Real-Time Simulation of Material Point Method on Modern GPUs

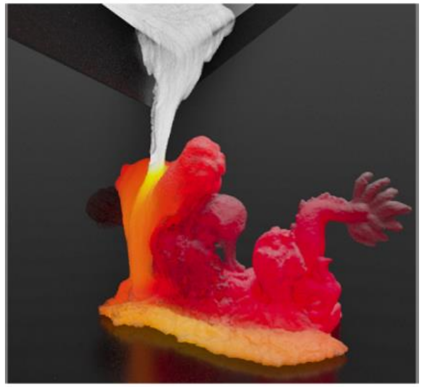
GTC 2022

Yun Fei, Yuhan Huang, Ming Gao

Content

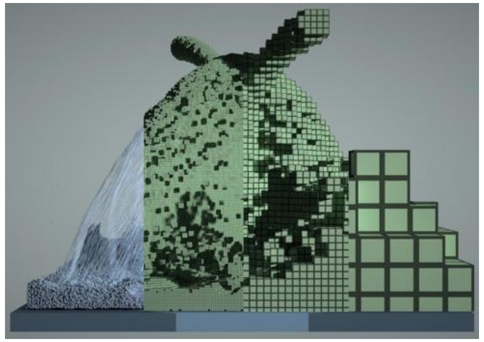
- Background
- Single GPU
- Multiple GPU
- Benchmark and demo

Previous work



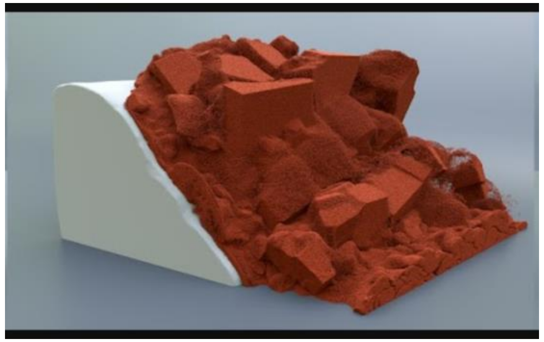
Single GPU
Ming Gao et al.
SIGGRAPH Asia 2018

1.2x



Single GPU
Taichi language
Yuanming Hu et al.
SIGGRAPH Asia 2019

2x

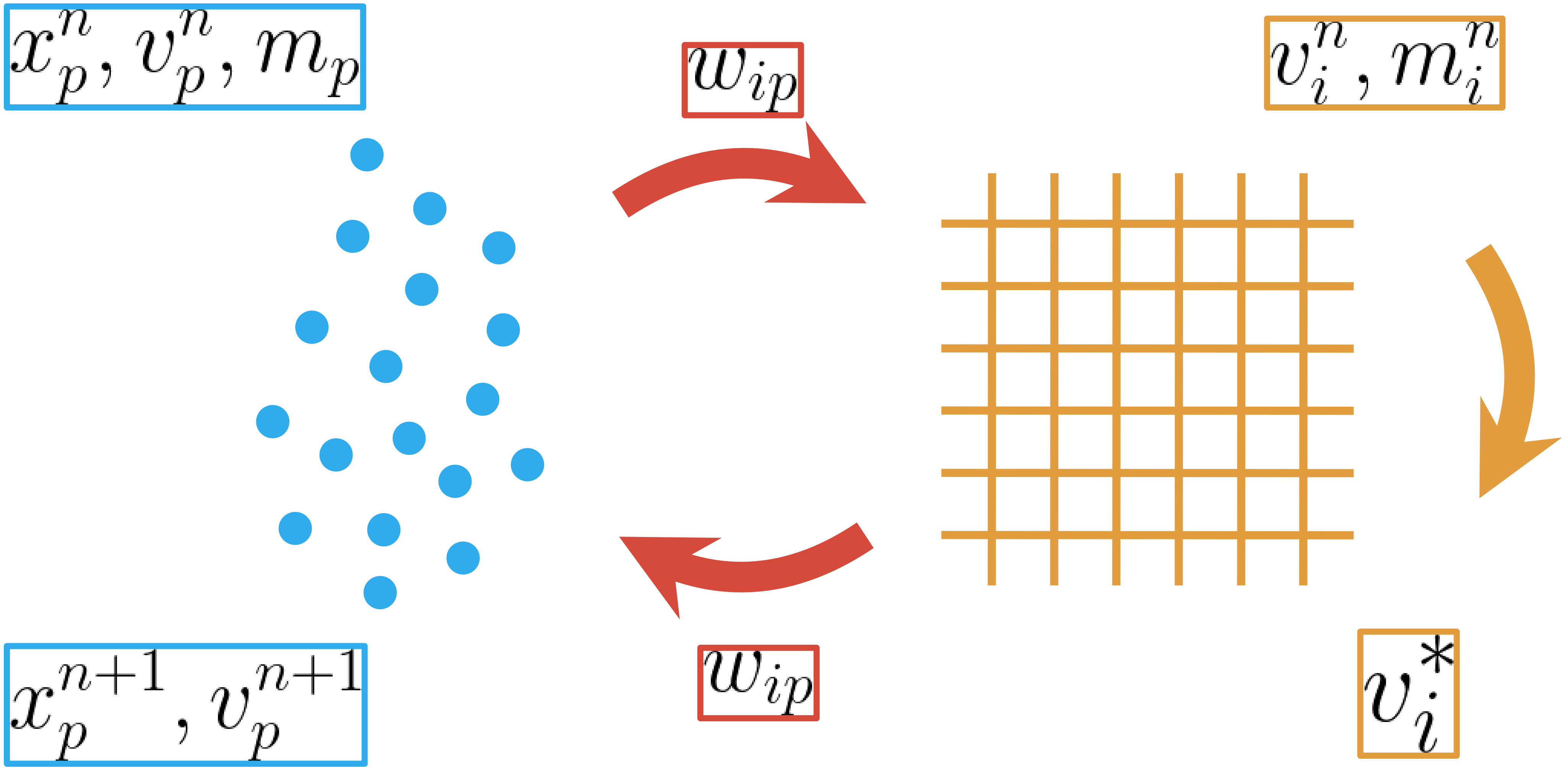


Single and *multi* GPU
Xinlei Wang et al.
SIGGRAPH 2020

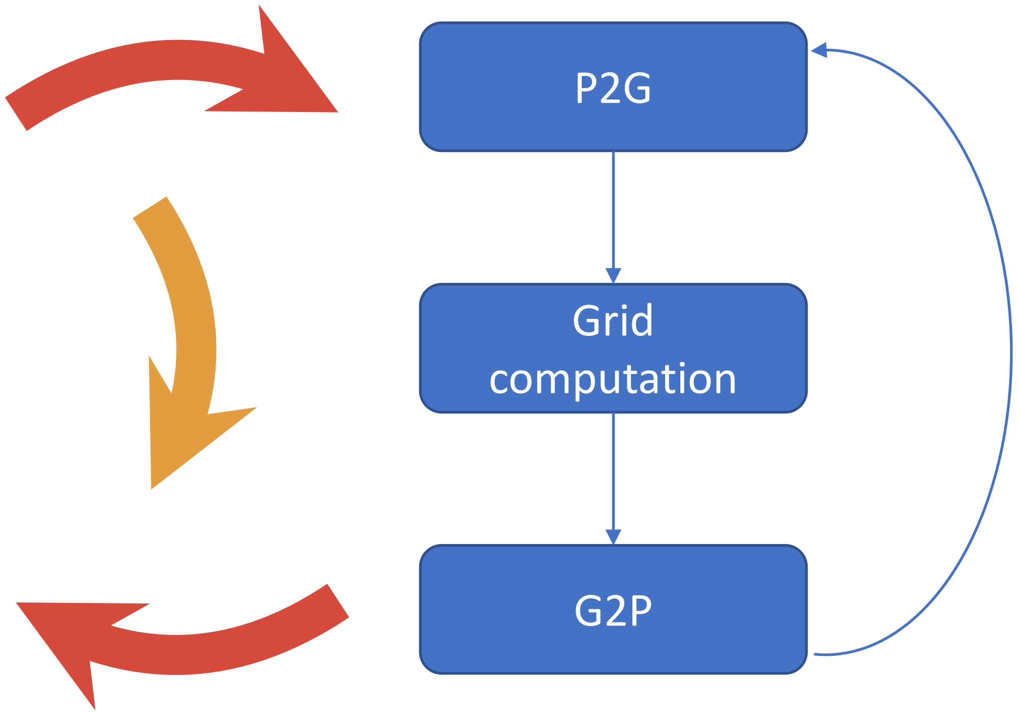
single: 1.7x-8.6x
multiple: 2.5x-14.8x

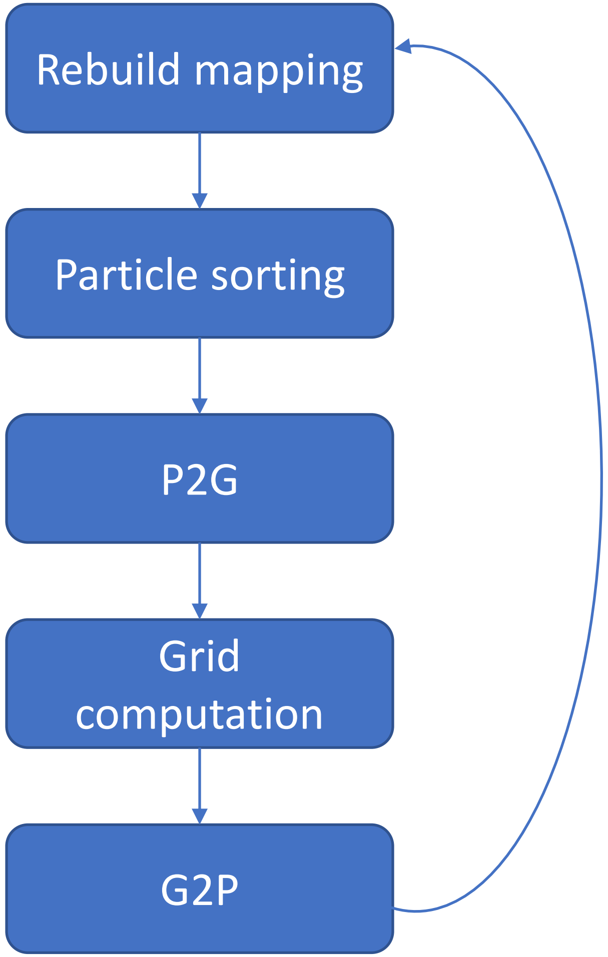
Single and *multi* GPU
Our work

Material point method (MPM)



GPU pipeline



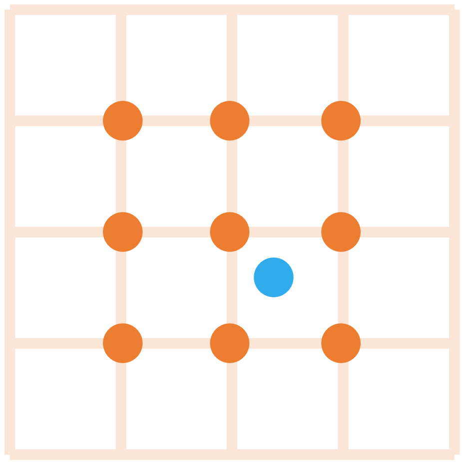


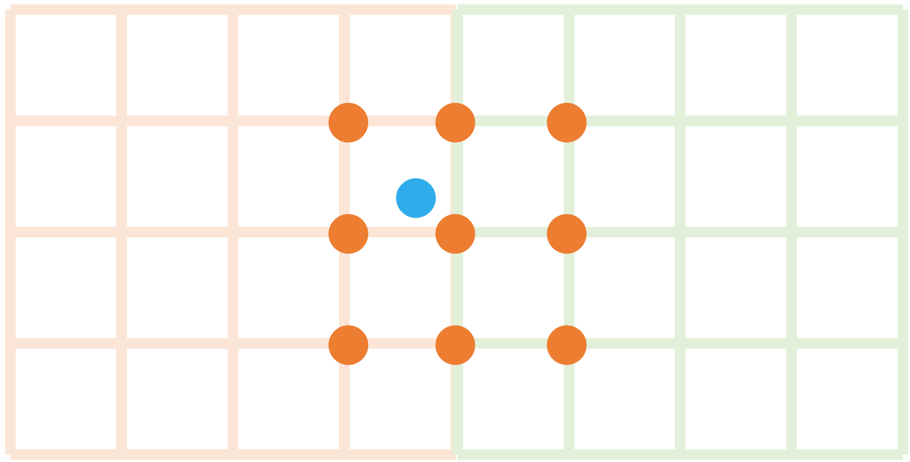
Sparse representation of grid

- Grid nodes are grouped as blocks
  - Only a finite number of blocks are stored in memory
- In space, one block corresponds to 4x4x4 cells
- In memory, one block corresponds to 4x4x4 nodes
  - We store information on the min corner of each cell

Challenge

- Given a group of particles, how to decide the sparsity of the underlying background grid?
  - The number of blocks and where they are
- Simpler version: given one particular particle, how to find the addresses of the nodes it interacts with?

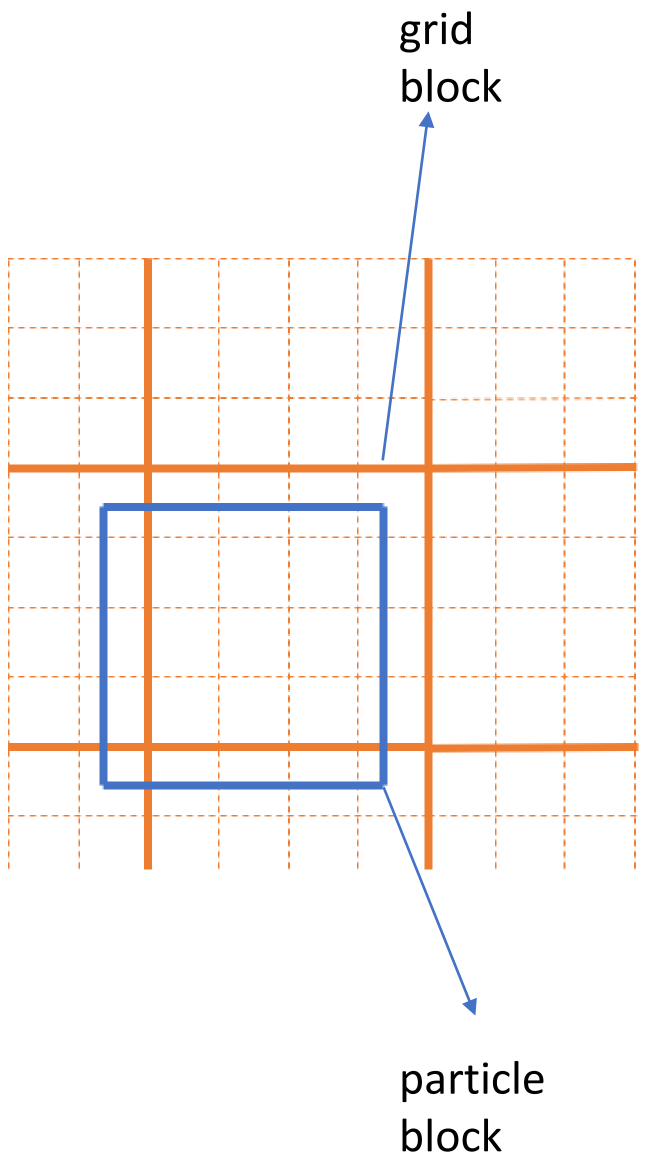




*B-spline quadratic kernel is assumed

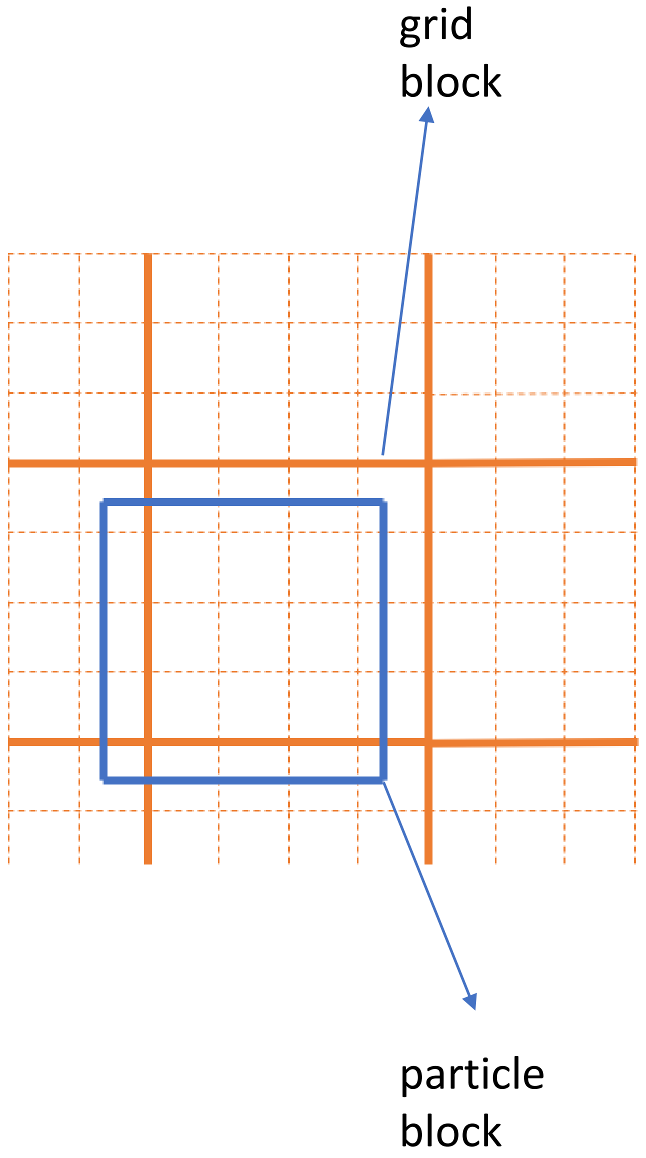
Particle partitioning

- Particles are partitioned into particle blocks
  - Particle blocks do not perfectly overlap with grid blocks
    - There is (α * dx) shift between the two
    - Different works adopt different α and we use -0.5



Particle partitioning

- Particles are partitioned into particle blocks
  - Particle blocks do not perfectly overlap with grid blocks
    - There is (α * dx) shift between the two
    - Different works adopt different α and we use -0.5
- The partitioning was applied every time step
  - However, partitioning itself is not the target
    - The target is much simpler: given a particle, we can find the addresses of the nodes it interacts with
- We make this partitioning much less frequent in this work



Gblock vs pblock

- Geometric blocks (gblock, in space)
  - The grid blocks
  - Correspond to particle blocks (with some shifts)
- Physical blocks (pblock, in memory)
  - As particles talk to 3x3x3 nodes, also allocate memories for the neighboring blocks
    - Given a gblock, explicitly store its 3x3x3 neighbors in a list
  - Gblock is a subset of pblock

Code vs id

Each particle (or the cell it resides in) has a code

- Simply interleave the 32-bit of the 3d index (i, j, k) to a 64-bit 1d code
    - i31, i30, …i0; j31, j30, …j0; k31, k30, …k0
    - (i20, i19 …, i2, j20, j19 …, j2, k20, k19 …, k2) + (i1, i0, j1, j0, k1, k0)
- The lower bits represent the cell inside a block (cell code)
- While the higher bits represent the block information (block code)

Code vs id

Each particle (or the cell it resides in) has a code

- Simply interleave the 32-bit of the 3d index (i, j, k) to a 64-bit 1d code
  - i31, i30, …i0; j31, j30, …j0; k31, k30, …k0
  - (i20, i19 …, i2, j20, j19 …, j2, k20, k19 …, k2) + (i1, i0, j1, j0, k1, k0)
- The lower bits represent the cell inside a block (cell code)
- While the higher bits represent the block information (block code)

We use hash table to decide the gblock group first, and then the pblock group

- The hash table assigns each pblock a unique id
- (key, value) pair is (block code, its unique id) pair
- code vs id
  - id is dense, starting from 0
  - code is sparse

Code vs id

Code

- 64 bits
- Sparse
  - a small subset
- Encode space information

Id

- 32 bits
- Dense
  - each id has a code
- Encode memory information

Content

- Background
- Single GPU
- Multiple GPU
- Benchmark and demo

Principles for Real-Time (Single GPU)

- Reducing memory reallocation once the simulation starts
- Minimizing the synchronization between GPU and CPU
- Fine-tuning the CUDA block size and the usage of on-chip memory
- Minimizing the number of CUDA kernels executed within a single time step
- Avoiding intrinsic functions without native hardware support

Principles for Real-Time (Single GPU)

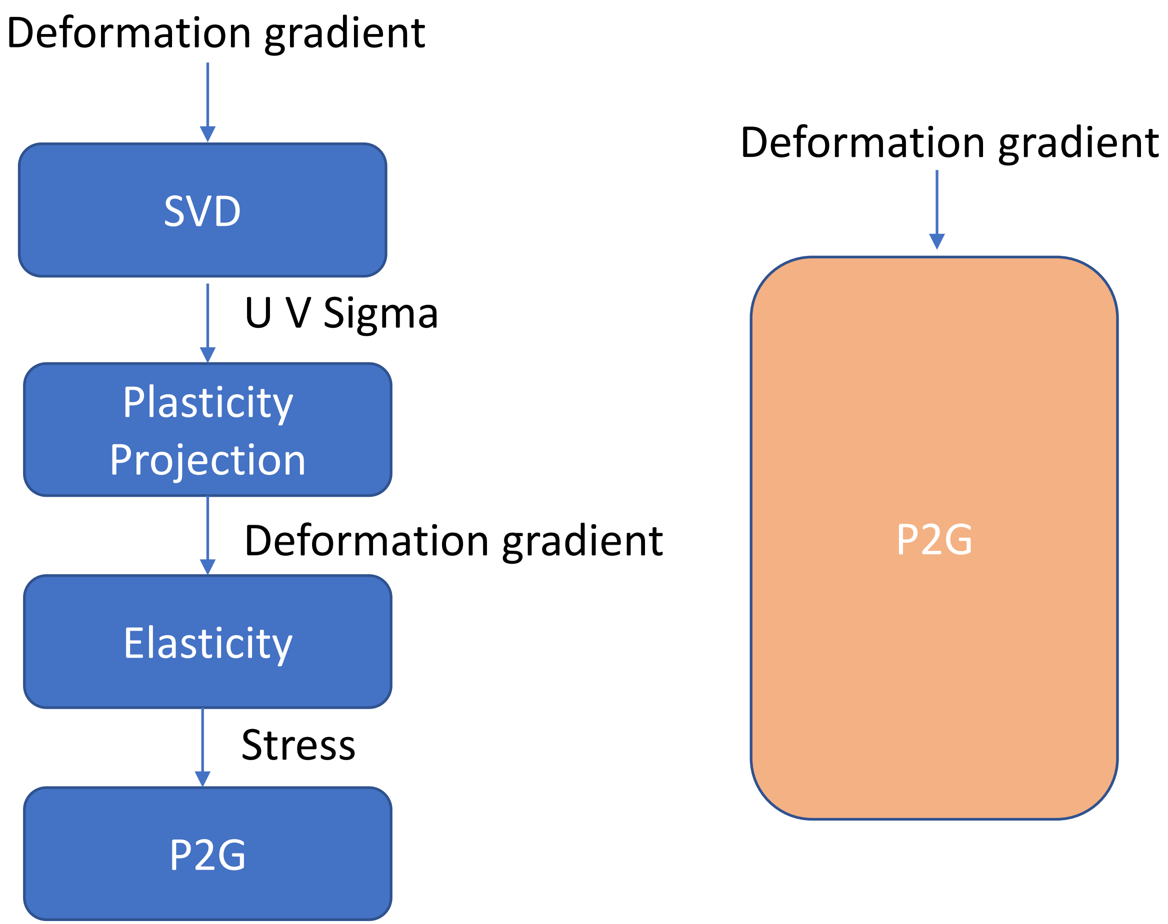
- Reducing memory reallocation once the simulation starts
- Minimizing the synchronization between GPU and CPU
- Fine-tuning the CUDA block size and the usage of on-chip memory
- Minimizing the number of CUDA kernels executed within a single time step
- Avoiding intrinsic functions without native hardware support

Principles for Real-Time (Single GPU)

- Reducing memory reallocation once the simulation starts
- Minimizing the synchronization between GPU and CPU
- Fine-tuning the CUDA block size and the usage of on-chip memory
- Minimizing the number of CUDA kernels executed within a single time step
  - Merge kernels
  - Avoid non-essential computations
- Avoiding intrinsic functions without native hardware support

Merge kernels

- Pros
  - Reduce global memory accesses
    - System state vs temporary state



Merge kernels

- Pros
  - Reduce global memory accesses
    - System state vs temporary state
  - Reduce tail effect
  - Better chance to overlap memory operations with computations

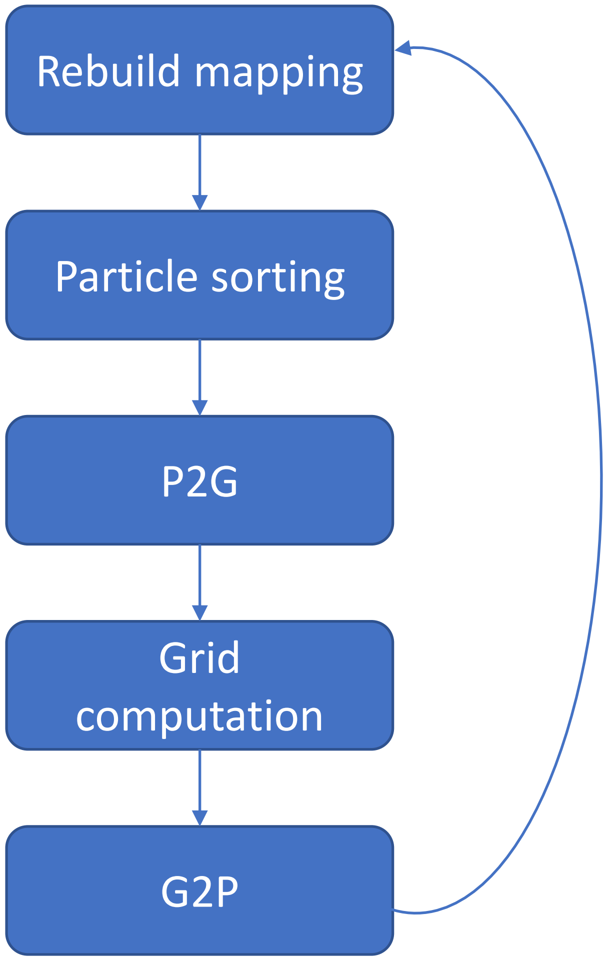
Merge kernels

- Pros
  - Reduce global memory accesses
    - System state vs temporary state
  - Reduce tail effect
  - Better chance to overlap memory operations with computations

- Cons (merge too many kernels)
  - Spill registers to local memory
  - Higher instruction cache miss
  - (G2P2G in Xinlei Wang et al. 2020) Forbid Lagrangian MPM model & particle insertion and deletion

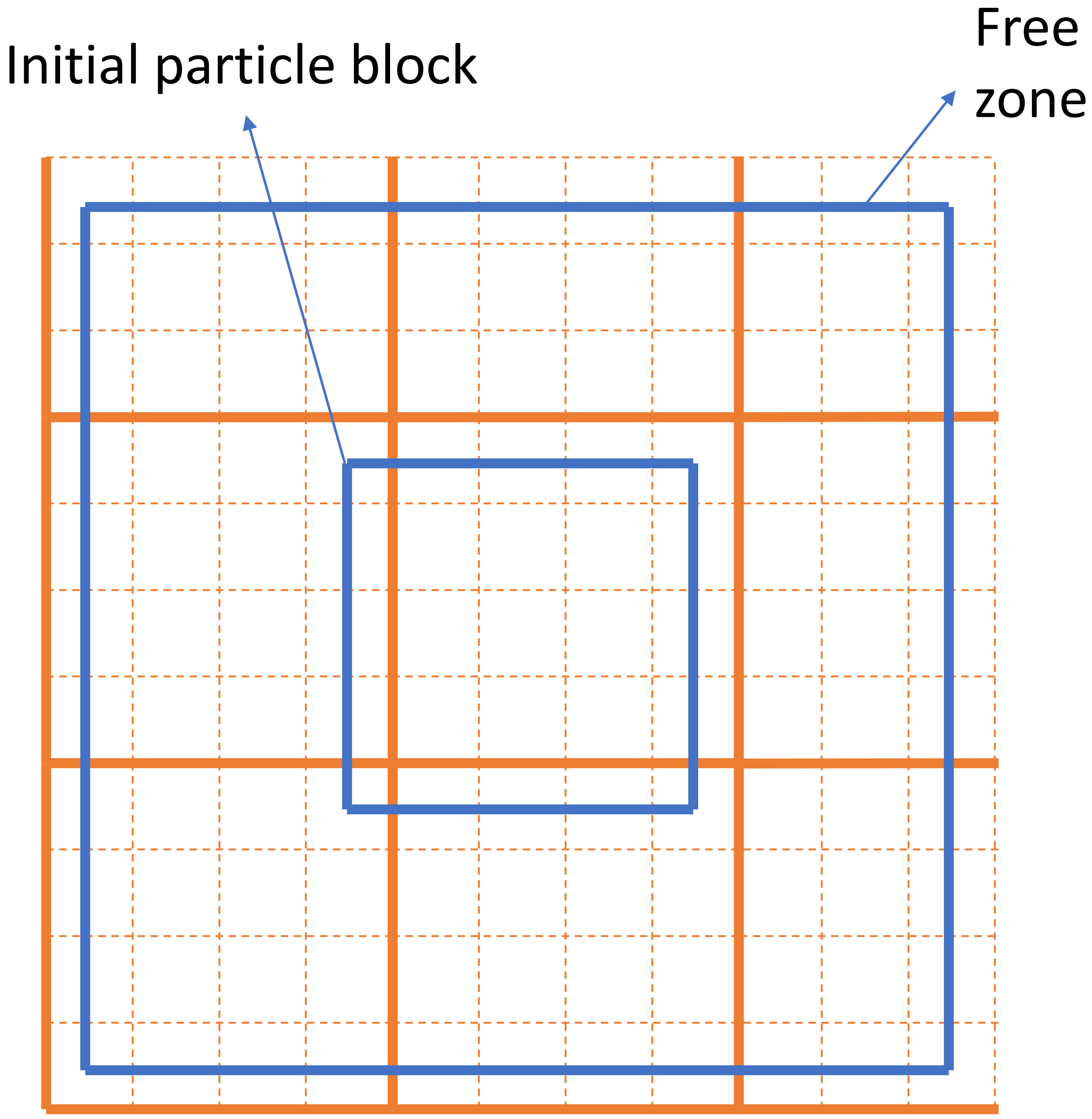
Minimize non-essential computations

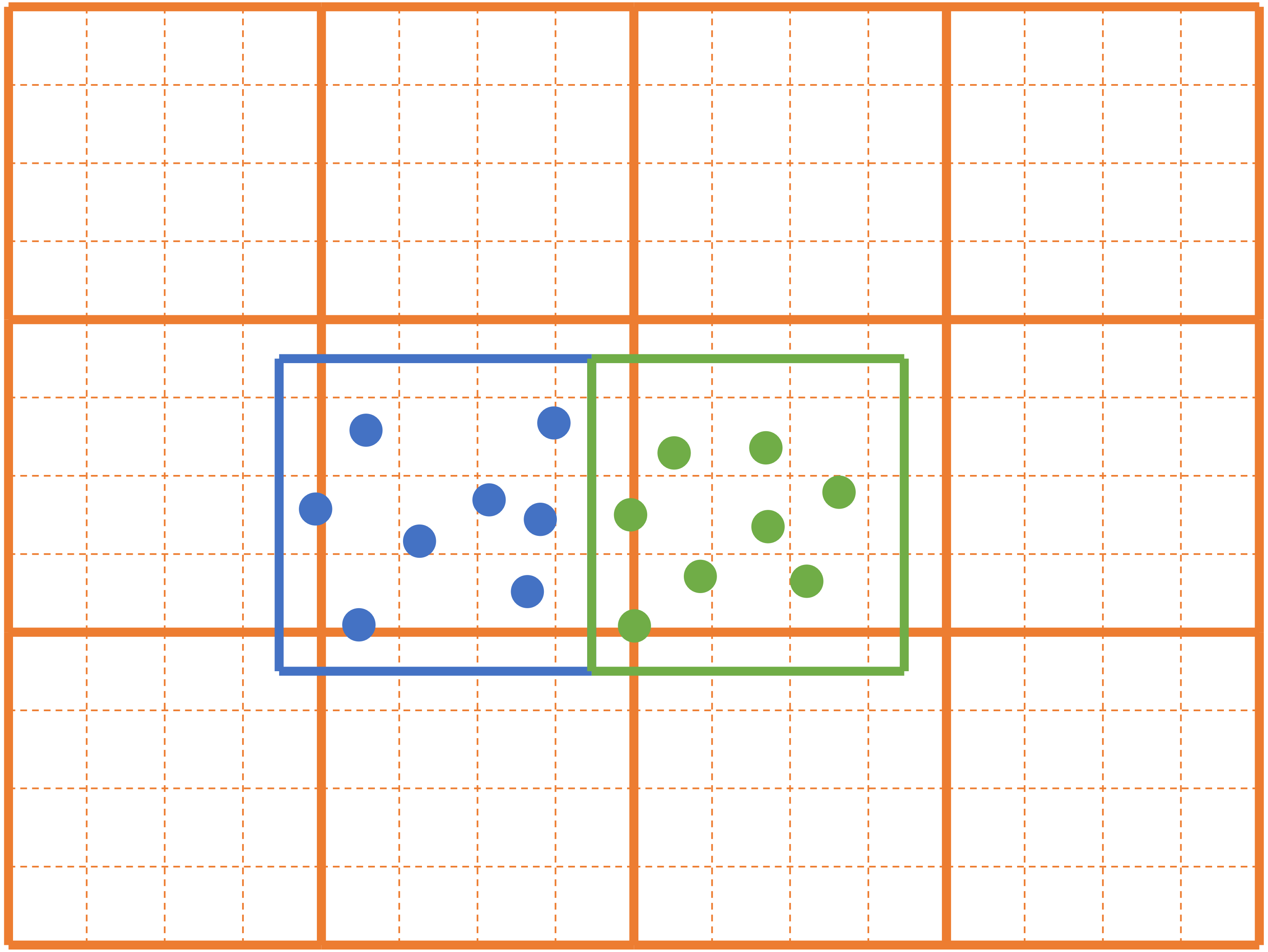
- Identify non-essential stages
  - Sparse grid -> dense grid
  - Sequential accesses -> random order access



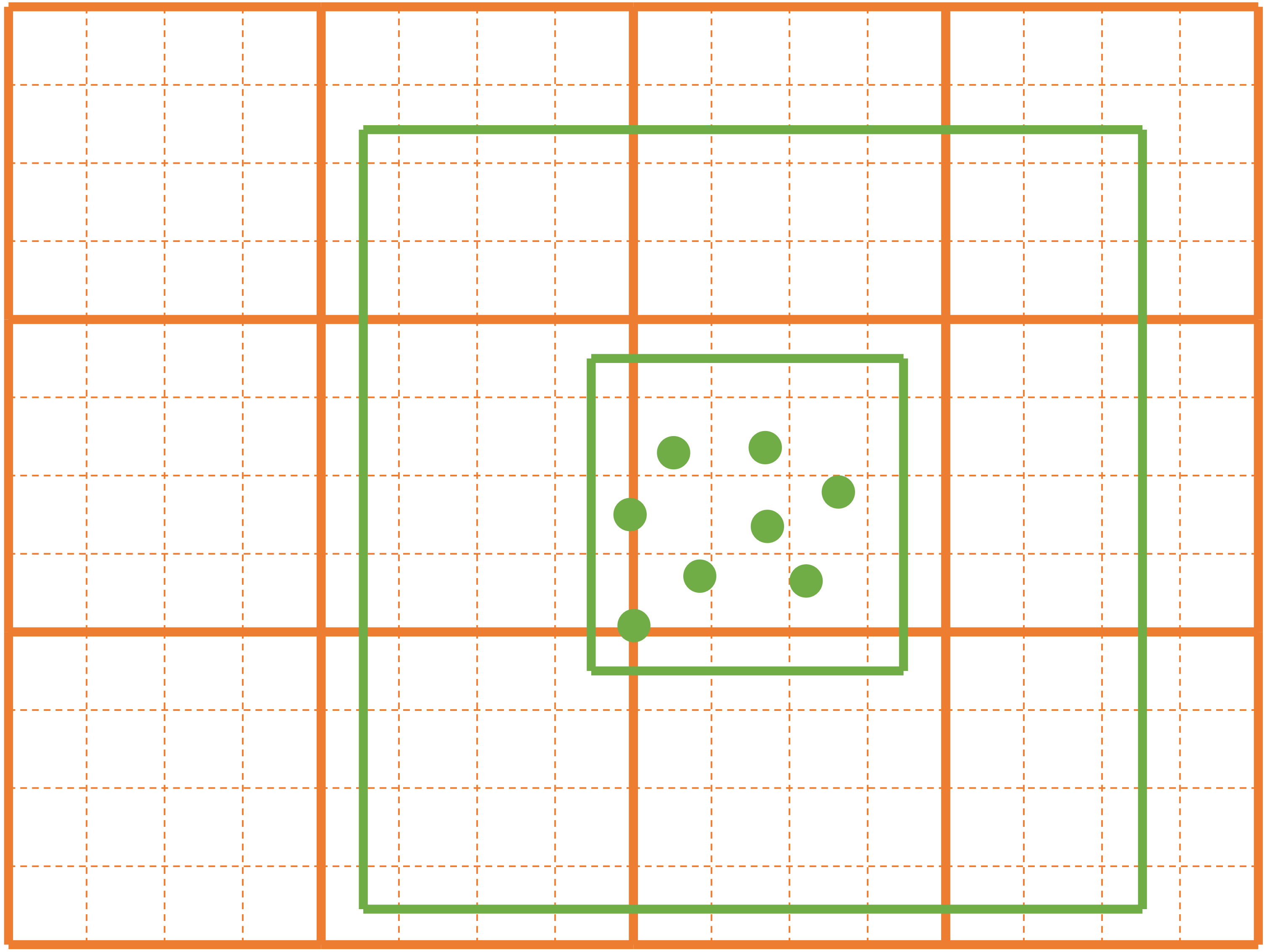
Rebuild-mapping

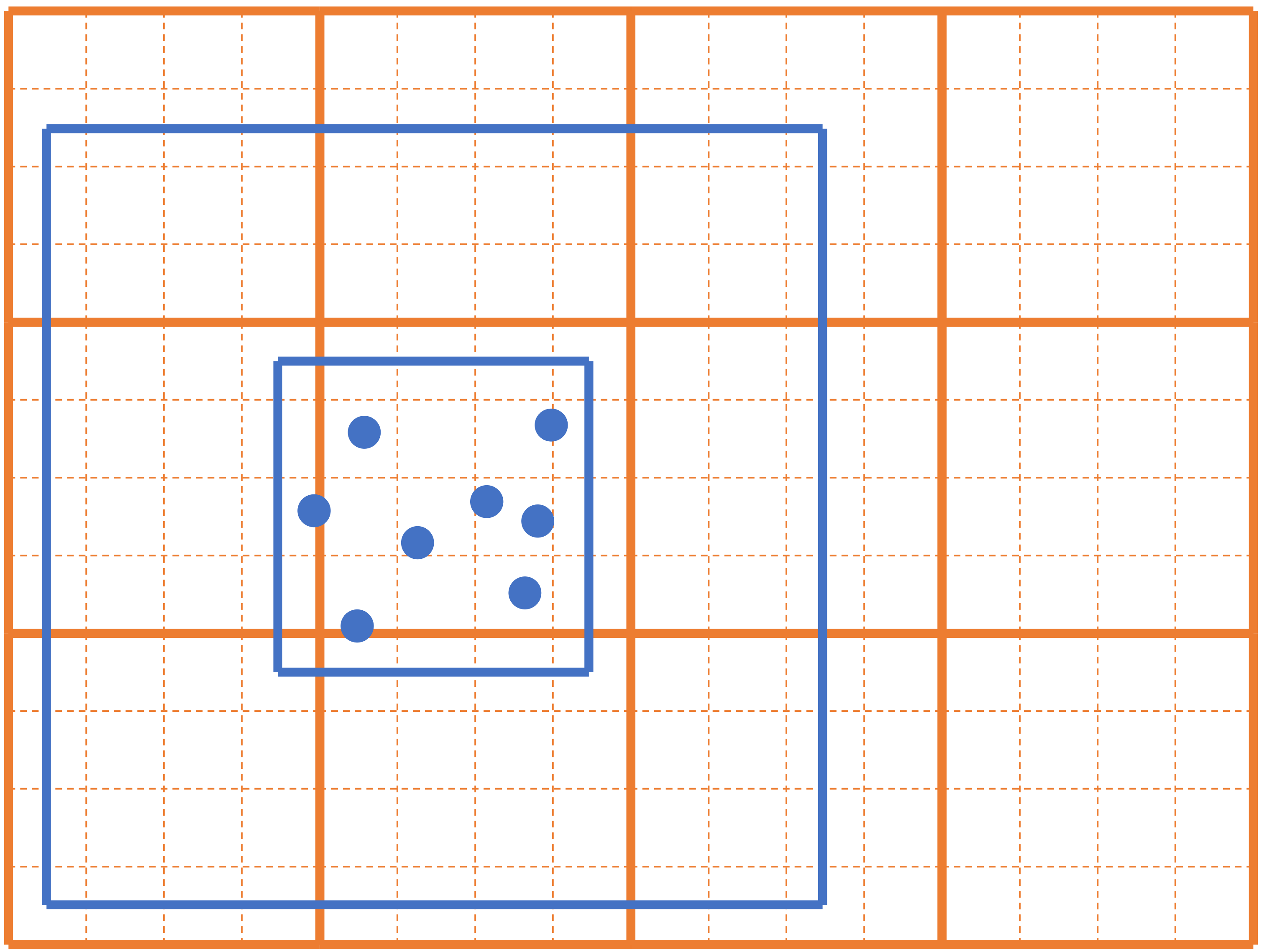
- Used to execute every time step, why?
  - Particles advect at the end of every time step
- We propose the idea of free zone
  - A zone that is free from rebuilding the mapping
  - Particles can freely move in a domain of (10dx)^3 without triggering

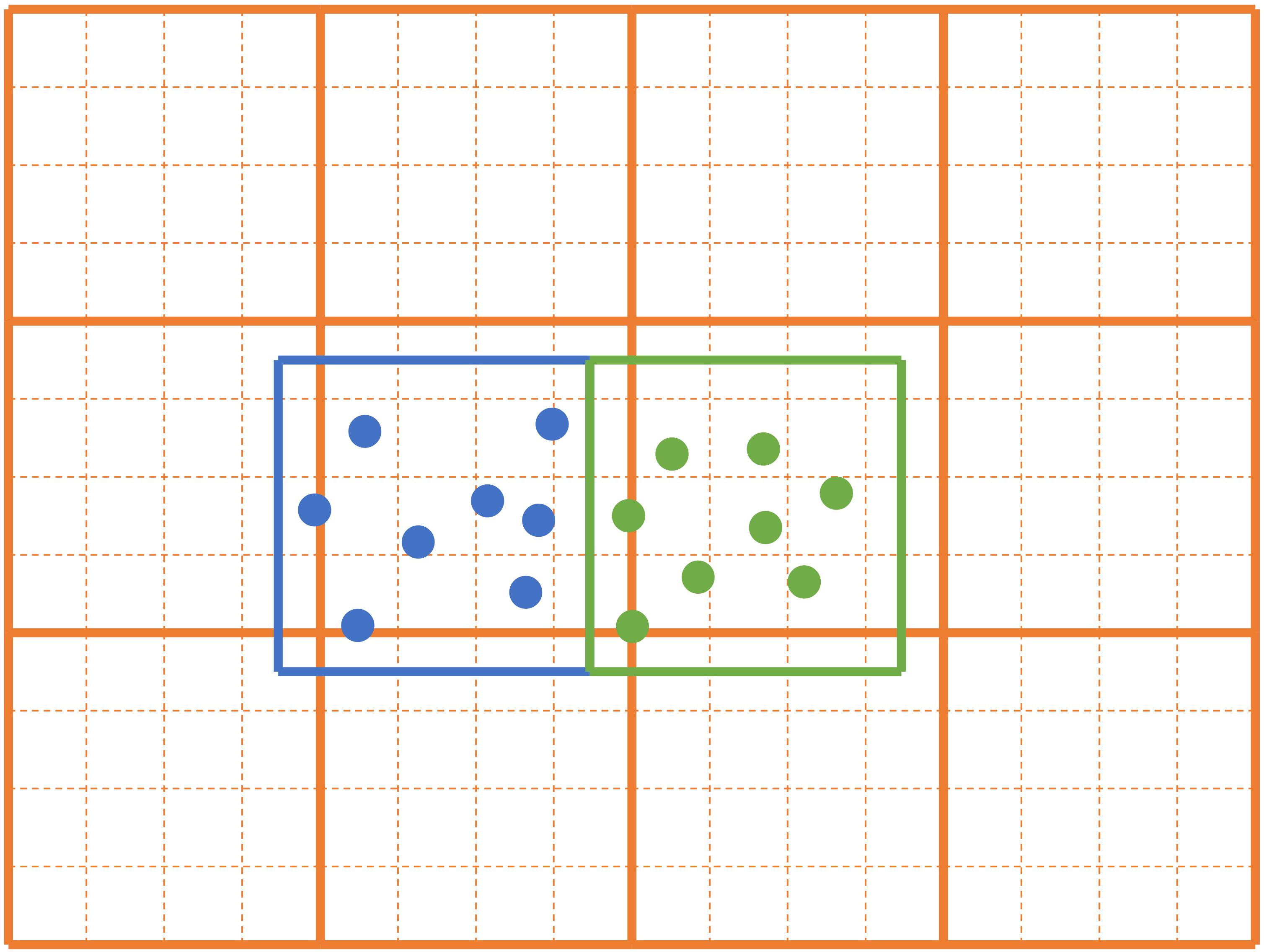




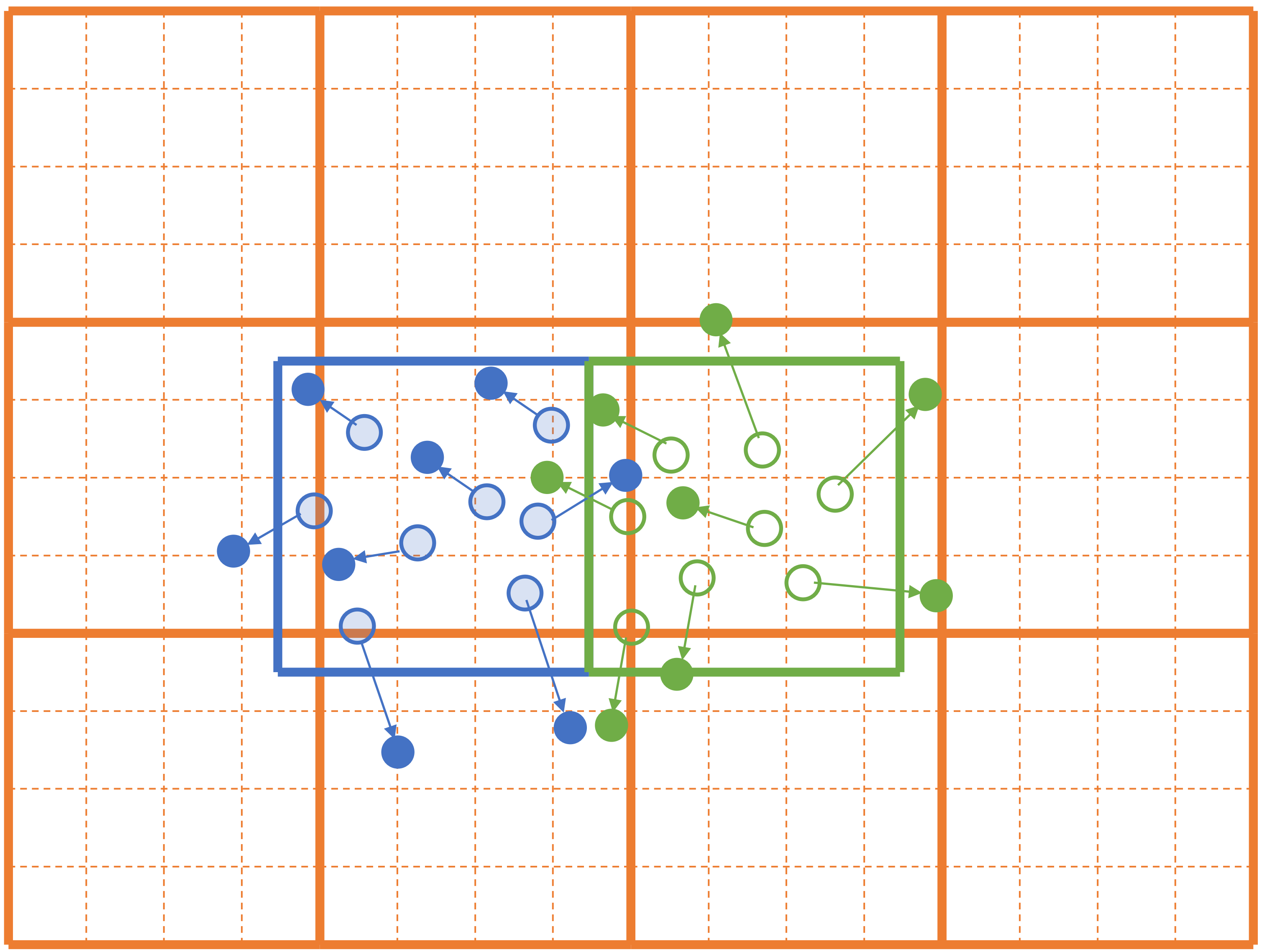
Perfectly portioned particle blocks



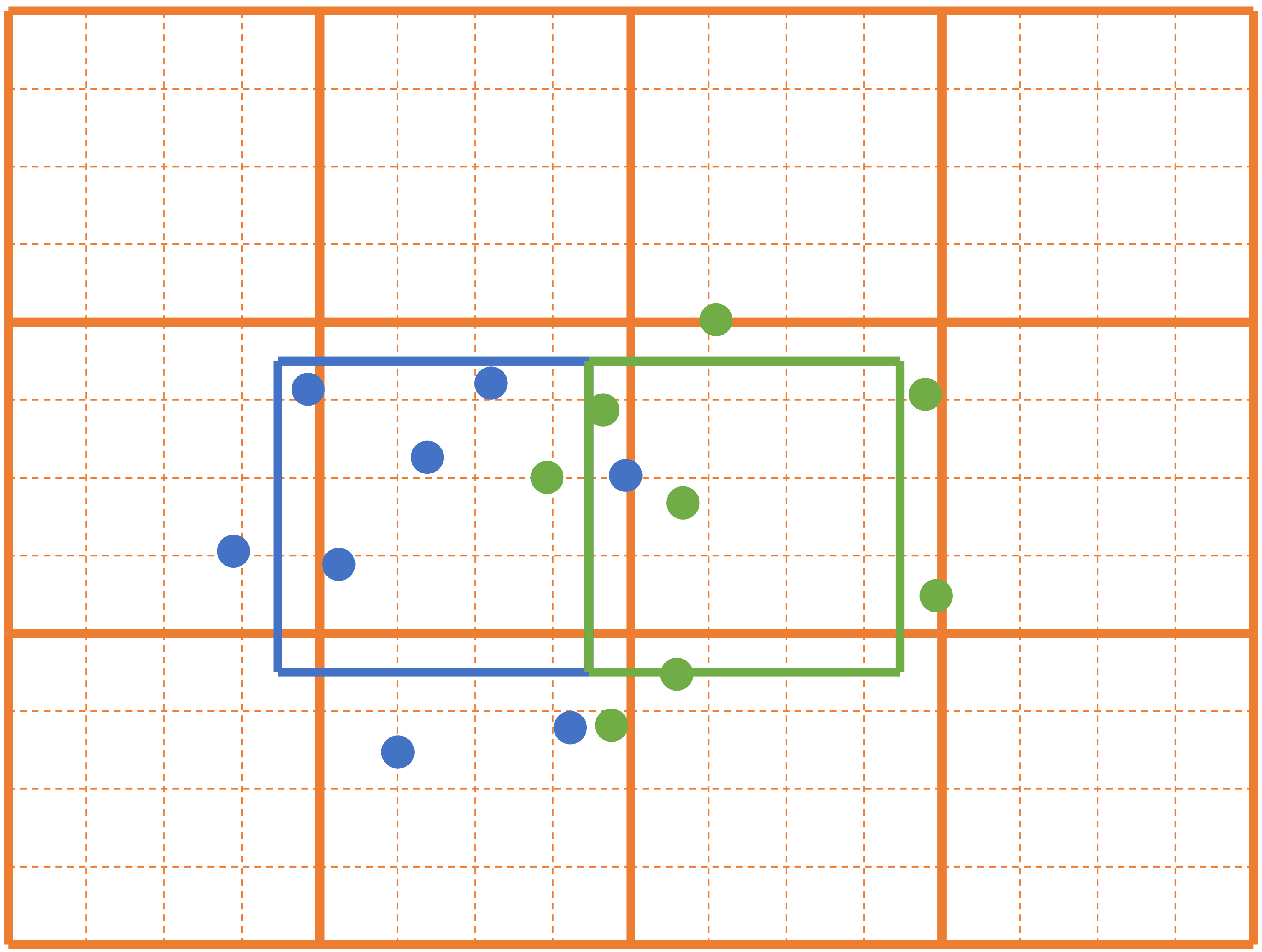




Perfectly portioned particle blocks



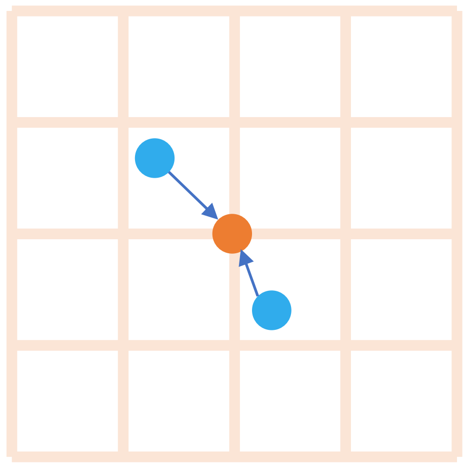
Particles move around



Overlap with each other but still no rebuild-mapping needed

Particle sorting

- Each thread handles one particle and one warp handles 32 particles in parallel
- Particles in the same warp may simultaneously write to the same node
  - Option 1 (Yuanming Hu et al. 2019): randomly shuffle particles such that the chance of conflict in a warp becomes low
  - Option 2 (Ming Gao et al. 2018): apply warp-level reduction (need to sort particles to cells)
- We propose a mixed sorting



Combine cheap and expensive sorting

Expensive/complete sorting

(During rebuild-mapping)

- Apply the complete sorting
  - both block-level and cell-level
  - update the number of warps and refresh the particles in each warp
- Reduce number of conflicts in a global sense

Combine cheap and expensive sorting

Expensive/complete sorting

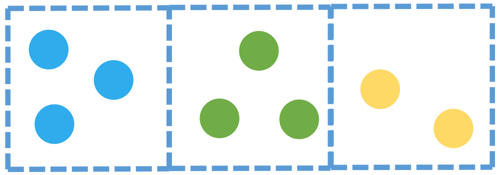
(During rebuild-mapping)

- Apply the complete sorting
  - both block-level and cell-level
  - update the number of warps and refresh the particles in each warp
- Reduce number of conflicts in a global sense

Cheap sorting

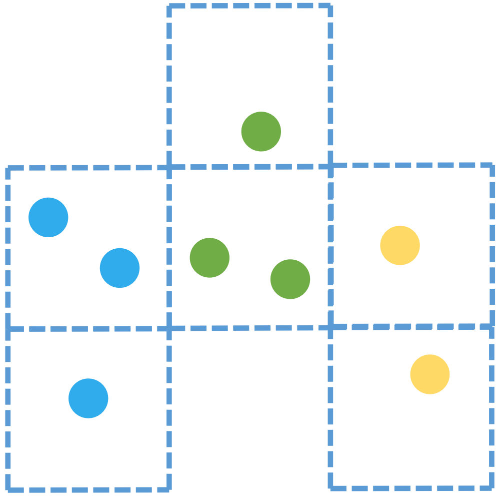
(Between two rebuild-mappings)

- Only apply radix sorting to 32 particles in each warp
  - only cell-level
  - the number of warps and the particles in each warp remain unchanged
  - merge the cheap sorting into P2G to further reduce cost
- Reduce number of conflicts in a local sense
  - Not optimal, but still reasonable

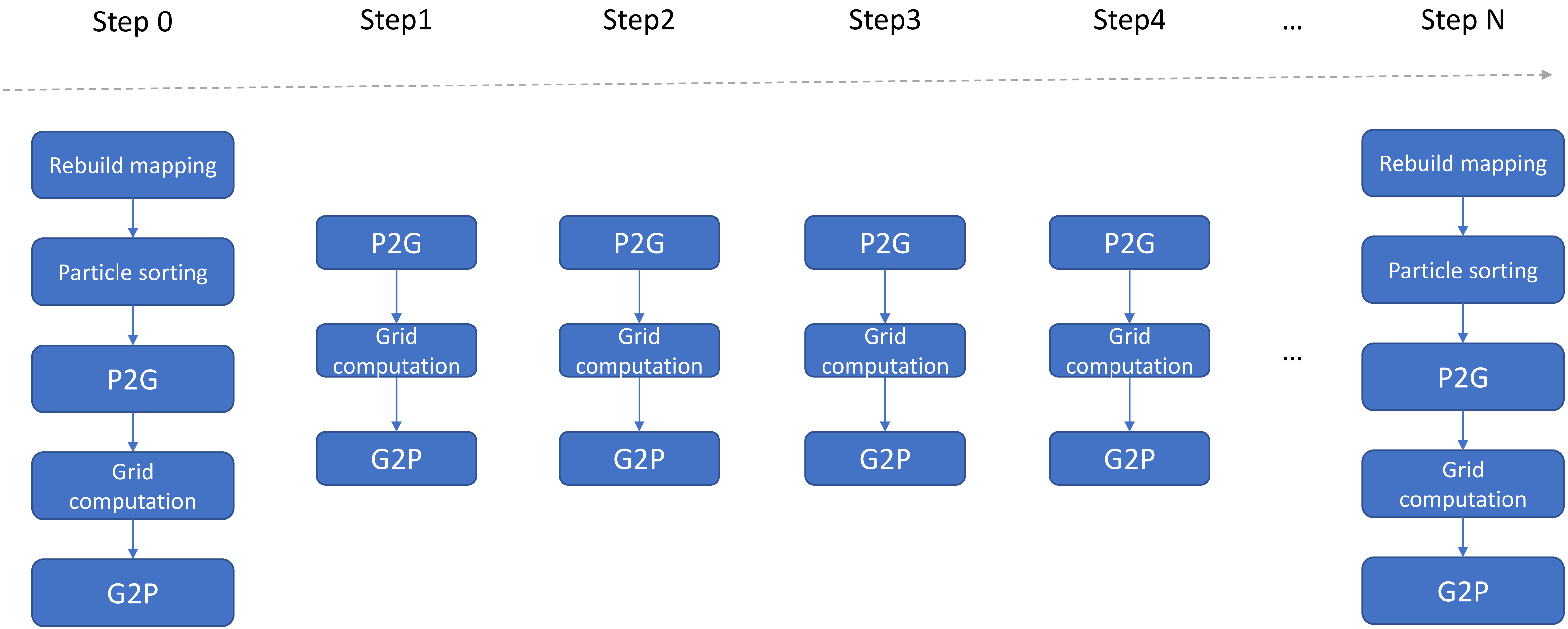


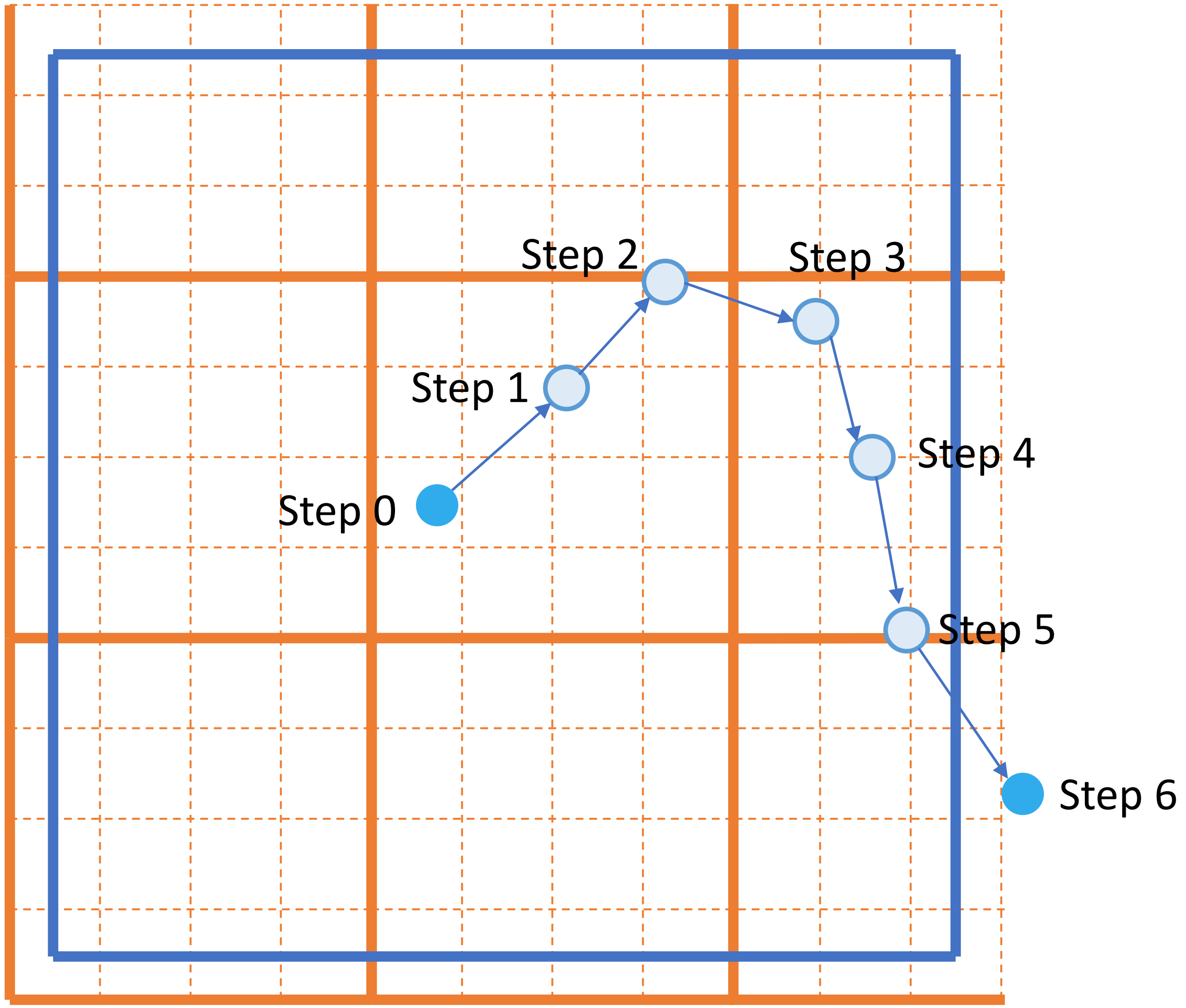
8 atomics reduce to 3 atomics by warp-level reduction proposed in Gao et al. 2018

After several steps:



8 atomics reduce to 6 atomics.
Not optimal, but still acceptable as cell-level sorting is much cheaper than a complete sorting


Step 0
Step1
Step2
Step3
Step4
...
Step N
Rebuild mapping
Particle sorting
P2G
Grid computation
G2P
P2G
Grid computation
G2P
P2G
Grid computation
G2P
P2G
Grid computation
G2P
P2G
Grid computation
G2P
...
Rebuild mapping
Particle sorting
P2G
Grid computation
G2P



From step 1 to step 5, no rebuild-mapping is needed

Principles for Real-Time (Single GPU)

- Reducing memory reallocation once the simulation starts
- Minimizing the synchronization between GPU and CPU
- Fine-tuning the CUDA block size and the usage of on-chip memory
- Minimizing the number of CUDA kernels executed within a single time step
    - Merge kernels
    - Avoid non-essential computations
- Avoiding intrinsic functions without native hardware support

Principles for Real-Time (Single GPU)

- Reducing memory reallocation once the simulation starts
- Minimizing the synchronization between GPU and CPU
- Fine-tuning the CUDA block size and the usage of on-chip memory
- Minimizing the number of CUDA kernels executed within a single time step
- Avoiding intrinsic functions without native hardware support

Avoid non-native intrinsics

- Native intrinsics – translated to only one or very few low level instructions
- With hardware support
- Example: float atomicAdd to global memory

- Non-native intrinsics – translated to multiple low level instructions
- Software implementation
- Example: float atomicAdd to shared memory
  - implemented by loop + atomic compare-and-swap
- Example: floating-point operations: $\frac{x}{y}$, sinf(x), logf(x)
  - when precision is not critical, compile with “-use_fast_math” flag

Revisit conflicts in P2G

- Multiple particles/threads simultaneously write to the same node
- Warp-level reduction resolves conflicts within each warp
- Still need to handle conflicts between threads from different warps/blocks

- Previous works all rely on shared memory to convert some of the global conflicts to shared conflicts
  - Idea is good
  - However, there does not exist native shared atomics
  - Bring in many restrictions
- We directly write from threads to global addresses without using shared memory as the scratchpad

Restrictions due to shared memory

- One CUDA block handles particles from the same block
  - Particles are grouped to virtual blocks

  (when one particle block has too many particles to fit in one CUDA block)
- Large CUDA block size
  - 512 threads per CUDA block
- Shared memory has limited size
  - 2x2x2 neighboring blocks are adopted
- Synchronization before writing from shared to global

Restrictions due to shared memory

- One CUDA block handles particles from the same block
  - Particles are grouped to virtual blocks

  (when one particle block has too many particles to fit in one CUDA block)
- Large CUDA block size
  - 512 threads per CUDA block
- Shared memory has limited size
  - 2x2x2 neighboring blocks are adopted
- Synchronization before writing from shared to global

- One CUDA block handles warps from different blocks
  - Particles are grouped to warps
- Flexible CUDA block size
  - 4 warps per CUDA block
- We can use larger 3x3x3 neighboring blocks
  - Compatible with free zone
- No synchronization required during P2G

Content

- Background
- Single GPU
- Multiple GPU
- Benchmark and demo

From single GPU to multiple GPUs

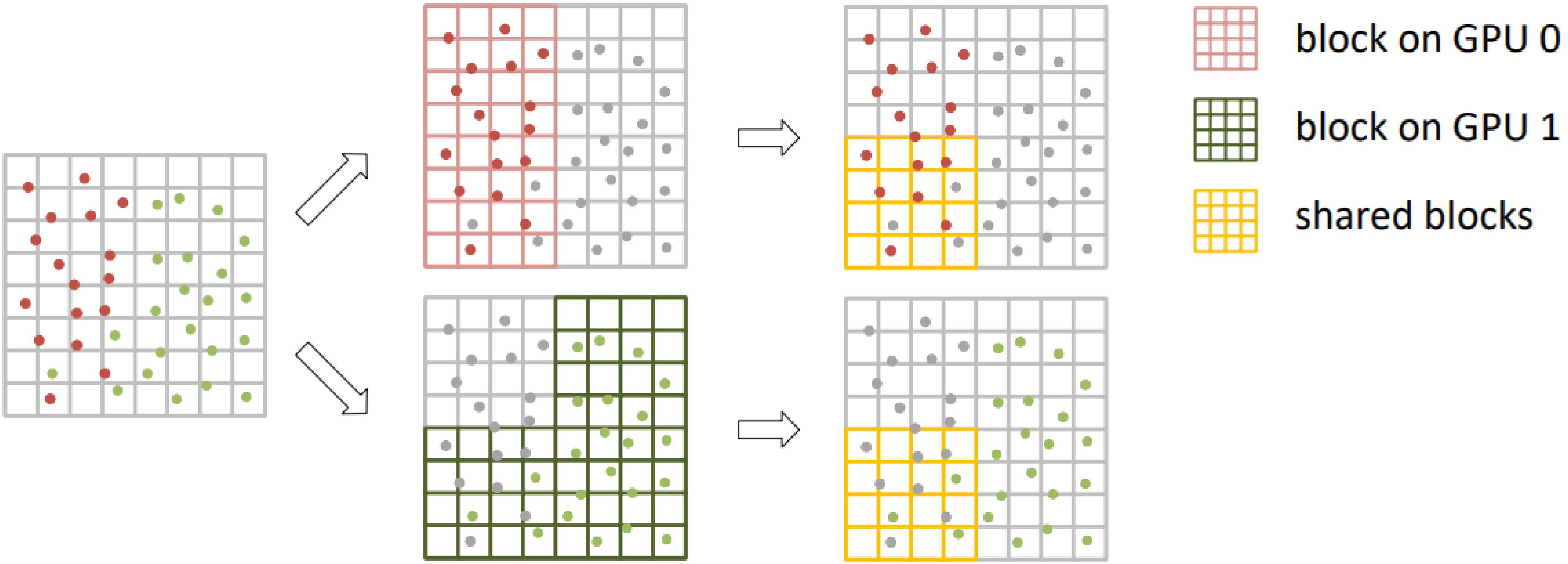
- Challenge from multiple-GPU:
  - Inter-GPU bandwidth is significant lower, and the latency is much higher.
  - We must minimize the cost on inter-GPU communication.

- Multiple GPU parallel approaches:
  - Job splitting by particles
    - Need reduce sum on grid data after P2G
  - Job splitting by grids
    - Need to move particles between GPUs

Job splitting by particles

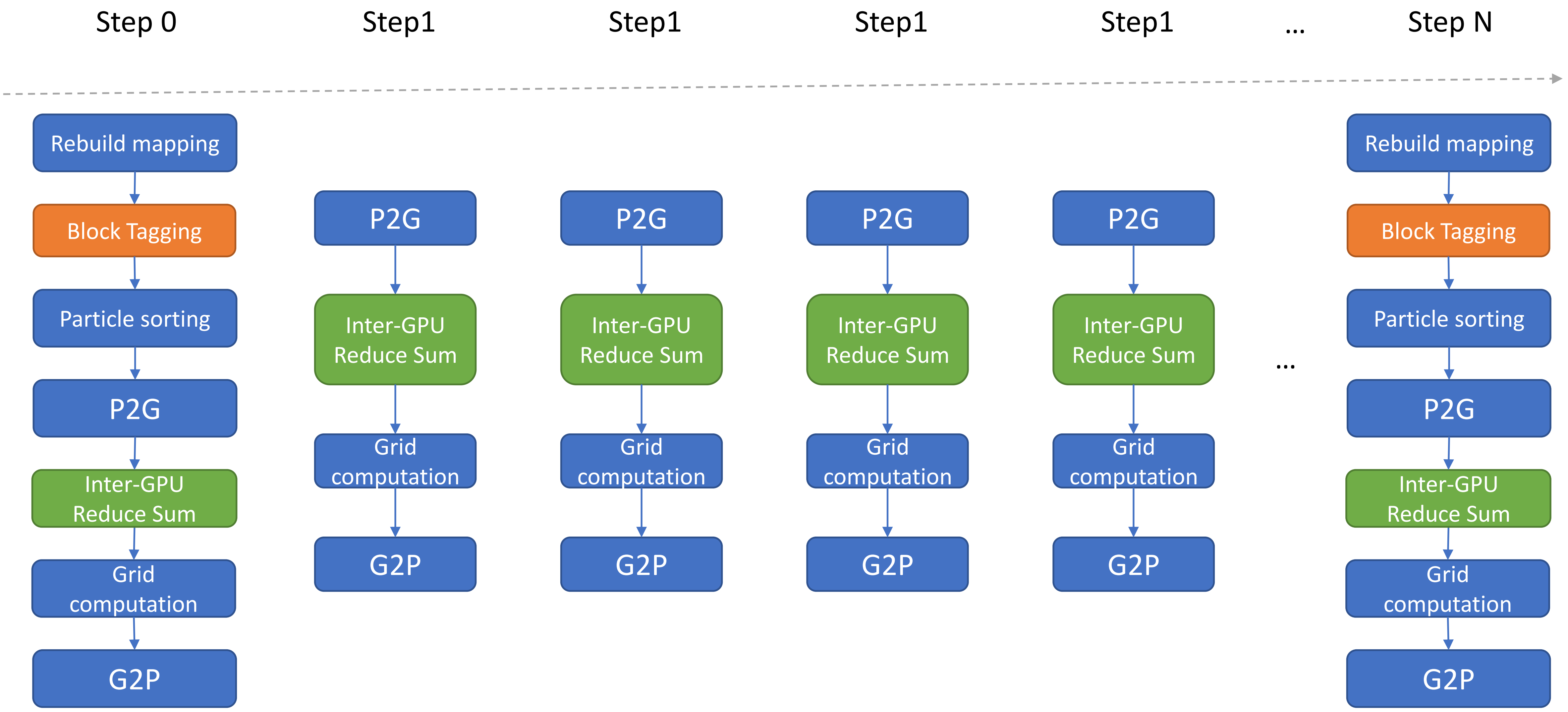
- Computation job are divided by assigning particles to different GPUs
- Most computations are independent between GPUs.
- Inter-GPU communication is limited to reduce sum of shared blocks.
- Inter-GPU Synchronization is required once a time step.

Block Tagging

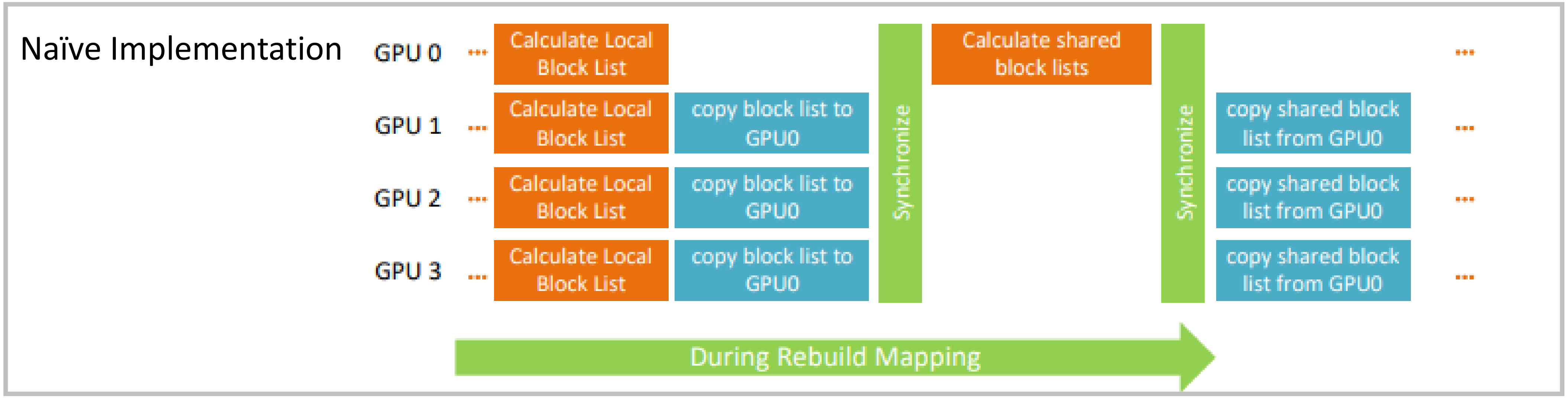


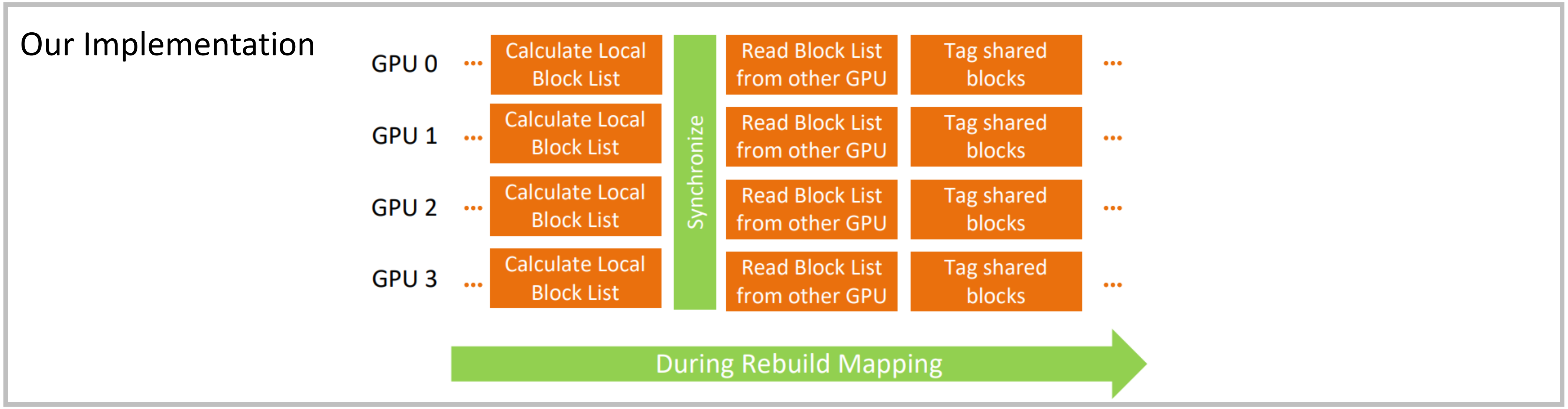
Particles are assigned to different GPUs → Generate list of blocks on each GPU → Calculate Shared Blocks through hash function → Shared blocks will tagged

Multiple GPU workflow

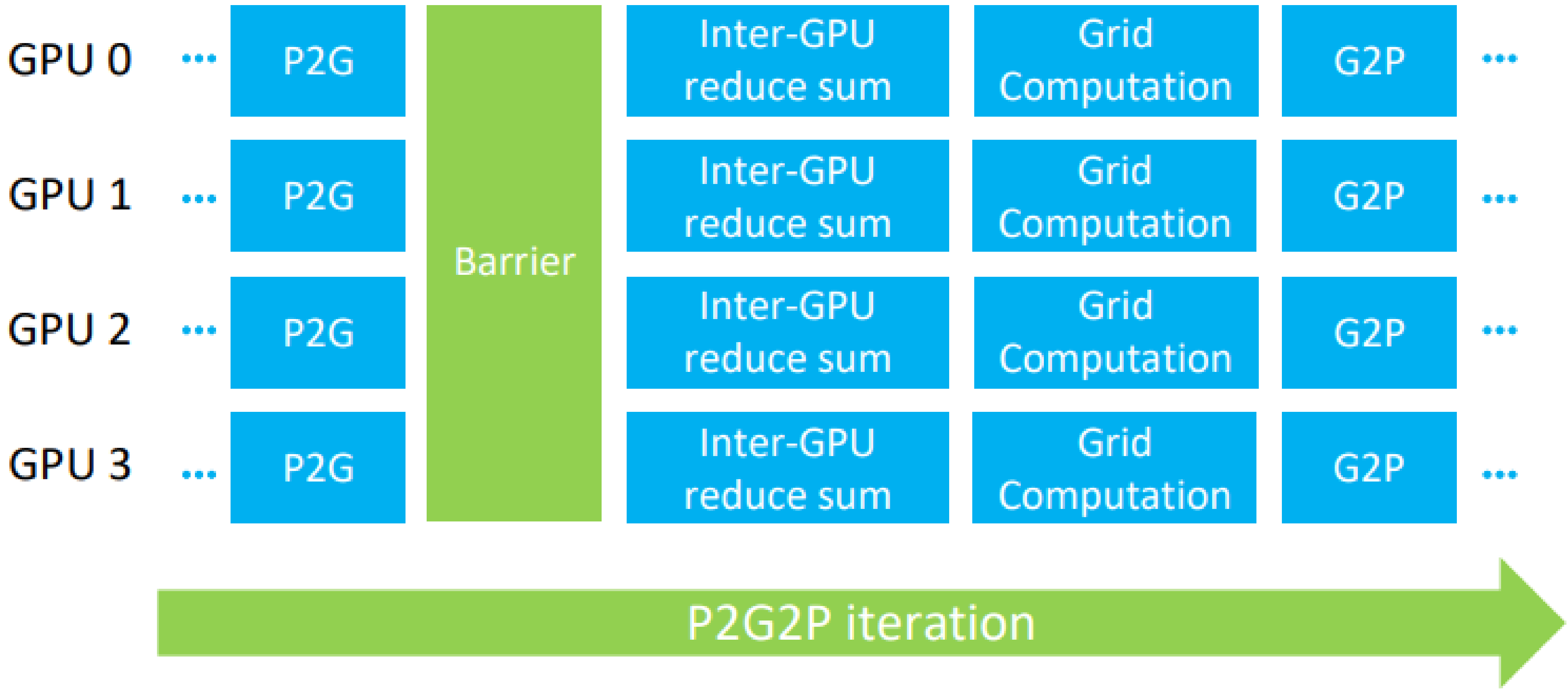


Block Tagging

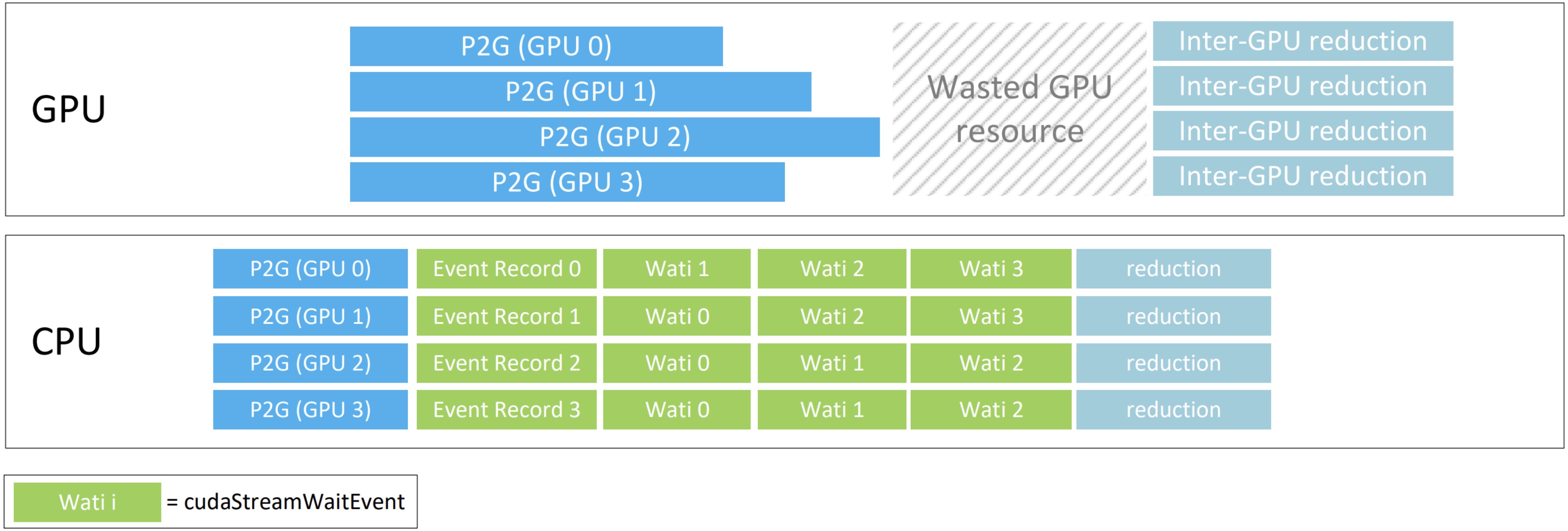




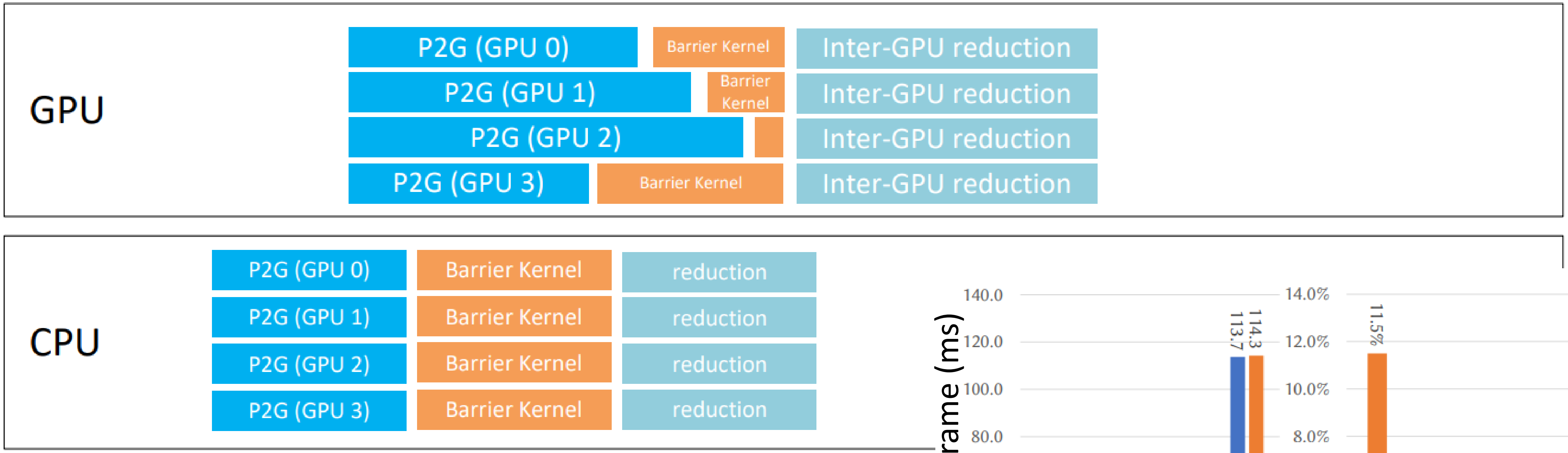
Multiple-GPU time step

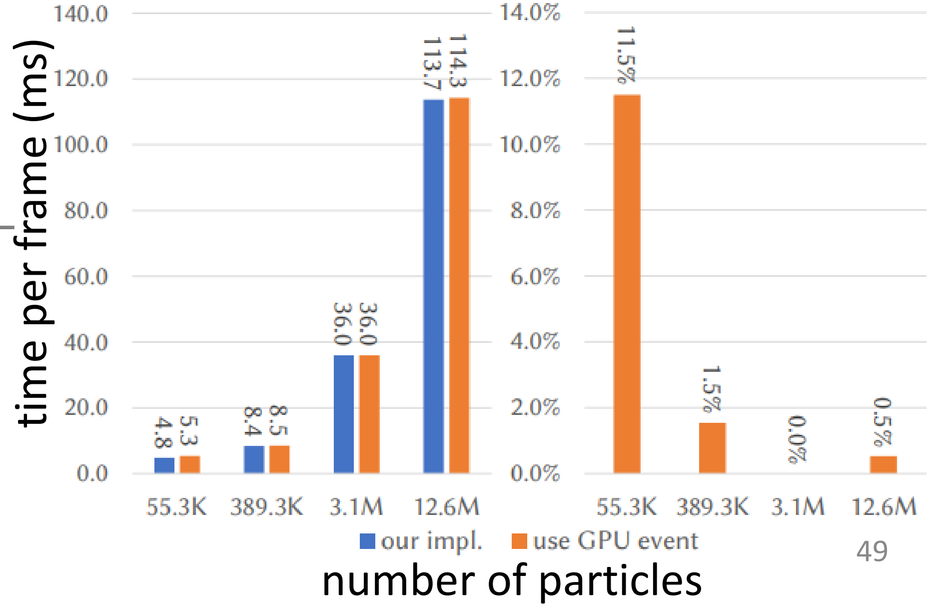


Implementation of inter-GPU barrier



Implementation of inter-GPU barrier

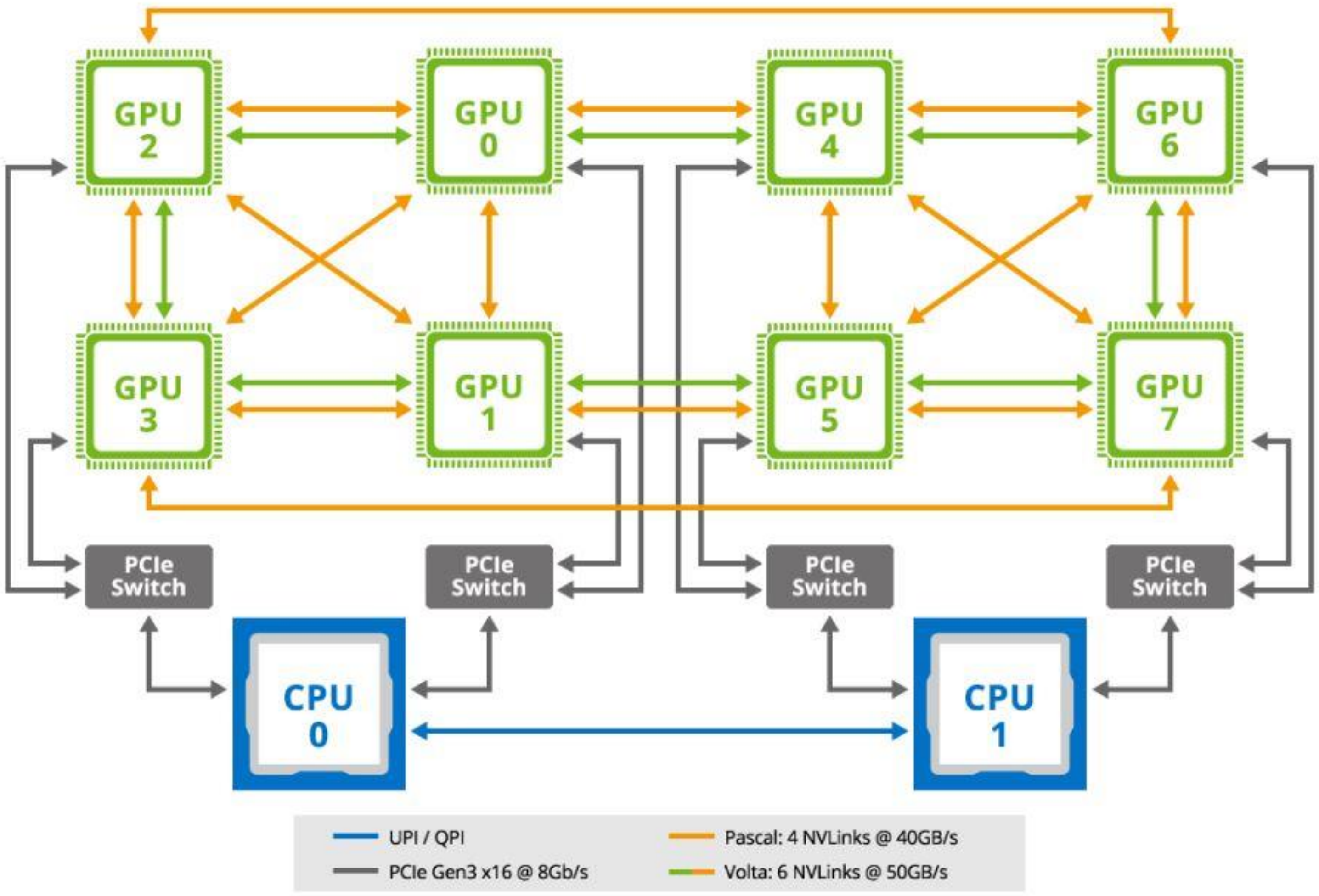




Implementation of the inter-GPU barrier

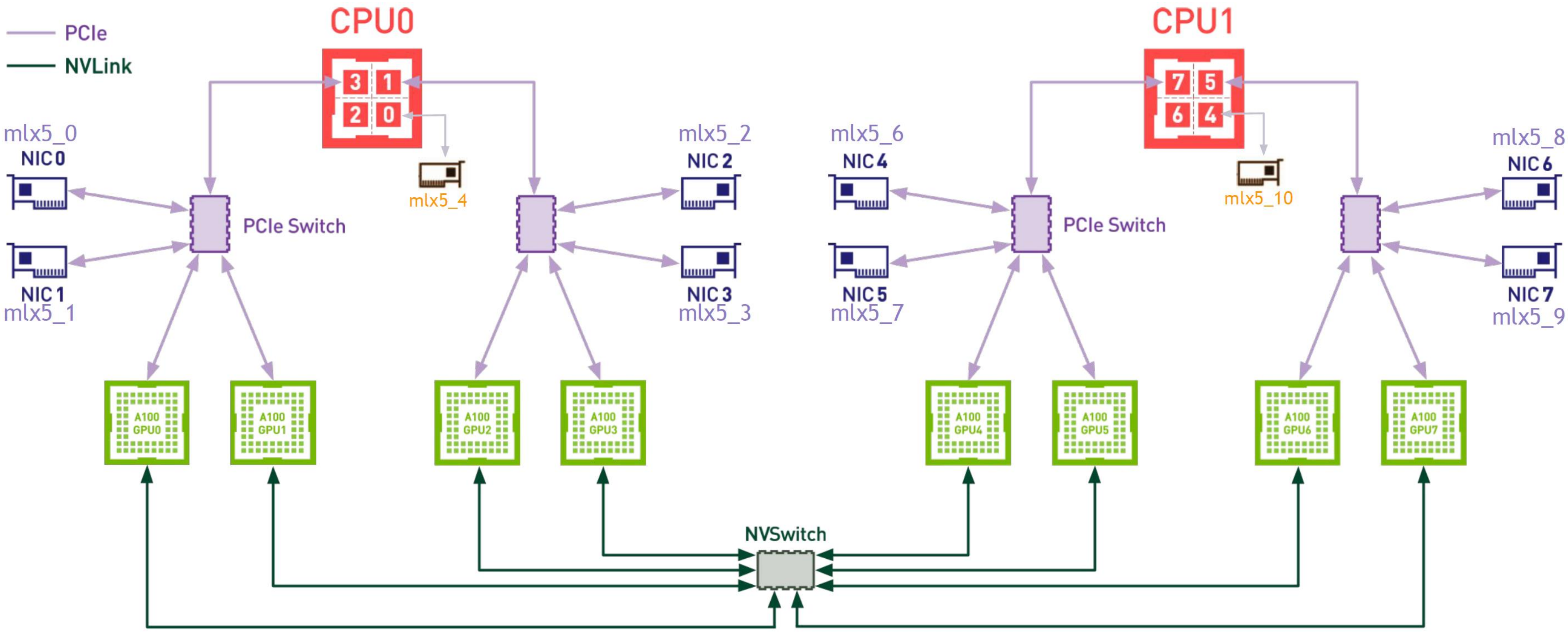
```
__global__ void MultiGPUSpinLock(int current_gpu_id, uint32_t n_gpu,
                                 uint32_t* lock) {
  int counter =
      atomicAdd_system(lock, 1);     /* increase the counter on the spin lock */
  while (counter < n_gpu)                                /* wait for other GPUs */
    counter = atomicCAS_system(lock, n_gpu, n_gpu);
  counter =
      atomicAdd_system(lock, 1); /* increase the counter again to notify GPU 0
                                    that the current GPU has finished waiting */
  if (current_gpu_id != 0) return;                     /* quit if not from GPU 0 */
  while (counter < 2 * n_gpu)    /* wait for all the other GPU's notification */
    counter = atomicCAS_system(lock, 2 * n_gpu, 2 * n_gpu);
  *lock = 0;                                         /* GPU 0 resets the spin lock */
}
```

Topology of GPU interconnection

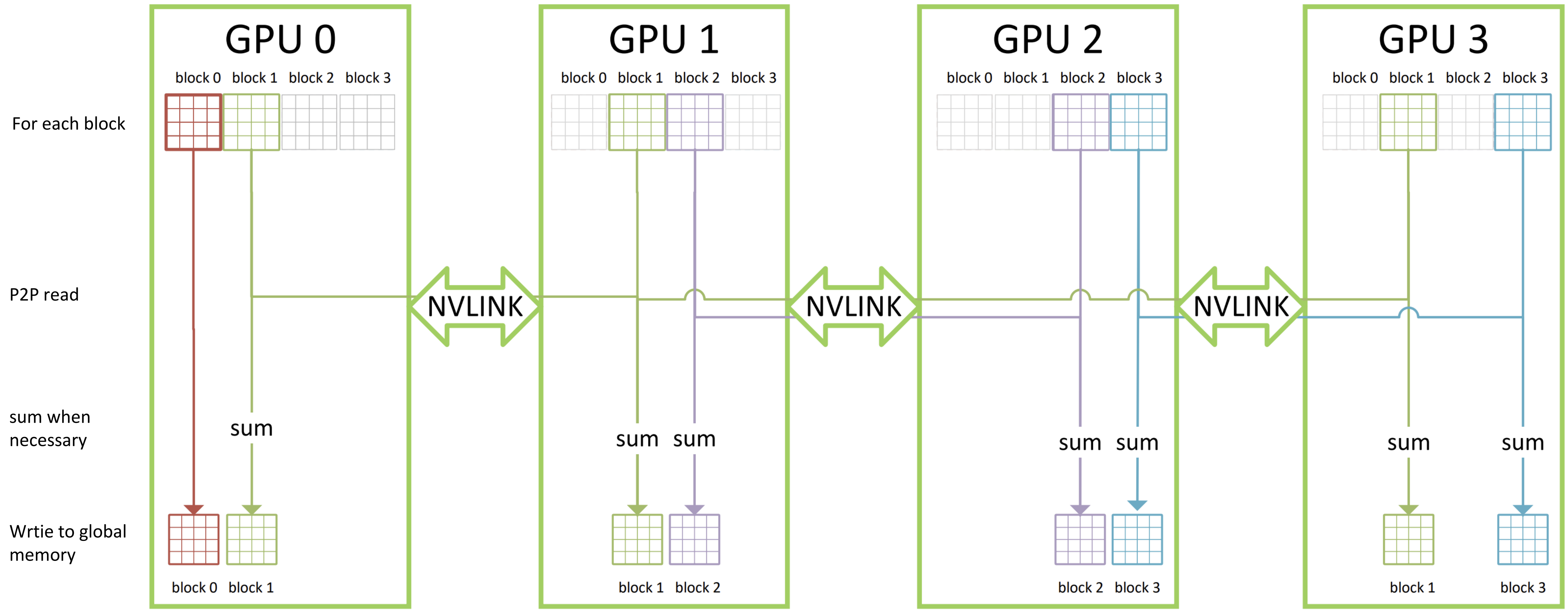


DGX A100

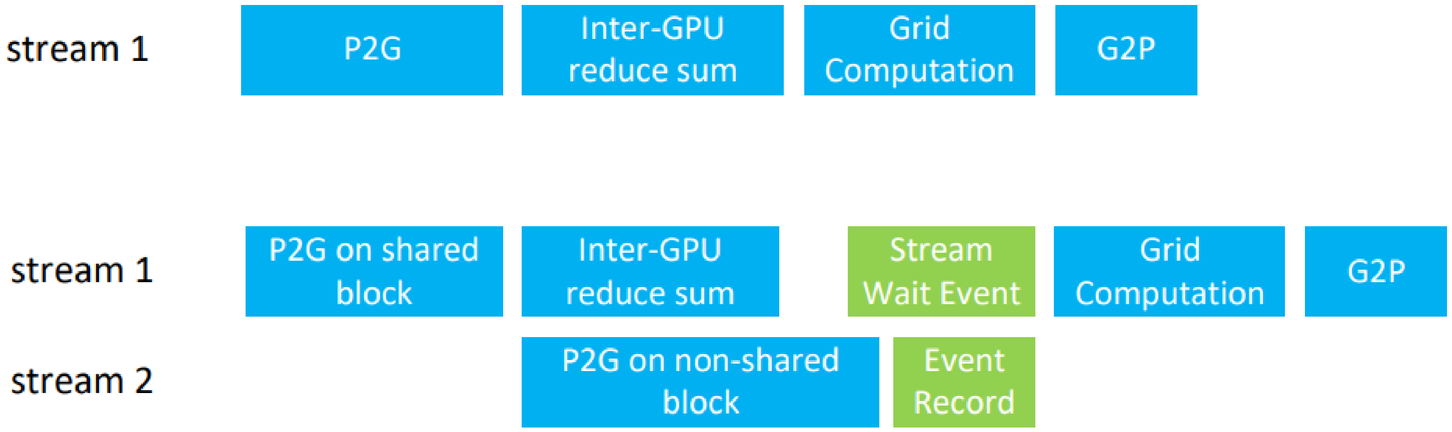
High-level Topology Overview

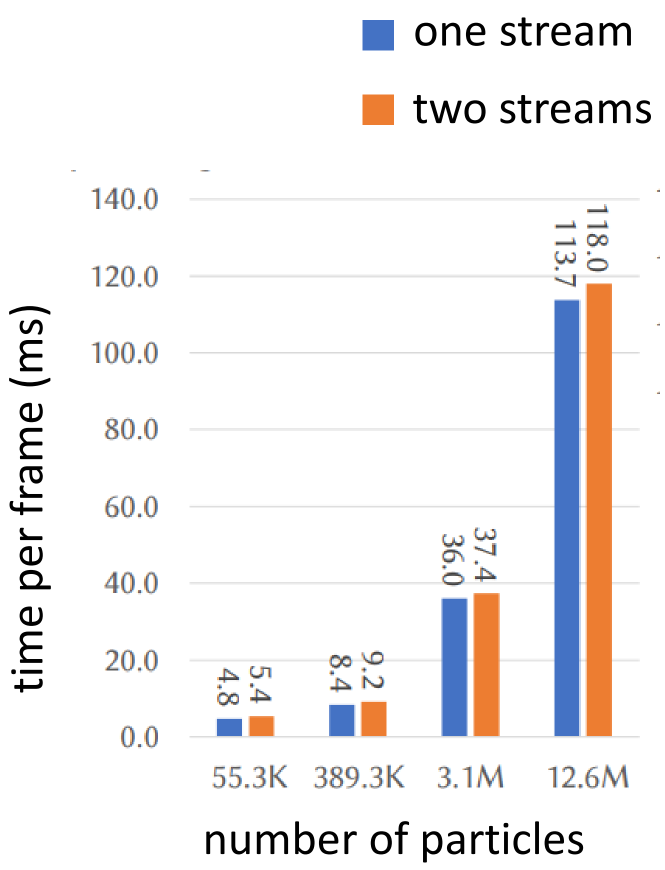


Implementation of inter-GPU reduce sum



Overlapping communication and compute





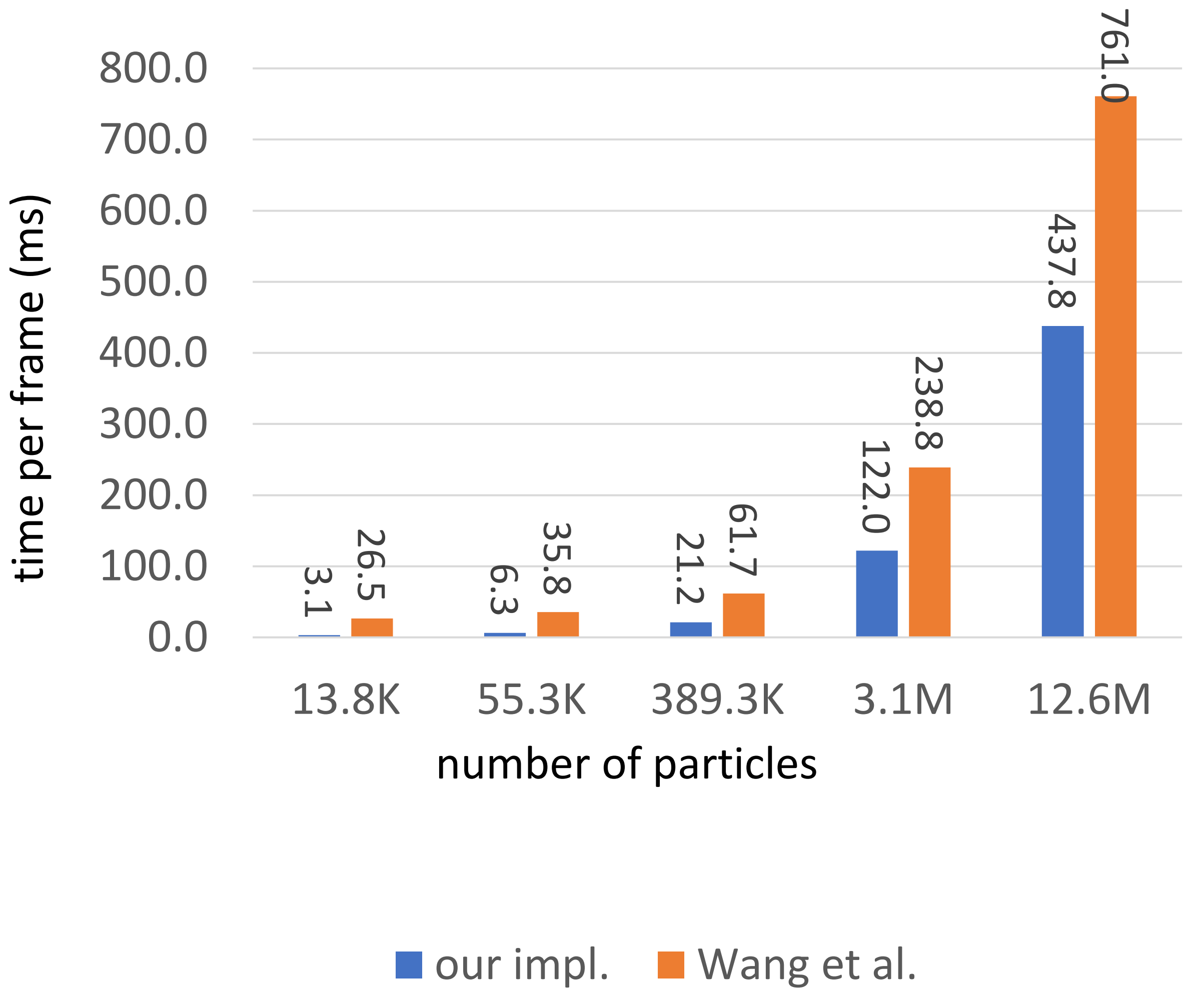
Principles of multiple GPU MPM

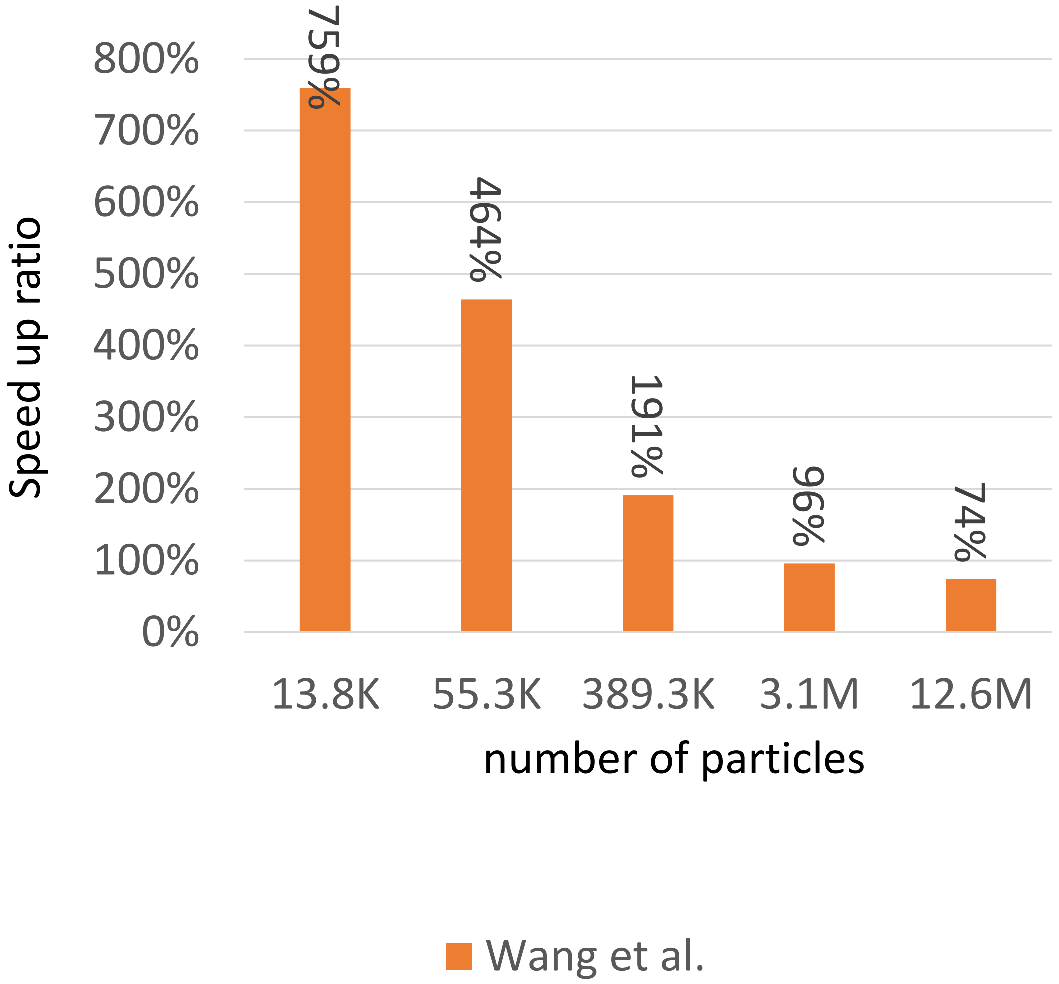
- Minimizing the number of transfers and synchronizations between GPUs
- Minimizing the amount of data transferred between the GPUs and the subsequent computations.
- Use in-kernel peer-to-peer (P2P) read/write operations for inter-GPU communication
- Overlap P2P data transfer and computation through warp-interleaved execution when using NVLINK.

Content

- Background
- Single GPU
- Multiple GPU
- Benchmark and demo

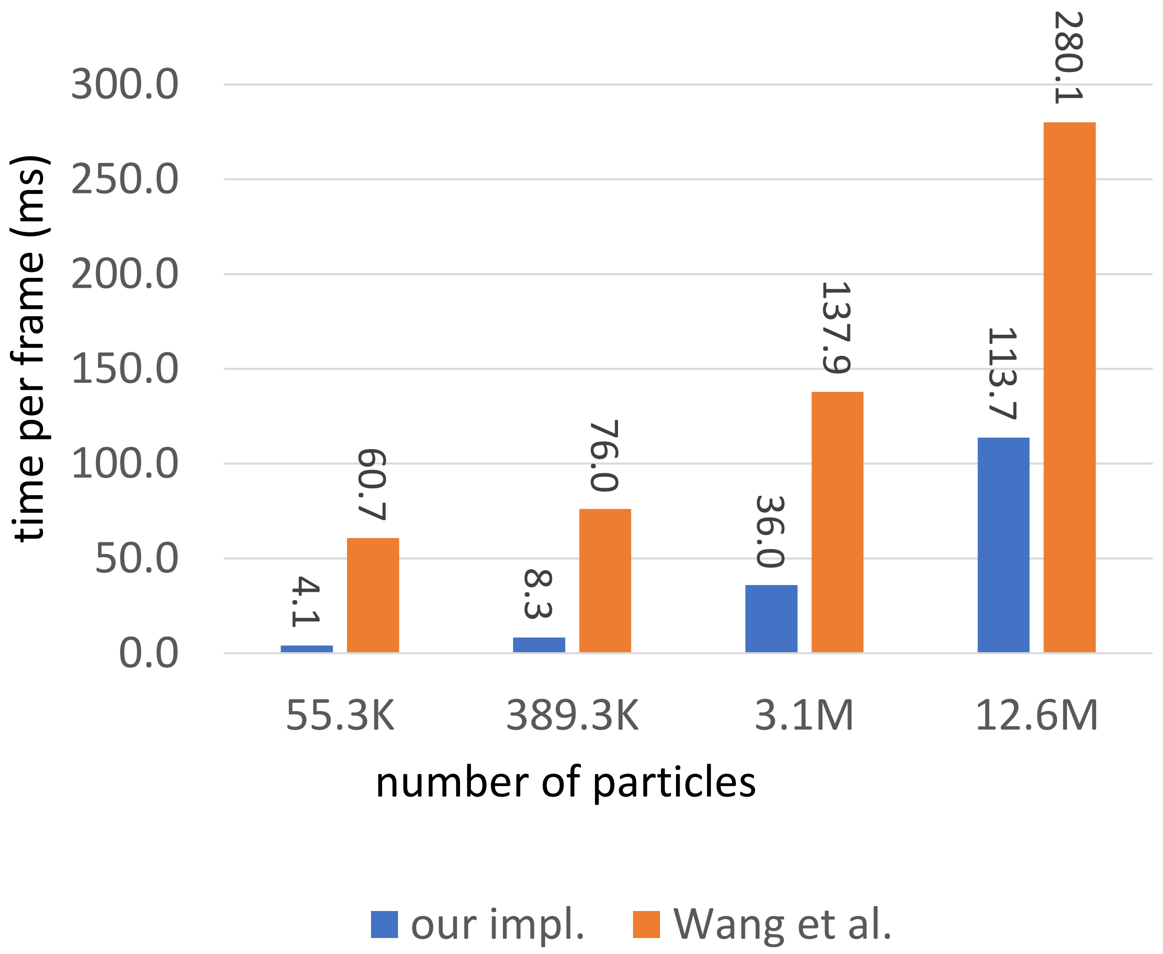
Benchmark – Single GPU

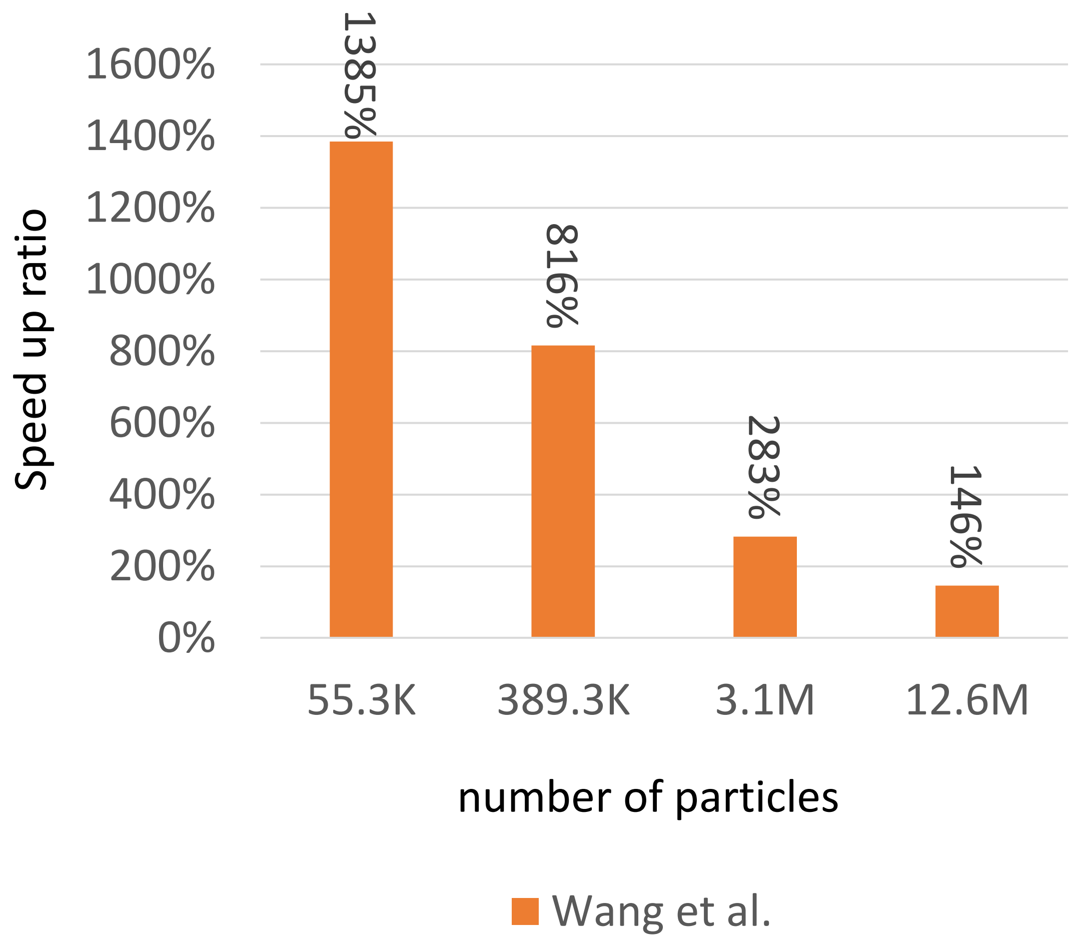




*Tested on NVIDIA Tesla V100

Benchmark – Four GPUs

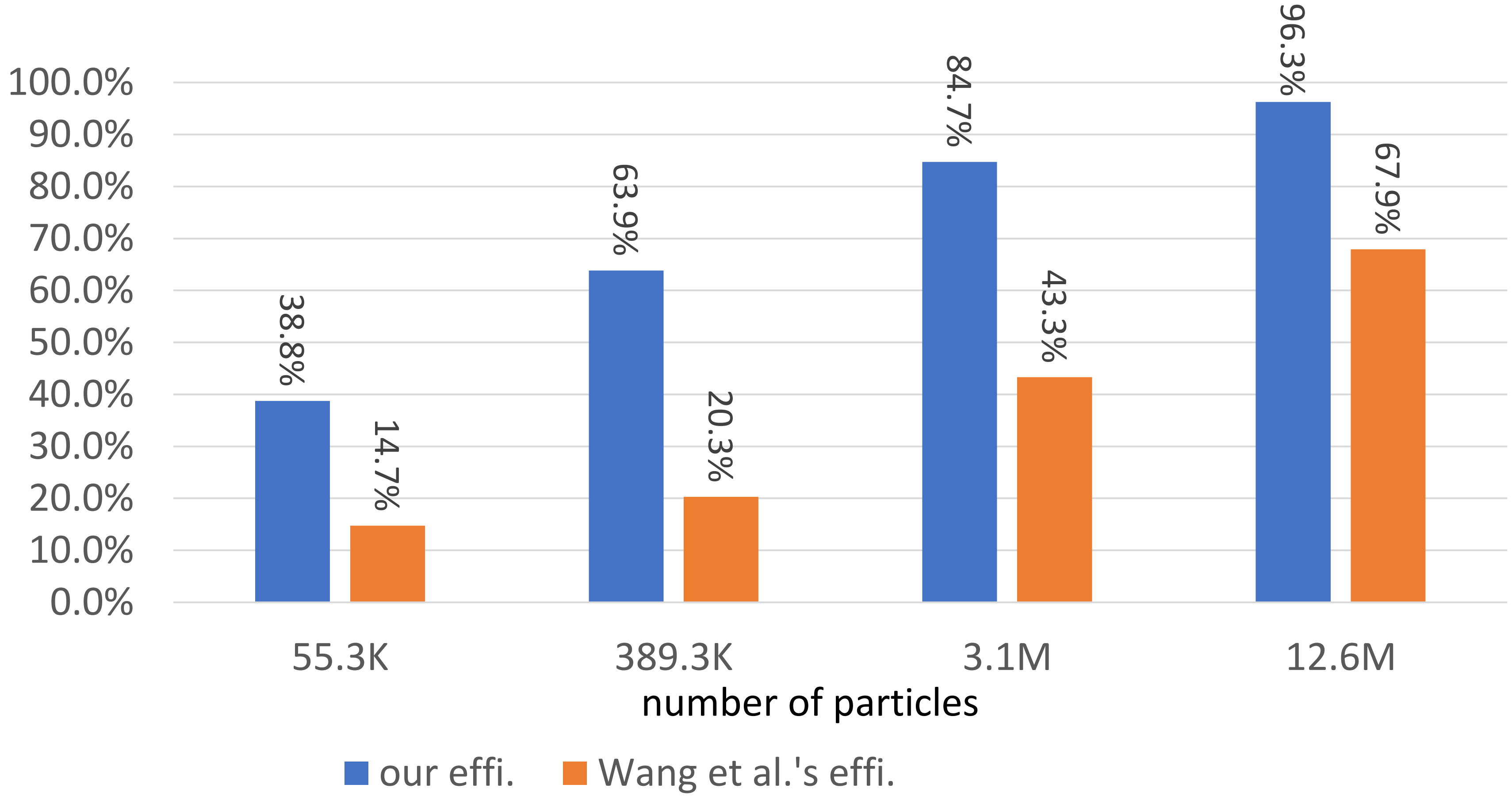




*On 4 x NVIDIA Tesla V100 with NVLINK

Benchmark – MultiGPU efficiency

$$e = \frac{time(1\ GPU)}{time(n\ GPU) * n}$$



*On 4 x NVIDIA Tesla V100 with NVLINK

Principles towards Real-Time Simulation of Material Point Method on Modern GPUs

Benchmark Scenarios