An HPC Perspective for Femtoscale Imaging of Nuclei using Exascale Platforms

About me

- I am a PhD student at Virginia Tech
  - My primary field is HPC
  - I have no background in Physics, much less QP, HEP, NP
- My work is primarily developing a performant theory module
  - fitpack_cpp – Low latency, differentiable theory* module, currently in production
  - GPU version of this theory
- I have also played around with
  - A PDF level experimental module
  - Using generative models for parameterization of distributions

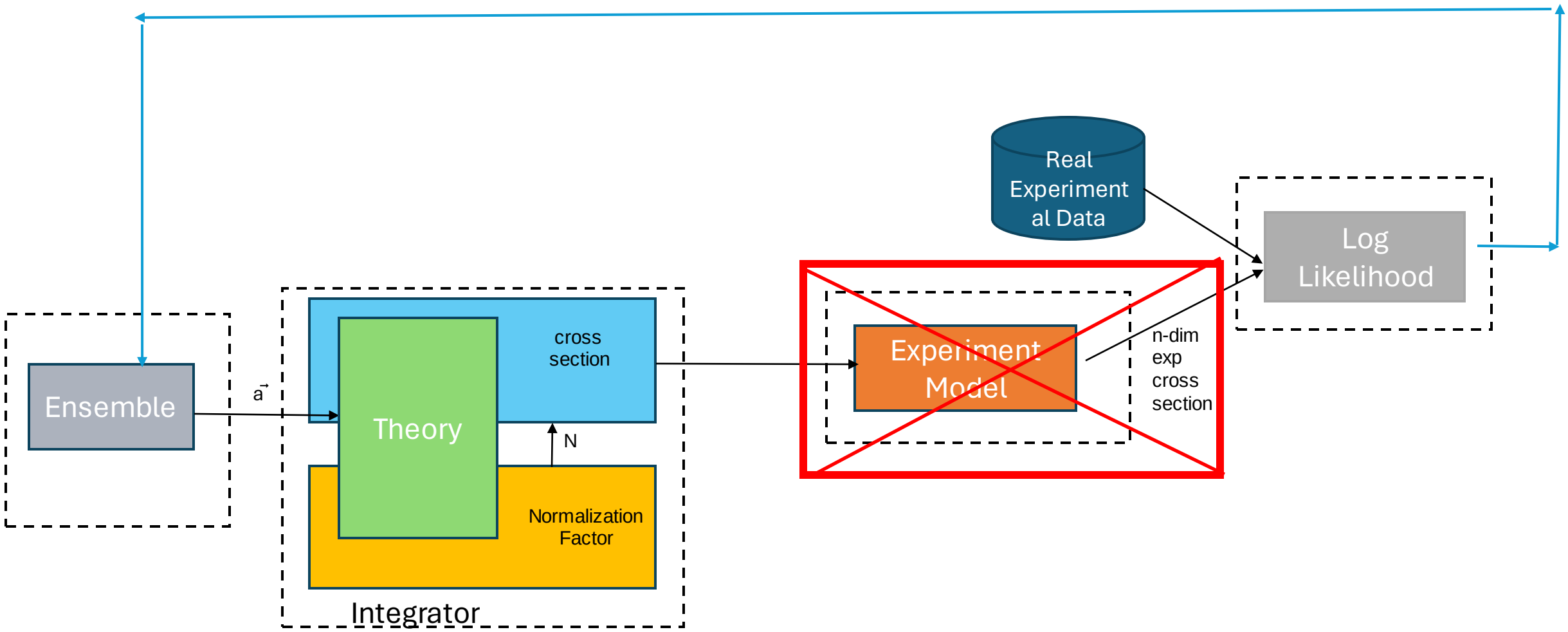
Disclaimer

- I have no background in Physics.
- I present a lot of "napkin math."
  - Doing exhaustive runs for everything would empty our supercomputing allocation.

An HPC Perspective for Femtoscale Imaging of Nuclei using Exascale Platforms

IDIS with 1-D QCFs in Mellin Space

("STATS-2")

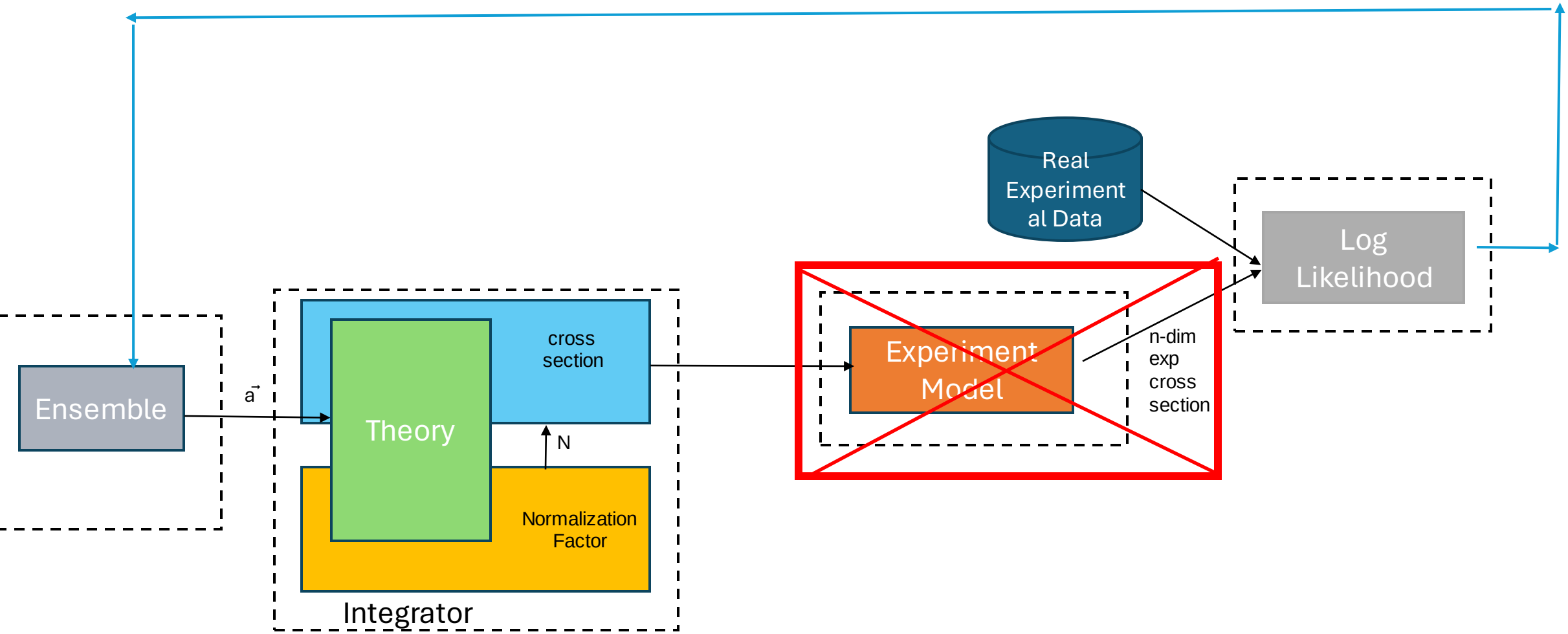


K Bootstraps

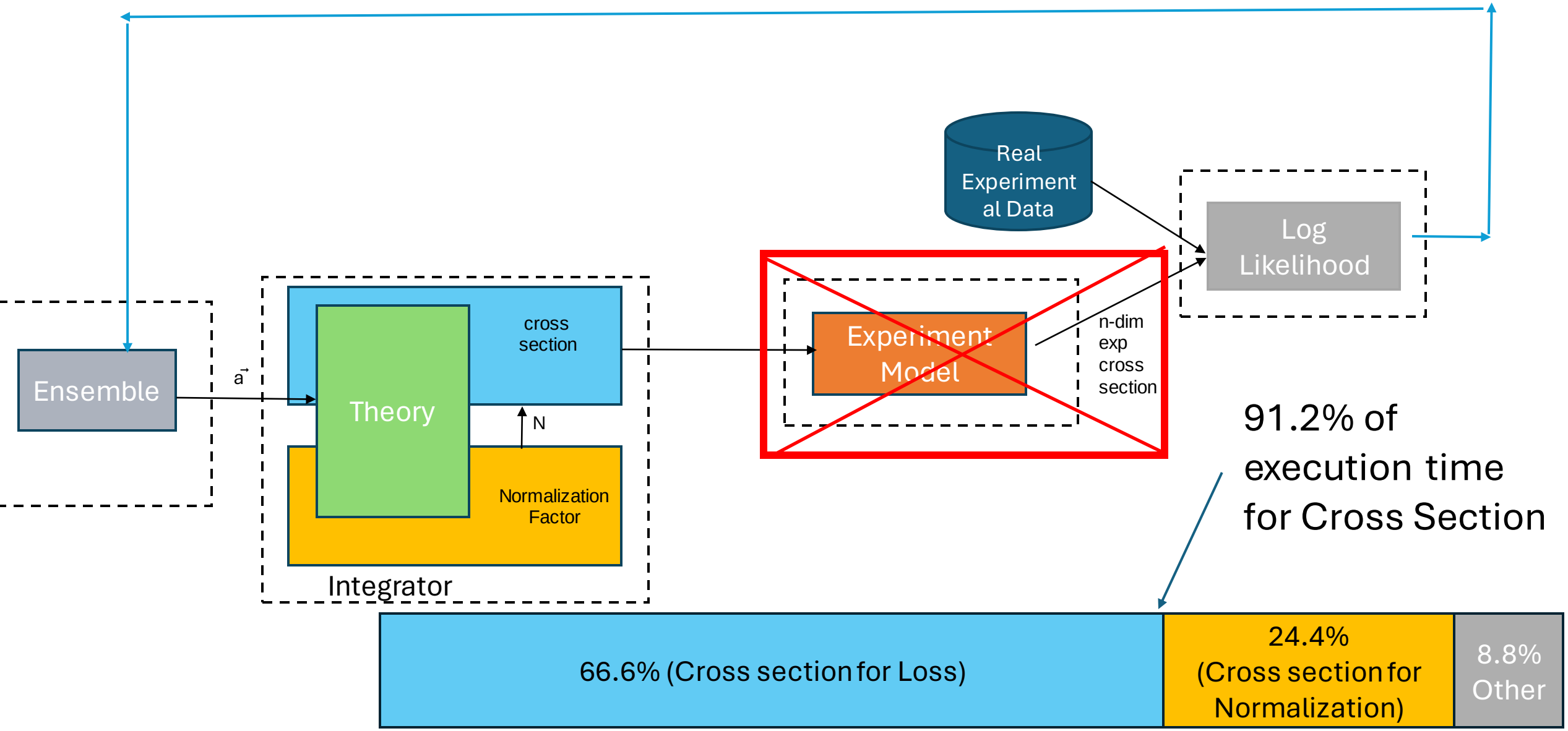
VT-Argonne Computational Experiments
("Stats-2")

- PDF-Event Loss
- 5,000 experimental events for p and n targets
- Computational Run on Virginia Tech Computing Clusters:
  - Nodes: 2
    - Tinkercliffs@VT CPU: AMD 7702x2 (128 Cores)
    - Infer@VT GPU: Nvidia V100
  - Bootstraps: 2,560
  - Theory: fitpack_cpp - Low latency, differentiable theory* module, currently in production

Profiling & Projecting Performance of IDIS with 1-D QCFs in Mellin Space



Profiling & Projecting Performance of IDIS with 1-D QCFs in Mellin Space



Projected Execution times

Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13

Projected Execution times

Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13

Projected Execution times

Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13
C++ CPU (fitpack_cpp)		
Pytorch Vectorized (GPU)		

Projected Execution times

Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13
C++ CPU (fitpack_cpp)	15.83	
PyTorch Vectorized (GPU)	24.49	

Pytorch GPU 54% faster than optimized CPU version!!!

Projected Execution times

Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13
C++ CPU (fitpack_cpp)	15.83	**17.70**
PyTorch Vectorized (GPU)	24.49	*11.99*

Pytorch GPU 54% faster than optimized CPU version!!!

Projected Execution times

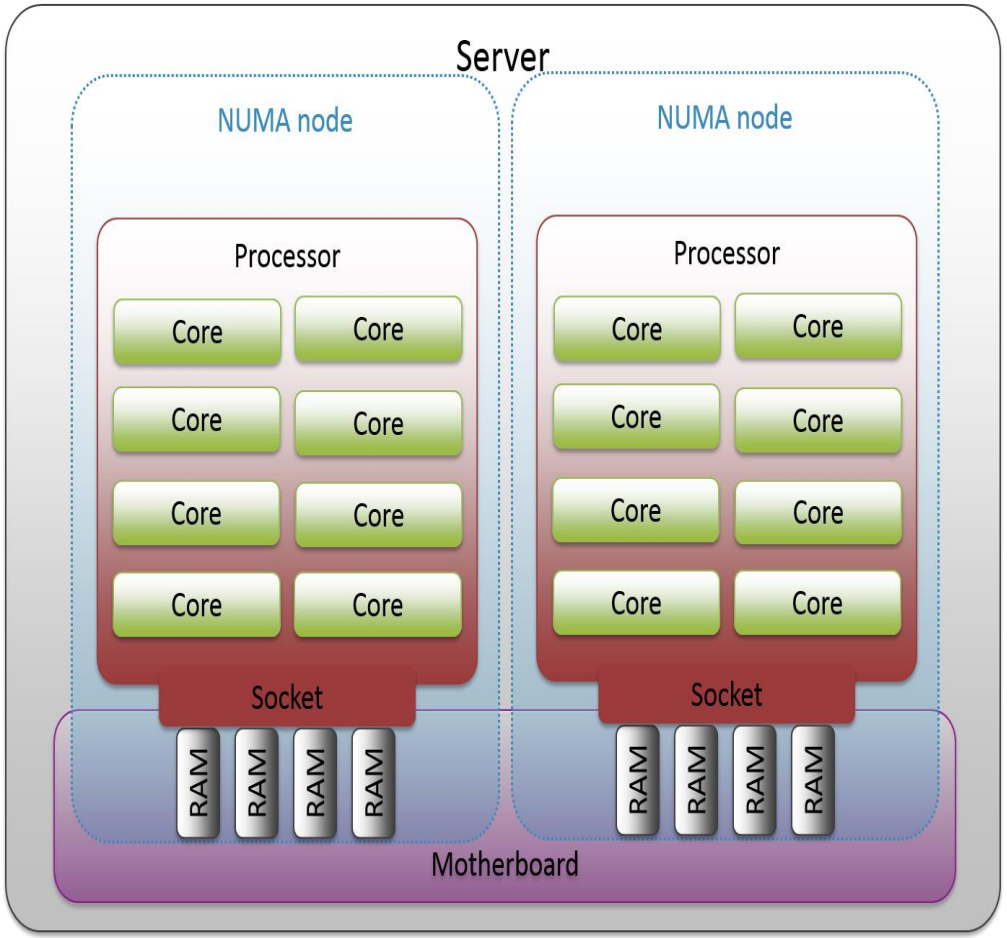
Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13
	15.83	**17.70**
	24.49	*11.99*

Pytorch GPU 54% faster than optimized CPU version!!!

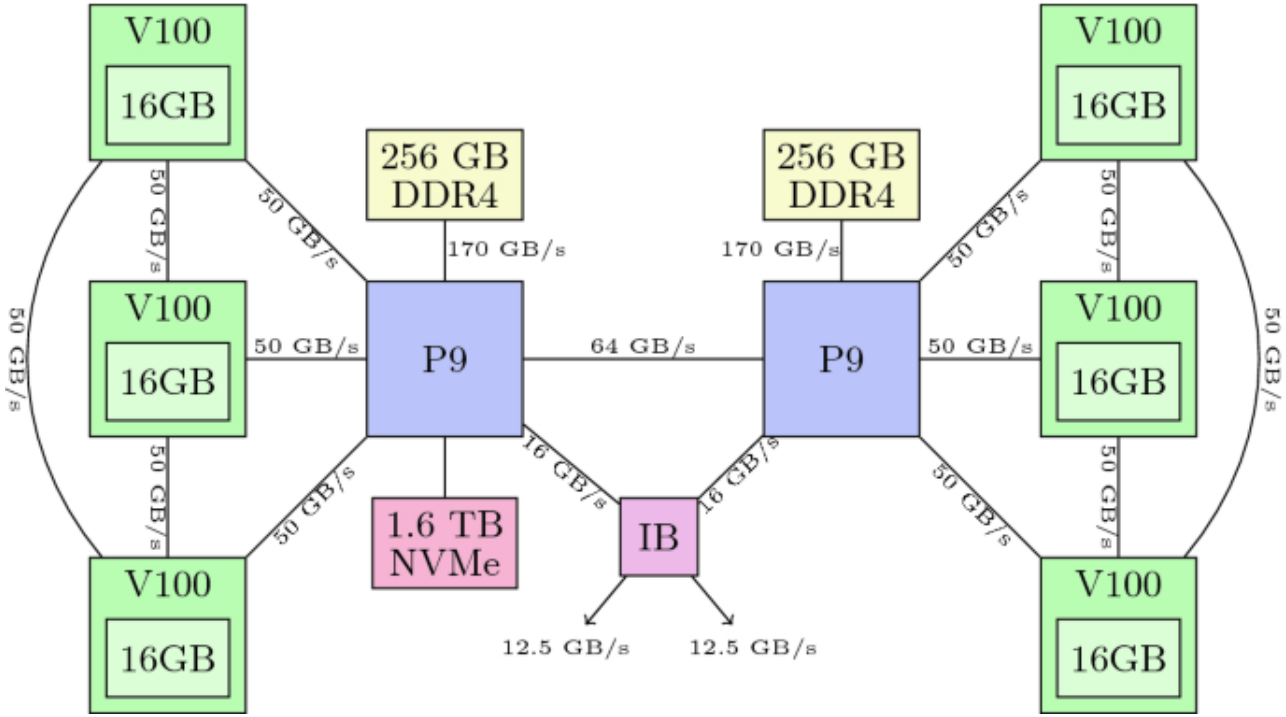
Projected Execution times

Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13
C++ CPU (fitpack_cpp)	15.83	**17.70**
PyTorch Vectorized (GPU)	24.49	*11.99*

HPC Computational Resources



CPU Node

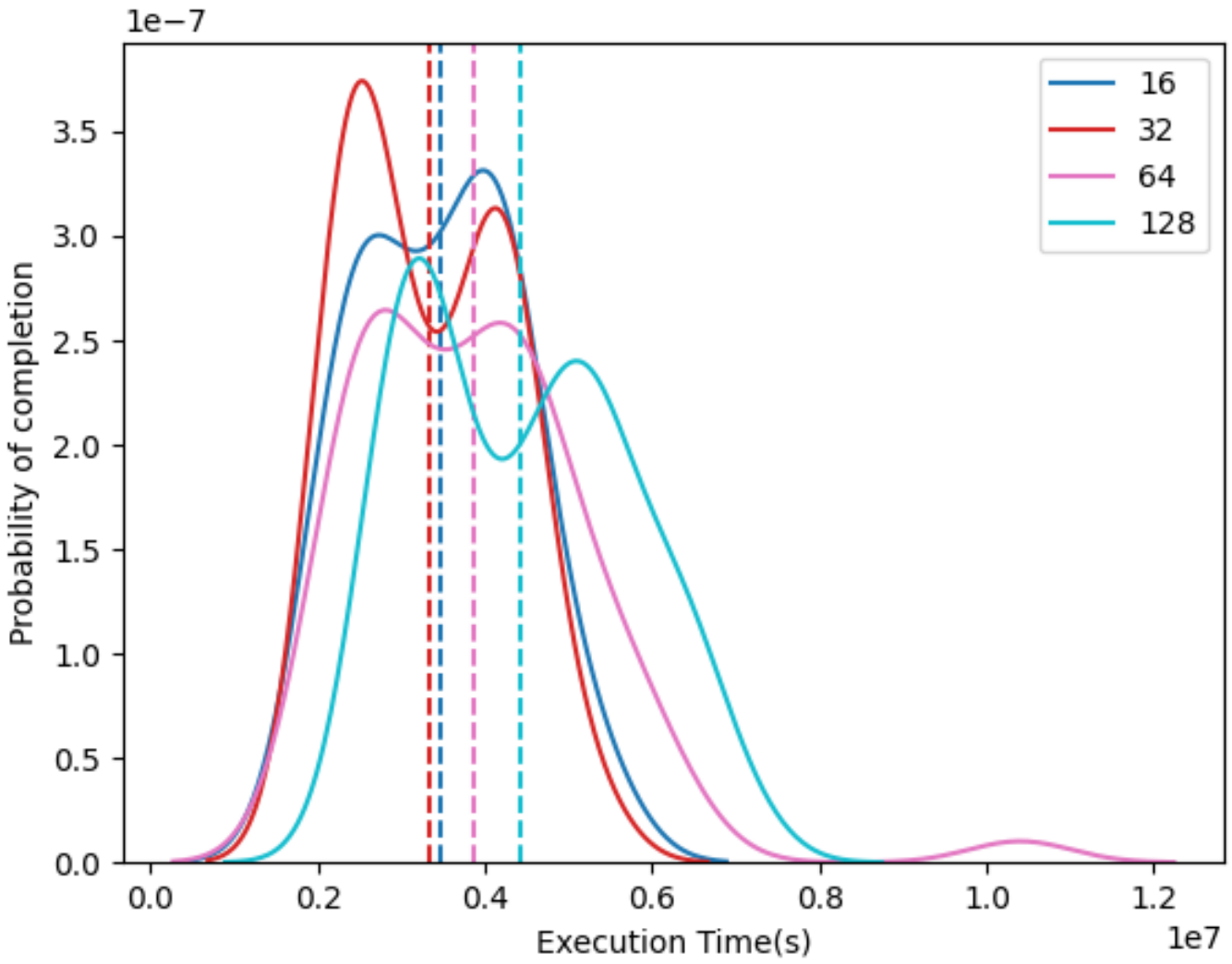


GPU Node

HPC Computational Resources

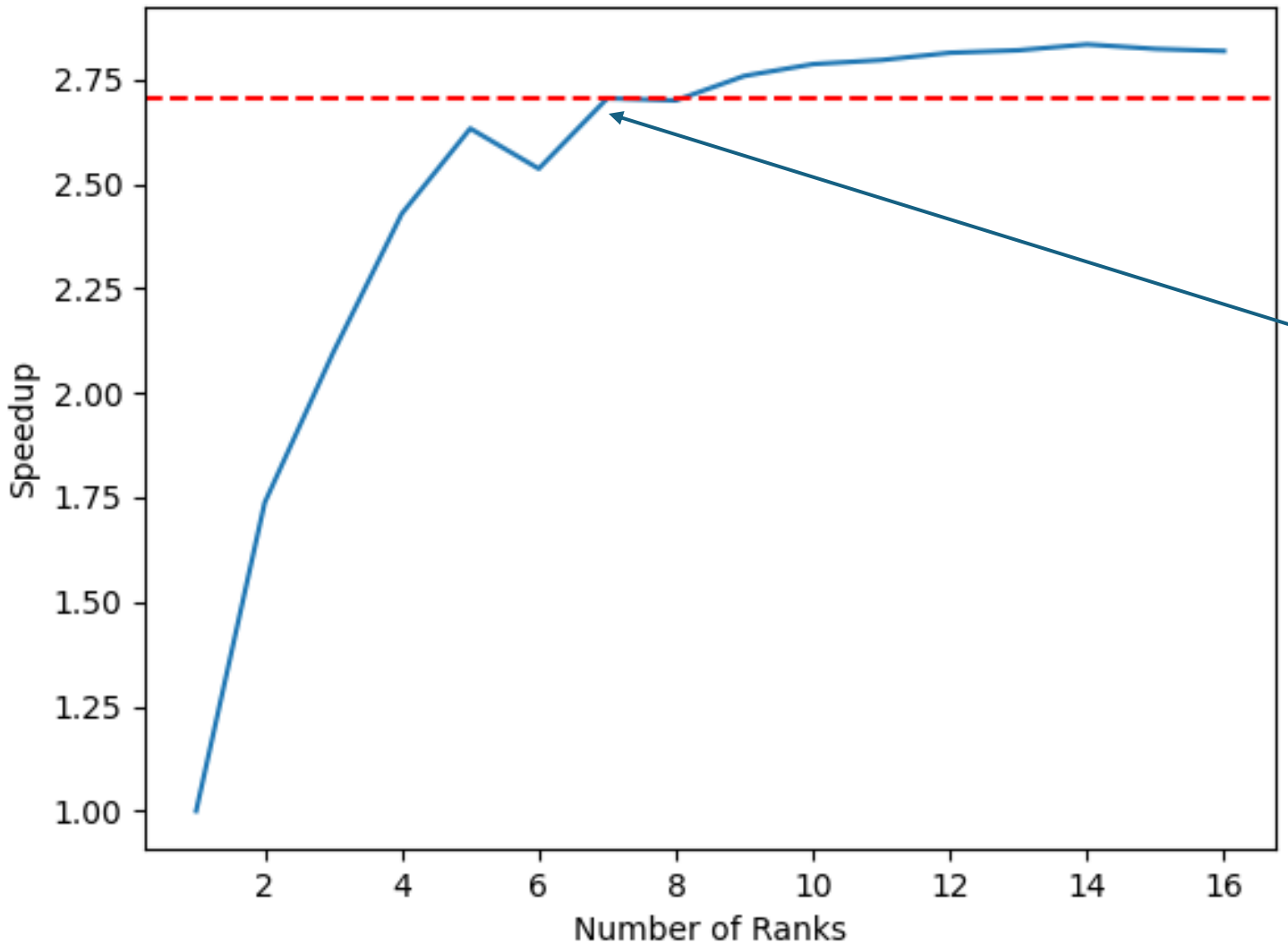
- We can execute 128 different ensembles per node for a CPU oriented implementation.
- If we assume we have 6 GPUs per node like DOE ORNL's Summit supercomputer, we can find the best count for number of ensembles per GPU.

Parallel Efficiency of Stats-2 (C++ CPU)



Number of Threads	Parallel Speedup
16	16
32	33.28
64	56.96
128	99.84

Parallel Efficiency of Stats-2 (Pytorch Vectorized GPU)



Using 7 threads per GPU gives 2.75X speedup

HPC Computational Resources

- Even with overhead due to process-level parallelization, we still effectively perform 100 units of work on CPUs.
- Multi-GPU Nodes perform 6 units of work per GPU, but 4.16X faster

HPC Computational Resources

- Even with overhead due to process-level parallelization, we still effectively perform 100 units of work on CPUs.
- Multi-GPU Nodes perform 6 units of work per GPU, but 4.16X faster

CPU version is 4 times faster than
PyTorch Vectorized GPU

The Memory Cost of Vectorization
10K evalutations

Library	Total Memory Used	Total Number of allocations
Numpy	122.559KB	53
Numpy Vectorized	748.081MB	82
Pytorch	1.323MB	78
Pytorch Vectorized	893.541MB	307

The Memory Cost of Vectorization
10K evalutations

Library	Total Memory Used	Total Number of allocations
Numpy	122.559KB	53
Numpy Vectorized	748.081MB	82
Pytorch	1.323MB	78
Pytorch Vectorized	893.541MB	307

6000 times more memory

Projected Execution times

Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13
C++ CPU (fitpack_cpp)	15.83	**17.70**
PyTorch Vectorized (GPU)	24.49	*11.99*

~ 66.12 Hours

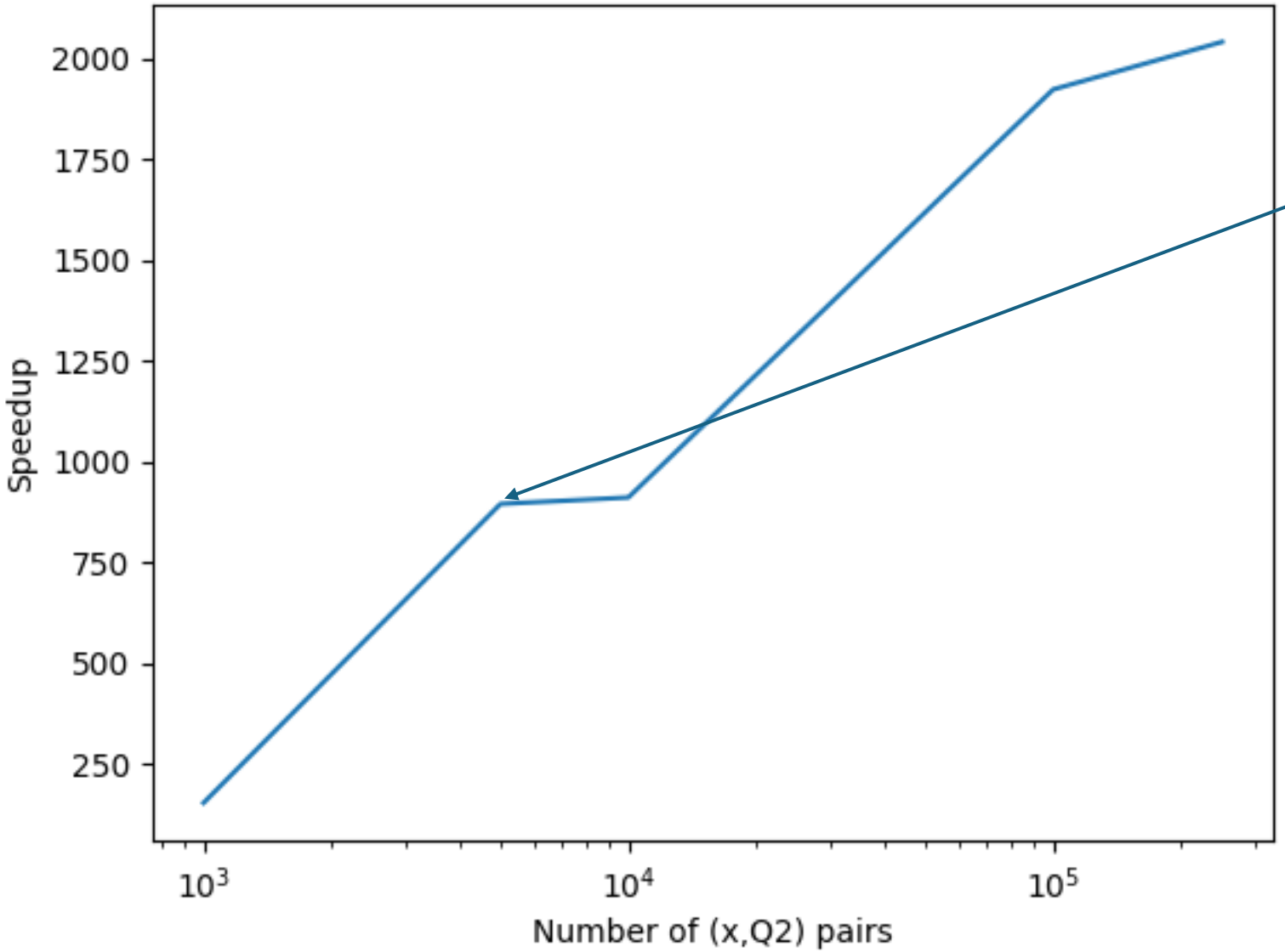
Projected Execution times

Version	Speedup for cross-section calculation	Execution time (Hours)
Numpy	1.00	*257.09*
Numpy vectorized*(~ Pytorch CPU)	7.39	36.13
C++ CPU (fitpack_cpp)	15.83	**17.70**
PyTorch Vectorized (GPU)	24.49	~~*11.99*~~
CUDA GPU (fitpack_cpp)	897.50	*5.25*

Not as high as expected speedup?

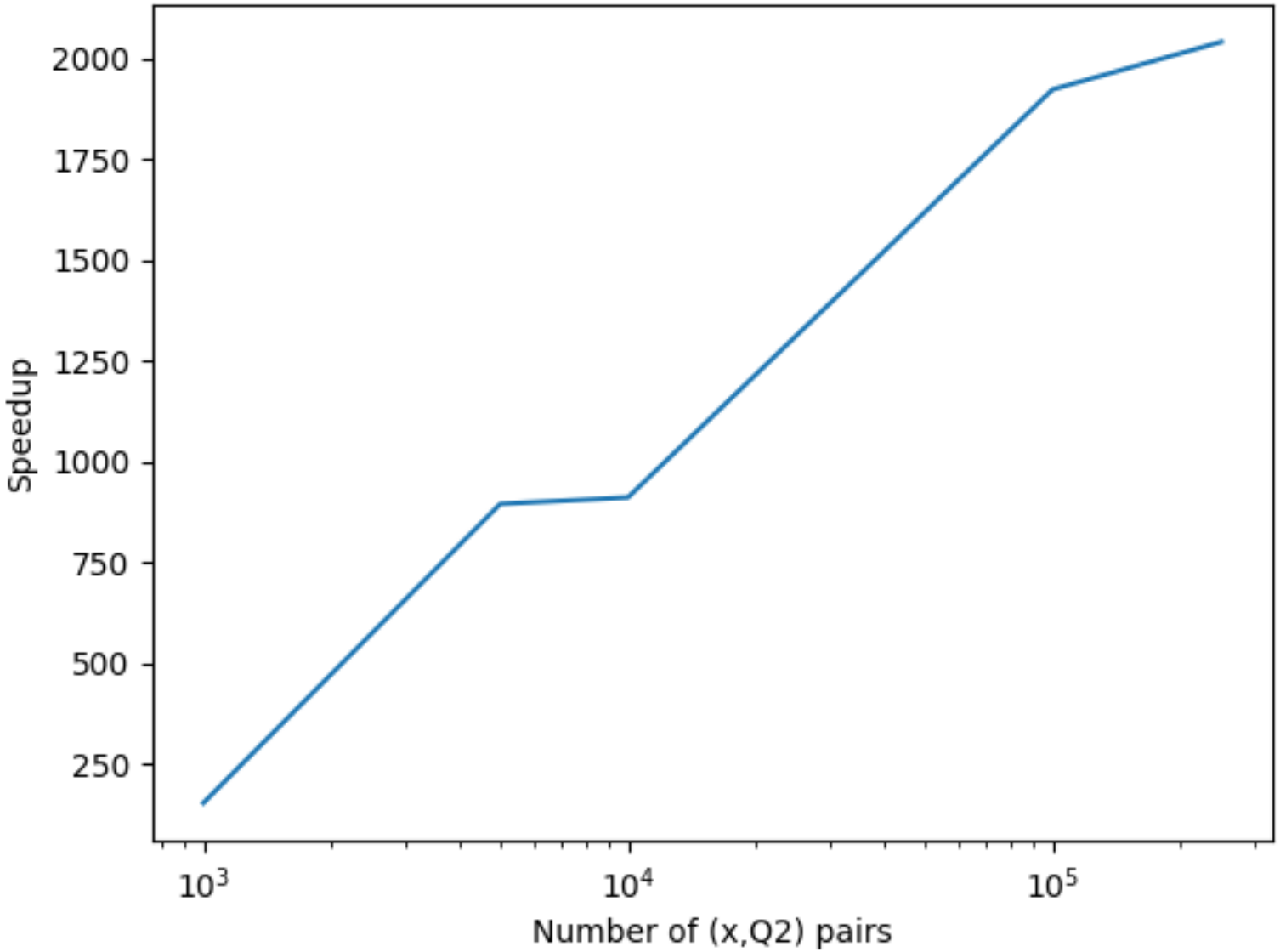
~3.3x faster than CPU

Speedup of GPU Theory



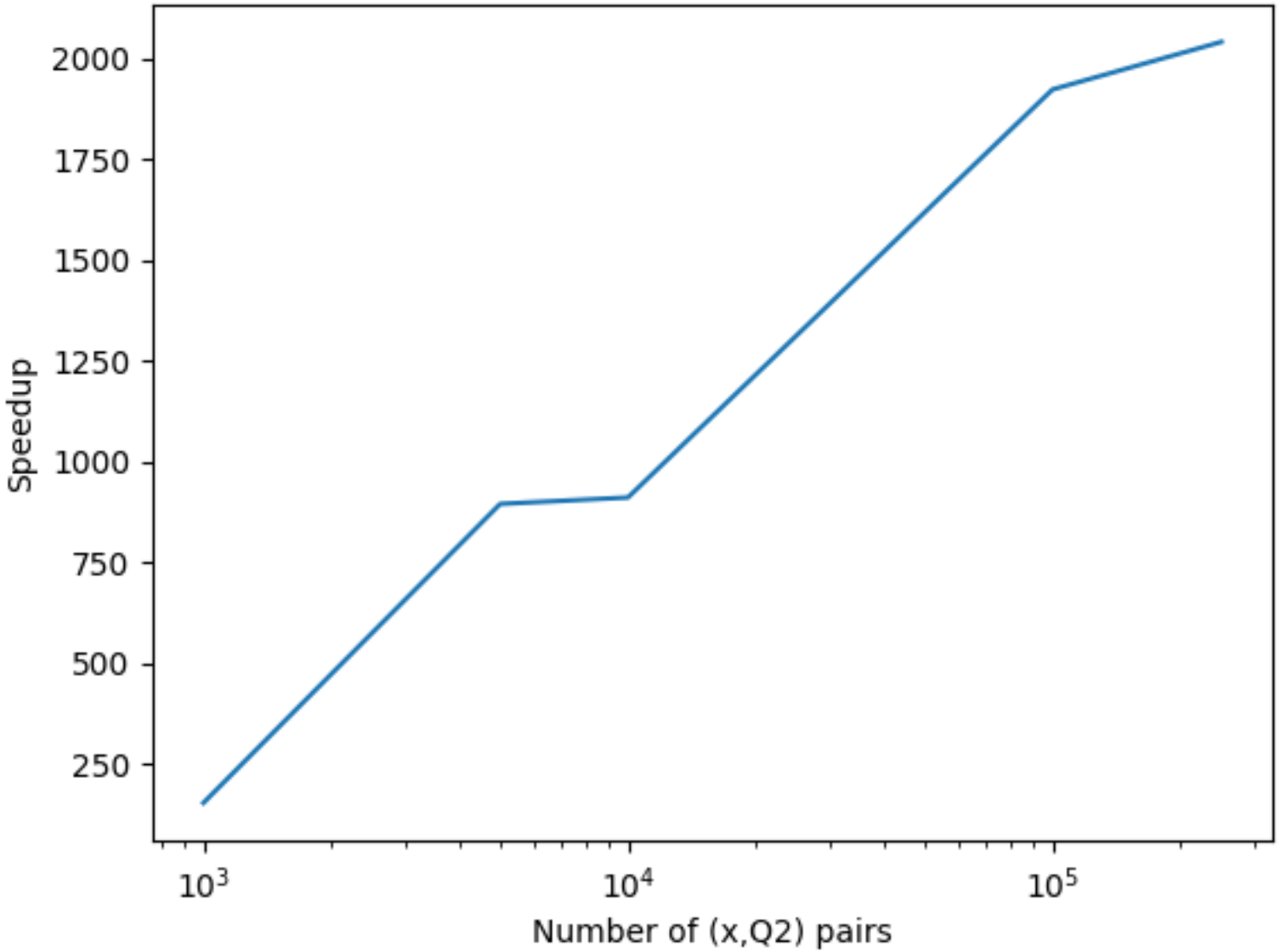
We need experiments 50X the current size before we can fully utilize the GPU with a single rank

Speedup of GPU fitpack



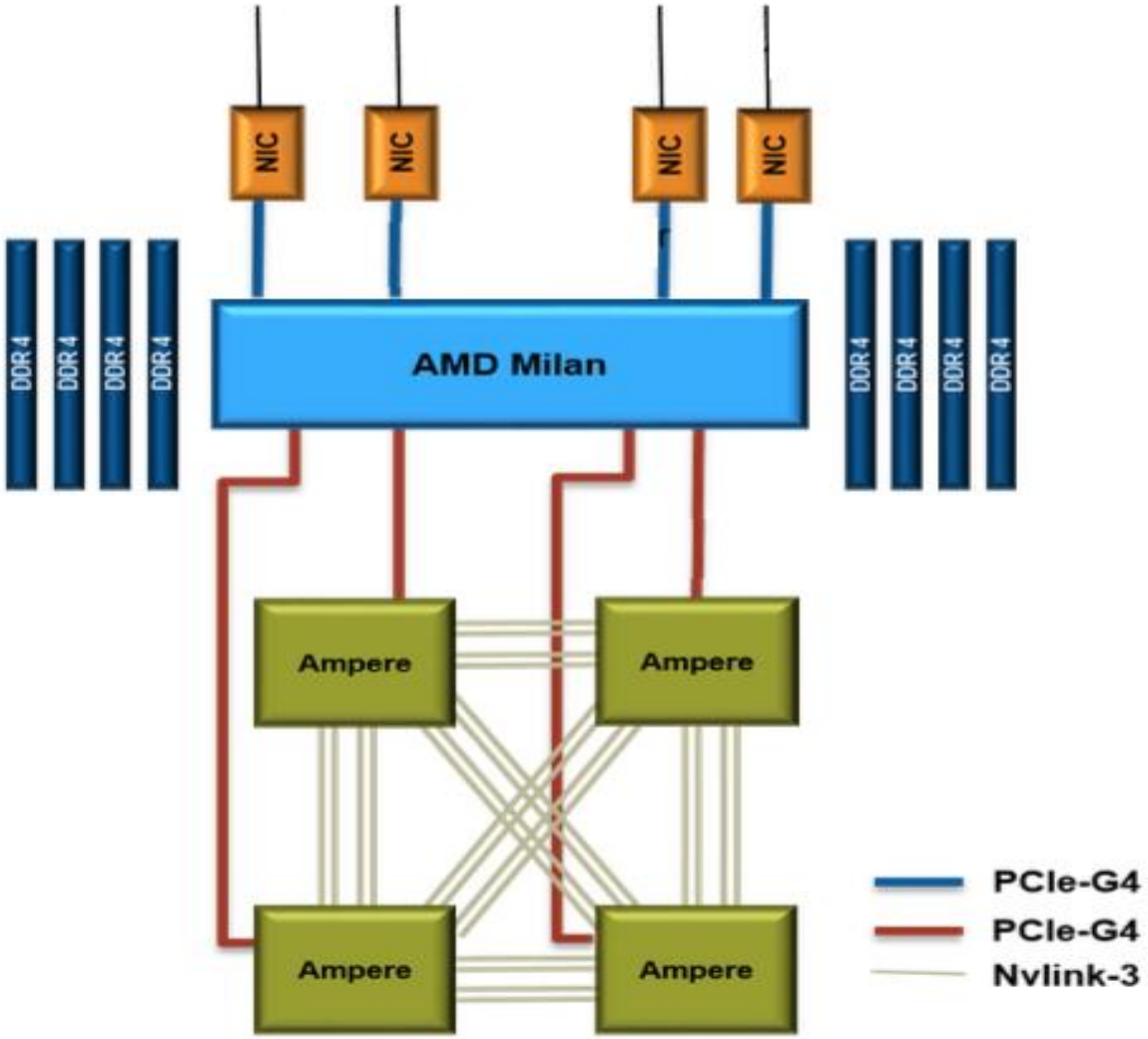
We can either run multiple ranks per GPU like we did for Pytorch GPU

Speedup of GPU fitpack

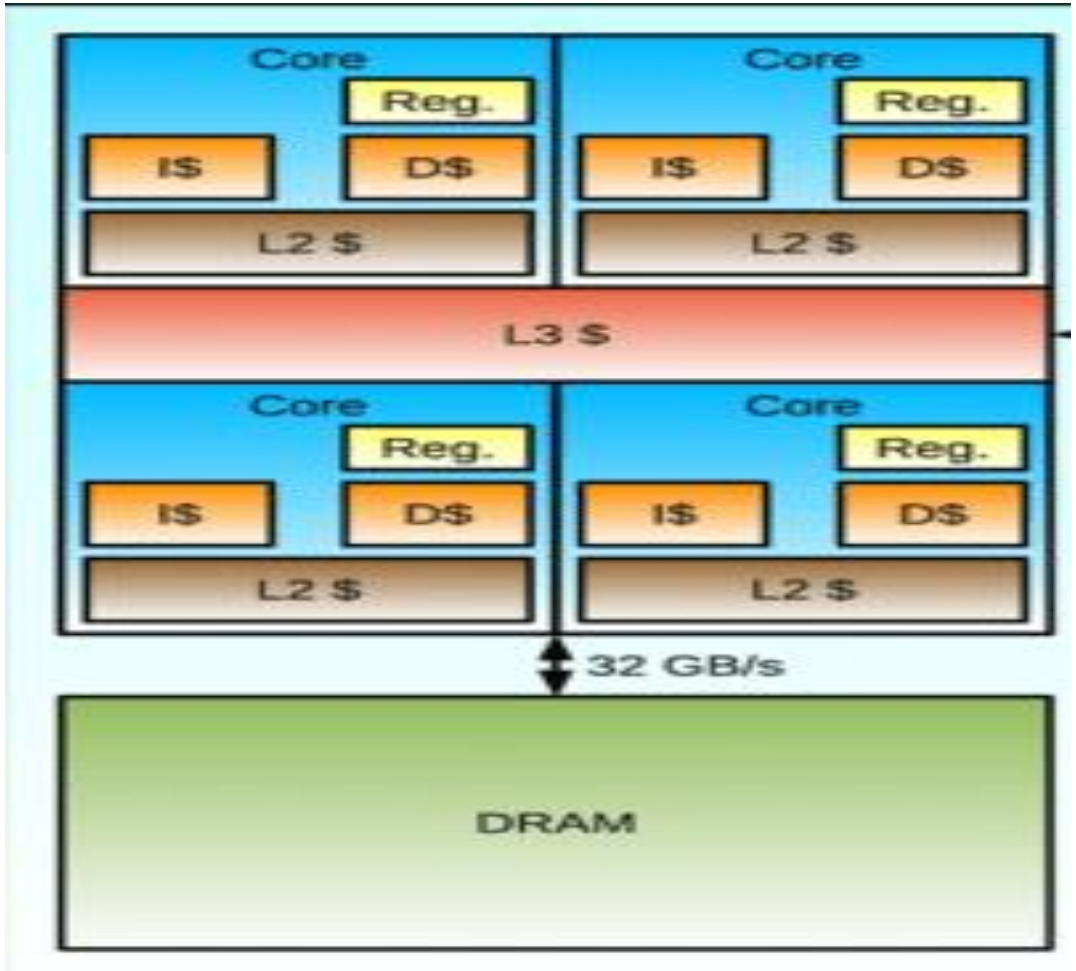


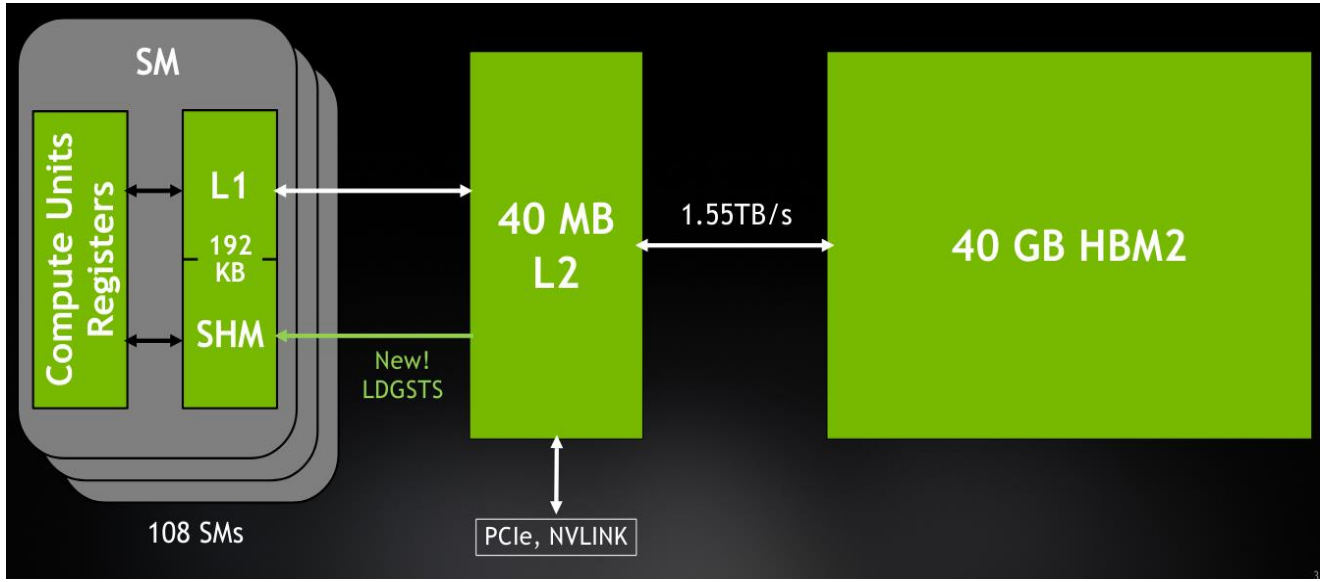
But the more scalable solution is to increase the parallel efficiency

So what goes into the Salad?

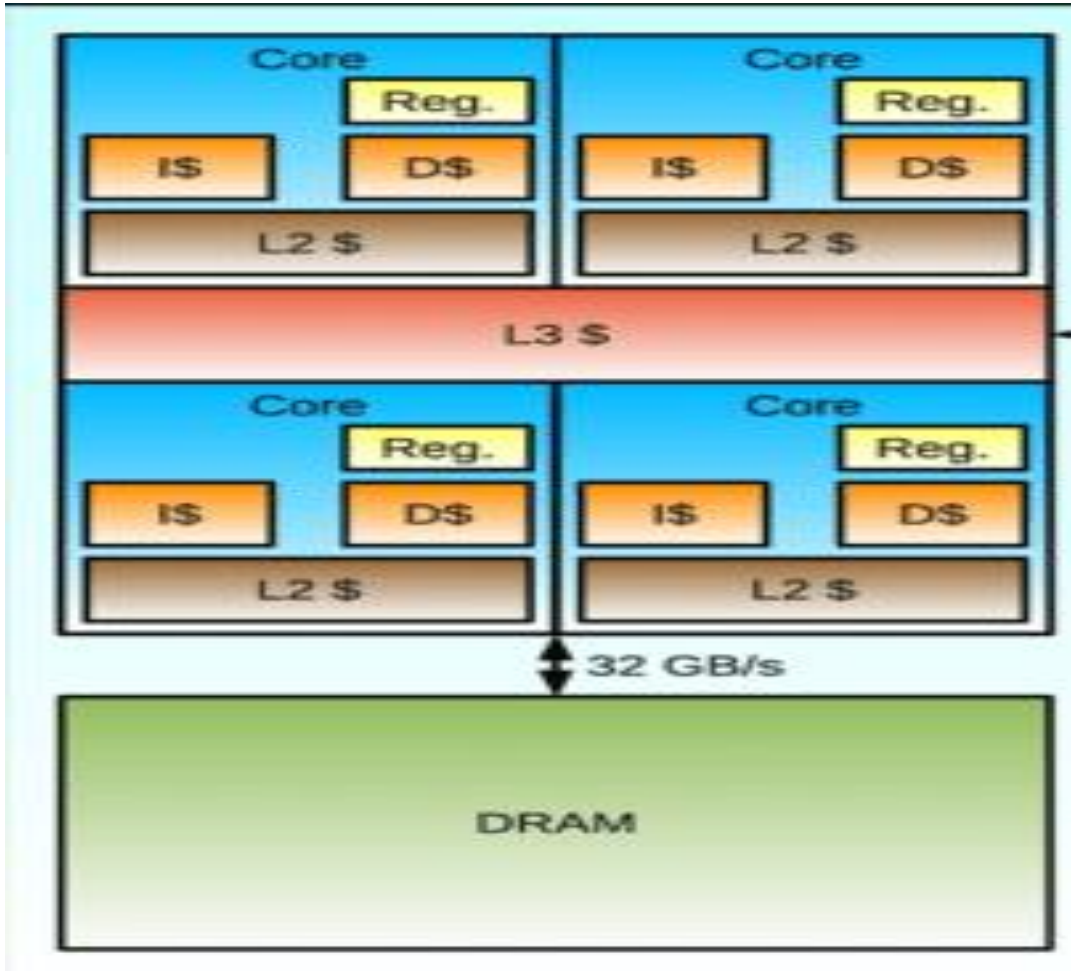


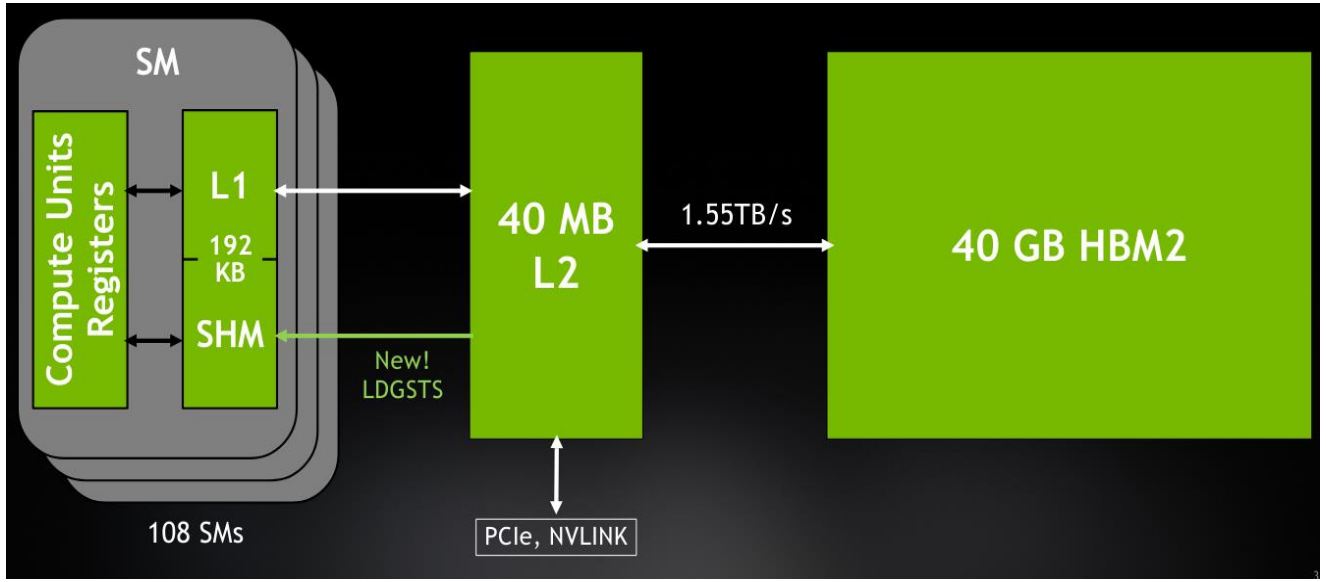
Memory Hierarchy





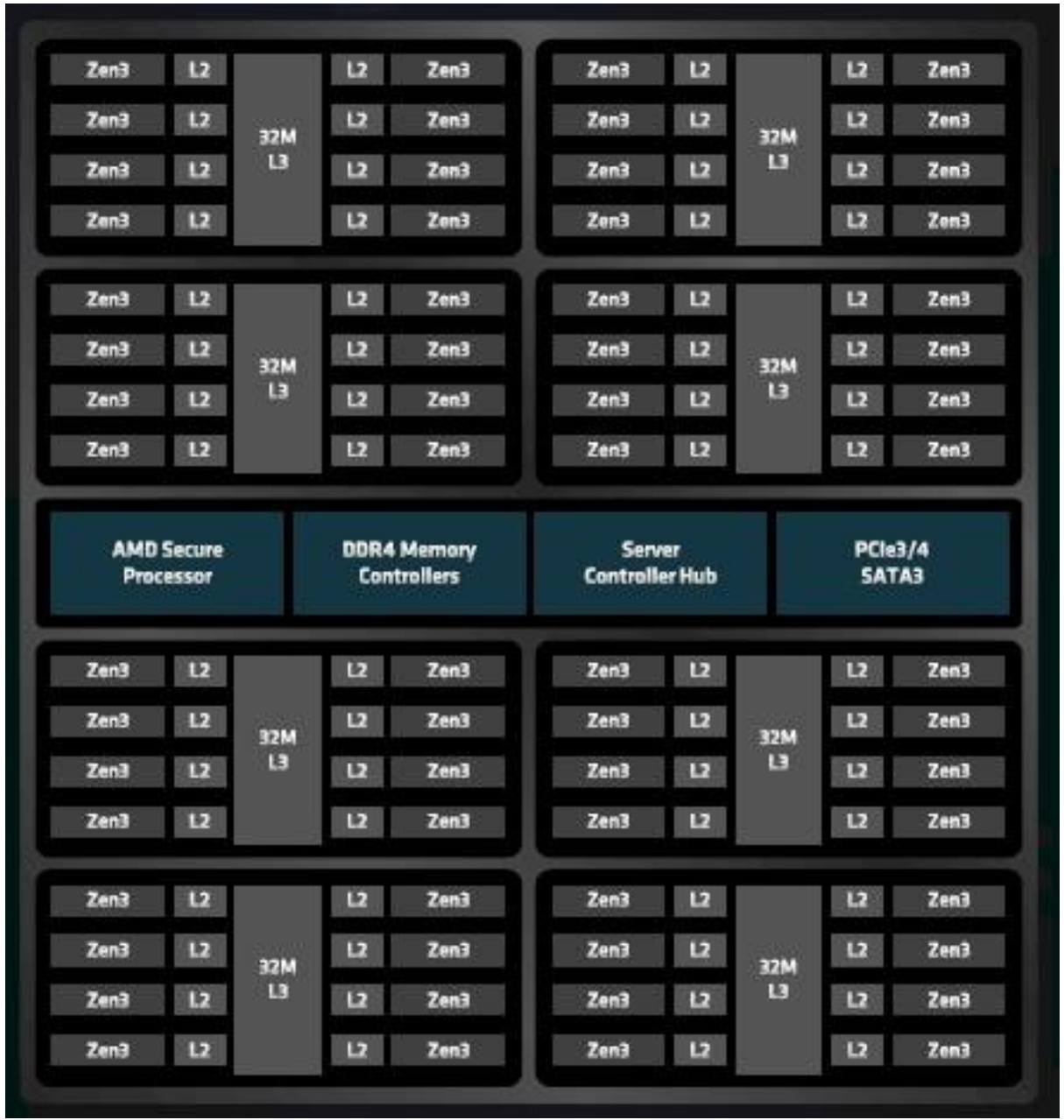
Memory Hierarchy





Caches are everywhere!

Compute Hierarchy (CPU)

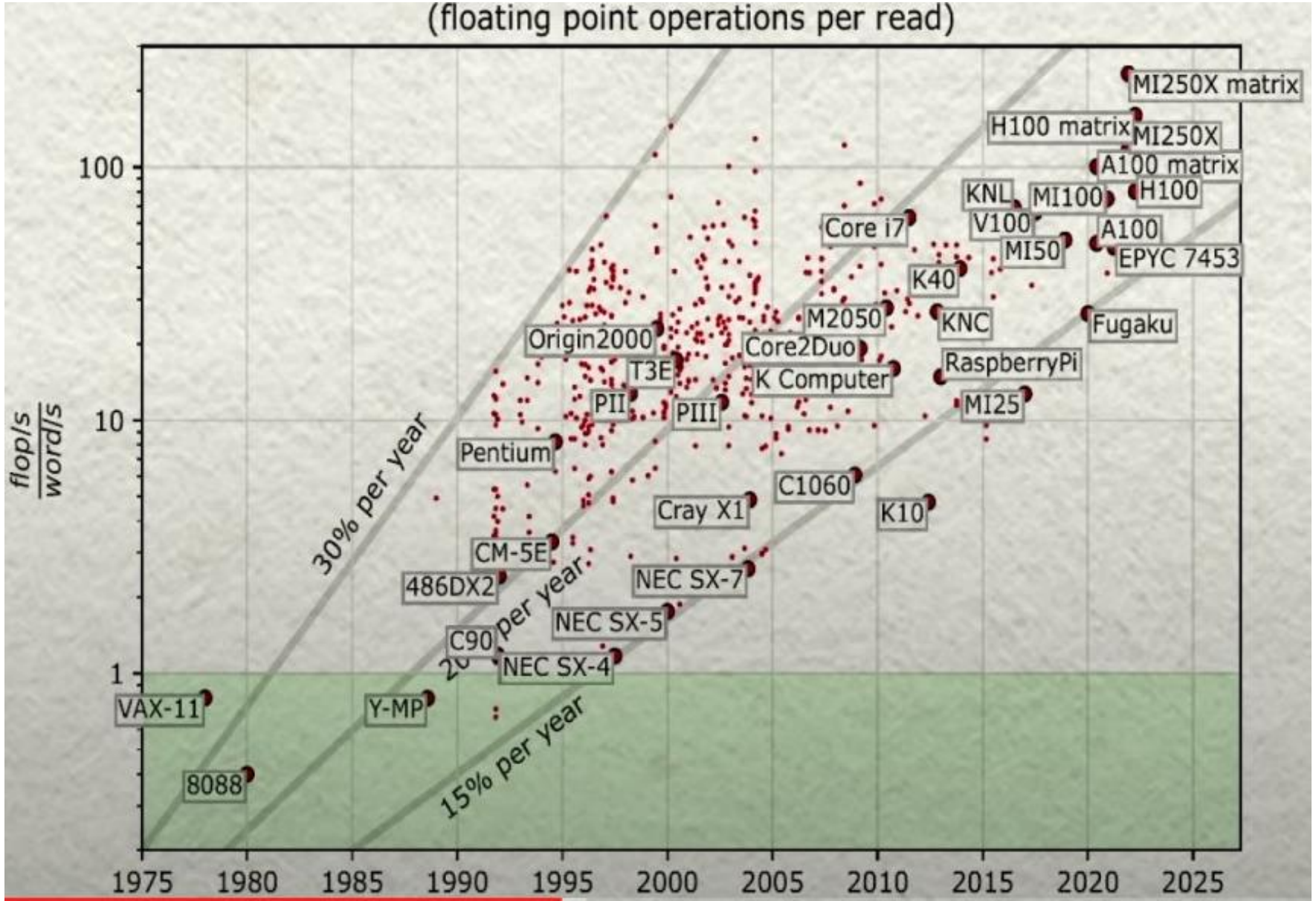


Compute Hierarchy (GPU)

Compute Hierarchy (GPU)

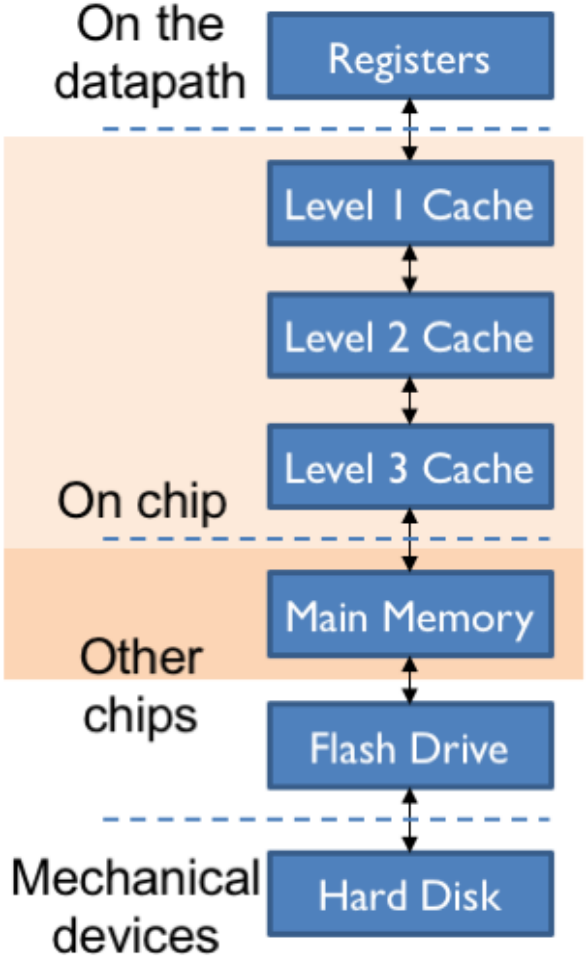
X 144

Machine Balance



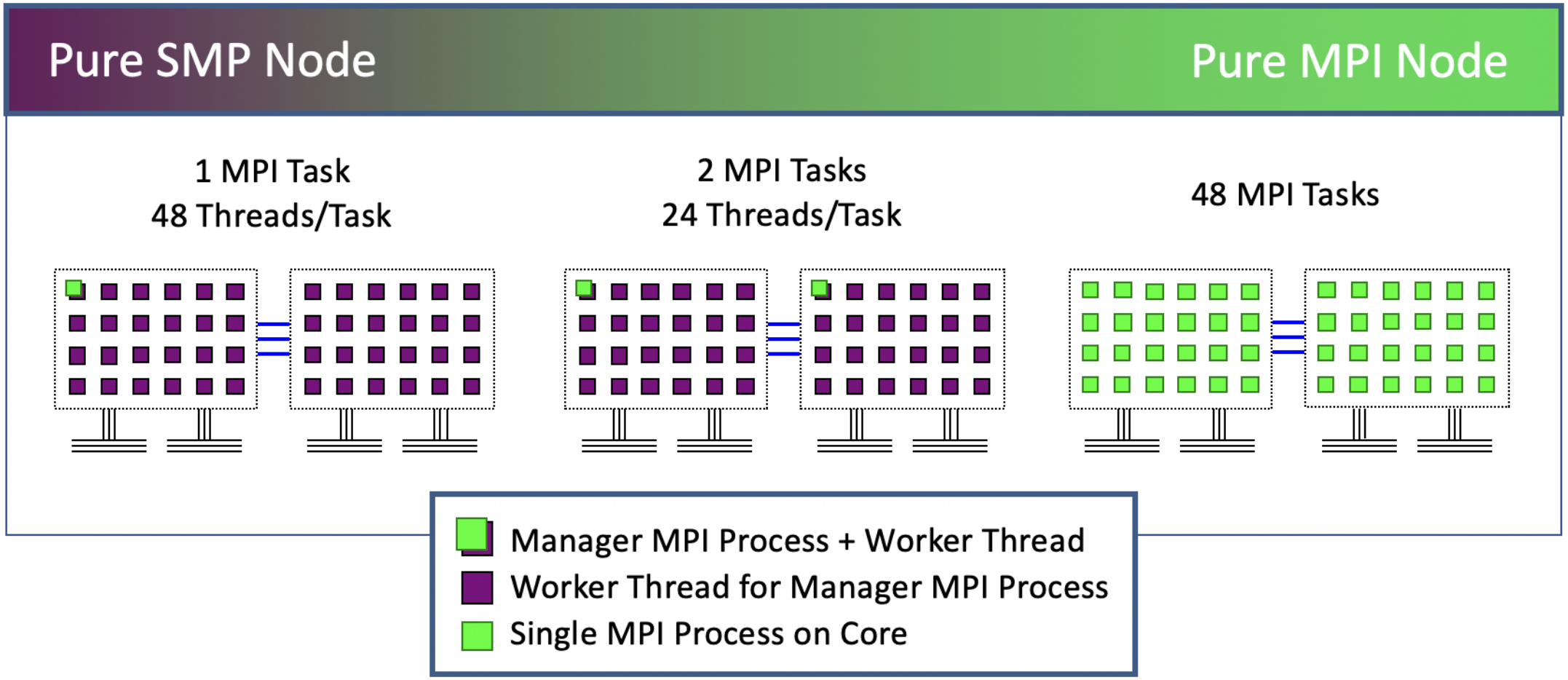
Memory Hierarchy

- Everything is a cache for something else



Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

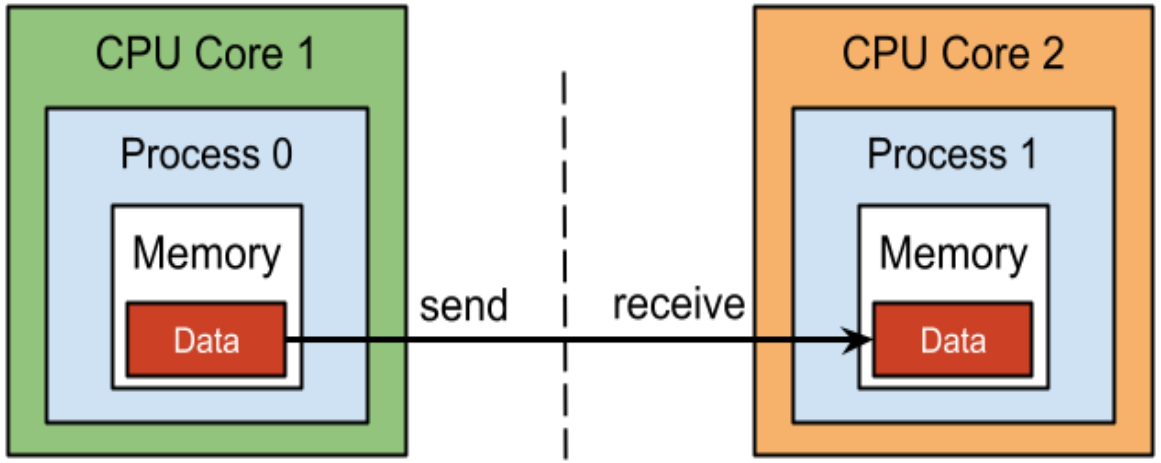
Parallel and Distributed Computing

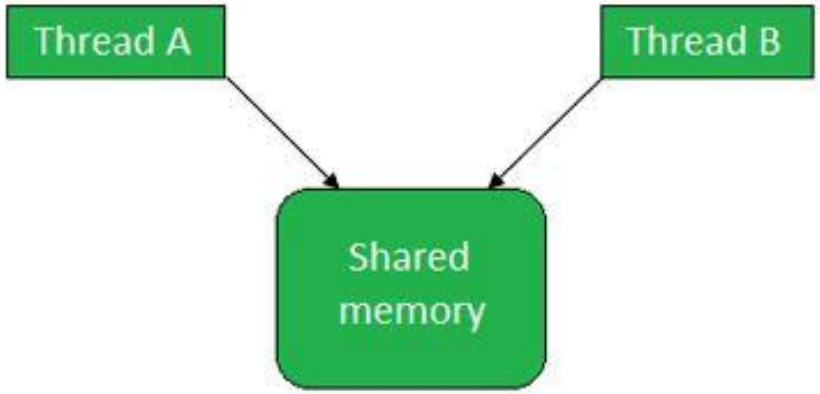


Overheads of Parallelism (Some…)

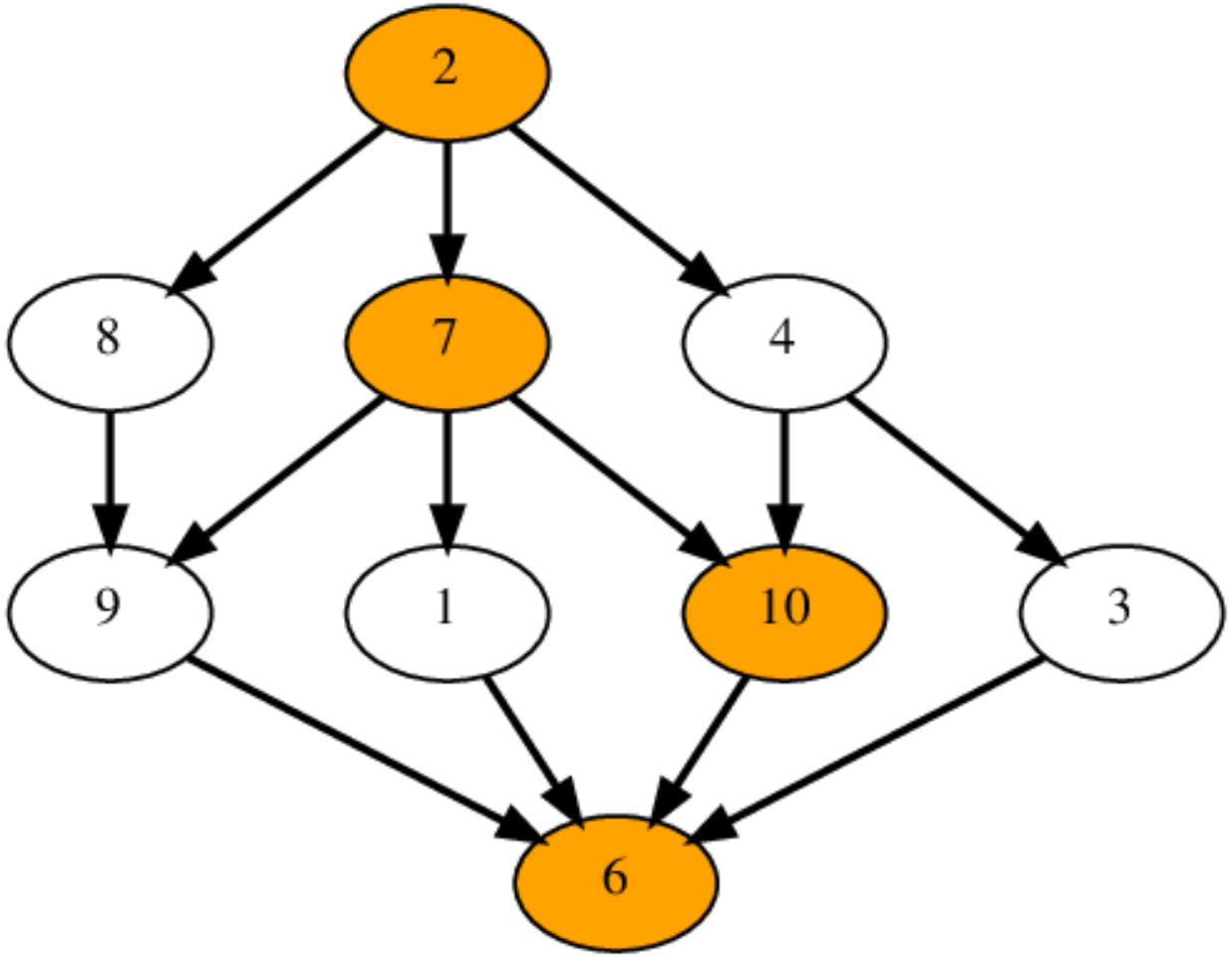
- Communication overhead
- Synchronization overhead
- Load balancing overhead
- Decomposition overhead

Overheads of Parallelism
Communication Overhead

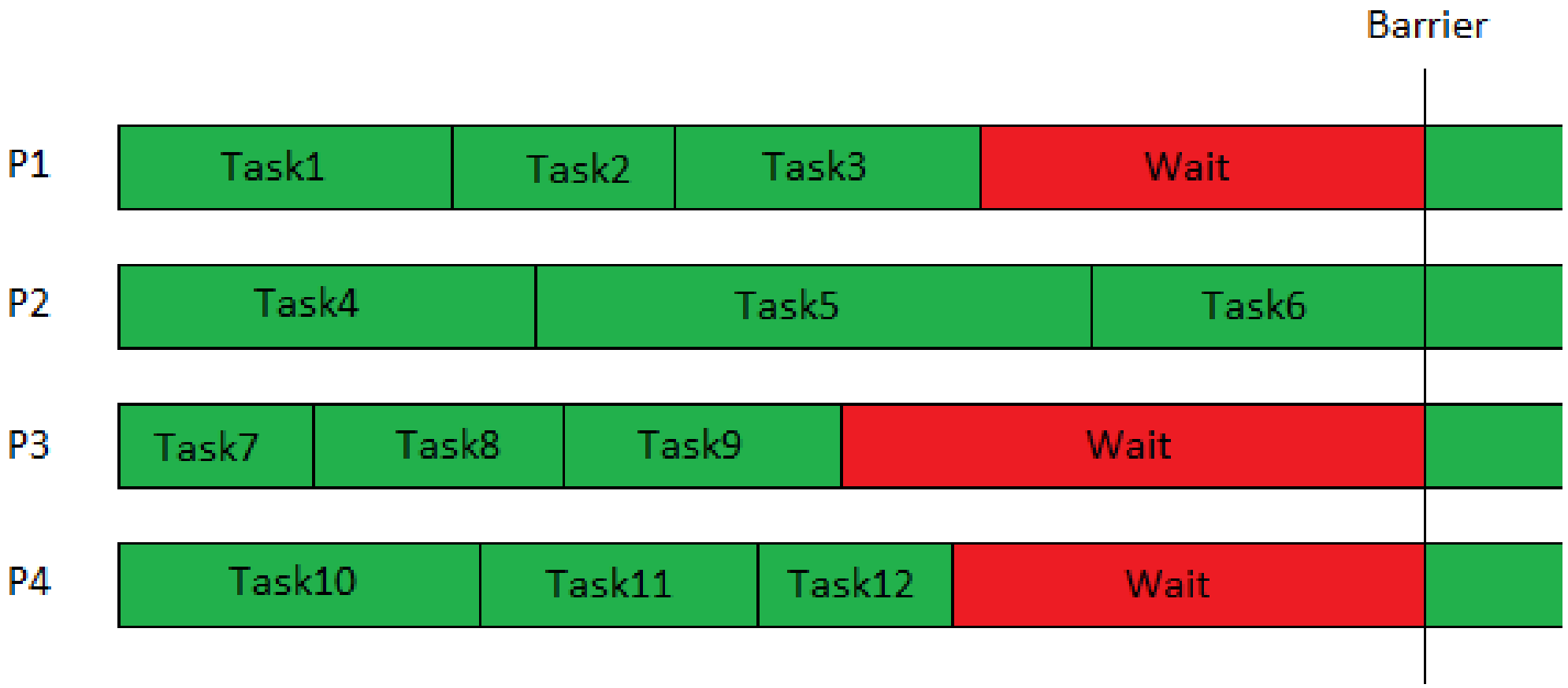




Overheads of Parallelism
Synchronization Overhead

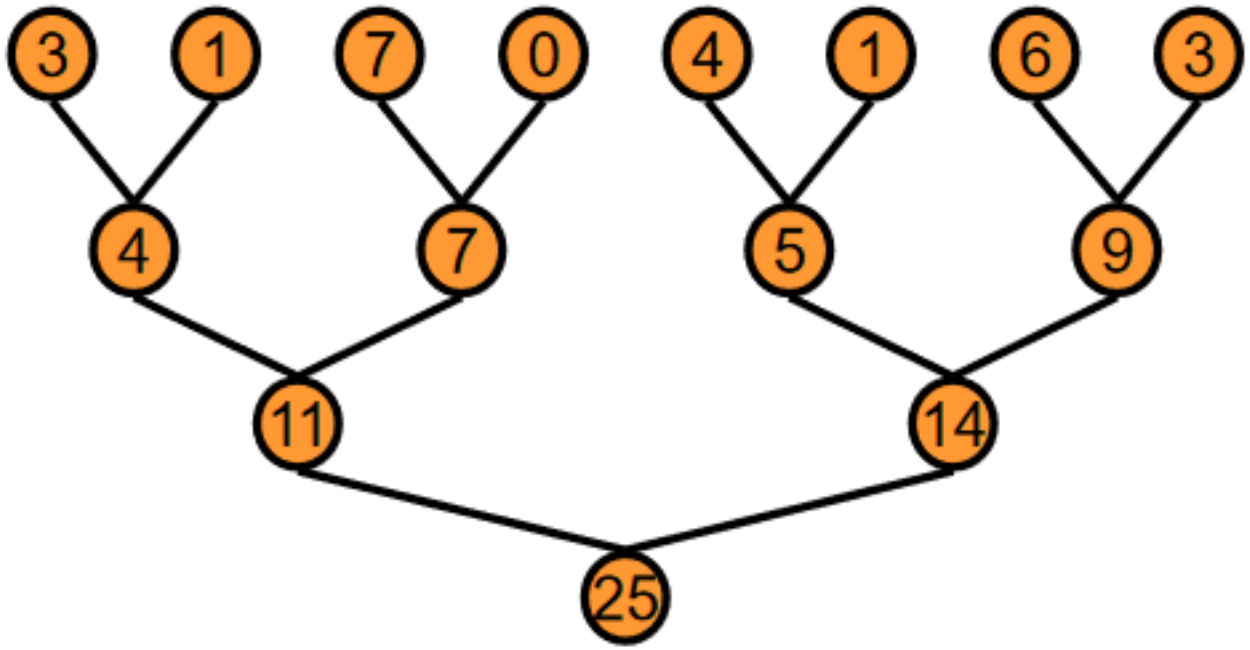


Overheads of Parallelism
Load balancing Overhead



Overheads of Parallelism

Decomposition overhead



Conclusion

- Our goal should be to reduce the time to solution
  - Look at not just computational efficiency of 1 worker, but of all of them working together.
- We need to ensure that we are comparing APPLES to APPLES!
- Parallel computing is not free, we need to be cognizant of the many overheads of parallelization
  - Poor parallelization may even degrade performance