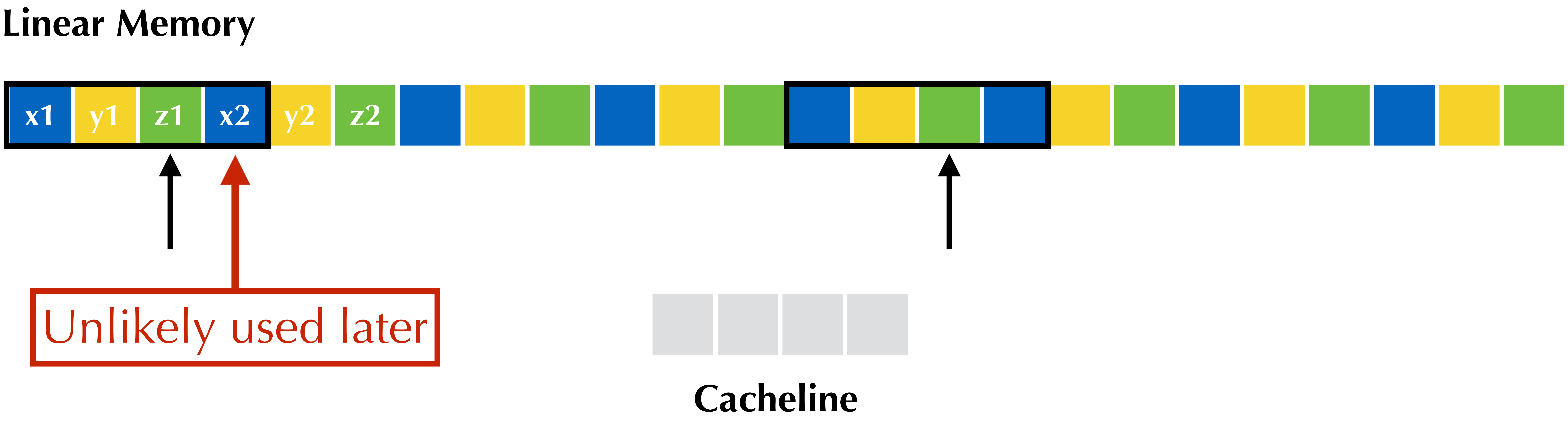
Array of Structures: Random Access



Array of Structures: Random Access



Array of Structures: Random Access



Cacheline Utilization: 75%

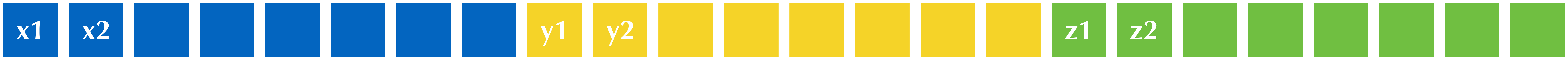
Array of Structures

Linear Memory

Each particle's fields are continuous in memory

Structure of Arrays

Linear Memory



Each field's particle instances are continuous in memory

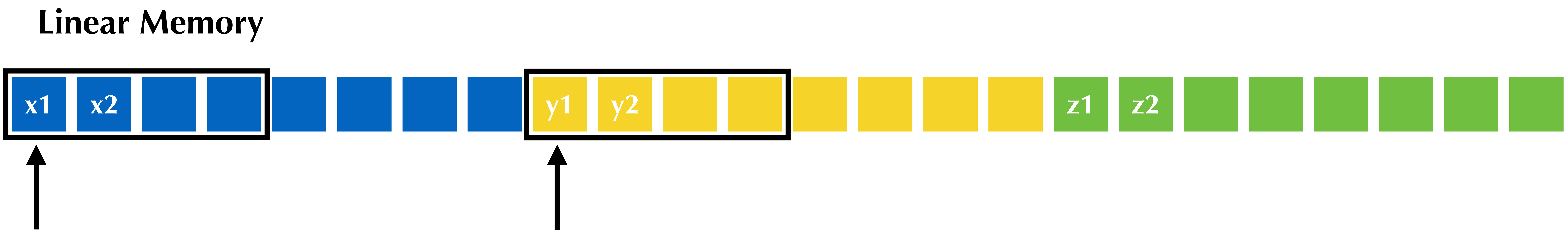
Structure of Arrays: Sequential Access

Structure of Arrays: Sequential Access

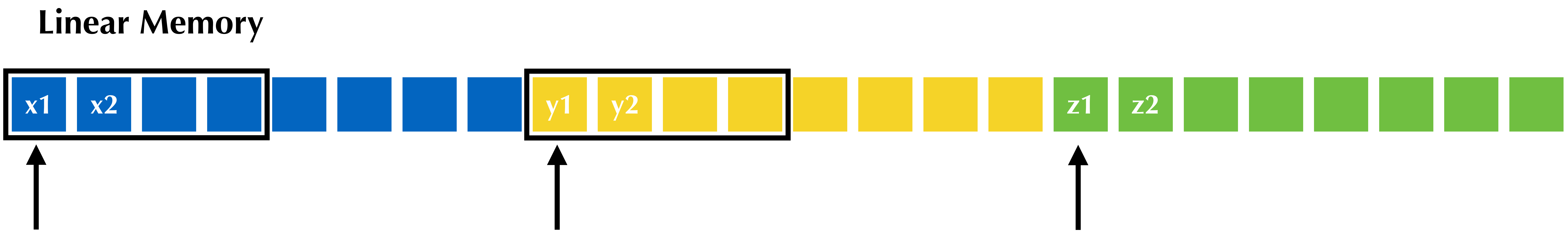
Structure of Arrays: Sequential Access

Structure of Arrays: Sequential Access

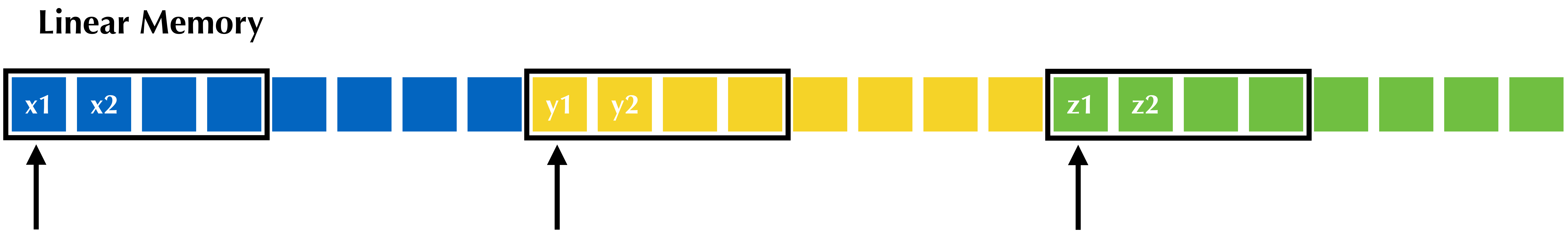
Structure of Arrays: Sequential Access



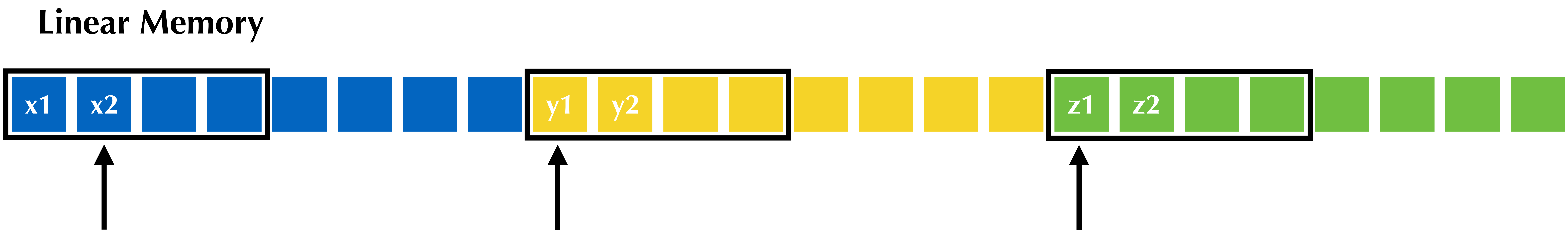
Structure of Arrays: Sequential Access



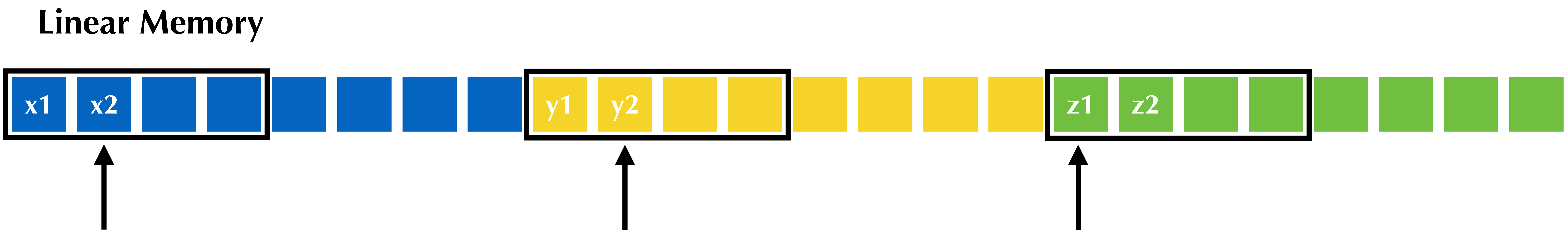
Structure of Arrays: Sequential Access



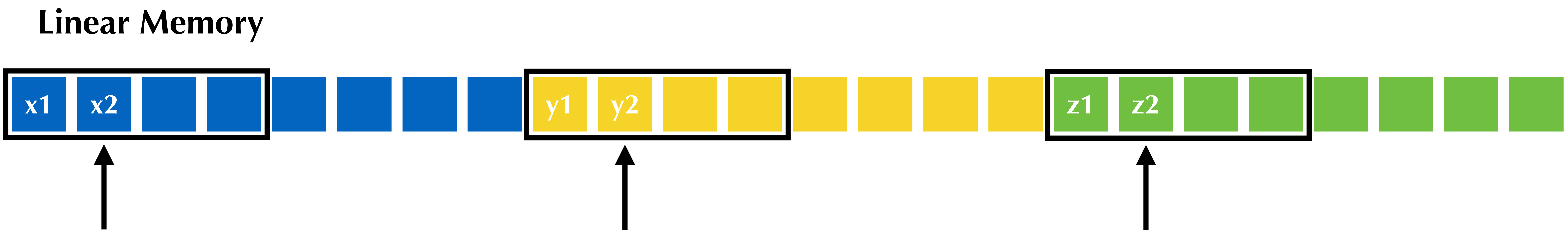
Structure of Arrays: Sequential Access



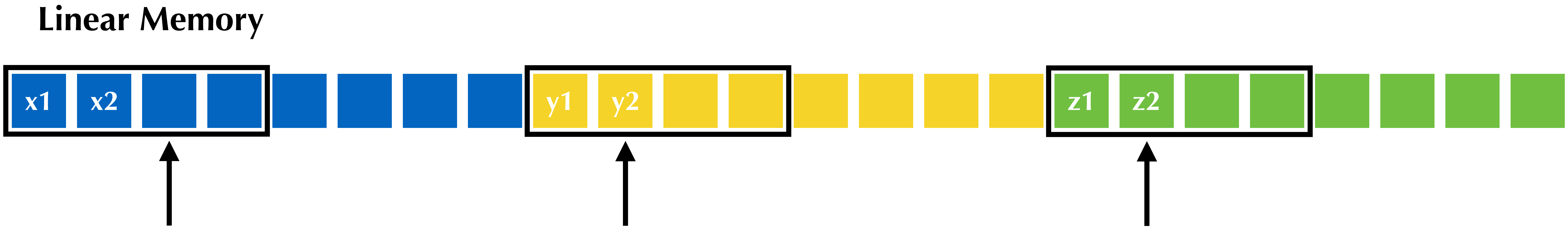
Structure of Arrays: Sequential Access



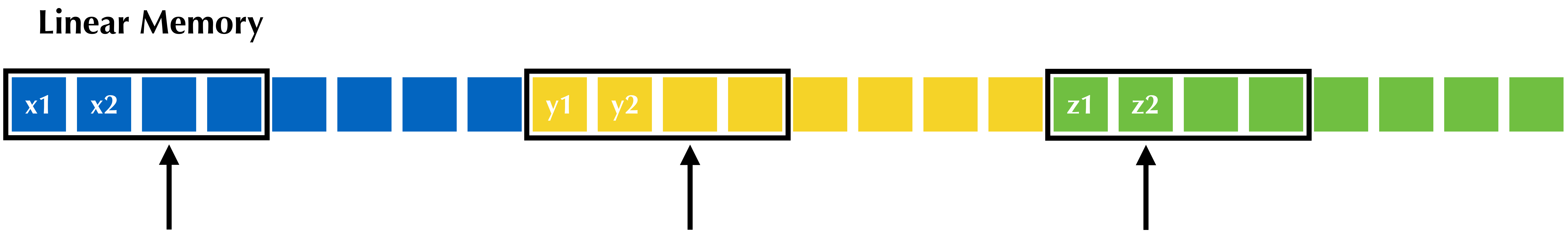
Structure of Arrays: Sequential Access



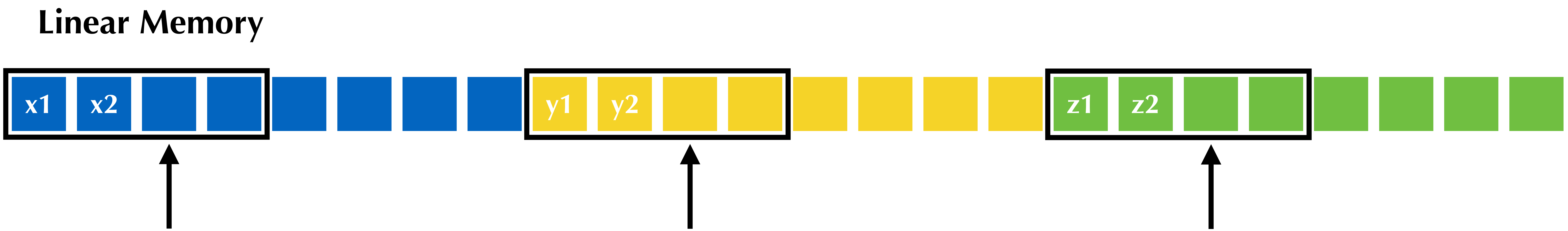
Structure of Arrays: Sequential Access



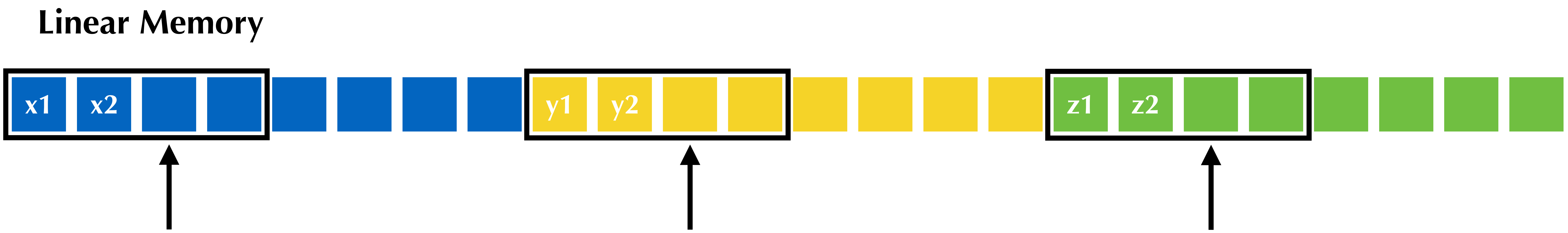
Structure of Arrays: Sequential Access



Structure of Arrays: Sequential Access



Structure of Arrays: Sequential Access



Cacheline Utilization: 100%

Structure of Arrays: Random Access

Linear Memory

Assuming cache size=3

Structure of Arrays: Random Access

Linear Memory

Assuming cache size=3

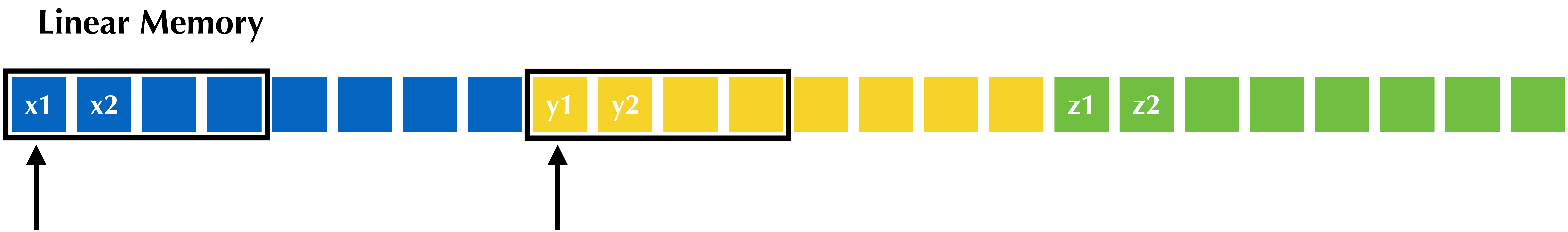
Structure of Arrays: Random Access

Assuming cache size=3

Structure of Arrays: Random Access

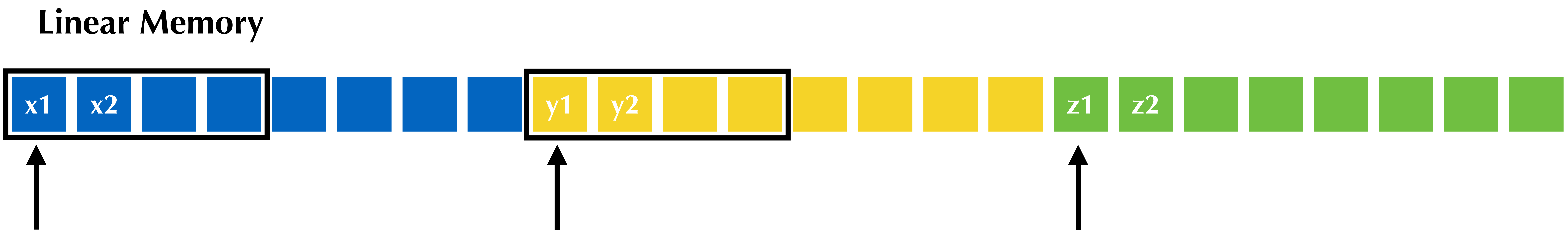
Assuming cache size=3

Structure of Arrays: Random Access



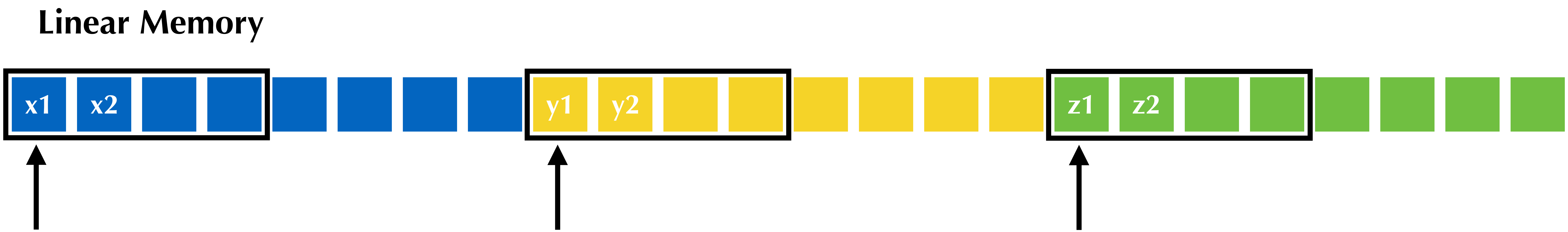
Assuming cache size=3

Structure of Arrays: Random Access



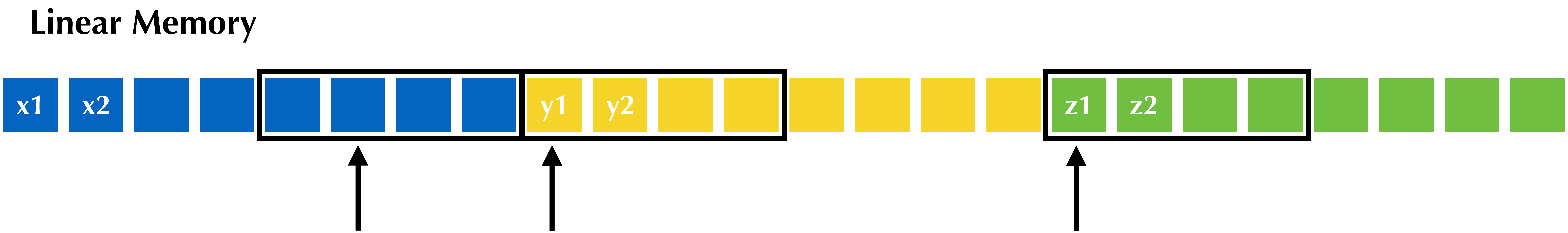
Assuming cache size=3

Structure of Arrays: Random Access



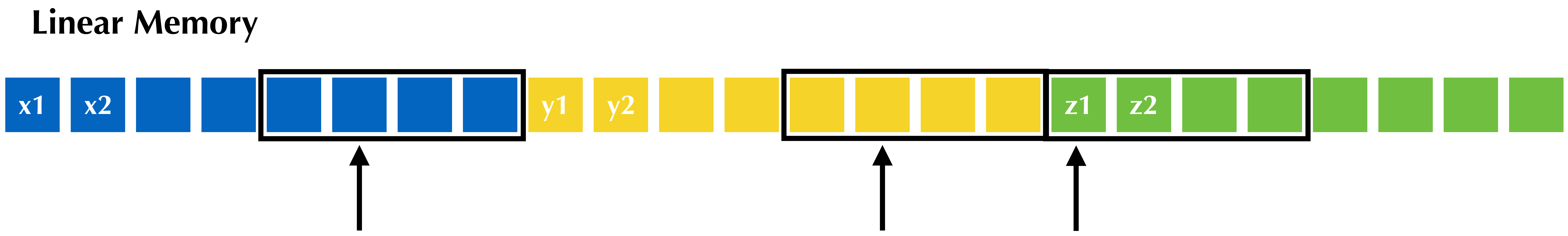
Assuming cache size=3

Structure of Arrays: Random Access



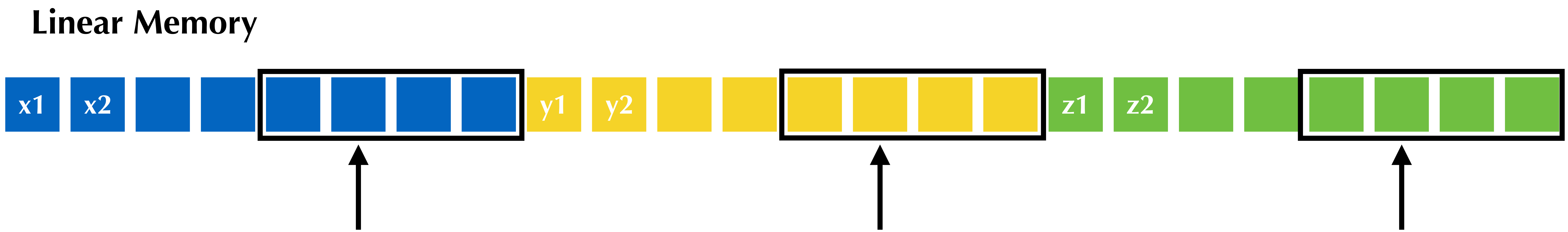
Assuming cache size=3

Structure of Arrays: Random Access



Assuming cache size=3

Structure of Arrays: Random Access



Assuming cache size=3

Structure of Arrays: Random Access

Linear Memory

Assuming cache size=3

Cacheline Utilization: 25%

Data Structures for MPM Particles

- **SOA: very efficient when sorted**
  - Coalesced access on GPUs
  - Large number of streams (e.g. 20): may invalidate prefetchers on CPU
    - not a problem for GPU - GPUs are designed for streaming and have no prefetching
  - Random access: low cacheline utilization
- **AOS: efficient even unsorted**
  - Random access: much better than SOA but should still be avoided if possible (cache/TLB misses)
  - Sequential access: Slightly inferior than AOS
    - Vector lane shuffling on CPUs
    - Non-coalseced access on GPUs
  - No sorting needed

Real world 3D MPM: **x**(3 floats), **v**(3 floats), **F**(9 floats), **C**(9 floats)…

Particle Layout	Ordered	Randomly Shuffled
SOA	3.52ms	21.23 ms
AOS	3.15ms	4.28 ms

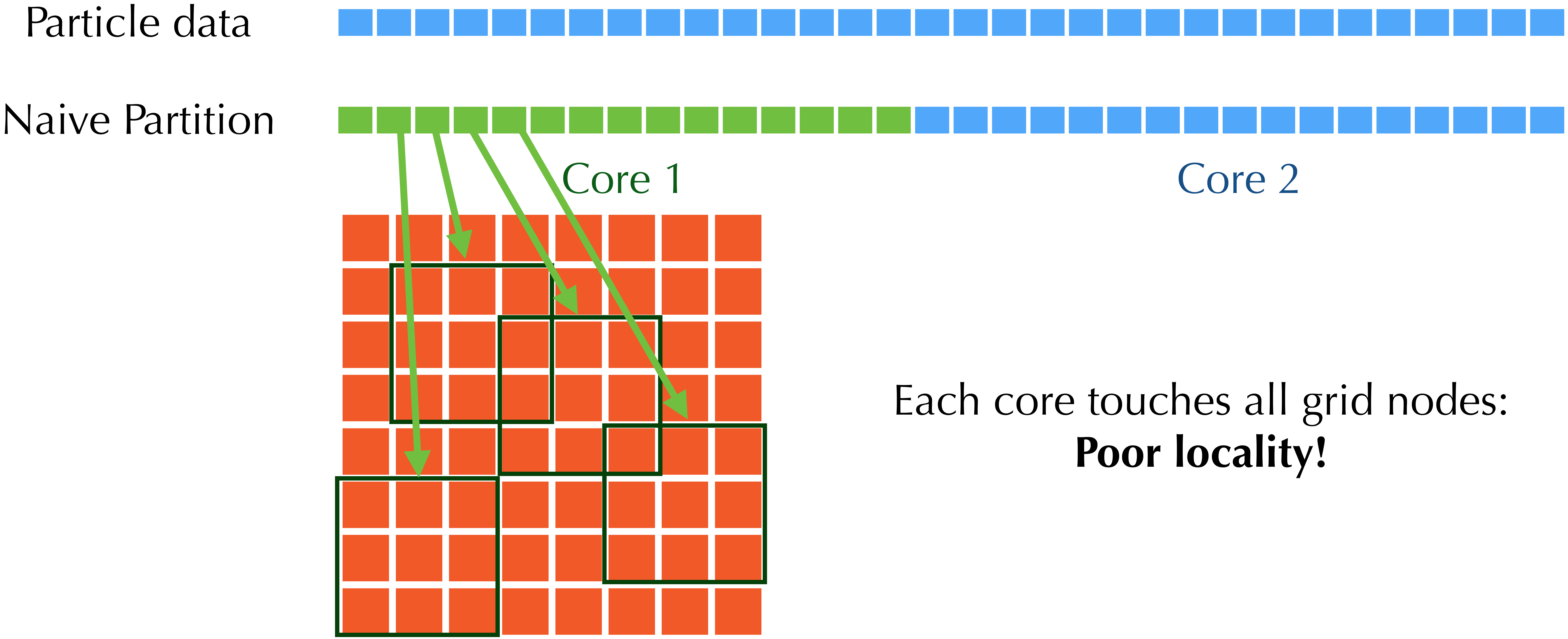
- ✦ **SOA 6x slower when random shuffled**
  - Periodic reodering is key to high performance for SOA
- ✦ **AOS is less sensitive to particle order**

Data Structures for MPM Grids

- **Multi-Level Hierarchical Sparse Grids**
  - e.g. VDB, SPGrid
- **Dense blocks (e.g. 4x4x4 grid nodes) organized in tree structures**
  - Grid nodes store velocity (momentum) and mass

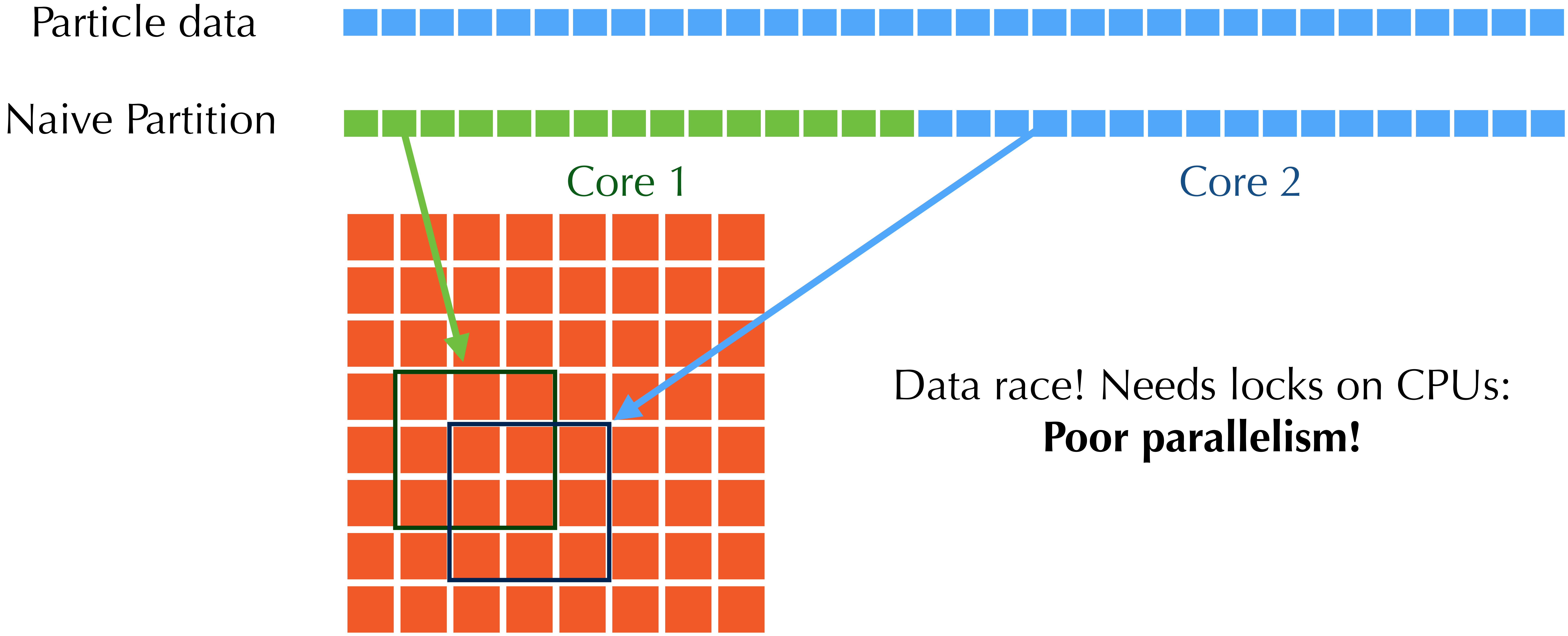
Organizing Grid Data & Parallelization based on Grid Blocks

Partition Particle for **Parallelization**



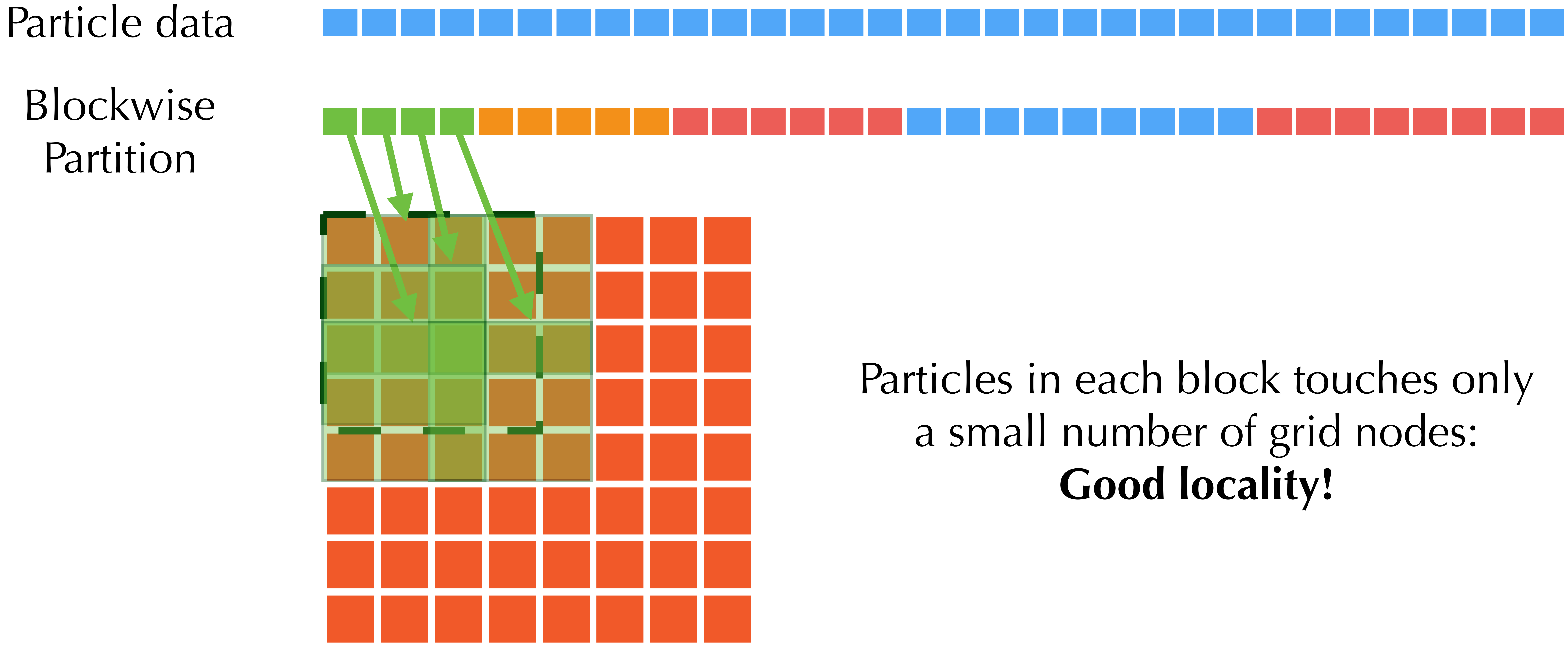
Each core touches all grid nodes:
Poor locality!

Partition Particle for **Parallelization**



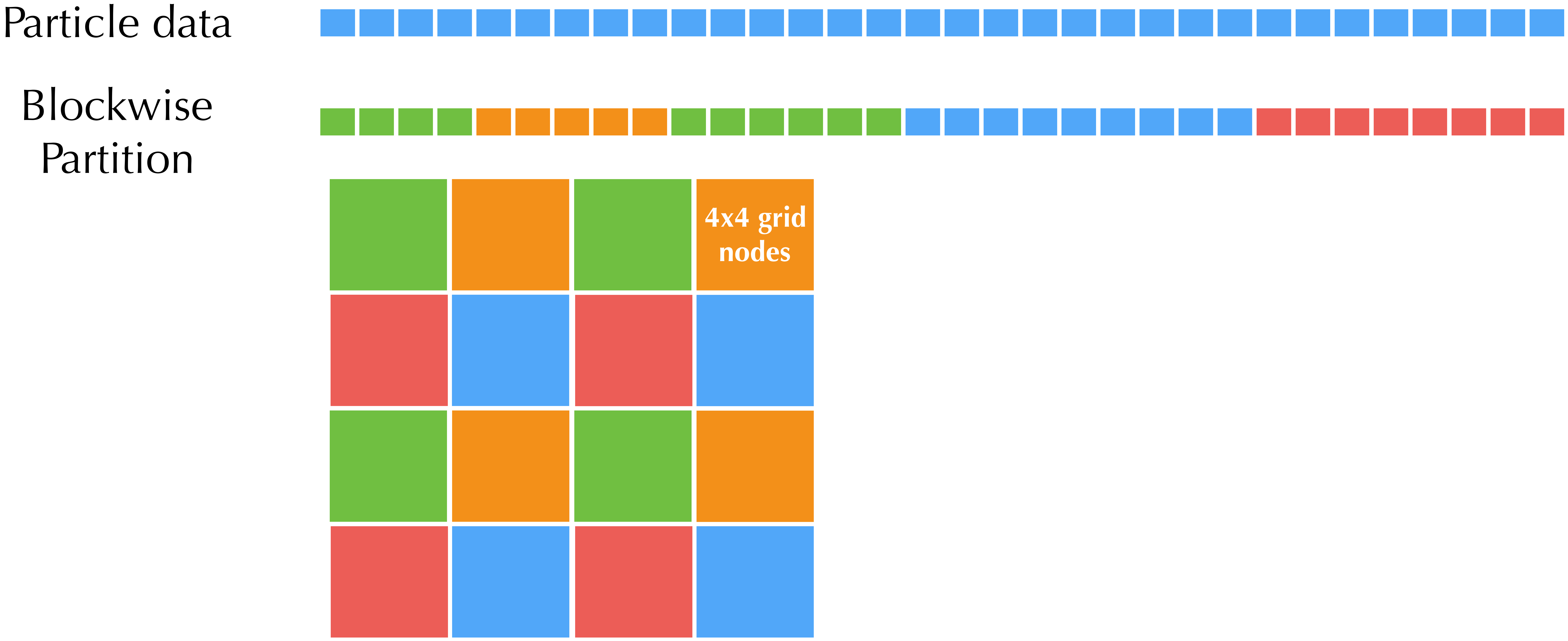
Data race! Needs locks on CPUs:
Poor parallelism!

Partition Particle for **Parallelization**

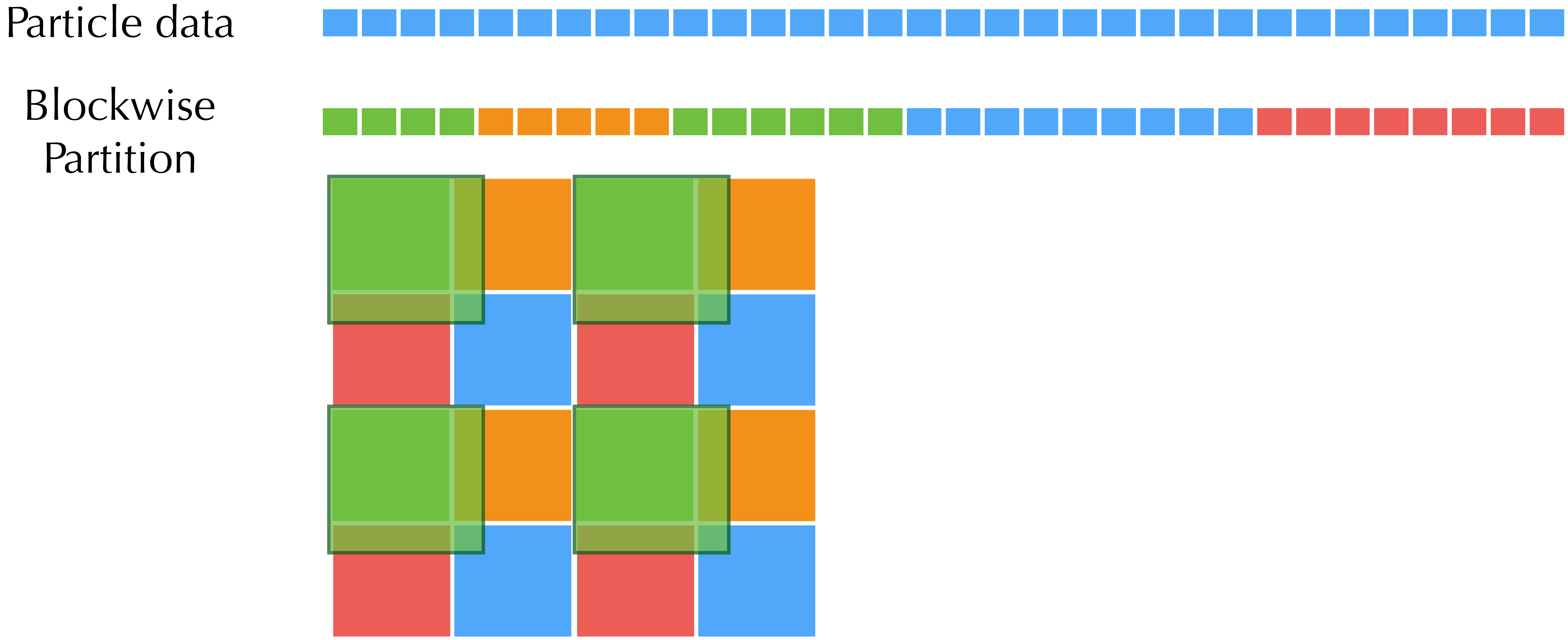


Particles in each block touches only a small number of grid nodes:
Good locality!

Partition Particle for **Parallelization**



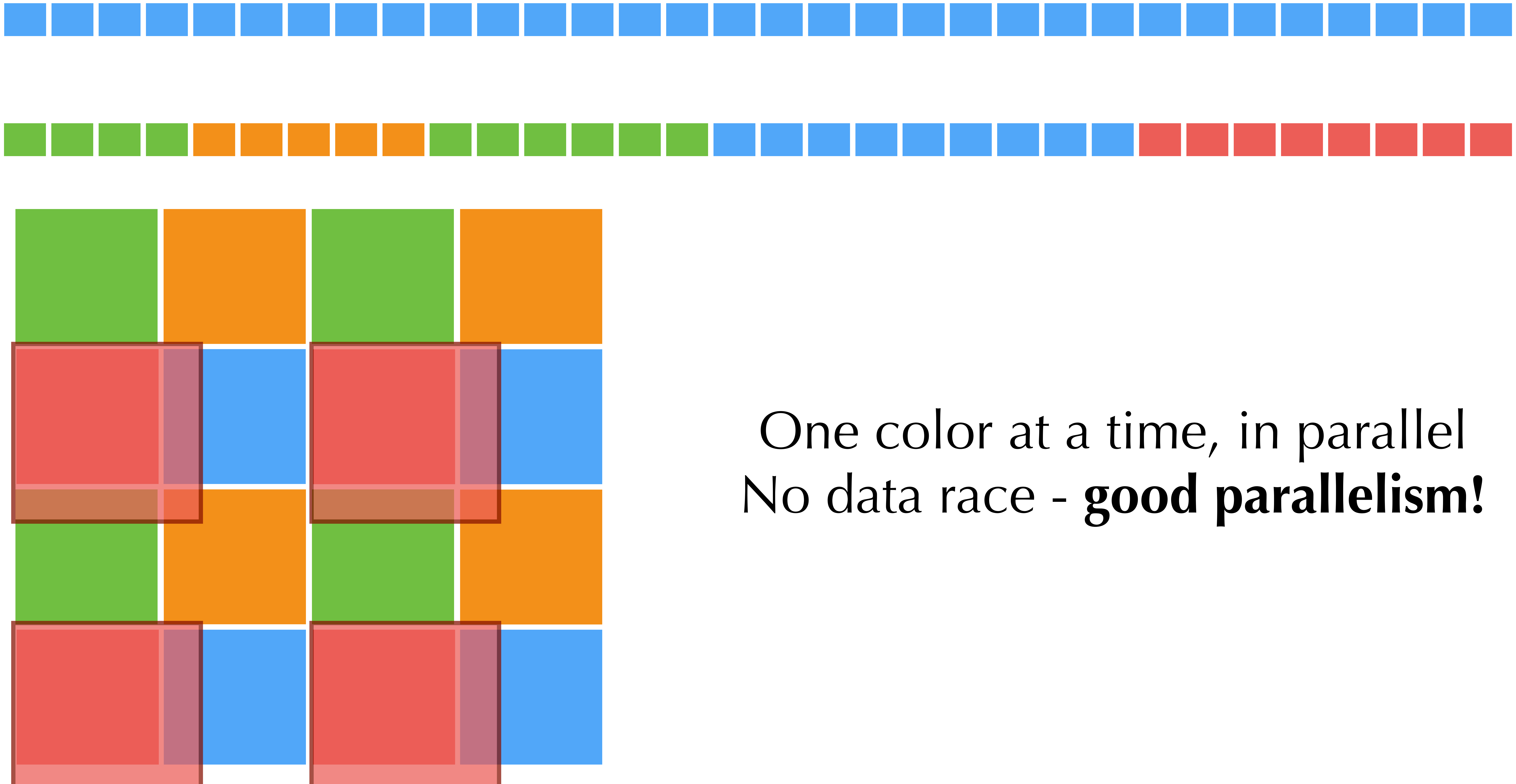
Partition Particle for **Parallelization**



Partition Particle for **Parallelization**

Particle data

Blockwise
Partition



One color at a time, in parallel
No data race - **good parallelism!**

Advertisement: ChainQueen 乾坤

- ***Differentiable*** **MLS-MPM Solver for Soft Robotics [Hu et al., ICRA 2019]**
- **Direct gradient-based optimization of robots and their controllers**
  - By the “Chain” rule, hence the name “ChainQueen”
  - Orders of magnitude faster than reinforcement learning!


Iteration 0


Iteration 0

On Hybrid Lagrangian-Eulerian Simulation Methods: Practical Notes and High-Performance Aspects

Yuanming Hu, Xinxin Zhang, **Ming Gao***, Chenfanfu Jiang

Tencent America & University of Pennsylvania

On Hybrid Lagrangian-Eulerian Simulation Methods: Practical Notes and High-Performance Aspects

Yuanming Hu, Xinxin Zhang, **Ming Gao***, Chenfanfu Jiang

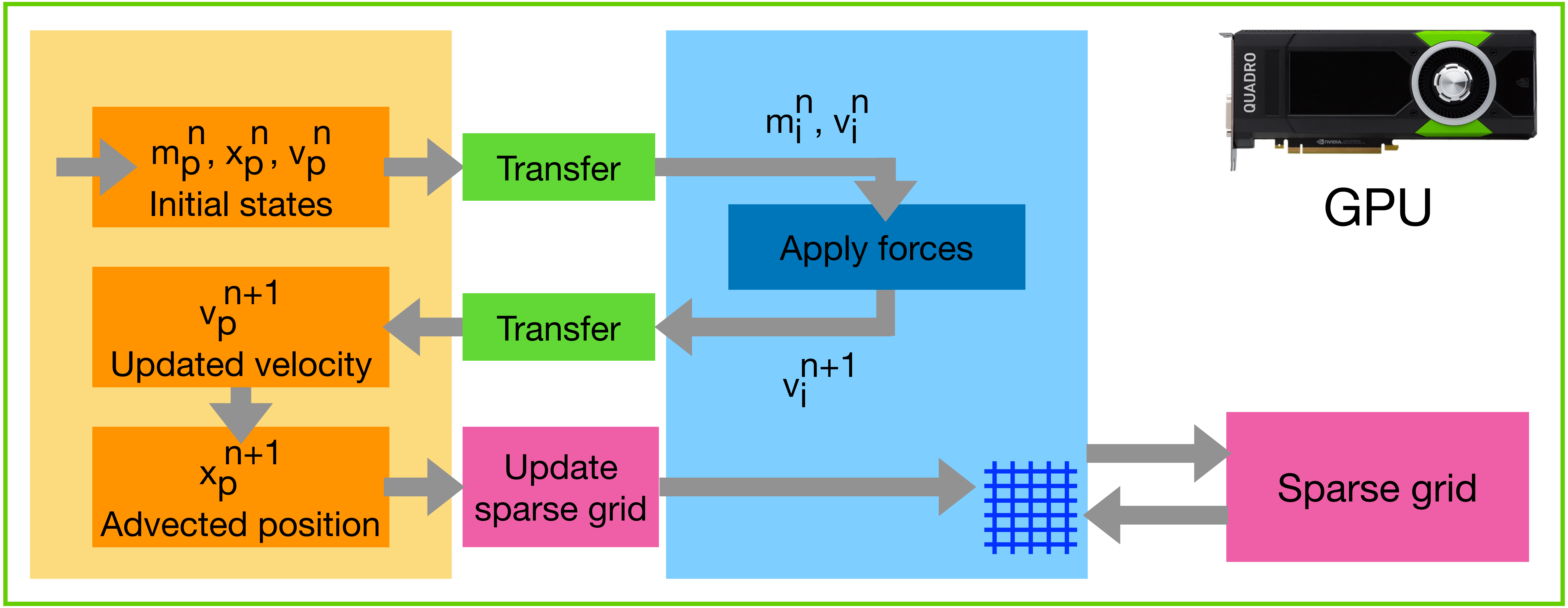
Tencent America & University of Pennsylvania

On Hybrid Lagrangian-Eulerian Simulation Methods: Practical Notes and High-Performance Aspects

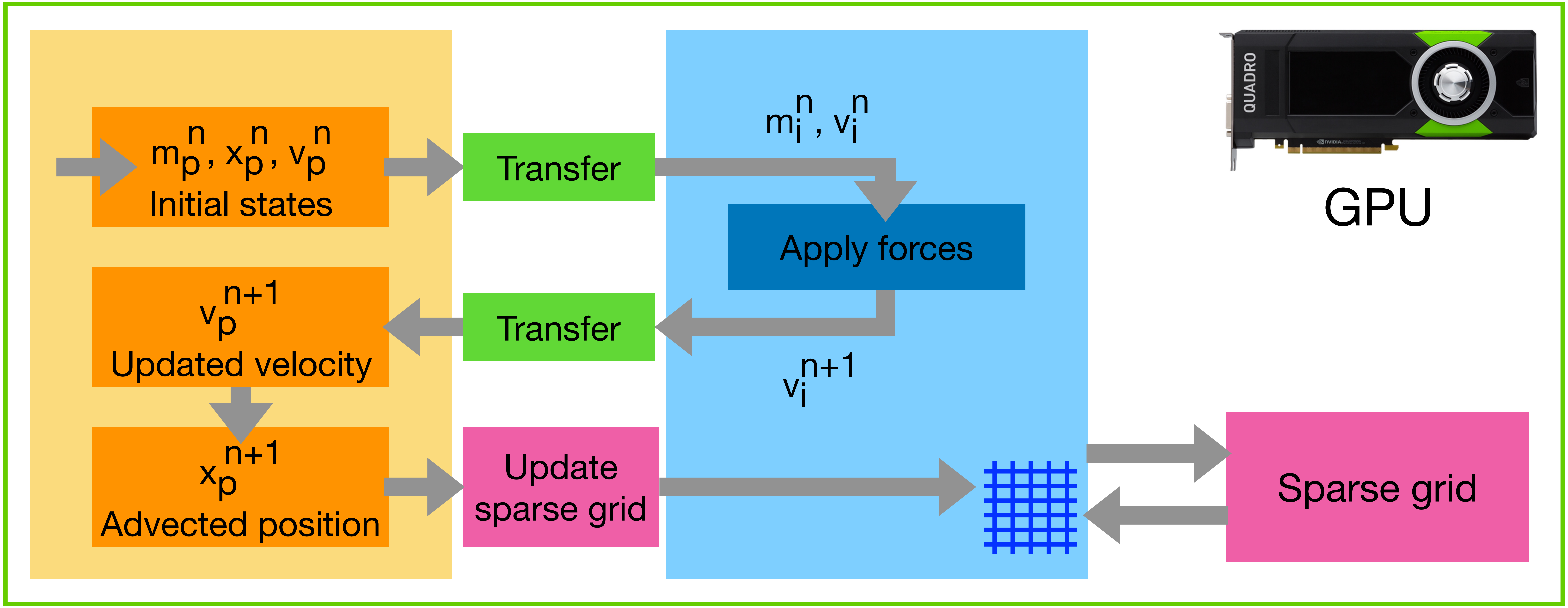
Yuanming Hu, Xinxin Zhang, **Ming Gao***, Chenfanfu Jiang

Tencent America & University of Pennsylvania

GPU MPM - explicit time integration



GPU MPM - explicit time integration



Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pipeline

Particles

Grid

Communications

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pure particle operations

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Compute dt

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Compute dt

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

1. CFL

Compute dt

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

1. CFL
2. Max dt

Update particle properties

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Update particle properties

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Update particle properties

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

SVD computation

Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations

A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis

SVD computation

Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations

A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis

0.37 ns per
3x3 matrix

SVD computation

Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations

A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis

0.37 ns per
3x3 matrix

$$U \Sigma V^T$$

SVD computation

Computing the Singular Value Decomposition of 3x3 matrices with minimal branching and elementary floating point operations

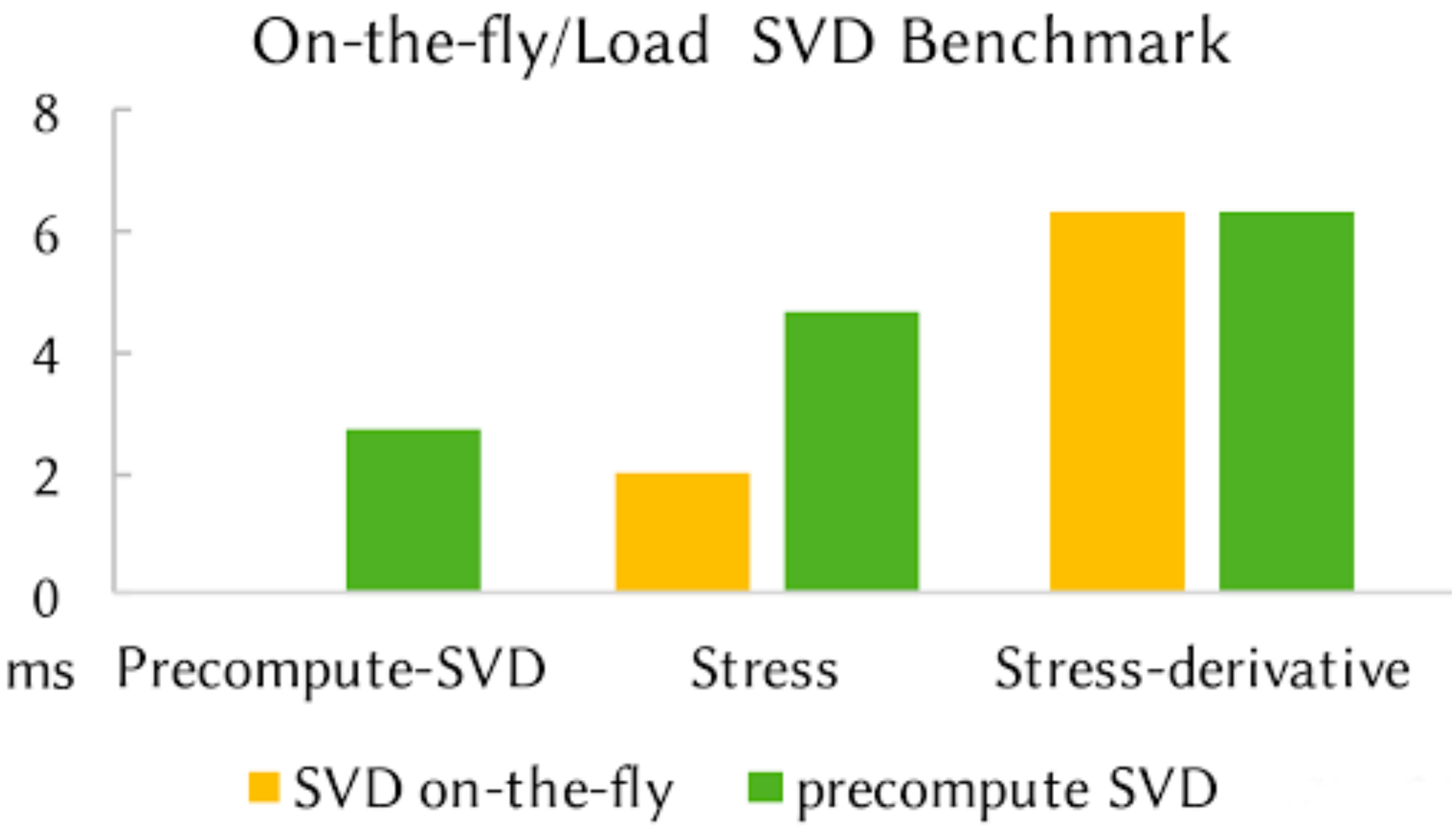
A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis

0.37 ns per 3x3 matrix

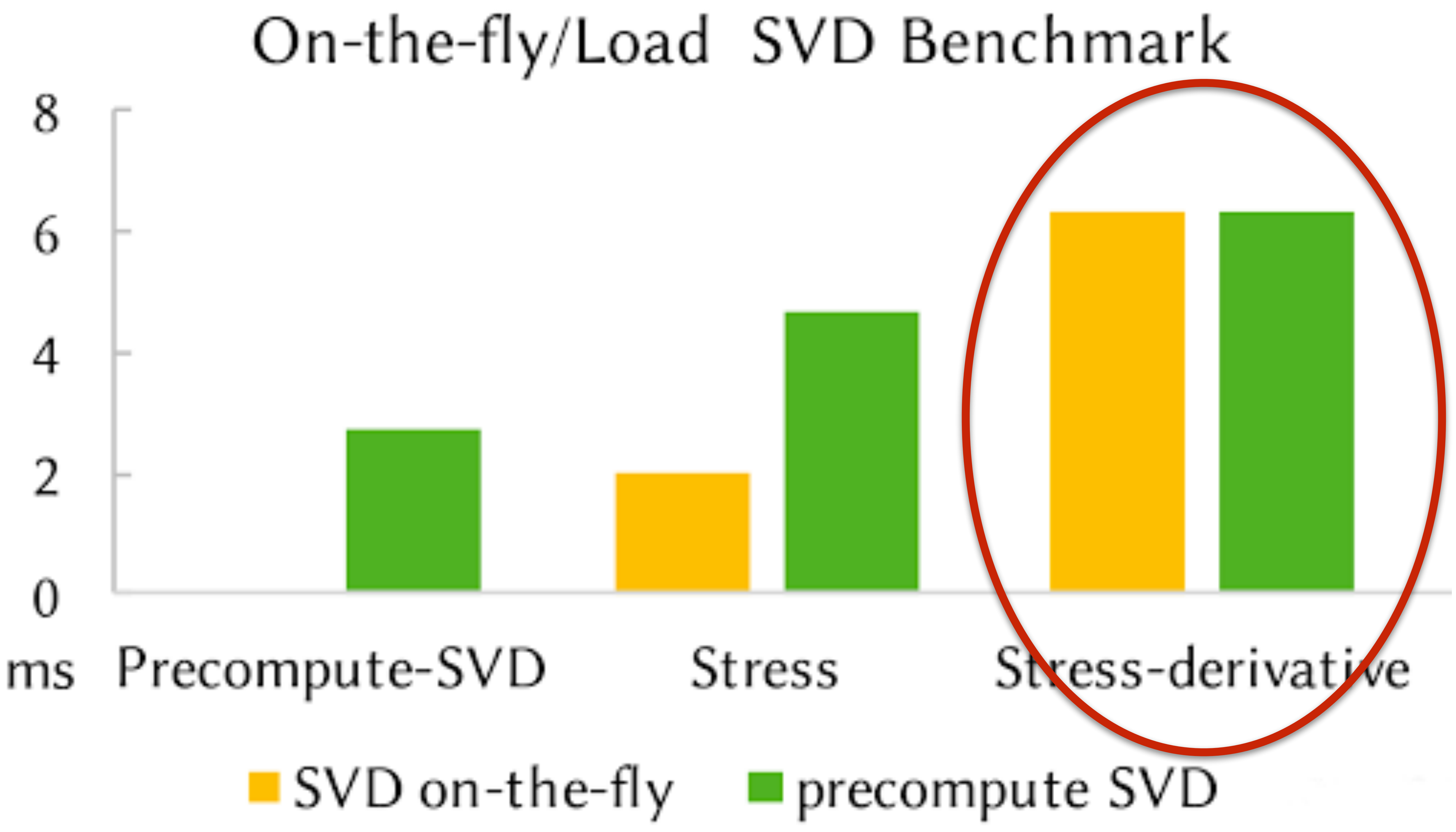
$$U \Sigma V^T$$

Store them in memory???

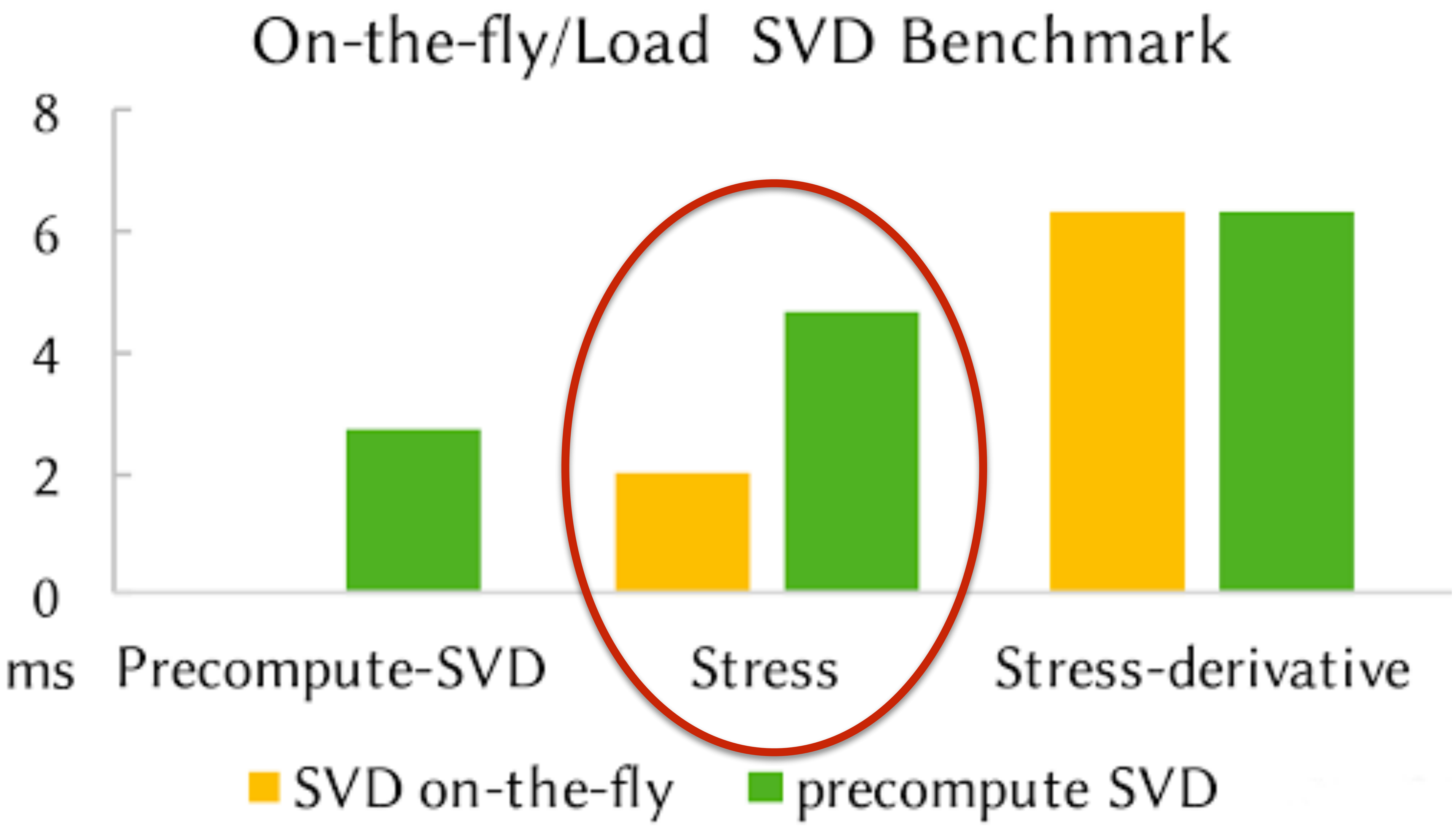
SVD computation



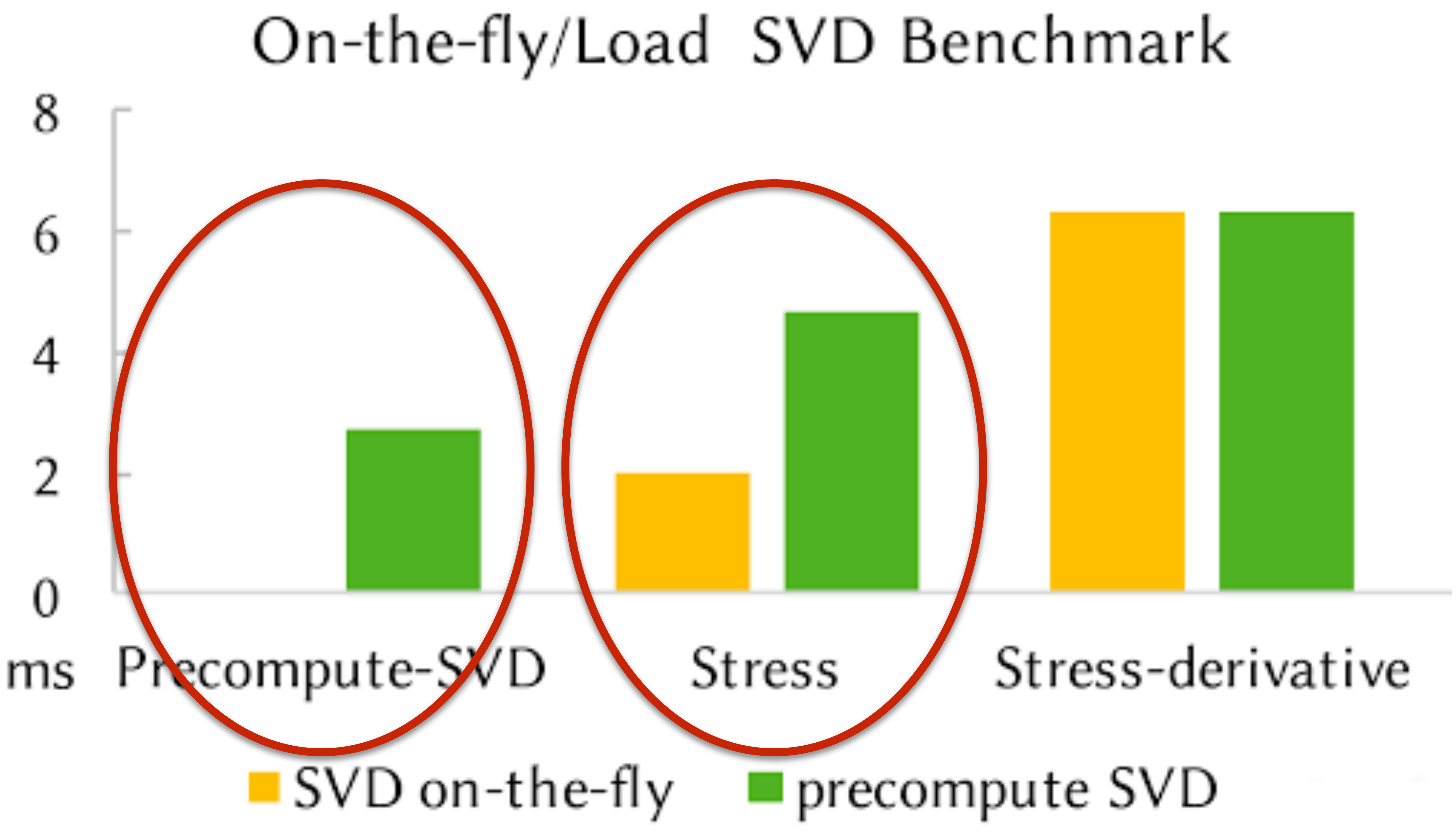
SVD computation



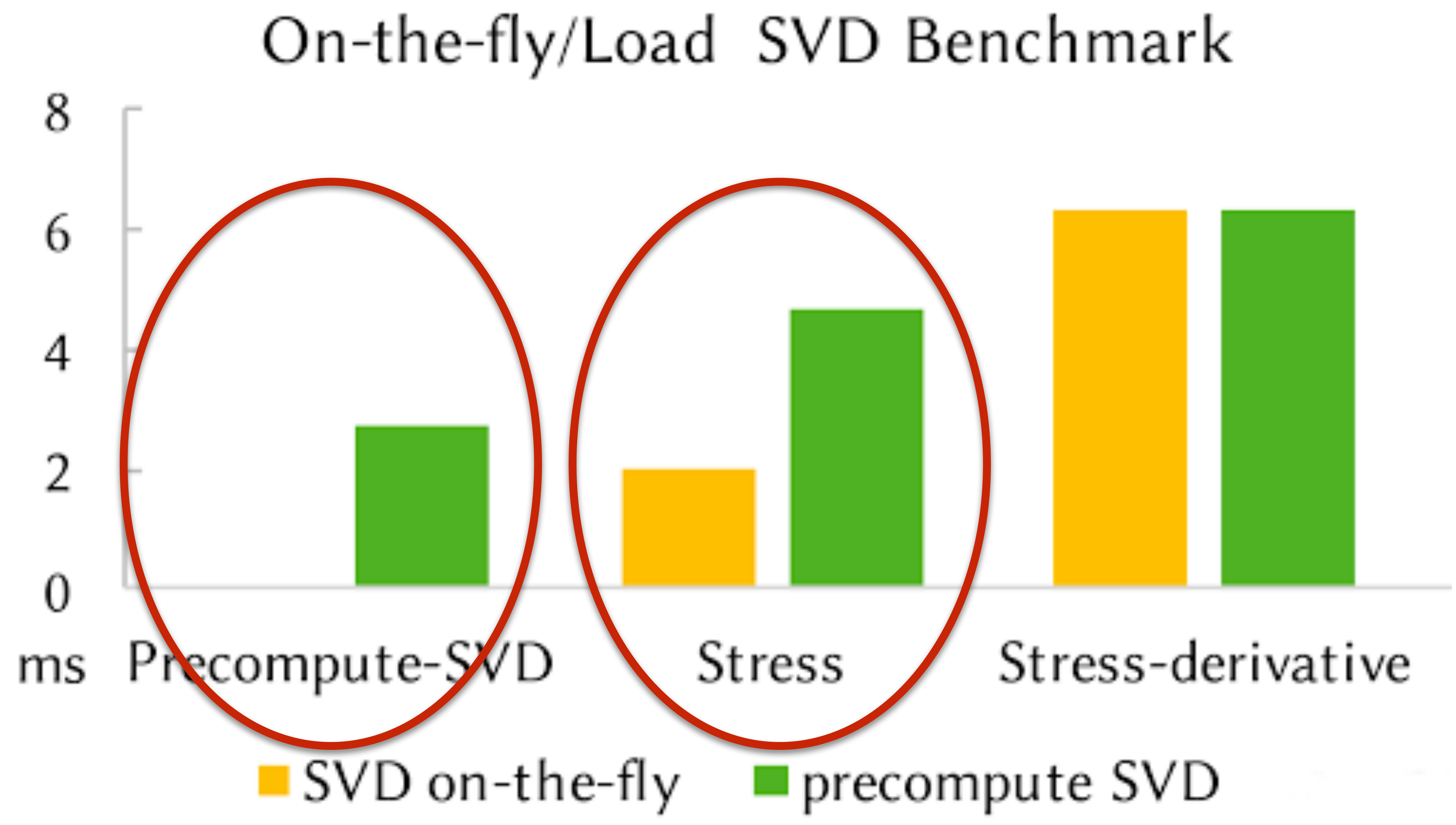
SVD computation



SVD computation



SVD computation



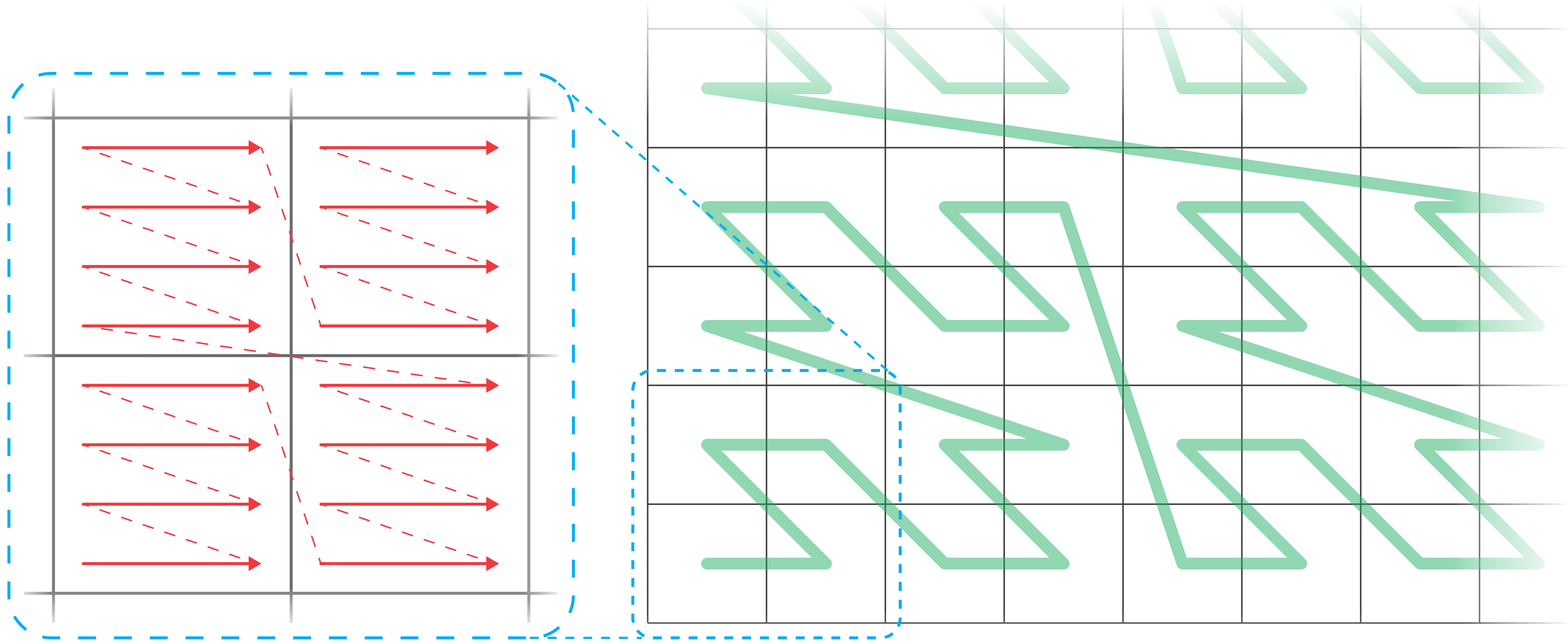
Resorting && reordering

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Particle storage

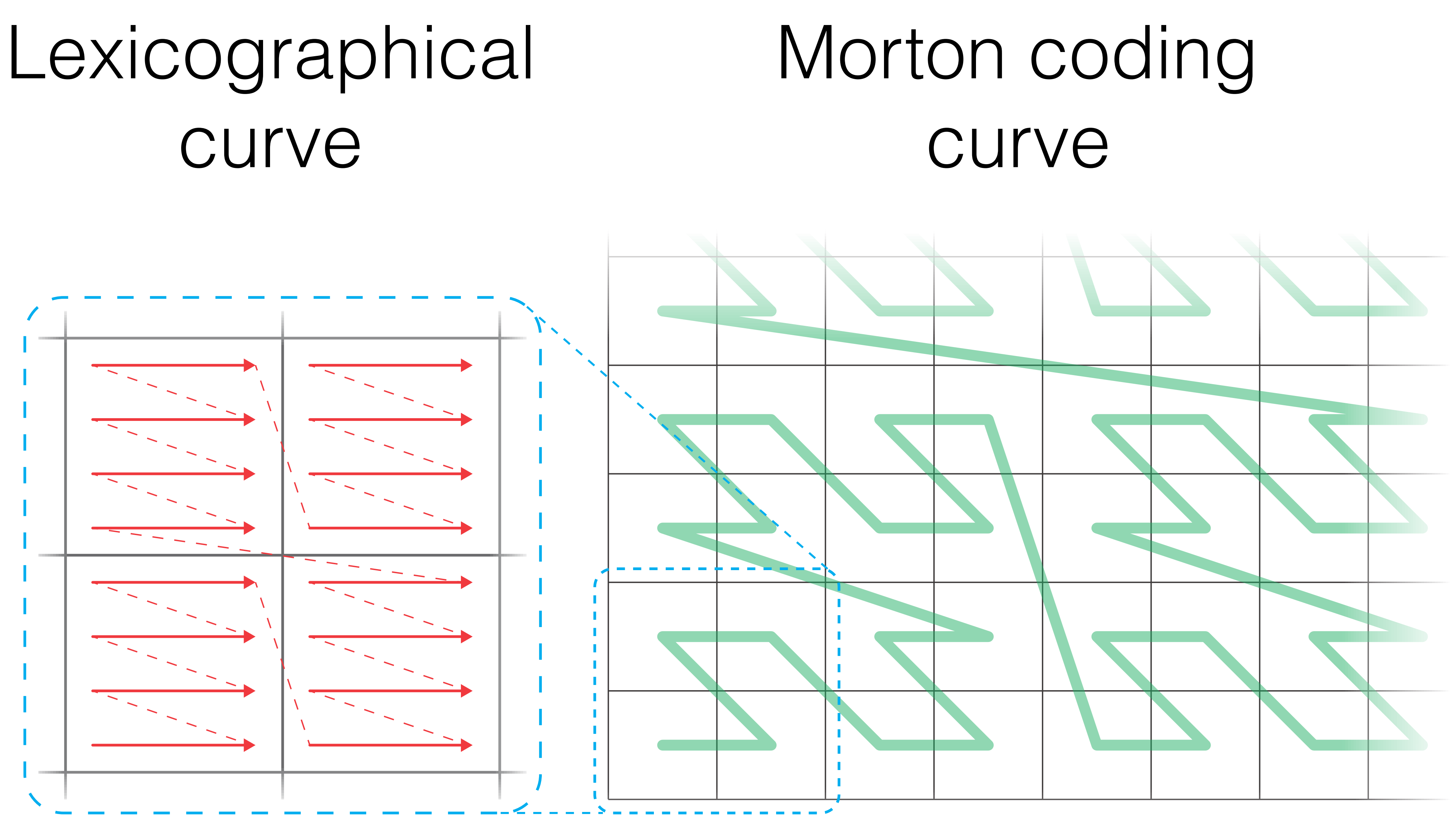
Lexicographical curve

Morton coding curve

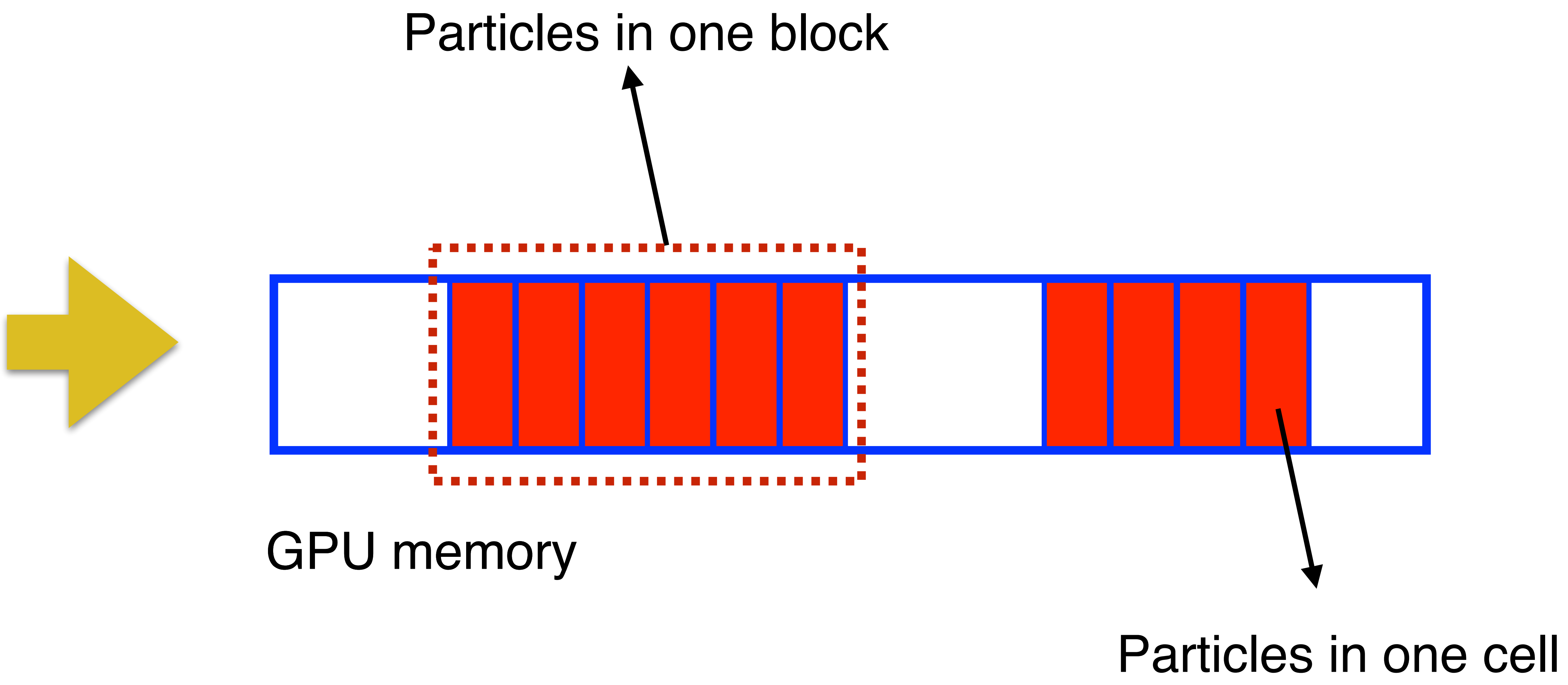


SPGrid
Setaluri et al 2014

Particle storage



SPGrid
Setaluri et al 2014



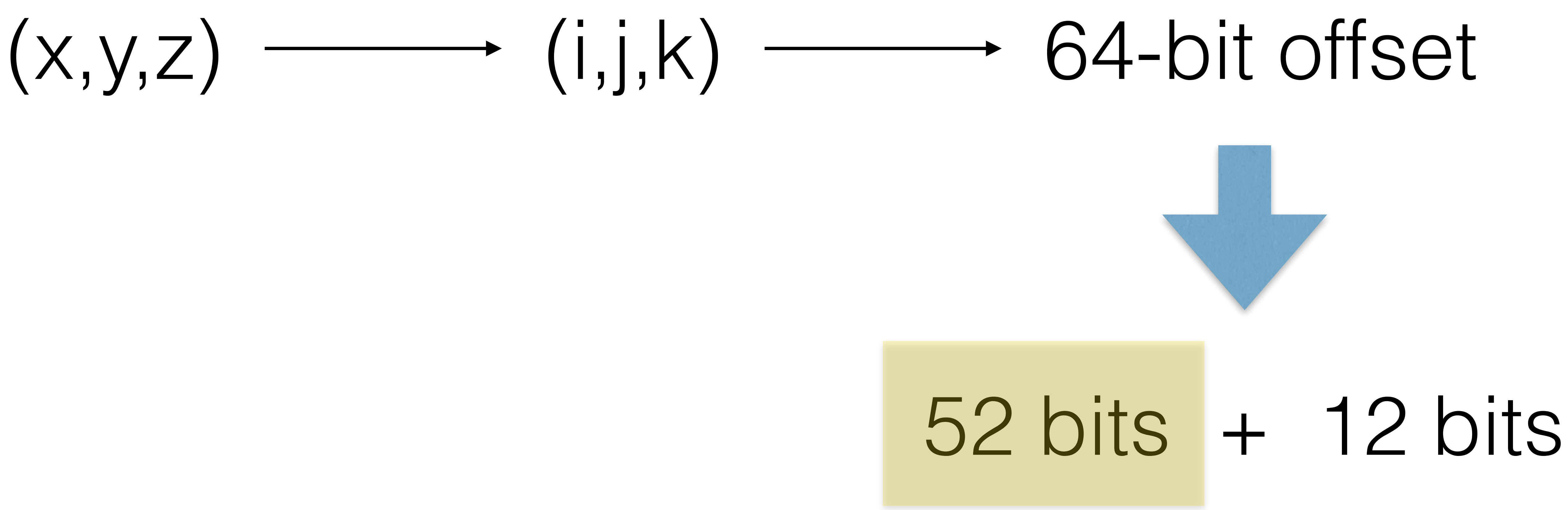
Radix vs histogram

(x,y,z) ⟶ (i,j,k) ⟶ 64-bit offset

Radix vs histogram

(x,y,z) → (i,j,k) → 64-bit offset

Radix vs histogram



Radix vs histogram

(x,y,z) → (i,j,k) → 64-bit offset

↓

52 bits + 12 bits

Continuous bins + 12 bits

Radix vs histogram

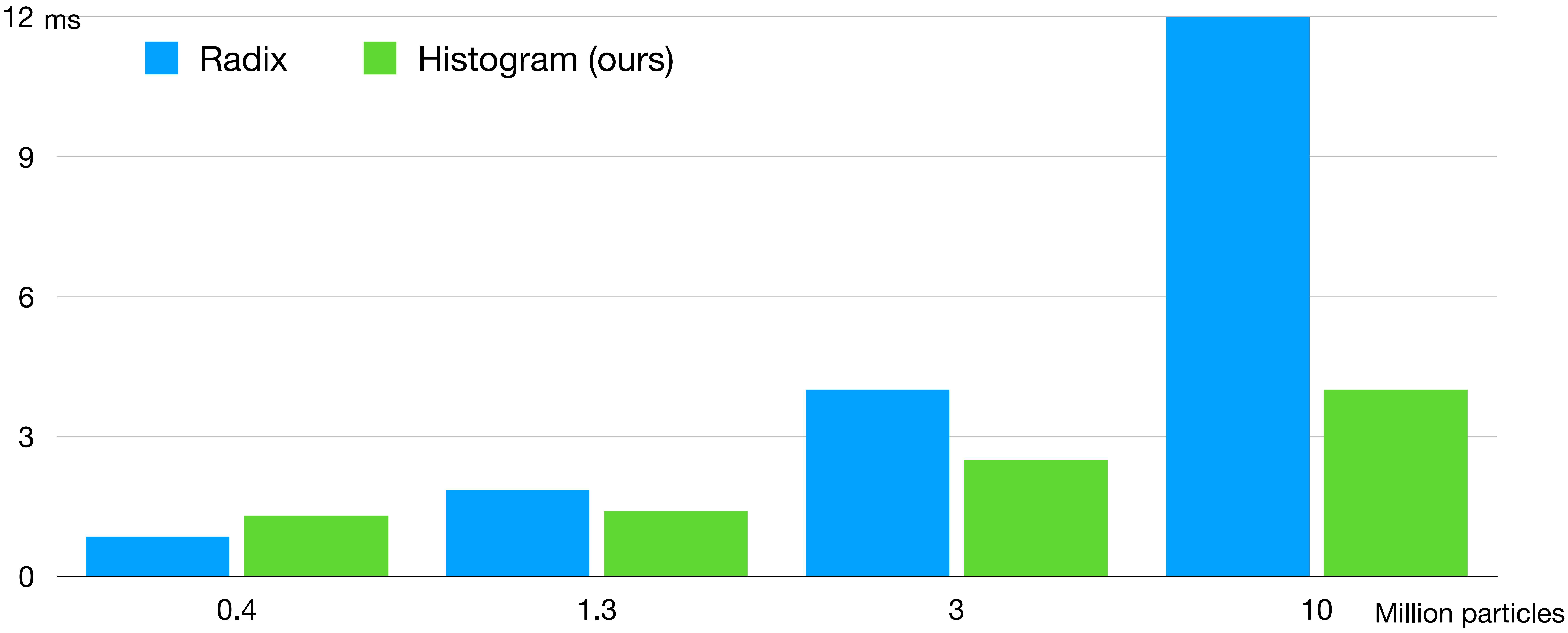
(x,y,z) → (i,j,k) → 64-bit offset

52 bits + 12 bits

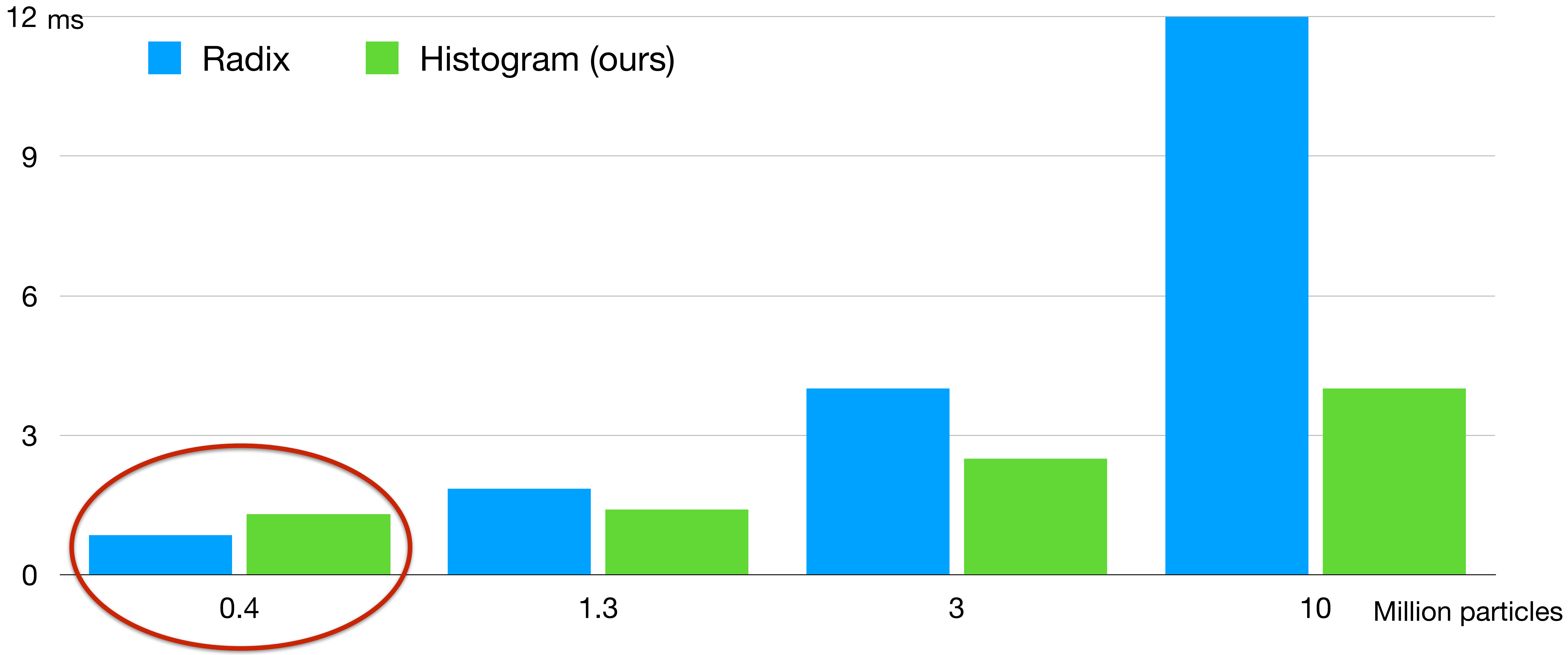
Sparsely occupied

Continuous bins + 12 bits

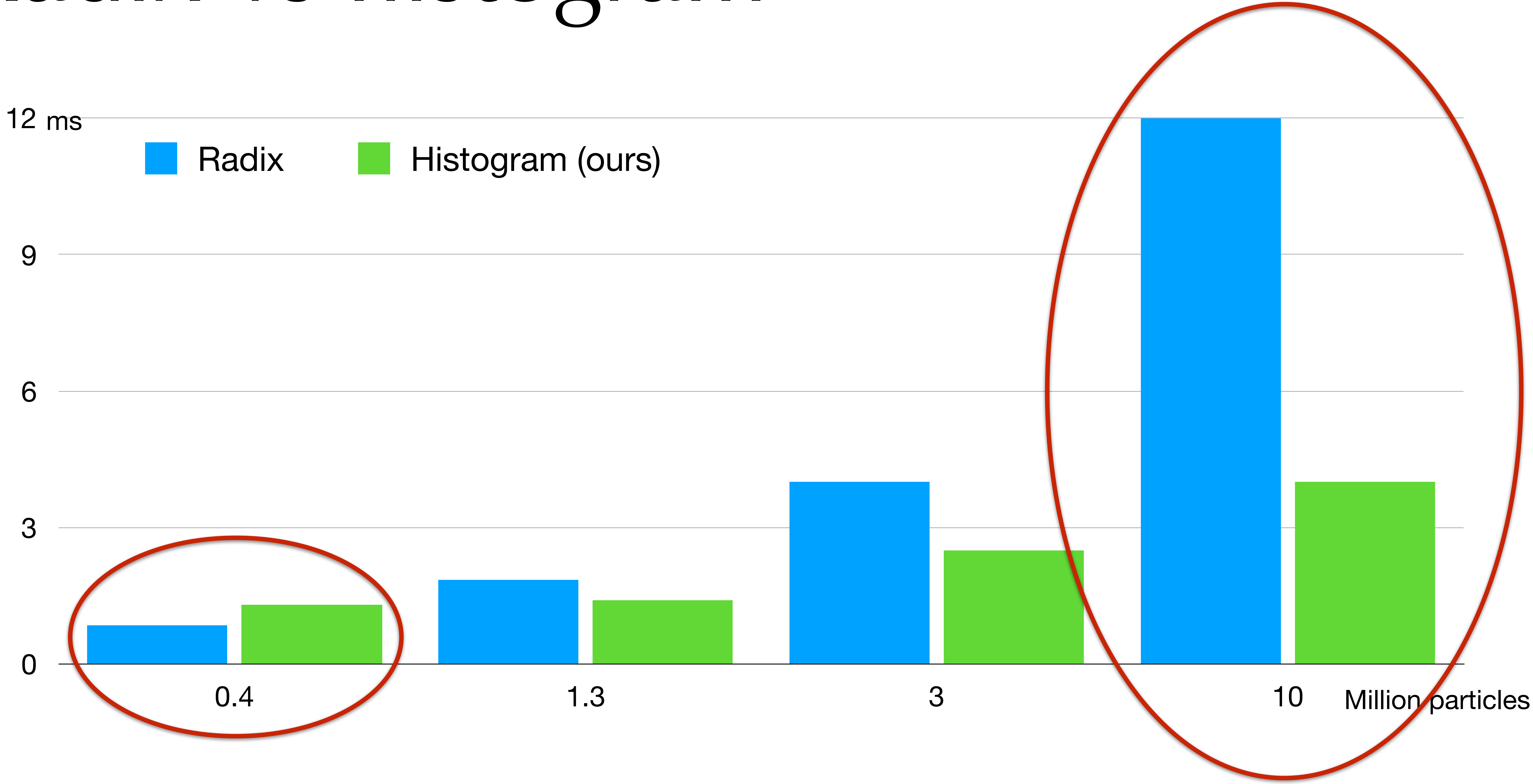
Radix vs histogram



Radix vs histogram



Radix vs histogram

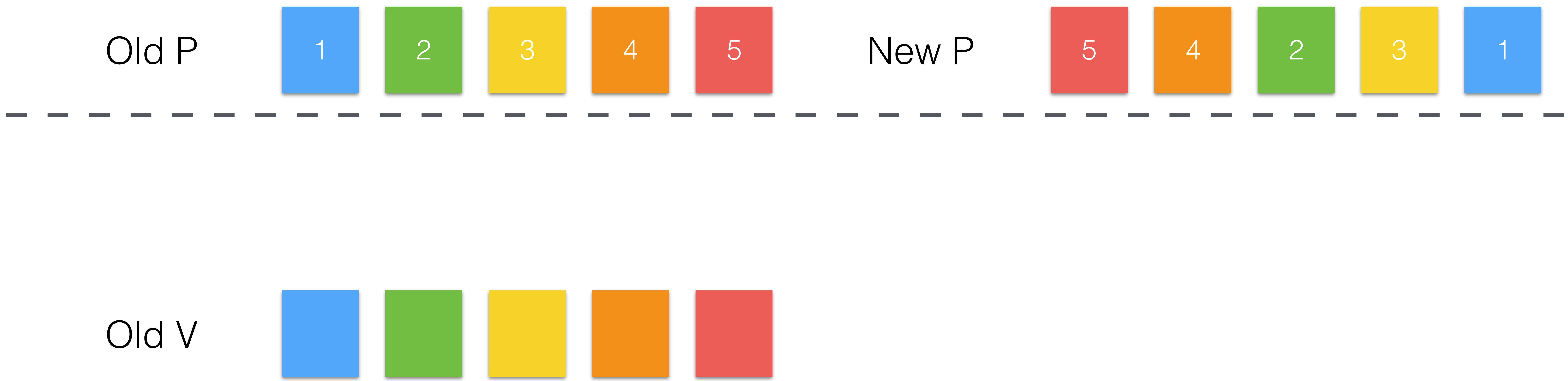


Reordering - memory movement

Reordering - memory movement

Reordering - memory movement

Reordering - memory movement

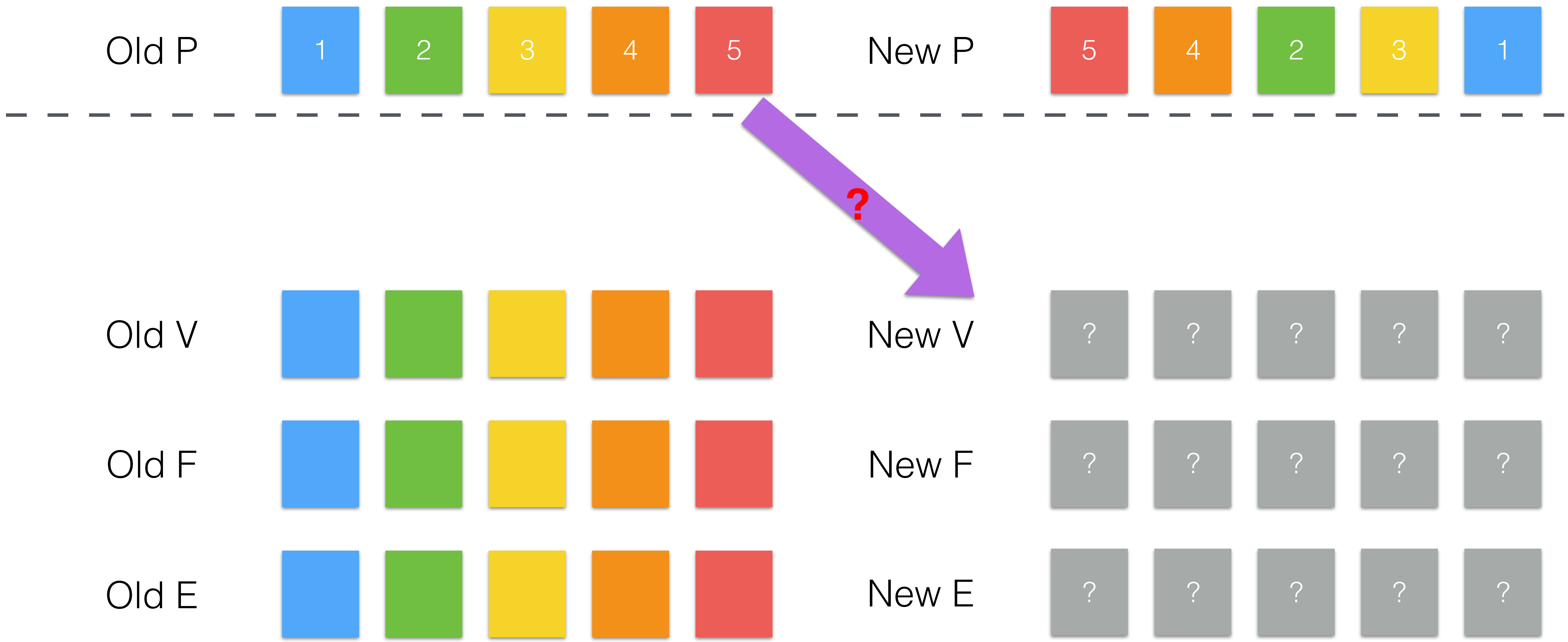


Reordering - memory movement

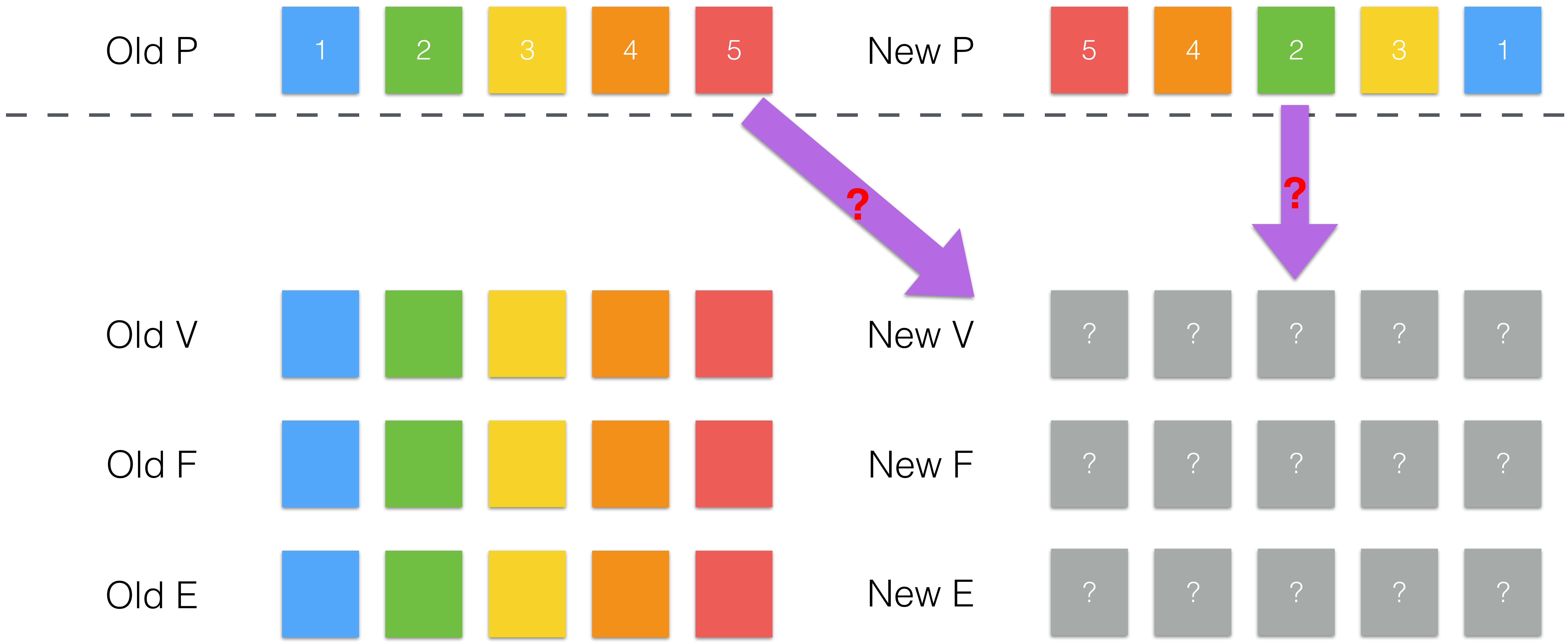
Reordering - memory movement

Reordering - memory movement

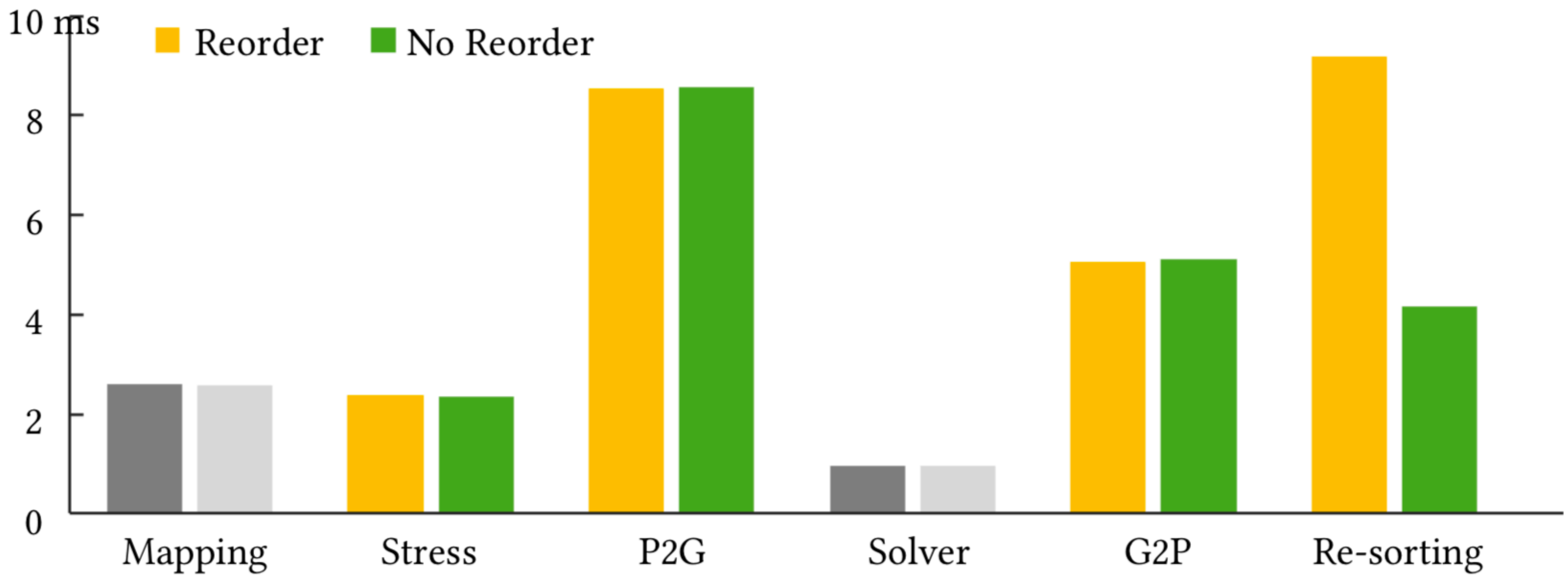
Reordering - memory movement



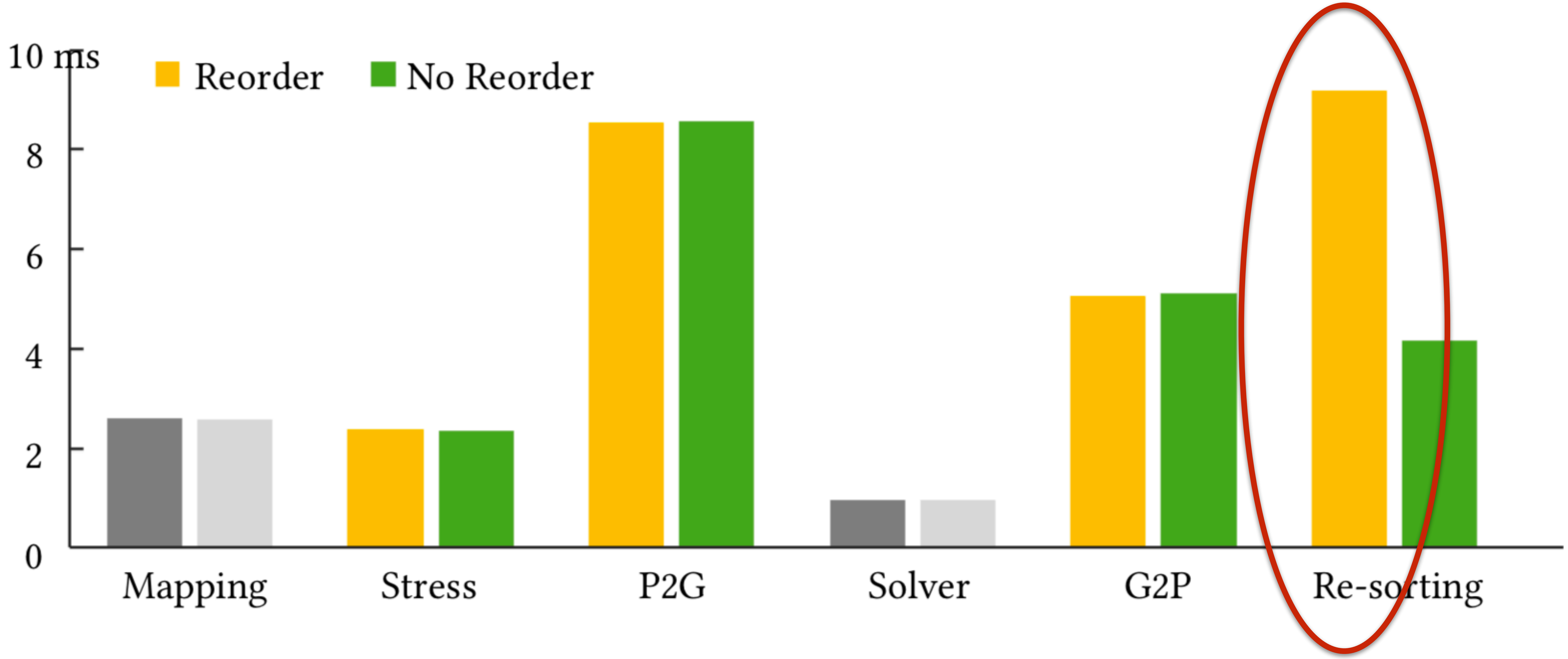
Reordering - memory movement



Pure memory operation - slow

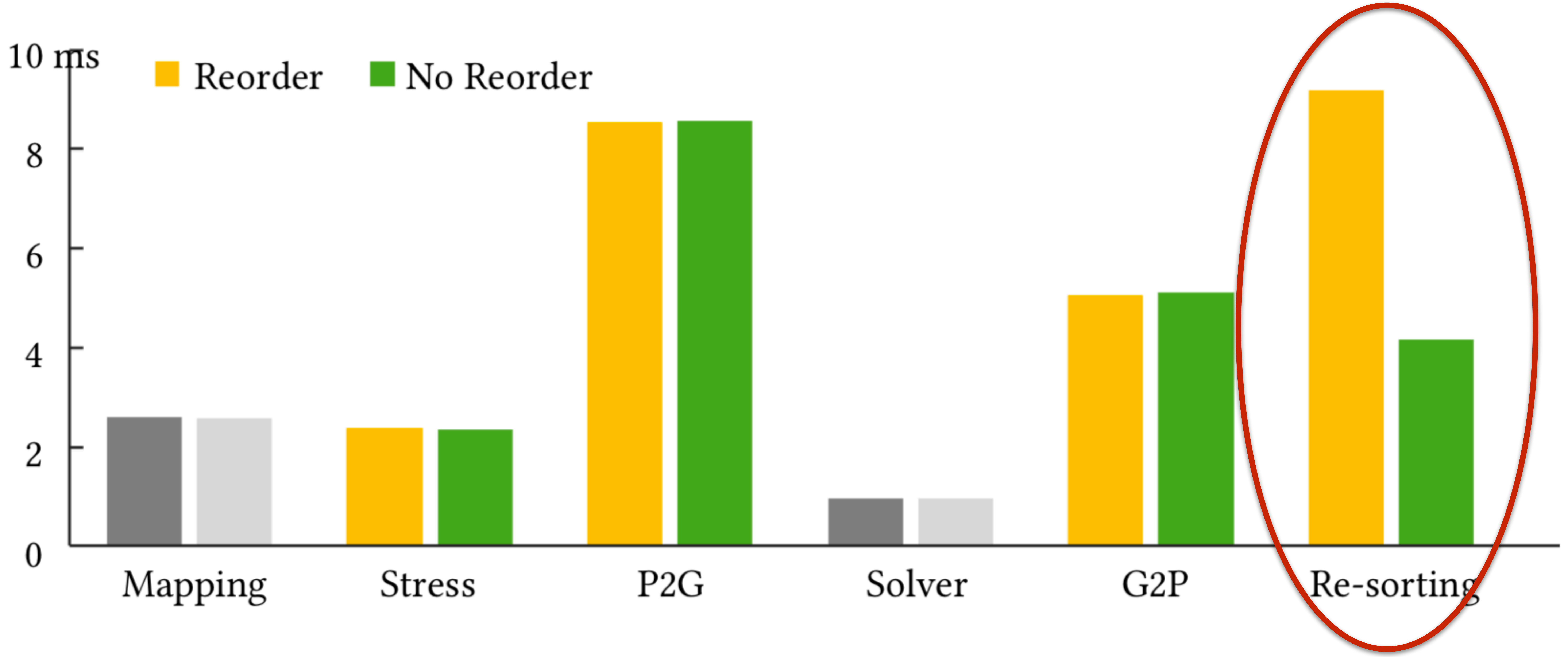


Pure memory operation - slow



Delayed reordering

Delayed reordering



Pure grid operations

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Pure grid operations

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Boundary condition

Particle grid communications

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Particle grid communications

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Build particle-grid mapping

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Build particle-grid mapping

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Particles <-> nodes

Build particle-grid mapping

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Particles <-> nodes

Particles <-> blocks

Particle to grid (P2G)

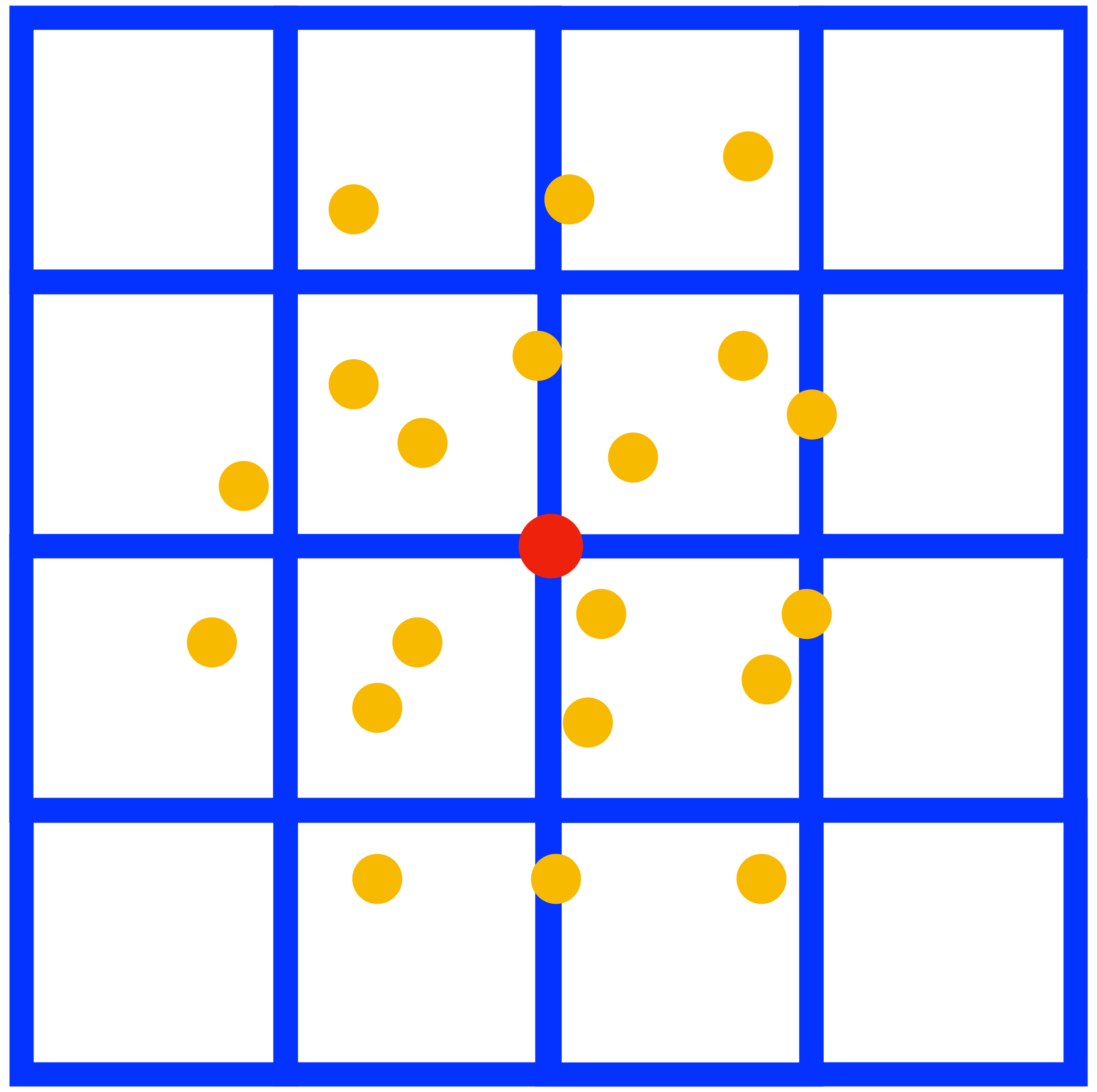
```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Particle to grid (P2G)

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

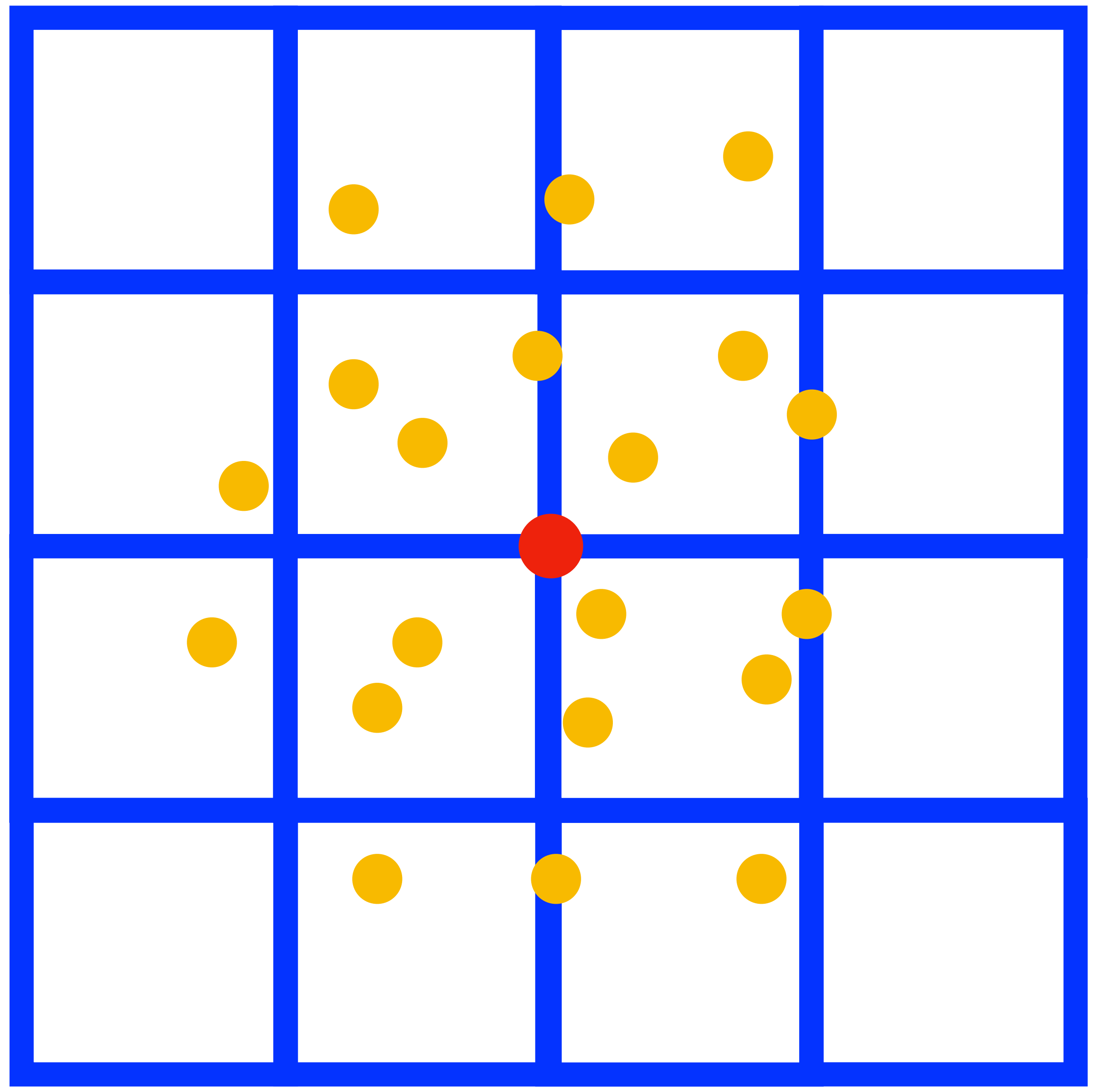
B-Spline

Particle V.S. node per thread



CUDA thread - grid node

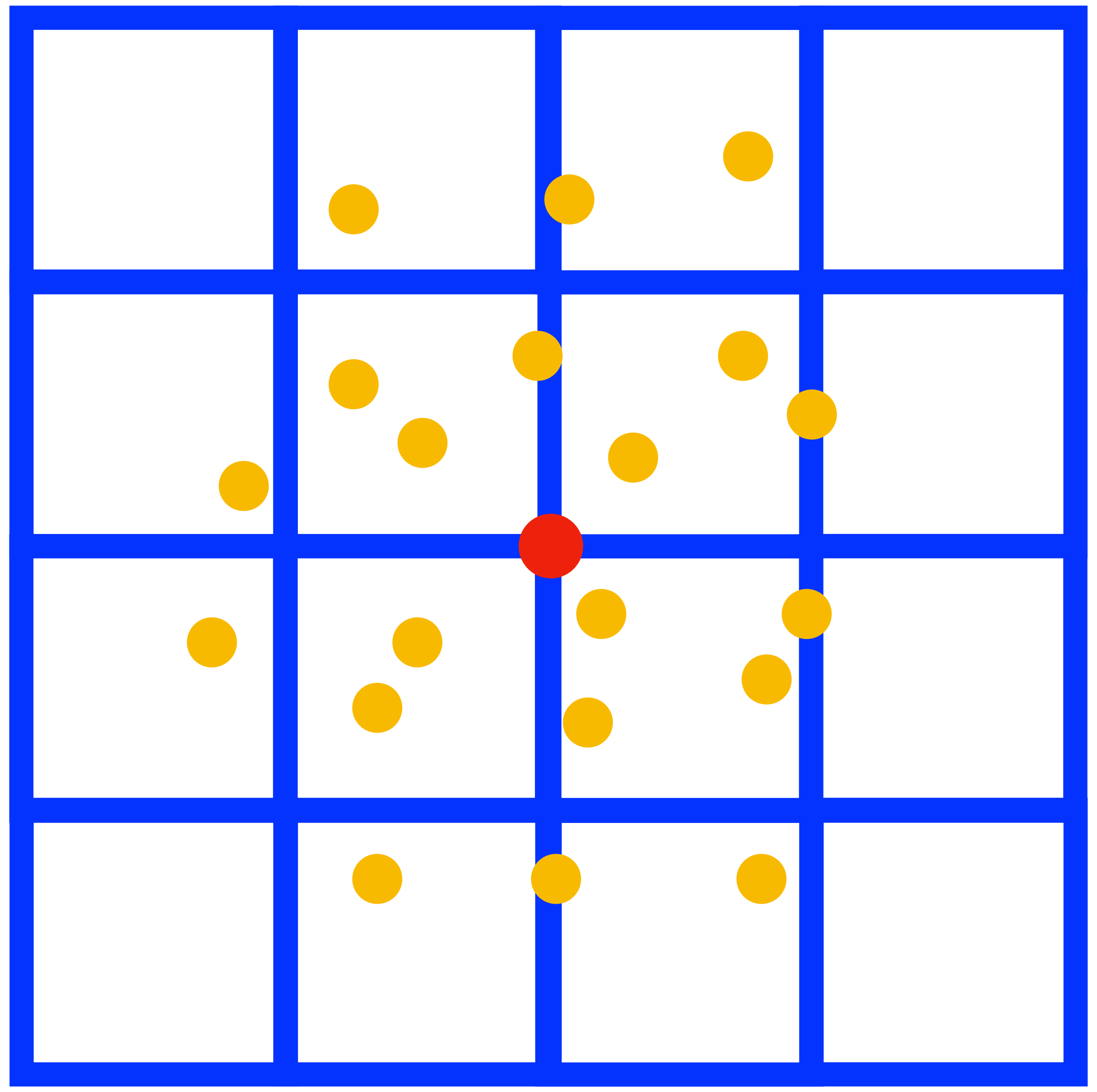
Particle V.S. node per thread



CUDA thread - grid node ●

V.S.

Particle V.S. node per thread



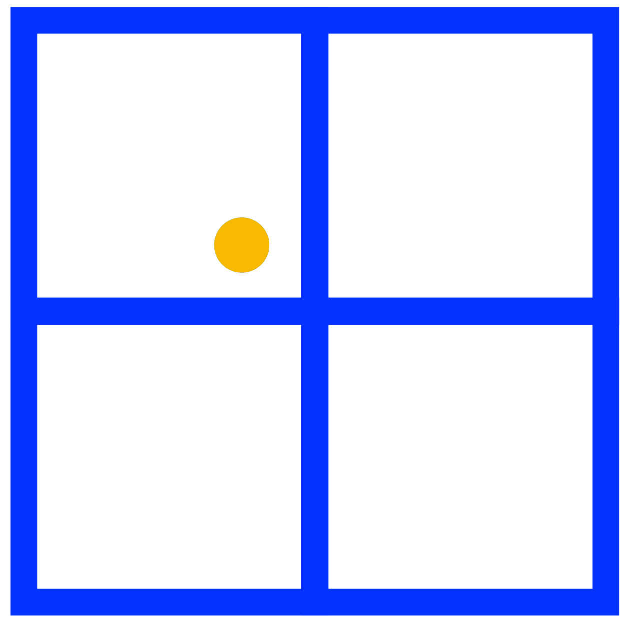
CUDA thread - grid node ●

V.S.

CUDA thread - particle ●

Scattering V.S. gathering

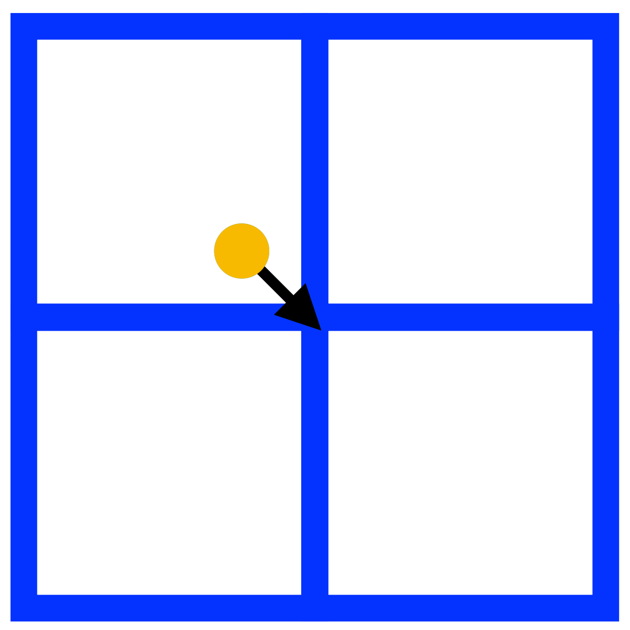
Particle
Grid node



CUDA thread -

Scattering V.S. gathering

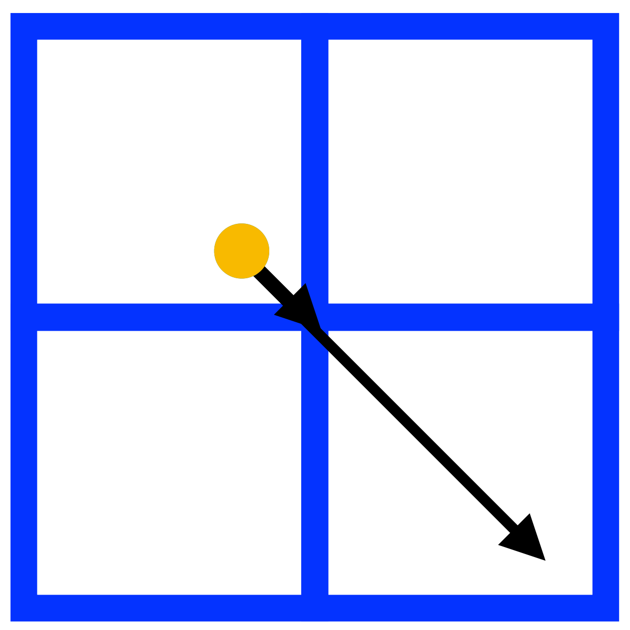
Particle
Grid node



CUDA thread -

Scattering V.S. gathering

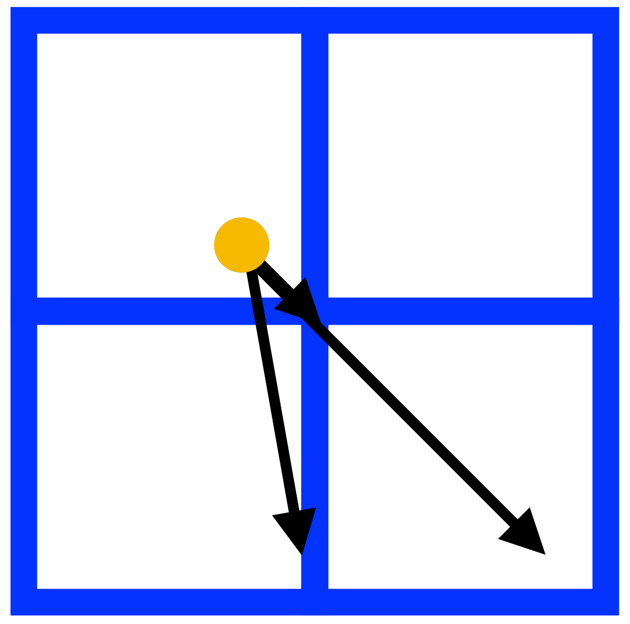
Particle ●
Grid node ●



CUDA thread - ●

Scattering V.S. gathering

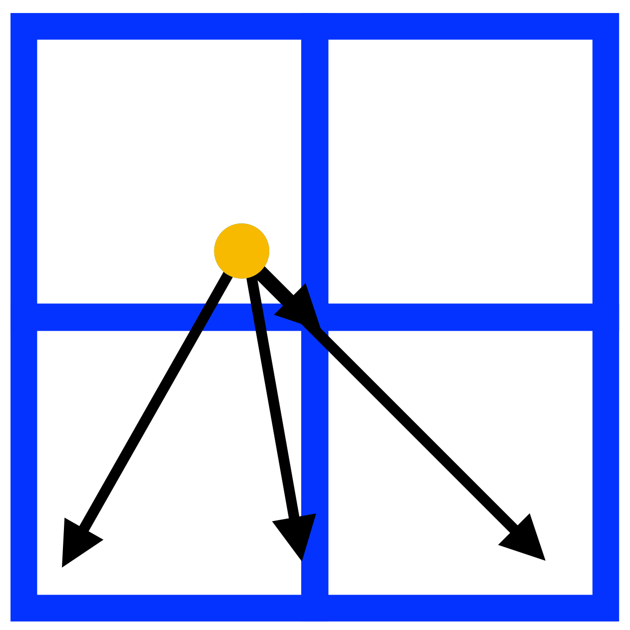
Particle
Grid node



CUDA thread -

Scattering V.S. gathering

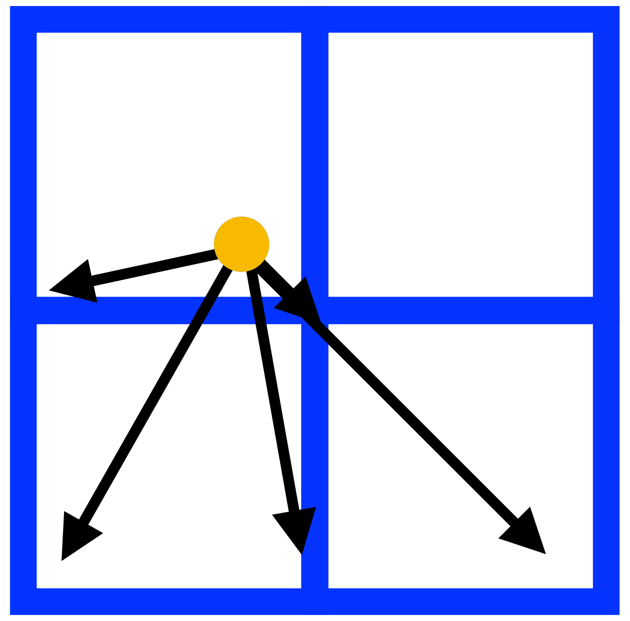
Particle
Grid node



CUDA thread -

Scattering V.S. gathering

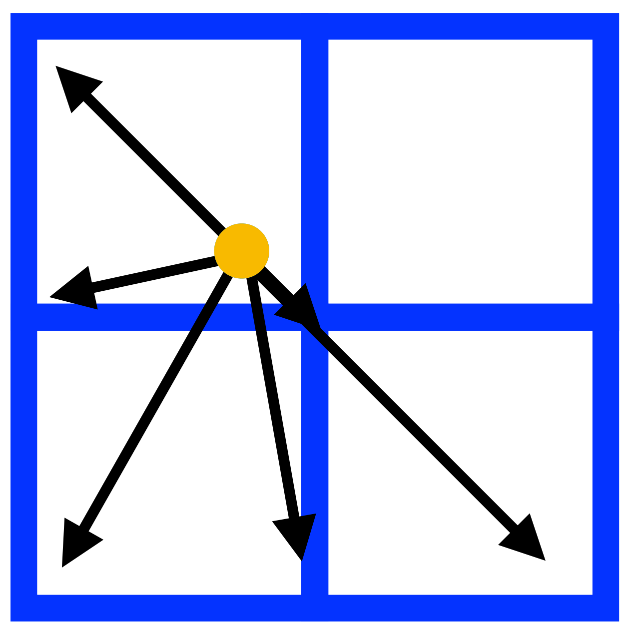
Particle
Grid node



CUDA thread -

Scattering V.S. gathering

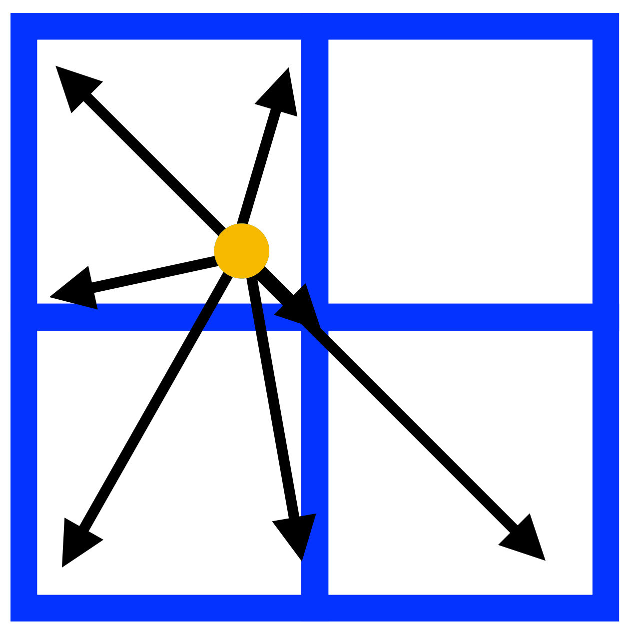
Particle
Grid node



CUDA thread -

Scattering V.S. gathering

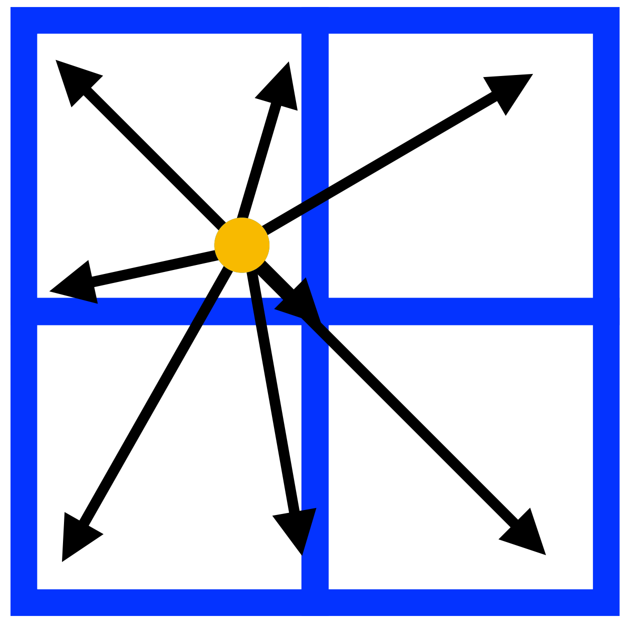
Particle
Grid node



CUDA thread -

Scattering V.S. gathering

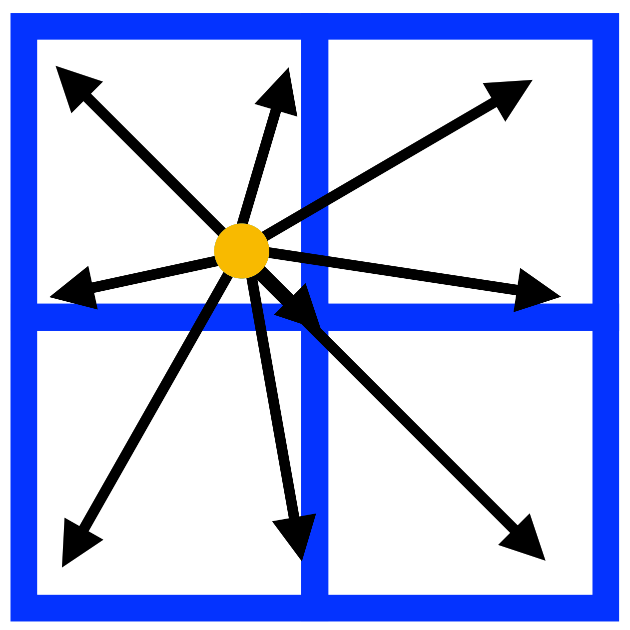
Particle
Grid node



CUDA thread -

Scattering V.S. gathering

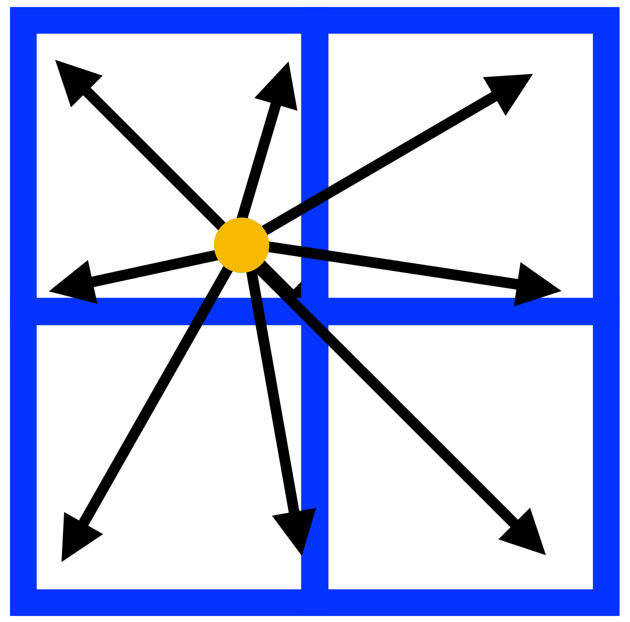
Particle
Grid node



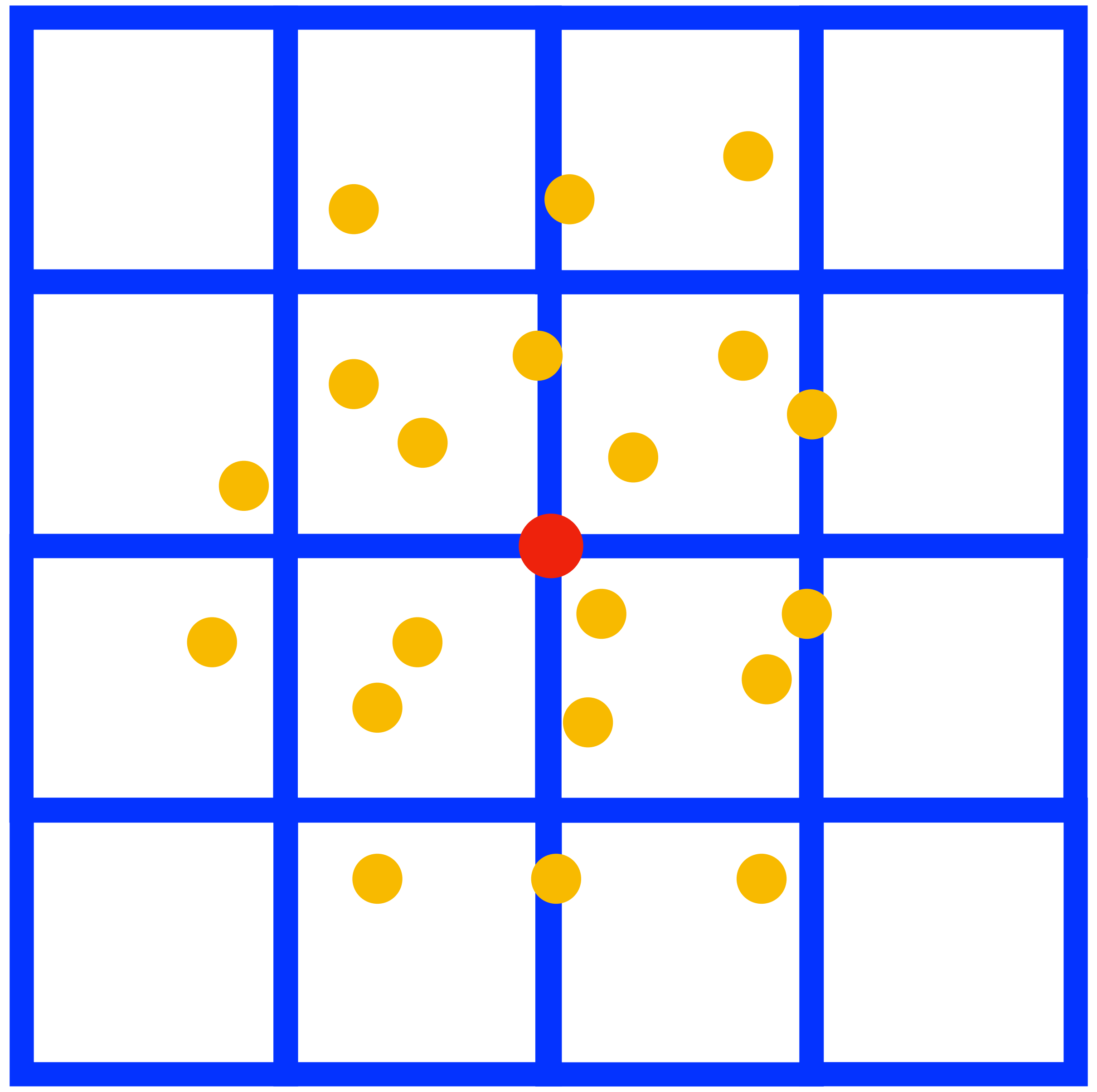
CUDA thread -

Scattering V.S. gathering

Particle
Grid node



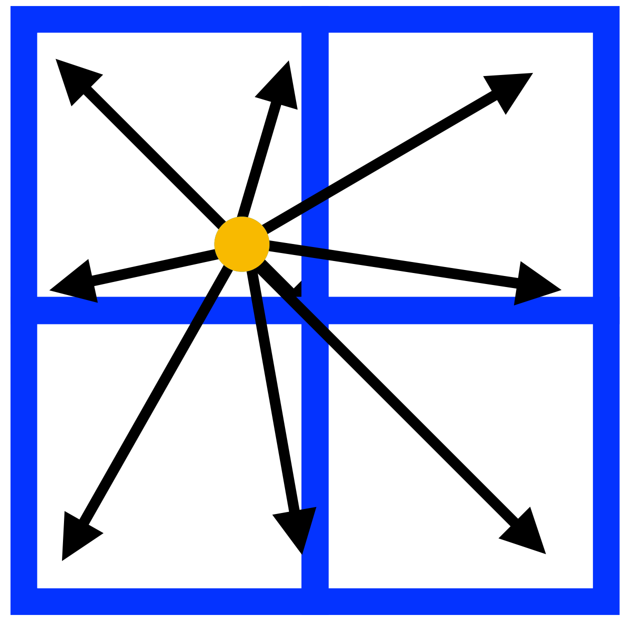
CUDA thread -



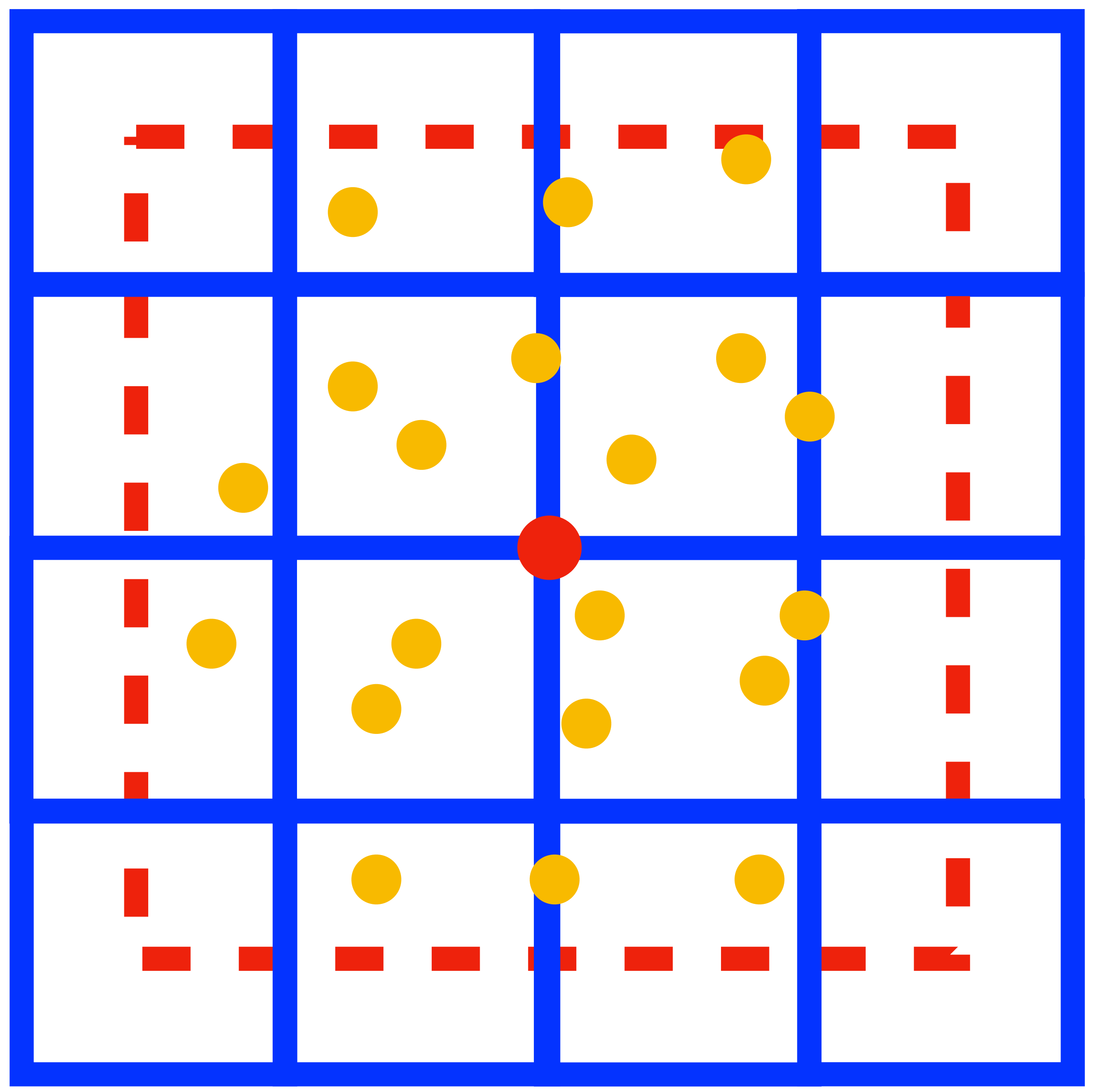
CUDA thread -

Scattering V.S. gathering

Particle
Grid node



CUDA thread -

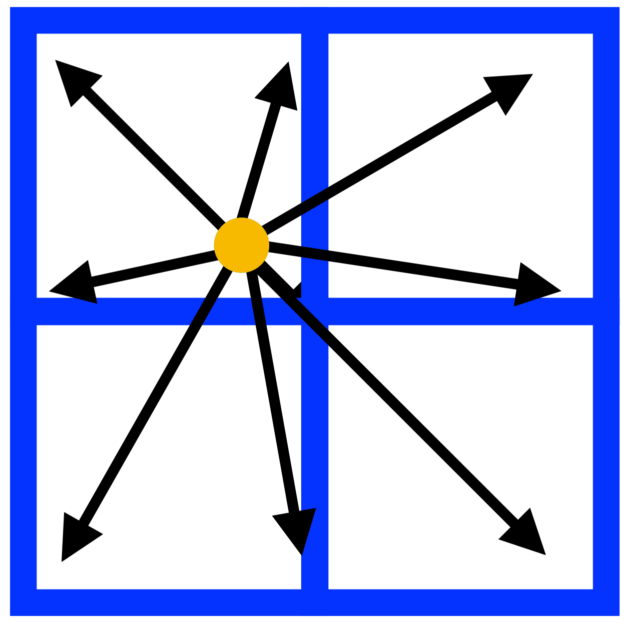


CUDA thread -

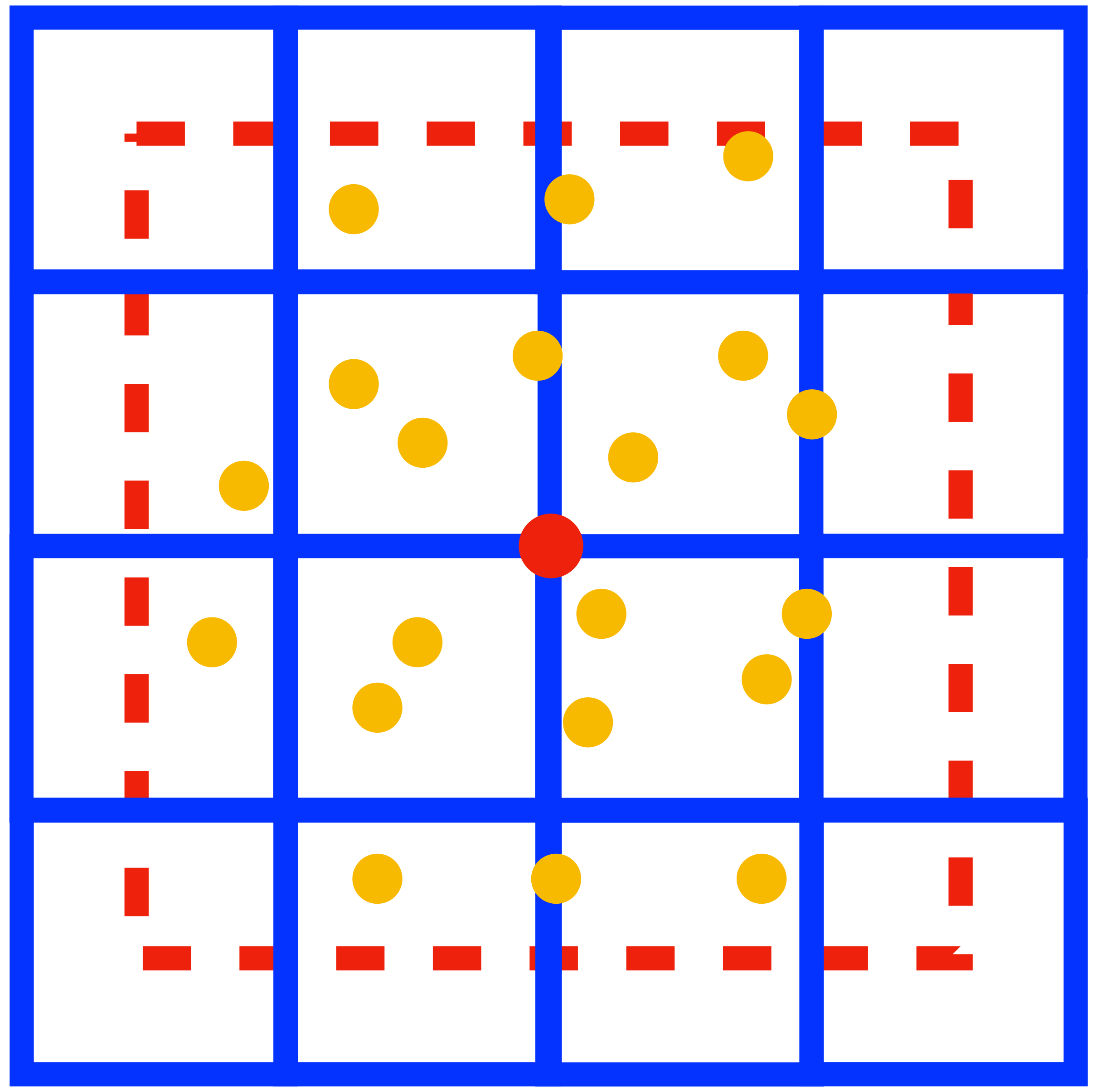
Scattering V.S. gathering

Particle

Grid node



CUDA thread -



CUDA thread -

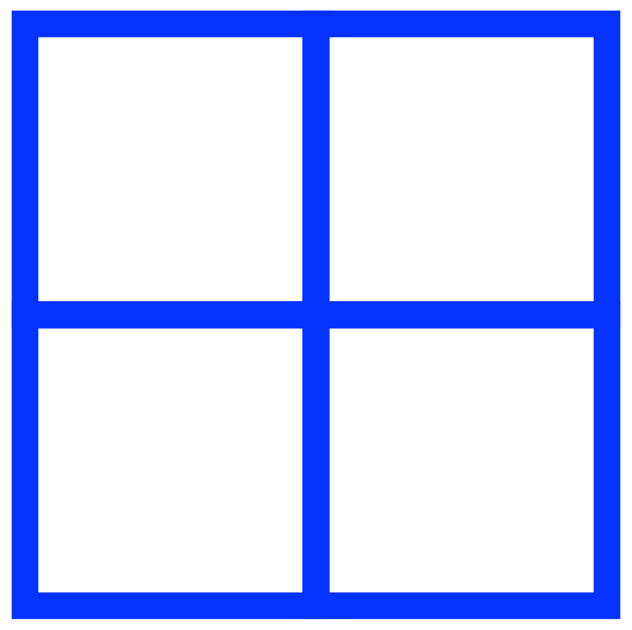
Particle list for a node -

Gathering - thread divergence

Write hazard

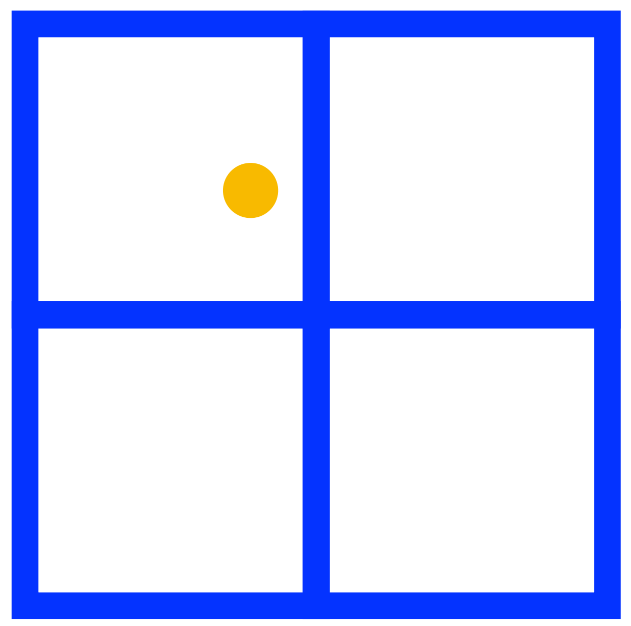
Particle

Grid node



Write hazard

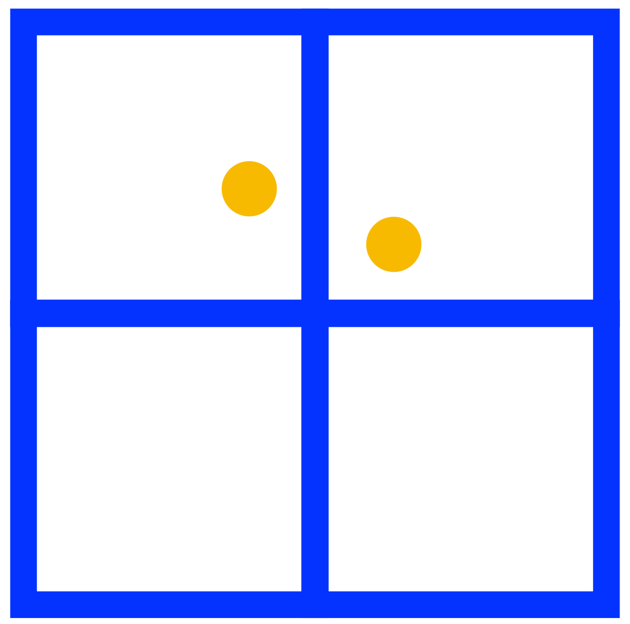
Particle
Grid node



Thread 1

Write hazard

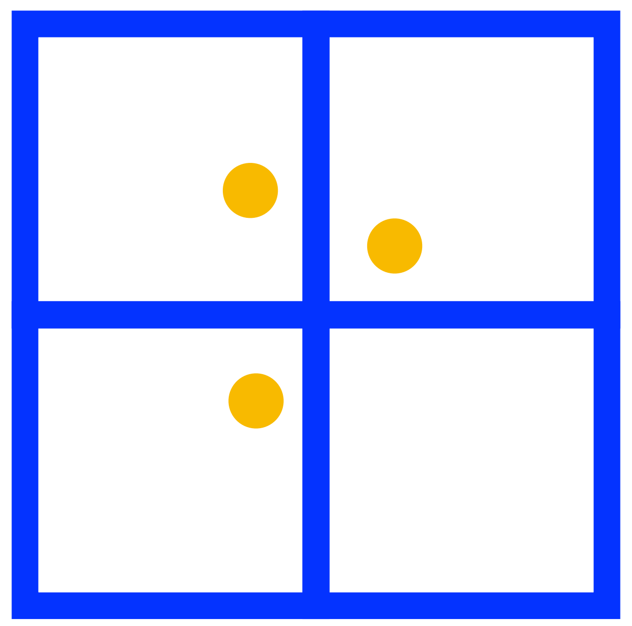
Particle
Grid node



Thread 1
Thread 2

Write hazard

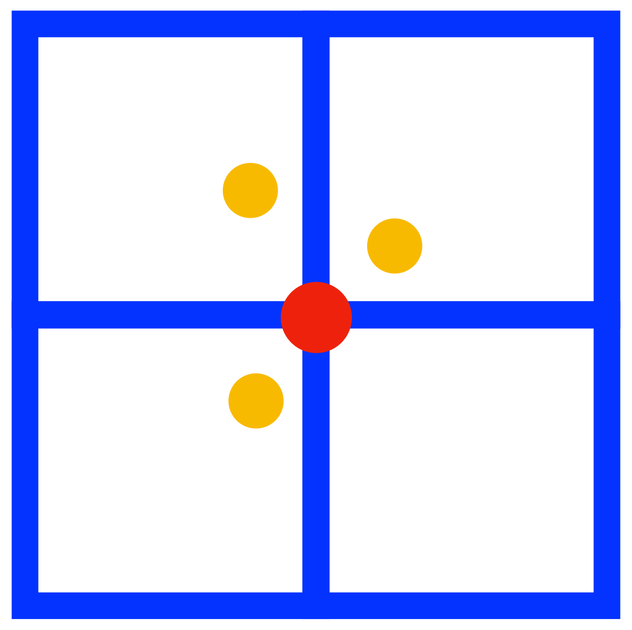
Particle
Grid node



Thread 1
Thread 2
Thread 3

Write hazard

Particle
Grid node



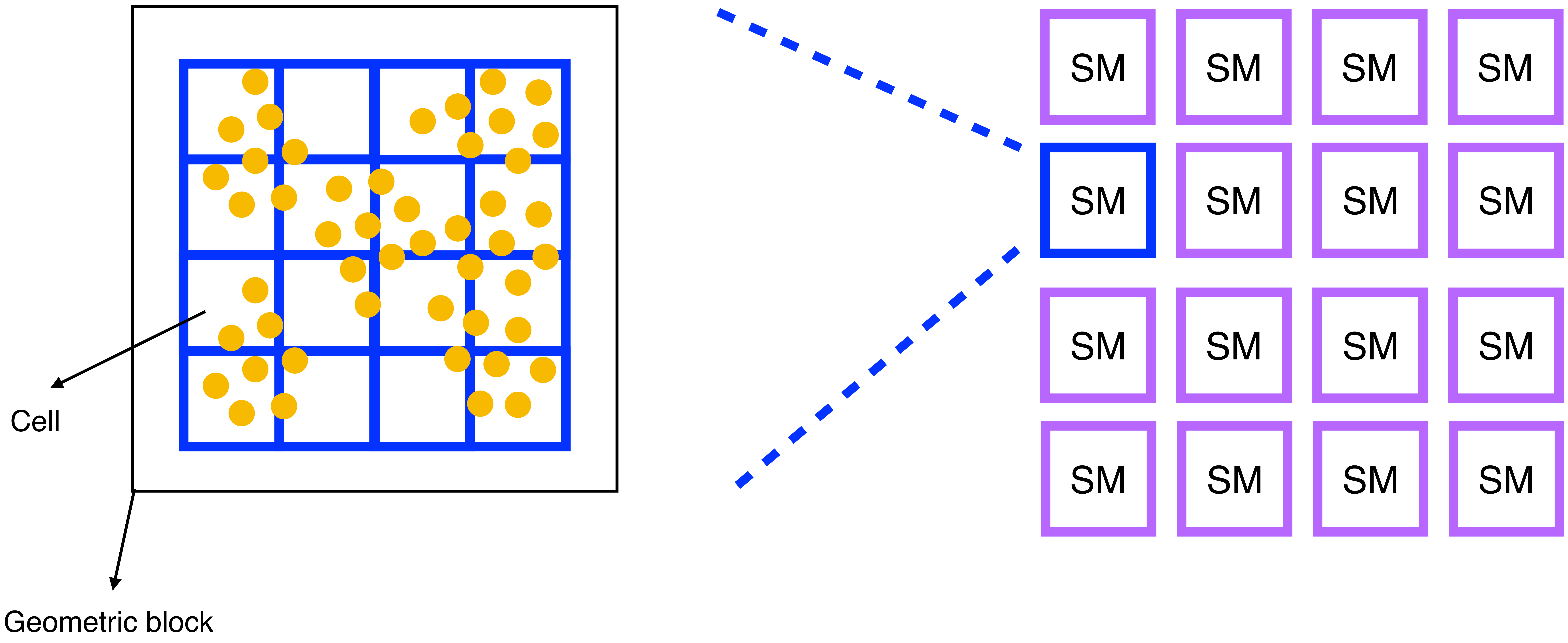
Thread 1
Thread 2
Thread 3

Write hazard

Particle
Grid node

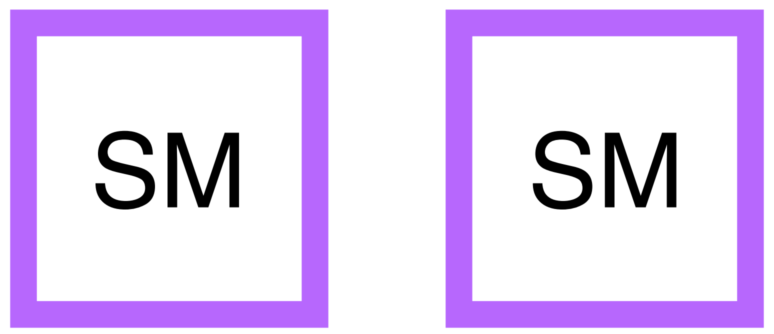
Hierarchy

Particle
Grid node

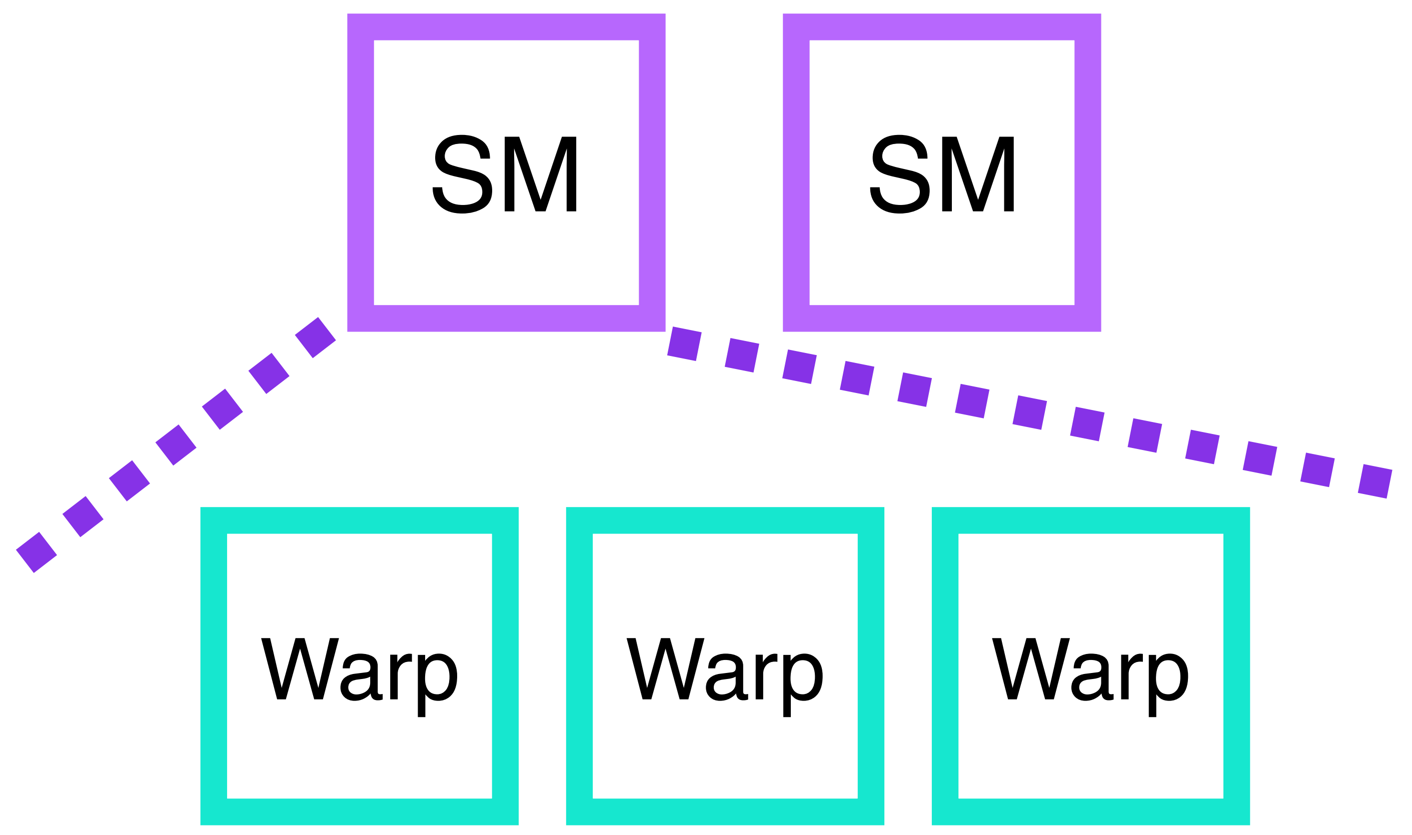


Hierarchy

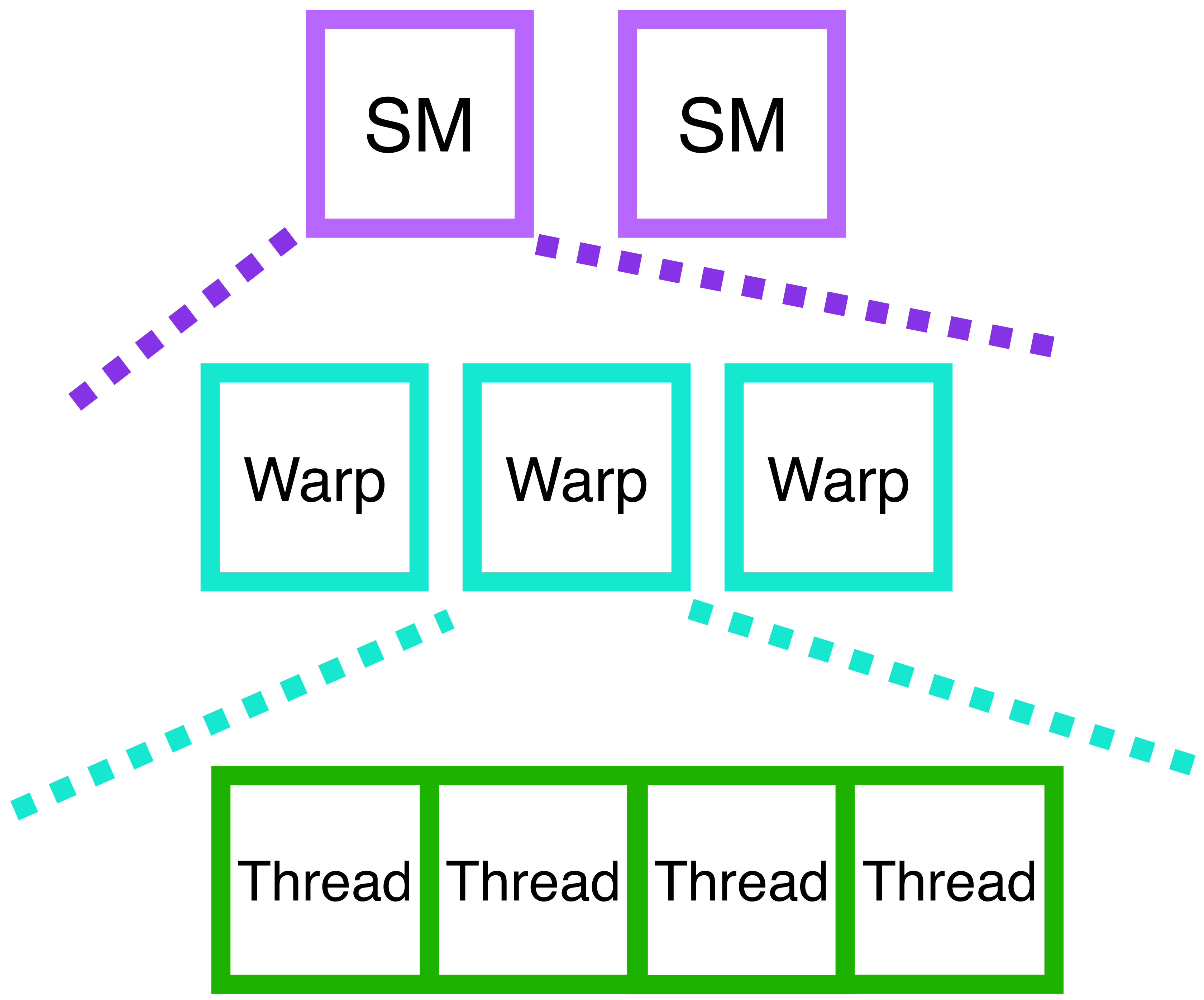
Hierarchy



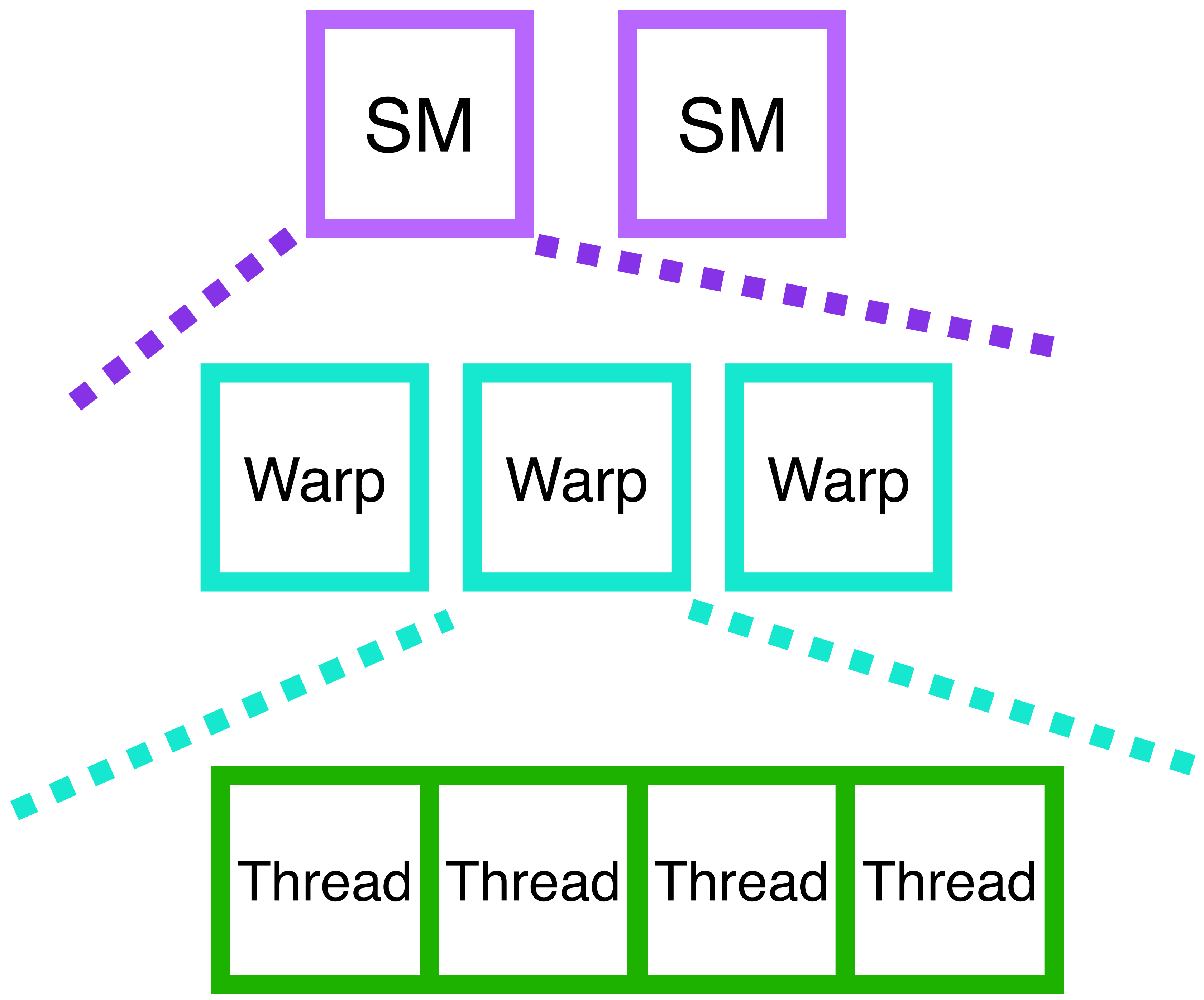
Hierarchy



Hierarchy

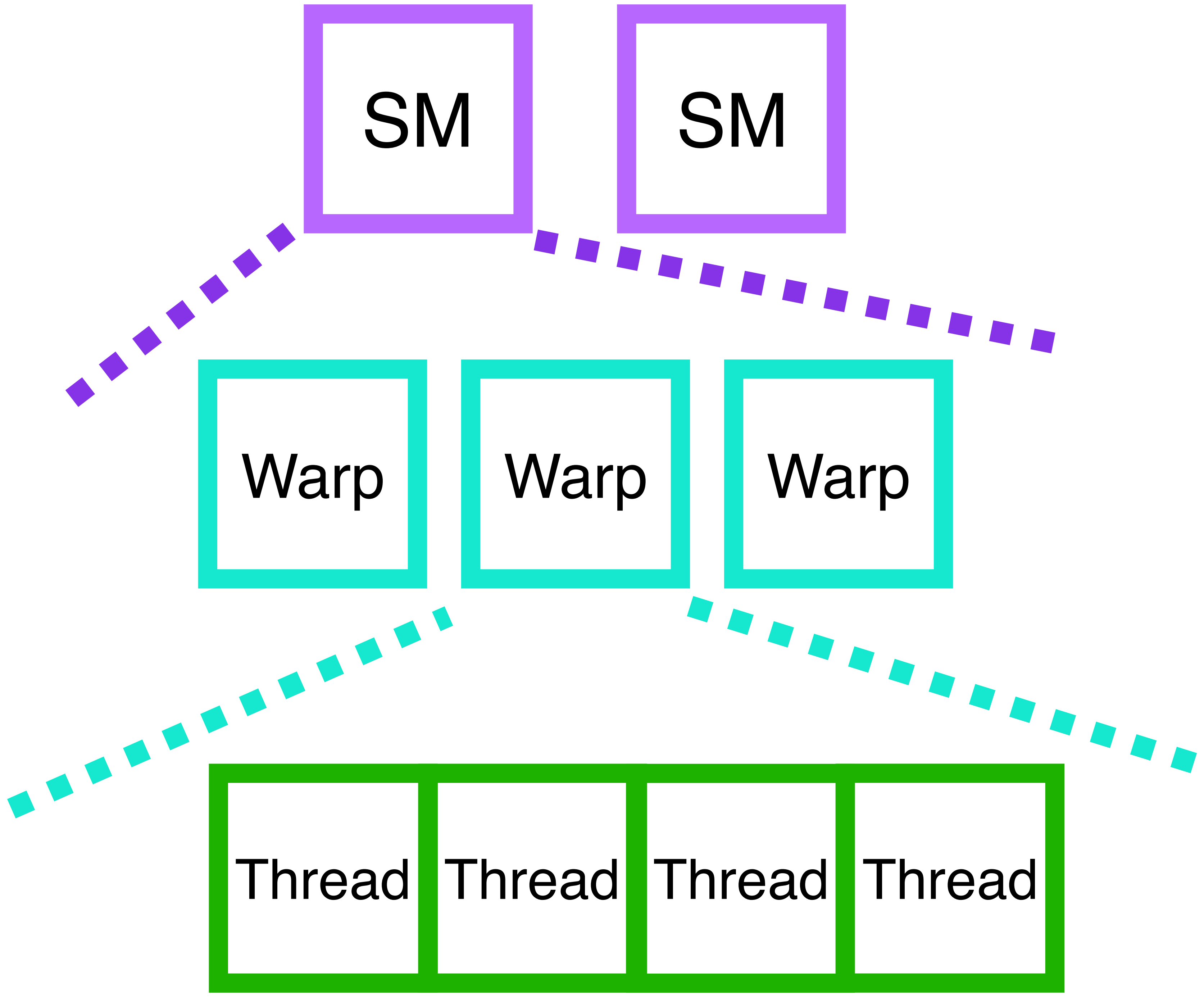


Hierarchy



Hazard!

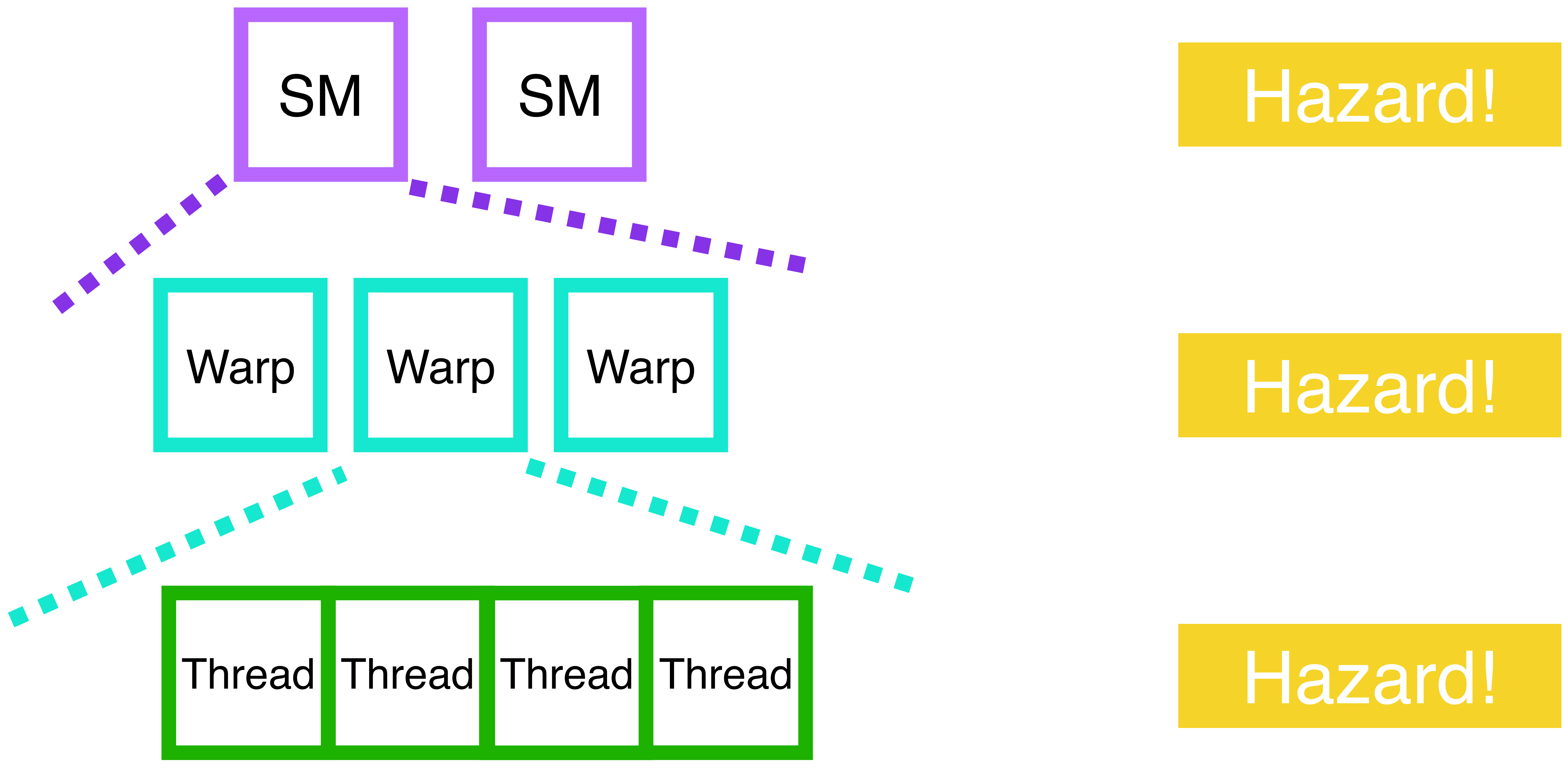
Hierarchy



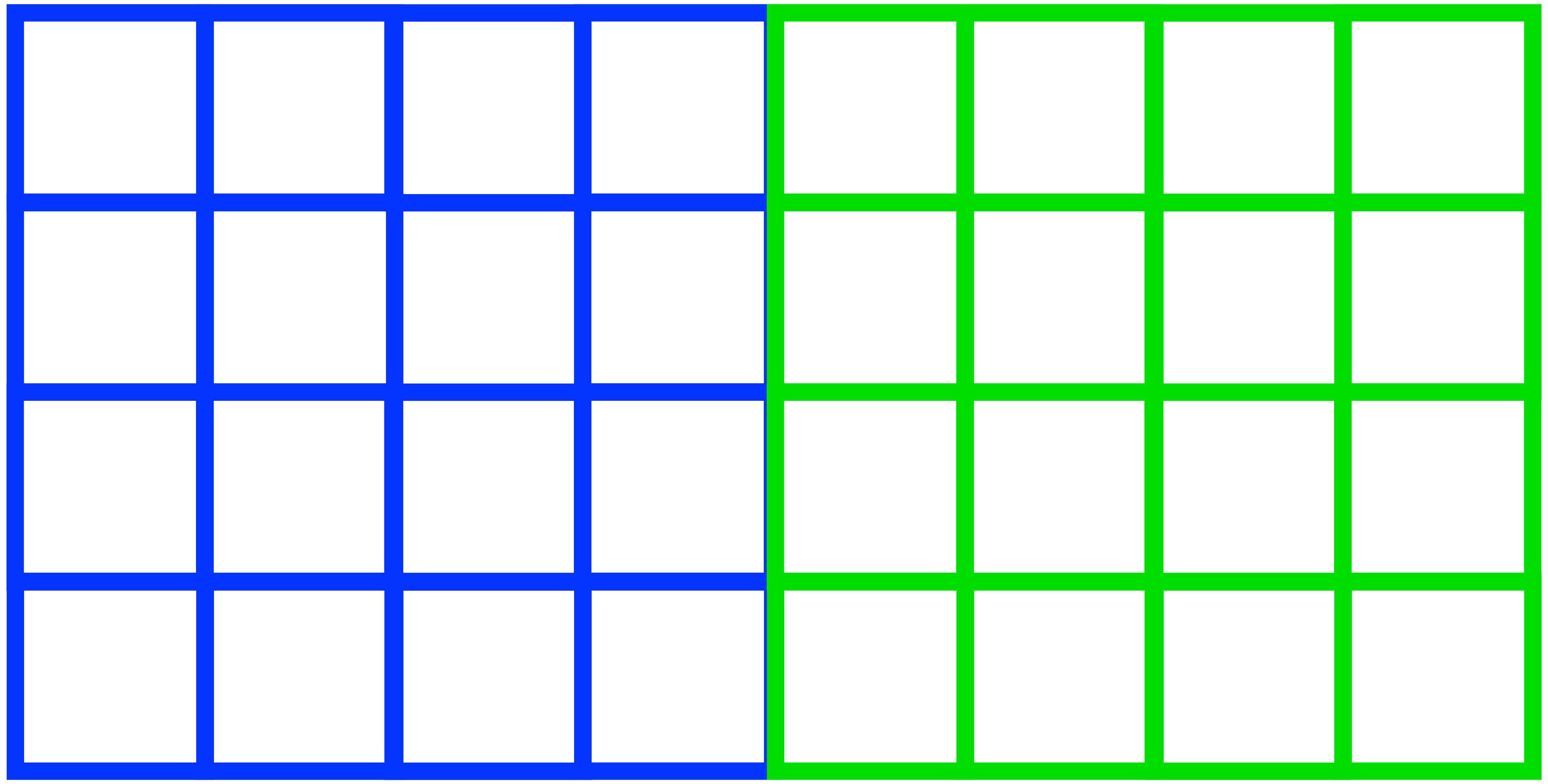
Hazard!

Hazard!

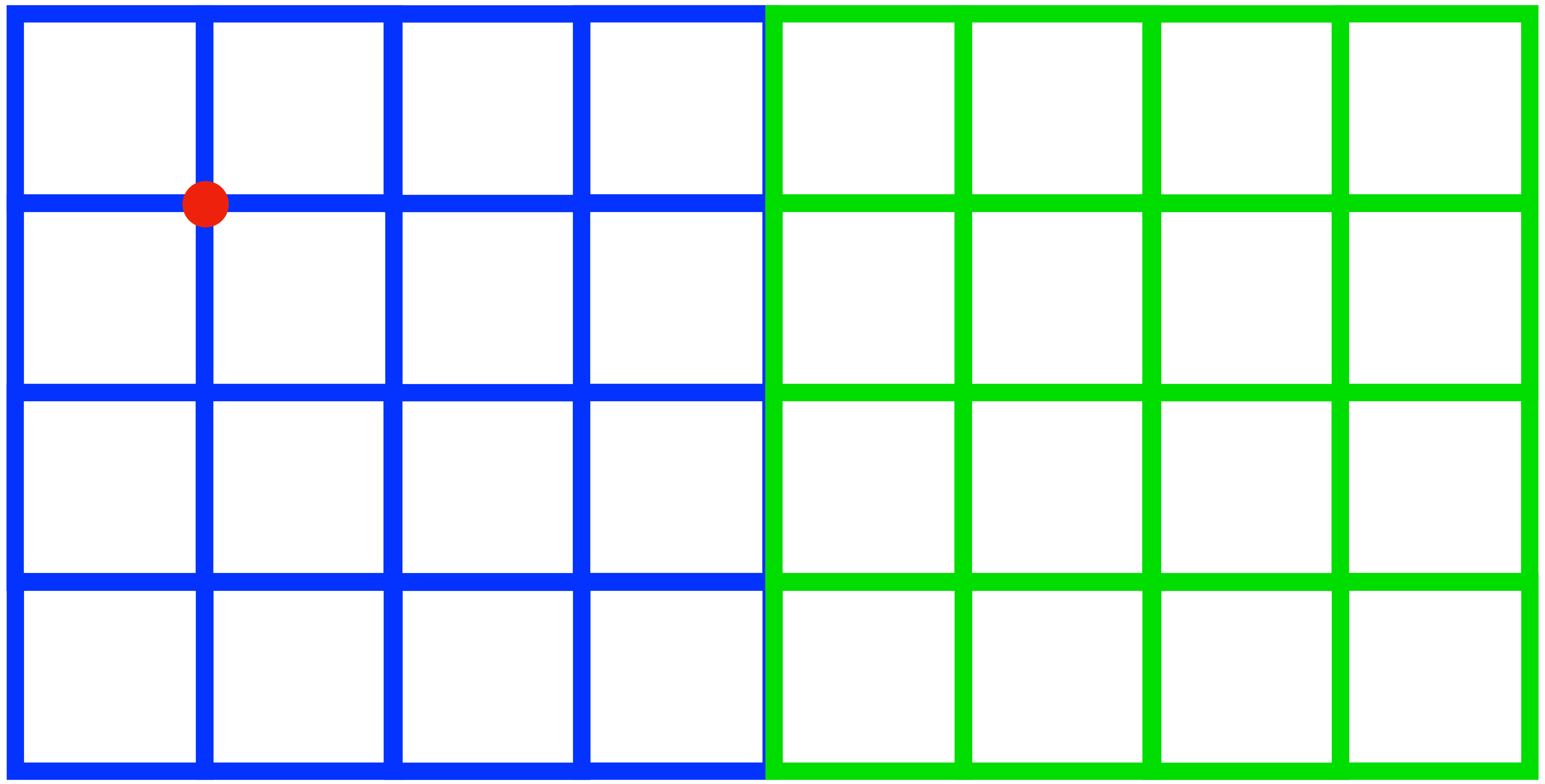
Hierarchy


SM
SM
Warp
Warp
Warp
Thread
Thread
Thread
Thread
Hazard!
Hazard!
Hazard!

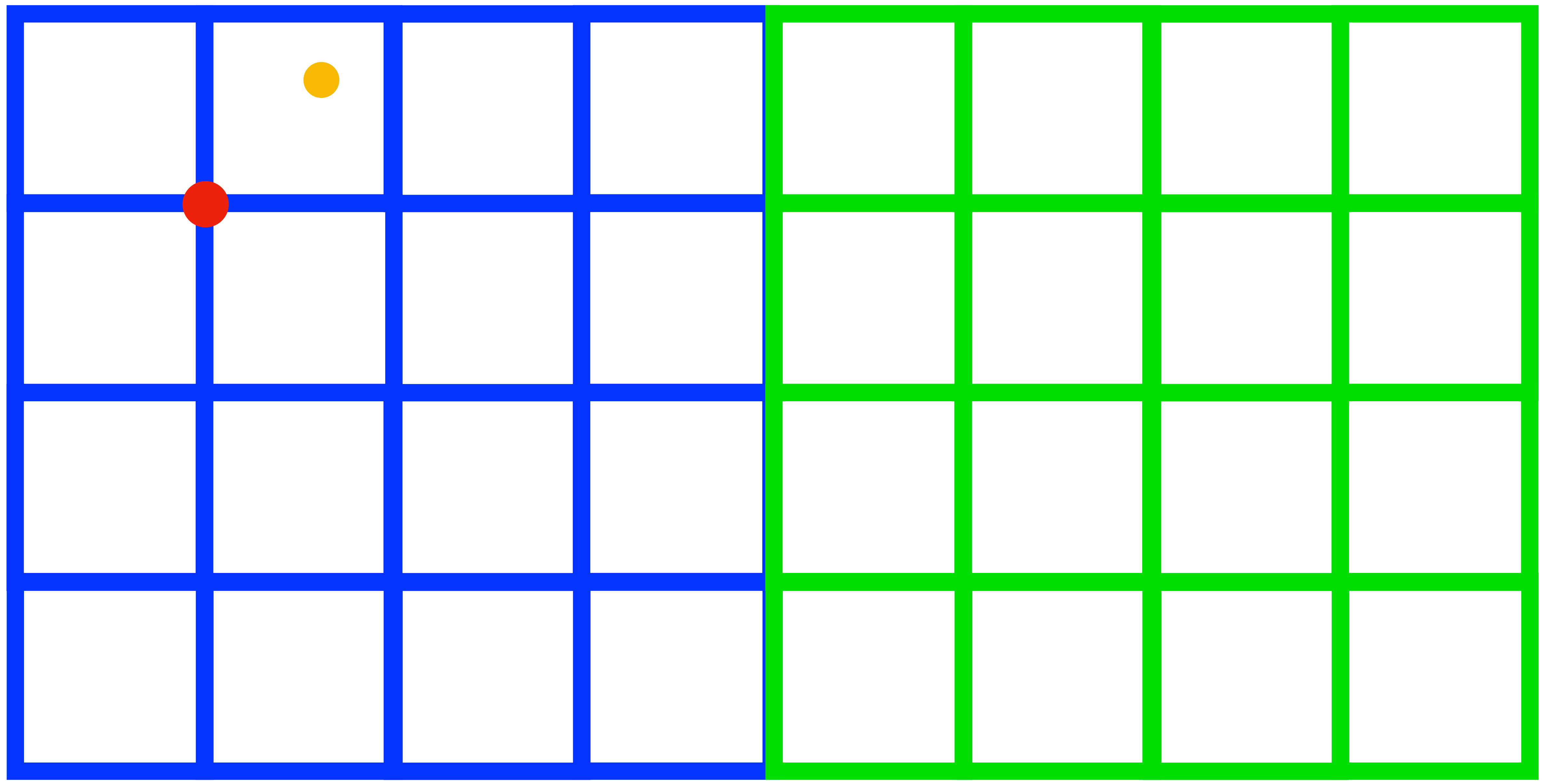
Naive solution



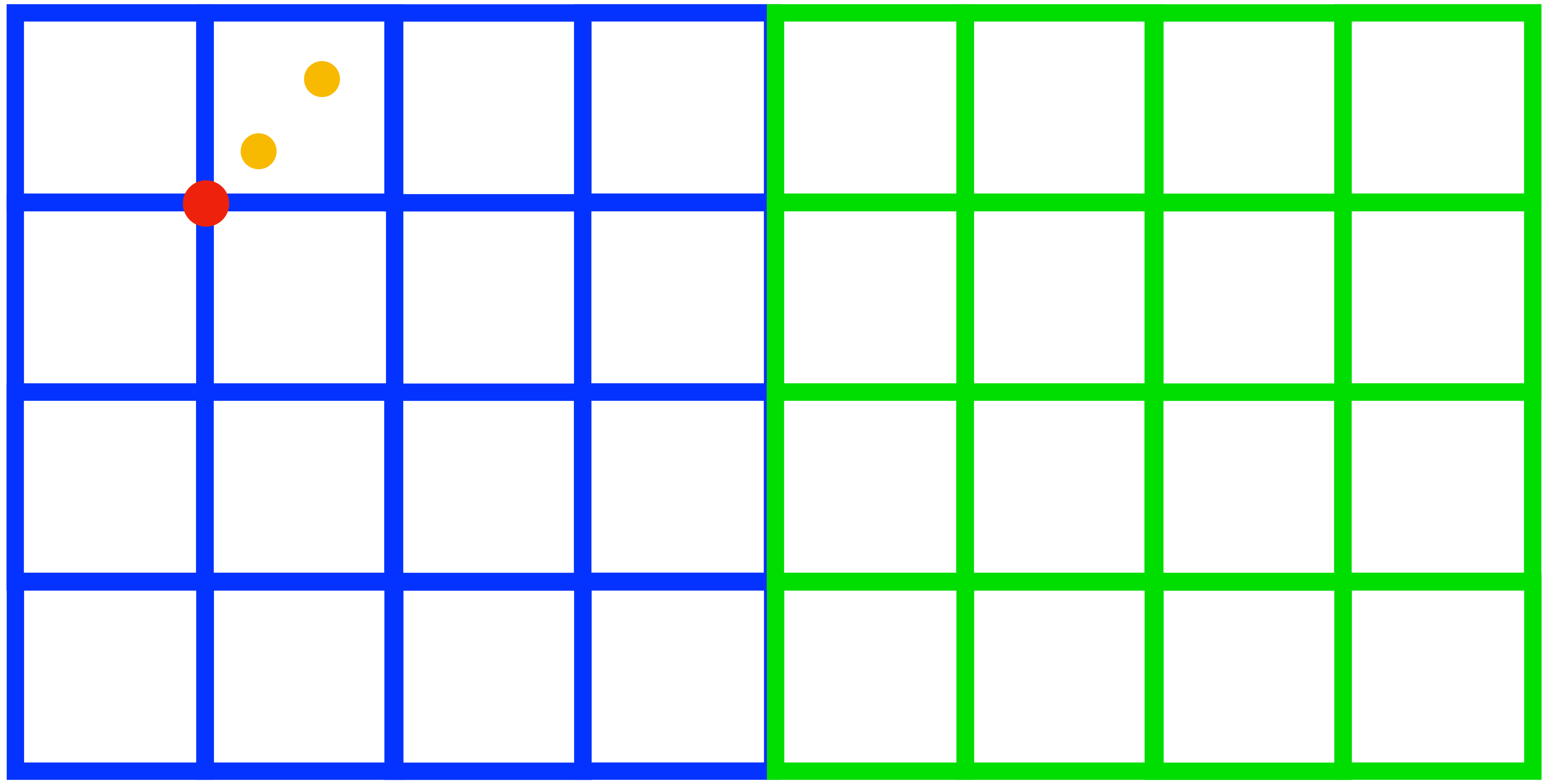
Naive solution



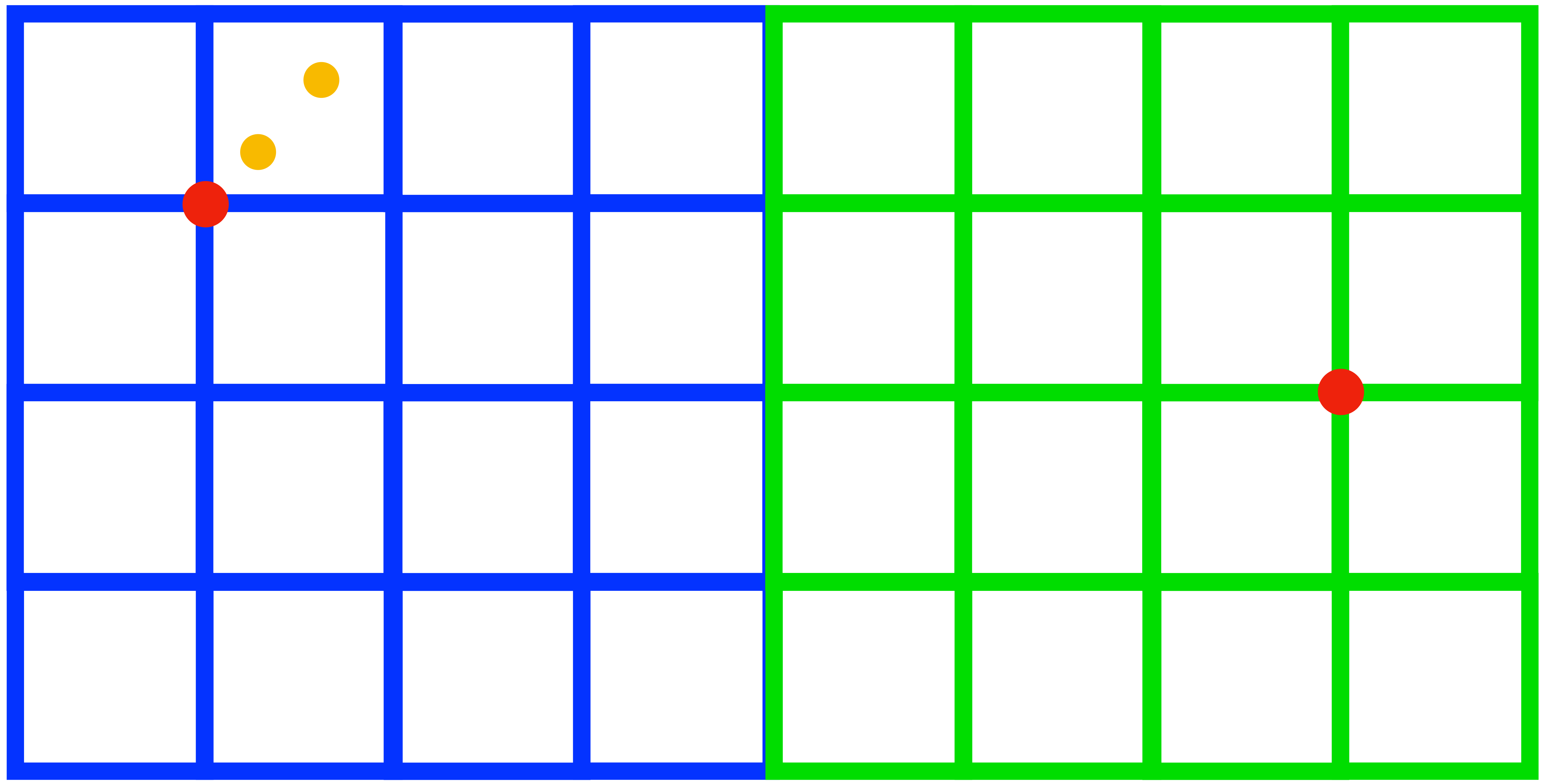
Naive solution



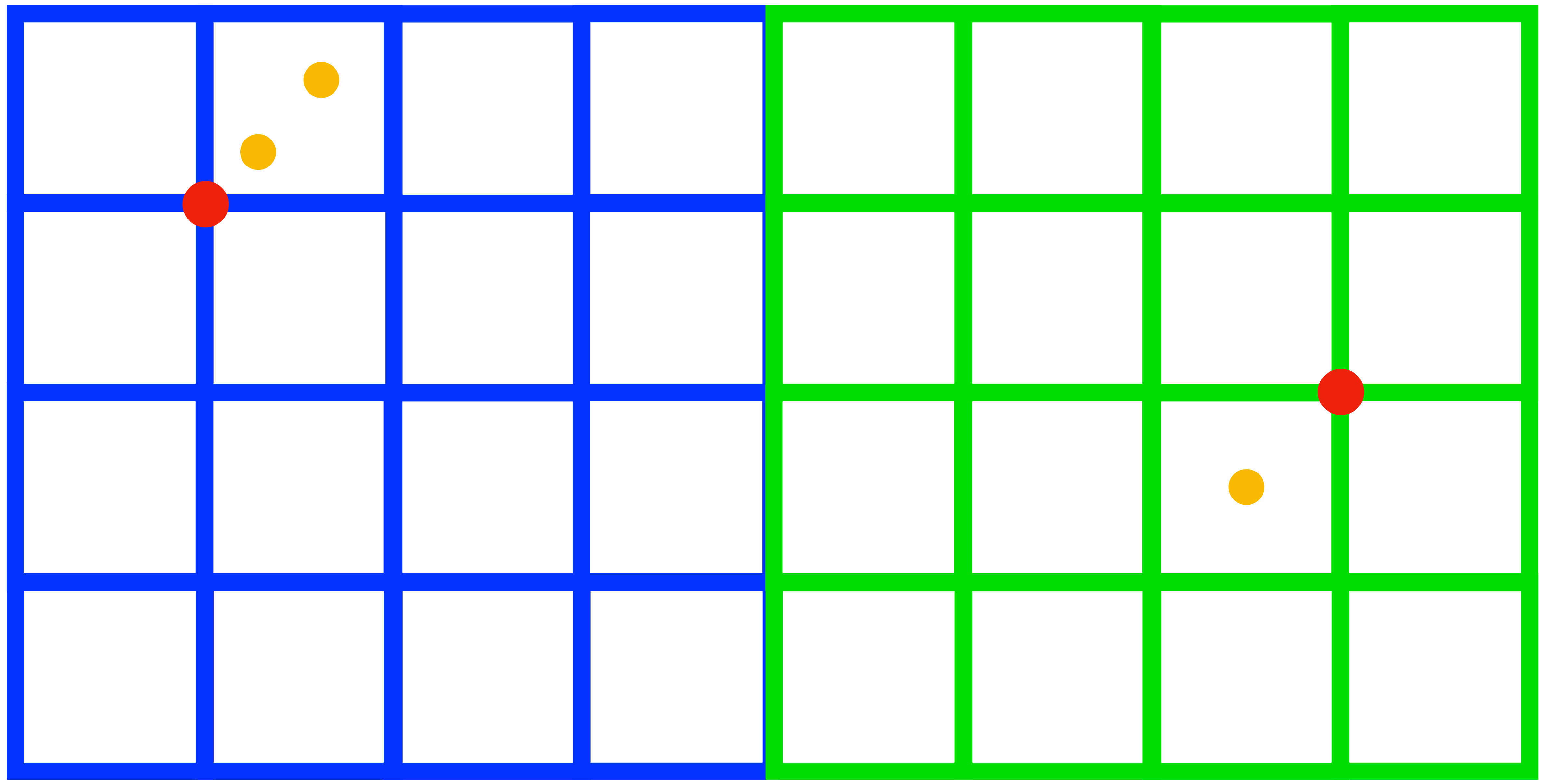
Naive solution



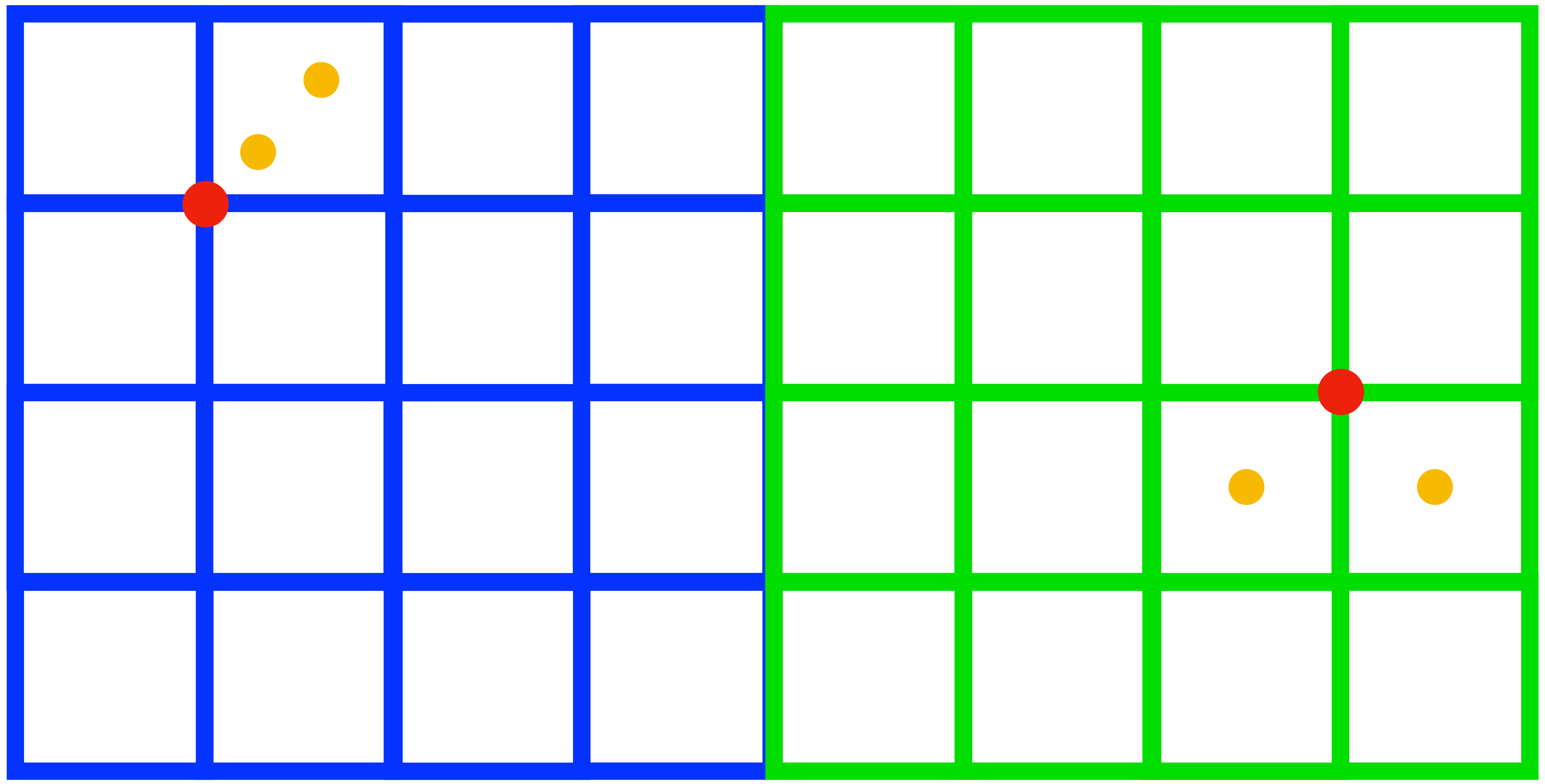
Naive solution



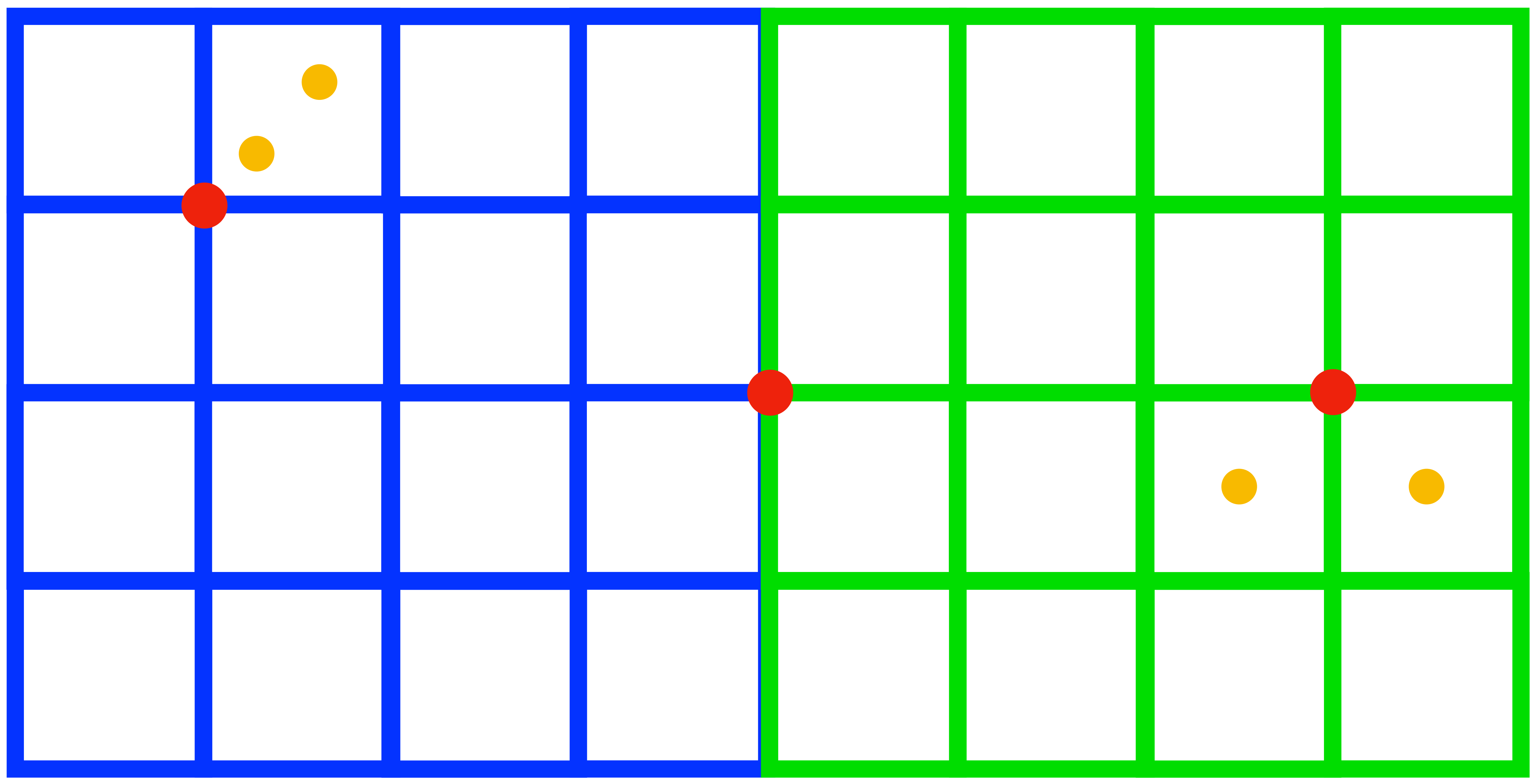
Naive solution



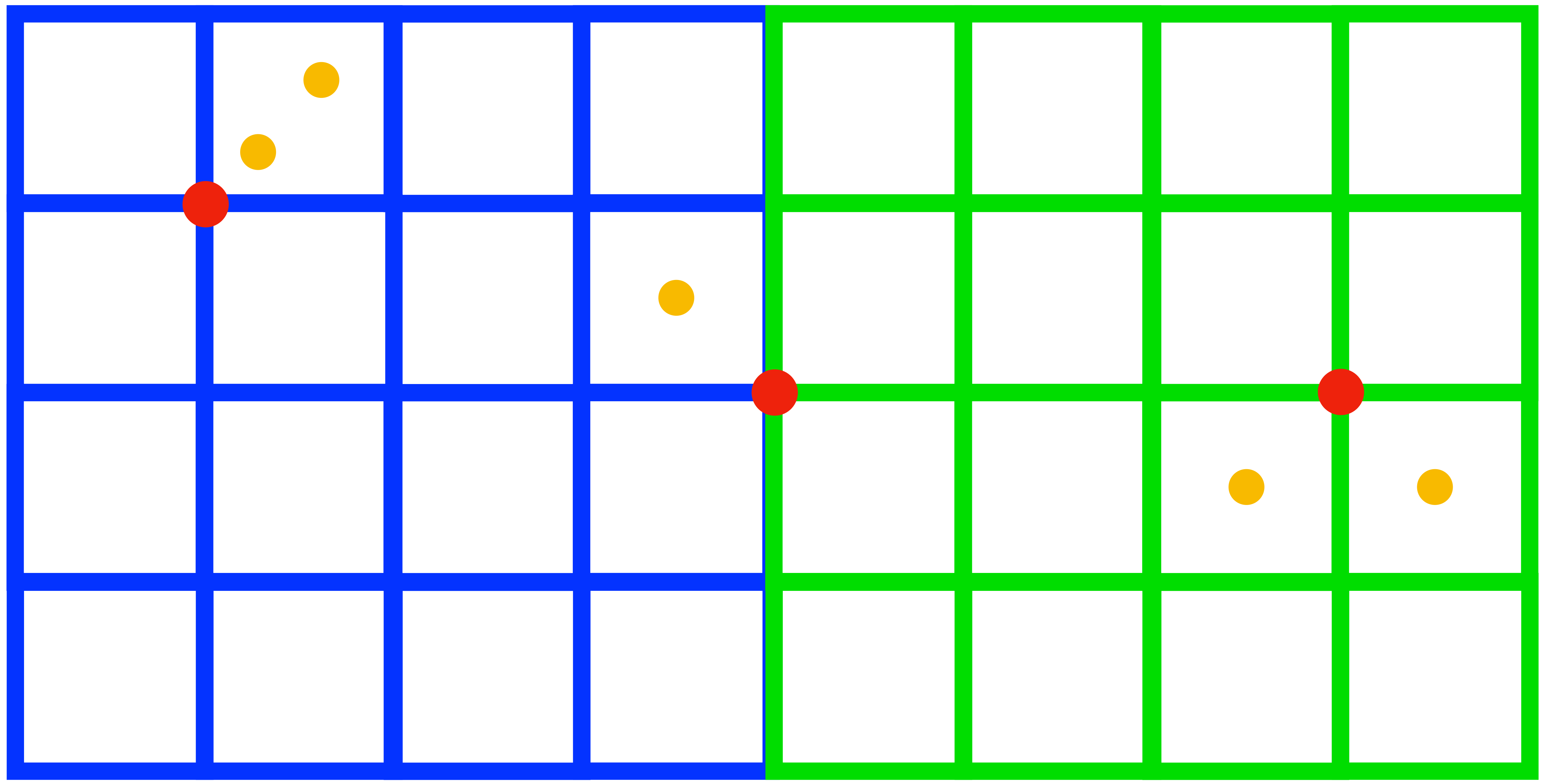
Naive solution



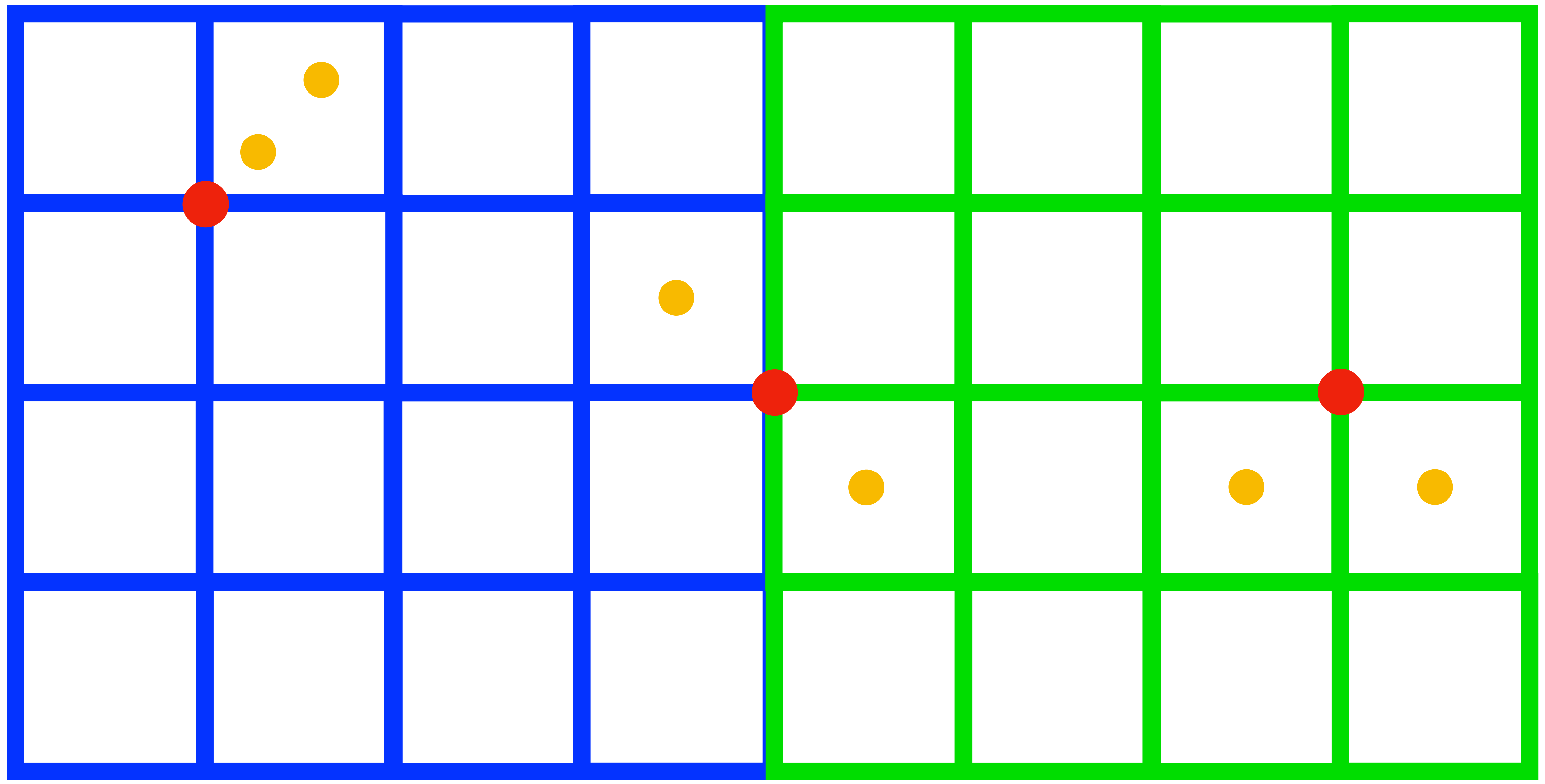
Naive solution



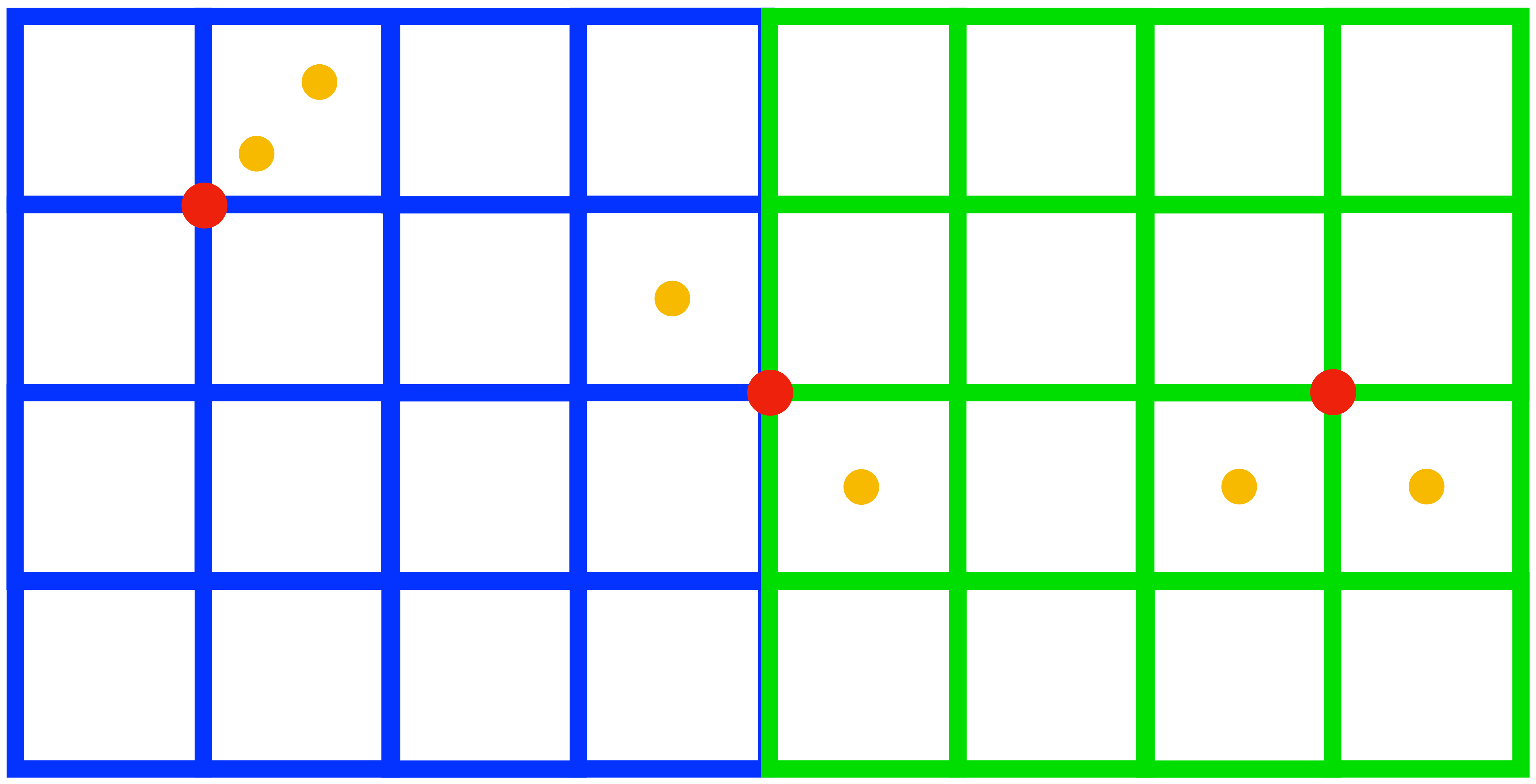
Naive solution



Naive solution

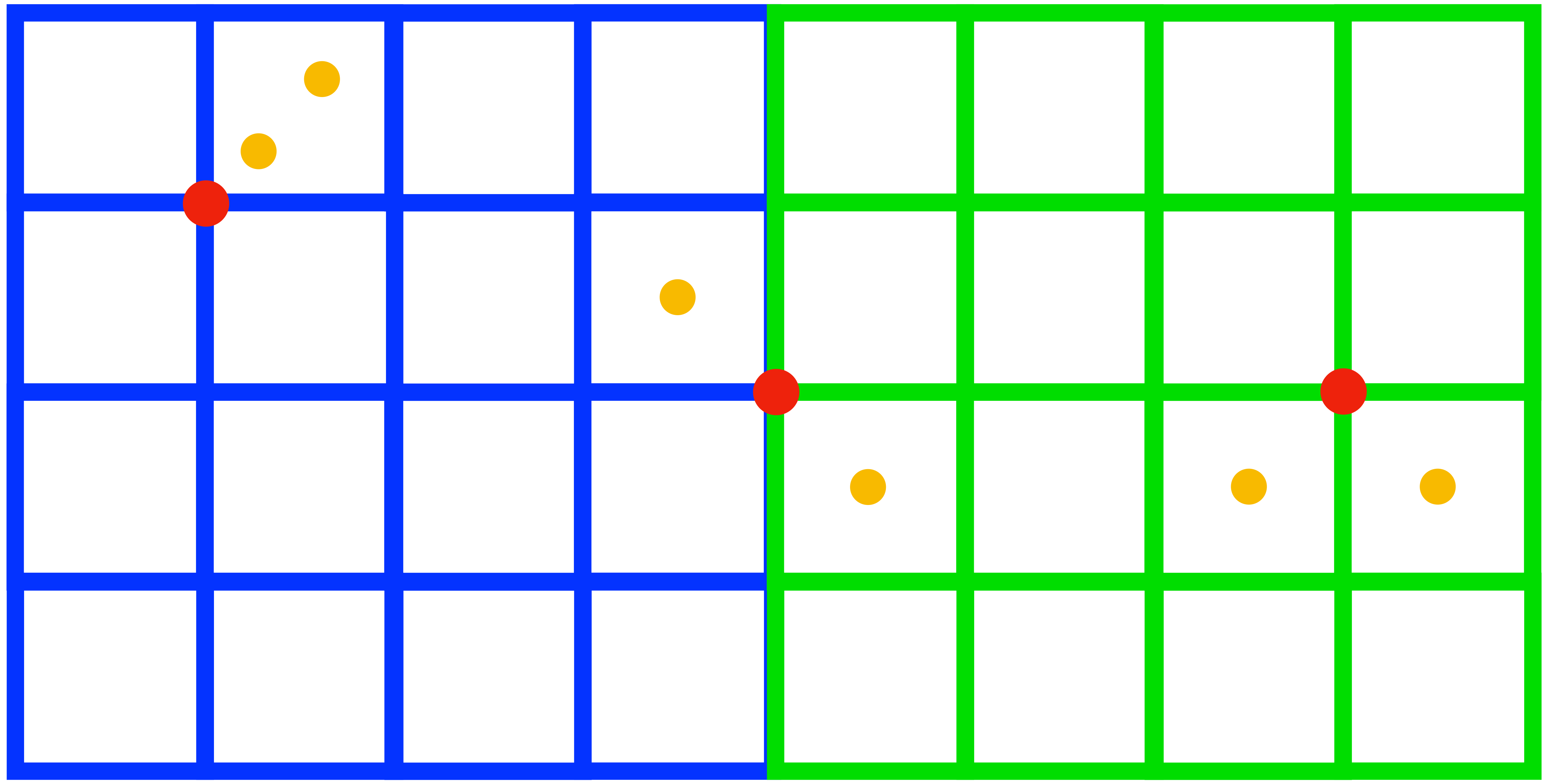


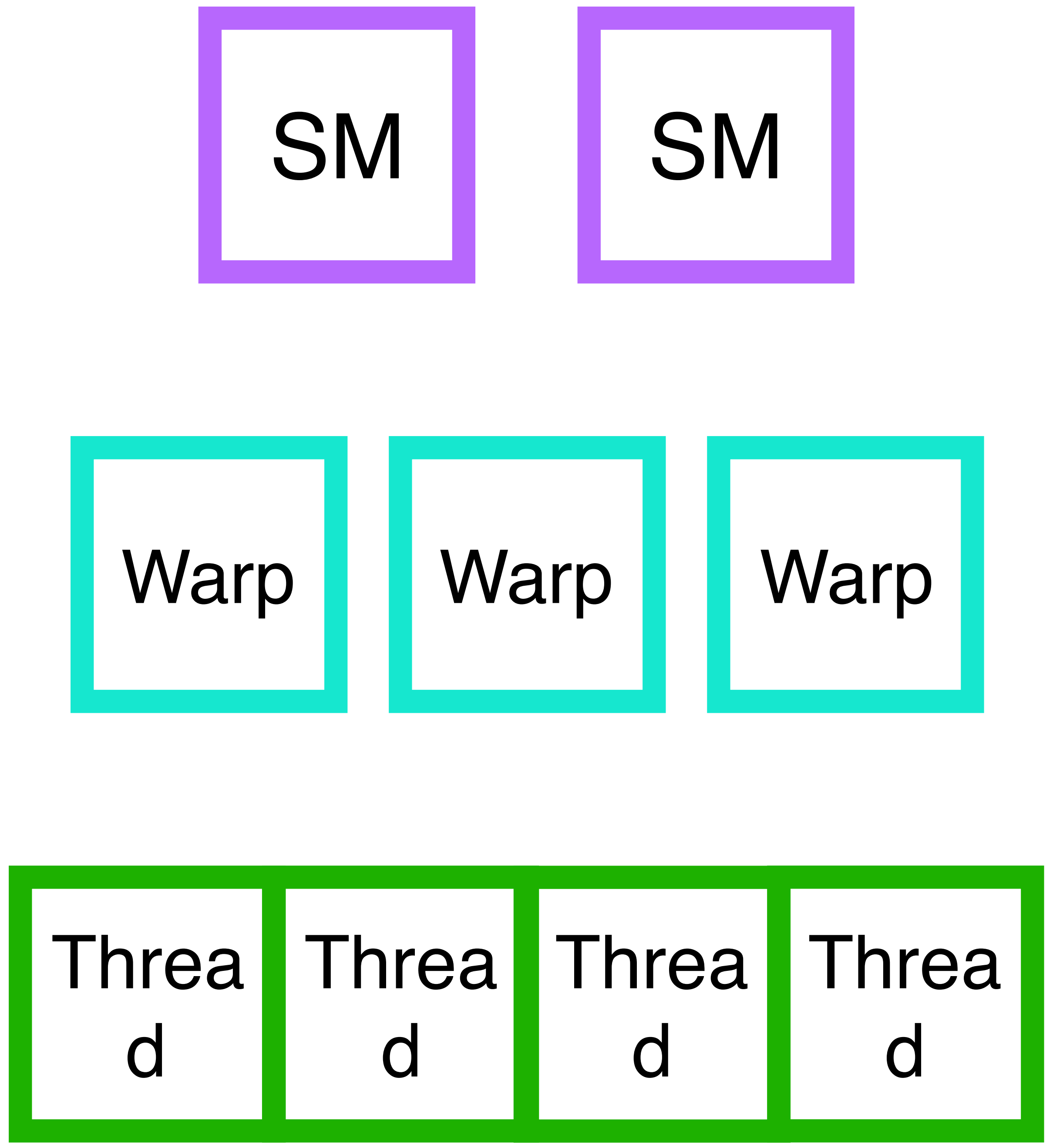
Naive solution



Naive scattering
P2G - 90%

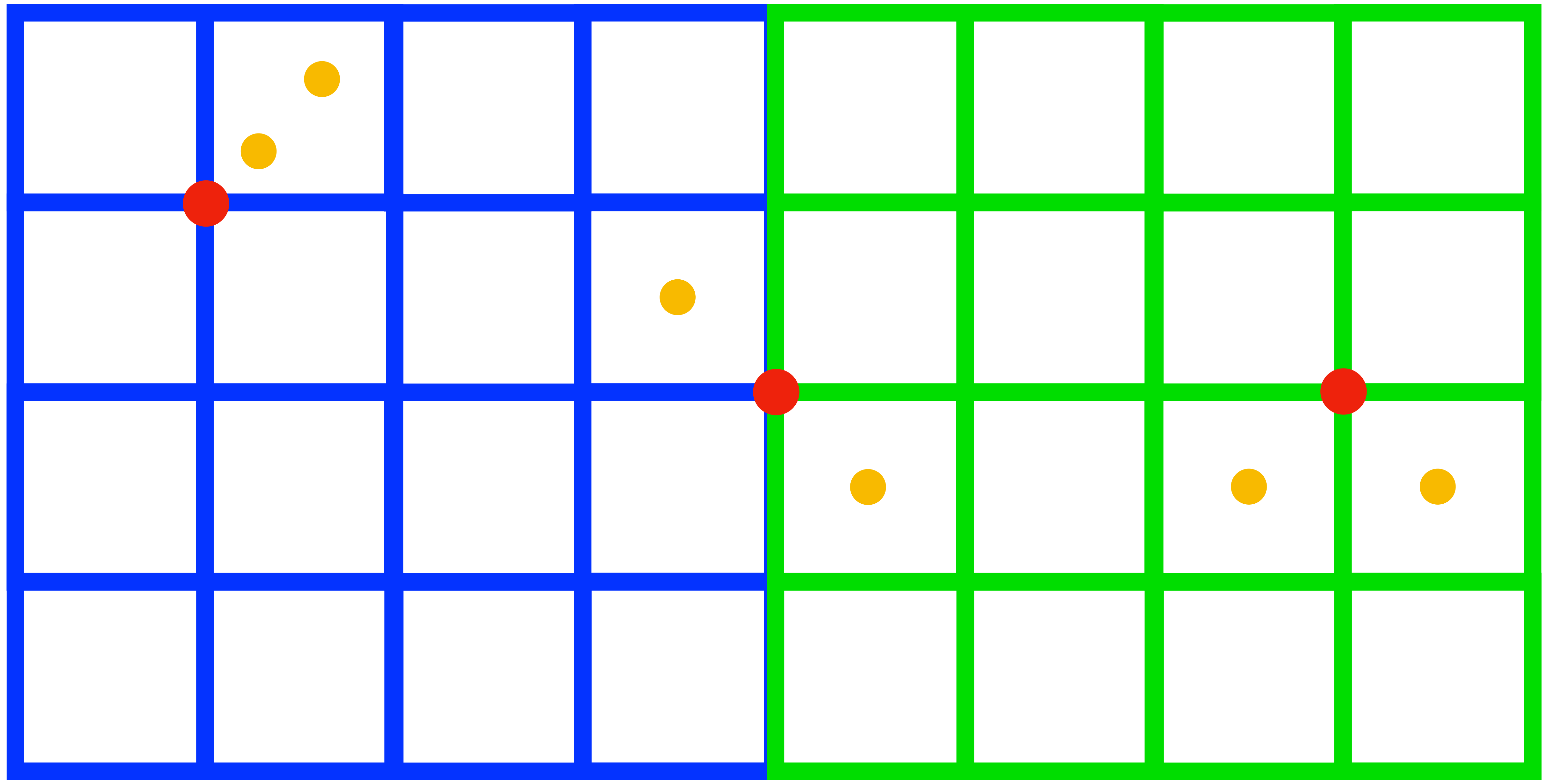
Naive solution



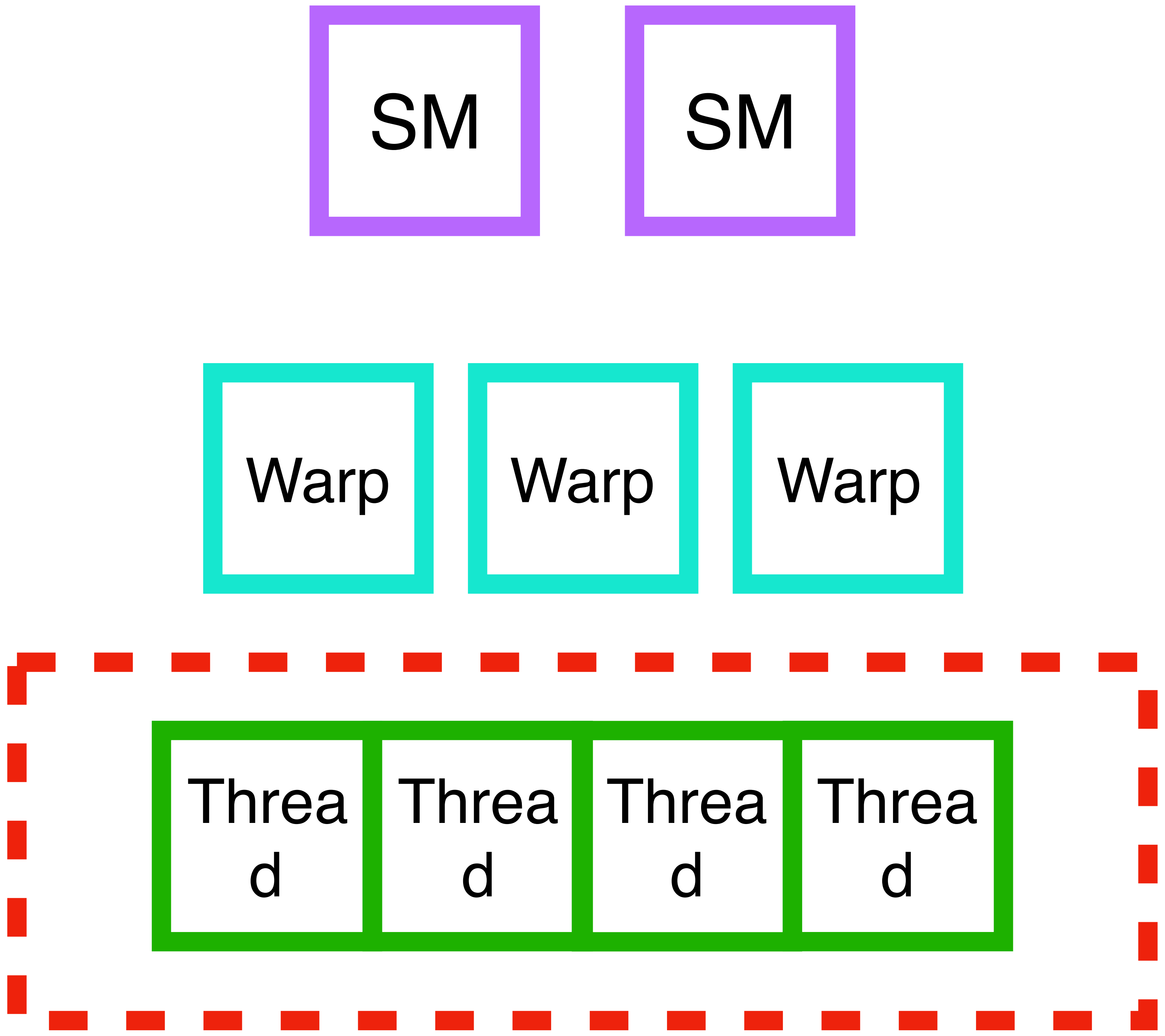


Naive scattering
P2G - 90%

Naive solution



Naive scattering
P2G - 90%



lane id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
node id	n	n+1	n+1	n+1	n+1	n+2	n+2	n+3

lane id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
node id	n	n+1	n+1	n+1	n+1	n+2	n+2	n+3

lane id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
node id	n	n+1	n+1	n+1	n+1	n+2	n+2	n+3

lane id	0	1	2	3	4	5	6	7
node id	n	n+1	n+1	n+1	n+1	n+2	n+2	n+3

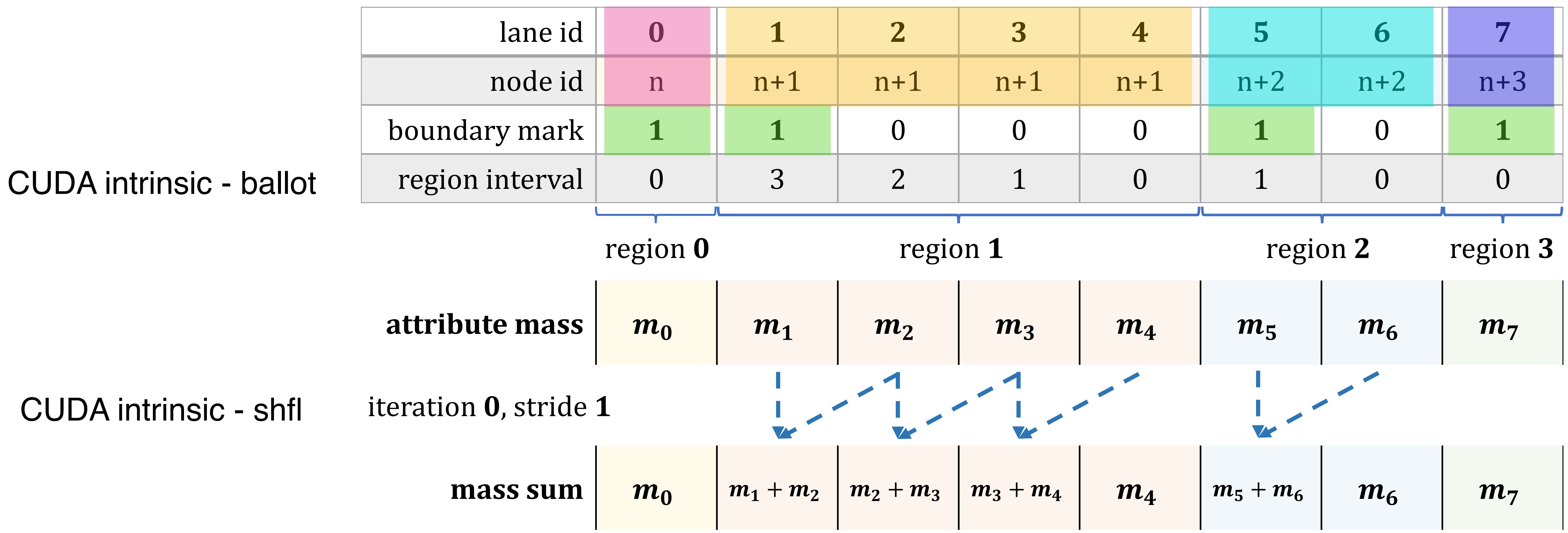
lane id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
node id	n	n+1	n+1	n+1	n+1	n+2	n+2	n+3

lane id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
node id	n	n+1	n+1	n+1	n+1	n+2	n+2	n+3
boundary mark	**1**	**1**	0	0	0	**1**	0	**1**

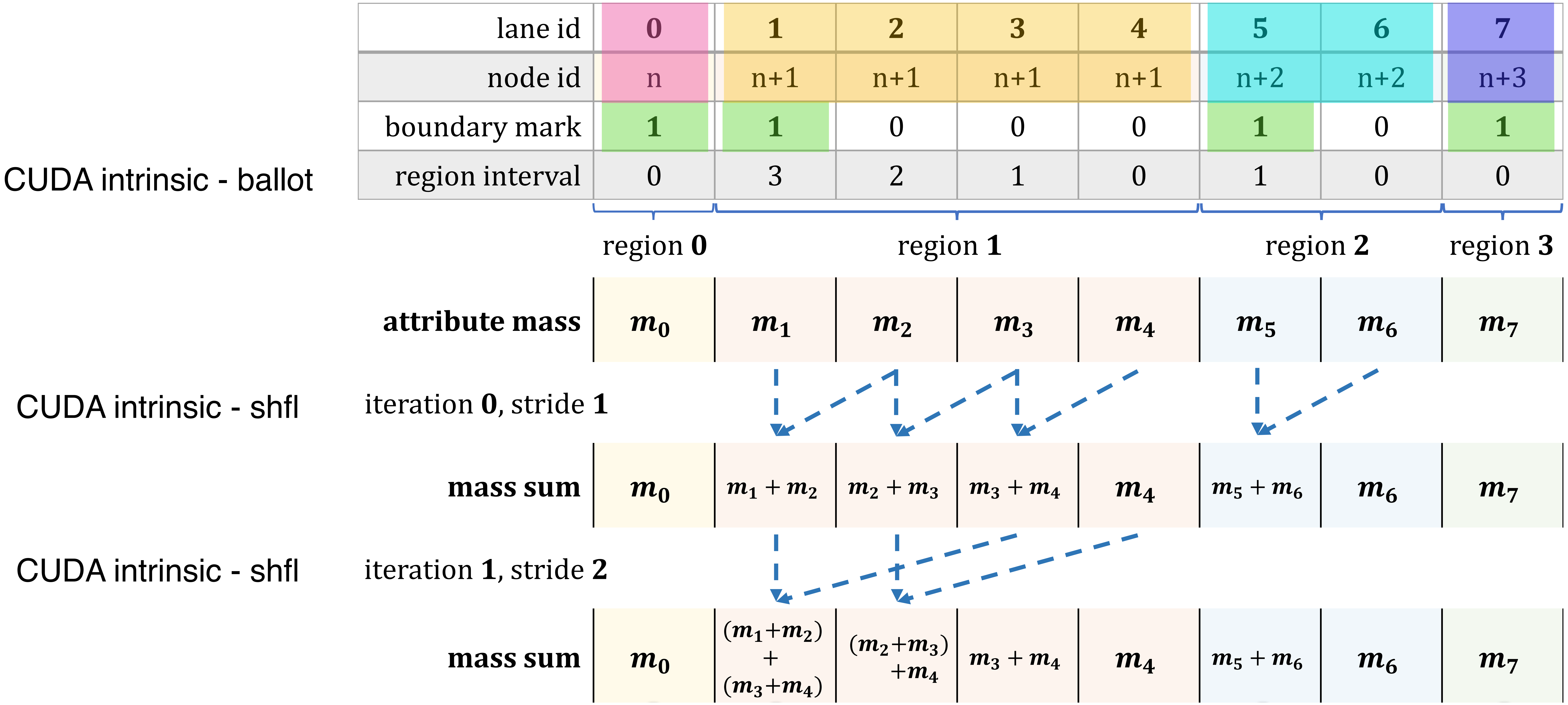
CUDA intrinsic - ballot

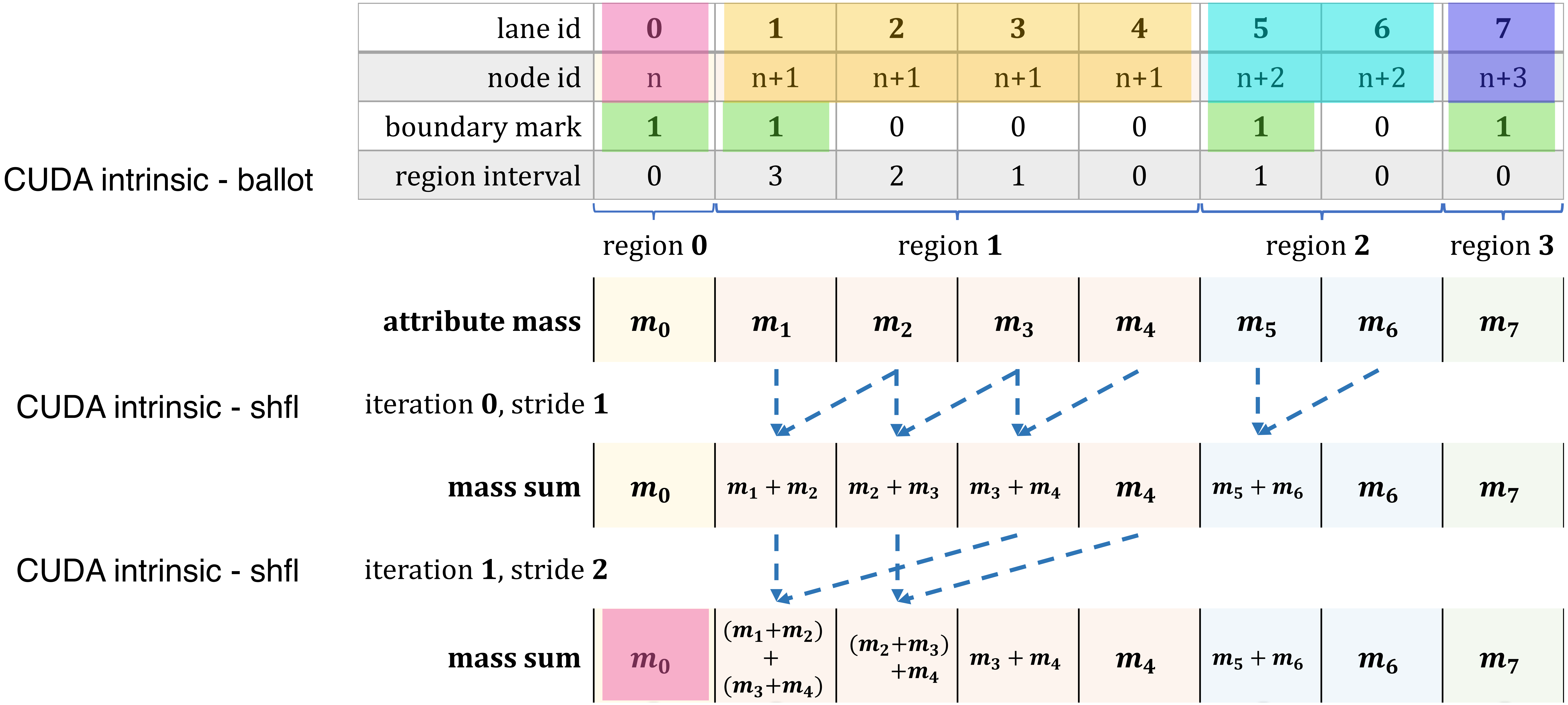
lane id	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
node id	n	n+1	n+1	n+1	n+1	n+2	n+2	n+3
boundary mark	**1**	**1**	0	0	0	**1**	0	**1**
region interval	0	3	2	1	0	1	0	0

region **0** (lane 0); region **1** (lanes 1–4); region **2** (lanes 5–6); region **3** (lane 7)

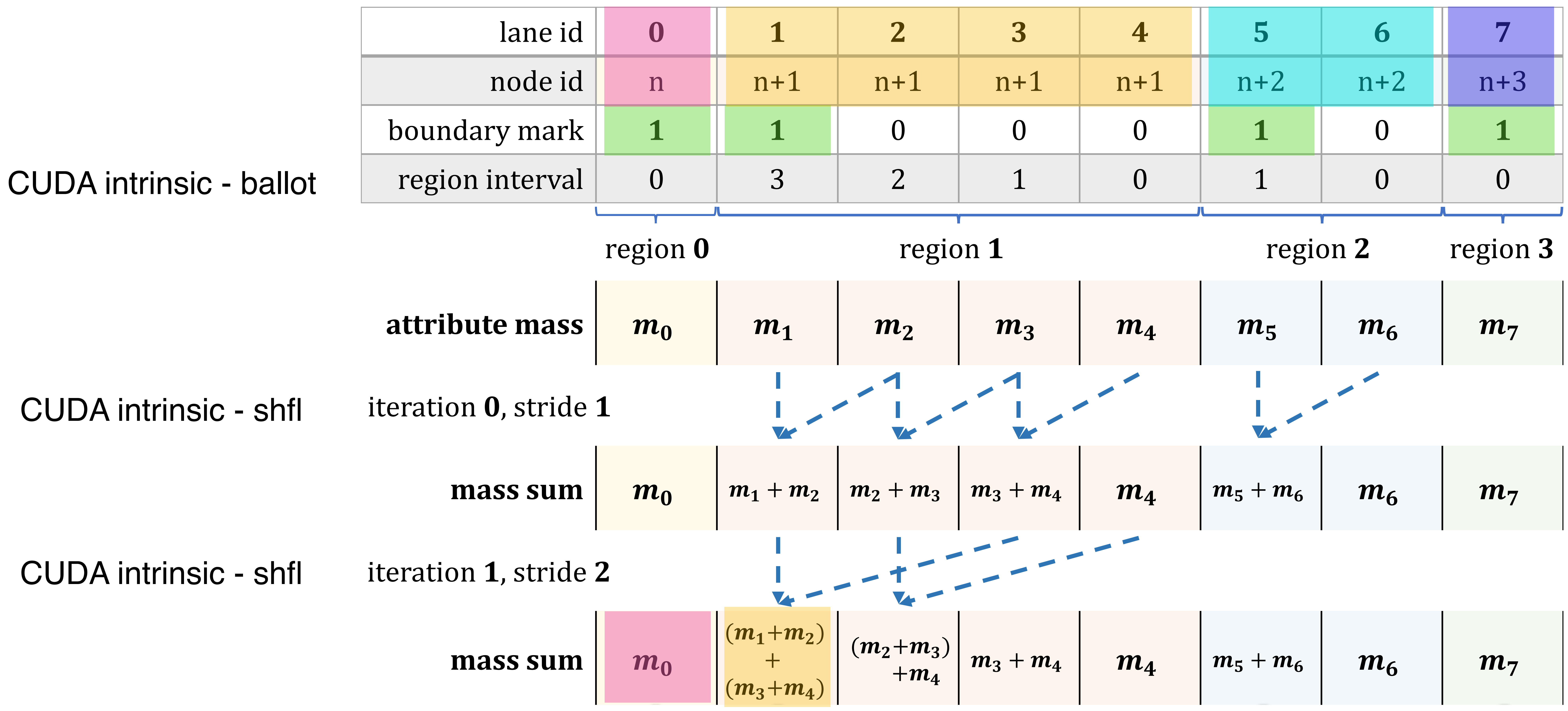


CUDA intrinsic - ballot
lane id
node id
boundary mark
region interval
0
1
2
3
4
5
6
7
n
n+1
n+1
n+1
n+1
n+2
n+2
n+3
1
1
0
0
0
1
0
1
0
3
2
1
0
1
0
0
region 0
region 1
region 2
region 3
attribute mass
m_0
m_1
m_2
m_3
m_4
m_5
m_6
m_7
CUDA intrinsic - shfl
iteration 0, stride 1
mass sum
m_0
$m_1 + m_2$
$m_2 + m_3$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7

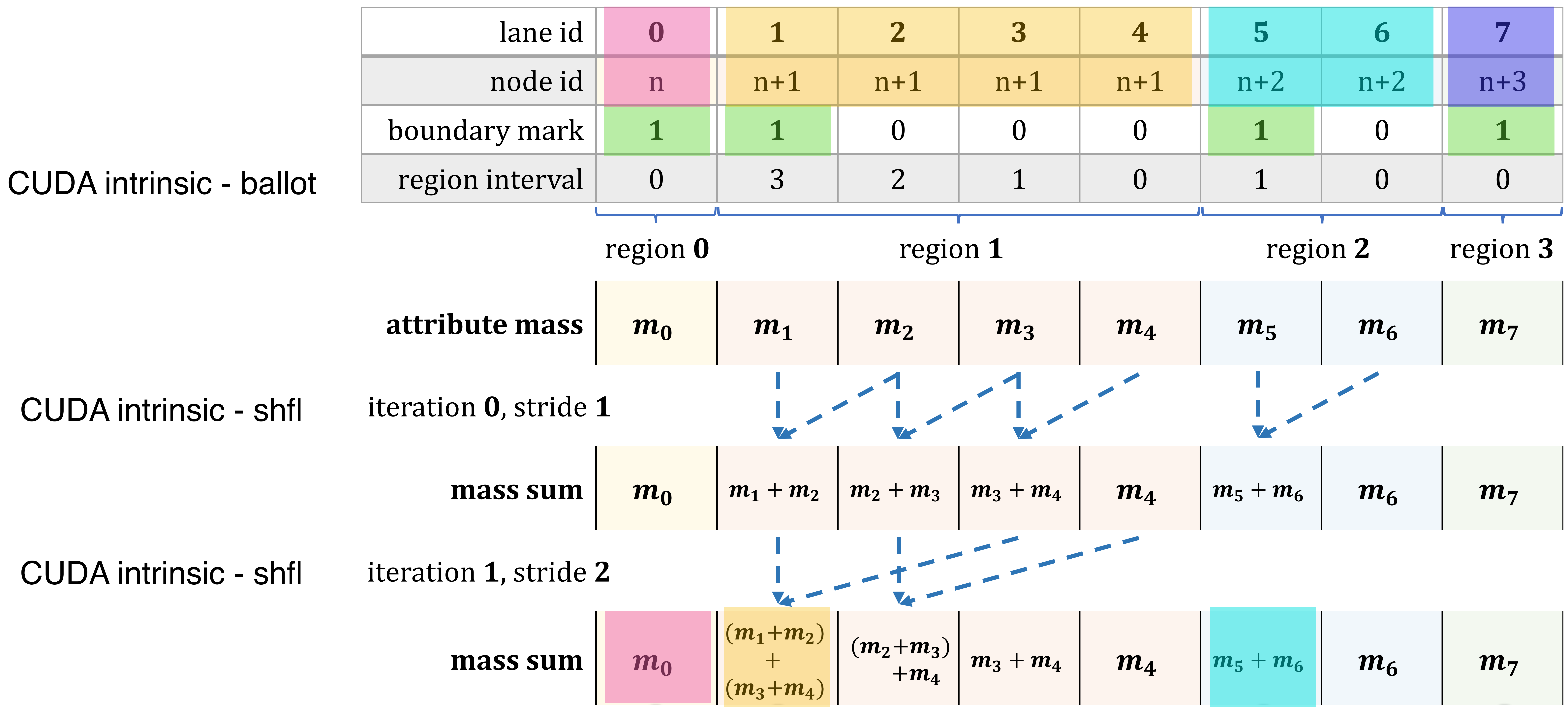

CUDA intrinsic - ballot
lane id
0
1
2
3
4
5
6
7
node id
n
n+1
n+1
n+1
n+1
n+2
n+2
n+3
boundary mark
1
1
0
0
0
1
0
1
region interval
0
3
2
1
0
1
0
0
region 0
region 1
region 2
region 3
attribute mass
m_0
m_1
m_2
m_3
m_4
m_5
m_6
m_7
CUDA intrinsic - shfl
iteration 0, stride 1
mass sum
m_0
$m_1 + m_2$
$m_2 + m_3$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7
CUDA intrinsic - shfl
iteration 1, stride 2
mass sum
m_0
$(m_1+m_2) + (m_3+m_4)$
$(m_2+m_3) +m_4$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7



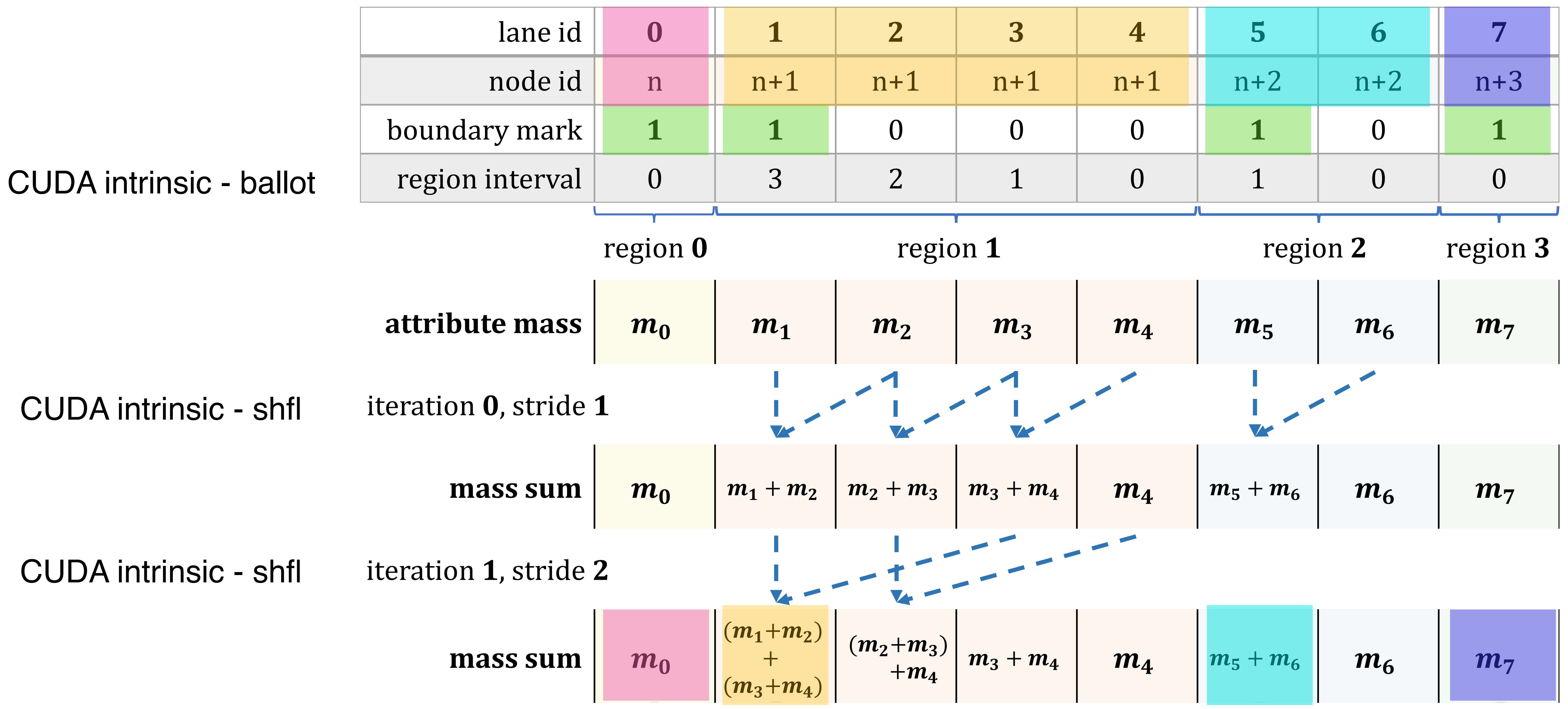
CUDA intrinsic - ballot
lane id
0
1
2
3
4
5
6
7
node id
n
n+1
n+1
n+1
n+1
n+2
n+2
n+3
boundary mark
1
1
0
0
0
1
0
1
region interval
0
3
2
1
0
1
0
0
region 0
region 1
region 2
region 3
attribute mass
m_0
m_1
m_2
m_3
m_4
m_5
m_6
m_7
CUDA intrinsic - shfl
iteration 0, stride 1
mass sum
m_0
$m_1 + m_2$
$m_2 + m_3$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7
CUDA intrinsic - shfl
iteration 1, stride 2
mass sum
m_0
$(m_1+m_2) + (m_3+m_4)$
$(m_2+m_3) + m_4$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7



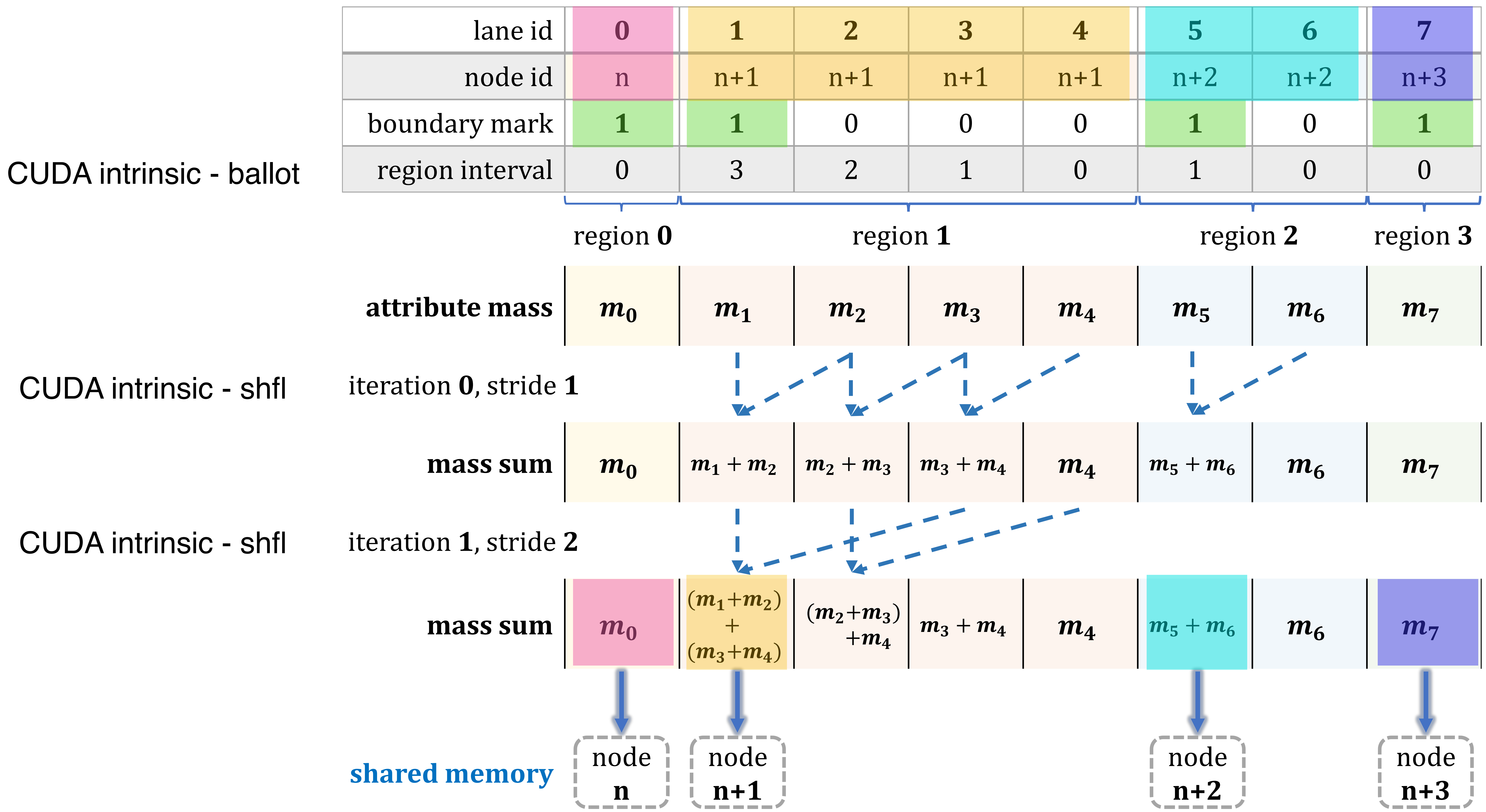
CUDA intrinsic - ballot
lane id
0
1
2
3
4
5
6
7
node id
n
n+1
n+1
n+1
n+1
n+2
n+2
n+3
boundary mark
1
1
0
0
0
1
0
1
region interval
0
3
2
1
0
1
0
0
region 0
region 1
region 2
region 3
attribute mass
m_0
m_1
m_2
m_3
m_4
m_5
m_6
m_7
CUDA intrinsic - shfl
iteration 0, stride 1
mass sum
m_0
$m_1 + m_2$
$m_2 + m_3$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7
CUDA intrinsic - shfl
iteration 1, stride 2
mass sum
m_0
$(m_1+m_2) + (m_3+m_4)$
$(m_2+m_3) +m_4$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7

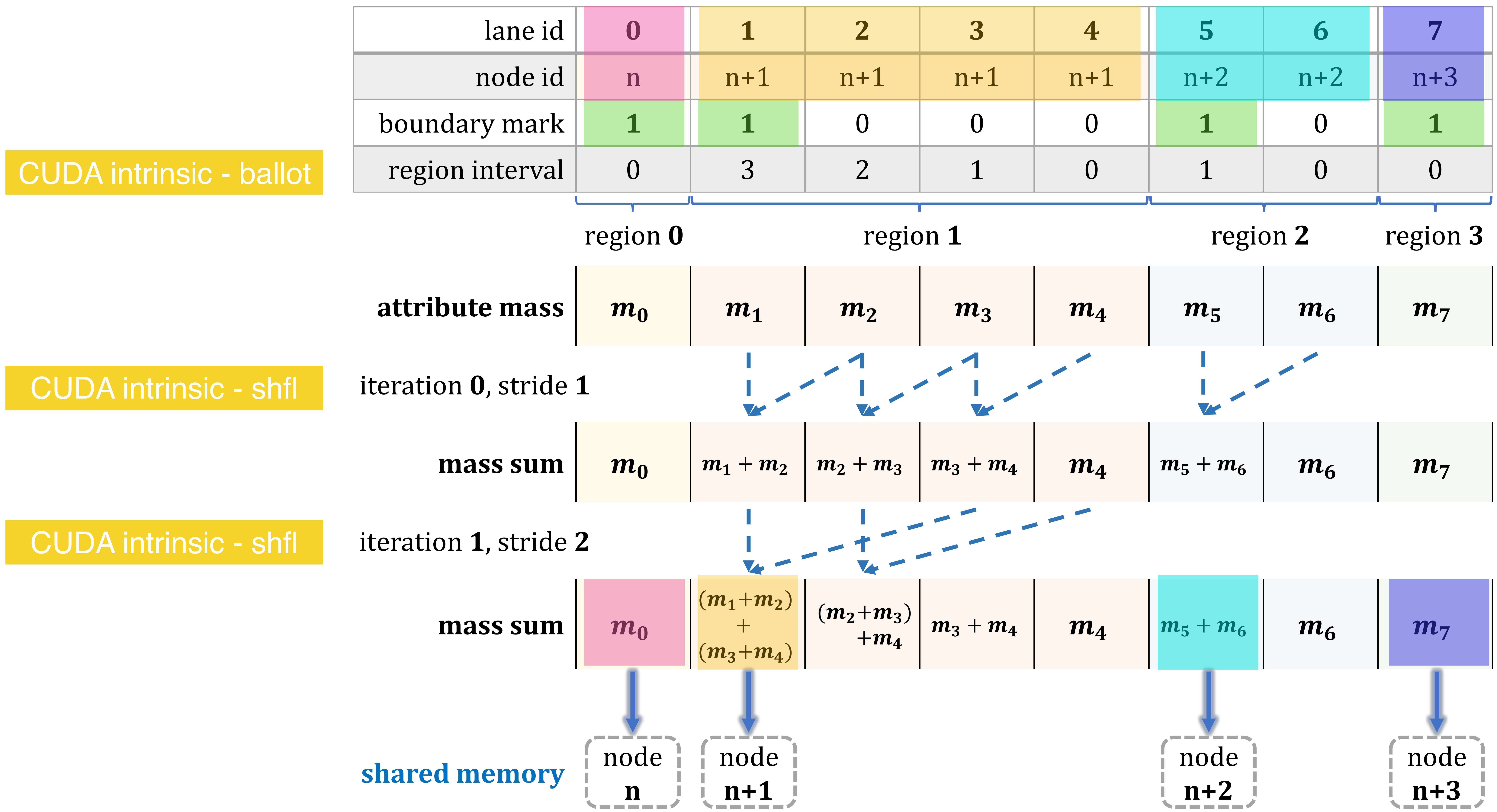


CUDA intrinsic - ballot
lane id
0
1
2
3
4
5
6
7
node id
n
n+1
n+1
n+1
n+1
n+2
n+2
n+3
boundary mark
1
1
0
0
0
1
0
1
region interval
0
3
2
1
0
1
0
0
region 0
region 1
region 2
region 3
attribute mass
m_0
m_1
m_2
m_3
m_4
m_5
m_6
m_7
CUDA intrinsic - shfl
iteration 0, stride 1
mass sum
m_0
$m_1 + m_2$
$m_2 + m_3$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7
CUDA intrinsic - shfl
iteration 1, stride 2
mass sum
m_0
$(m_1+m_2) + (m_3+m_4)$
$(m_2+m_3) + m_4$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7



CUDA intrinsic - ballot
lane id
0
1
2
3
4
5
6
7
node id
n
n+1
n+1
n+1
n+1
n+2
n+2
n+3
boundary mark
1
1
0
0
0
1
0
1
region interval
0
3
2
1
0
1
0
0
region 0
region 1
region 2
region 3
attribute mass
m_0 m_1 m_2 m_3 m_4 m_5 m_6 m_7
CUDA intrinsic - shfl
iteration 0, stride 1
mass sum
m_0 $m_1 + m_2$ $m_2 + m_3$ $m_3 + m_4$ m_4 $m_5 + m_6$ m_6 m_7
CUDA intrinsic - shfl
iteration 1, stride 2
mass sum
m_0 $(m_1+m_2) + (m_3+m_4)$ $(m_2+m_3) + m_4$ $m_3 + m_4$ m_4 $m_5 + m_6$ m_6 m_7

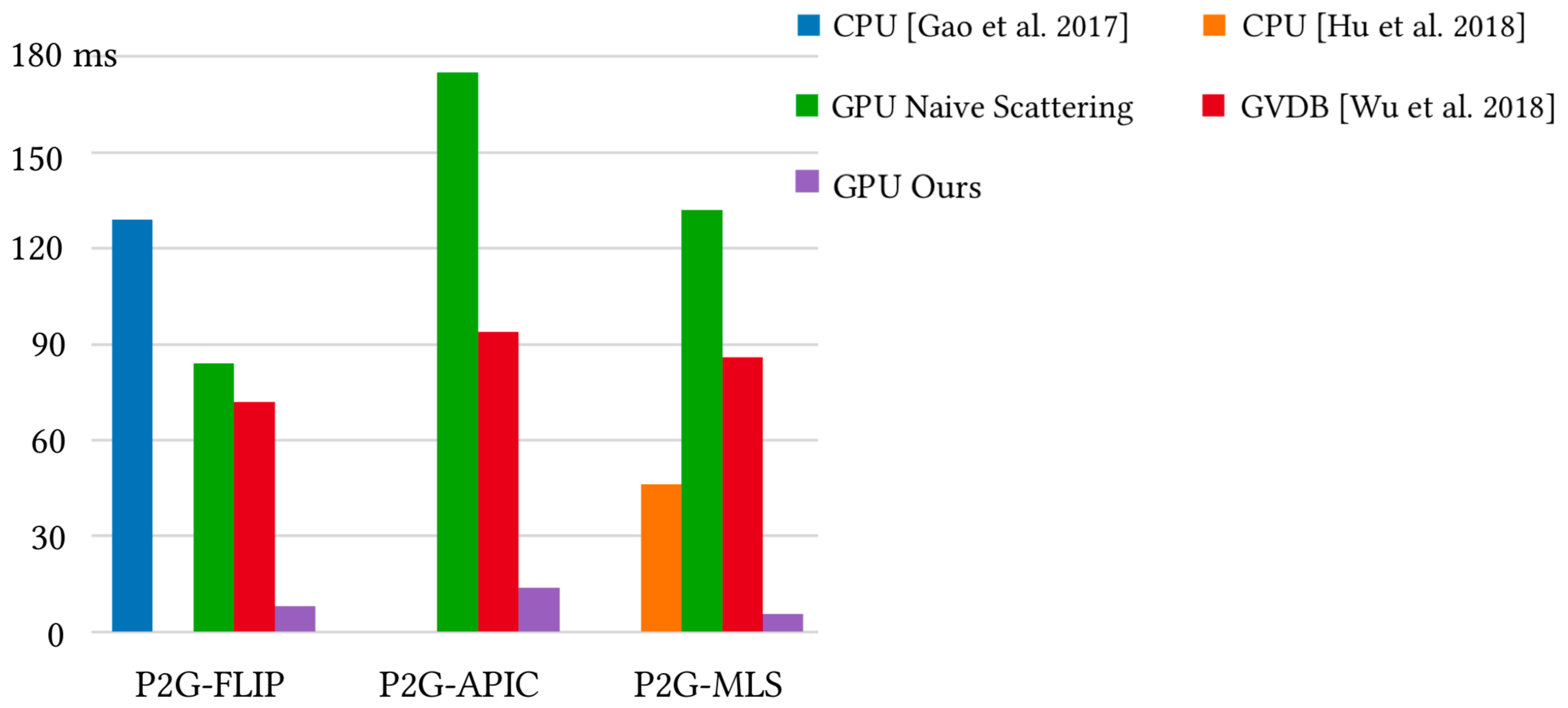

CUDA intrinsic - ballot
lane id
0
1
2
3
4
5
6
7
node id
n
n+1
n+1
n+1
n+1
n+2
n+2
n+3
boundary mark
1
1
0
0
0
1
0
1
region interval
0
3
2
1
0
1
0
0
region 0
region 1
region 2
region 3
attribute mass
m_0
m_1
m_2
m_3
m_4
m_5
m_6
m_7
CUDA intrinsic - shfl
iteration 0, stride 1
mass sum
m_0
$m_1 + m_2$
$m_2 + m_3$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7
CUDA intrinsic - shfl
iteration 1, stride 2
mass sum
m_0
$(m_1+m_2) + (m_3+m_4)$
$(m_2+m_3) + m_4$
$m_3 + m_4$
m_4
$m_5 + m_6$
m_6
m_7
shared memory
node n
node n+1
node n+2
node n+3



CUDA intrinsic - ballot
CUDA intrinsic - shfl
CUDA intrinsic - shfl
lane id
0 1 2 3 4 5 6 7
node id
n n+1 n+1 n+1 n+1 n+2 n+2 n+3
boundary mark
1 1 0 0 0 1 0 1
region interval
0 3 2 1 0 1 0 0
region 0
region 1
region 2
region 3
attribute mass
m_0 m_1 m_2 m_3 m_4 m_5 m_6 m_7
iteration 0, stride 1
mass sum
m_0 $m_1 + m_2$ $m_2 + m_3$ $m_3 + m_4$ m_4 $m_5 + m_6$ m_6 m_7
iteration 1, stride 2
mass sum
m_0 $(m_1+m_2) + (m_3+m_4)$ $(m_2+m_3) + m_4$ $m_3 + m_4$ m_4 $m_5 + m_6$ m_6 m_7
shared memory
node n
node n+1
node n+2
node n+3

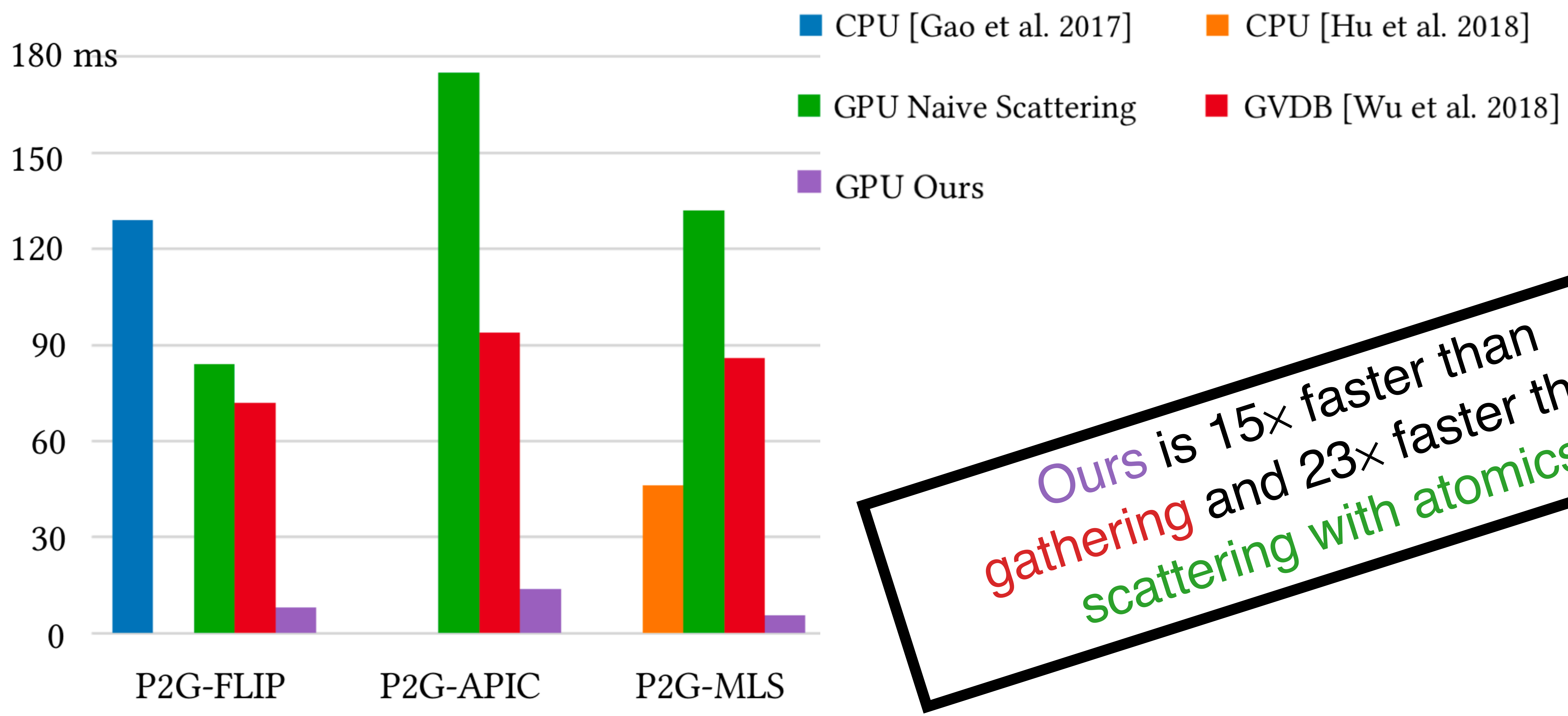
Benchmark

NVIDIA TITAN Xp
Particles #: 7M
Grid res: 128^3



Benchmark

NVIDIA TITAN Xp
Particles #: 7M
Grid res: 128^3



Ours is 15× faster than gathering and 23× faster than scattering with atomics

Grid to particle (G2P)

```
procedure GPUMPM( )
    P ← Initialize particle positions
    P ← Sort and reorder (P)
    for each time step do
        dt ← Compute dt (P)
        G ← Refresh GSPGrid (P)
        M ← Build particle-grid mapping (P, G)
        G ← Transfer from particles to grid (P, M)
        G ← Apply external forces (G)
        G ← Solve on the grid (G, dt)
        P ← Transfer from grid to particles (G, M)
        P ← Update particle attributes (P, dt)
        P ← Resort and reorder (P)
```

Particles: 4.2 M
Grid resolution: 256^3
Simulation: 10.48 secs/frame

Particles: 4.2 M
Grid resolution: 256^3
Simulation: 10.48 secs/frame

Thank you!

Source code - https://github.com/kuiwuchn/GPUMPM

Contact - ming.gao07@gmail.com