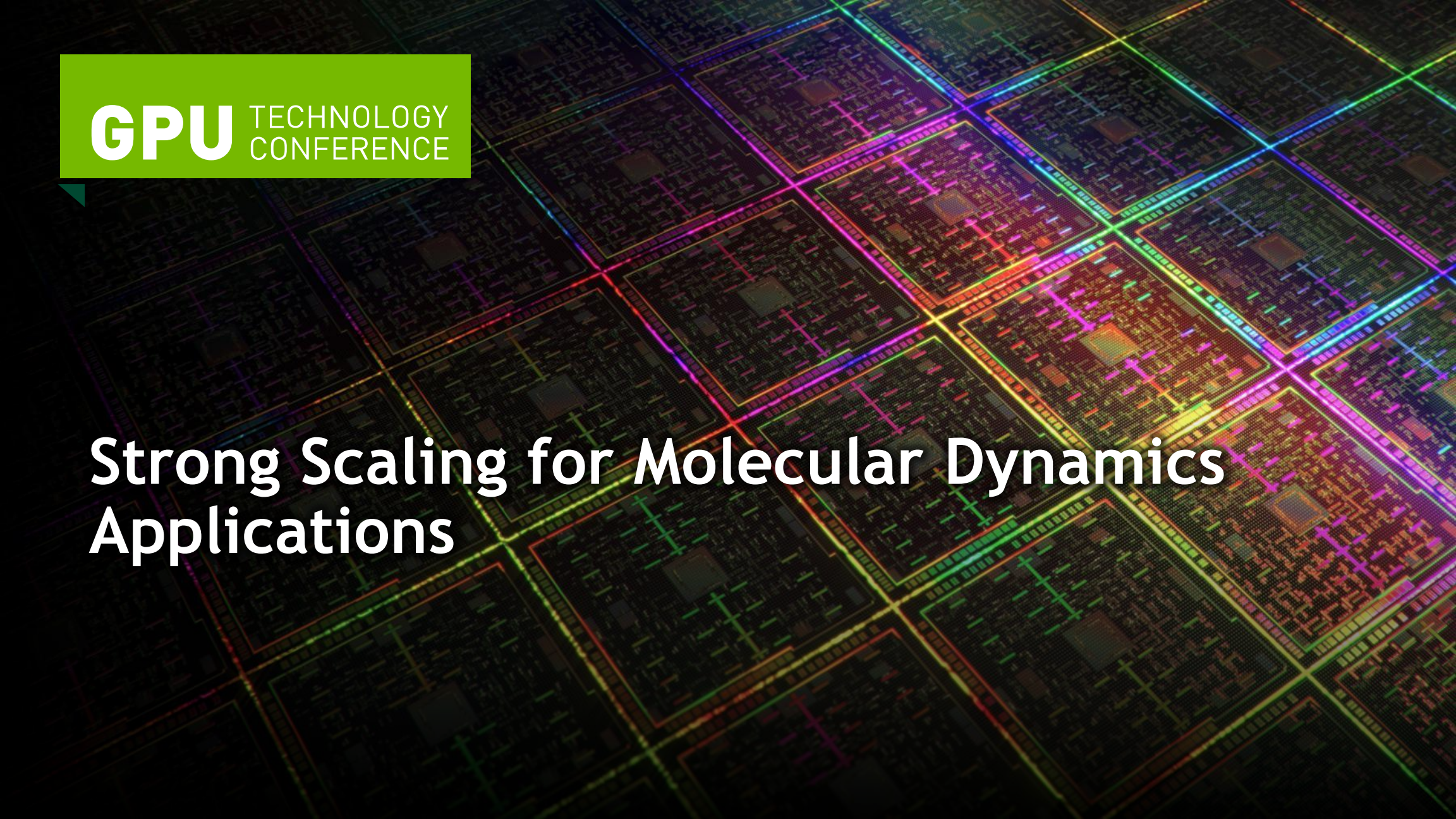
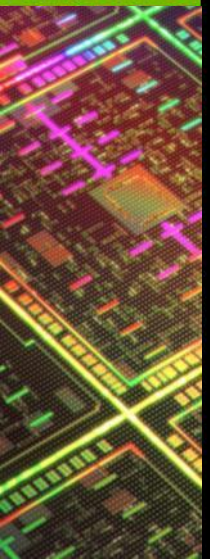
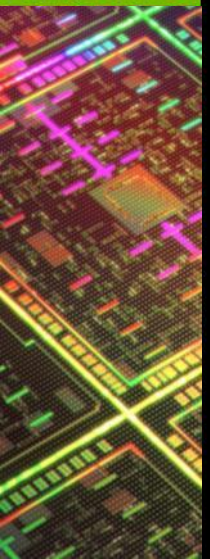


Strong Scaling for Molecular Dynamics Applications



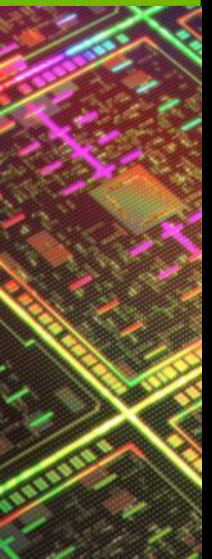
Scaling and Molecular Dynamics

- Strong Scaling
  - How does solution time vary with the number of processors for a fixed problem size
- Classical Molecular Dynamics
  - at each time step
    - Compute pairwise forces on atoms
    - integrate atoms forward
  - Forces can either be bonded (the atoms are covalently bound), or non bonded
  - Limit amount of direct forces computed by using a cutoff distance
    - Estimate forces outside cut off to infinity with FFT
- Strong Scaling + Molecular Dynamics
  - For a single run can only improve performance by speeding up the time step
    - Time steps are already in milliseconds
  - Need strong scaling because need large number of time steps to study phenomenon of biological interest



Overview of talk

- This talk is a look at the issues that we found while trying to improve the scaling performance of MD codes, mostly focusing on NAMD
  - I will try to make these lessons as general as possible
  - This is not a comprehensive list of all scaling issues an application might face



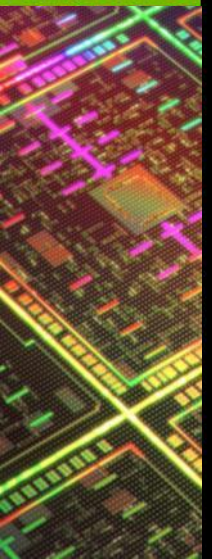
What could cause scaling issues on GPU nodes

- We will focus on issues that would be different than what you would face on your CPU path

1. GPU Kernel(s) itself does not scale

2. The code that is setting up work for the GPU / managing the GPU / processing work from the GPU does not scale

3. Something else, like communication is becoming the bottleneck
   - the GPU might be speeding up the computation to the extent that it no longer is the bottleneck



Process

- Profile! Over multiple processor counts observe how the following vary
  - Time spent in GPU functions
  - Time spent in CPU functions
  - Communication cost
- Look at timeline traces to help understand bottlenecks and and validate hypothesis

Example - AMBER

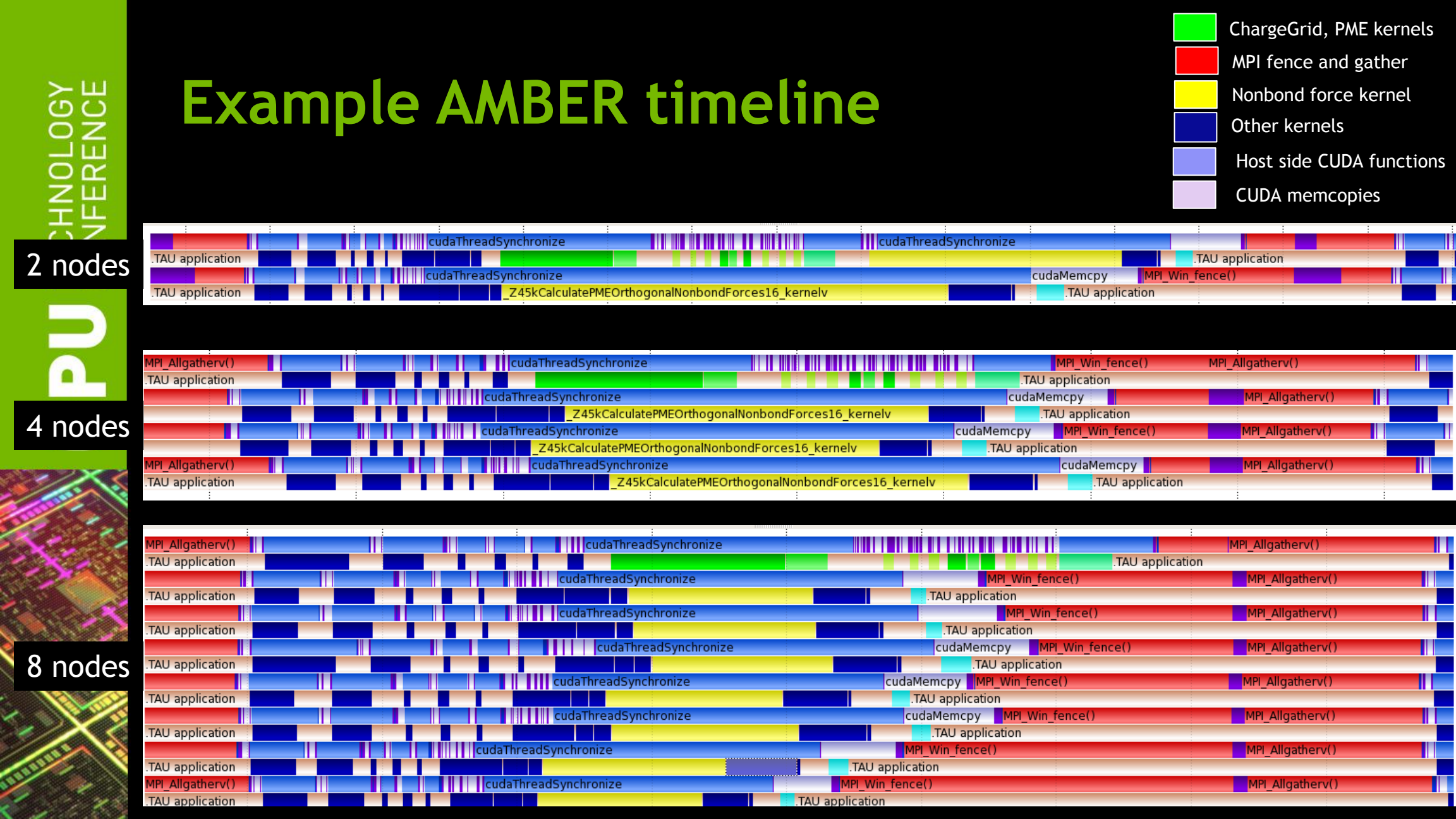
- JAC NVE on M2090

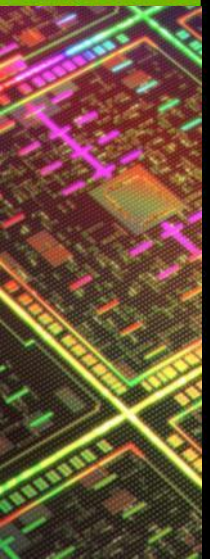
	1 node	2 nodes	4 nodes	6 nodes	8 nodes
ns/day	35.7	49.16	69.68	79.73	85.21
Parallel efficiency	1.00	0.69	0.49	0.37	0.30

Example AMBER profile

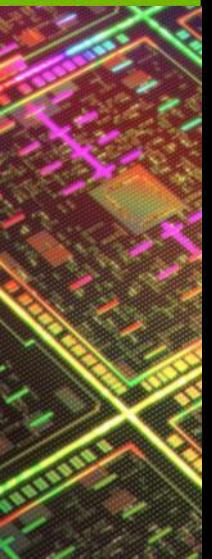
	Time in ms			% of timestep			Parallel efficiency		
	2 nodes	4 nodes	8 nodes	2 nodes	4 nodes	8 nodes	2 nodes	4 nodes	8 nodes
Build neighbor list	0.171	0.087	0.051				1.000	0.982	0.844
Non bond forces	1.030	0.548	0.347				1.000	0.939	0.742
MPI win fence	0.291	0.205	0.280				1.000	0.710	0.259
MPI all gather	0.193	0.324	0.404				1.000	0.298	0.119
PME fill charge buffer	0.339	0.346	0.386				1.000	0.490	0.220

Example AMBER timeline





Kernel Scaling



Possible reasons for Kernel timing not scaling

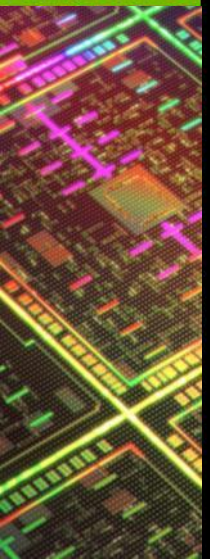
- Is the work to the kernel scaling? I.e. Is it reduced to 1/n per GPU when running n GPUs?
- Are you running less threads than what you need to saturate the machine?
- Are we running into the tail effect?

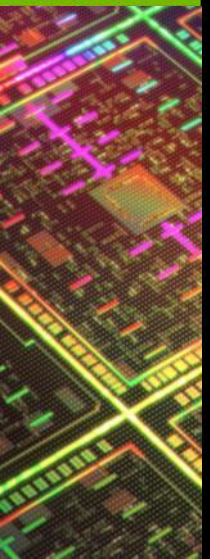
Work executed by the GPU not scaling? AMBER

- Some of the functions executed on only one GPU

	2	4	6	8
Time taken for serial PME				

- Amdahl's law





Not enough work to saturate the GPU

- Try to increase occupancy
  - is occupancy limited by register or shared memory usage?
  - Can you reduce the amount of work each thread does in order to increase total amount of threads on GPU?
- Are the multiple kernels that you could run concurrently?
- Is there more work that you can port to the GPU?

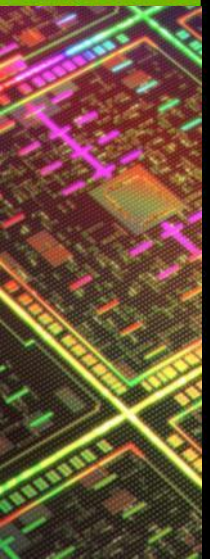
Kernel not Scaling NAMD

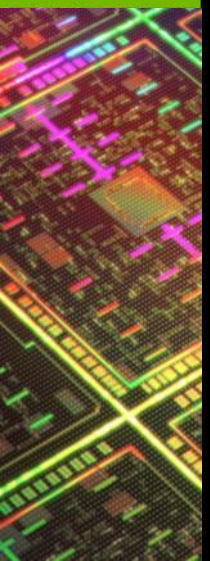
- NAMD kernel scaling

	1 node	8 nodes	16 nodes	32 nodes
Apoa1	27.3	3.9	2.3	1.5
Parallel efficiency	1	0.88	0.74	0.57

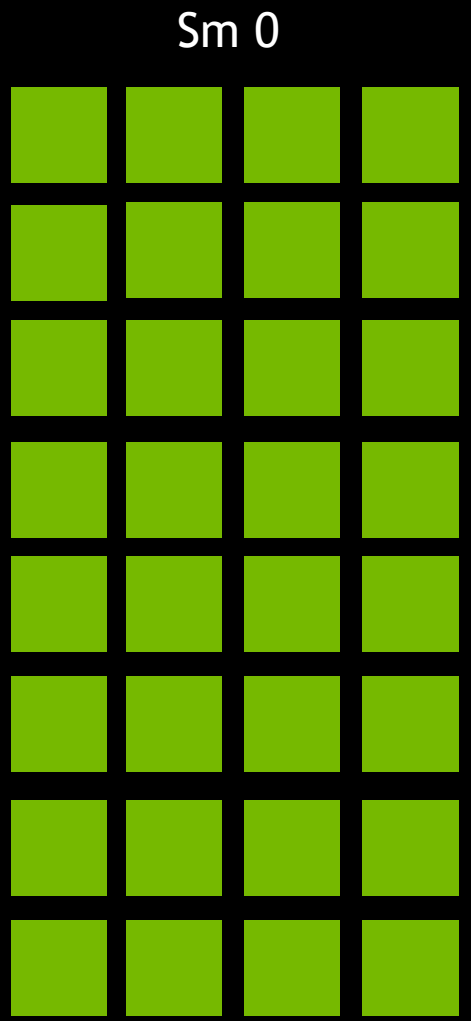
- In the case of NAMD the amount of work to the kernel was scaling perfectly and there was enough work

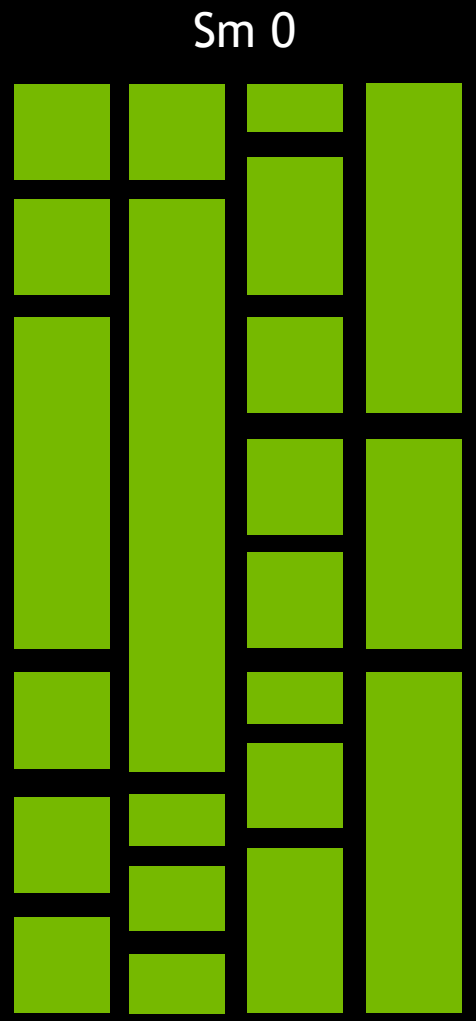
- Hypothesis: some blocks are taking a very long time, causing low SM occupancy towards the tail end of the kernel, resulting in the scaling issues



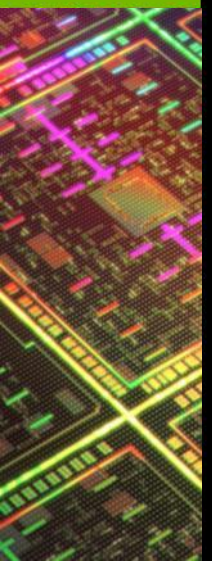


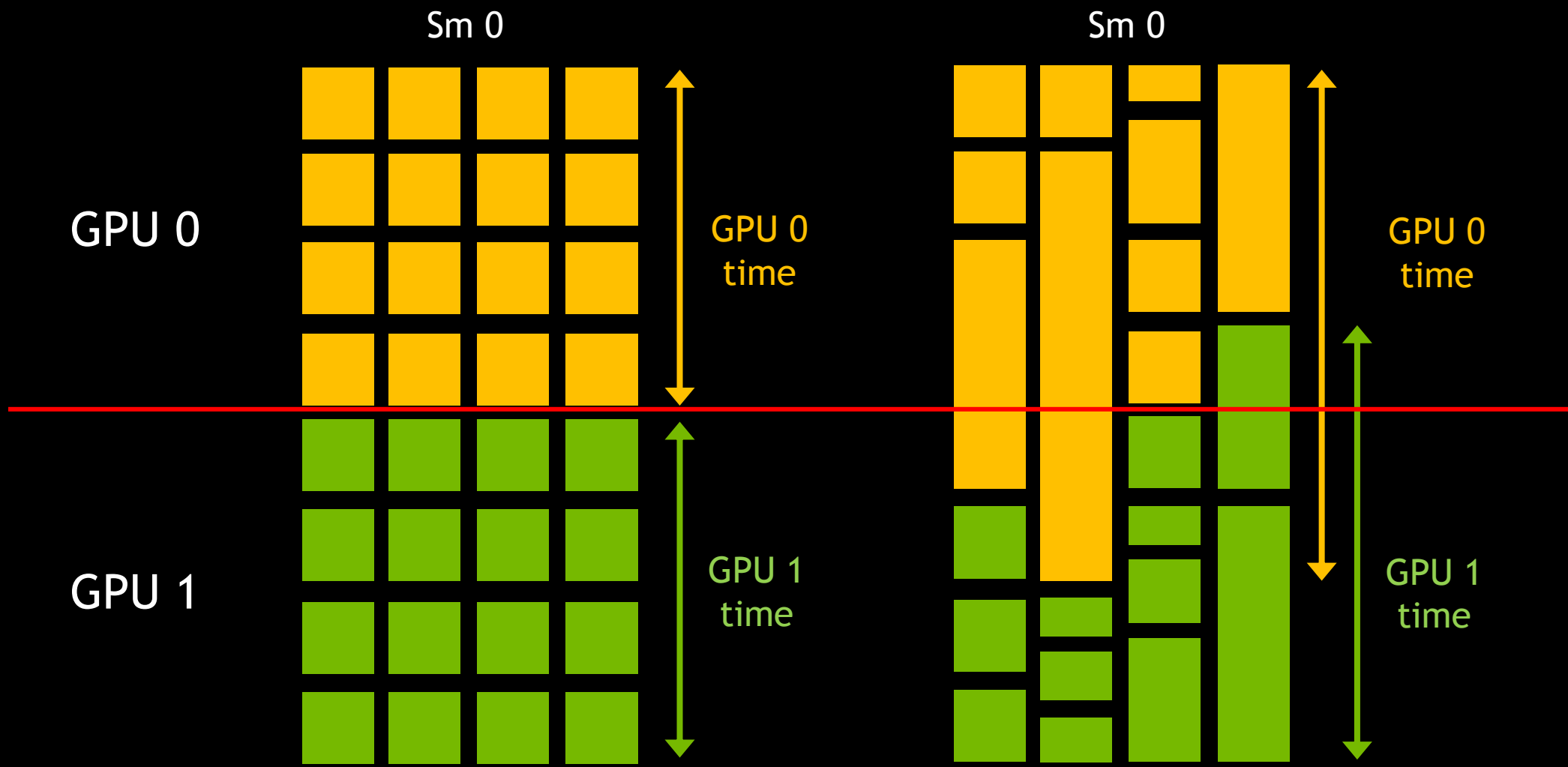
Tail effect



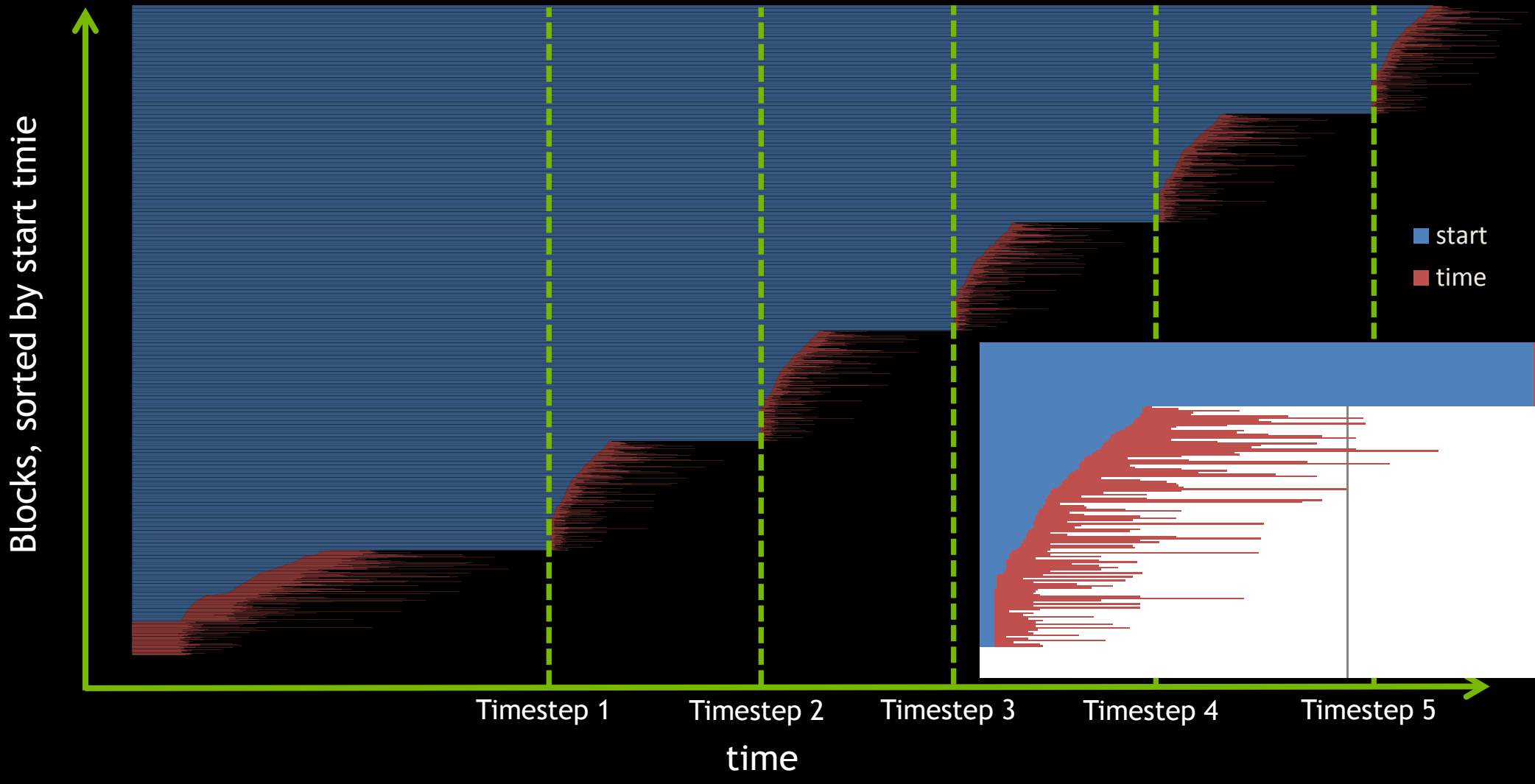


Tail effect

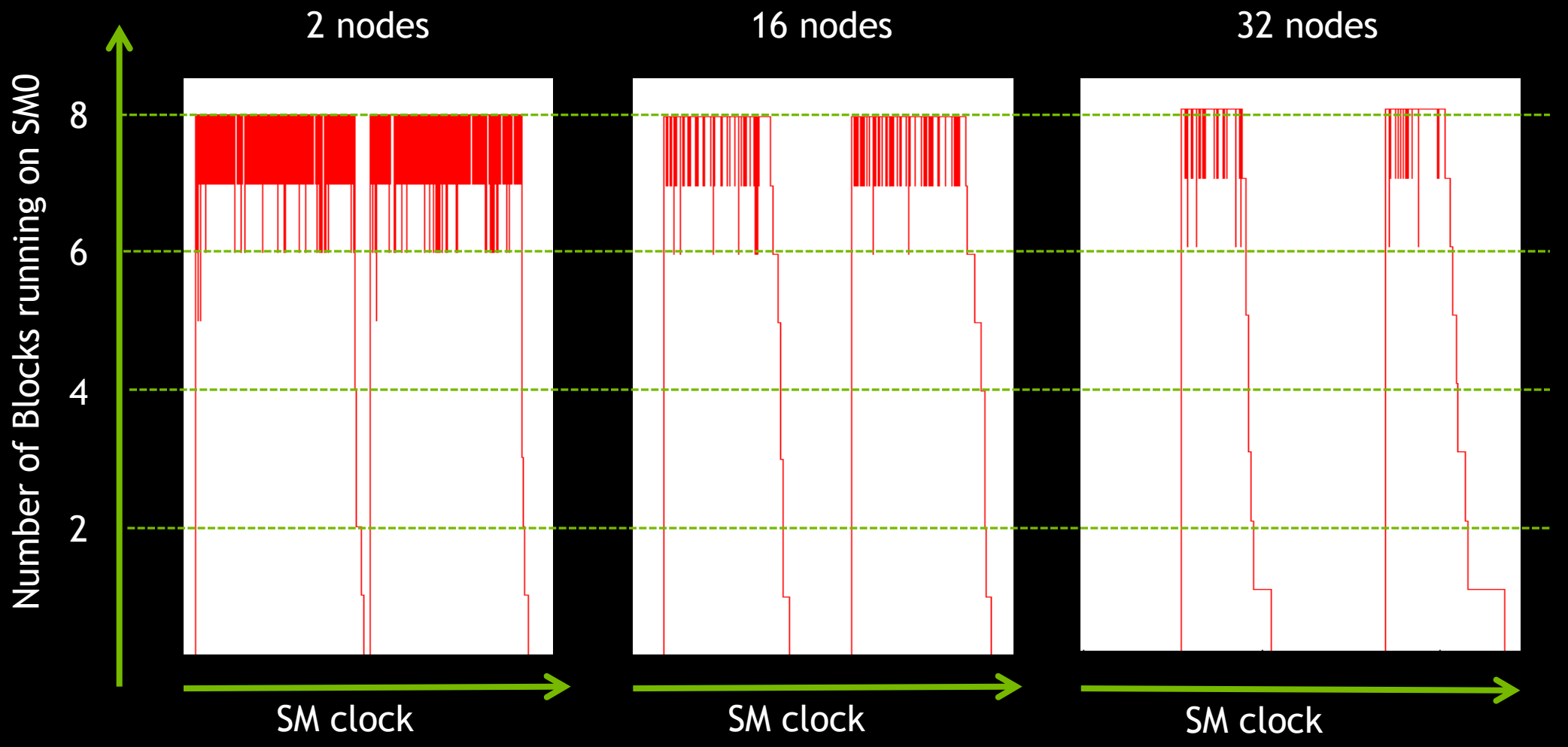


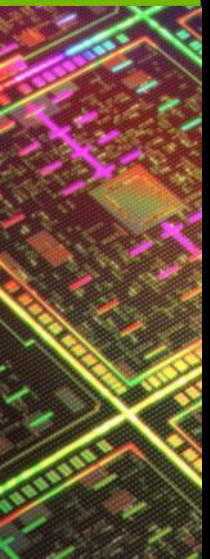


Block timing, Apoa1 at 32 nodes



Block timing



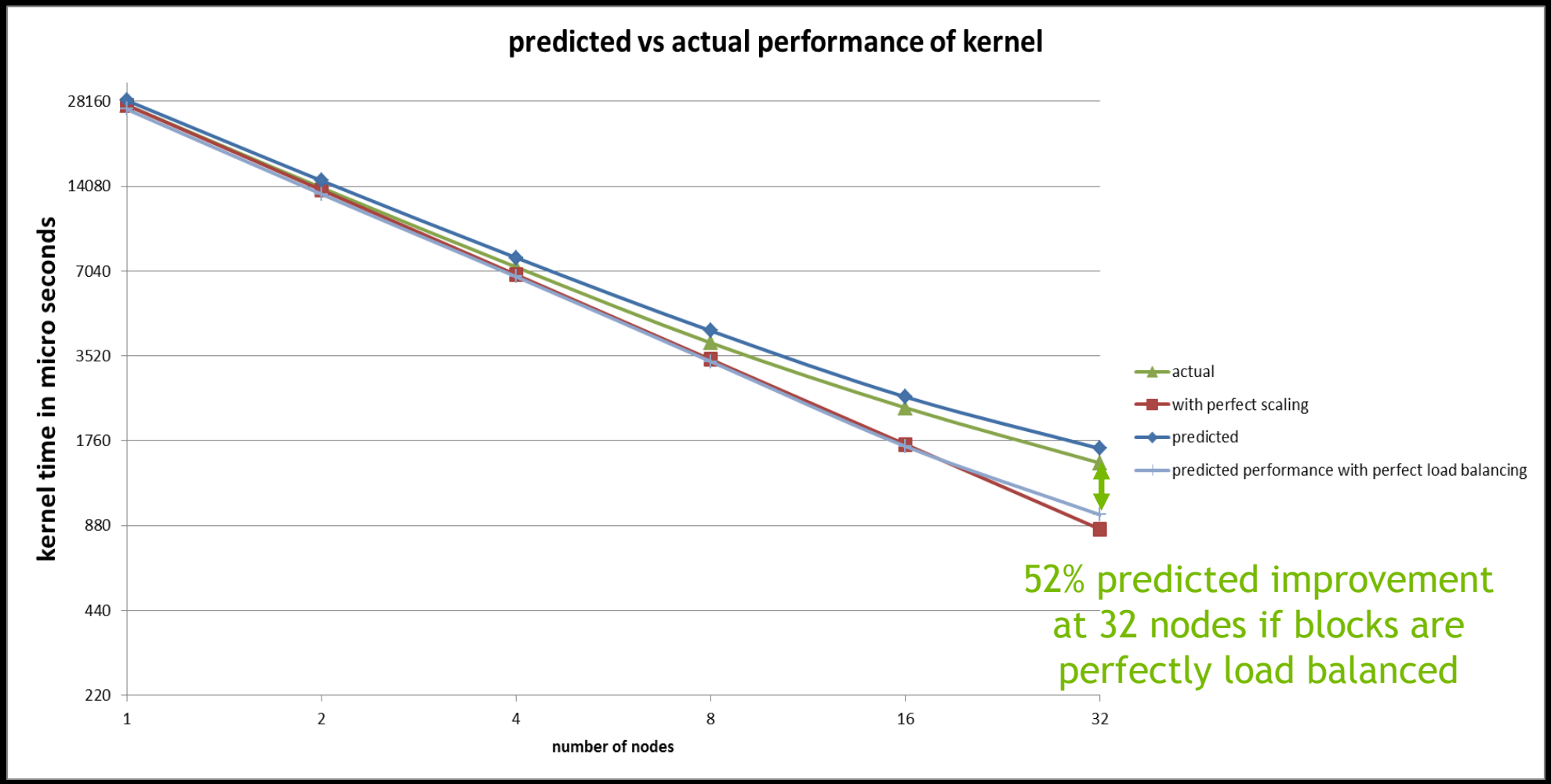


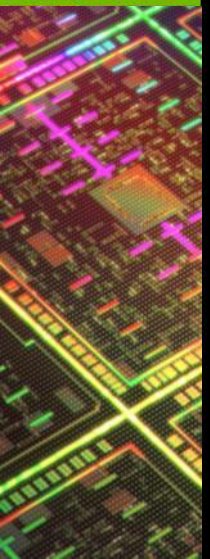
Performance Model

- Can build a performance model to estimate improvement in kernel time as a result of perfect load balancing

- Distribute blocks to SMs in round robin fashion
  - Inside each block have n queues, n = number of concurrent blocks
- Model actual kernel behavior by sampling from measured block runtimes
- Model kernel behavior with perfect load balancing by setting all block runtimes to average value

Estimates from Performance model

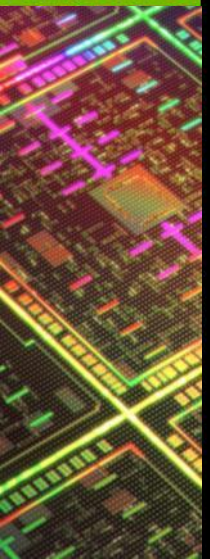




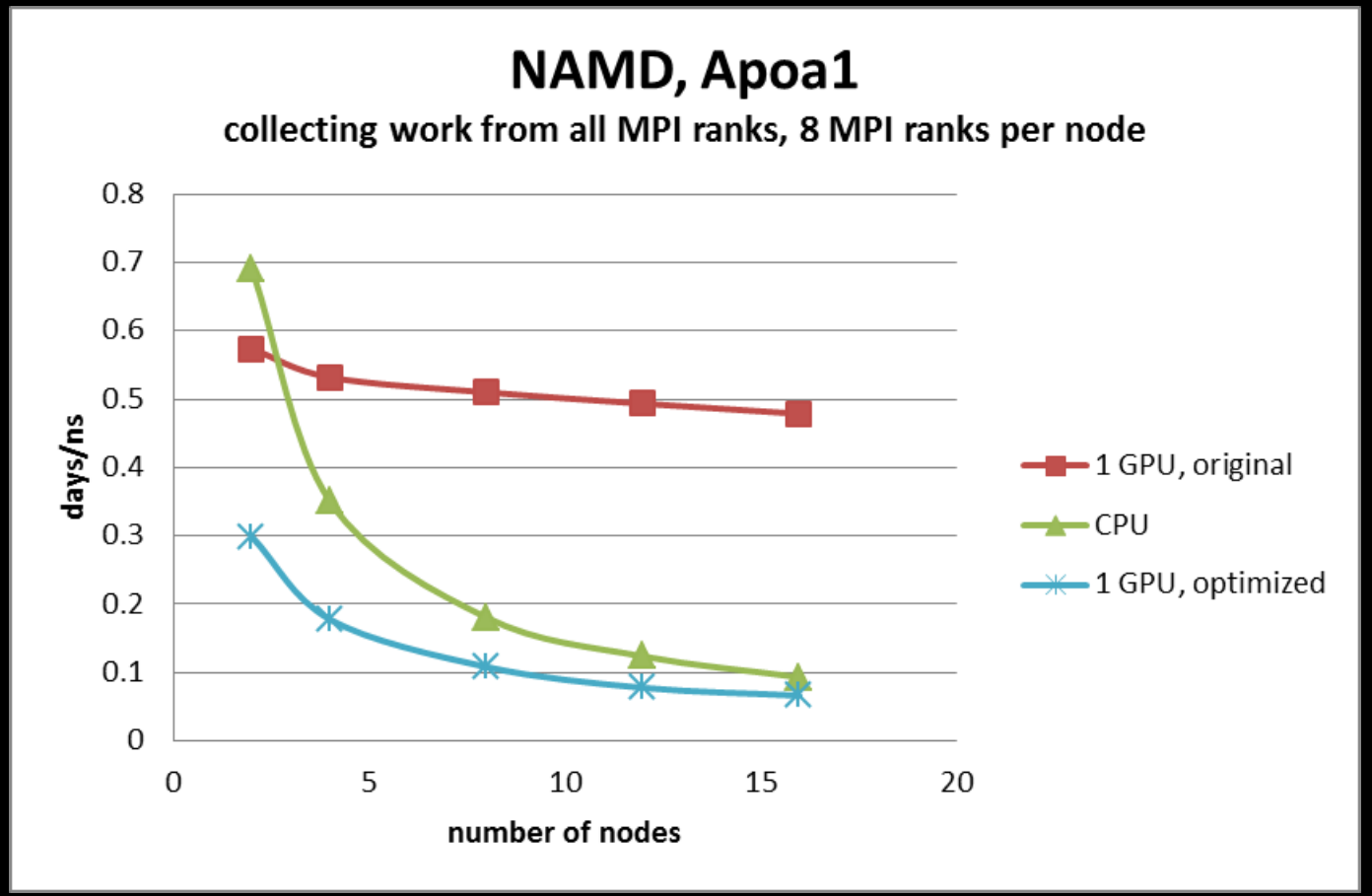
Managing the GPU

Collecting work from multiple cores

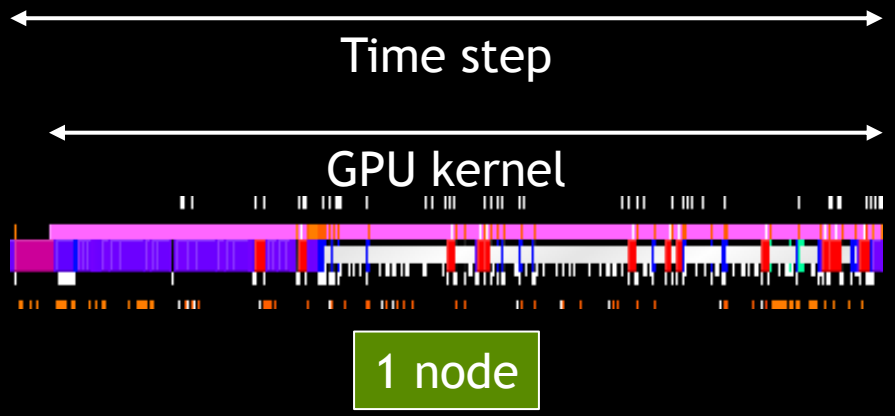
- Potential performance issues if running multiple MPI ranks with each rank submitting work to the GPU
  - each rank has to submit to its own GPU context and kernels from different contexts cannot run concurrently
- you can do better by
  - Using threads instead of MPI ranks
    - This way all threads can submit to the same context
  - Use PROXY
    - Introduced in CUDA 5.0
  - Collect work from multiple MPI ranks to be submitted by one context

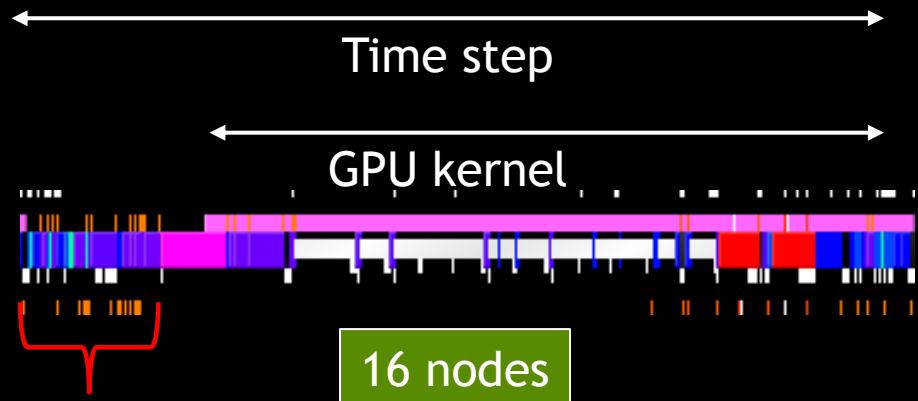


NAMD - collect work from multiple ranks

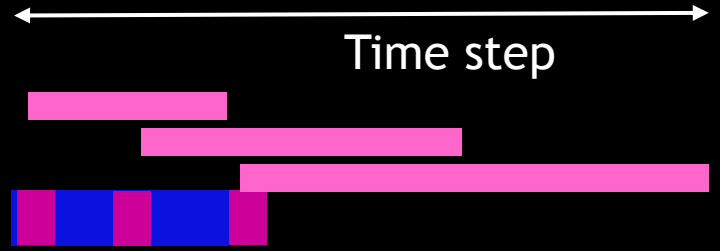


Waiting for all the data to arrive





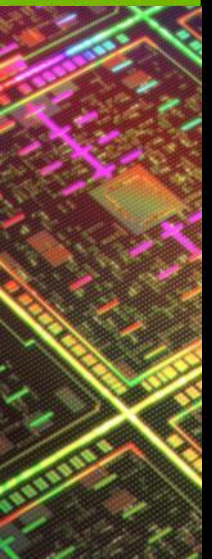
Waiting for data to arrive



Packaging work for the GPU

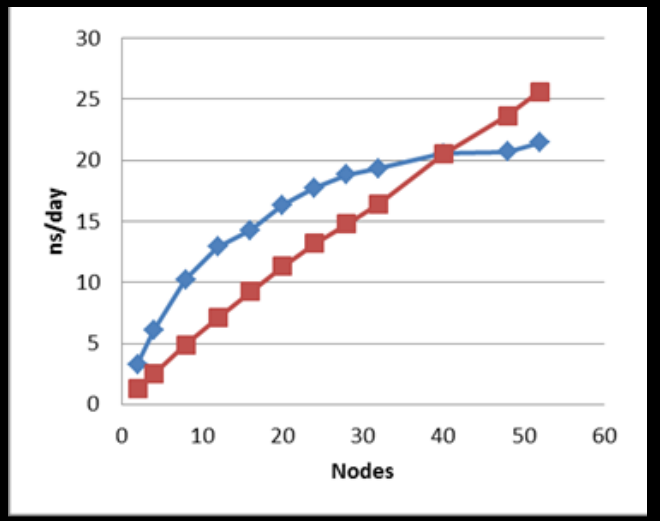
- Is the function that is packaging work for the GPU and processing work off the GPU scaling?
  - Across nodes?
  - Across cores?

- NAMD:
  - pushing enqueueCUDA work to the sender nodes improved both scaling across nodes and across cores

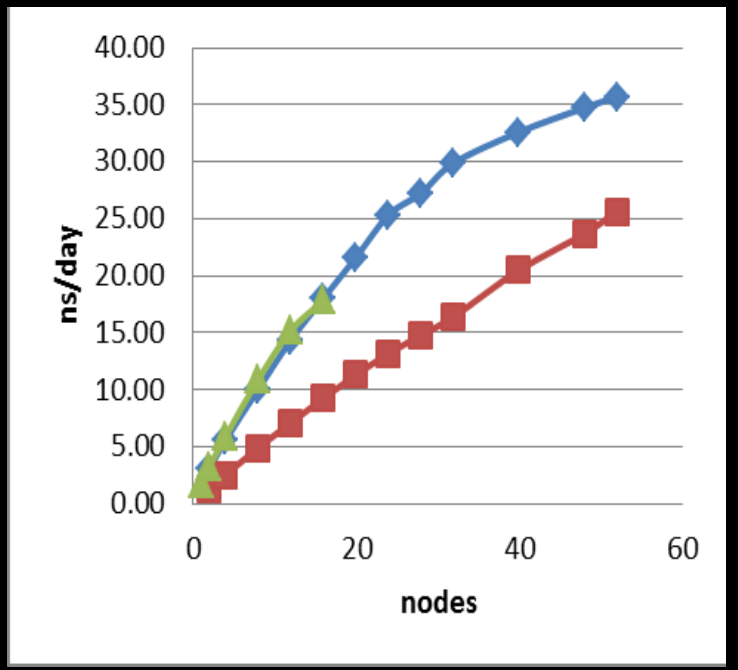


Figuring out kernel completion

- Reducing time to query event signaling kernel completion
  - 10% improvement at 16 nodes, 57% improvement at 52 nodes for Apoa1



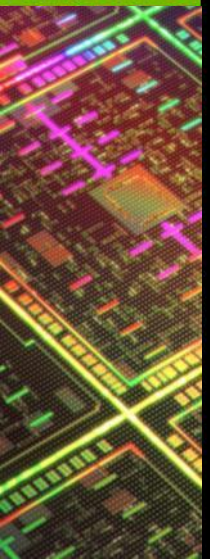
before



after

GPU

CPU

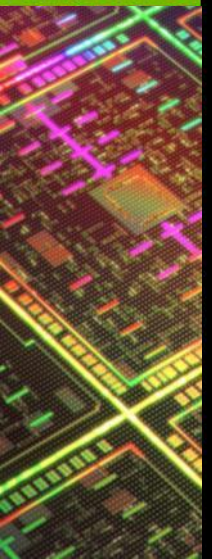


Communication Issues

Communication Issues

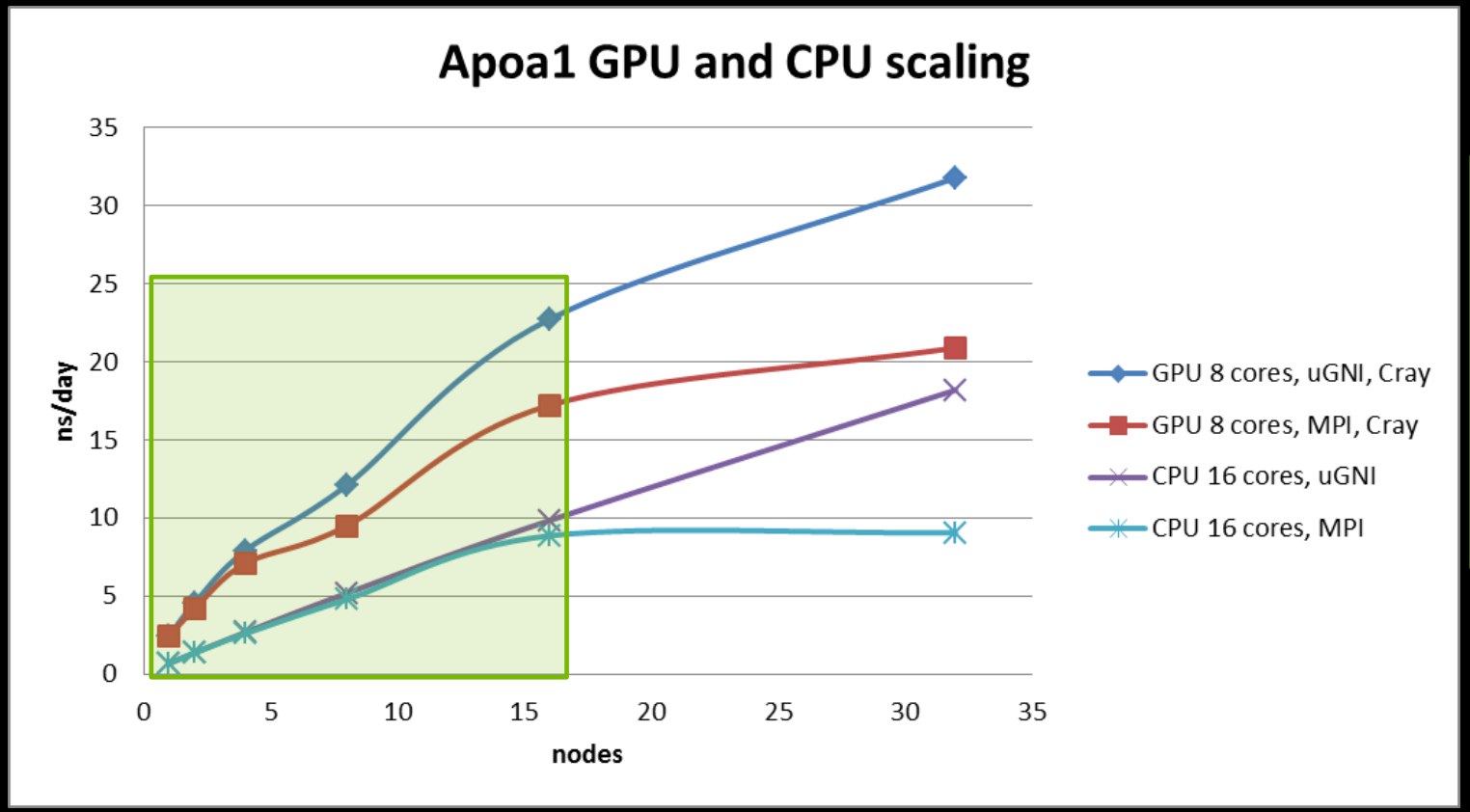
- If all your CPU functions and GPU kernels are scaling well but the overall application is not, communication could be a bottleneck

- Communication issues are more critical to address in the GPU path - the GPU is speeding up the computation so communication can more easily become the bottleneck



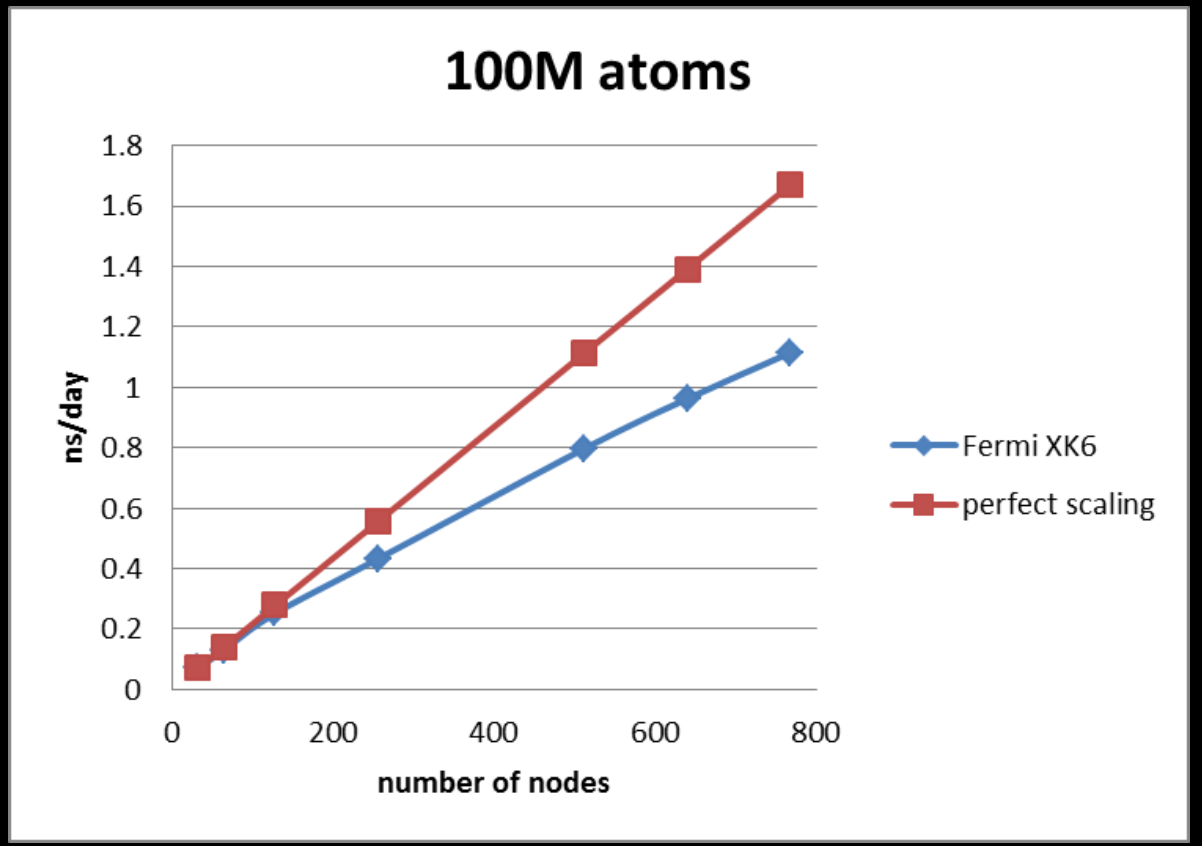
Improving the communication performance

- Using native UGNI messaging rather than generic MPI on a Cray cluster



Improving the communication performance helps improve the scaling performance of the GPU path more than the CPU path

NAMD scaling of 100 M atoms



- Performance results generated on Titan at Oak Ridge National Labs

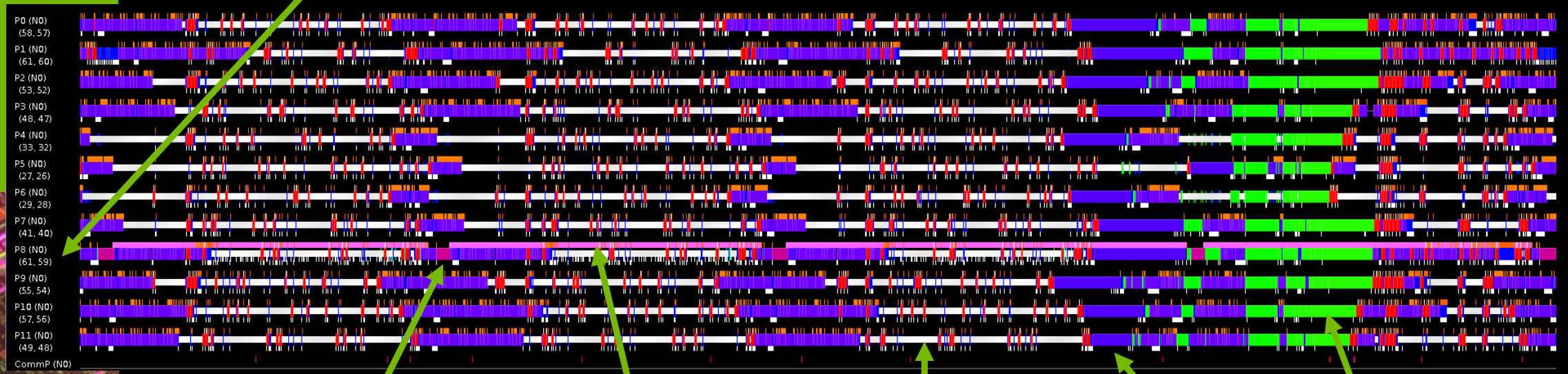
Analysis

	Average time step	non PME step	PME step	sum PME computation	kernel
GPU, 100 stmv, 32 nodes	1.249	1.009	1.833	0.796	0.964
GPU, 100 stmv, 768 nodes	0.085	0.046	0.176	0.034	0.042
strong scaling	0.616	0.905	**0.435**	0.988	0.966

Example NAMD timelines

Showing 11 CPU threads and one communication thread

Communication thread



Magenta = enqueueCUDA(work)

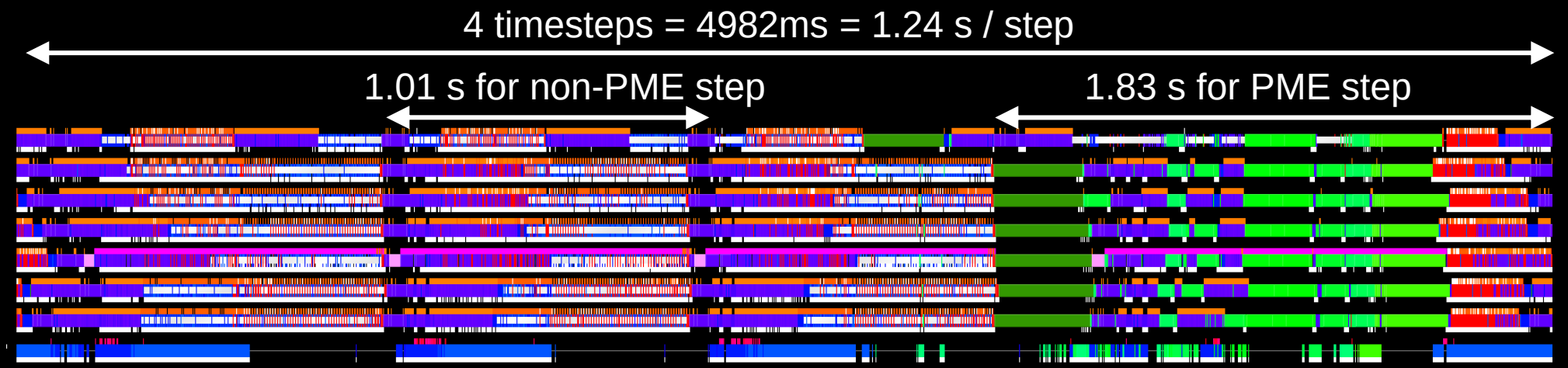
Pink superscript = GPU kernel

White = idle time

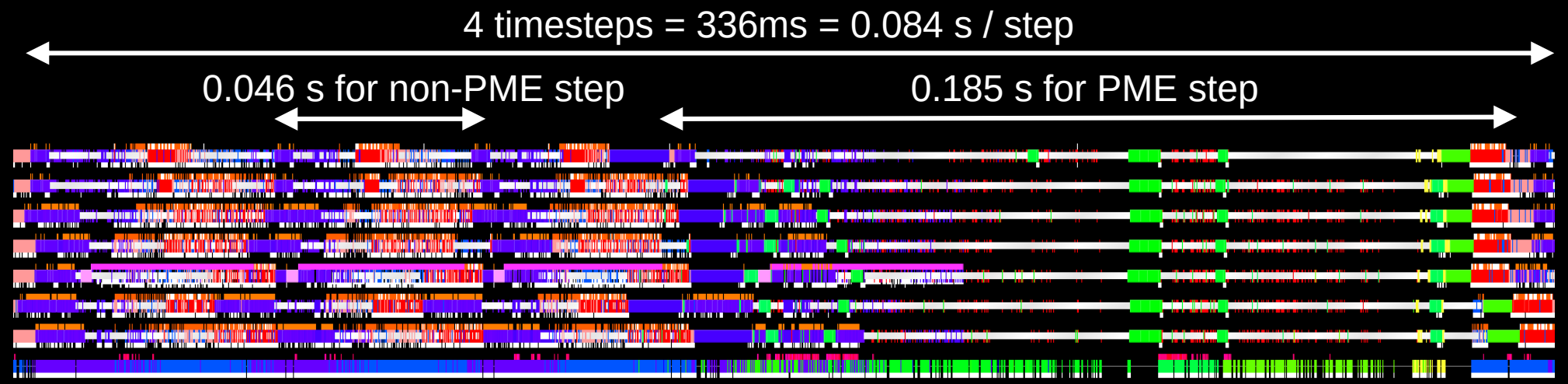
Blue/purple = CPU functions

Green = PME functions

NAMD, Strong Scaling, 100 M atoms, GPU

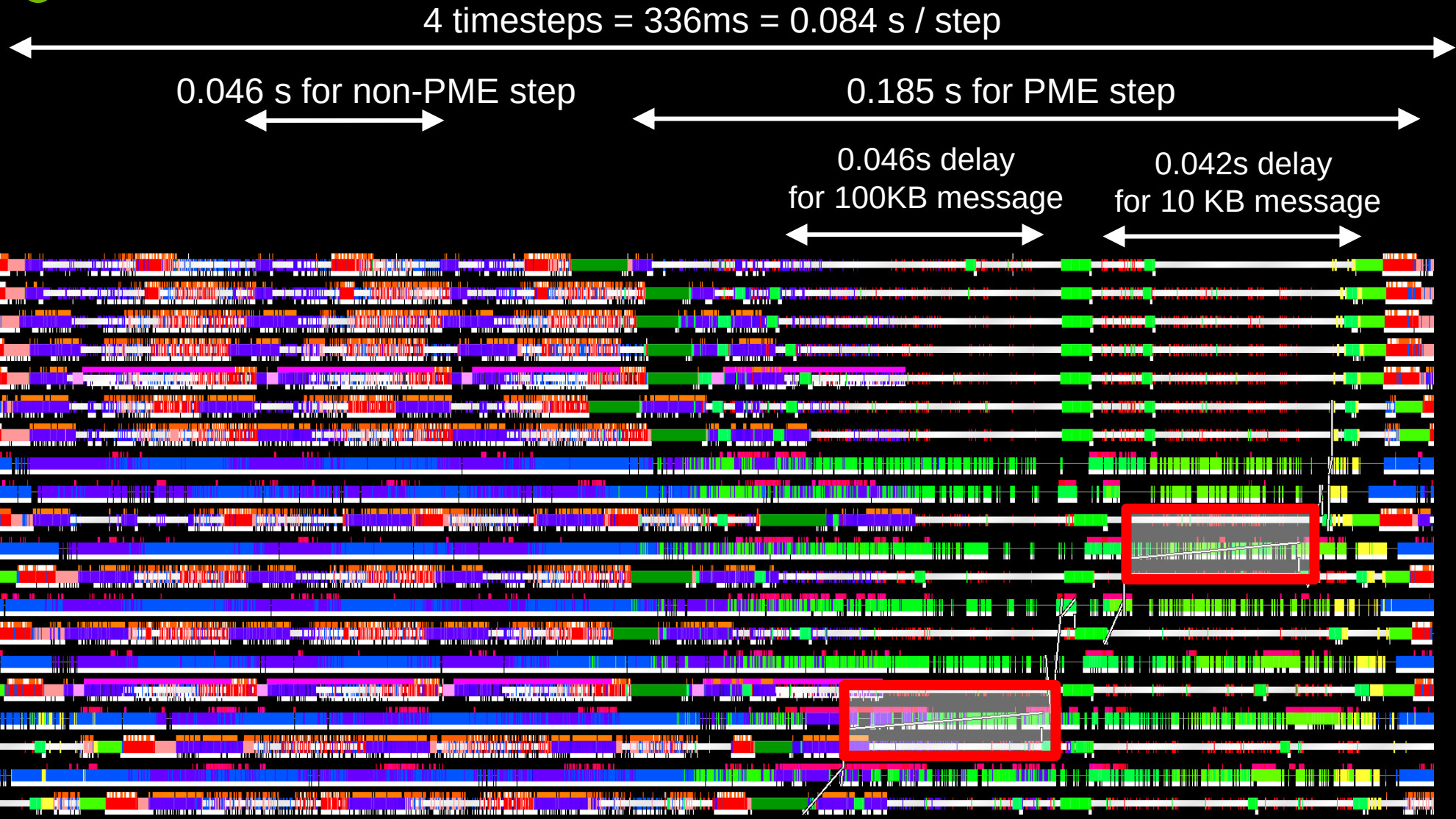


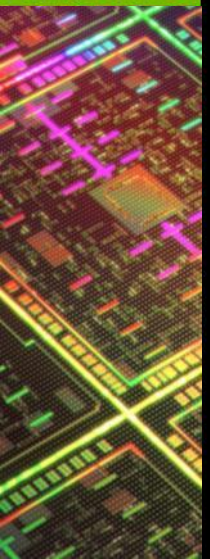
100 stmv
32 nodes



100 stmv
768 nodes

Communication delays in PME - tracing data needed for one ungrid calculation



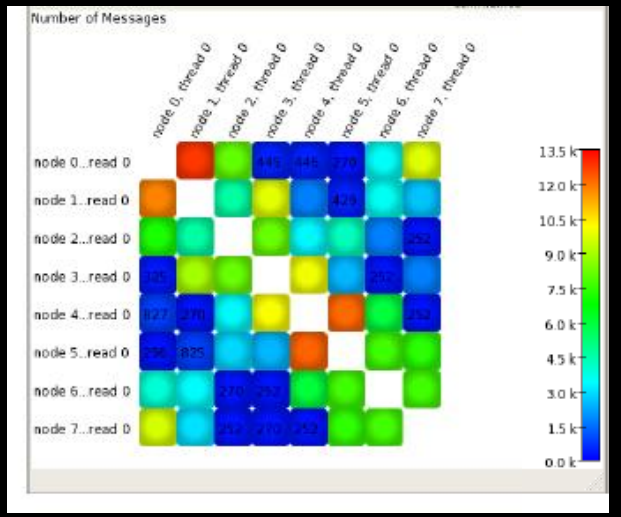


How to improve performance

- Algorithmic changes to reduce communication needed
  - Reduce amount of PME data by doing more non bonded work
- Improve communication

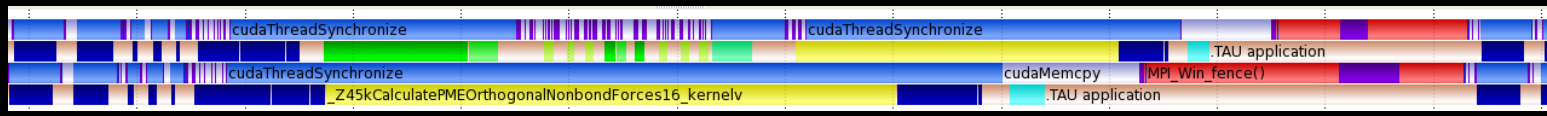
Topology

- Ideal topology?
  - Do you have the ideal topology of nodes in order to maximize bandwidth?
  - Do the CPU only nodes have the same or better topology?
- Topology aware mapping
- Randomization
  - As an experiment try randomizing the ordering of nodes you receive

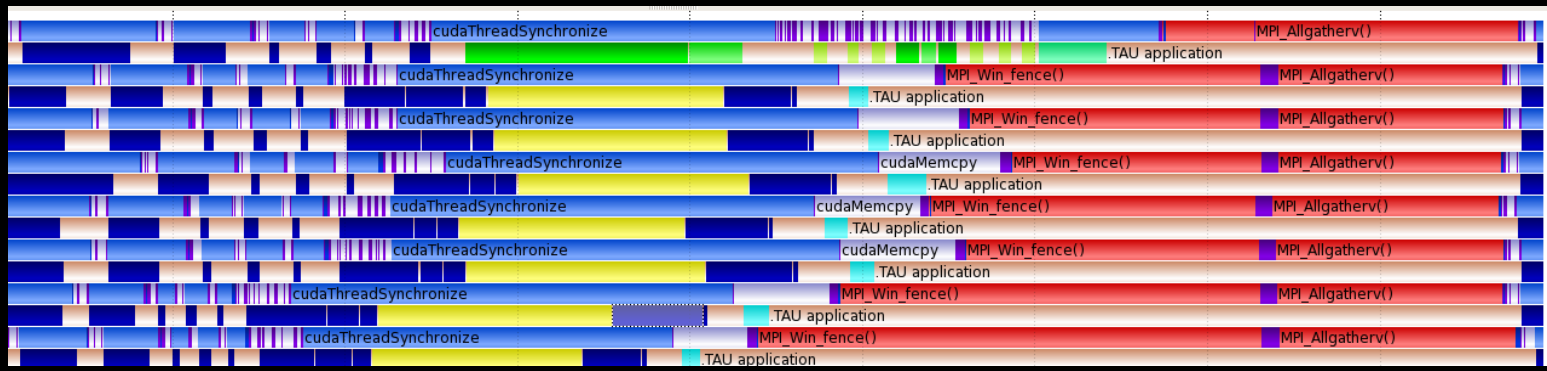


Reducing Communication Delays

2 nodes
MPI_Win_Fence = 338 us
MPI_allgatherv = 183 us



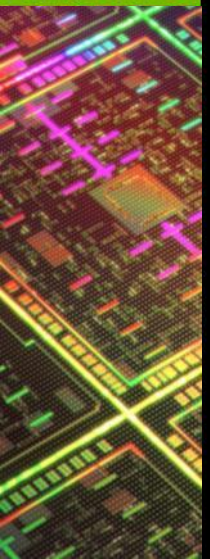
8 nodes
MPI_Win_Fence = 537 us
MPI_allgatherv = 413 us



- Reduce / remove global syncs and pairwise syncs
  - Try to do one sided RMA operations
- If MPI is slow try others like Global Arrays toolkit, UPC, CoArray, ddi
- P2P / Interprocess P2P for GPUs on same node

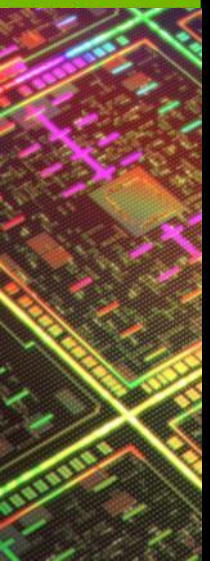
Contention

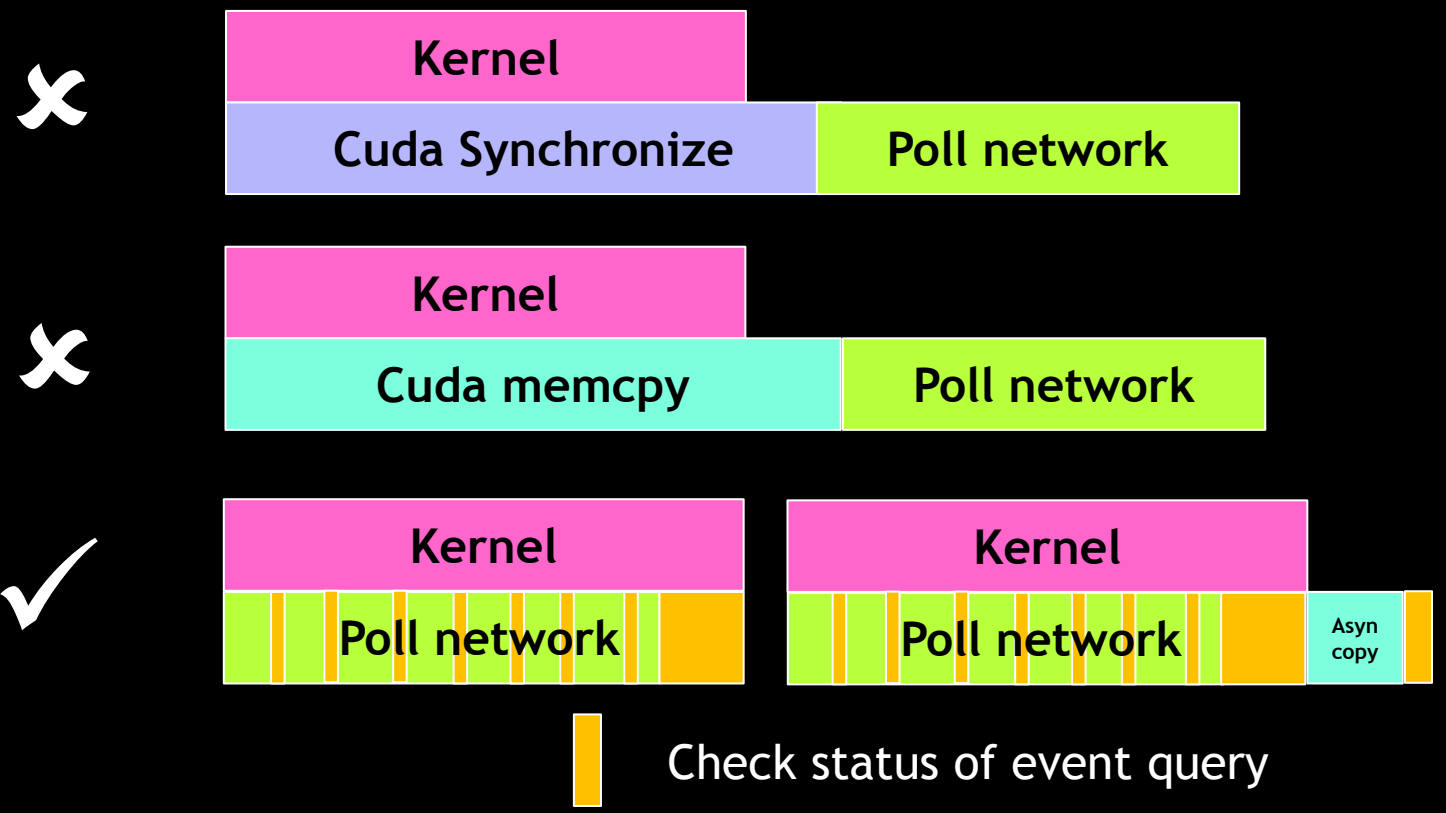
- If you are running into network contention try Rendezvous messaging (sender sends small control message, receiver initiates get)
  - Vs greedy / fire and forget
- Can throttle and control congestion on both the sender and the receiver side

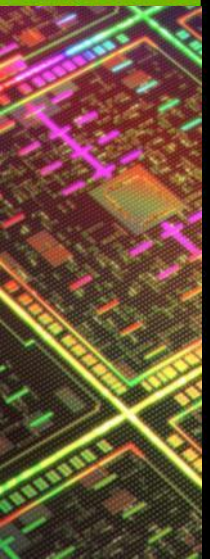


Overlap kernel computation and network activity

If your CPU core is also responsible for the communication

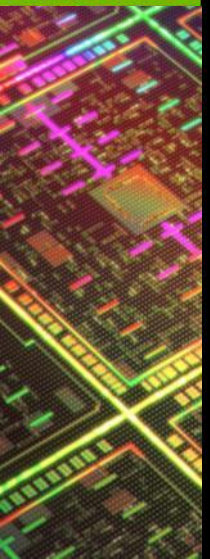






Acknowledgement

- Oak Ridge National Lab
- Scott Le Grand and Ross Walker – AMBER
- Jim Phillips, Sanjay Kale, Yanhua Sun, Gengbin Zhang and Eric Bohm – NAMD
- Sarah Anderson and Ryan Olson - CRAY
- Sky Wu, Nikolai Sakharnykh, Mike Parker and Justin Luitjens – NVIDIA



Questions?

Thank You!