

# In-Situ Assessment of Device-Side Compute Work for Dynamic Load Balancing in a GPU-Accelerated PIC Code

Michael Rowan

Work with Kevin Gott, Axel Huebl, Jack Deslippe

See preprint here: <https://arxiv.org/abs/2104.11385>

PASC '21 — 07.05.2021



# Outline:

---

1. Load balancing intro
2. Dynamic load balancing in  
PIC code run on GPUs

# GPU-accelerated machines entered the TOP500 rankings just over a decade ago.

Nov. 2008

Rank	System	Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power [kW]
1	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband, IBM DOE/NNSA/LANL United States	129,600	1,105.0	1,456.7	2,483
2	Jaguar - Cray XT5 QC 2.3 GHz, Cray/HPE DOE/SC/Oak Ridge National Laboratory United States	150,152	1,059.0	1,381.4	6,950
3	Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz, HPE NASA/Ames Research Center/NAS United States	51,200	487.0	608.8	2,090
4	BlueGene/L - eServer Blue Gene Solution, IBM DOE/NNSA/LLNL United States	212,992	478.2	596.4	2,329
5	Kraken XT5 - Cray XT5 QC 2.3 GHz, Cray/HPE National Institute for Computational Sciences/University of Tennessee United States	66,000	463.3	607.2	
6	Intrepid - Blue Gene/P Solution, IBM DOE/SC/Argonne National Laboratory United States	163,840	450.3	557.1	1,260
7	Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband, Oracle Texas Advanced Computing Center/Univ. of Texas United States	62,976	433.2	579.4	2,000
8	Franklin - Cray XT4 QuadCore 2.3 GHz, Cray/HPE DOE/SC/LBNL/NERSC United States	38,642	266.3	355.5	1,150
9	Jaguar - Cray XT4 QuadCore 2.1 GHz, Cray/HPE DOE/SC/Oak Ridge National Laboratory United States	30,976	205.0	260.2	1,580
10	Red Storm - Sandia/Cray Red Storm, XT3/4, 2.4/2.2 GHz dual/quad core, Cray/HPE NNSA/Sandia National Laboratories United States	38,208	204.2	284.0	2,506

Nov. 2020

Rank	System	Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power [kW]
1	Superccomputer Fugaku - Superccomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	JUWELS Booster Module - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.80Hz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	HPCS - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	Frontiera - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
10	Dammam-7 - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 5XM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	



# GPU-accelerated machines entered the TOP500 rankings just over a decade ago.

Nov. 2008

Rank	System	Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power [kW]
1	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband, IBM DOE/NNSA/LANL United States	129,600	1,105.0	1,456.7	2,483
2	Jaguar - Cray XT5 QC 2.3 GHz, Cray/HPE DOE/SC/Oak Ridge National Laboratory United States	150,152	1,059.0	1,381.4	6,950
3	Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz, HPE NASA/Ames Research Center/NAS United States	51,200	487.0	608.8	2,090
4	BlueGene/L - eServer Blue Gene Solution, IBM DOE/NNSA/LLNL United States	212,992	478.2	596.4	2,329
5	Kraken XT5 - Cray XT5 QC 2.3 GHz, Cray/HPE National Institute for Computational Sciences/University of Tennessee United States	66,000	463.3	607.2	
6	Intrepid - Blue Gene/P Solution, IBM DOE/SC/Argonne National Laboratory United States	163,840	450.3	557.1	1,260
7	Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband, Oracle Texas Advanced Computing Center/Univ. of Texas United States	62,976	433.2	579.4	2,000
8	Franklin - Cray XT4 QuadCore 2.3 GHz, Cray/HPE DOE/SC/LBNL/NERSC United States	38,642	266.3	355.5	1,150
9	Jaguar - Cray XT4 QuadCore 2.1 GHz, Cray/HPE DOE/SC/Oak Ridge National Laboratory United States	30,976	205.0	260.2	1,580
10	Red Storm - Sandia/Cray Red Storm, XT3/4, 2.4/2.2 GHz dual/quad core, Cray/HPE NNSA/Sandia National Laboratories United States	38,208	204.2	284.0	2,506

Nov. 2020

Rank	System	Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power [kW]
1	Superccomputer Fugaku - Superccomputer Fugaku, A64FX 48C 2.2GHz, Tohoku Interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	JUWELS Booster Module - Bull Sequana XH2000, AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	HPCS - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	Frontiera - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
10	Dammam-7 - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 5XM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	



# How do we get optimal performance from these supercomputers?

---

- Compilers
- Algorithms/data structures
- Load balancing

# How do we get optimal performance from these supercomputers?

---

- Compilers
- Algorithms/data structures
- Load balancing

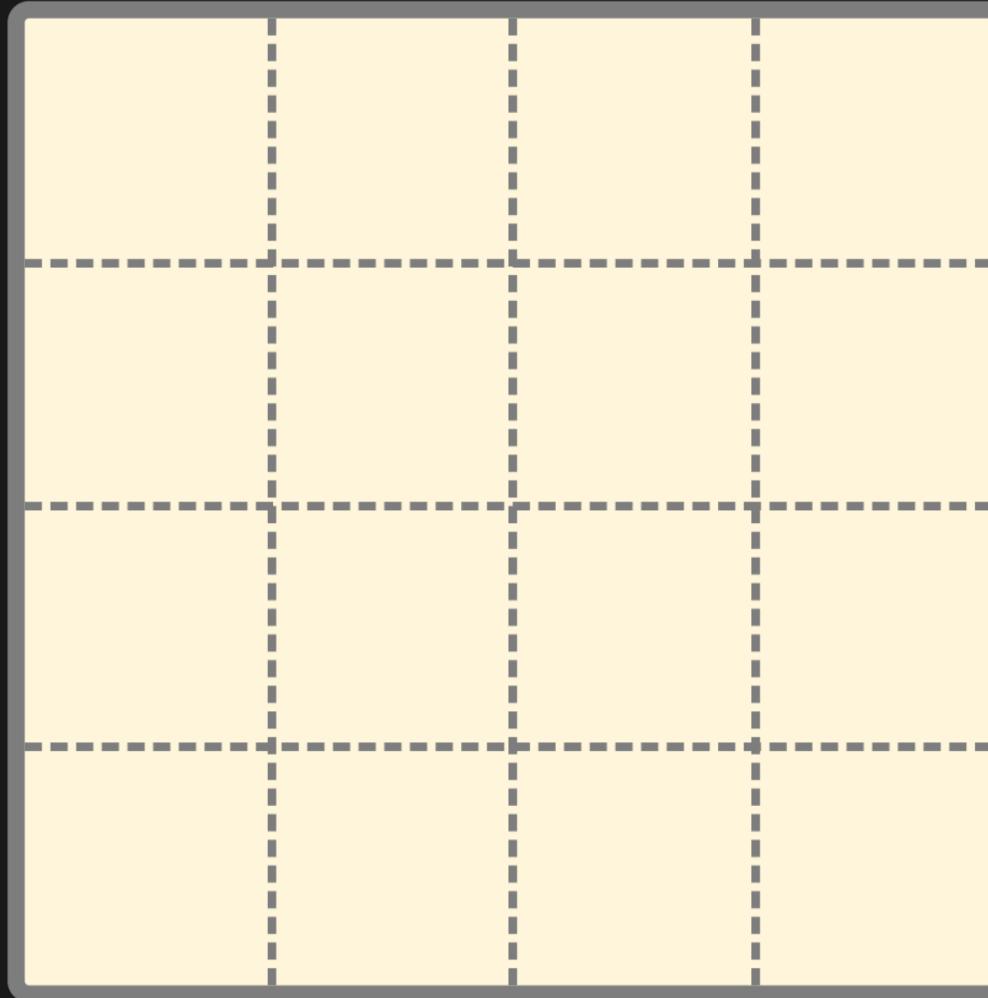
Particle-mesh codes parallelize via *domain decomposition*.

---



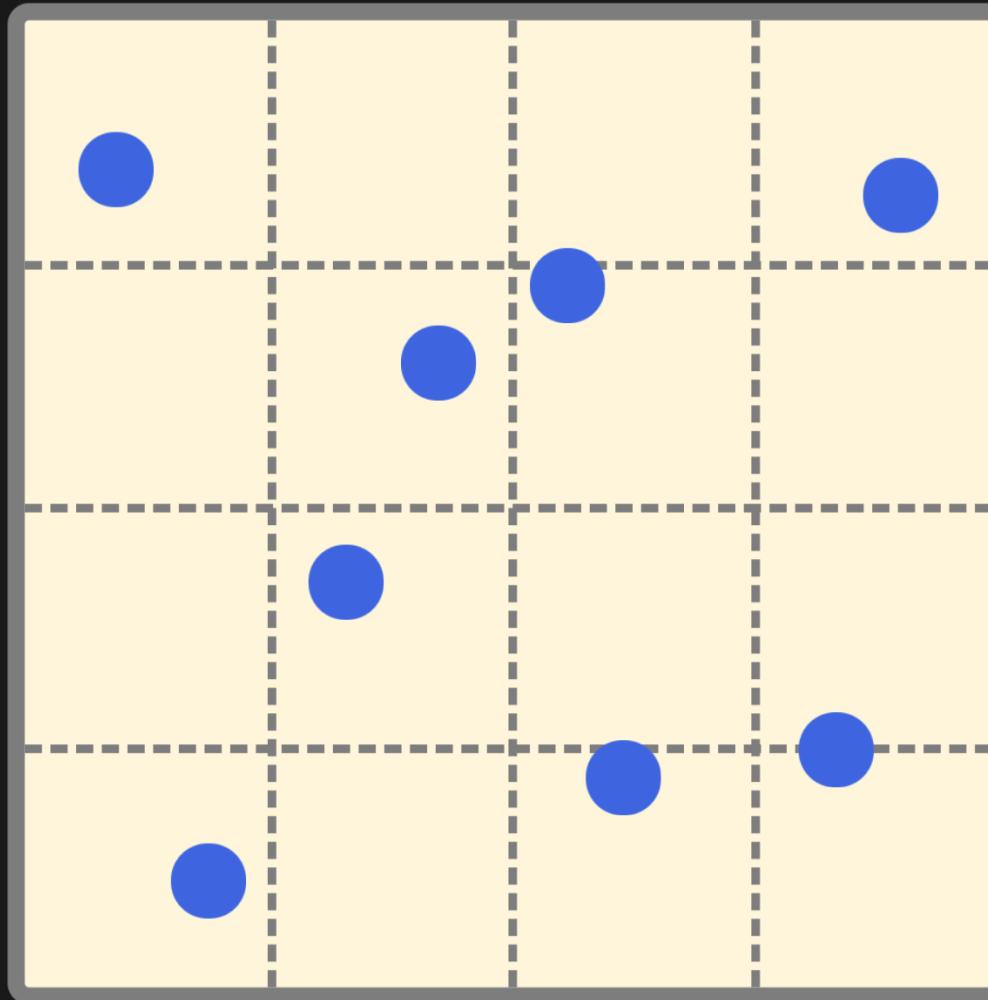
Particle-mesh codes parallelize via *domain decomposition*.

---



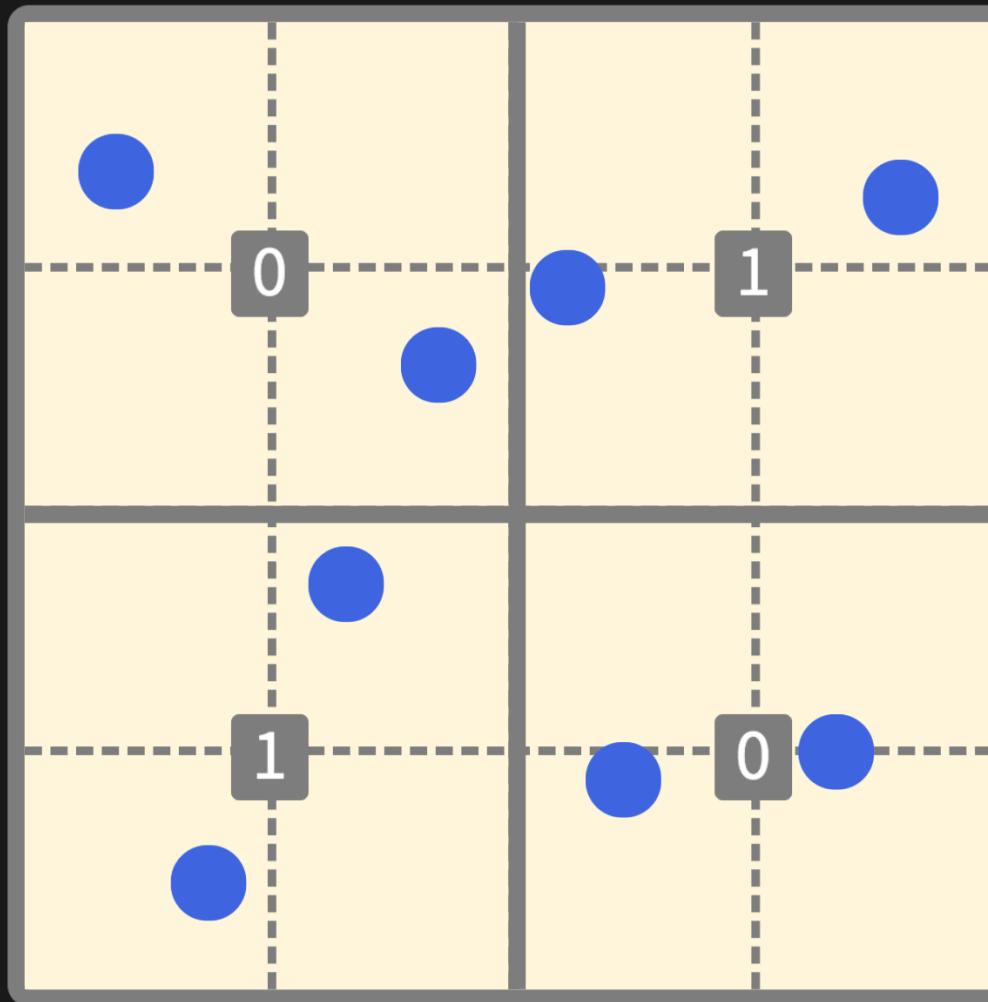
Particle-mesh codes parallelize via *domain decomposition*.

---

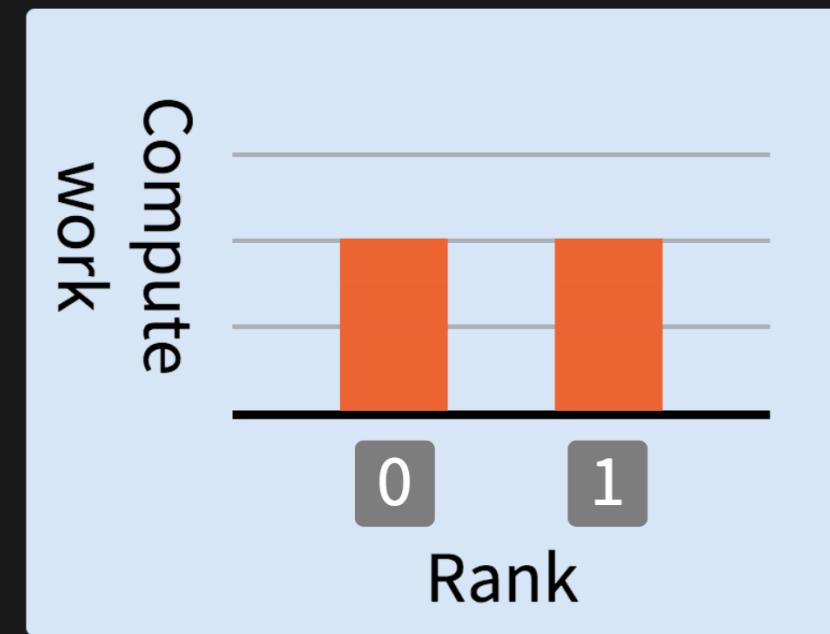
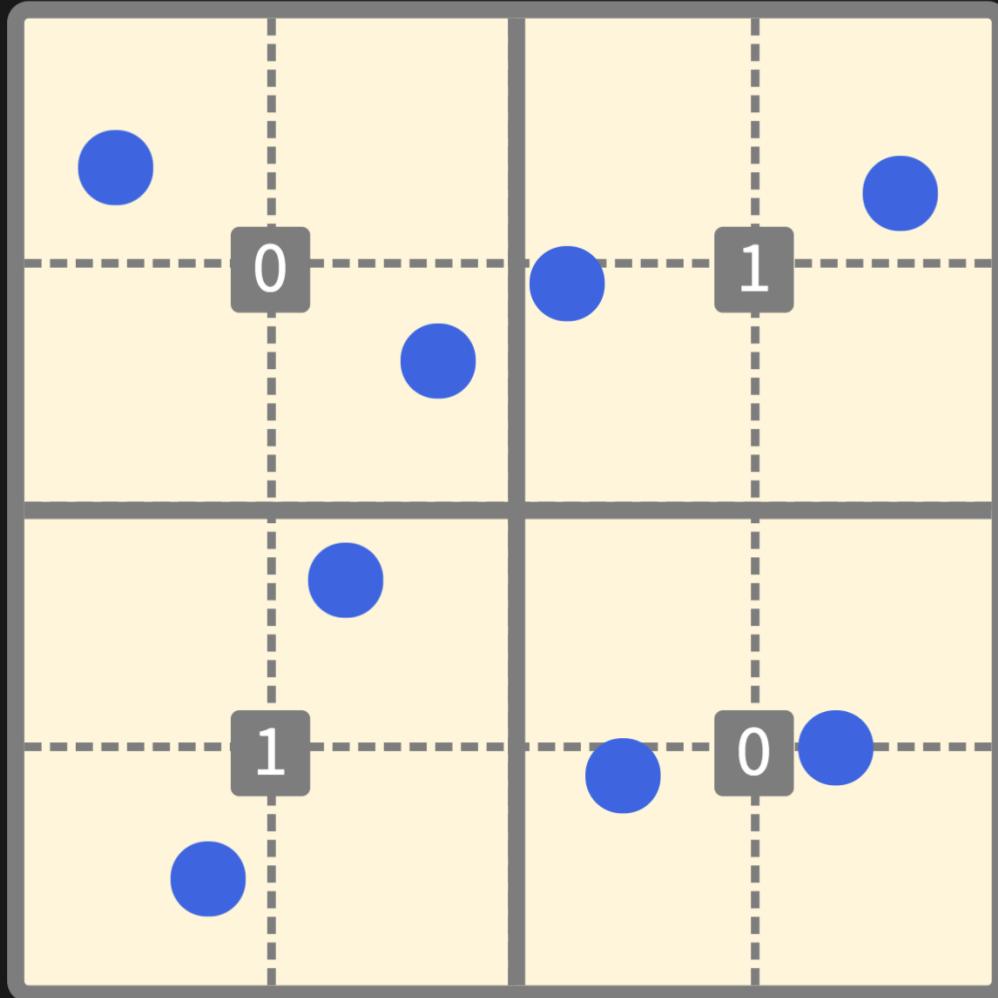


# Particle-mesh codes parallelize via *domain decomposition*.

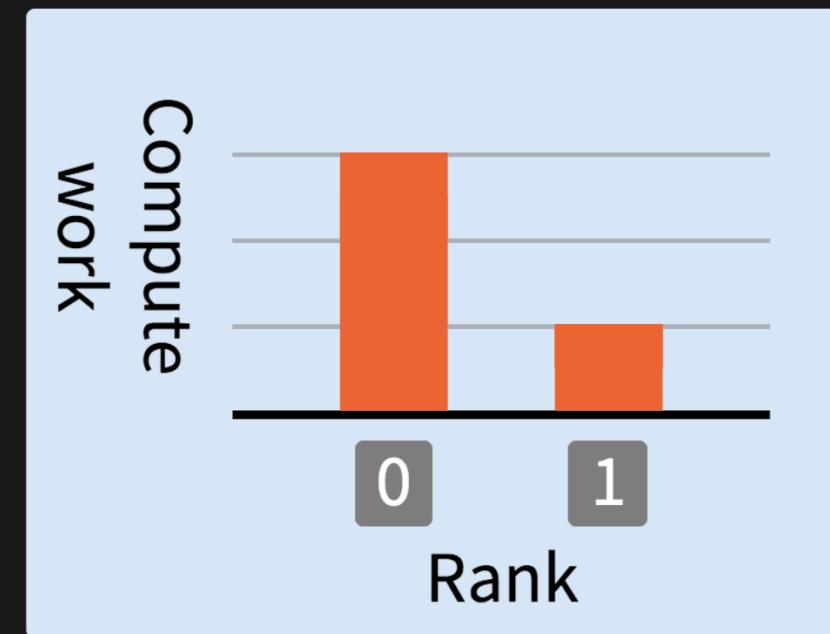
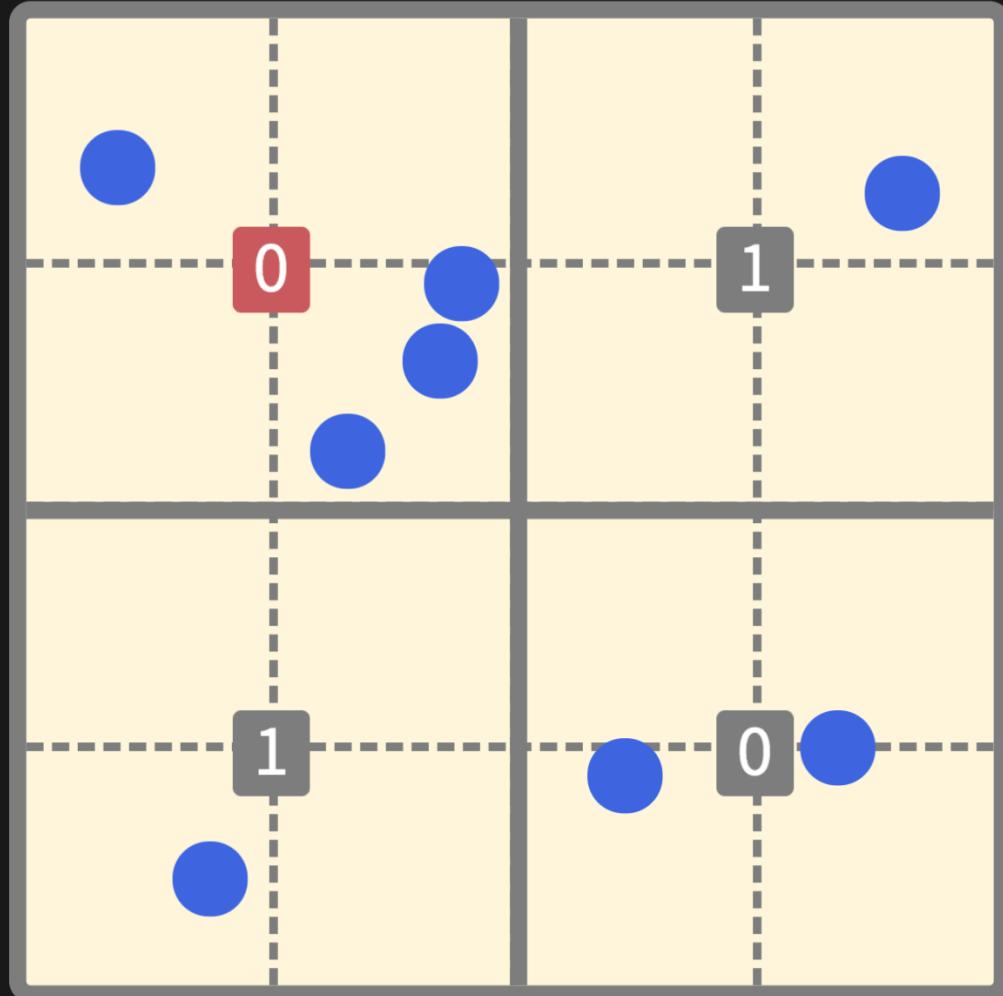
---



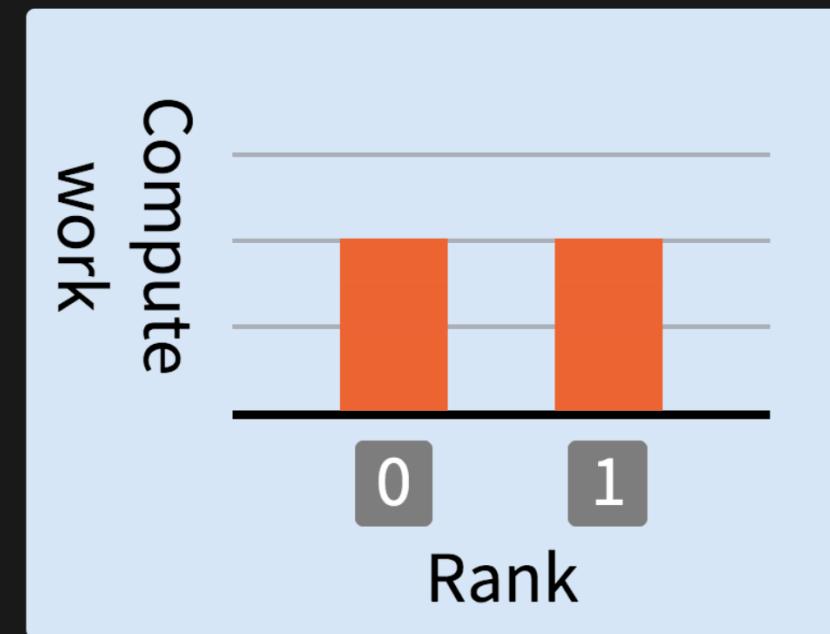
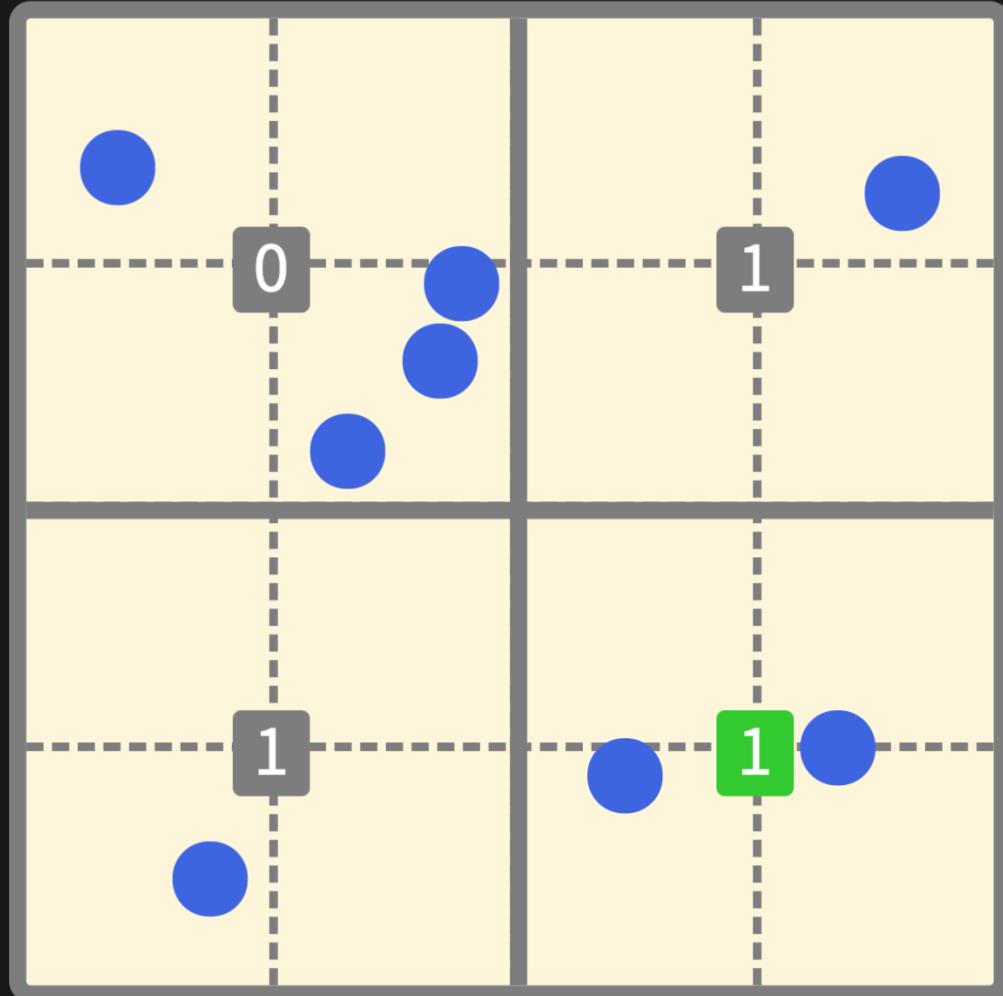
# Particle-mesh codes parallelize via *domain decomposition*.



# Particle-mesh simulations can suffer from load imbalance.



# Particle-mesh simulations can suffer from load imbalance.



# Load imbalance can be corrected at run time.

---

Basic load balance algorithm for distributed memory particle-mesh:

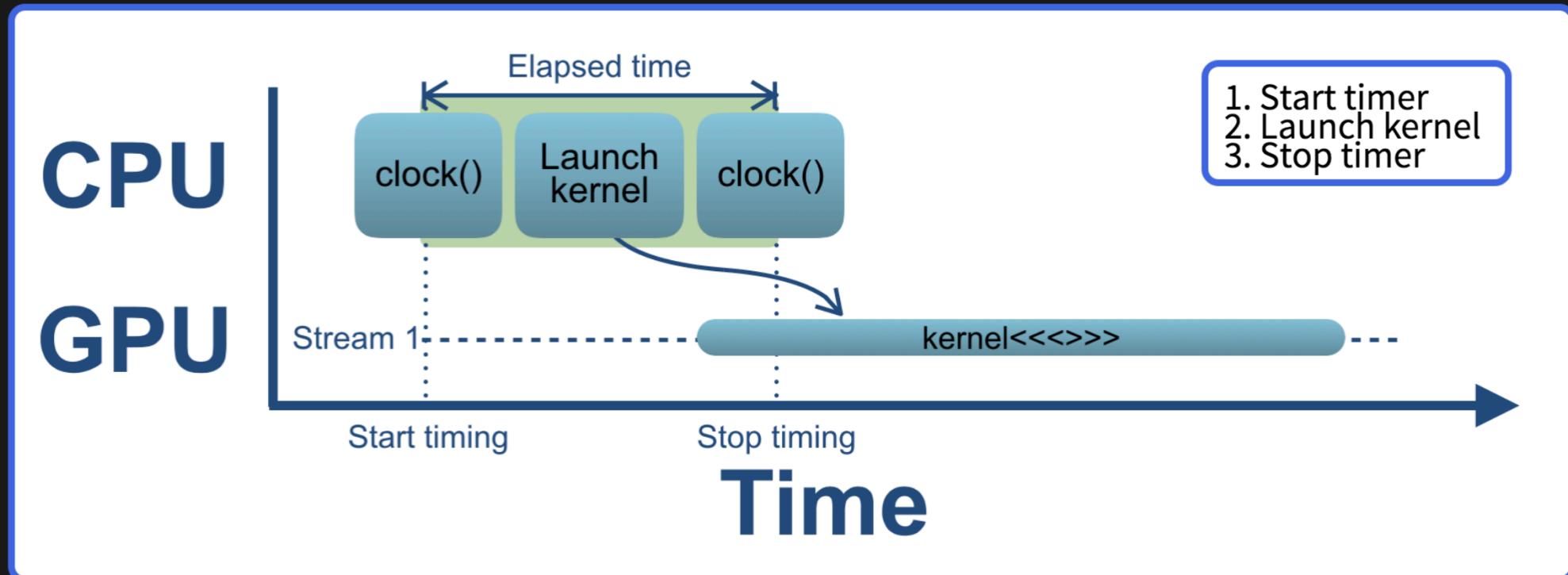
```
1 if (step % loadBalanceInterval == 0) {  
2     float currEff = 0.0, propEff = 0.0;  
3     DistMapping newDM = makeNewDM(costs,  
4                                     currEff, propEff);  
5     bool globUpdateDM = false;  
6     if (myRank == root) {  
7         globUpdateDM = (propEff > 1.1*currEff);  
8     }  
9     bcast(&globUpdateDM, 1, root);  
10    if (globUpdateDM) {  
11        bcast(&newDM[0], newDM.size(), root);  
12        updateDistributionMapping(newDM);  
13    }  
14 }
```

# How should *costs* be measured when running on a GPU-accelerated machine?

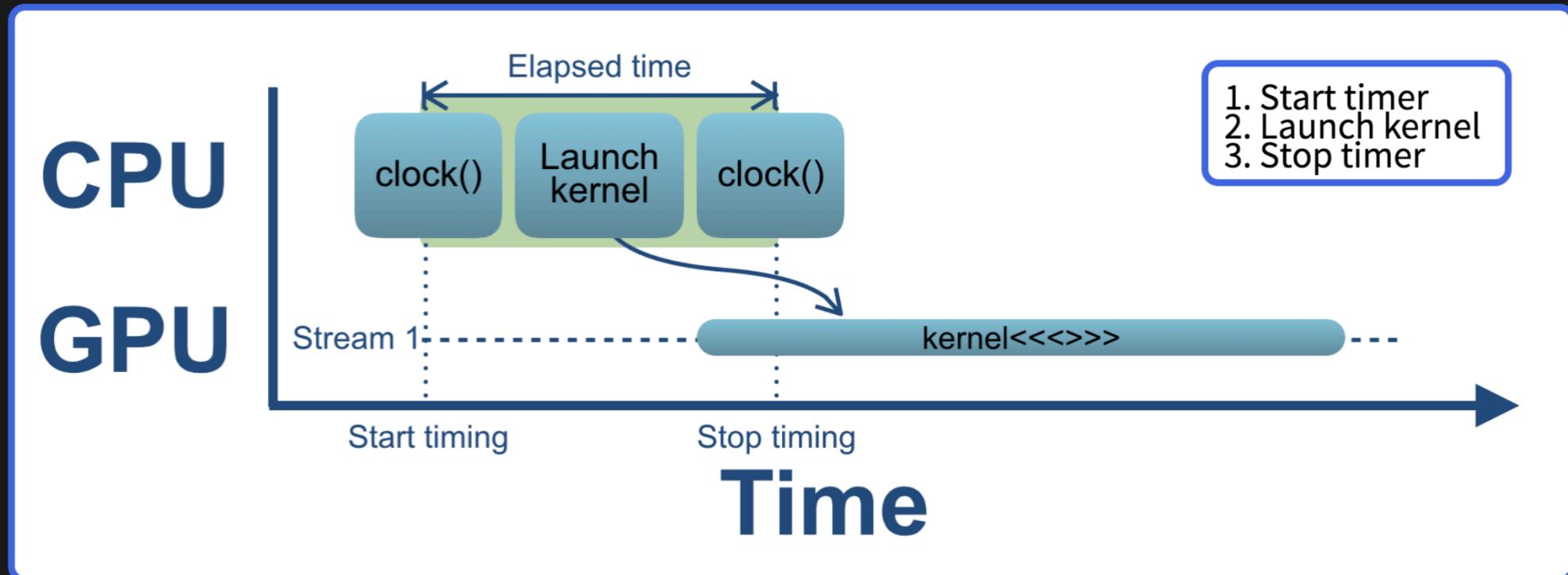
---

1. Start timer
2. Launch kernel
3. Stop timer

# How should *costs* be measured when running on a GPU-accelerated machine?



# How should *costs* be measured when running on a GPU-accelerated machine?



Not like this! CPU and GPU are asynchronous.

# These are a few strategies appropriate for cost assessment on GPU machines:

---

- *Heuristic*: number of particles and cells as proxy for compute work

# These are a few strategies appropriate for cost assessment on GPU machines:

---

- *Heuristic*: number of particles and cells as proxy for compute work
- *CUPTI*: use *CUDA Profiling Tools Interface* to access kernel times

# These are a few strategies appropriate for cost assessment on GPU machines:

---

- *Heuristic*: number of particles and cells as proxy for compute work
- *CUPTI*: use *CUDA Profiling Tools Interface* to access kernel times
- *GPU clock*: use thread-summed kernel times as relative measure of compute work

# How to measure costs with *heuristic*?

---

Cost for rank  $i$  is linear combination of number of particles and cells:

$$c_i = \alpha \cdot n_{\text{particles}} + \beta \cdot n_{\text{cells}}$$

- $\alpha$  and  $\beta$  are parameters representing relative computational cost of single particle vs. single cell
- $\alpha$  and  $\beta$  change depending on algorithm, hardware
- In general,  $\alpha$  and  $\beta$  should be measured

# How to measure costs with *heuristic*?

---

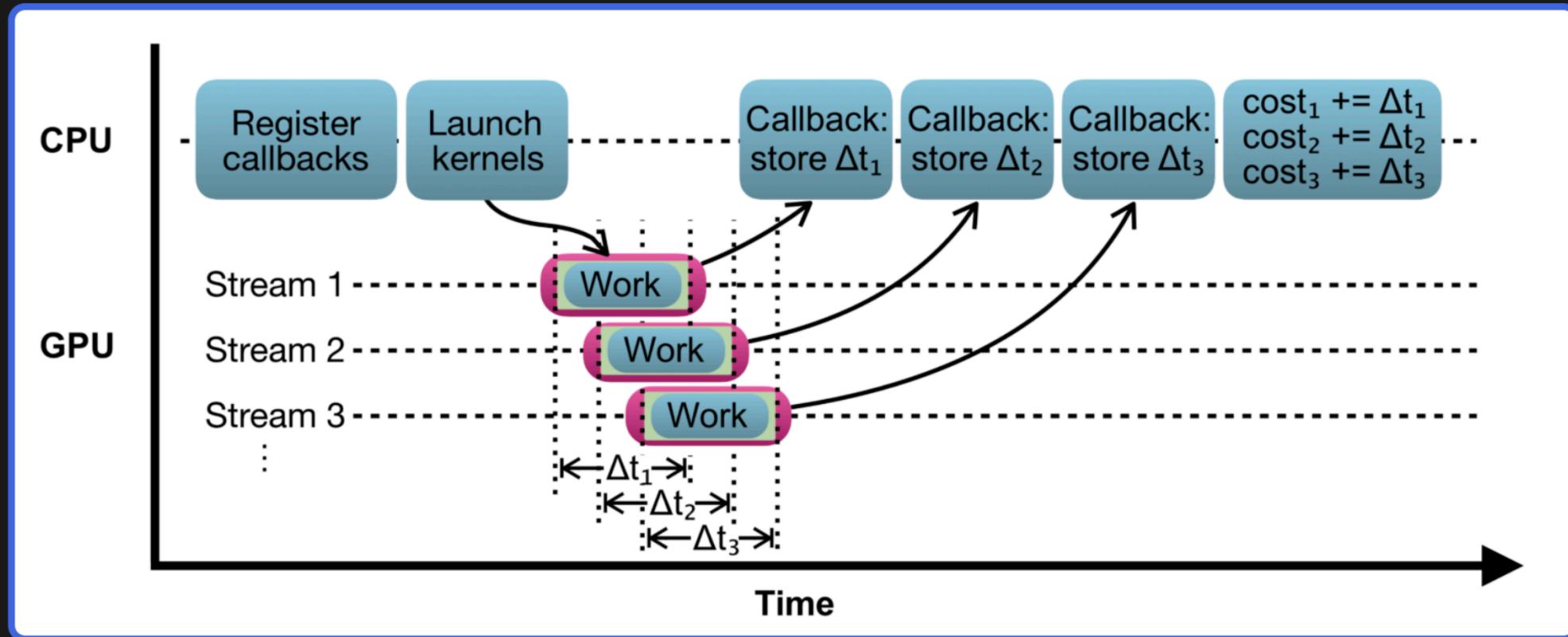
Cost for rank  $i$  is linear combination of number of particles and cells:

$$c_i = \alpha \cdot n_{\text{particles}} + \beta \cdot n_{\text{cells}}$$

- $\alpha$  and  $\beta$  are parameters representing relative computational cost of single particle vs. single cell
- $\alpha$  and  $\beta$  change depending on algorithm, hardware
- In general,  $\alpha$  and  $\beta$  should be measured
- **Cons:** cumbersome tuning of parameters

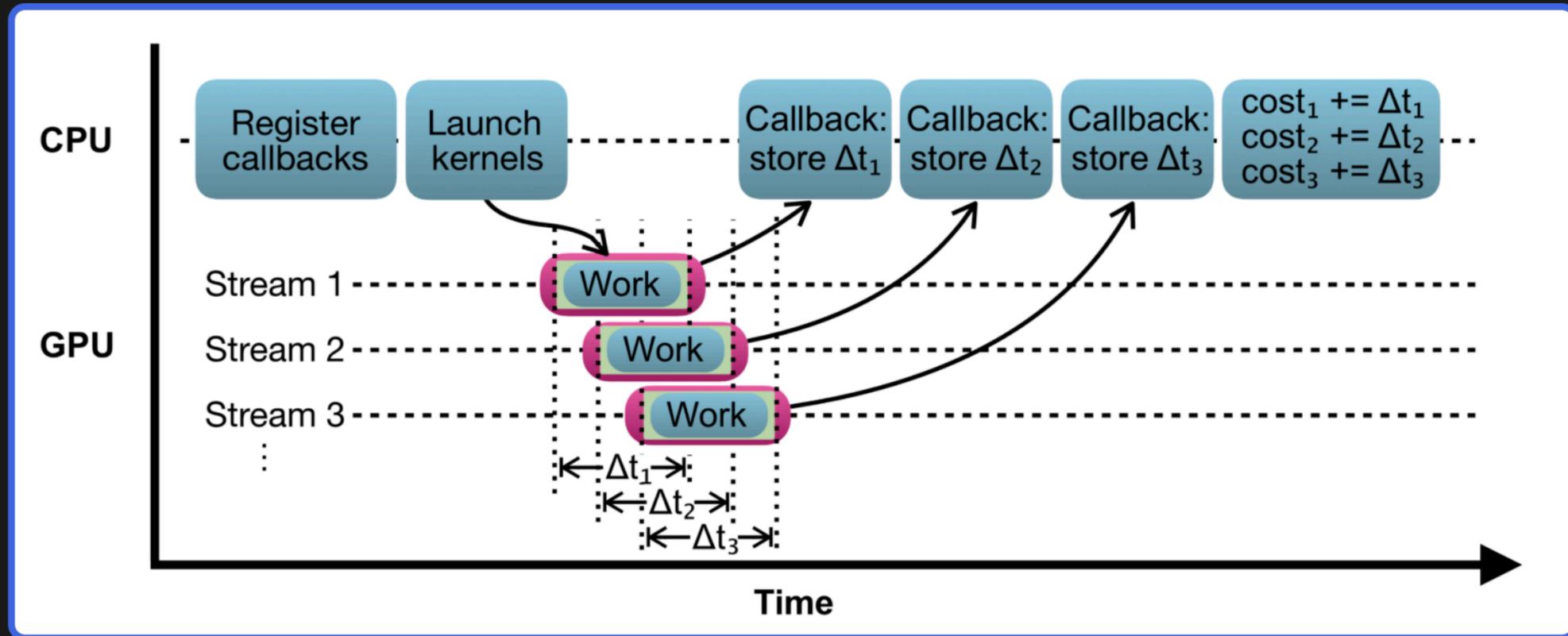
# How to measure costs with CUPTI?

*CUDA Profiling Tools Interface (CUPTI)*: [docs.nvidia.com/cuda/cupti](https://docs.nvidia.com/cuda/cupti)  
GPU activity triggers callback functions to return CUPTI buffers



# How to measure costs with CUPTI?

*CUDA Profiling Tools Interface (CUPTI)*: [docs.nvidia.com/cuda/cupti](https://docs.nvidia.com/cuda/cupti)  
GPU activity triggers callback functions to return CUPTI buffers



- **Cons:** overhead, vendor specific

# How to measure costs with CUPTI?

---

Initialize the trace:

```
1 cuptiActivityEnable(CUPTI_ACTIVITY_KIND_CONCURRENT_KERNEL);  
2 cuptiActivityRegisterCallbacks(bfrRequest, bfrCompleted);
```

Trigger callback functions:

```
1 void CUPTI API bfrRequest (uint8_t **bfr, ...)  
2 {  
3     // Signal to CUPTI client that an empty buffer  
4     // is needed by CUPTI  
5 }  
6 void CUPTI API bfrCompleted (uint8_t *bfr, ...)  
7 {  
8     // Return a buffer of completed activity records  
9     // to CUPTI client  
10 }
```

# How to measure costs with CUPTI?

---

Initialize the trace:

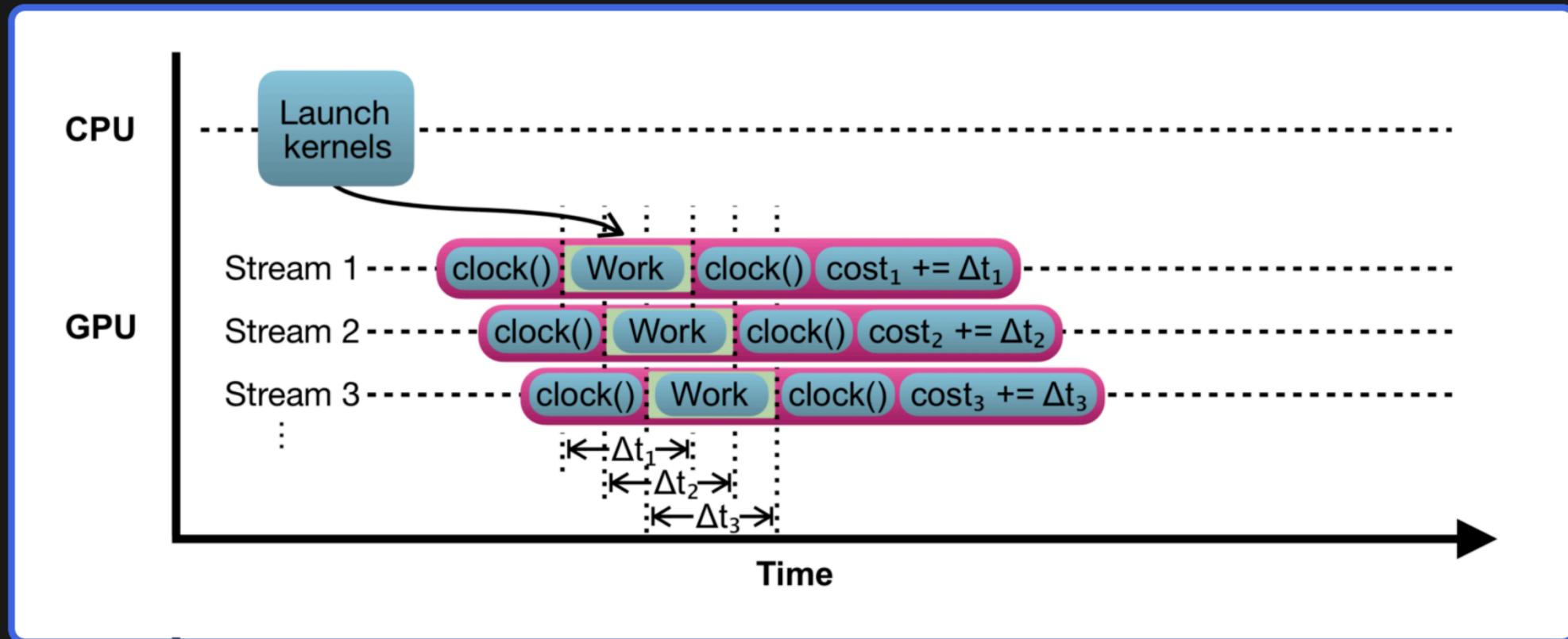
```
1 cuptiActivityEnable(CUPTI_ACTIVITY_KIND_CONCURRENT_KERNEL);  
2 cuptiActivityRegisterCallbacks(bfrRequest, bfrCompleted);
```

Trigger callback functions:

```
1 void CUPTI API bfrRequest (uint8_t **bfr, ...)  
2 {...}  
3 void CUPTI API bfrCompleted (uint8_t *bfr, ...)  
4 {...}  
5  
6 :  
7  
8 mykernel<<<...>>>(...);  
9 cuptiActivityFlushAll(0); // Wait for return of CUPTI  
10 → bfrCompleted(...); // records via callback function
```

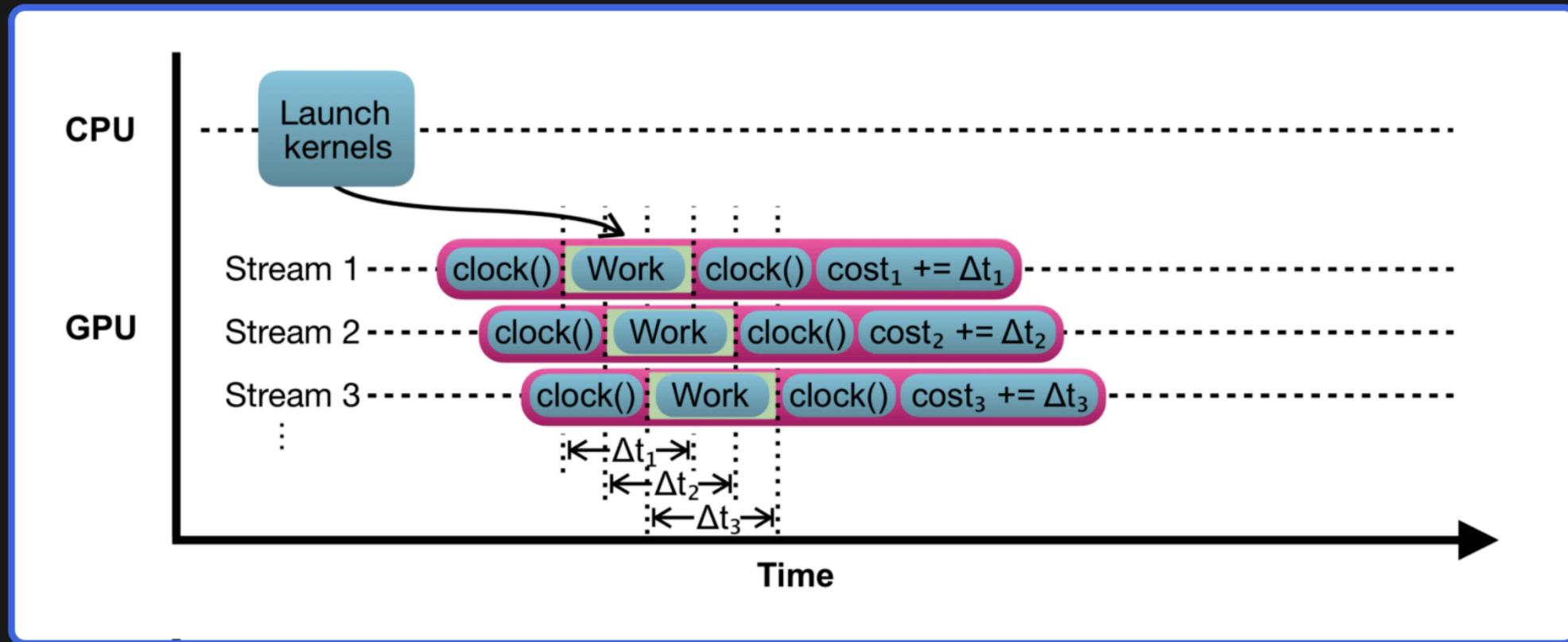
# How to measure costs with GPU clock?

Estimate relative compute work from thread-summed kernel time



# How to measure costs with GPU clock?

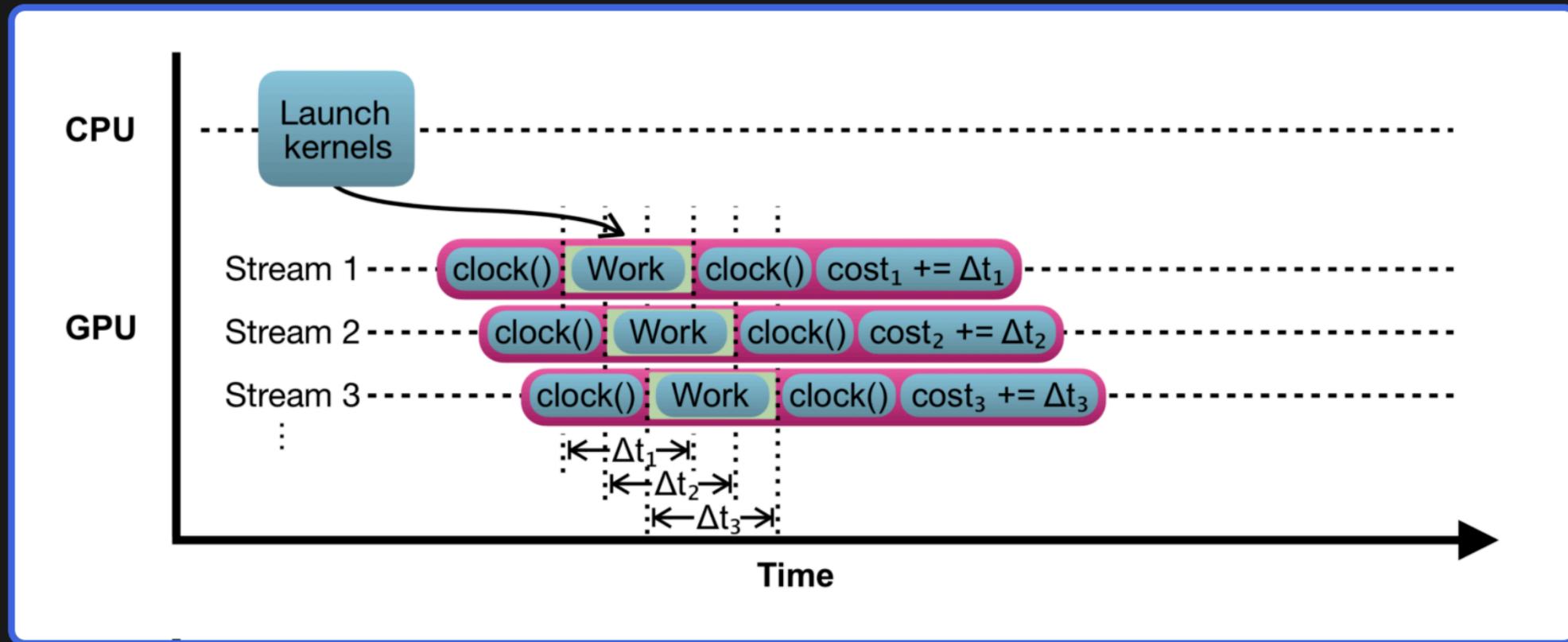
Estimate relative compute work from thread-summed kernel time



- **Pros:** vendor agnostic, no hyperparameter tuning

# How to measure costs with GPU clock?

Estimate relative compute work from thread-summed kernel time



- **Pros:** vendor agnostic, no hyperparameter tuning
- **Cons:** requires some data movement

# How to measure costs with *GPU clock*?

---

Add the thread cycles, using `atomicAdd` for thread safety:

```
1 __global__ void mykernel (...) {
2     float cycles = clock();
3     :
4     // thread work
5     :
6     cycles = clock() - cycles;
7
8     // cost_ptr is the pointer to rank's cost
9     atomicAdd(cost_ptr, cycles);
10 }
```

- Reduced overhead using pinned host memory

# How to measure costs with *GPU clock*?

---

Add the thread cycles, using `atomicAdd` for thread safety:

```
1 __global__ void mykernel (...) {
2     float cycles = clock();
3     :
4     // thread work
5     :
6     cycles = clock() - cycles;
7
8     // cost_ptr is the pointer to rank's cost
9     atomicAdd(cost_ptr, cycles);
10 }
```

- Reduced overhead using pinned host memory
- To use this: instrument most expensive kernels

# How to measure costs with *GPU clock*?

---

Add the thread cycles, using `atomicAdd` for thread safety:

```
1 __global__ void mykernel (...) {
2     float cycles = clock();
3     :
4     // thread work
5     :
6     cycles = clock() - cycles;
7
8     // cost_ptr is the pointer to rank's cost
9     atomicAdd(cost_ptr, cycles);
10 }
```

- Reduced overhead using pinned host memory
- To use this: instrument most expensive kernels
- Overcomes weakness of heuristic: that has no sensitivity to how much particles move

# Outline:

---

1. Load balancing intro
2. Dynamic load balancing in  
PIC code run on GPUs

# We studied these strategies in the particle-in-cell code WarpX.

WarpX: advanced PIC code

- [github.com/ECP-WarpX/WarpX](https://github.com/ECP-WarpX/WarpX)

AMReX: mesh framework

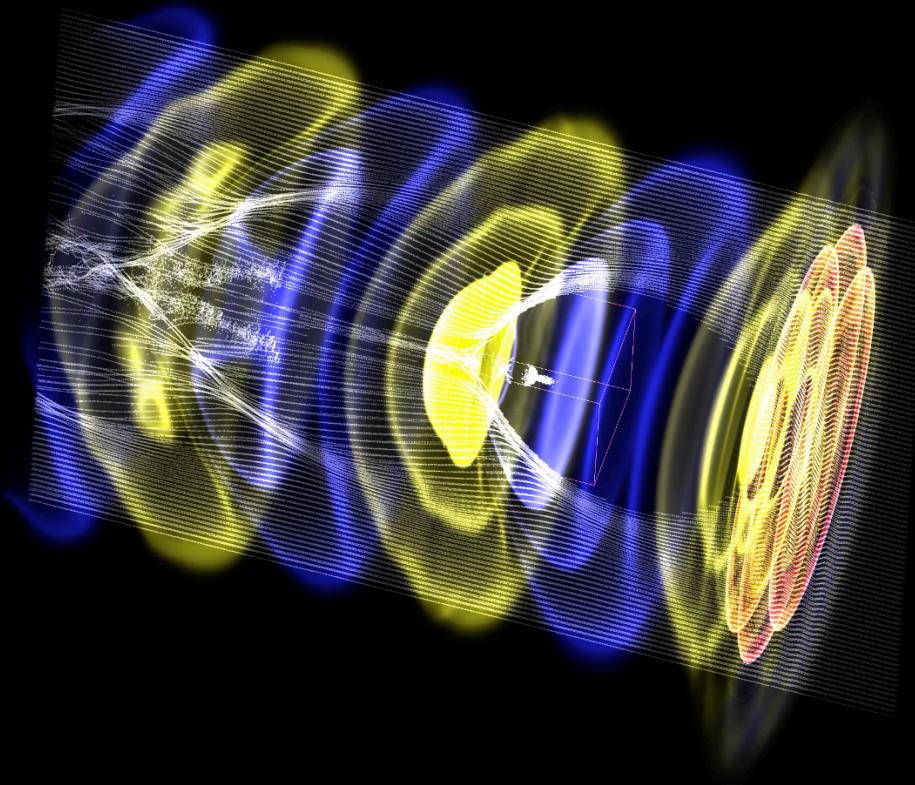
- [github.com/AMReX-Codes/amrex](https://github.com/AMReX-Codes/amrex)

**WarpX**  
advanced physics

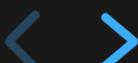
**AMReX**  
mesh infrastructure, algorithms

MPI

CUDA, OpenMP, DPC++, HIP



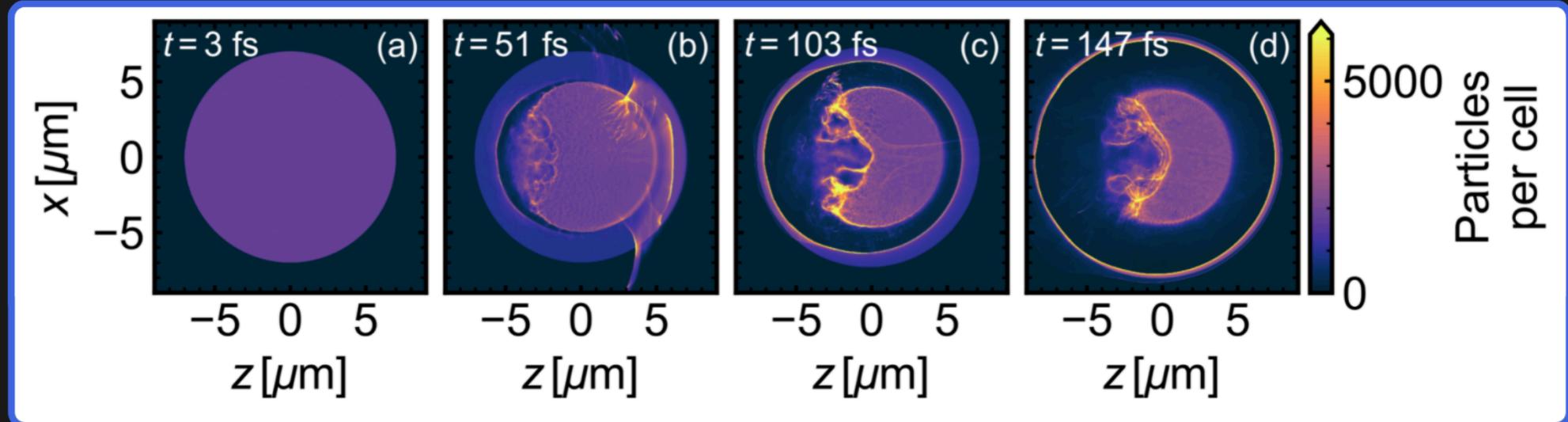
Courtesy of Max Thevenet



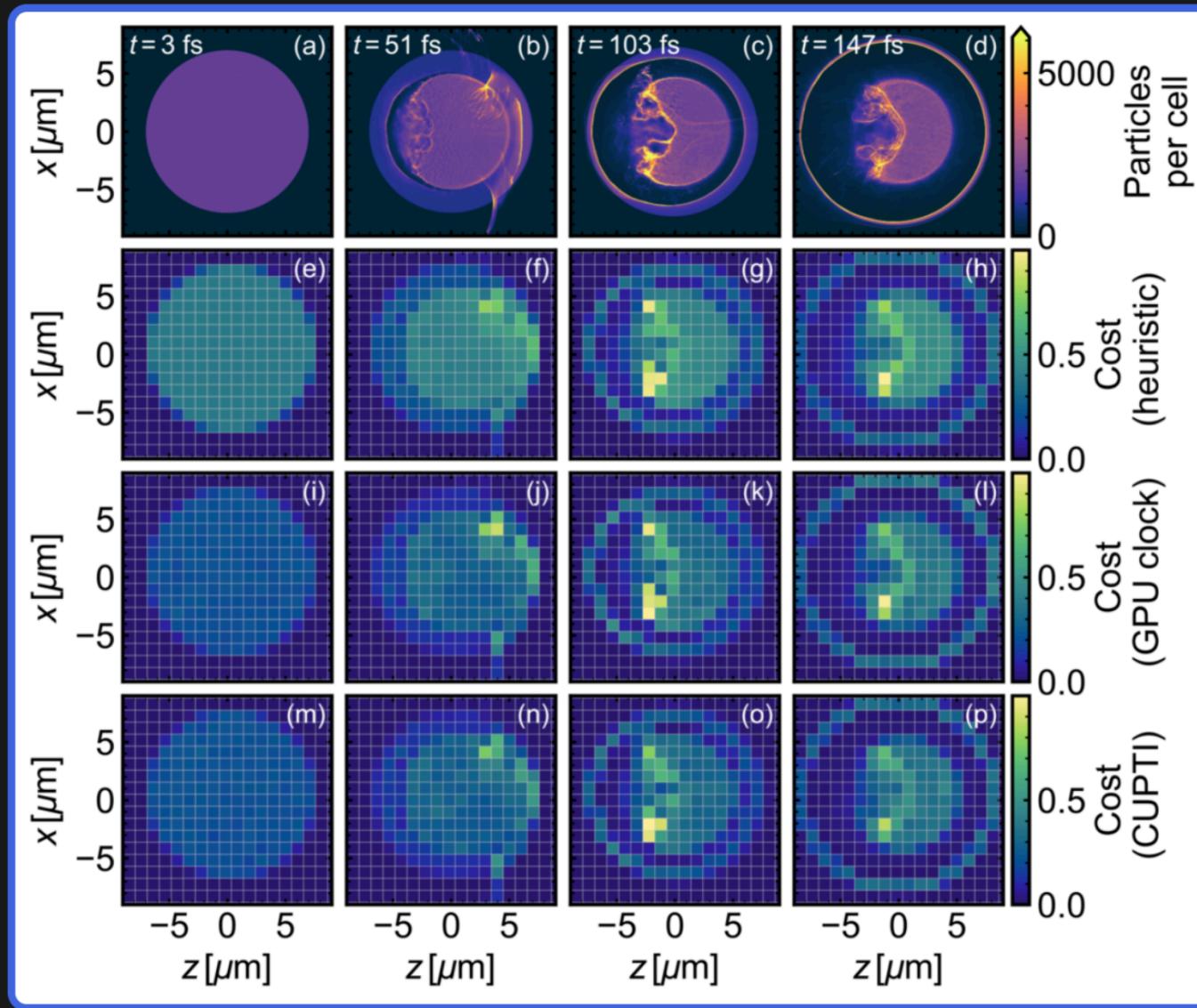
# We choose *laser-ion acceleration* as a challenging test problem.

Rapid changes in particle, field spatial profiles → challenge problem

Numerical experiments: 6–6144 Nvidia V100 GPUs on OLCF Summit



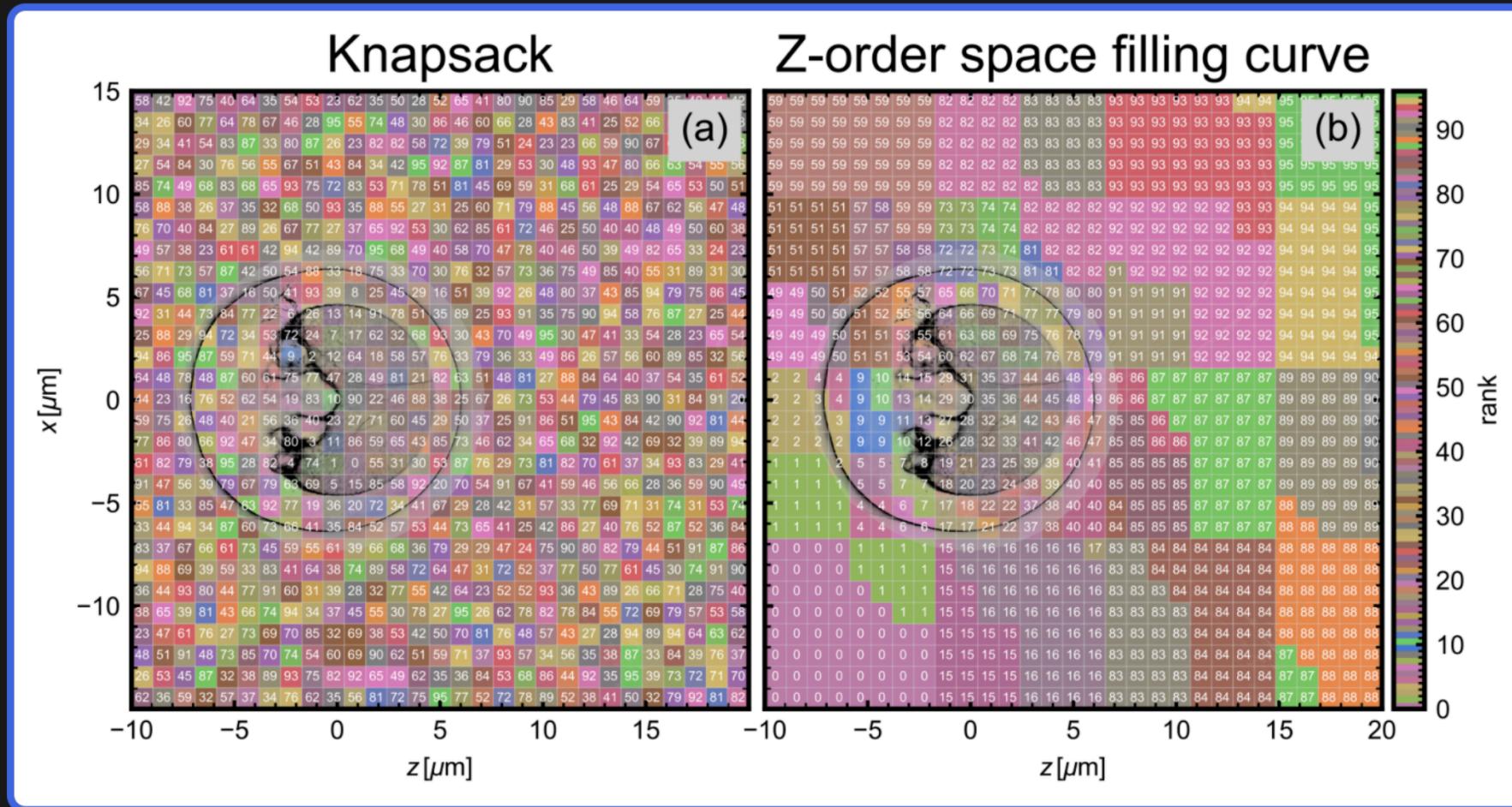
# The inhomogeneity translates to different computational costs.



# Computational costs are used to compute optimal mapping from MPI rank to domain.

*Knapsack*: distribute costs to ranks as equally as possible

*Space-filling curve (SFC)*: enumerate ranks along curve and partition

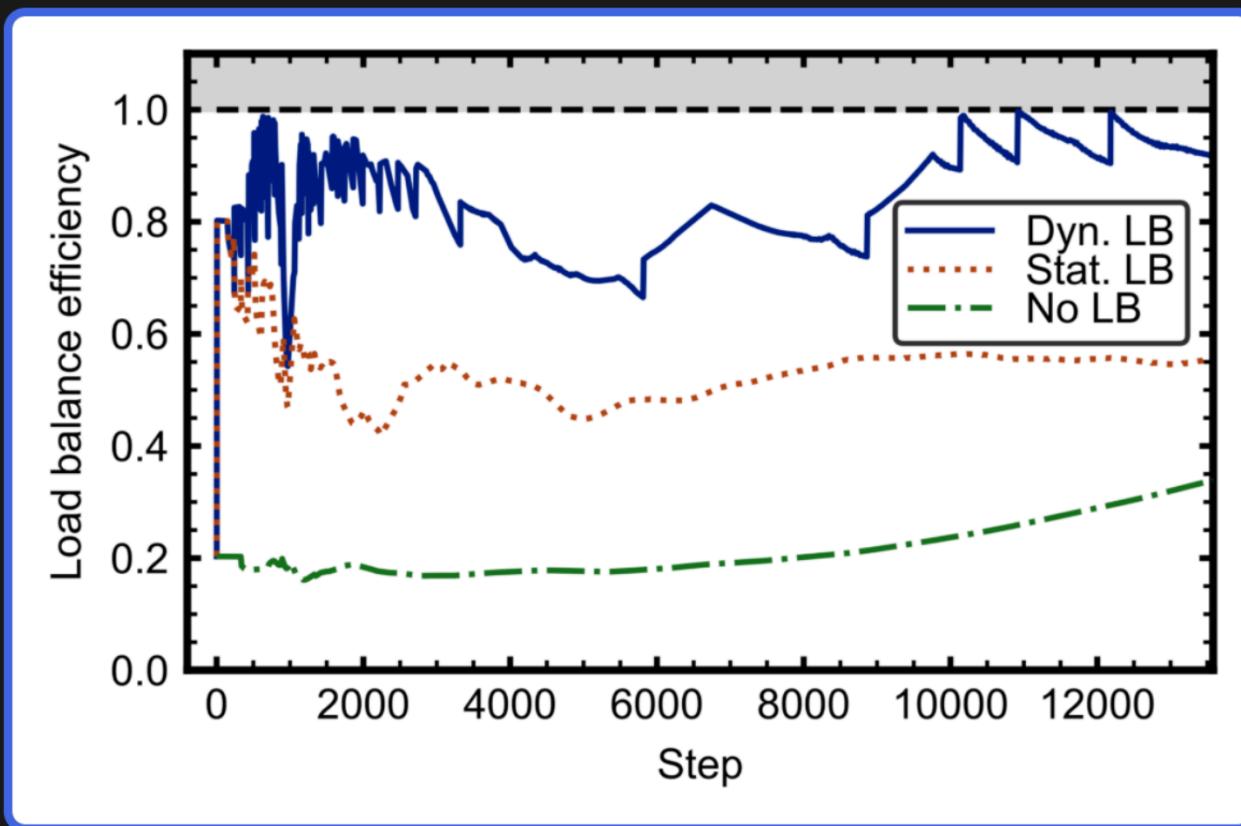


# Dynamic load balancing is crucial to performance.

Static load balancing  
is not enough!

Efficiency: average  
cost/mean cost

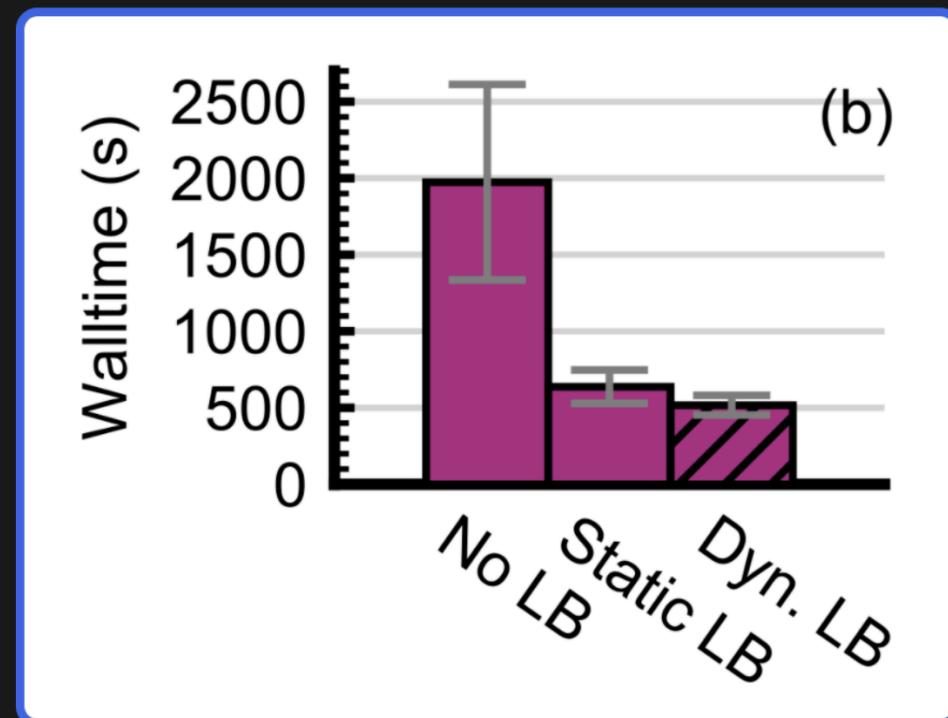
$$E \equiv c_{\text{avg}}/c_{\text{max}}$$



# With optimal selection of parameters, we achieve around 3x–4x speedup.

Optimal performance with:

- GPU clock cost collection
- Knapsack algorithm
- 9 boxes per GPU
- 10 steps to check rebalance
- 10% improvement threshold

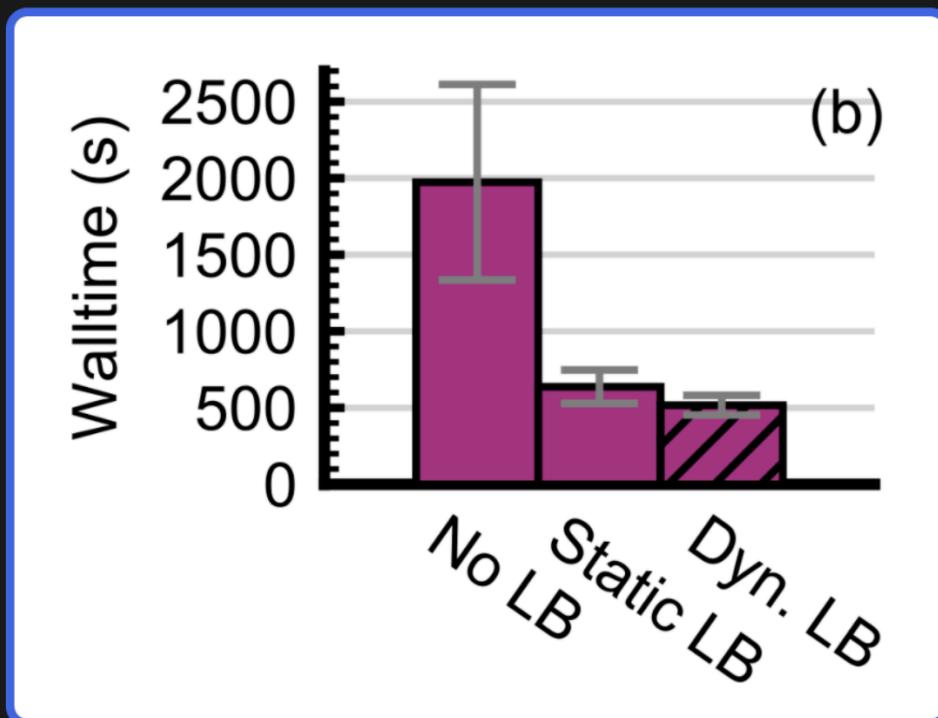


# With optimal selection of parameters, we achieve around 3x–4x speedup.

Optimal performance with:

- GPU clock cost collection
- Knapsack algorithm
- 9 boxes per GPU
- 10 steps to check rebalance
- 10% improvement threshold

1.2x speedup over static lb



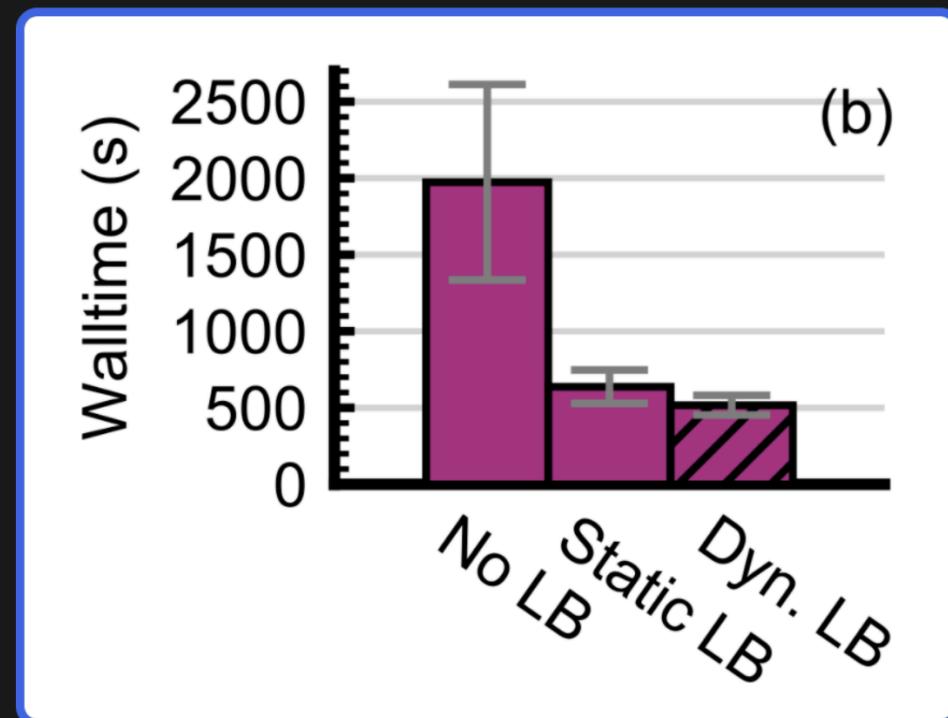
# With optimal selection of parameters, we achieve around 3x–4x speedup.

Optimal performance with:

- GPU clock cost collection
- Knapsack algorithm
- 9 boxes per GPU
- 10 steps to check rebalance
- 10% improvement threshold

1.2x speedup over static lb

3.8x speedup over no lb

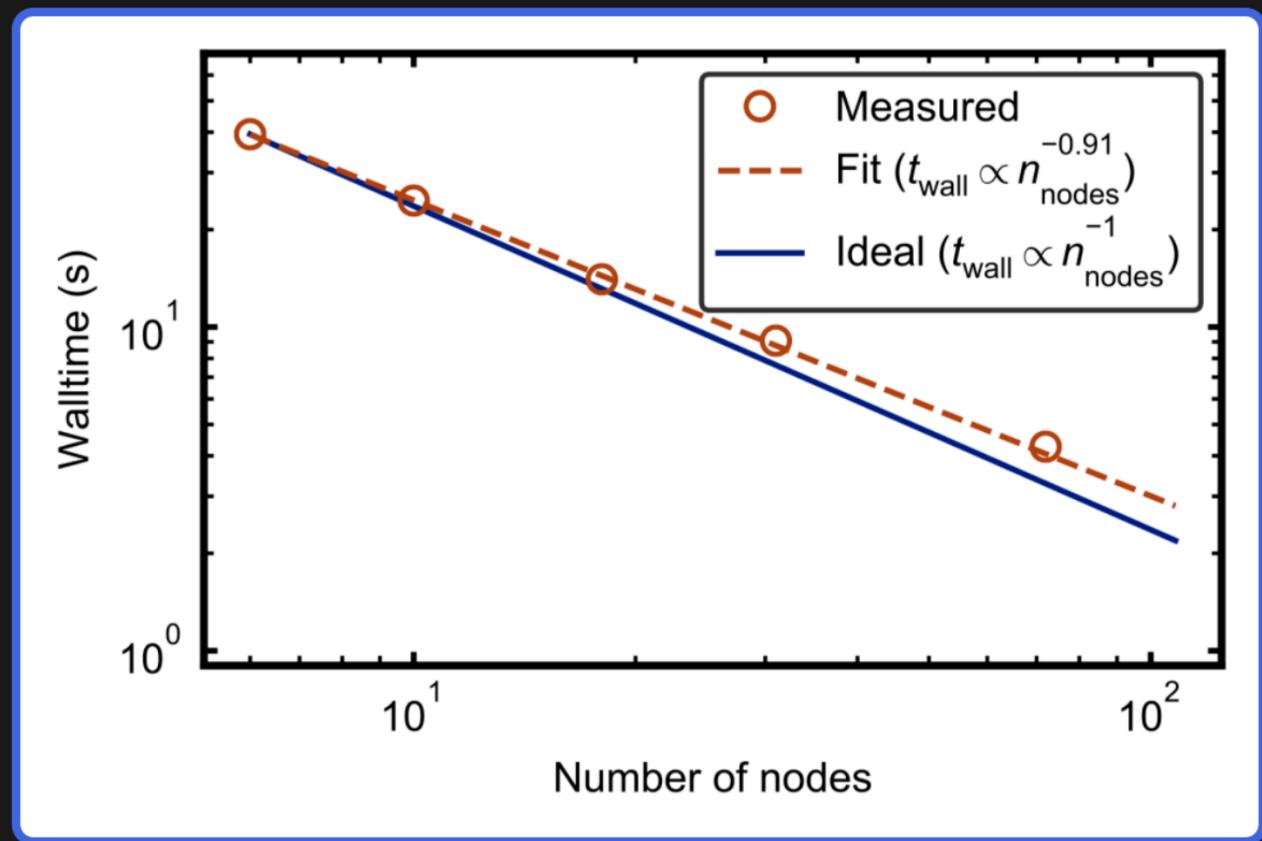


# How much improvement expected from load balancing?

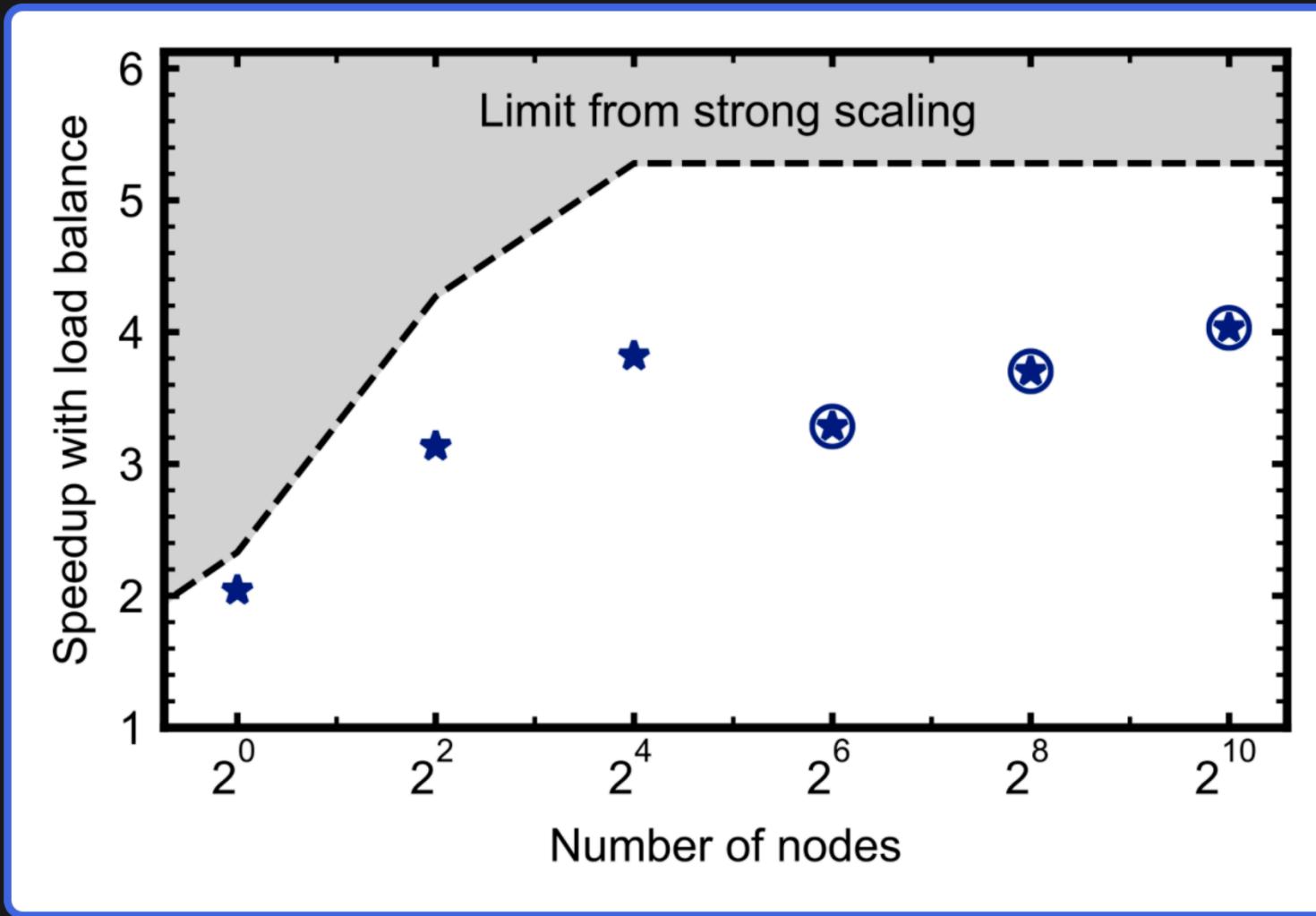
Performance model w/  
strong-scaling as input:

$$S = \left( \frac{c_{\max 0}}{c_{\text{avg}0}} \right)^x = \left( \frac{1}{E_0} \right)^x$$

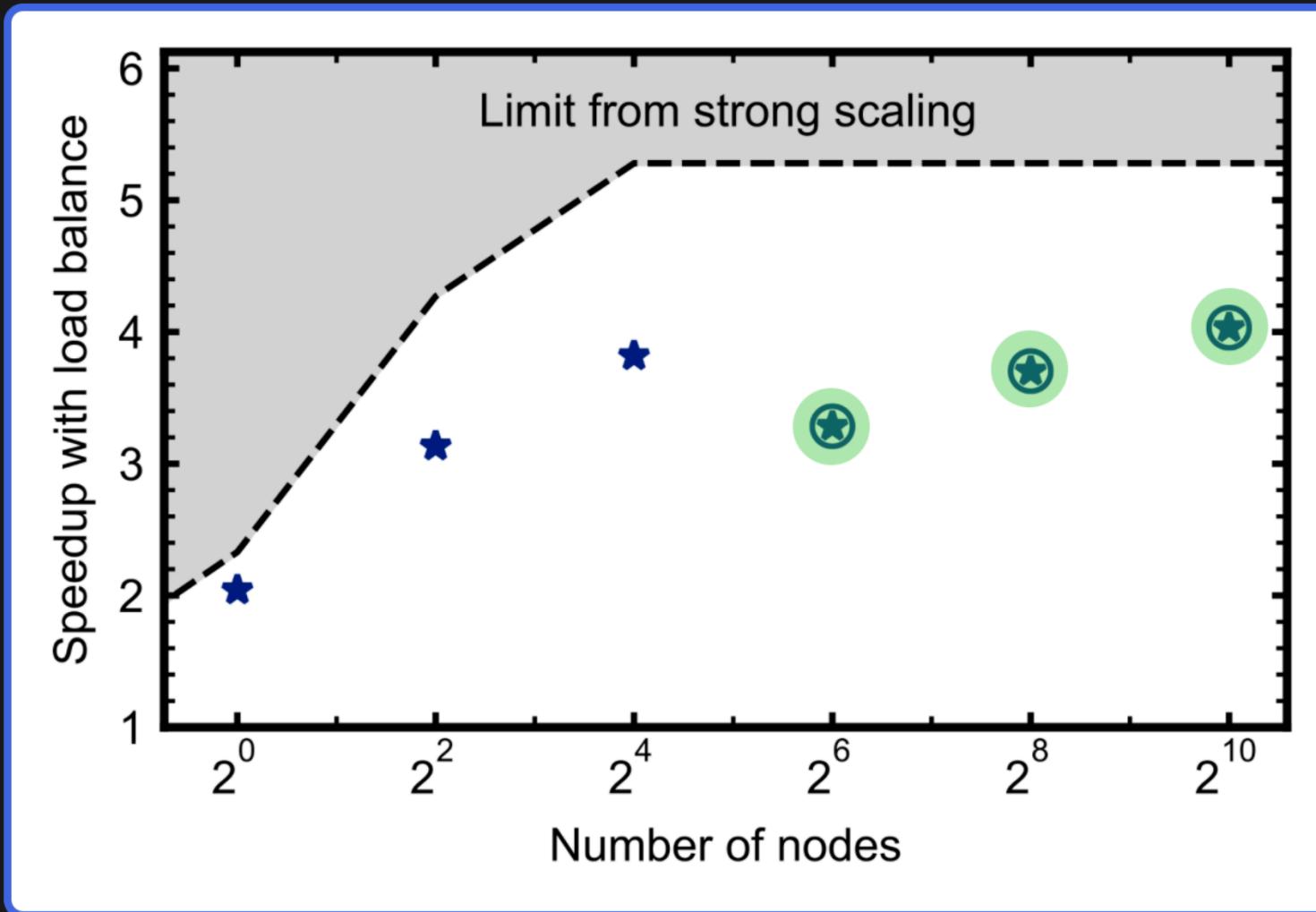
Estimate speedup  $S$  as  
 $\propto$  initial load imbalance



The load balancing scheme achieves 62%-74% of theoretical maximum.



The load balancing scheme achieves 62%-74% of theoretical maximum.



Avoid out-of-memory on GPUs with load balancing!

# What is new/innovative about this work?

---

- Introduced GPU-applicable strategies for measuring relative computational costs of sub-domains of computational work

# What is new/innovative about this work?

---

- Introduced GPU-applicable strategies for measuring relative computational costs of sub-domains of computational work
- Implemented potentially vendor-neutral, in-situ, in-kernel cost measurement strategy based on GPU clock

# What is new/innovative about this work?

---

- Introduced GPU-applicable strategies for measuring relative computational costs of sub-domains of computational work
- Implemented potentially vendor-neutral, in-situ, in-kernel cost measurement strategy based on GPU clock
- Implemented Nvidia CUPTI cost measurement → overhead

# What is new/innovative about this work?

---

- Introduced GPU-applicable strategies for measuring relative computational costs of sub-domains of computational work
- Implemented potentially vendor-neutral, in-situ, in-kernel cost measurement strategy based on GPU clock
- Implemented Nvidia CUPTI cost measurement → overhead
- Demonstrated effective GPU dynamic load balancing running challenging use case WarpX at scale (6-6144 GPUs) on Summit

# What is new/innovative about this work?

---

- Introduced GPU-applicable strategies for measuring relative computational costs of sub-domains of computational work
- Implemented potentially vendor-neutral, in-situ, in-kernel cost measurement strategy based on GPU clock
- Implemented Nvidia CUPTI cost measurement → overhead
- Demonstrated effective GPU dynamic load balancing running challenging use case WarpX at scale (6-6144 GPUs) on Summit
- Introduced strong-scaling based performance model

# With new strategies for GPU cost assessment, we achieved 3x–4x speedup on challenging plasma physics problem.

---

Work is open source:

- WarpX: [github.com/ECP-WarpX/WarpX](https://github.com/ECP-WarpX/WarpX)
- AMReX: [github.com/AMReX-Codes/amrex](https://github.com/AMReX-Codes/amrex)

Code, environment, tests all available at:

- <https://zenodo.org/record/4708449#.YIEmmJNKhR0>

See preprint here:

- <https://arxiv.org/abs/2104.11385>

Personal github:

- <https://github.com/mrowan137>

# WarpX team\*: physicists + applied mathematicians + computer scientists



Jean-Luc  
Vay (PI)



Diana  
Amorim



Axel  
Huebl



Rémi  
Lehe



Olga  
Shapoval



Yinjian  
Zhao



Edoardo  
Zion



Ann  
Almgren (coPI)



John  
Bell



Kevin  
Gott



Revathi  
Jambunathan



Andrew  
Myers



Michael  
Rowan



Eloise  
Yang



Weiwen  
Zhang



ACCELERATOR TECHNOLOGY &  
APPLIED PHYSICS DIVISION



David  
Grote (coPI)



+ growing list of international collaborators

Henri  
Vincenti



Luca  
Fedeli



Antonin  
Sainte-Marie



Neil  
Zaim



Marc  
Hogan (coPI)



Lixin  
Ge



Cho  
Ng



Maxence  
Thévenet



Severin  
Diederichs



Lorenzo  
Giacomel



# WarpX team\*: physicists + applied mathematicians + computer scientists



Jean-Luc Vay (PI)	Diana Amorim	Axel Huebl	Rémi Lehe	Olga Shapoval	Yinjian Zhao	Edoardo Zion	ATAP ACCELERATOR TECHNOLOGY & APPLIED PHYSICS DIVISION
Ann Almgren (coPI)	John Bell	Kevin Gott	Revathi Jambunathan	Andrew Myers	Michael Rowan	Eloise Yang	AMReX
							AMReX
David Grote (coPI)	(NESAP)	(NESAP)	(NESAP)	(NESAP)	(NESAP)	Weiqun Zhang	NERSC NESAP
							NESAP
Marc Hogan (coPI)	Lixin Ge	Cho Ng	+ growing list of international collaborators	Henri Vincenti	Luca Fedeli	Antonin Sainte-Marie	Neil Zaim
SLAC	Maxence Thévenet	Severin Diederichs	Lorenzo Giacometti				



Thank you! I am happy to answer any questions.

# Performance is tuned with additional algorithm-specific parameters.

*Heuristic, GPU clock, CUPTI* : cost collection method

*Knapsack, SFC* : algorithm to update distribution mapping

*Boxes per GPU* : controls size of domain decomposition

*Load balance interval* : how often to try rebalancing

*Improvement threshold* : required improvement to rebalance

