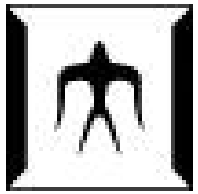
Realizing Out-of-Core Stencil Computations using Multi-Tier Memory Hierarchy on GPGPU Clusters

~ Towards Extremely Big & Fast Simulations ~

Toshio Endo

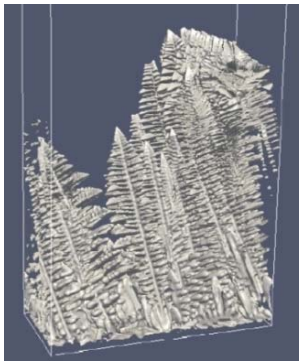
GSIC, Tokyo Institute of Technology (東京工業大学)



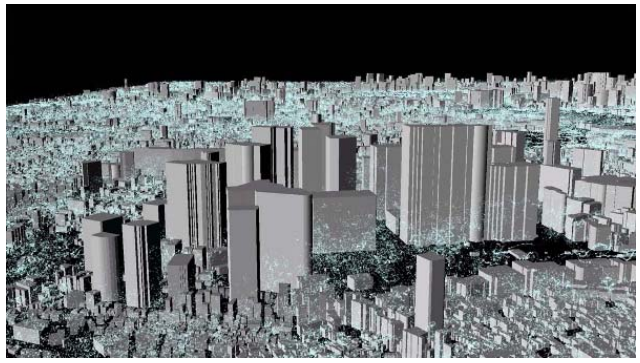
Stencil Computations

Important kernels for various simulations (CFD, material…)

ASUCA weather simulator

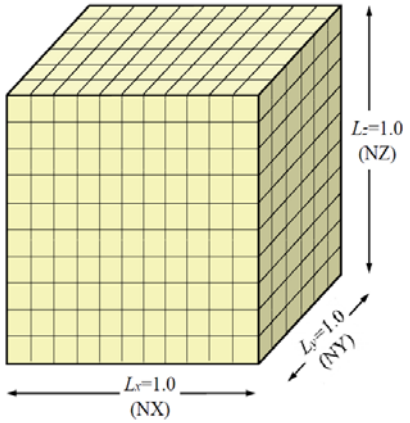


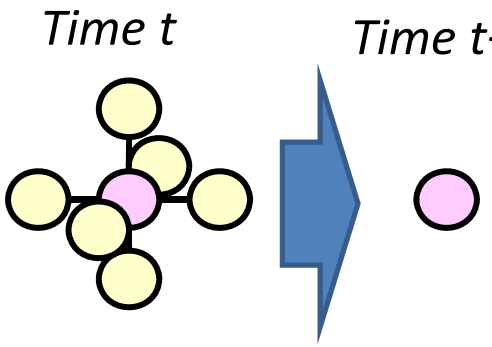
Phase-Field computation (2011 Gordon Bell)



Air flow simulation

=



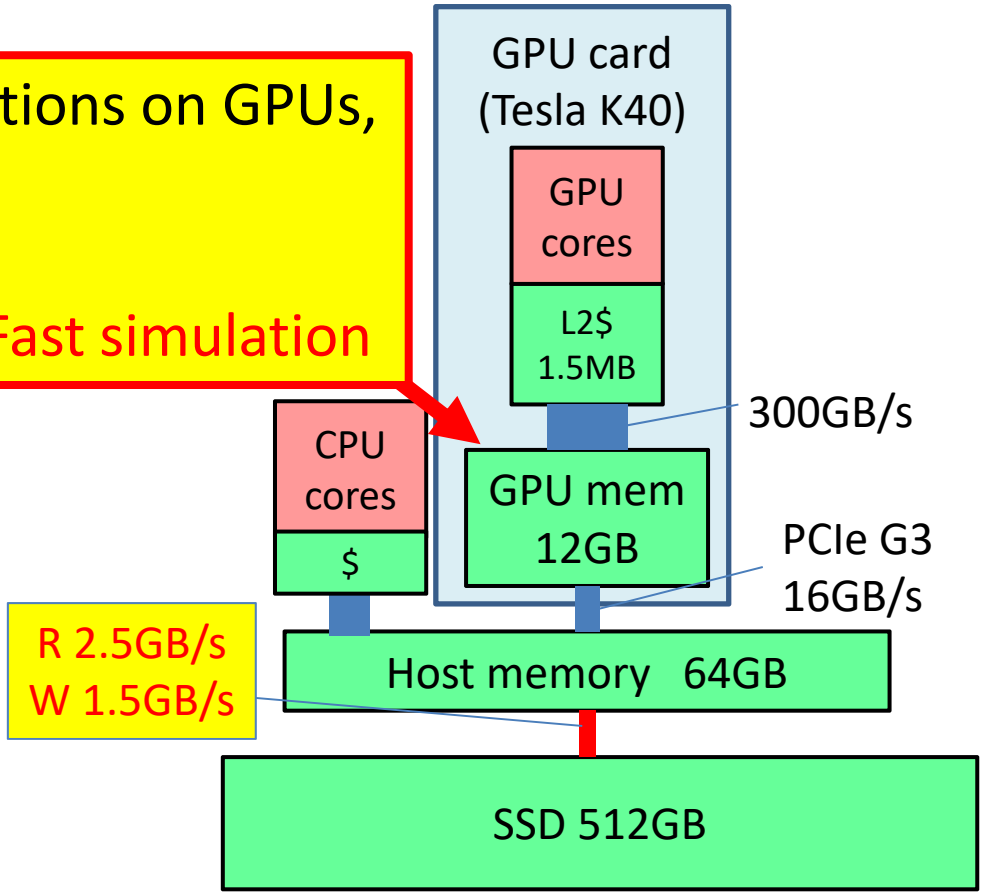


Stencil computations are "memory intensive" →

On GPU clusters,
Highly successful *in speed*
But not *in scale*

Issues on Typical Stencil Implementations on GPUs

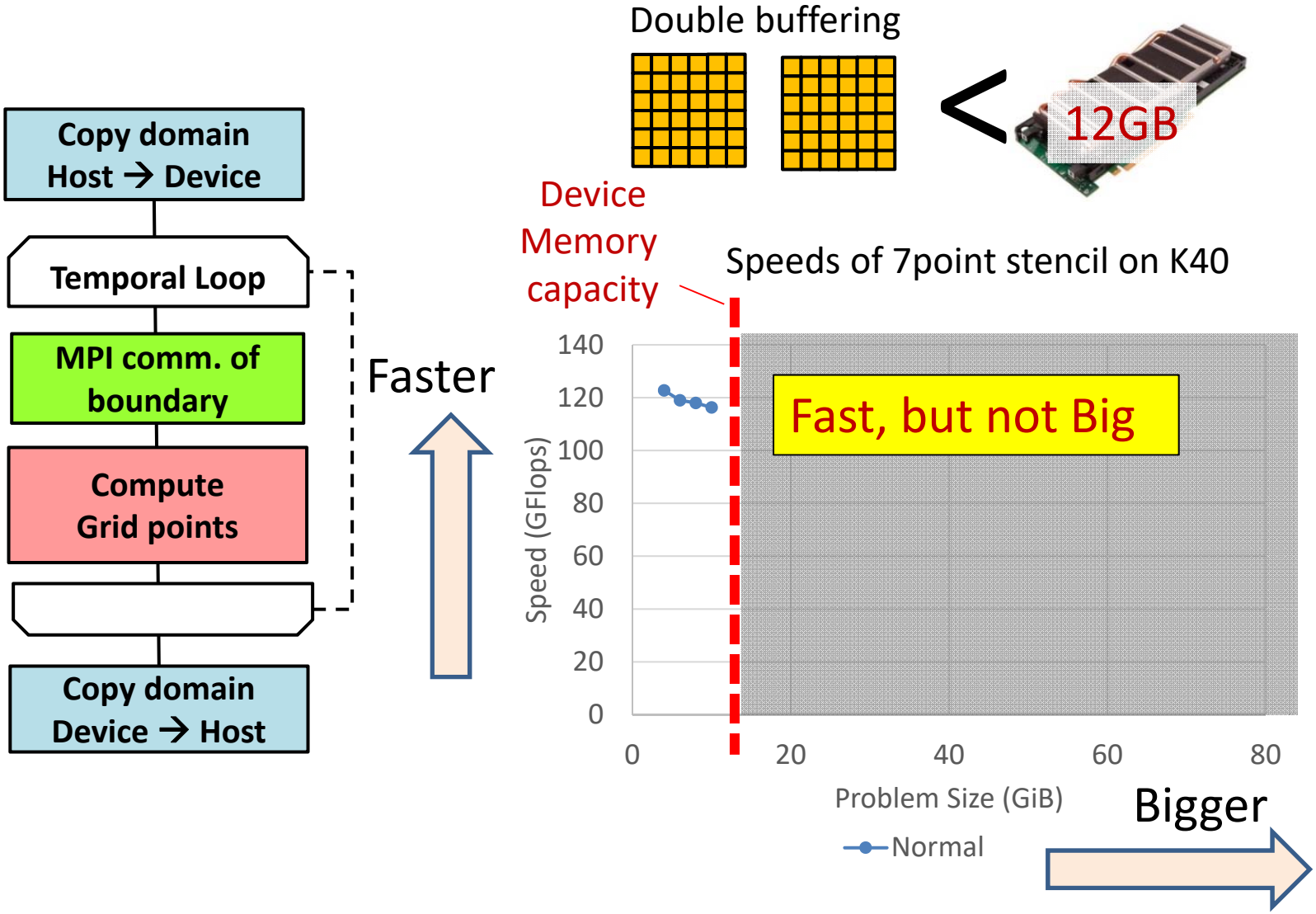
In typical stencil implementations on GPUs, array sizes are configured as
< (aggregated) GPU memory
➔ Prohibits extremely Big&Fast simulation



Using multiple GPUs is a solution

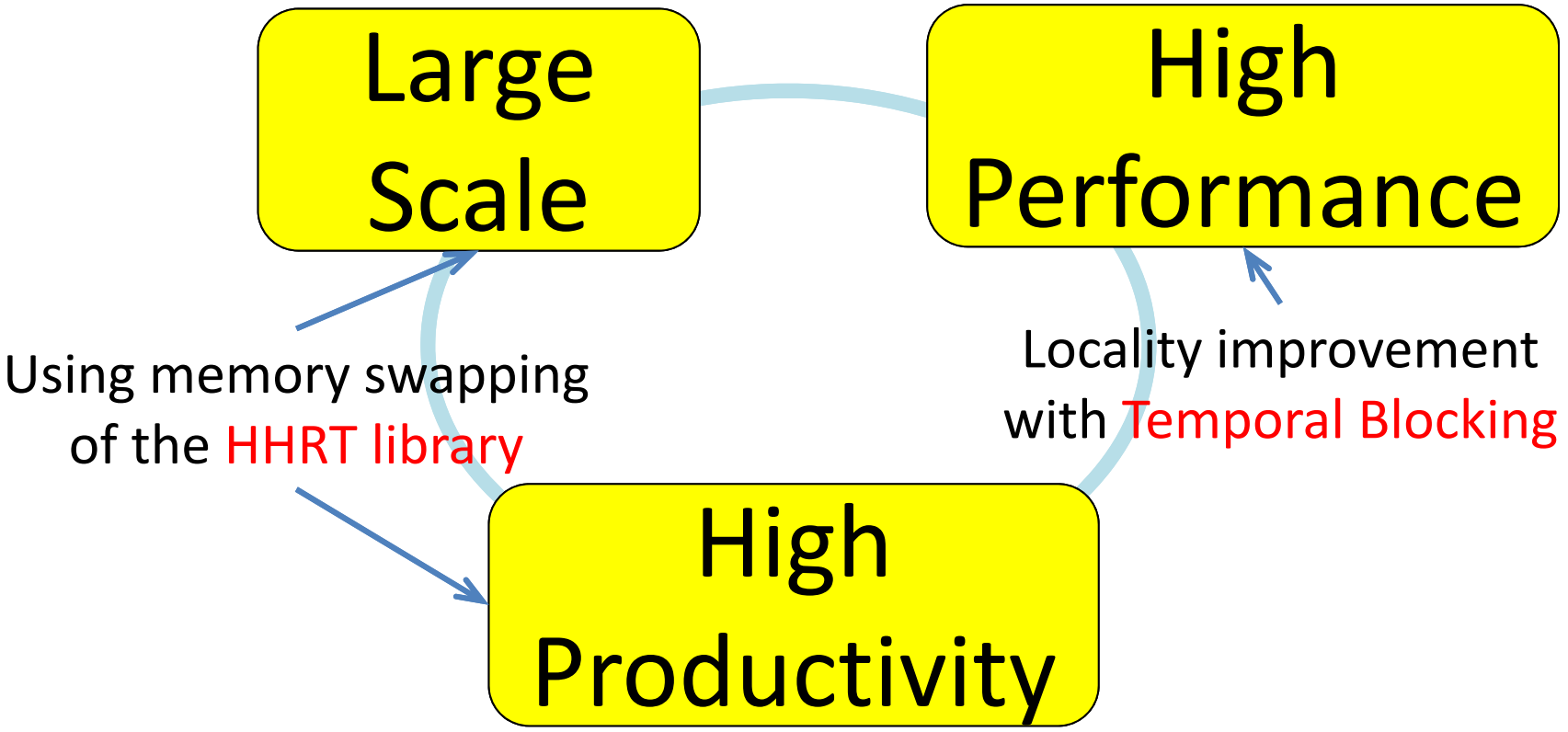
- But we are still limited by “GPU memory capacity ×#GPUs”
- Larger capacity of lower memory hierarchy is not utilized

Stencil Code Example on GPU



Goals of This Work

When we have existing apps, we want to realize followings



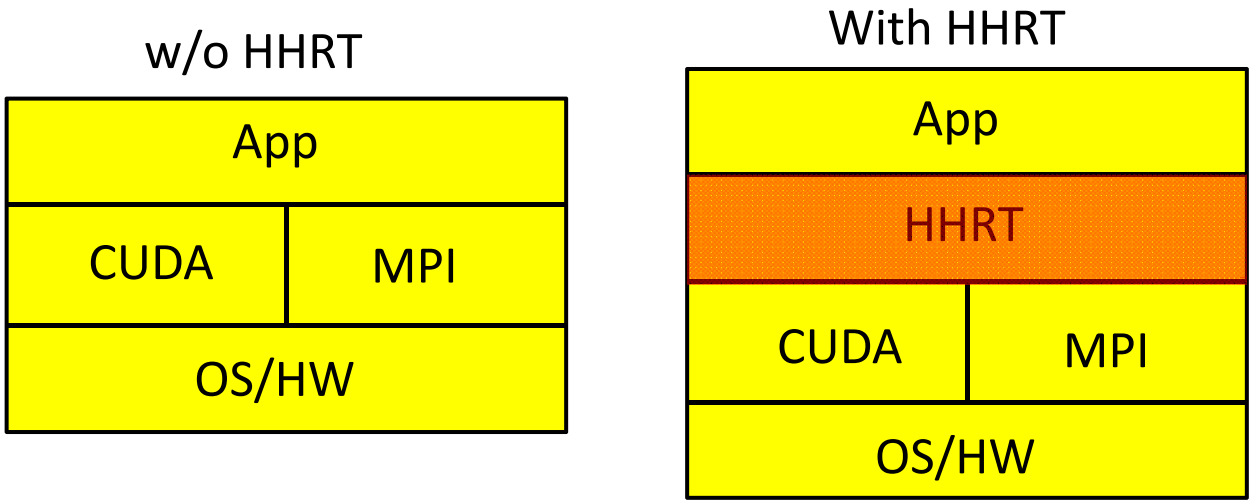
Co-design approach that spans
Algorithm layer, Runtime layer, Architecture layer

Contents

- Step 1: using HHRT library
  - Expands available memory capacity by data swapping
  - Supporting multi-tier memory hierarchy
- Step 2: using Temporal blocking (briefly)
  - Optimizations of stencils for locality improvement

The HHRT Runtime Library for GPU Memory Swapping

- 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



github.com/toshioendo/hhrt

T. Endo and Guanghao Jin. Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations. IEEE CLUSTER2014

Functions of HHRT

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

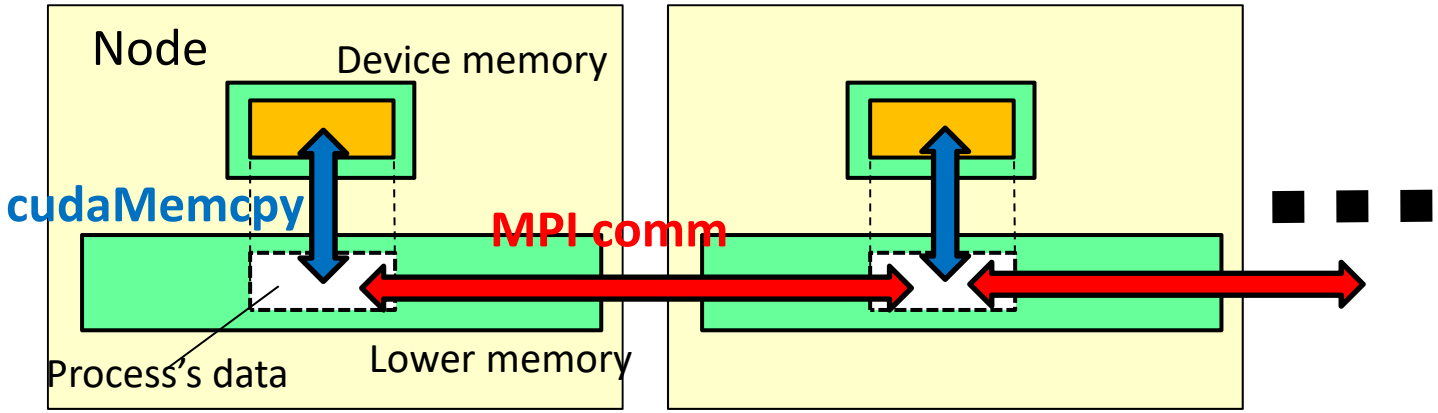
- Each GPU is shared by *m* MPI processes

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

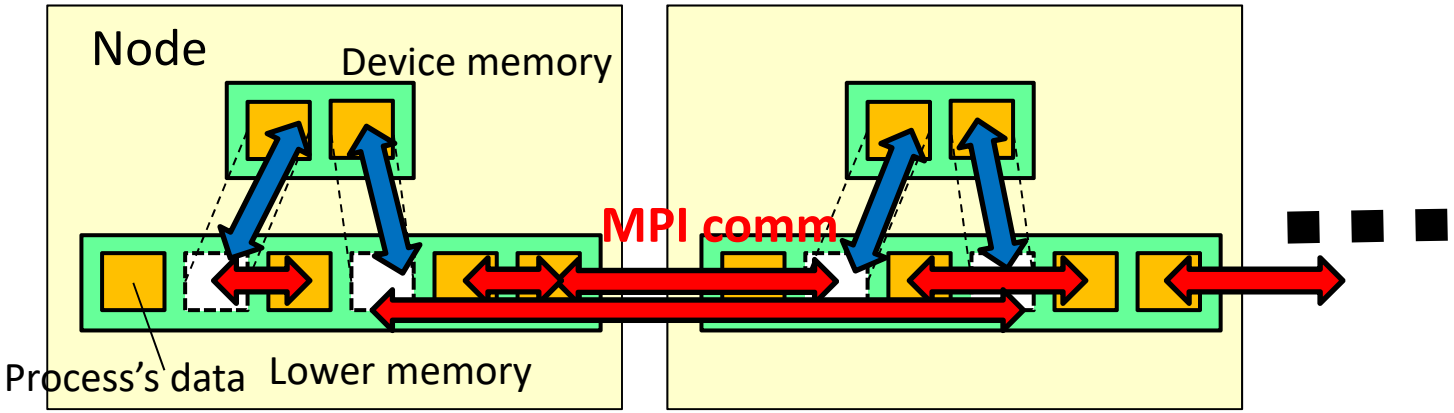
- *"process-wise"* swapping
- OS-like "page-wise" swapping is currently hard, without modifying original CUDA device/runtime

Execution model of HHRT

w/o HHRT (typically)

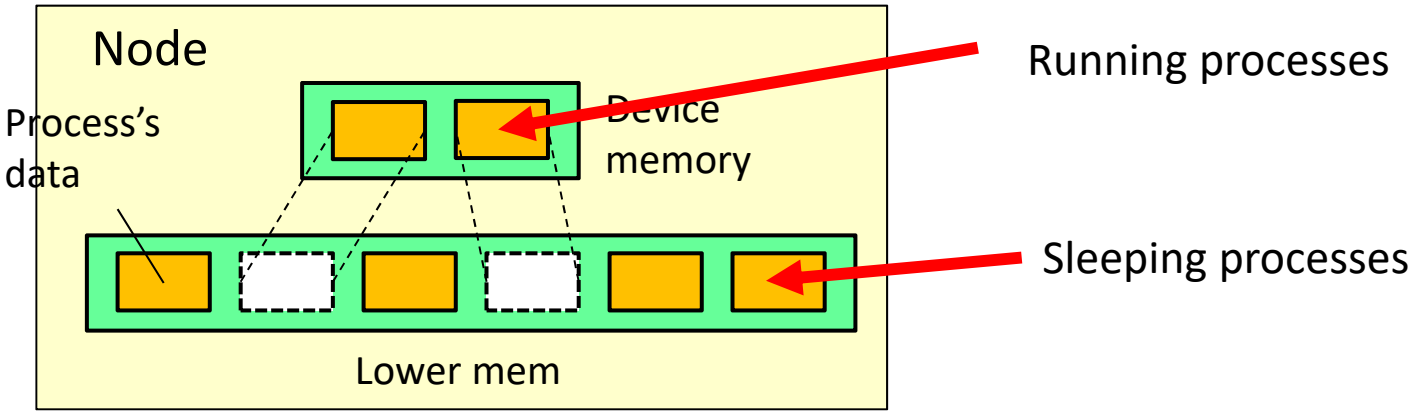


With HHRT



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

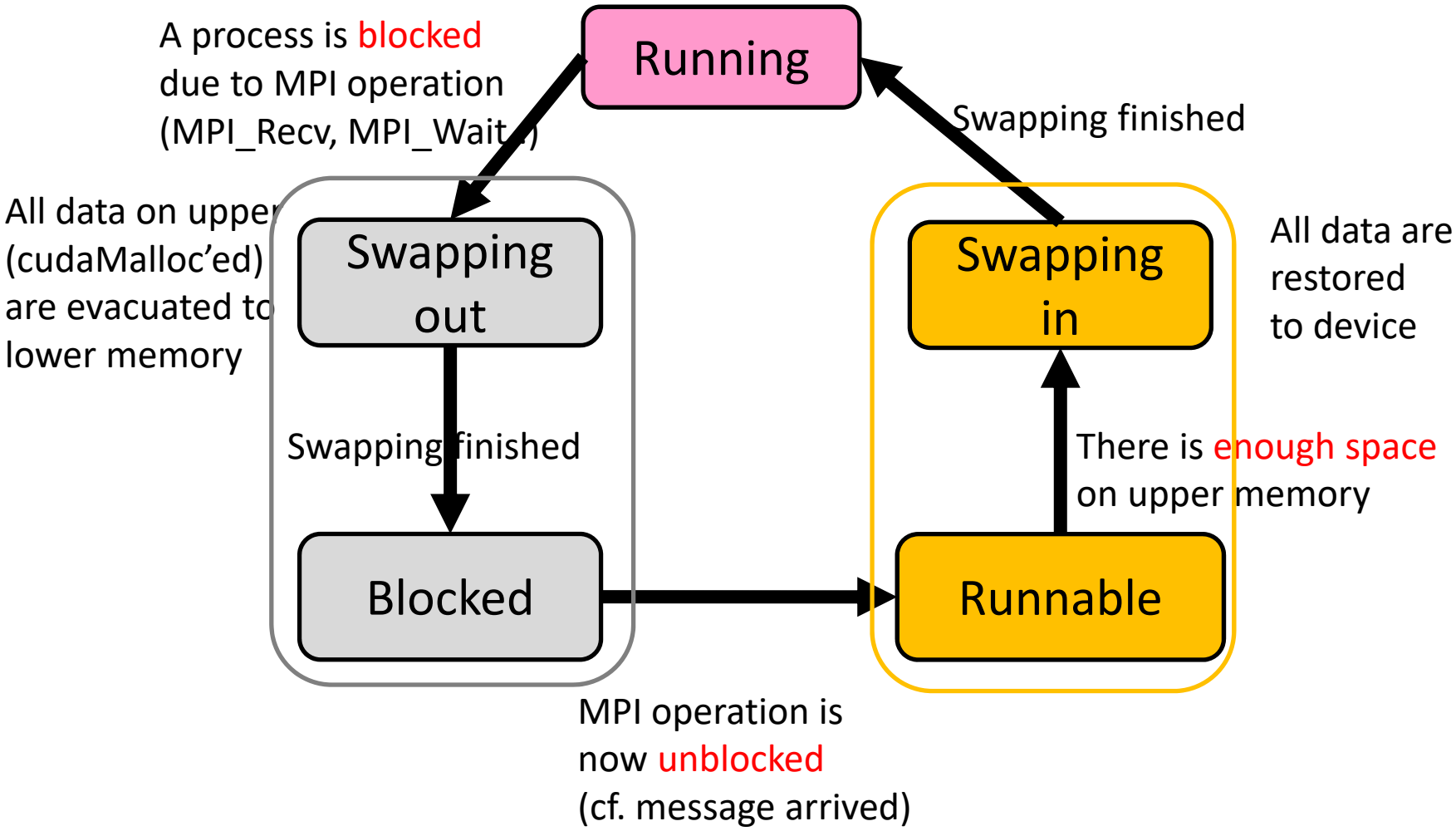
→ We can support larger data size than device memory in total

- We cannot keep all of *m* processes running

→ HHRT makes some processes "sleep" forcibly and implicitly

- Blocking MPI calls are "yield" points

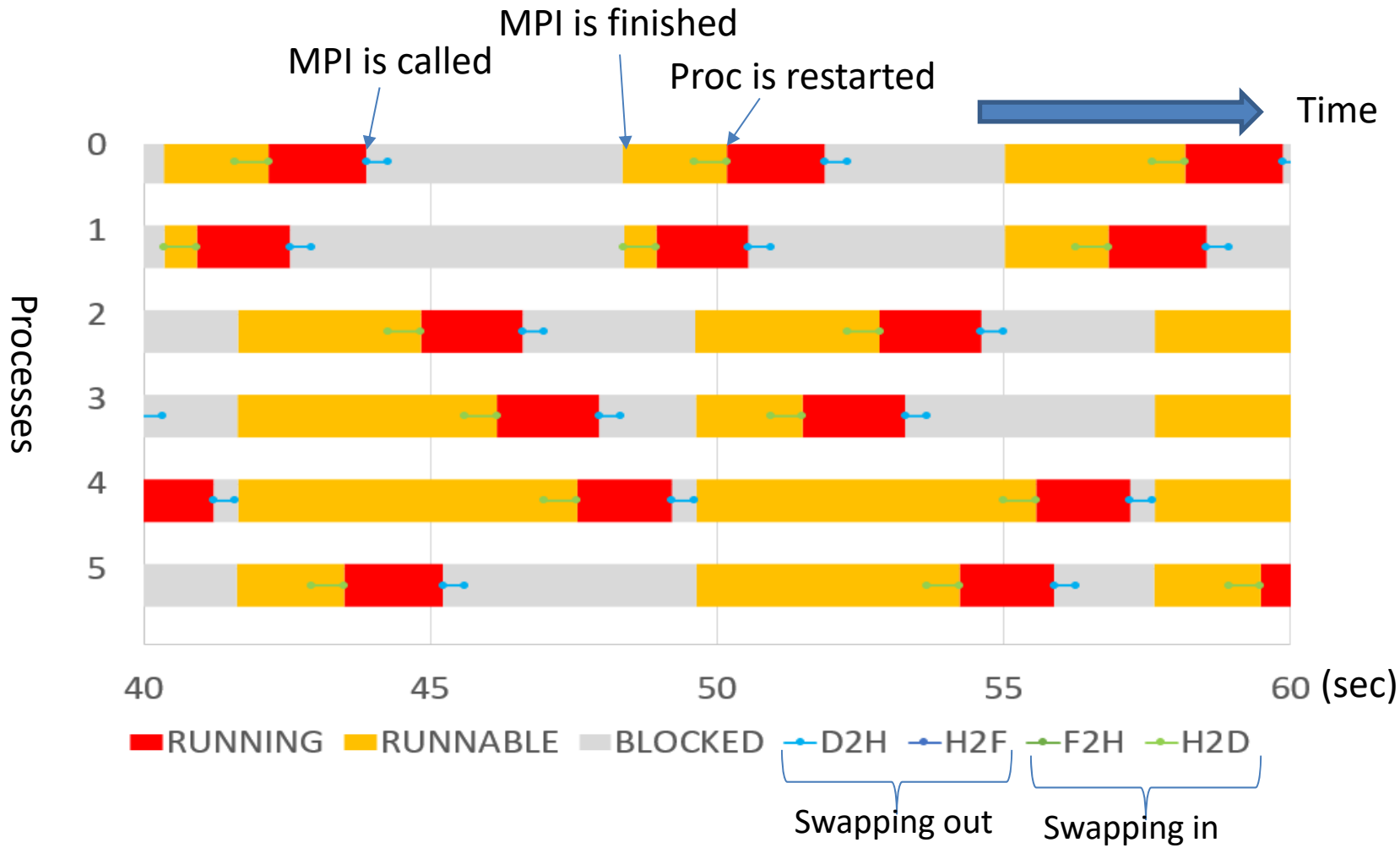
State Transition of Each Process



Executions on HHRT

6 processes are time-sharing a GPU

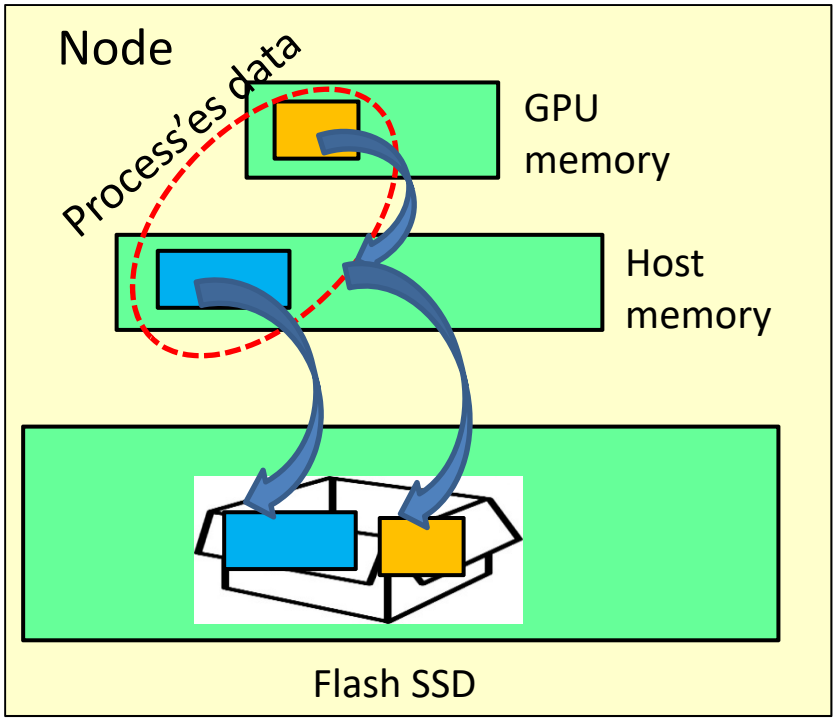
Two-tier (Device/Host) is used



What HHRT does NOT

- It does NOT automate data transfer (cudaMemcpy) → It is not OpenACC
  - Supports (traditional) CUDA programming
  - Instead, it implicitly swaps out data on device memory to lower hierarchy
- It does NOT swap in page-wise style like OS → It is NOT NVIDIA Unified Memory
  - In stencil, page-wise swapping tends to be slow
  - Instead, it adopts process-wise swapping
- It does NOT extend memory for a single process
  - Instead, our focus is to extend the aggregate capacity for multiple processes

Swapping Data in Multi-tier Memory Hierarchy



[What data are swapped]

Following data allocated by user processes

- On device memory (cudaMalloc)
- On host memory (malloc)

For this purpose, cudaMalloc, malloc… are wrapped by HHRT

Exceptionally, buffers just used for MPI communications must be remained on upper

[Where data are swapped out]

- Host memory first
- And then Flash SSD

For swapping, HHRT internally uses

- cudaMemcpy() for device ⇔ host
- read(), write() for host ⇔ Flash SSD

Evaluation Environment

	TSUBAME2.5 (K20X GPU)	TSUBAME-KFC (K80 GPU)	PC server with m.2 SSD (K40 GPU)
Device memory	6GB・250GB/s	12GB・240GB/s	12GB・288GB/s
Host memory (Speeds are via PCIe)	54GB・8GB/s	64GB・16GB/s	64GB・16GB/s
Flash SSD	120GB・R 0.2GB/s	960GB・R 1GB/s (with two SSDs)	512GB・R 2GB/s

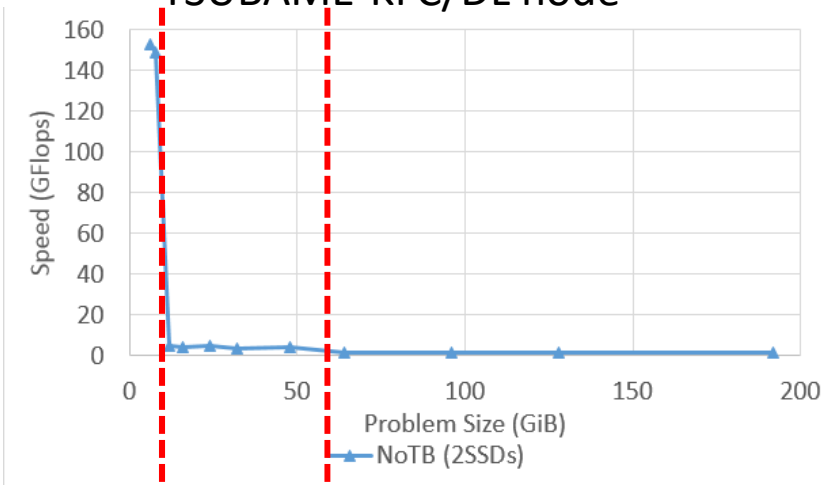
Samsung 950PRO

In our context, both of speed and capacity are insufficient
(SSDs installed in 2010)

Result of Step 1:
Exceeding Memory Capacity Wall

7点ステンシル、計算には1GPUを利用

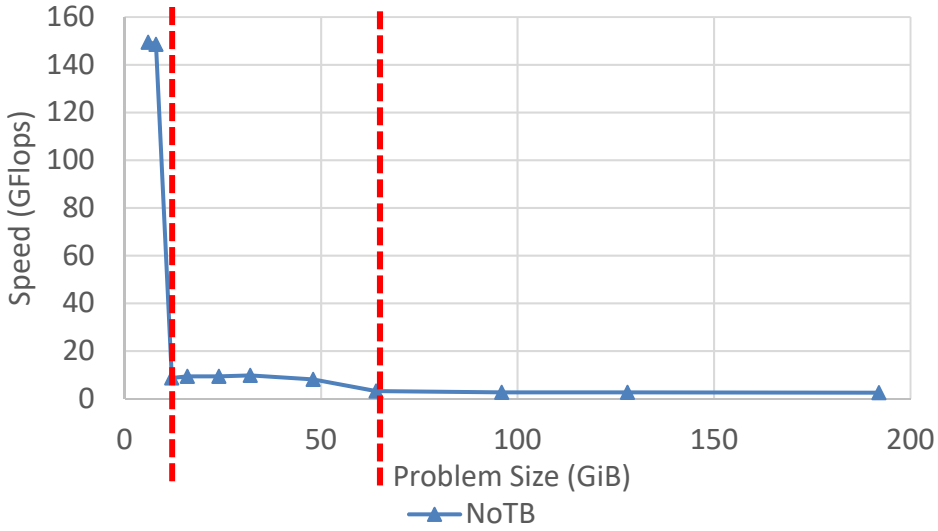
TSUBAME-KFC/DL node



Device memory

Host memory

m.2 搭載PC

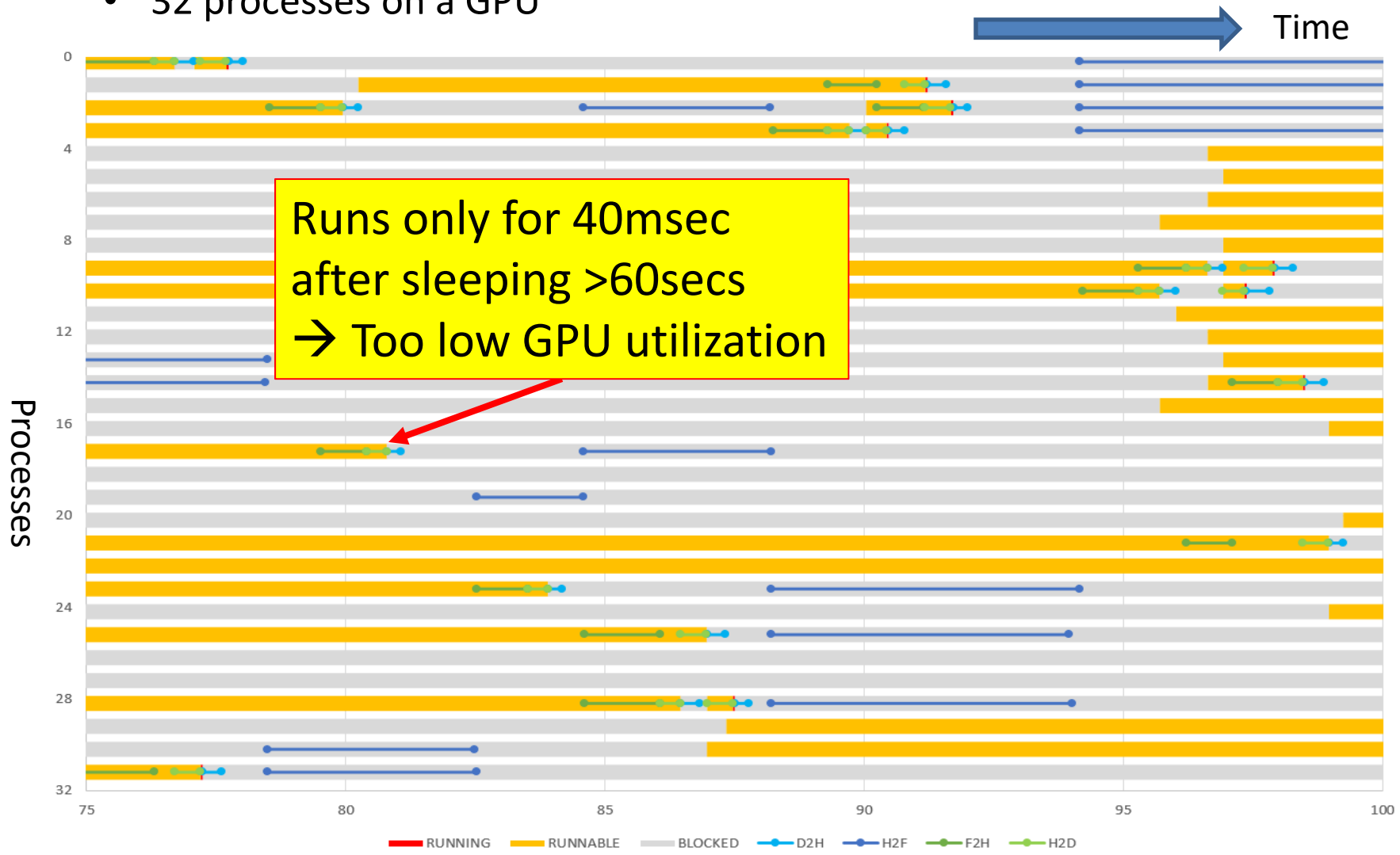


- Certainly we exceed capacity wall for scale, however, the performance is seriously bad!

Issues in Step1: Too low GPU utilization

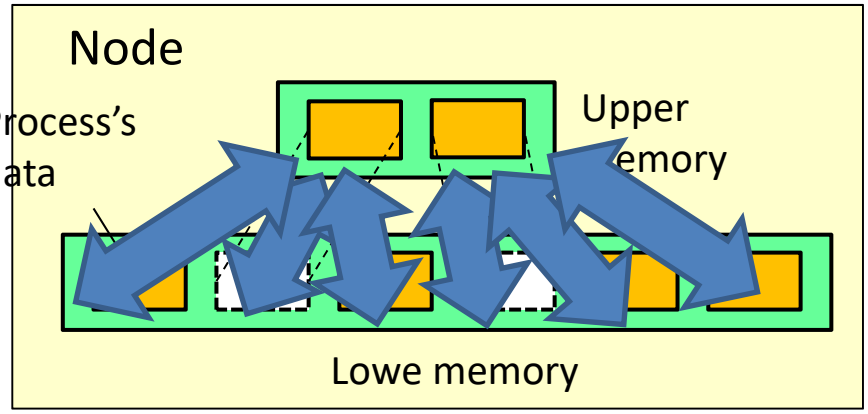
In the case of 96GB problem

- 32 processes on a GPU



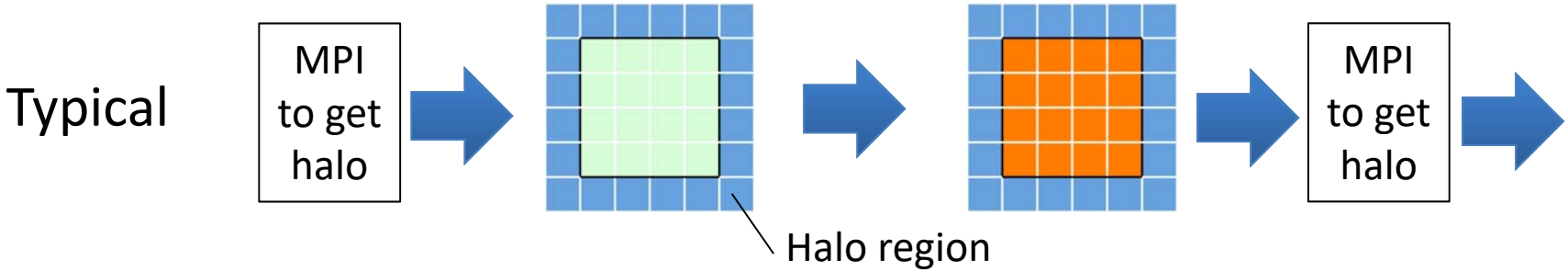
Why is GPU Utilization Too Low?

- Each process can suffer from heavy memory swapping costs every iteration
  - It incurs transfer of the entire process'es sub-domain between memory hierarchy
- This is done automatically, but too heavy to hide



- This is due to lack of locality of stencil computations
  - Array data are swapped out every iteration
- We need optimizations to improve locality as step 2!!

Step 2:
Temporal Blocking (TB) for Locality Improvement

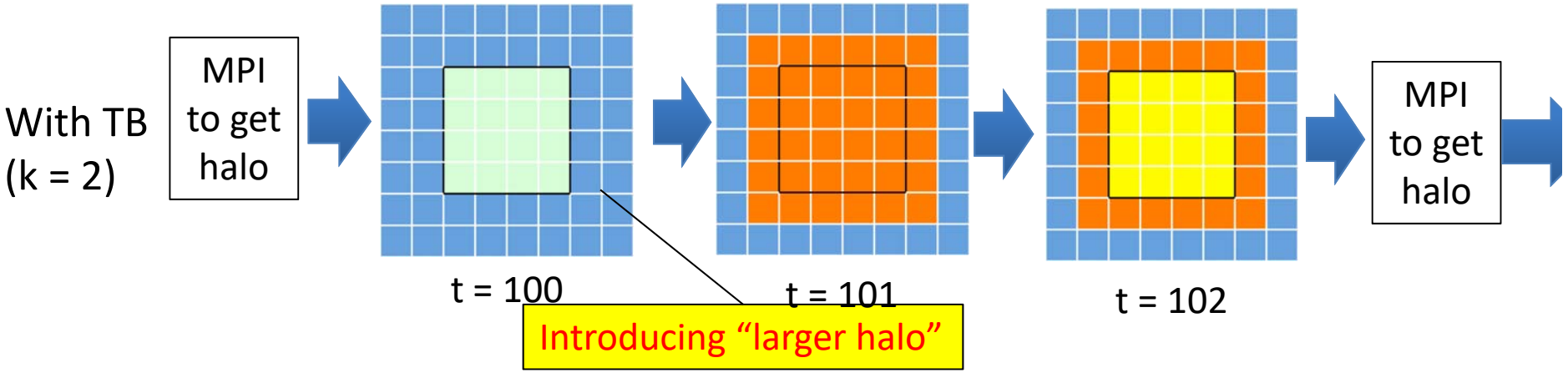


Temporal blocking (in our context):

Larger halo region, with width of k, is introduced per process

After a process receives halo with MPI, we do **k-step update at once** without MPI

k is "temporal block size"



Frequency of MPI comm (yielding points on HHRT) is reduced to 1/k

Appropriate Temporal Block Sizes (*k*)

- If *k* is too small, we suffer from swapping costs (if swap occurs)
- If *k* is too large, we suffer from redundant computation costs for larger halo

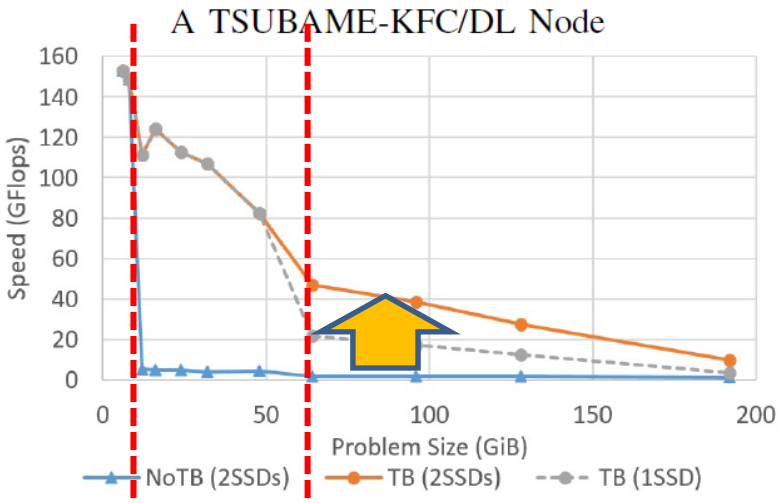
PC server with an m.2 SSD

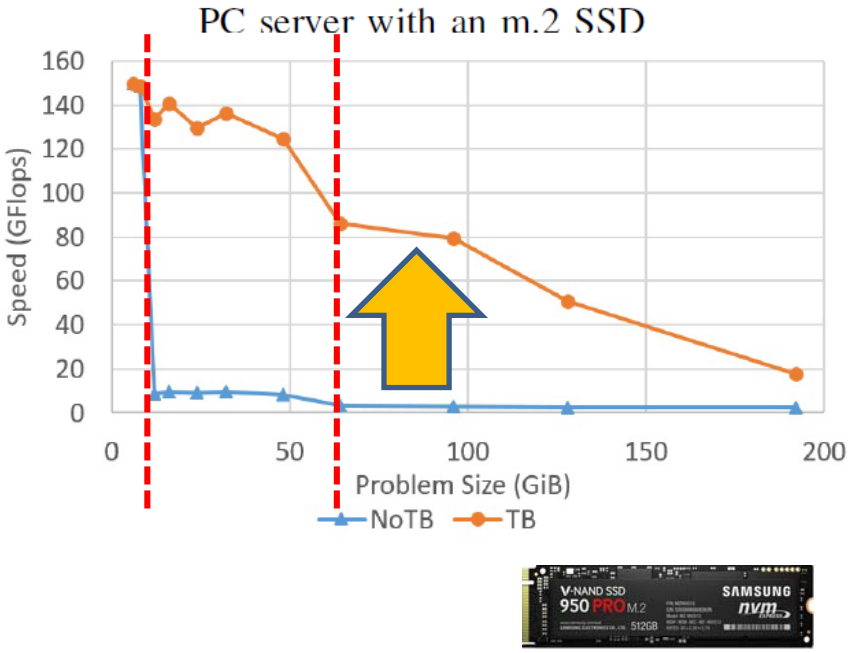
Problem Sizes	k=1	8	16	24	32	48	64	96
6(GiB)	149	148	145	142	137	134	129	119
8	149	147	145	142	139	133	129	121
12	8.72	65.7	101	130	134	132	126	114
16	9.39	63.2	108	138	141	135	130	106
24	9.37	63.3	98.8	122	125	130	122	110
32	9.79	58.3	89.5	121	136	127	121	98 .3
48	8.12	61.7	88.7	116	125	*72.7*	*87.9*	*91.5*
64	*3.23*	*22.3*	*34.4*	*49.0*	*57.7*	*85.9*	*82.6*	*75.5*
96	*2.68*	*20.7*	*33.4*	*47.7*	*53.4*	*79.3*	OOM	OOM
128	*2.67*	*18.8*	*38.4*	*45.4*	*50.6*	OOM	OOM	OOM
192	*2.55*	*17.7*	OOM	OOM	OOM	OOM	OOM	OOM

Device Memory capacity

Host Memory capacity

Results of Step 2: Performance Improvement





- With high-speed with ~2GB/s Read, we obtain ~55% performance with 1.5x larger problem than host memory
  - We observe performance difference of SSDs
  - We still see significant slow down with > 100GB sizes

Current Limitations on Performance and Discussion

PC server with an m.2 SSD

Problem Sizes	k=1	8	16	24	32	48	64	96
6(GiB)	149	148	145	142	137	134	129	119
8	149	147	145	142	139	133	129	121
12	8.72	65.7	101	130	134	132	126	114
16	9.39	63.2	108	138	141	135	130	106
24	9.37	63.3	98.8	122	125	130	122	110
32	9.79	58.3	89.5	121	136	127	121	98 .3
48	8.12	61.7	88.7	116	125	*72.7*	*87.9*	*91.5*
64	*3.23*	*22.3*	*34.4*	*49.0*	*57.7*	*85.9*	*82.6*	*75.5*
96	*2.68*	*20.7*	*33.4*	*47.7*	*53.4*	*79.3*	OOM	OOM
128	*2.67*	*18.8*	*38.4*	*45.4*	*50.6*	OOM	OOM	OOM
192	*2.55*	*17.7*	OOM	OOM	OOM	OOM	OOM	OOM

Device memory

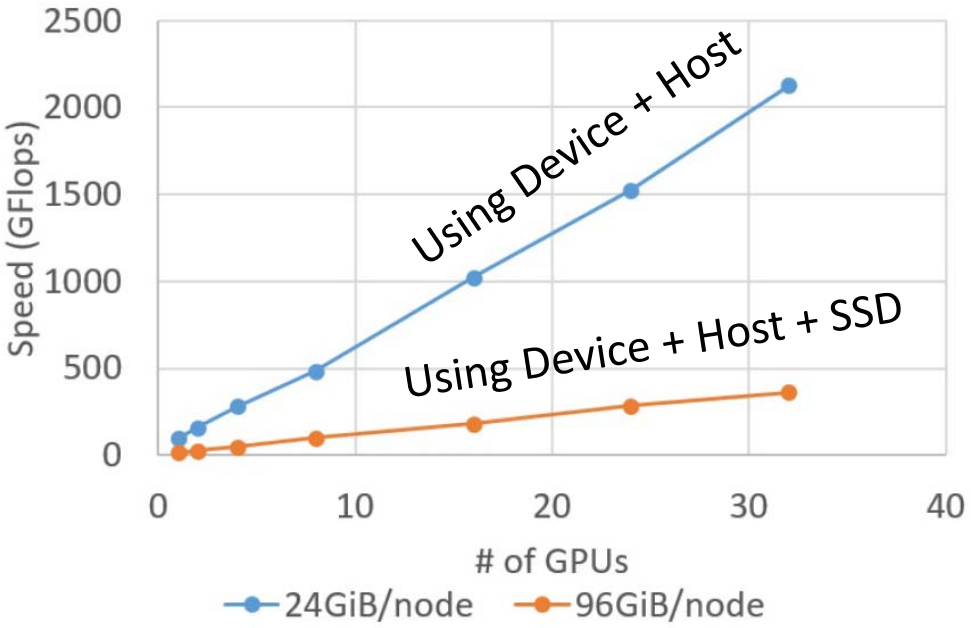
Host memory

Execution failure due to out-of-memory limits us. Why?

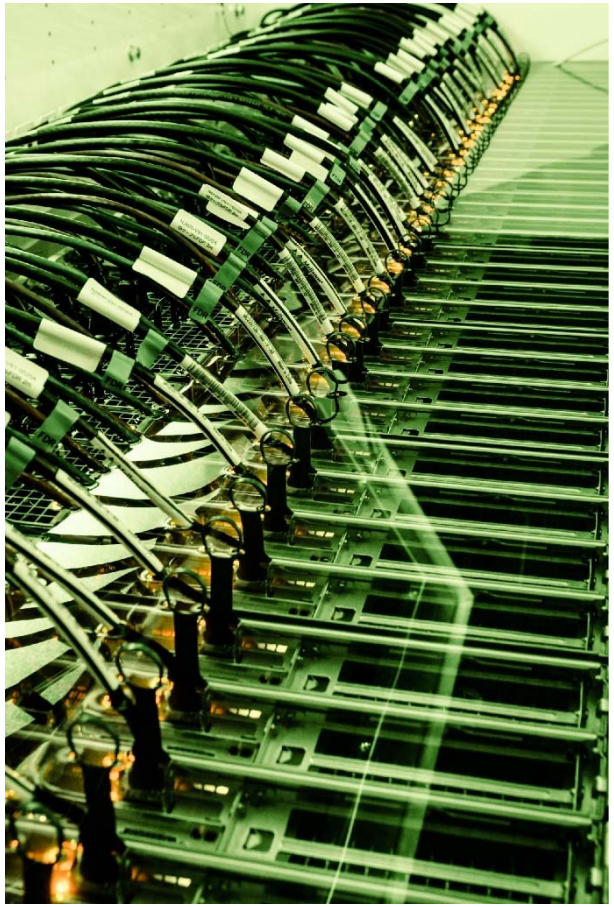
- Even with swapping facility, there is still memory pressure for:
  - MPI communication buffers
    - Both on user space and on MPI internally
  - CUDA's internal device memory consumption
    - ~75MB (per proc) × 80 proc= 6GB ➔ ~50% of GPU memory!!

Weak Scalability on Multi GPU/Node

The TSUBAME-KFC Cluster
(1 K80 GPU + 2 SSDs) per node are used



Fairly good weak scalability,
But costs of SSDs are still heavy



Future Work

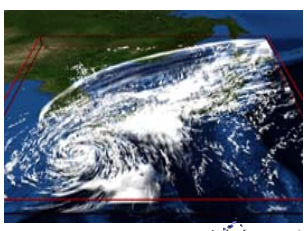
- More performance
  - We still suffer from memory pressure
    - Dozens of processes share MPI/CUDA
    - Scalable MPI/CUDA multiplexor will be the key
- More scale
  - Using burst buffers?
- More productivity
  - Integrating DSL (Exastencil, Physis..)
  - Integrating Polyhedral compilers

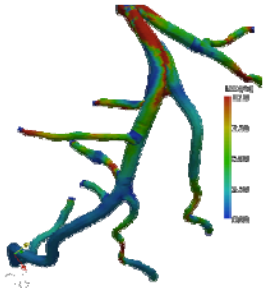
Summary

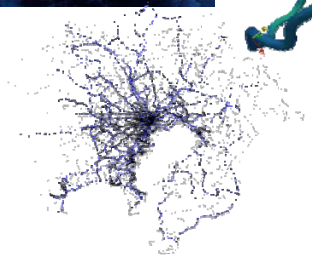
Out-of-core stencil computations on 3-tier memory hierarchy has been described

- Architecture level:
  - High performance (>GB/s) Flash SSDs
- Middleware level:
  - HHRT library for data swapping
- App. Algorithm level:
  - Temporal blocking for locality improvement

Co-design is the key







System Software For Mem Hierarchy