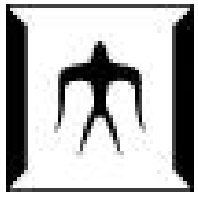
Realizing Extremely Large-Scale Stencil Applications on GPU Supercomputers

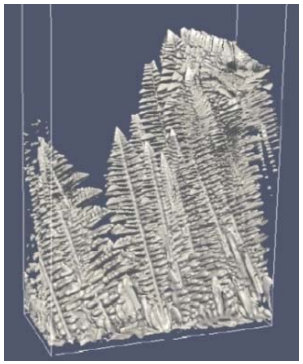
Toshio Endo, Yuki Takasaki, Satoshi Matsuoka
GSIC, Tokyo Institute of Technology (東京工業大学)



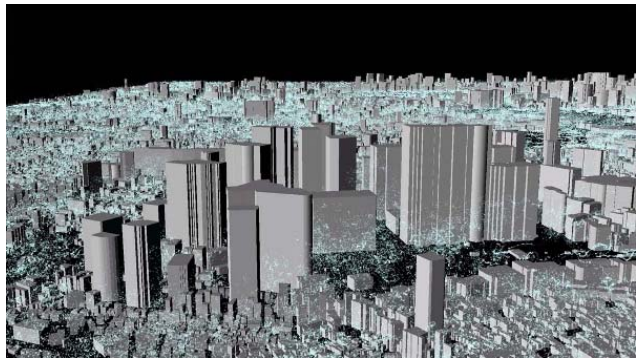
Stencil Computations

Important kernels for various simulations (CFD, material…)

ASUCA weather simulator

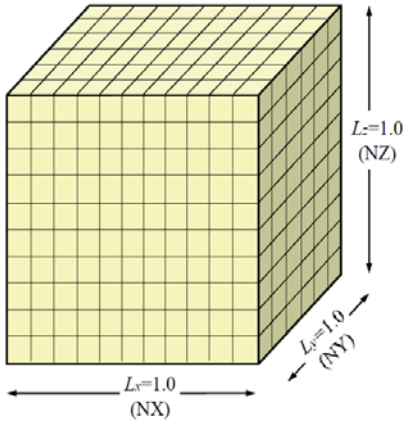


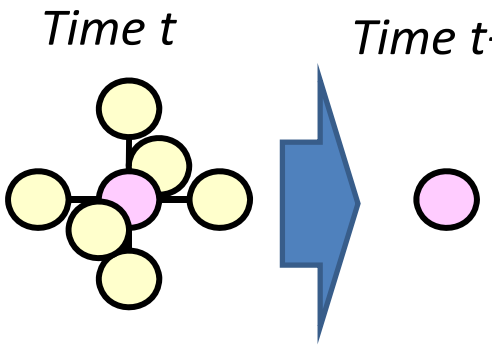
Phase-Field computation (2011 Gordon Bell)



Air flow simulation

=





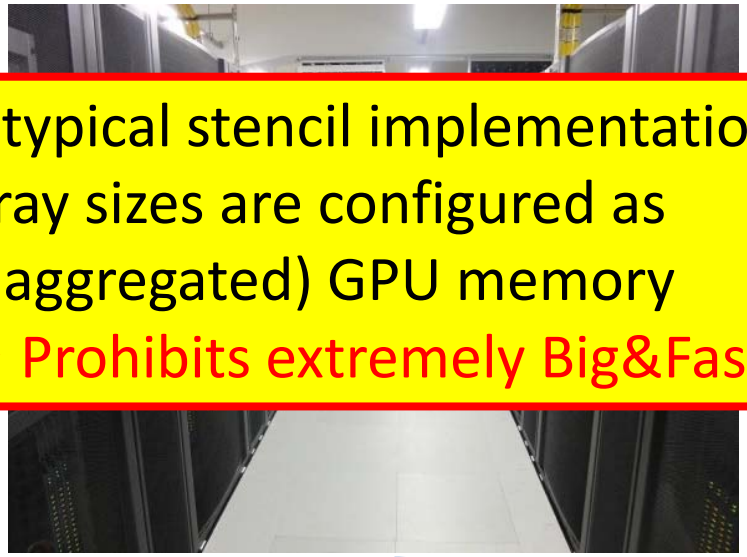
Memory intensive computations →

Highly successful *in speed*

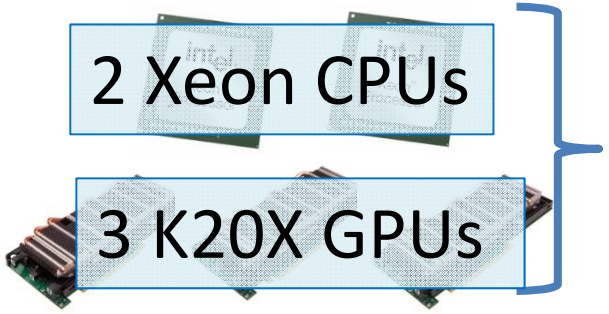
But not *in scale*

CPU-GPU Hybrid Supercomputers with Memory Hierarchy

Tokyo Tech TSUBAME2.5 Supercomputer

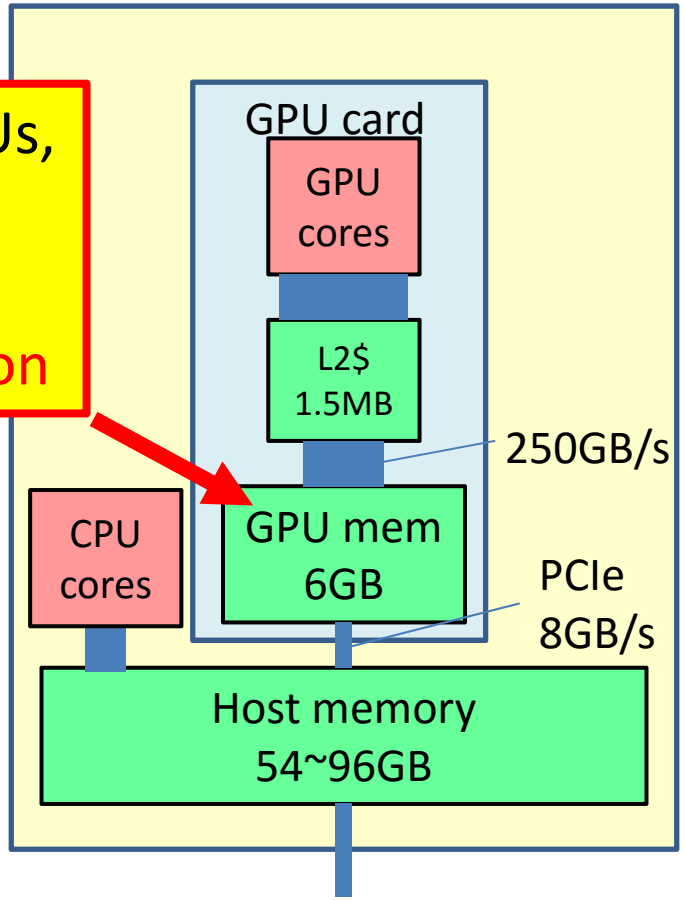


In typical stencil implementations on GPUs, array sizes are configured as
< (aggregated) GPU memory
➔ Prohibits extremely Big&Fast simulation

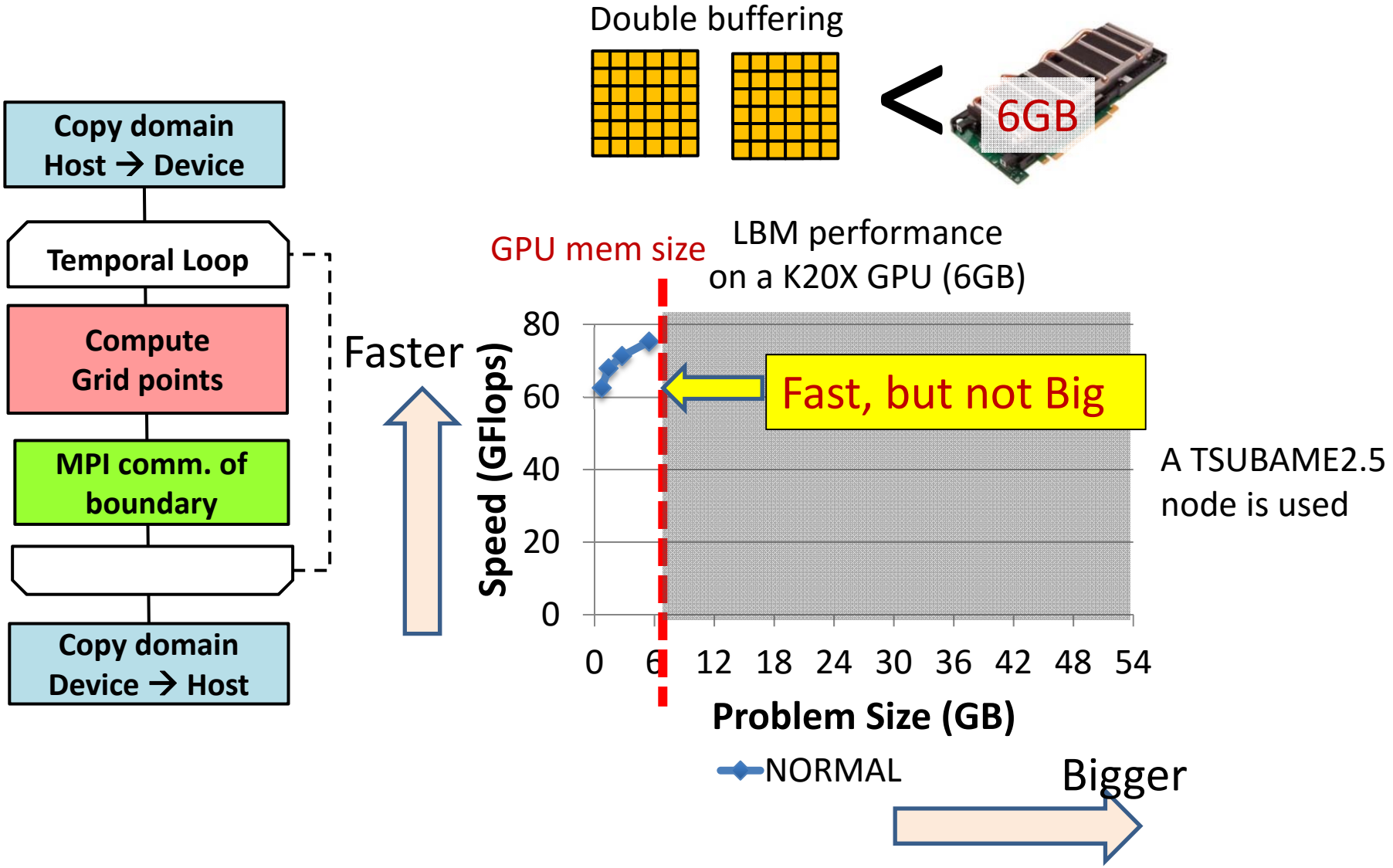


× 1408

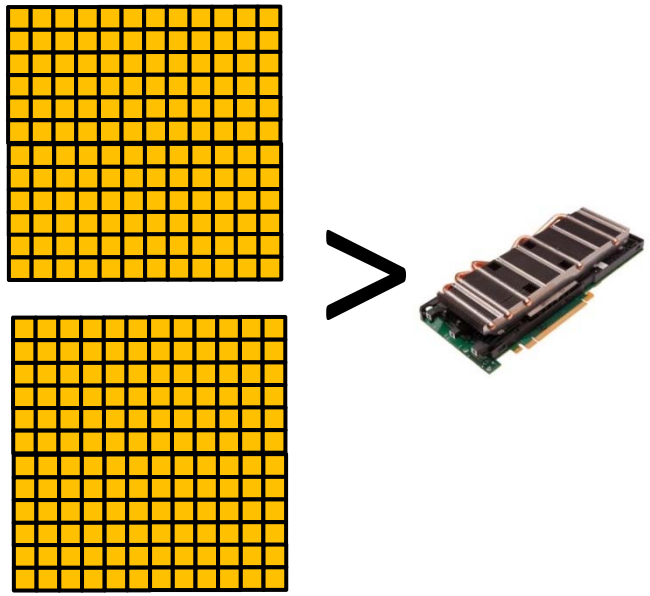
Node Architecture (Simplified)



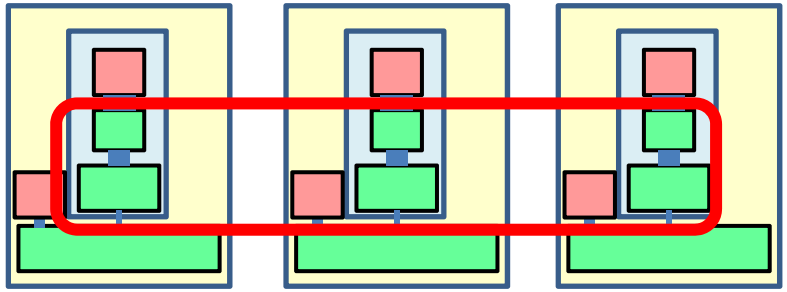
Stencil Code Example on GPU



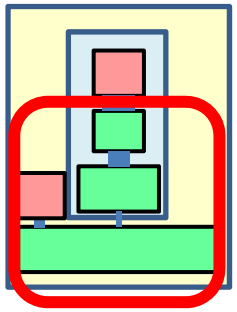
How Can We Exceed Memory Size?



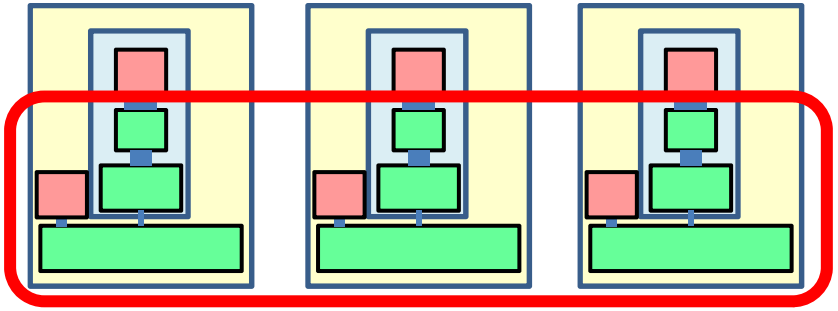
(1) Using many GPUs



(2) Using capacity of host memory



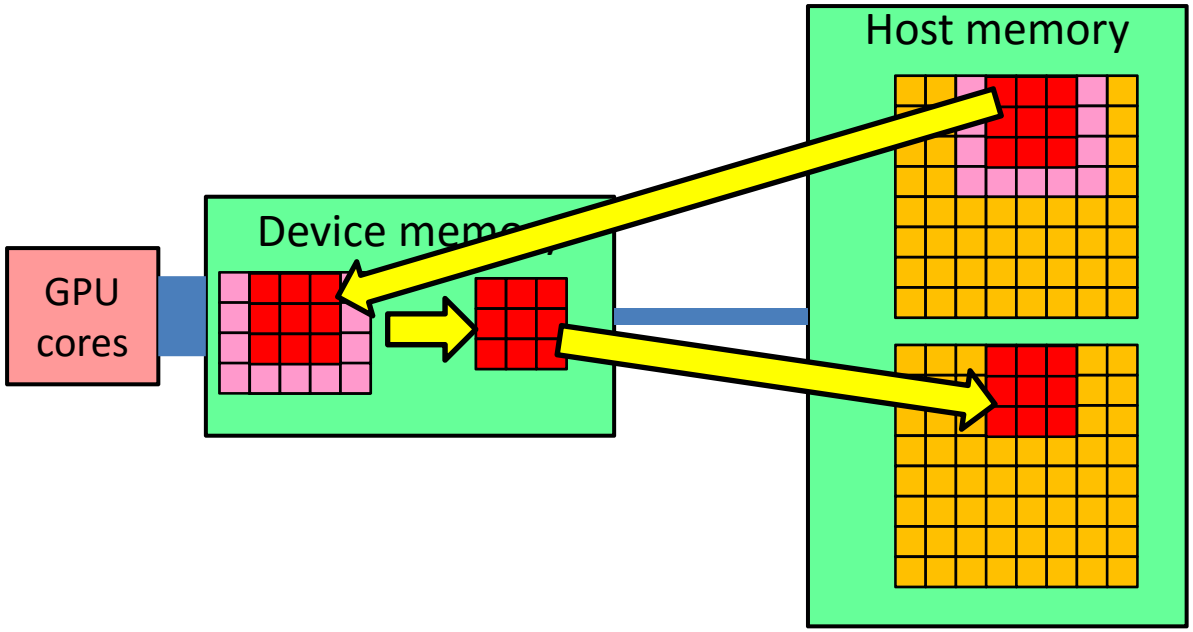
(3) Using both



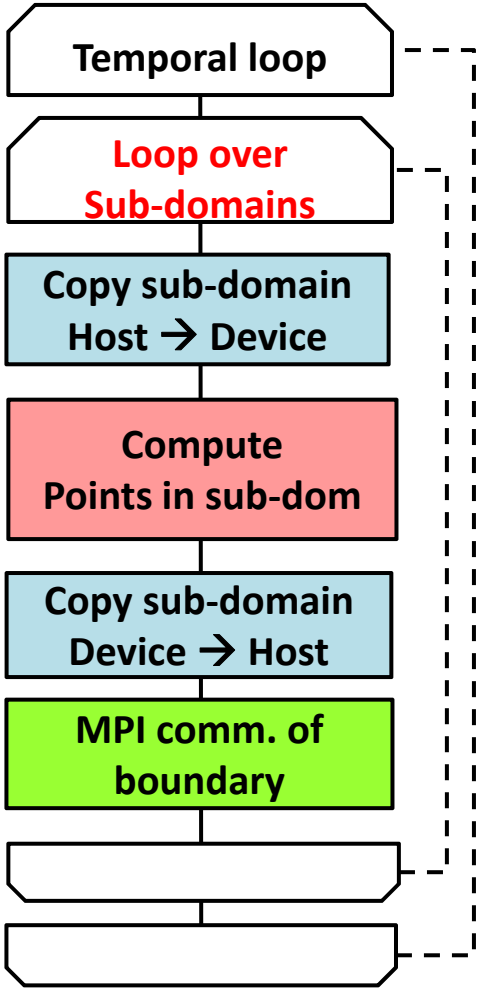
Motivating Example:
What if We Exceed GPU memory *Simply*? (1)

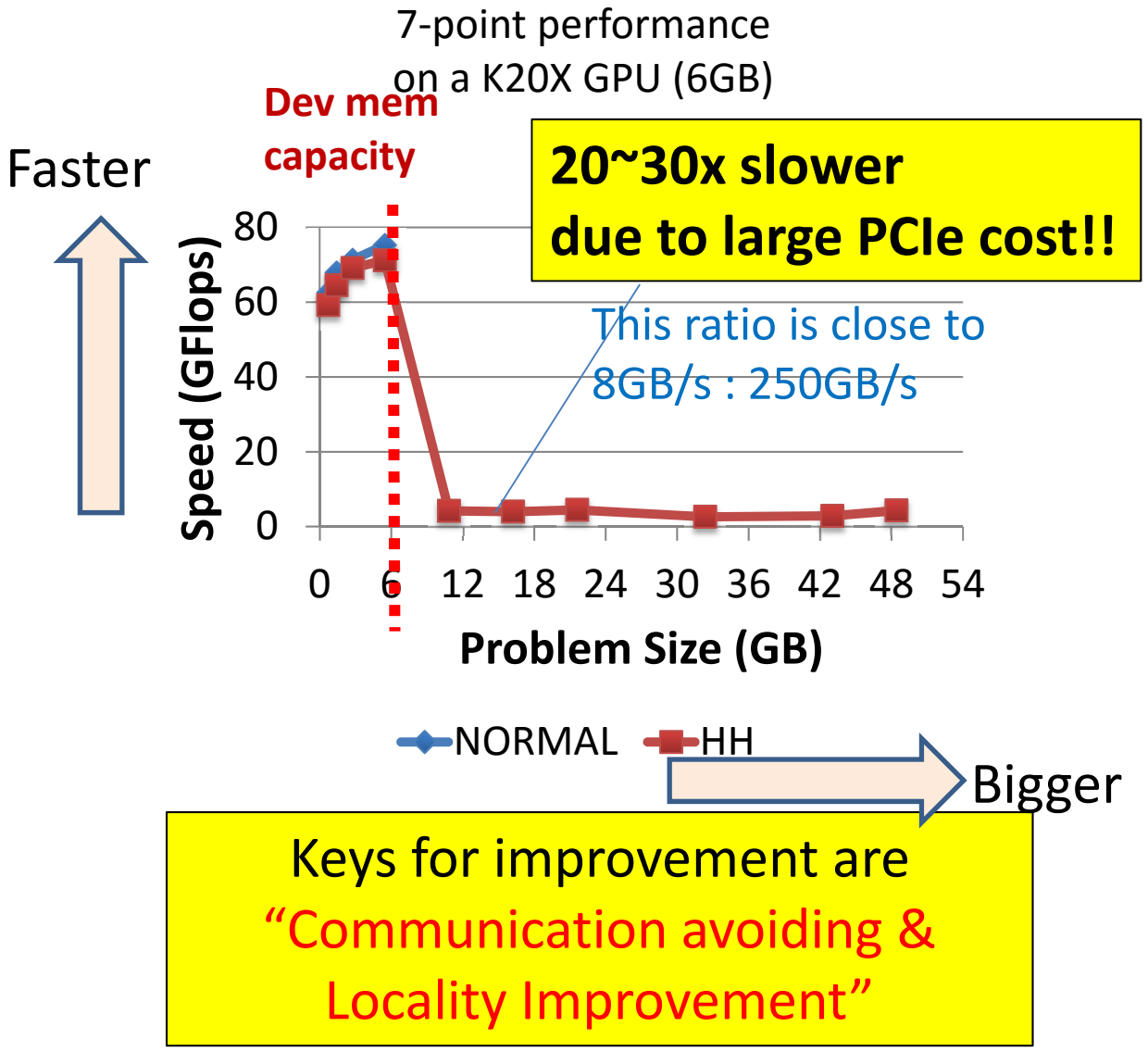
- A simple method is:
  - Put domain data on host memory
  - We divide the domain into small sub-domains (or spatial block)
  - Repeat

    Copy a "sub-domain" into GPU → Compute → Copy back results



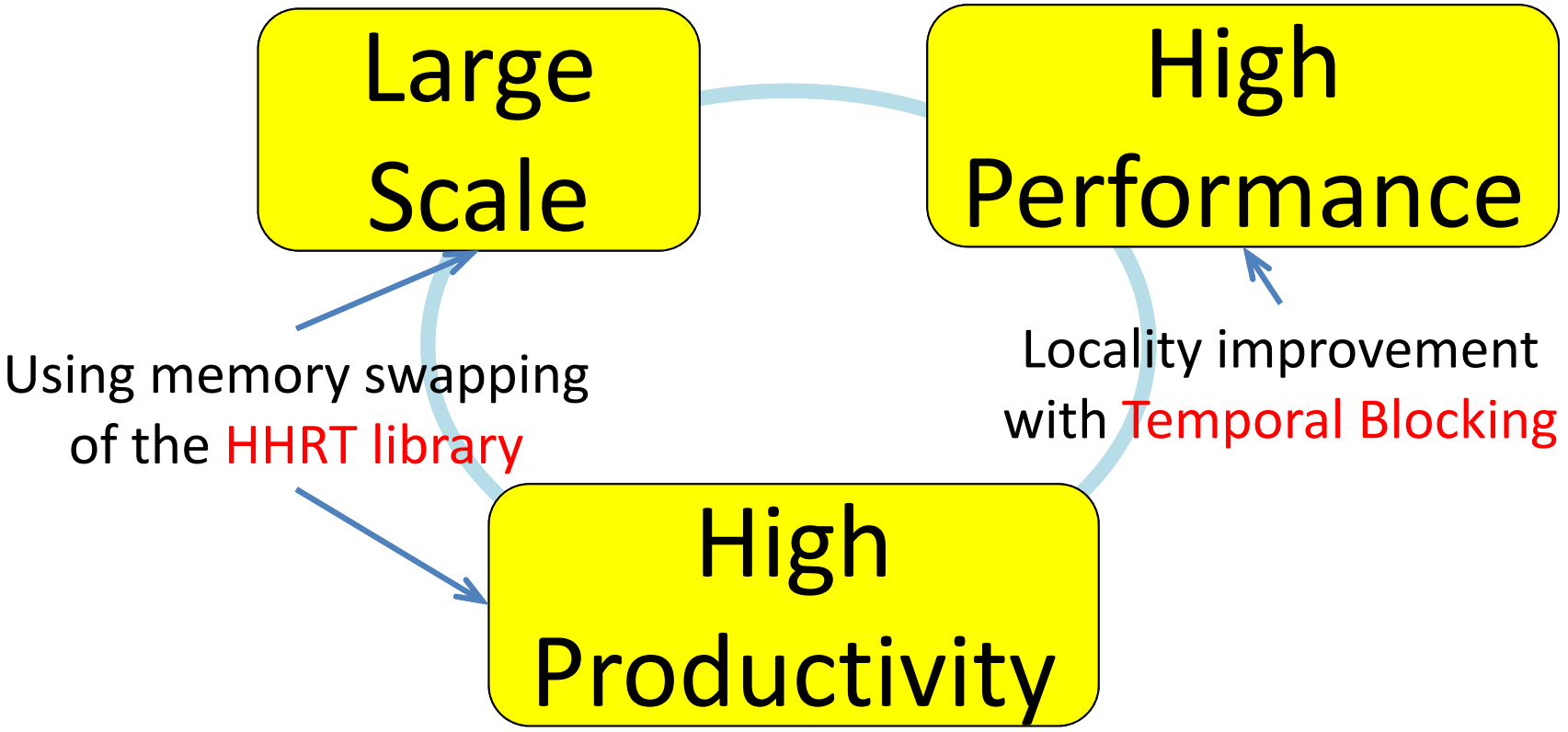
Motivating Example: What if We Exceed GPU memory *Simply*? (2)





Goals of This Work

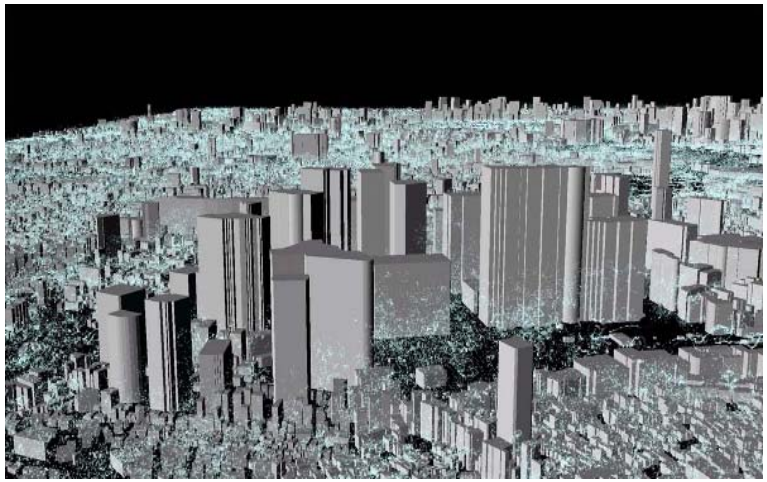
When we have existing apps, we want to realize followings



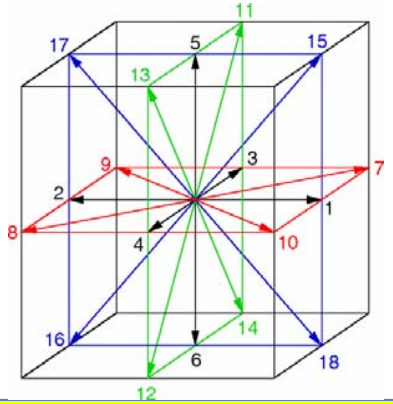
Co-design approach that spans
Algorithm layer, Runtime layer, Architecture layer

Current Target GPU Stencil Application

- City-Wind Simulation by Naoyuki Onodera
  - Based on Lattice-Botlzmann method
  - Written in MPI+CUDA
  - ~12,000 Lines of code
  - 600TFlops with ~4,000 GPUs



D3Q19
Model
(19point stencil)



In Original design,
"Total Array size < Total GPU memory"
How can we exceed this limitation?

Contributions

- For real existing applications, the followings are realized
  - *[Scale]* > GPU memory size is realized
  - *[Performance]* Compared with smaller cases, up to 85% performance is obtained
  - *[Productivity]* Required modification of ~150 lines for basic change, and ~1000 lines for optimization

Contents

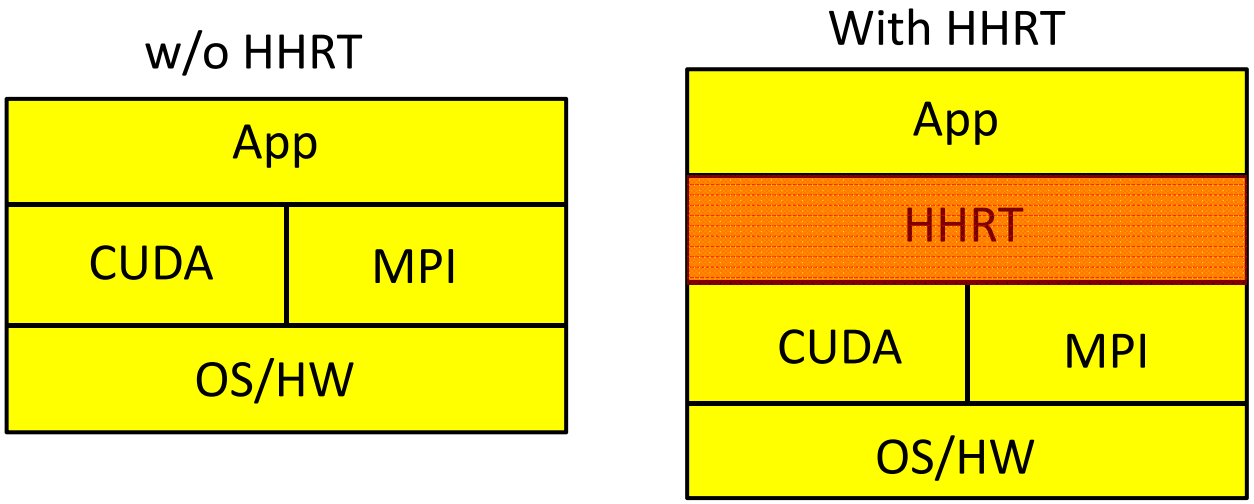
- HHRT library
  - Expands available memory capacity by data swapping
- Temporal blocking
  - Optimizations of stencils for locality improvement
- Combining the above two on real applications
- Results

Contents

- HHRT library
  - Expands available memory capacity by data swapping
- Temporal blocking
  - Optimizations of stencils for locality improvement
- Combining the above two on real applications
- Results

The HHRT Runtime Library for GPU Memory Swapping [Endo, Jin Cluster 14]

- HHRT supports applications written in CUDA and MPI
  - HHRT is as a wrapper library of CUDA/MPI
  - Original CUDA and MPI are not modified
  - Not only for stencil applications



Functions of HHRT

(1) HHRT supports overprovisioning of MPI processes on each GPU

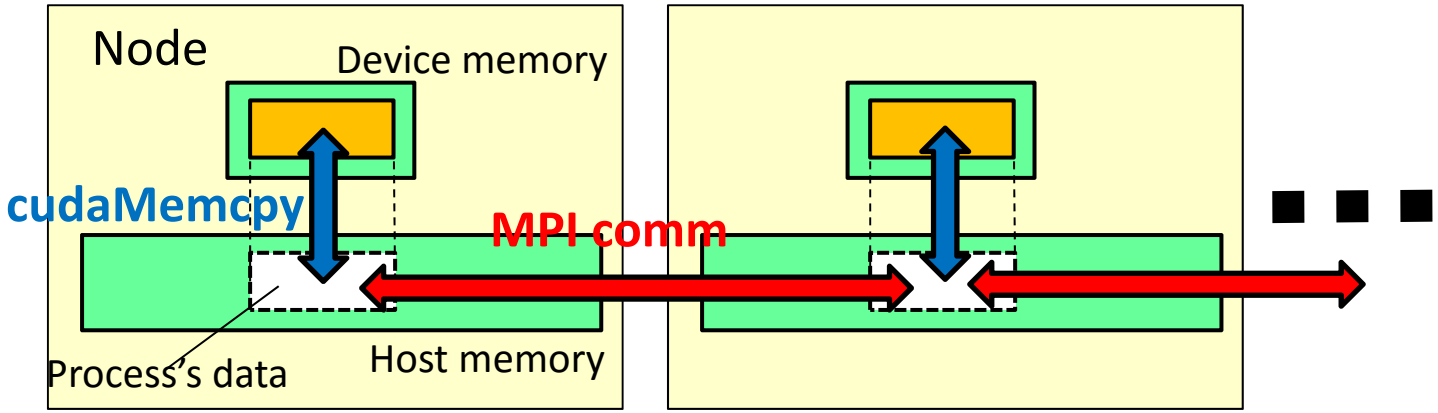
- Each GPU is shared by *m* MPI processes

(2) HHRT executes implicitly memory swapping between device memory and host memory

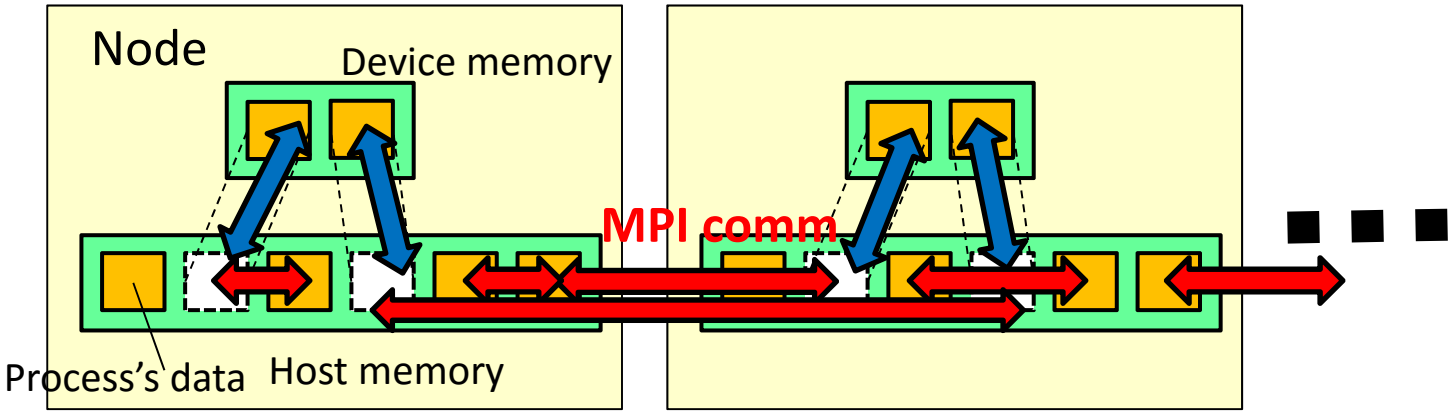
- *"process-wise"* swapping
- OS-like "page-wise" swapping is currently hard, without modifying original CUDA device/runtime

Execution model of HHRT

w/o HHRT (typically)



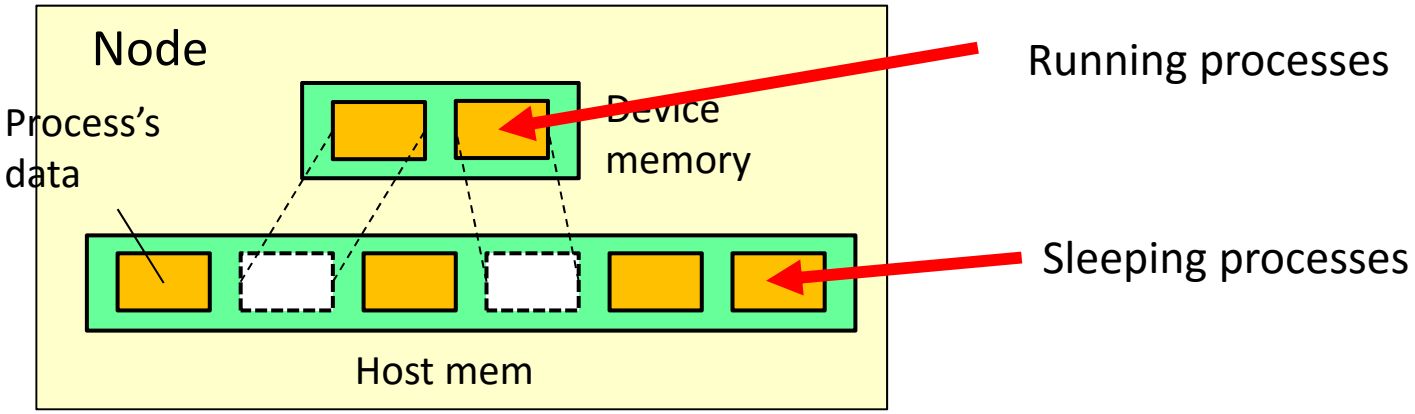
With HHRT



m MPI processes share a single GPU

In this case, m=6

Processes on HHRT



- We suppose

  s < Device-memory-capacity < $m\ s$

  s: Size of data that each process allocates on device memory

  m: The number of processes sharing a GPU

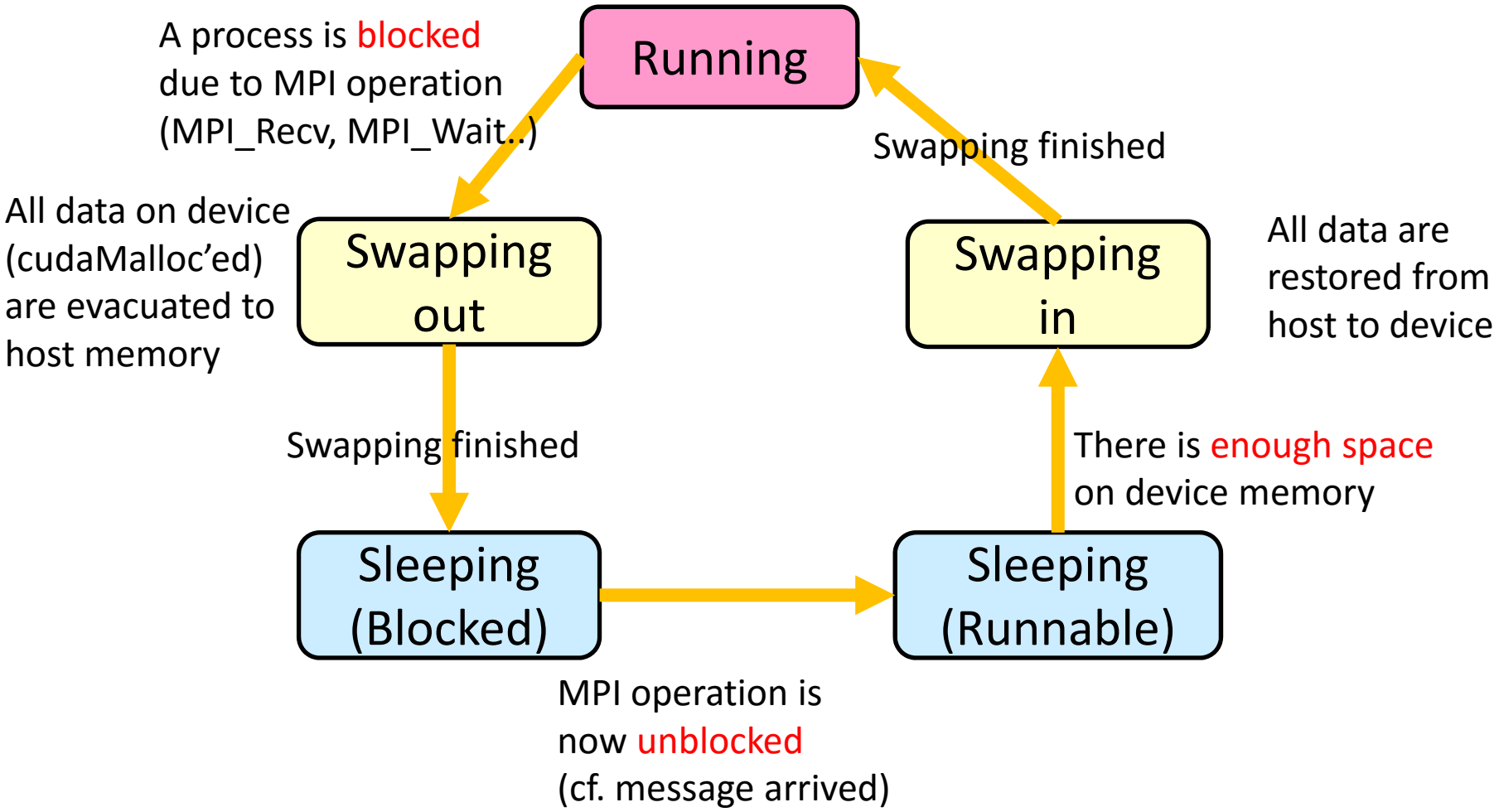
→ We can support larger data size than device memory in total

- We cannot keep all of *m* processes running

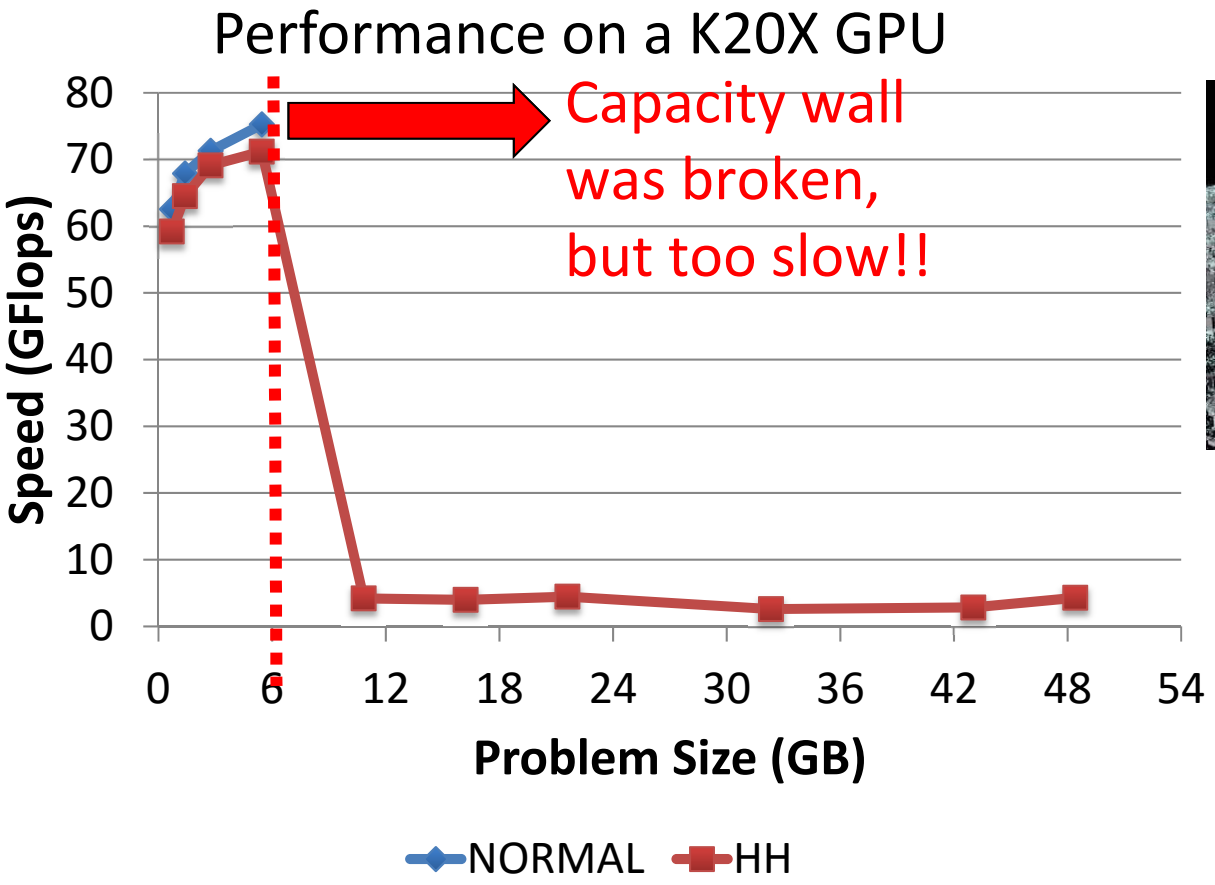
→ HHRT makes some processes "sleep" forcibly and implicitly

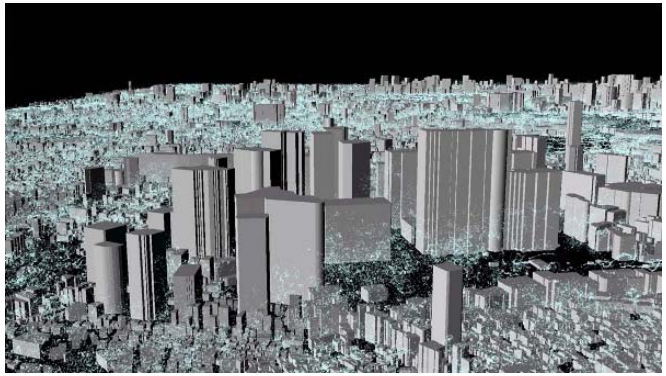
- Blocking MPI calls are "yield" points

State Transition of Each Process



Running LBM Code on HHRT





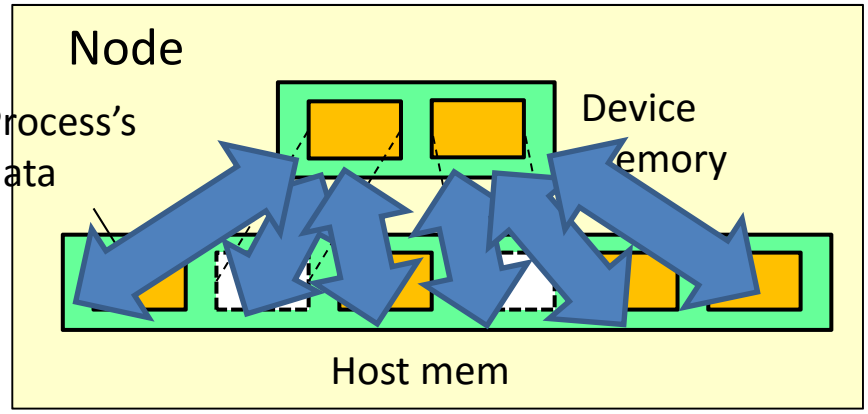
We can support “larger problem sizes > GPU memory” with HHRT, but too slow → We need aggressive optimization!

Contents

- HHRT library
  - Expands available memory capacity by data swapping
- Temporal blocking
  - Optimizations of stencils for locality improvement
- Combining the above two on real applications
- Results

Why Slow if We Use Host Memory?

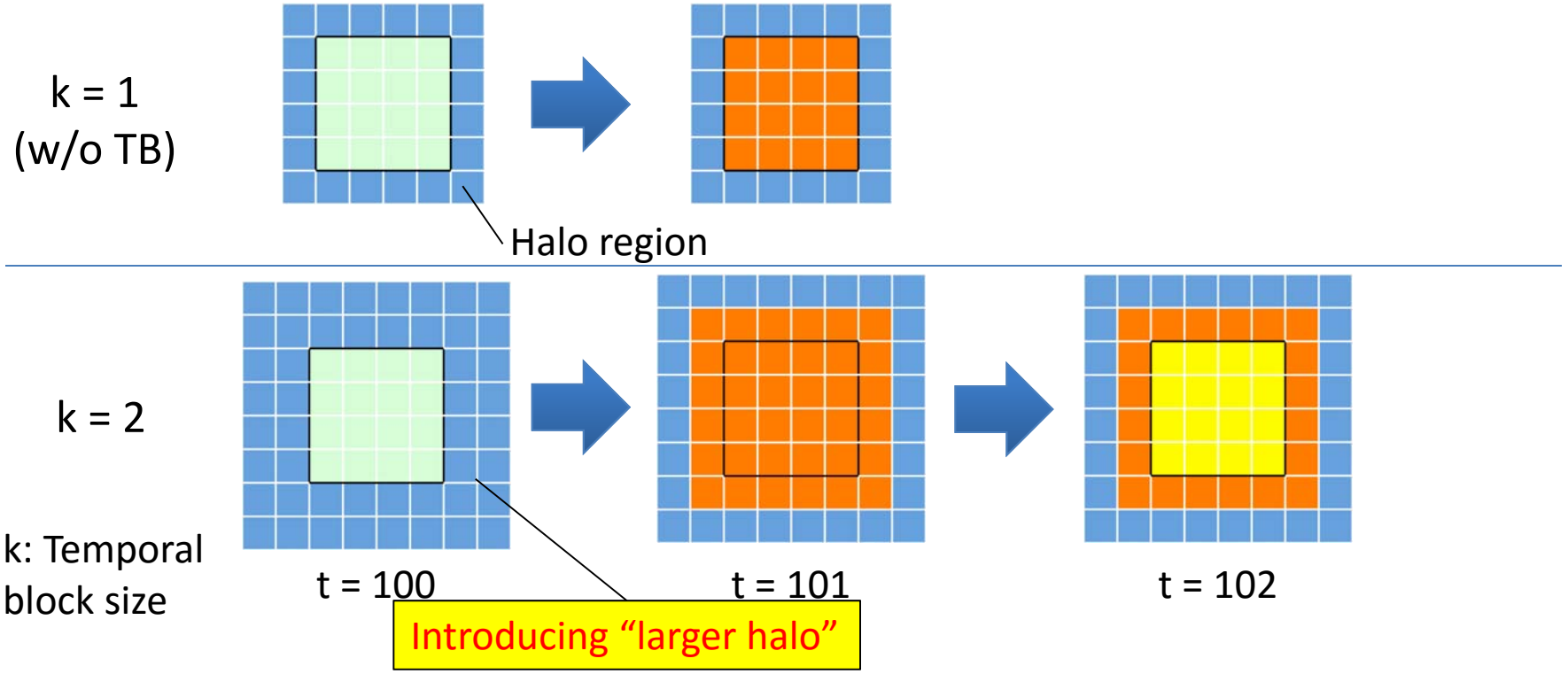
- Each process can suffer from heavy memory swapping costs, every iteration
  - This corresponds to transfer of the entire process'es sub-domain between GPU and CPU
- This is done automatically, but too heave costs are not hidden



- This is due to lack of locality of stencil computations
  - Array data are swapped out every iteration
- We need optimizations to improve locality!!

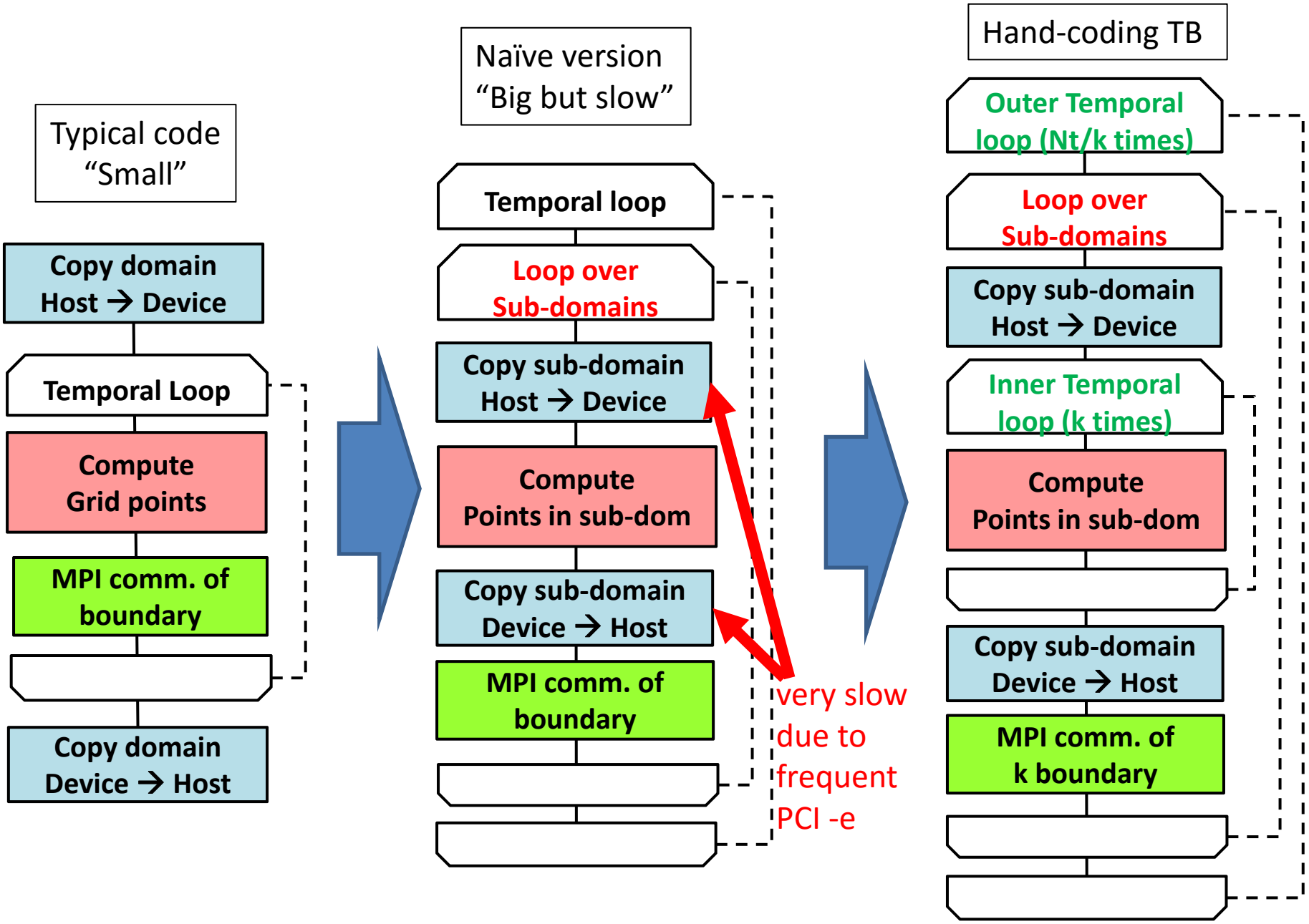
Temporal Blocking (TB) for Locality Improvement

- ***Temporal blocking***: When we pick up a sub-domain, we do **k-step update at once** on it on a small block, before going to the next sub-domain [Wolf 91]



- Mainly used for cache optimization [Wonnacott 00] [Datta 08] ... (k=2~8)
- We use it to reduce PCIe commucation (k=10~200)

Code Structure with TB



What Makes TB Code Complex?

Differences between “typical” and “hand-coding TB”

(1) “Sub-domain” loop is introduced → Automated by HHRT runtime

(2) Temporal loop is divided into “inner” and “outer”

(3) Considering larger “Halo”

(4) PCIe and MPI comm is done out of “inner” loop

→ Yes, we currently rely on Code refactoring!

Contents

- HHRT library
  - Expands available memory capacity by data swapping
- Temporal blocking
  - Optimizations of stencils for locality improvement
- Combining the above two on real applications
- Results

Implementing Temporal Blocking on HHRT

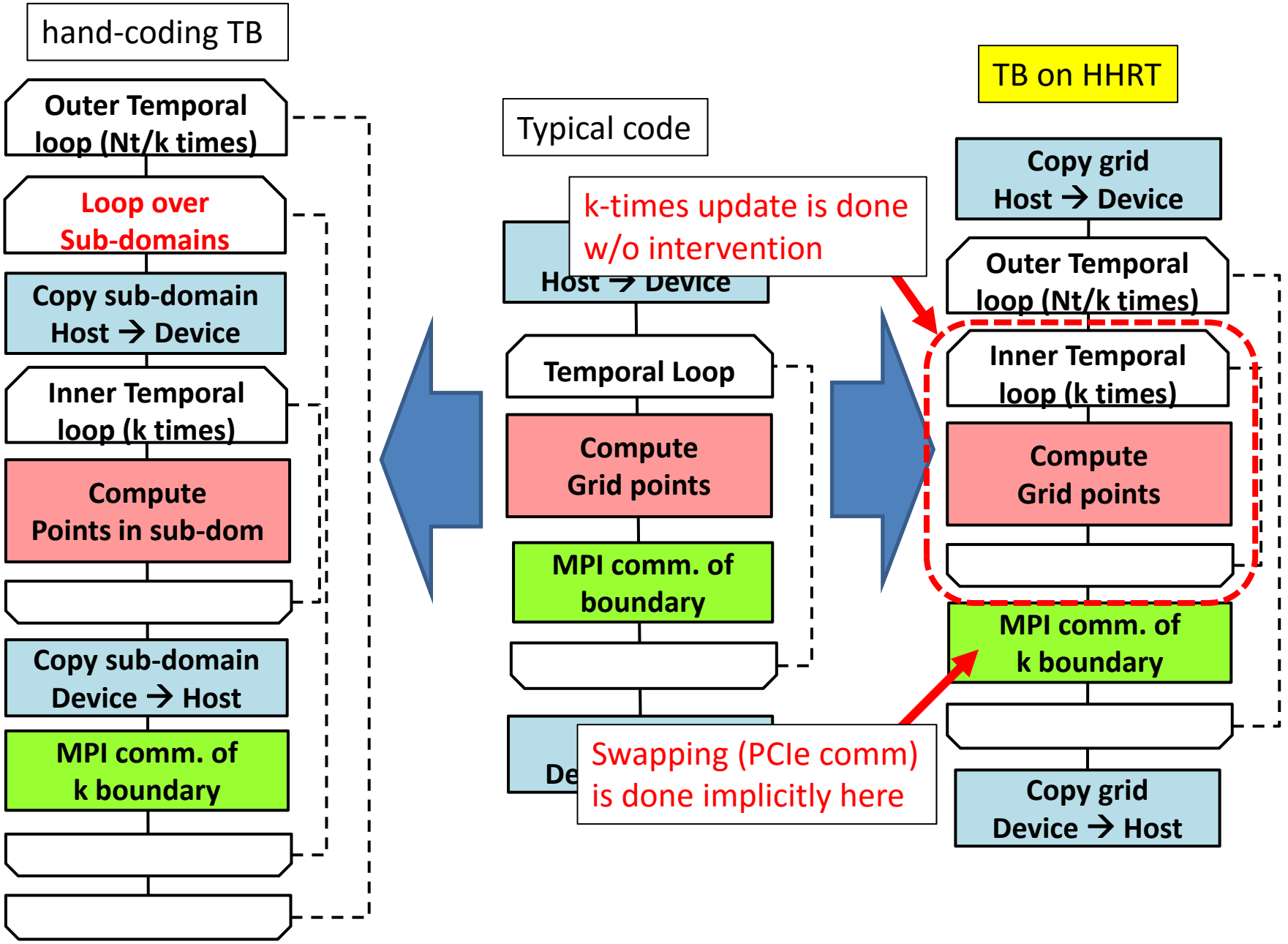
How do we reduce refactoring costs of existaing apps?

- How do we map multiple sub-domains to a GPU?
  - w/o HHRT: 1GPU ← 1 process ← m sub-domains
  - With HHRT: 1GPU ← m processes ← m domains

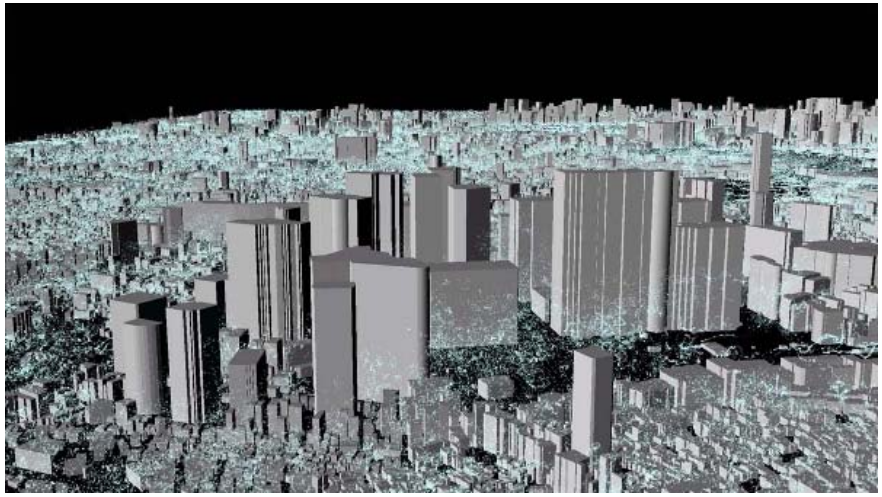
    Each process maintains only one domain

    We don't need additional sub-domain loop
- How is domain data moved?
  - w/o HHRT: PCIe comm is done explicitly
  - With HHRT: Implicitly within MPI comm
- On the other hand, doubly nested temporal loops should be (still) written in hand

Implementing Temporal Blocking on HHRT (2)

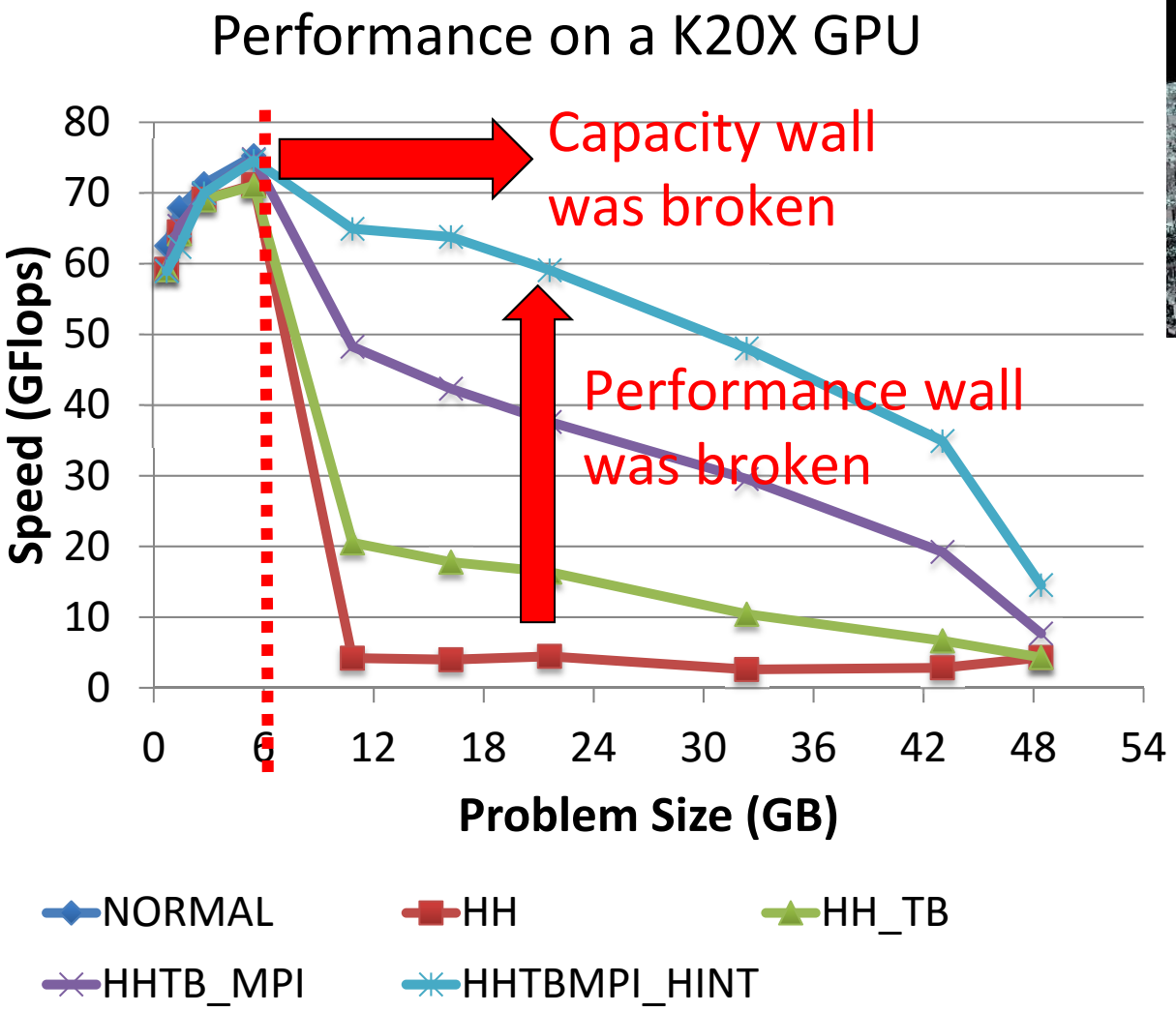


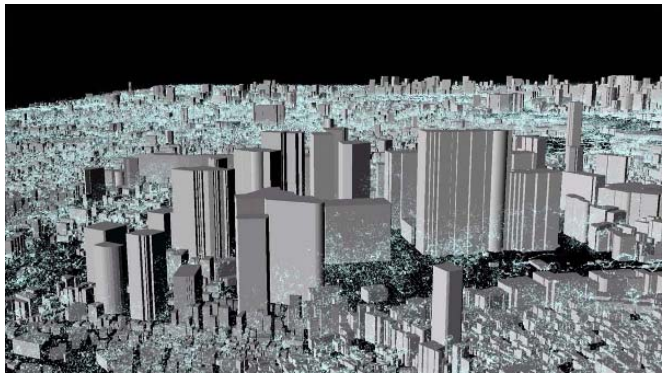
Code Refactoring



- Original: ~12,000 lines (MPI+CUDA)
  - ~4000 lines correspond to computation kernels
- Basic code change: ~150 lines
  - Introducing outer/inner temporal loop
- Communication optimization: ~900 more lines
  - X, Y, Z boundary communications use MPI_Waitall

Performance of Real LBM Code with Larger Problem Sizes





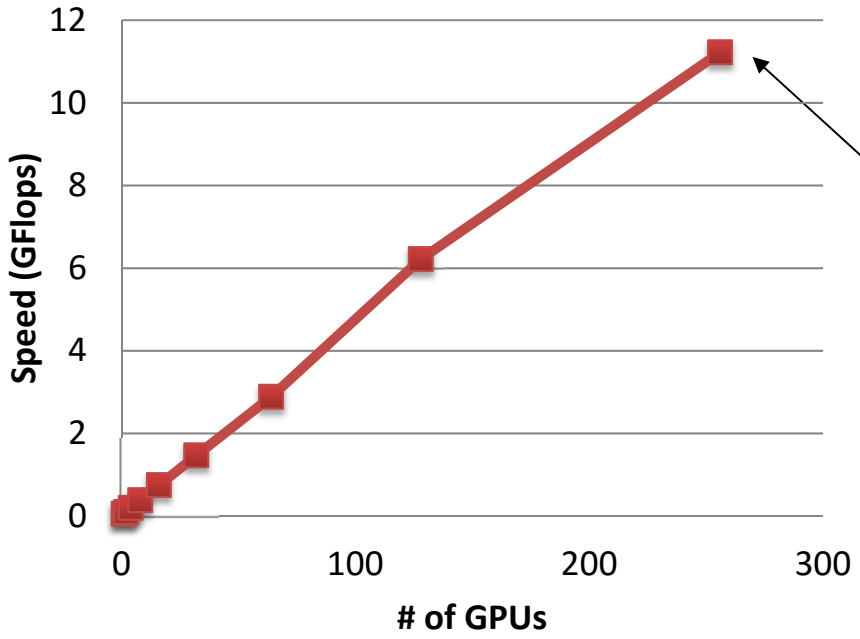
>15x Speed-up with temporal blocking!

First step toward "Extreme Big&Fast" simulation

Multi GPU/Node Performance

TSUBAME2.5 (1GPU per GPU)

Weak scalability (11GB > 6GB per node)



Performance : 11.2TFlops, 5.9TB/s
Problem Size : 2.8TB

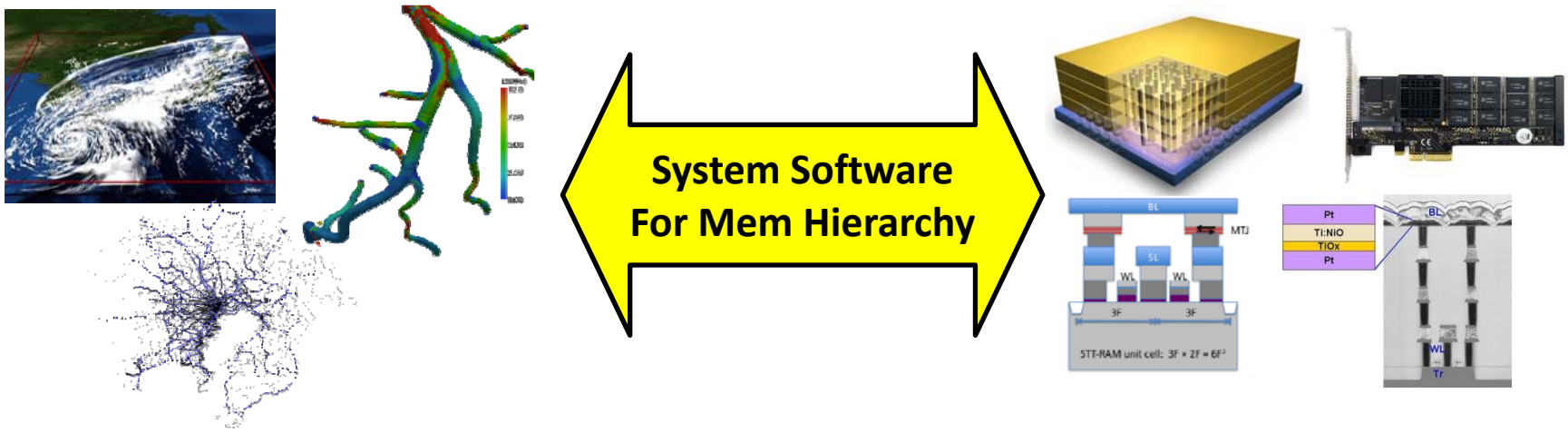
Good weak scalability
x203 speedup with 256 GPUs
(though 1GPU case already suffers cost)

Summary

Towards Extreme Fast&Big Simulations

- Architecture: Hierarchical Hybrid memory
- System software: Reducing programming cost
- App. Algorithm: Reducing communication

Co-design is the key



Future Work

- More performance
  - We still suffer from several costs
    - Redundant computations
    - Costs for process oversubscription
- More scale
  - Using SSD, burst buffers
- More productivity
  - Integrating DSL (Exastencil, Physis..)
  - Integrating Polyhedral compilers