Introduction to High Performance Computing

Bálint Joó

Jefferson Lab, Newport News, VA, USA

HUGS'15
June, 12, 2015

Part 1

- Computational Science in General
- Parallel Computing
  - Computing
  - Hardware Trends
  - Multi-Core Chips and GPUs
  - Very brief introduction to parallel programming

HPC In Science & Engineering

- The “3rd Pillar of Science and Engineering”

SCIENCE & ENGINEERING

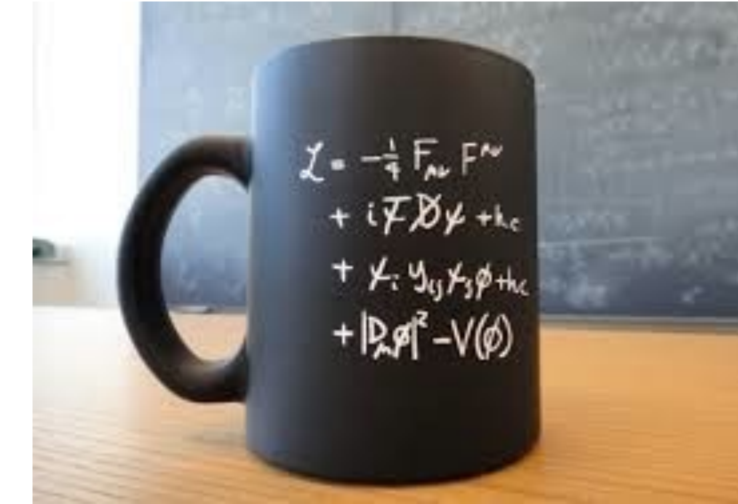
Theory

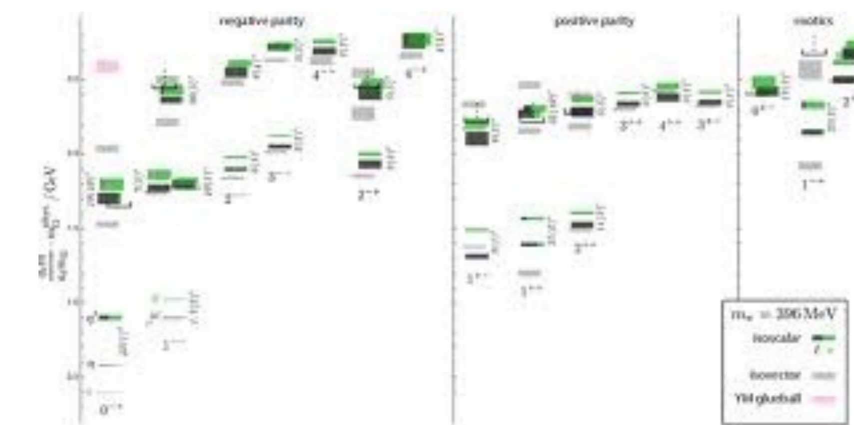
Computation

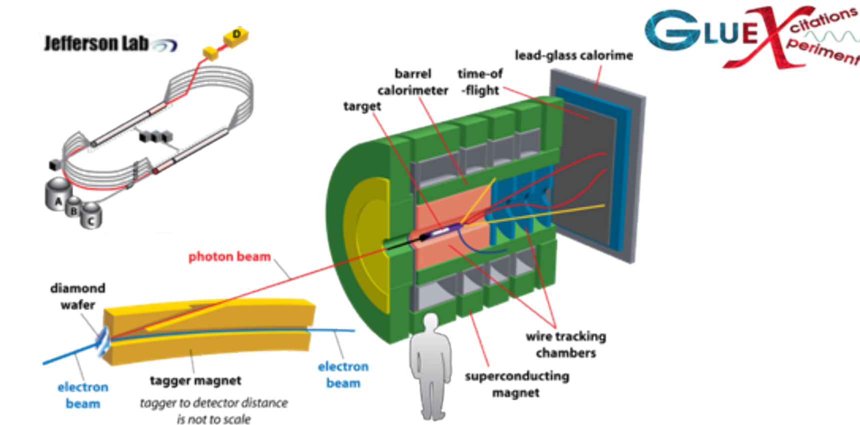
Experiment

HPC In Science

- The “3rd Pillar of Science and Engineering”
  - Connect Theory to Experiment





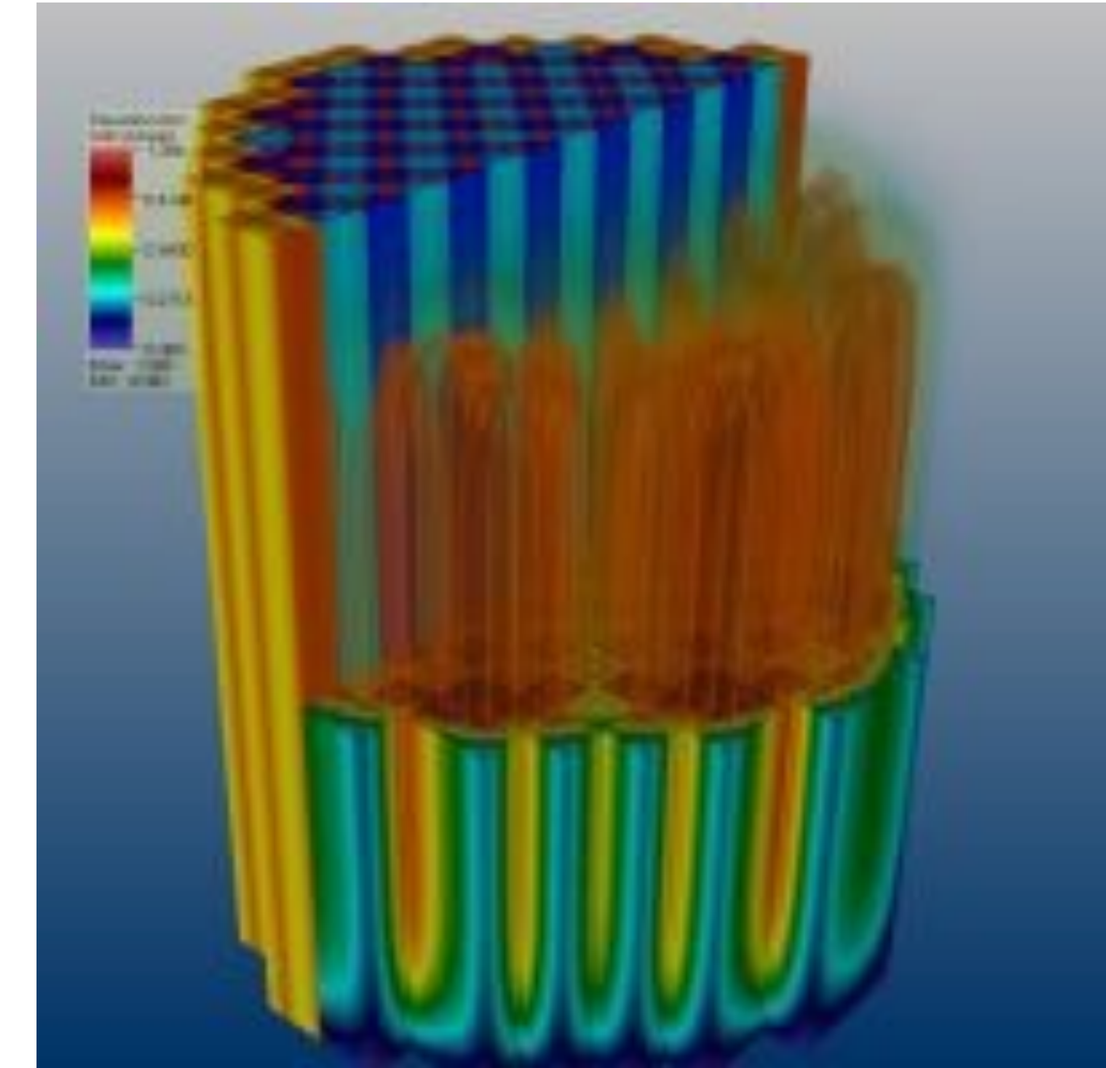


HPC In Science

- The “3rd Pillar of Science and Engineering”
  - Connect Theory to Experiment
  - Virtual experiment where
    - real controlled experiments are not possible

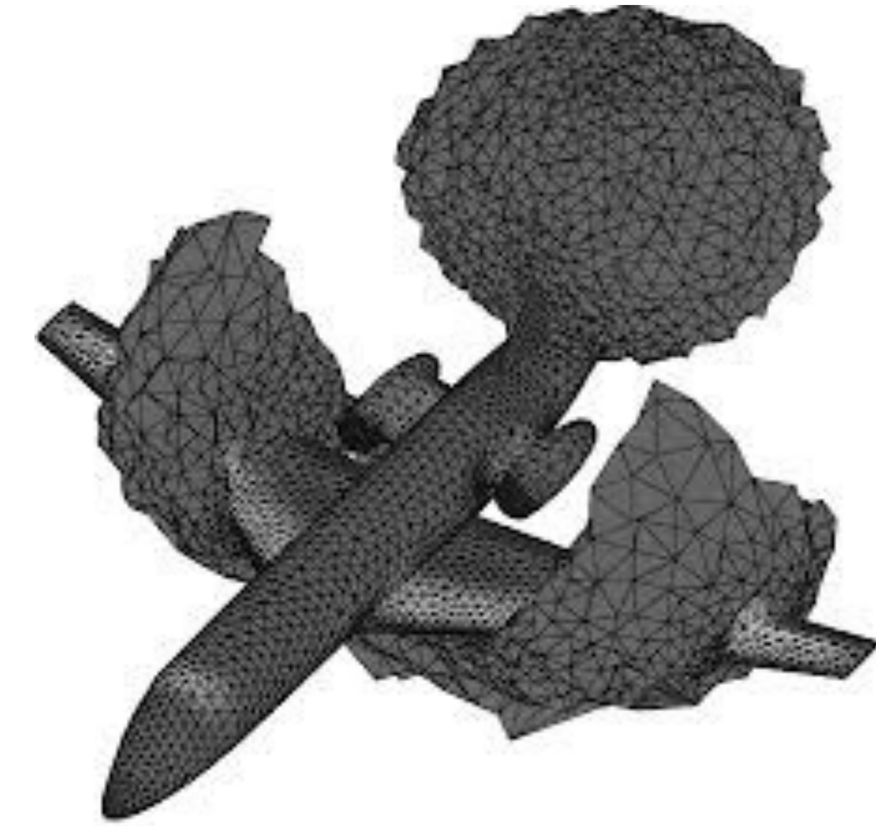
HPC In Science

- The “3rd Pillar of Science and Engineering”
  - Connect Theory to Experiment
  - Virtual experiment where
    - real controlled experiments are not possible
    - or may be perhaps hazardous



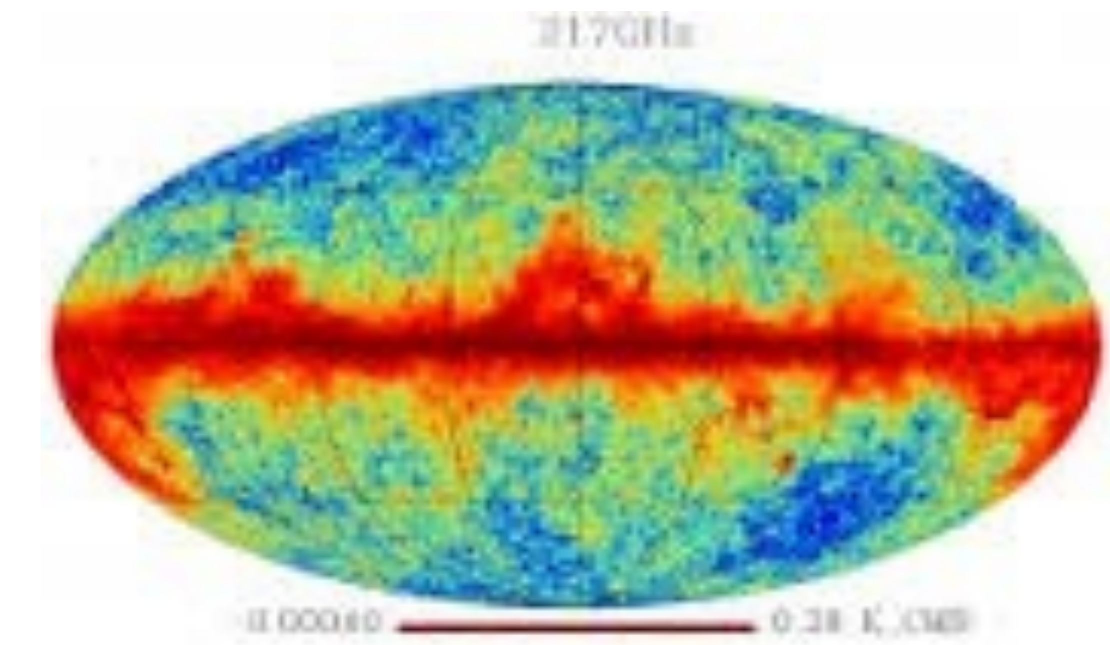
HPC In Science

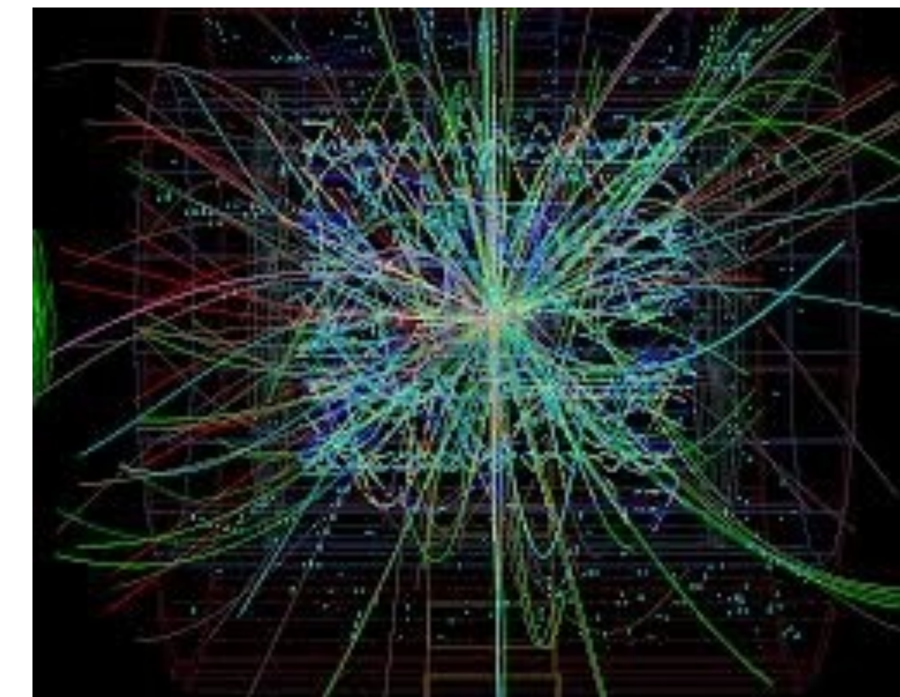
- The “3rd Pillar of Science and Engineering”
  - Connect Theory to Experiment
  - Virtual experiment where
    - real controlled experiments are not possible
    - or may be perhaps hazardous
    - real experiment can be expensive
      - engineering and design applications



HPC In Science

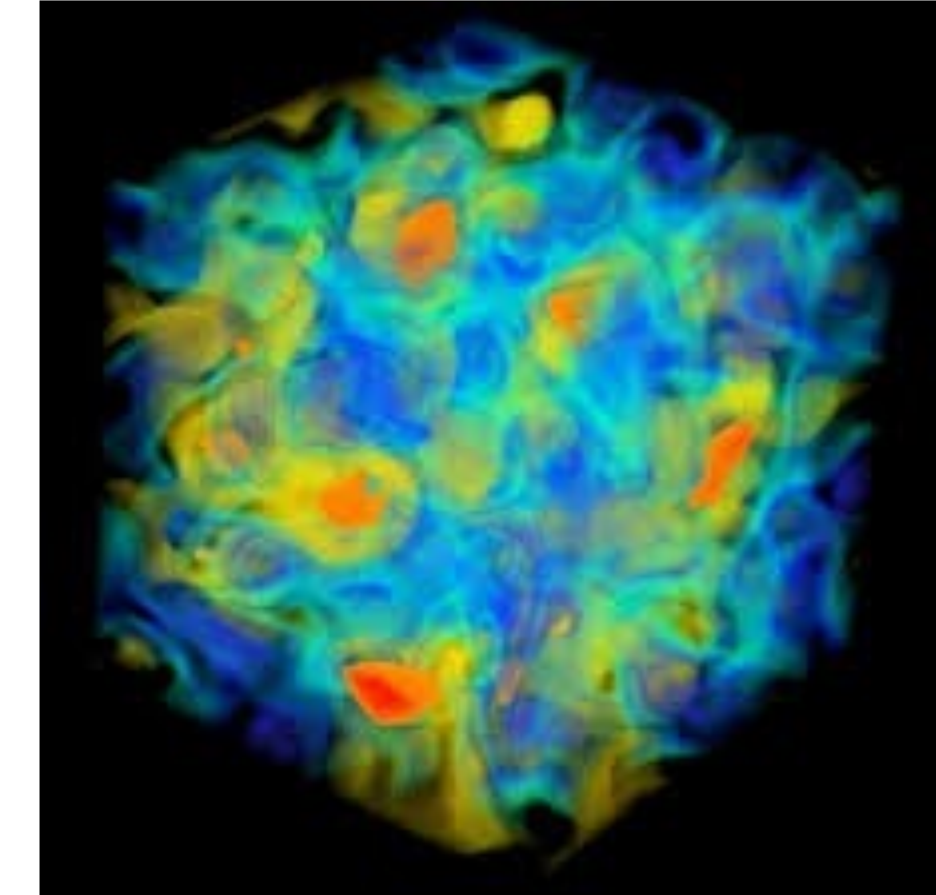
- The “3rd Pillar of Science and Engineering”
  - Connect Theory to Experiment
  - Virtual experiment where
    - real controlled experiments are not possible
    - or may be perhaps hazardous
    - real experiment can be expensive
      - engineering and design applications
- Can be Data Driven
  - E.g. Planck Satellite Analysis, CERN data analysis

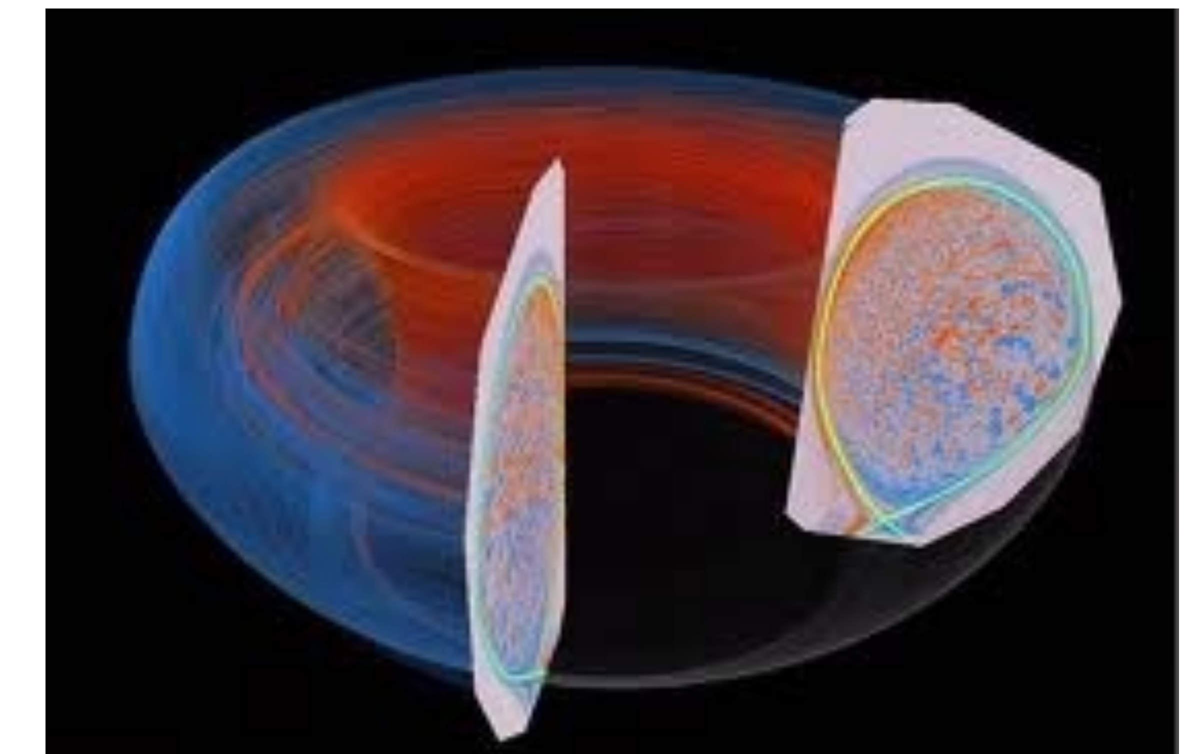




HPC In Science

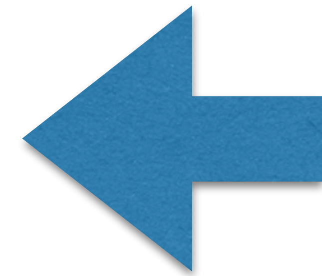
- The “3rd Pillar of Science and Engineering”
  - Connect Theory to Experiment
  - Virtual experiment where
    - real controlled experiments are not possible
    - or may be perhaps hazardous
    - real experiment can be expensive
      - engineering and design applications
- Can be Data Driven
  - E.g. Planck Satellite Analysis, CERN data analysis
- Or Driven by Computation
  - Evolve Simulations of Physical Systems

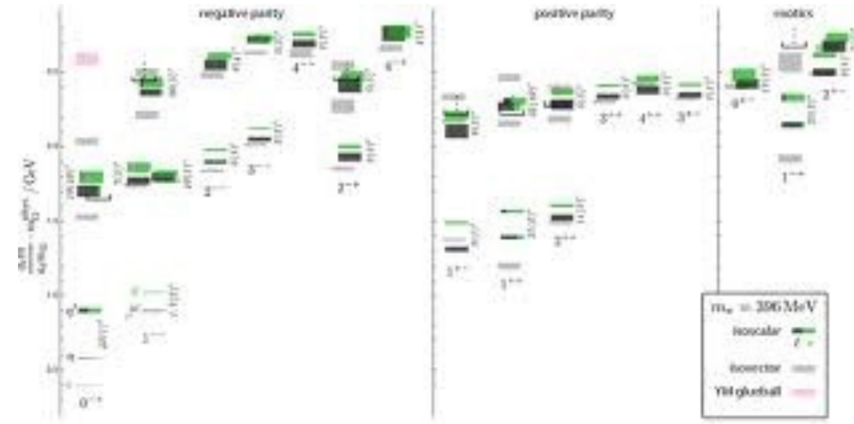




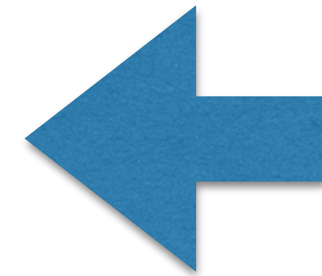
HPC Cycle

Science Question

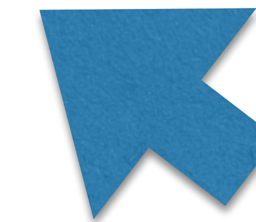




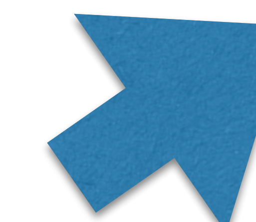
Answer



“Production”: perform computation

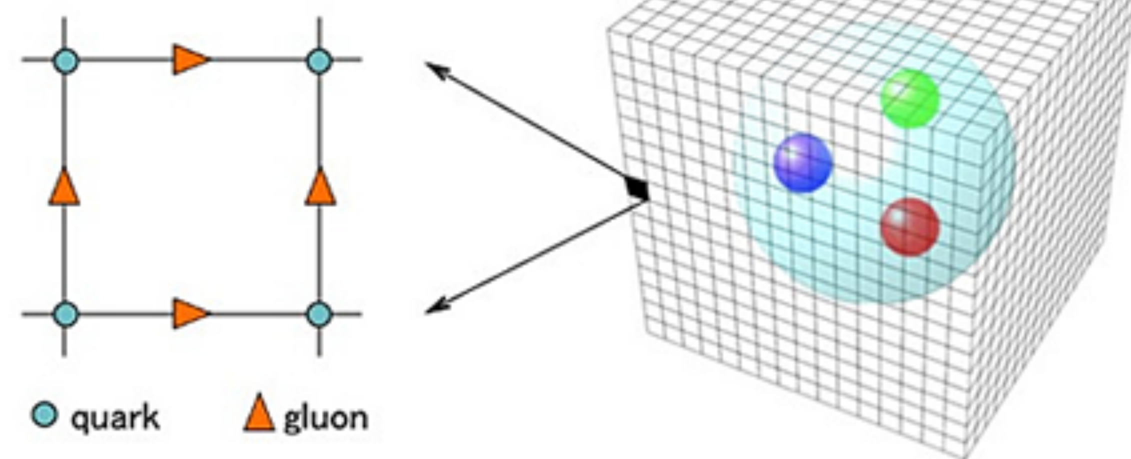


Software Implementation

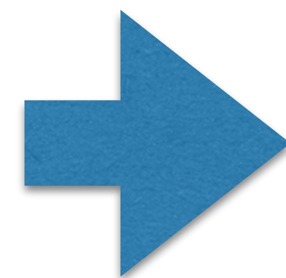


QCD Lagrangian

$$\mathcal{L} = -\frac{1}{4}F^{\mu\nu}F_{\mu\nu} + \sum_{q=u,d,s,c,b,t} \bar{q}\left[i\gamma^{\mu}(\partial_{\mu} - igA_{\mu}) - m_q\right]q$$



Pose computational question



$$e^{\delta\tau\hat{Q}}$$

$$e^{\frac{\delta\tau}{2}\hat{P}} \quad e^{\frac{\delta\tau}{2}\hat{P}}$$

$$e^{\frac{\delta\tau}{2}\hat{P}} e^{\delta\tau\hat{Q}} e^{\frac{\delta\tau}{2}\hat{P}}$$

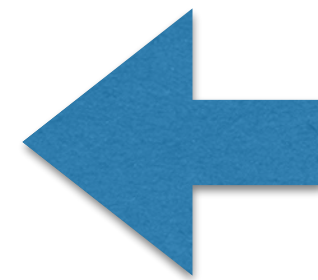
$$P = \min\left(1, e^{-H(U',p')+H(U,p)}\right)$$

1. Compute $r_0 = \chi - M^{\dagger}M\phi_0,\ p_0 = r_0$
2. For $j = 0, 1, \ldots$ until convergence:
3. $\alpha_j = \frac{\langle r_j, r_j \rangle}{\langle M p_j, M p_j \rangle}$
4. $\phi_{j+1} = \phi_j + \alpha_j p_j$
5. $r_{j+1} = r_j - \alpha_j \left(M^{\dagger}M\right) p_j$
6. $\beta_j = \frac{\langle r_{j+1}, r_{j+1} \rangle}{\langle r_j, r_j \rangle}$
7. $p_{j+1} = r_{j+1} + \beta_j p_j$
8. End For

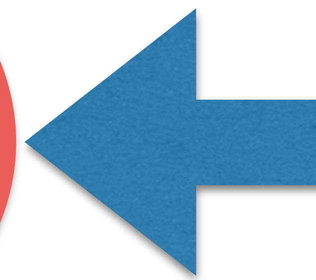
Develop/Collect Computational Algorithms

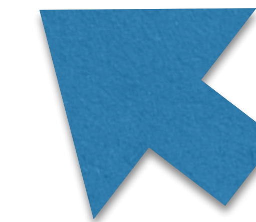
HPC Cycle

Science Question

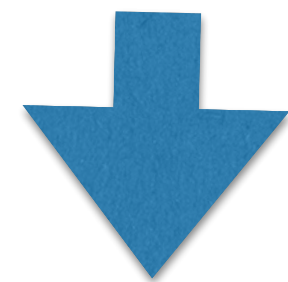


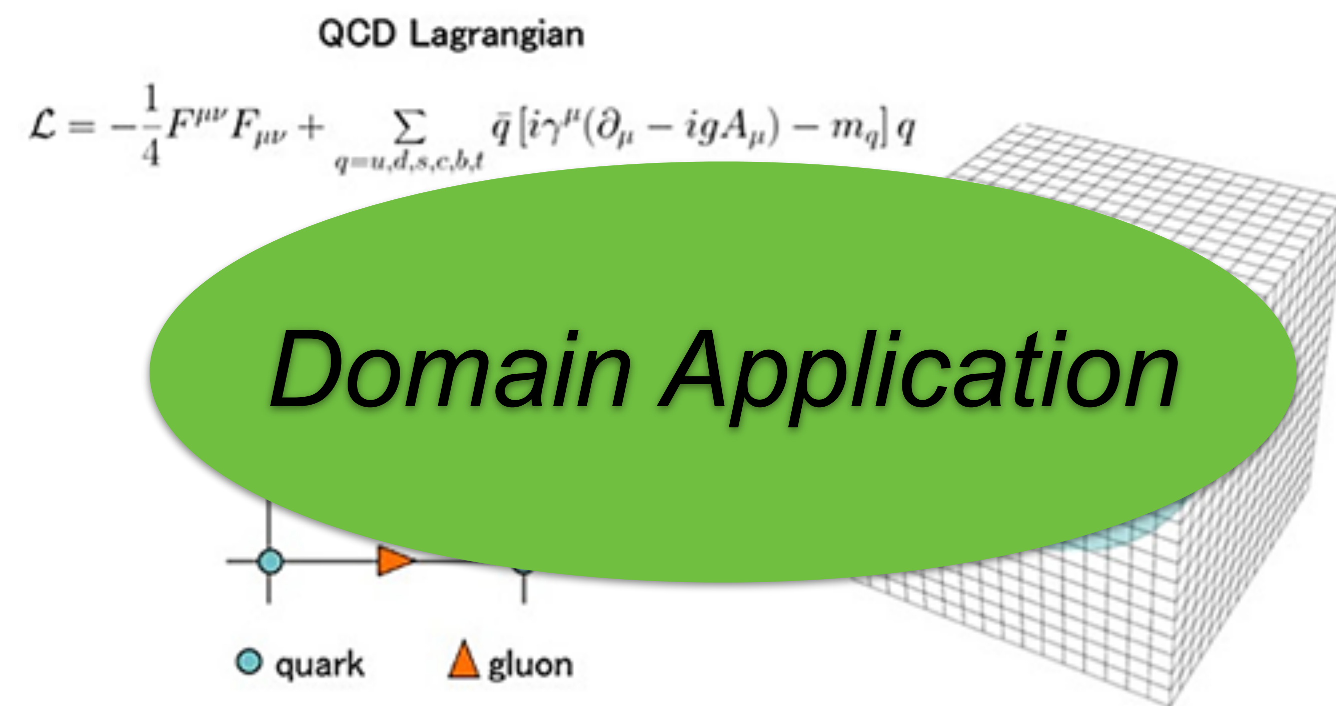
Answer



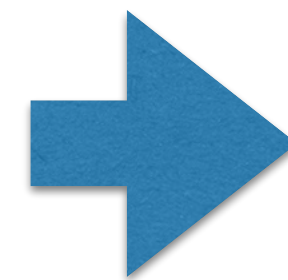


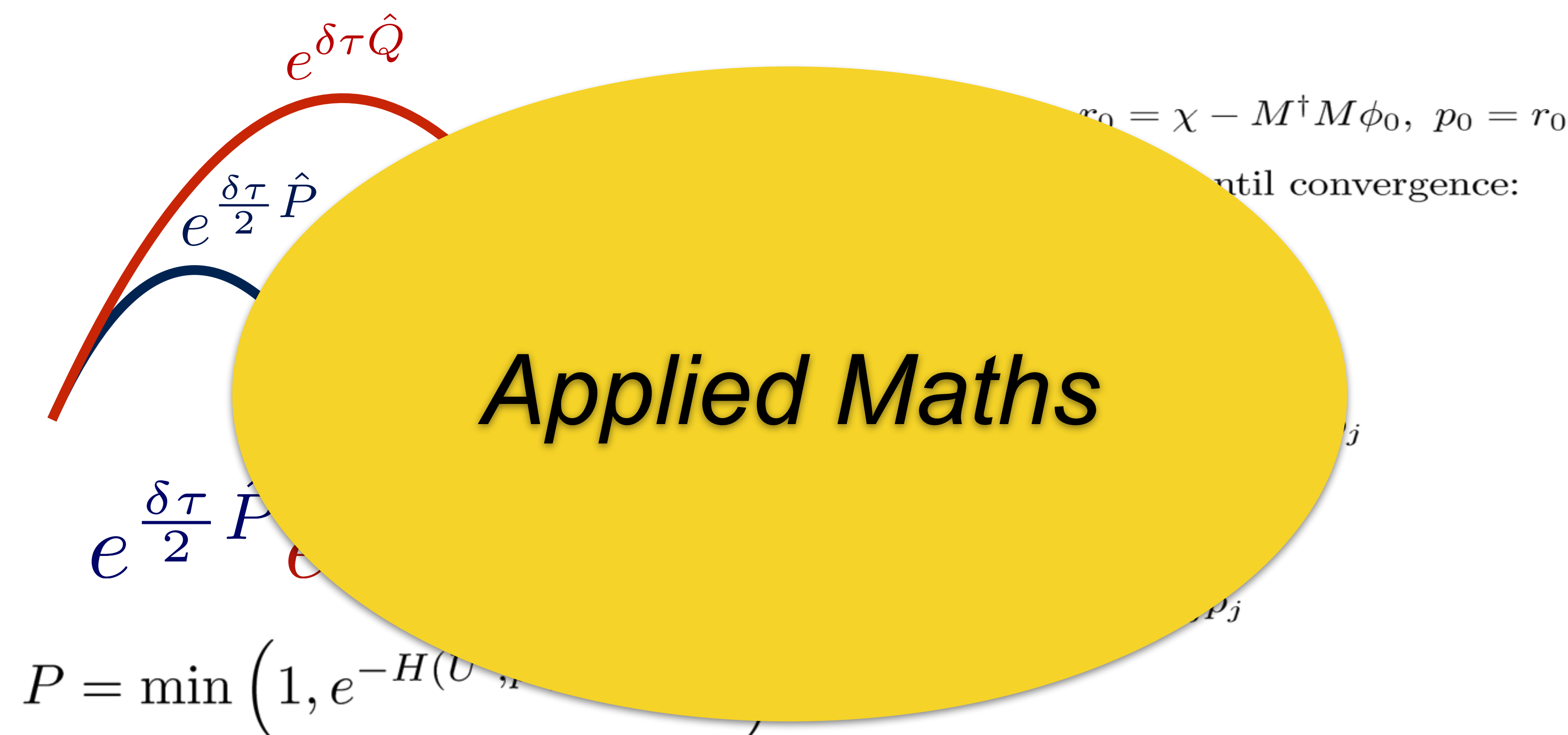
"Production": perform computation



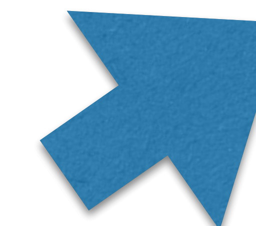


Pose computational question





Develop/Collect Computational Algorithms



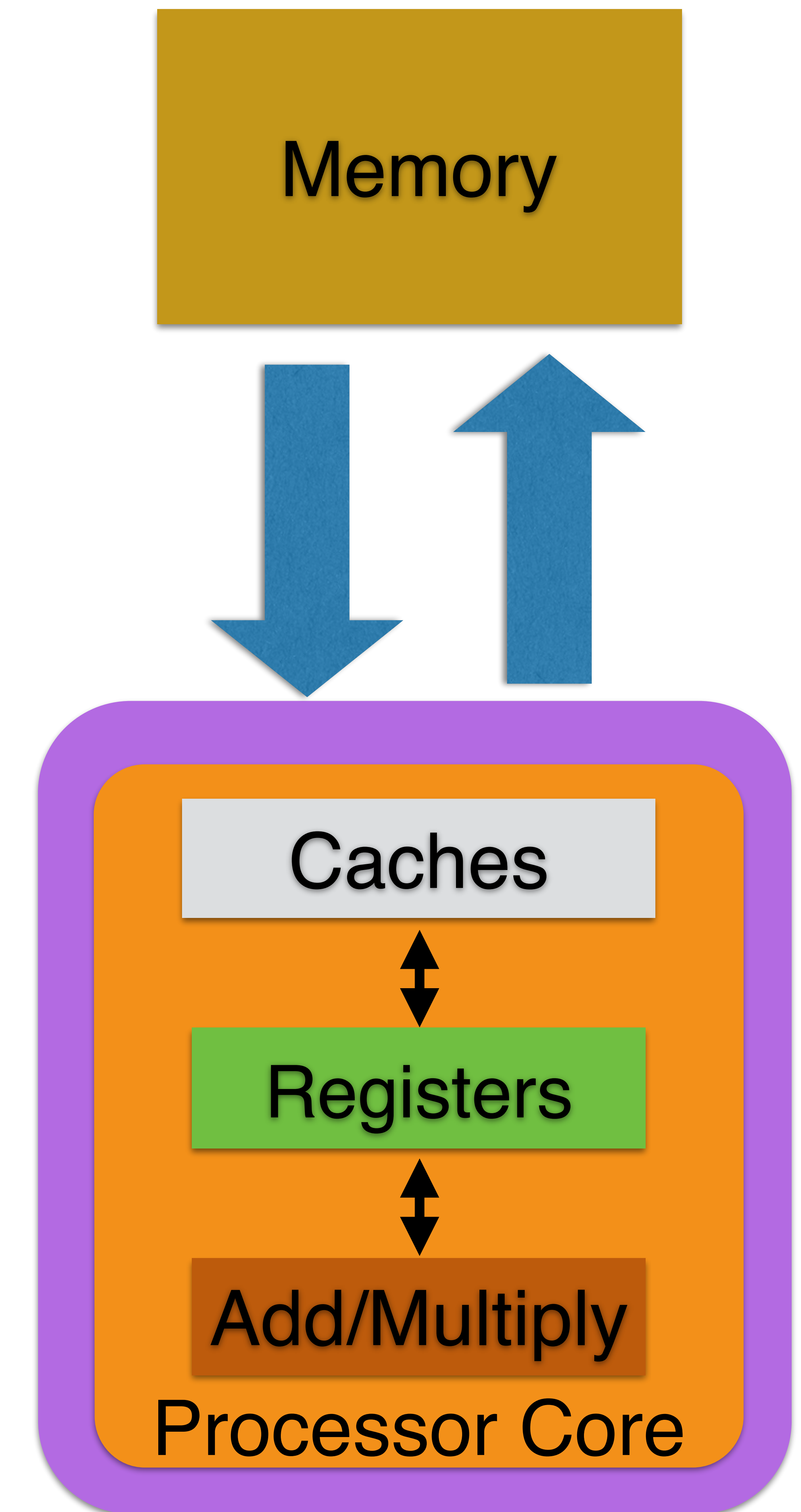
Software Implementation

Parallel Computing

- These days, high performance computing is parallel computing
  - "Several tasks are accomplished concurrently"
- This is partially hardware driven (see later)
- However, many tasks are naturally parallel
  - "Embarassingly Parallel": a collection of independent tasks
    - e.g. event analysis, ray tracing, LQCD contractions
    - parallelism brings throughput
  - Highly Coupled: tasks that interact and need coordination
    - e.g. Finite Difference Stencils, Molecular Dynamics
    - parallelism increases speed of single problem
  - Mixtures of the two: e.g. LQCD propagator calculation

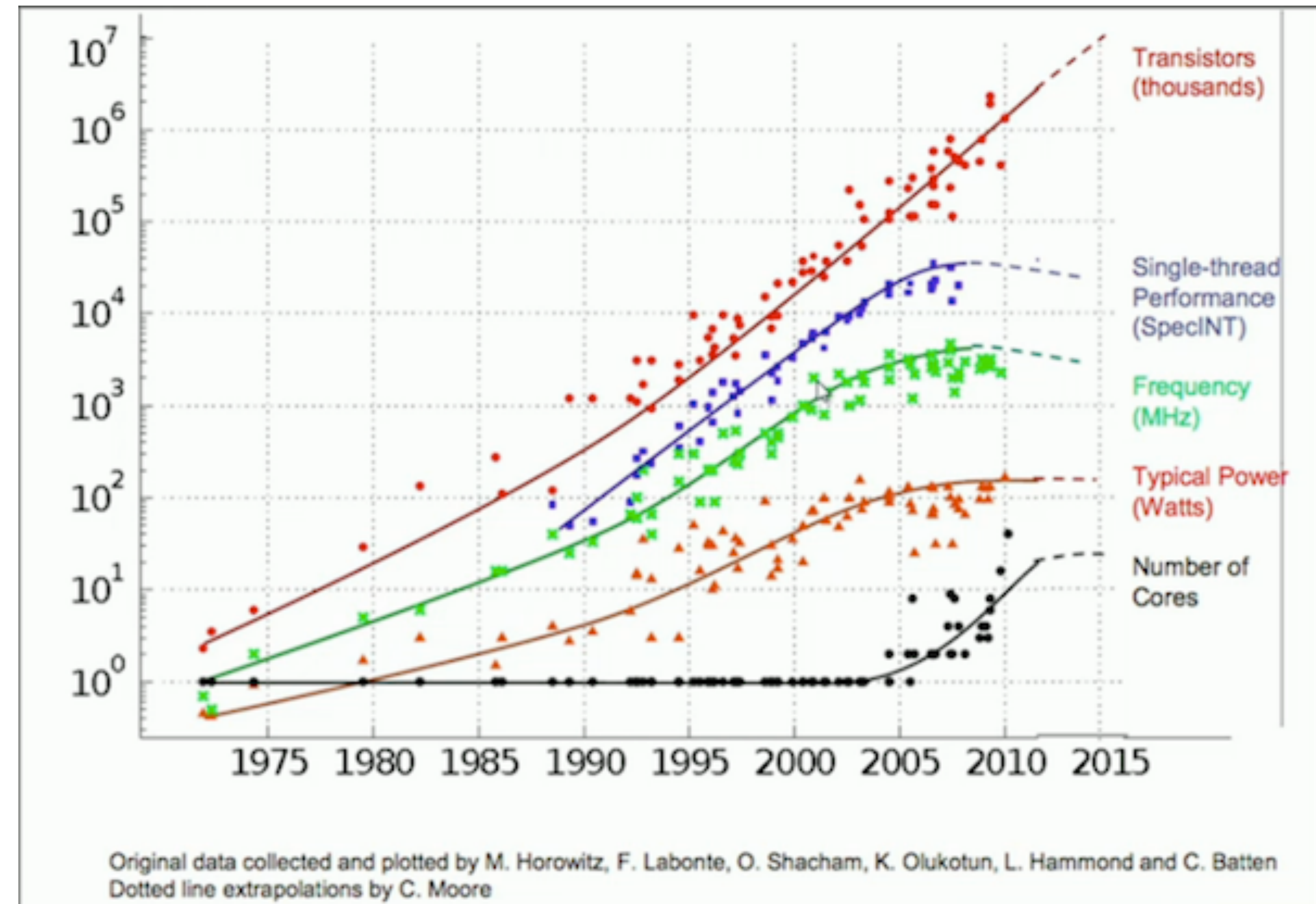
Computer Basics

- Read-Execute-Write
  - read data from Memory into the Processor
  - processor executes Computation
  - result is written back to Memory
- Processor contains
  - Compute components (e.g. Add/Multiply Floating Point units)
  - Registers
    - Floating point unit reads input and writes output to registers
  - Caches
    - Hold data read from memory, in case it is needed again soon



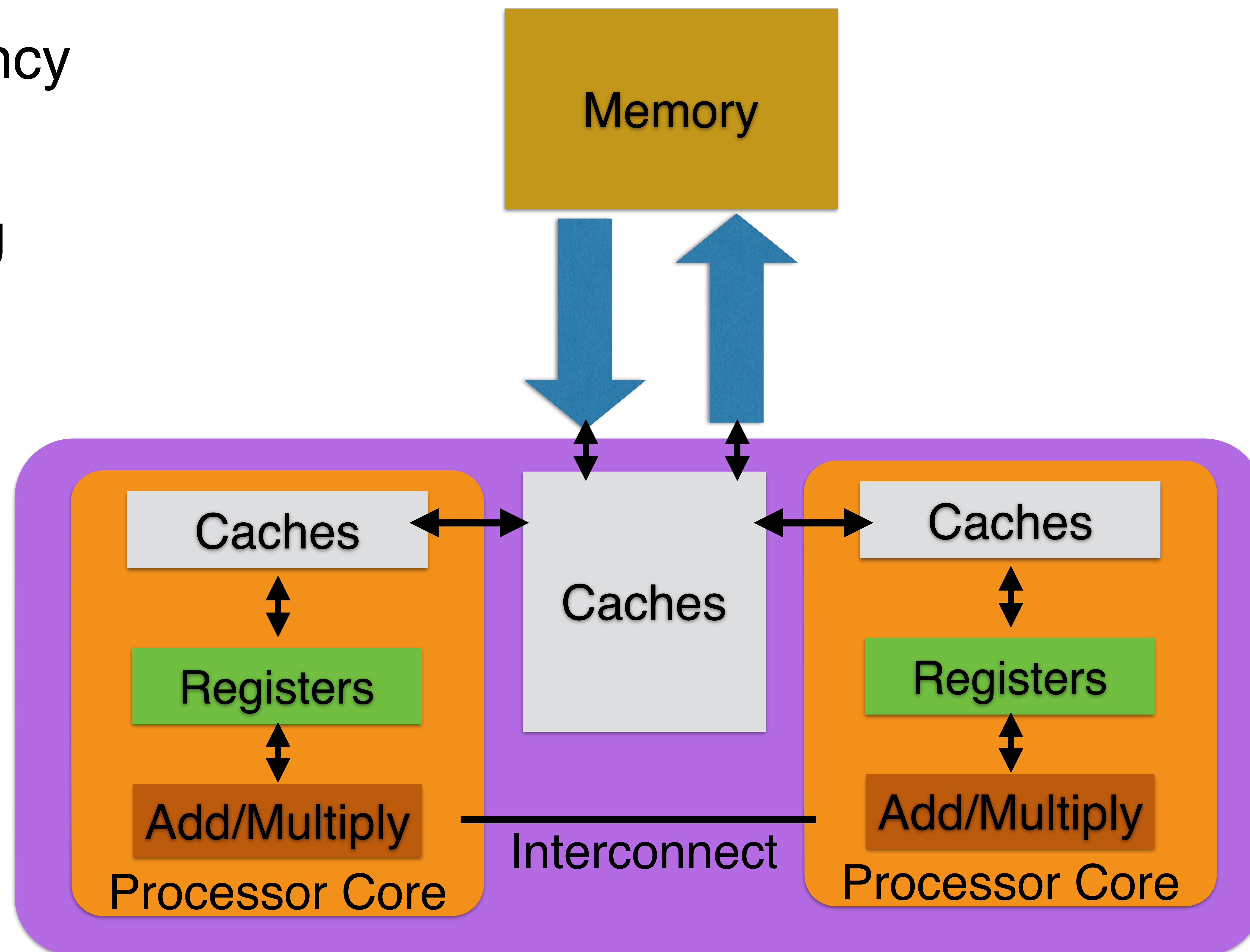
Processor Performance Trends

- The number of transistors is still doubling every 18 months or so
  - Moore's law
- But both clock speeds and single thread performance are flattening
  - have flattened
- Power consumption needs to remain flat
  - only so much power from the wall-plugs
- Really the only way to get higher performance is through a greater number of cores and parallelism



Multi-Core Chips

- Power use grows with clock frequency
  - cannot make clocks faster
- But transistor density is still growing
  - can add more cores
- Multi-core chips
  - replicate cores on silicon
  - often add extra 'shared' cache and on chip interconnect
- Current generations
  - up to 18 cores per chip (Socket)
  - up to 4 sockets per server

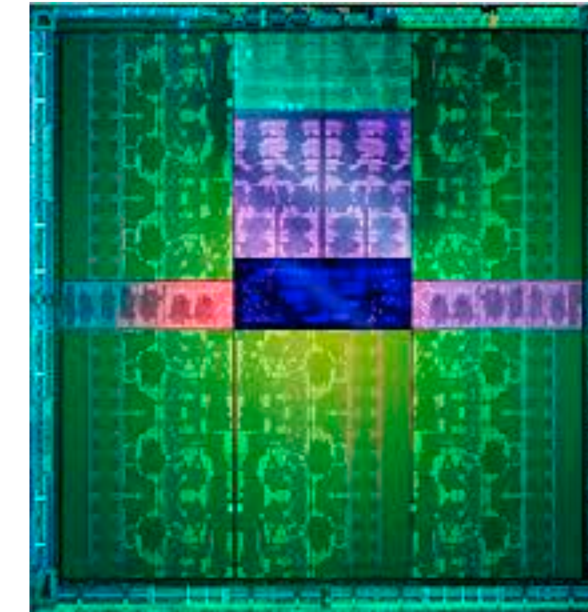


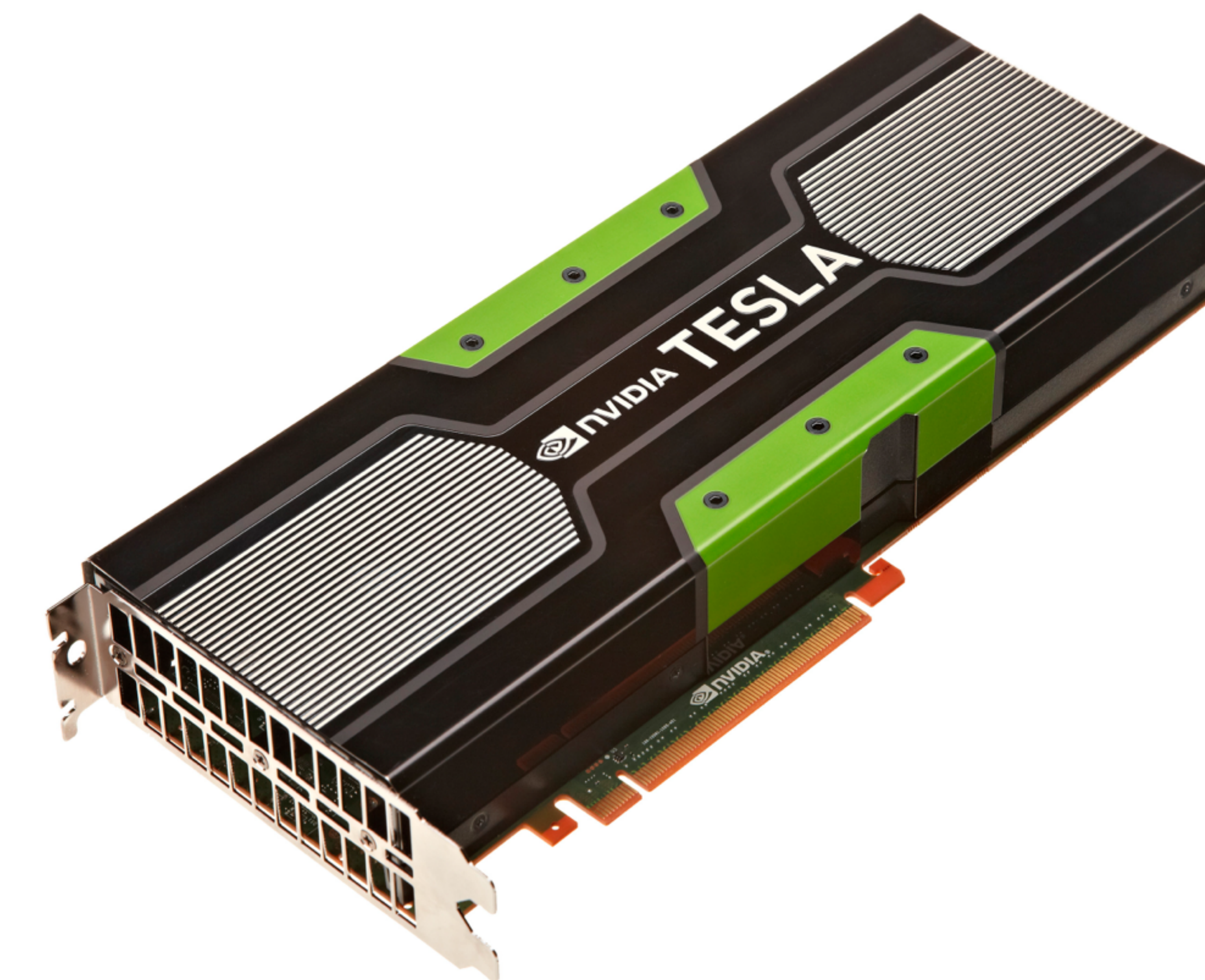
GPUs

- If one wants
  - high floating point throughput
  - at low power (High FLOP/Watt)
- Use the silicon area for more floating point units
  - reduce/eliminate chip real estate regular cores reserve for latency hiding
  - use the space for floating point processors
  - hide latencies through massive parallelism
- NVIDIA Kepler architecture
  - SMX multiprocessing engine: 192 single precision cores, 64 double precision cores

GPUs

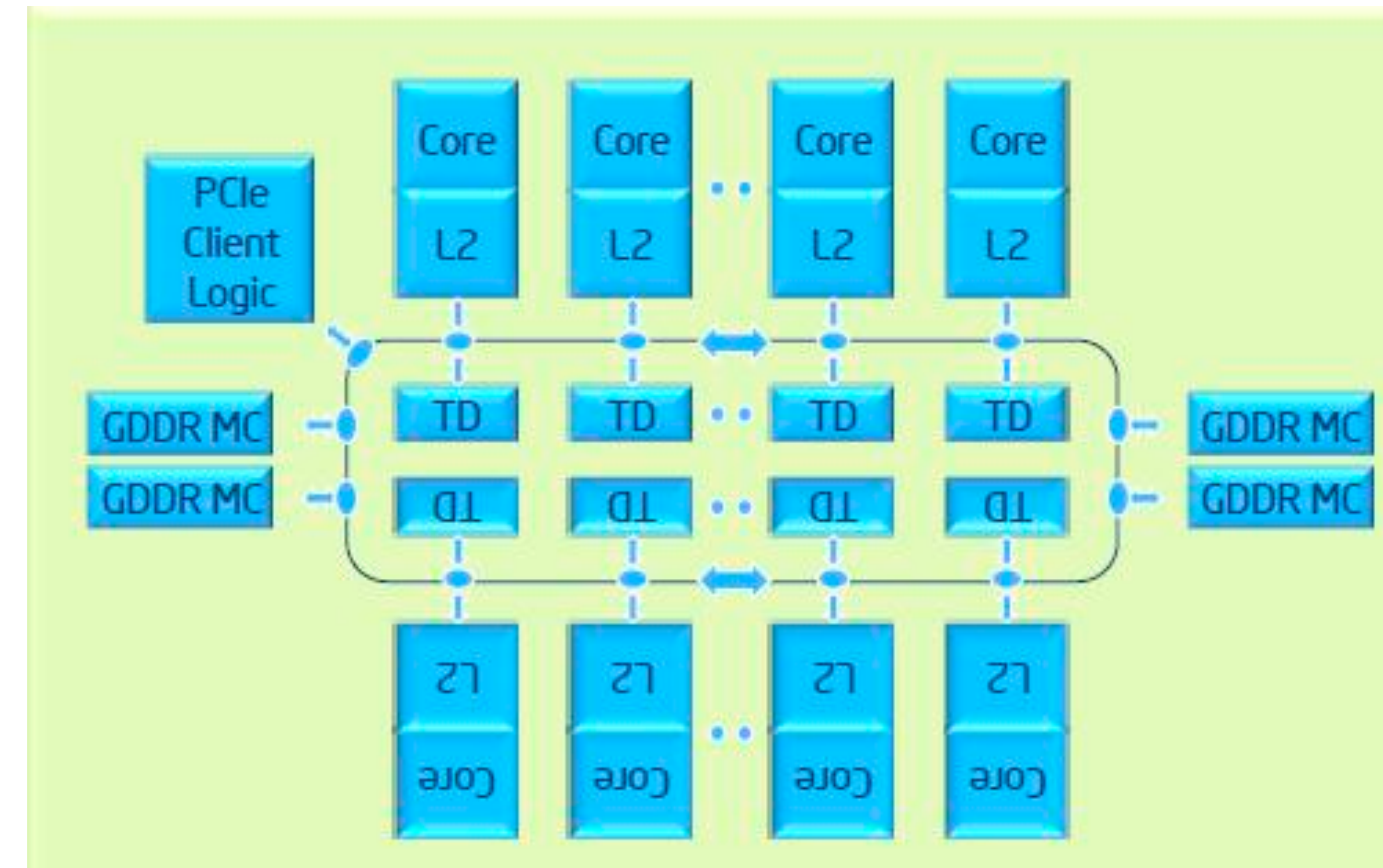
- A GPU Chip is made of 15+ SMXs
  - K20X Peak perf: 1.31 TF (DP), 3.95 TF (SP)
- Chip is packaged as an accelerator 'card'
  - needs a host system to run.
- User code is comprised of many threads
  - oversubscribe the SMXs
  - have threads waiting for SMXs to run
  - if a thread stalls e.g. to wait for data from memory, another bunch of threads is launched on its SMXs





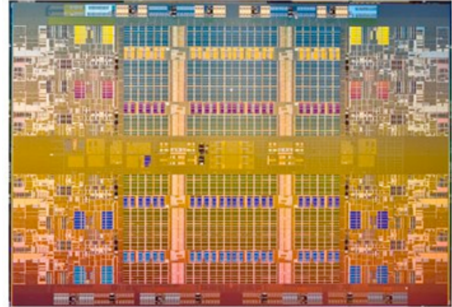
Intel Xeon Phi Architecture

- Another approach to better FLOP/W is the Intel Xeon Phi architecture
  - lots of 'regular' x86 cores on the chip
- Current Generation: Knight's Corner (KNC)
  - 60+ low power (Pentium-like) in-order cores on a chip, at low frequency (~1GHz)
  - Each chip has L1 and L2 caches, and supports 4 threads/core
  - Each chip has a wide vector unit (see later)
    - 2 x 16 SPs FLOP per cycle per chip
    - ~2 TFLOPS (SP), ~1 TFLOPS (DP) per chip
  - packaged as an 'accelerator' card

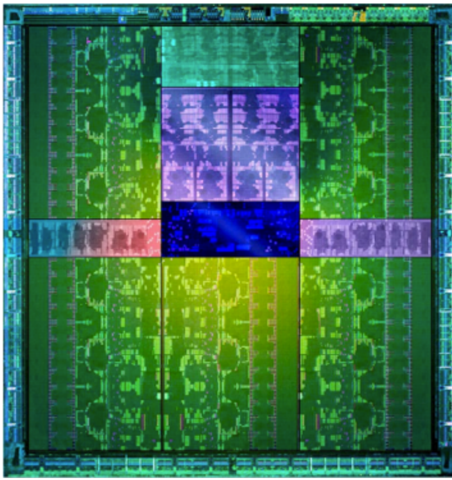


From Chips to Systems

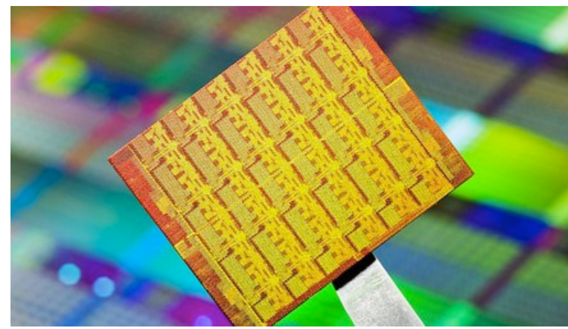
Processing Element



CPU Socket: 4-18 cores x 8(SP)/4(DP) way vectors



GPU: 2880 SP/960 DP CUDA cores



Xeon Phi: 60-61 cores x 16 (SP)/ 8(DP) way vector units

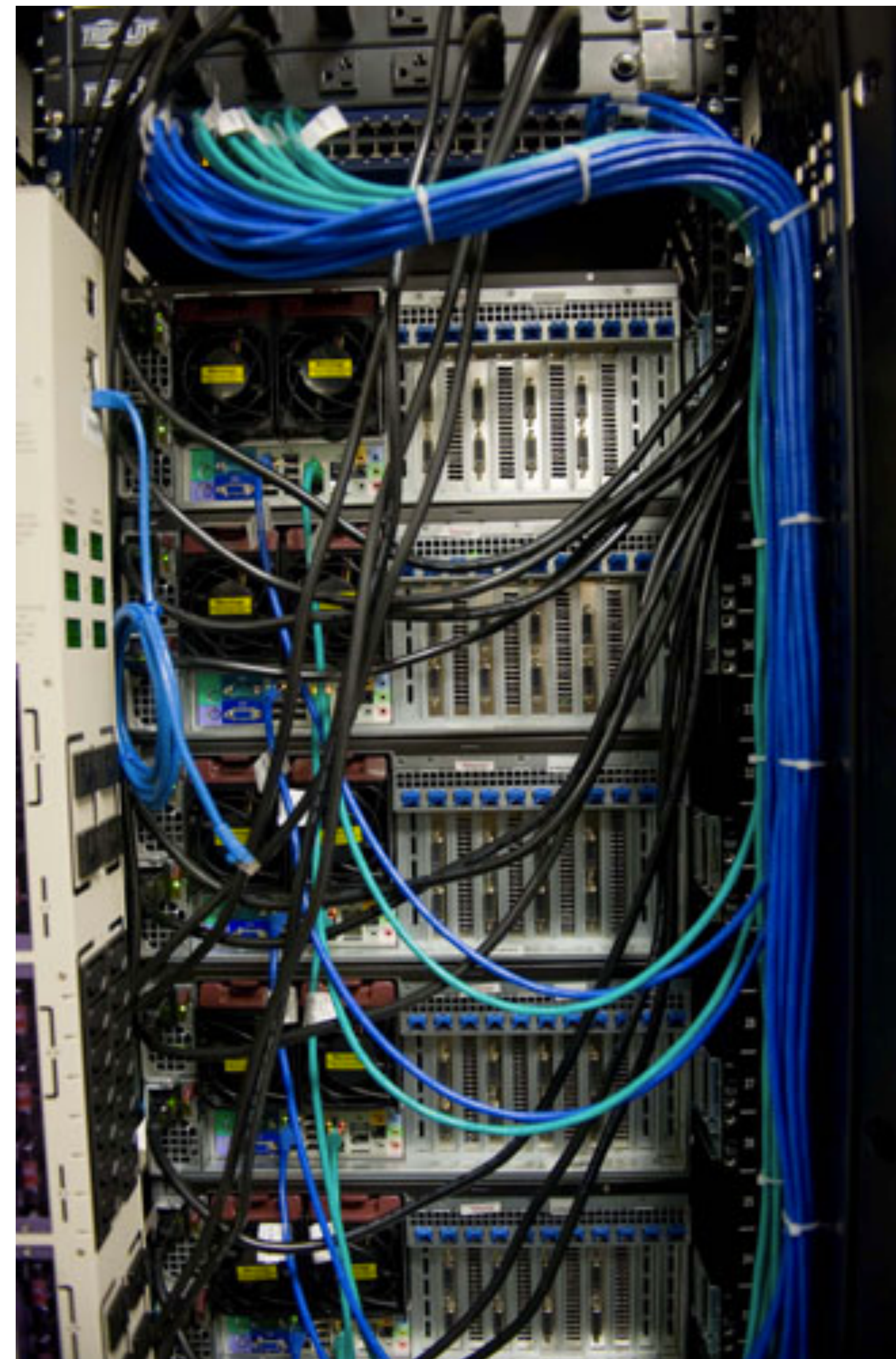
On Chip Parallelism

Node

2-4 sockets, + co-processors (e.g. 4 GPU/Xeon Phi)

On Node Parallelism

Rack



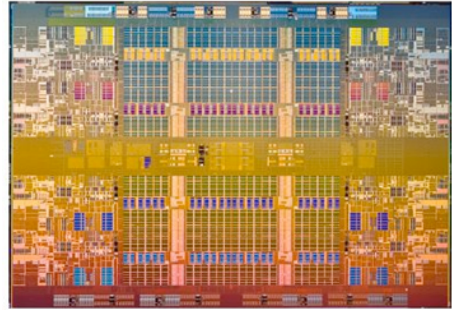
Nodes Connected w. Fabric

System

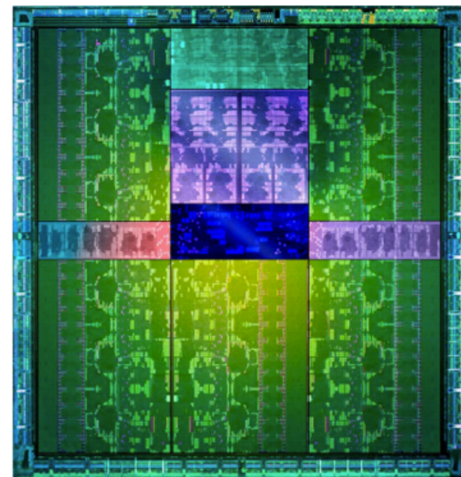
Racks Connected w. Fabric

From Chips to Systems

Processing Element



CPU Socket:16 cores x 8(SP)/4(DP) way vectors



GPU: 2880 SP/960 DP CUDA cores

On Chip Parallelism

2 Node Blade

2x (16 core CPU + GPU)
Memory
Connections for Interconnect

On Node Parallelism

System: Built from Cabinets of 96 nodes (48 blades)

18,688 Nodes Connected w. Fabric
299,008 AMD CPU Cores
18,688 GPUs (17.9M DP CUDA Cores)

New Technology Coming Soon

http://www.anandtech.com/show/7900/nvidia-updates-gpu-roadmap-unveils-pascal-architecture-for-2016

- NVIDIA Pascal GPUs
  - Unified Memory (host & gpu)
  - 3D Memory (high bandwidth)
  - NVLink (high speed interconnect)
  - Will Power Summit Supercomputer at Oak Ridge Leadership Computing Facility

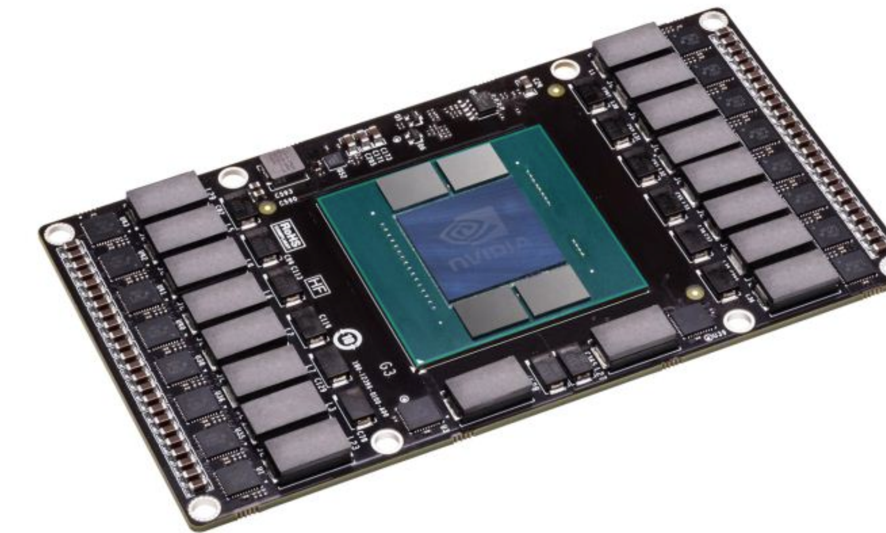


Image courtesy of Oak Ridge National Laboratory
www.ornl.gov

- Intel Xeon Phi Knights' Landing
  - High Performance On Package Memory
  - 60+ Cores based on Intel Atom (Silvermont) with HPC enhancements
  - Will power Cori Supercomputer at NERSC

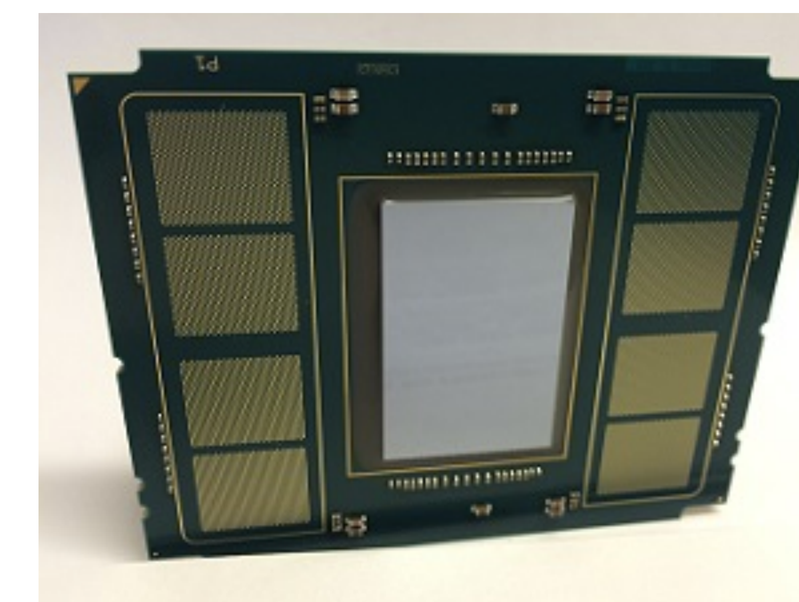
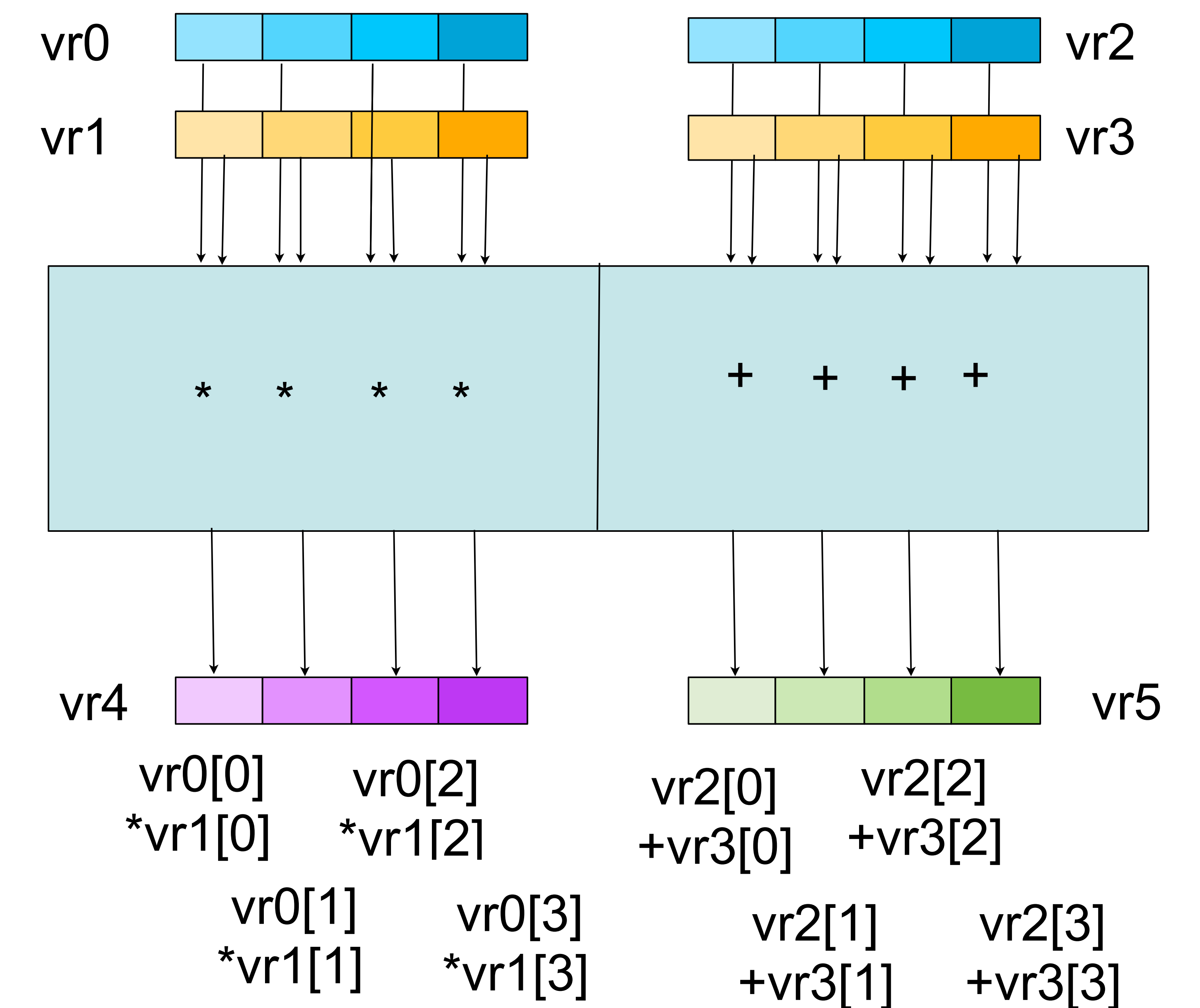


Image courtesy of hpcwire.com

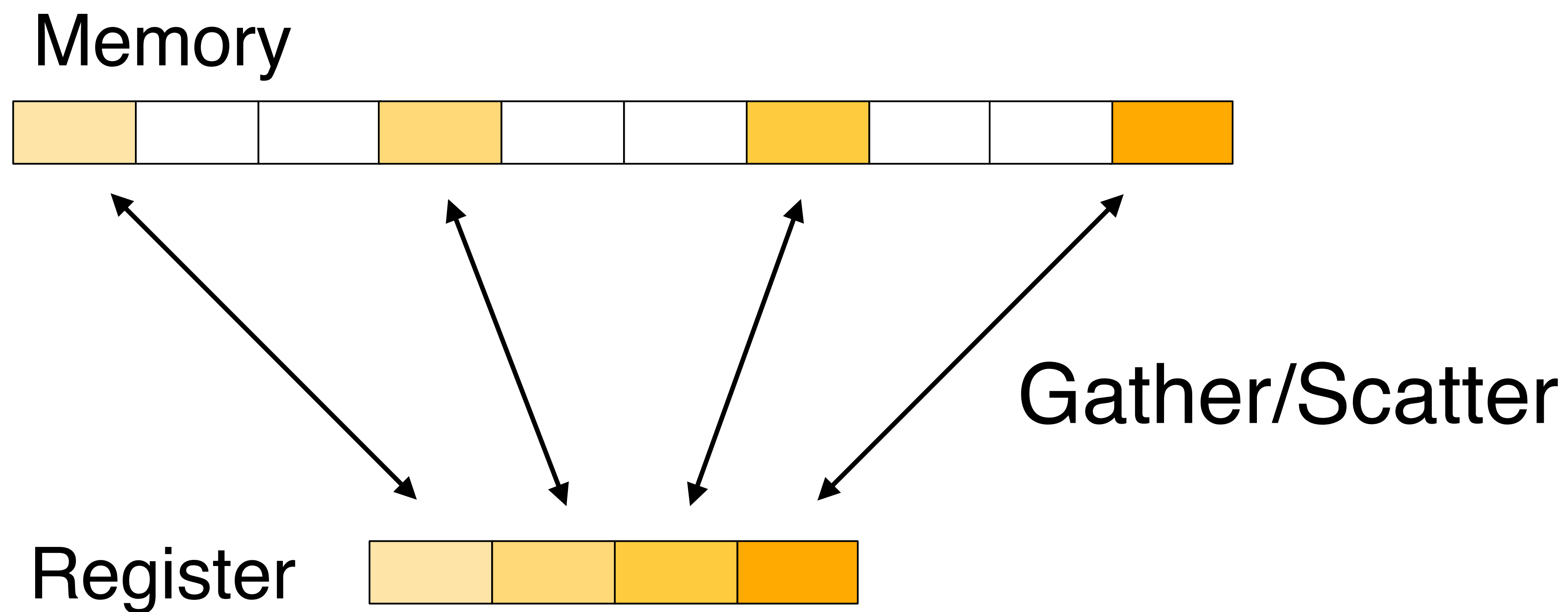
https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing

On-core Vector parallelism

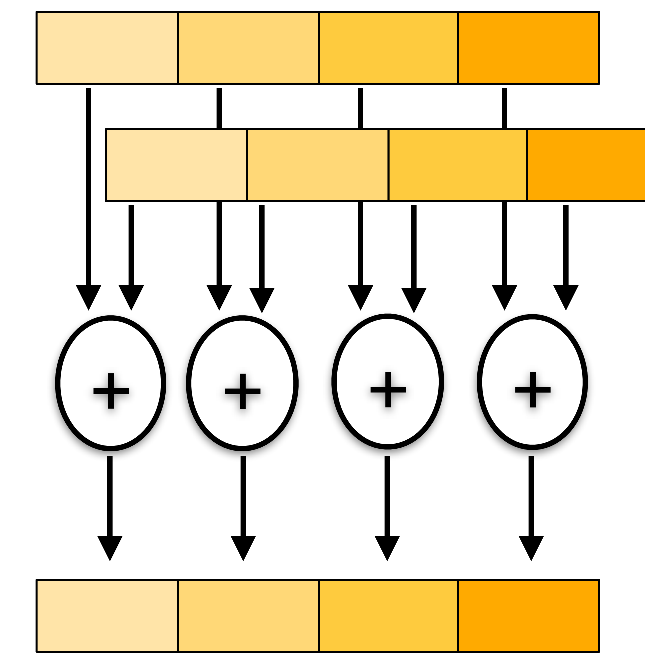
- Modern processors offer ‘Vector’ Parallelism
- Operate on several pieces of data (array) simultaneously
- Length of the vectors:
  - AVX Instructions: 8 single precision, 4 double
  - SSE Instructions: 4 single precision, 2 double
  - Xeon Phi: 16 single precision, 8 double
  - BlueGene/Q: 4 double precision
- GPUs offer “Warps of threads”
  - e.g. 32 threads executing in lock-step
  - very similar to Vector parallelism



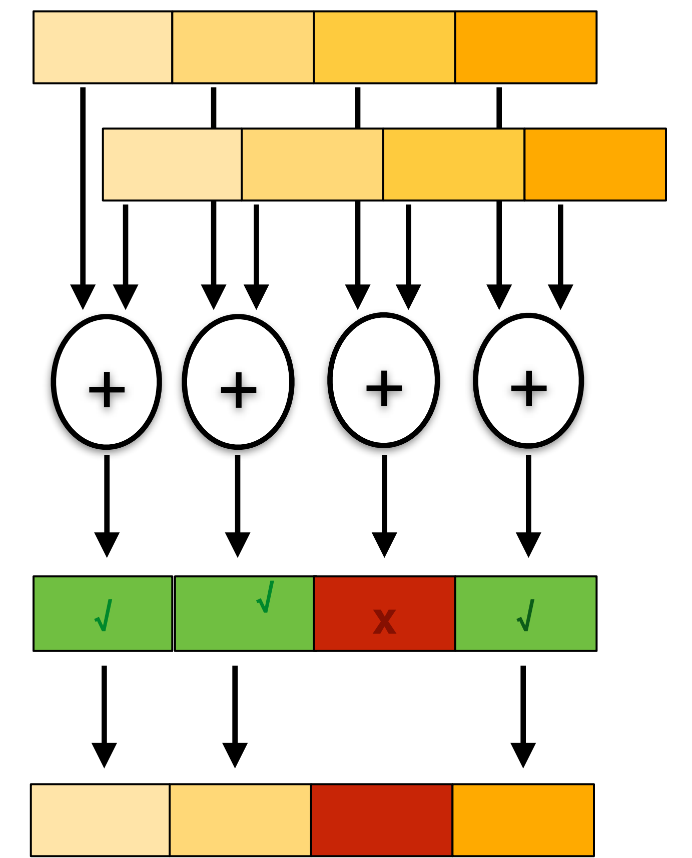
Programming SIMD Vectors



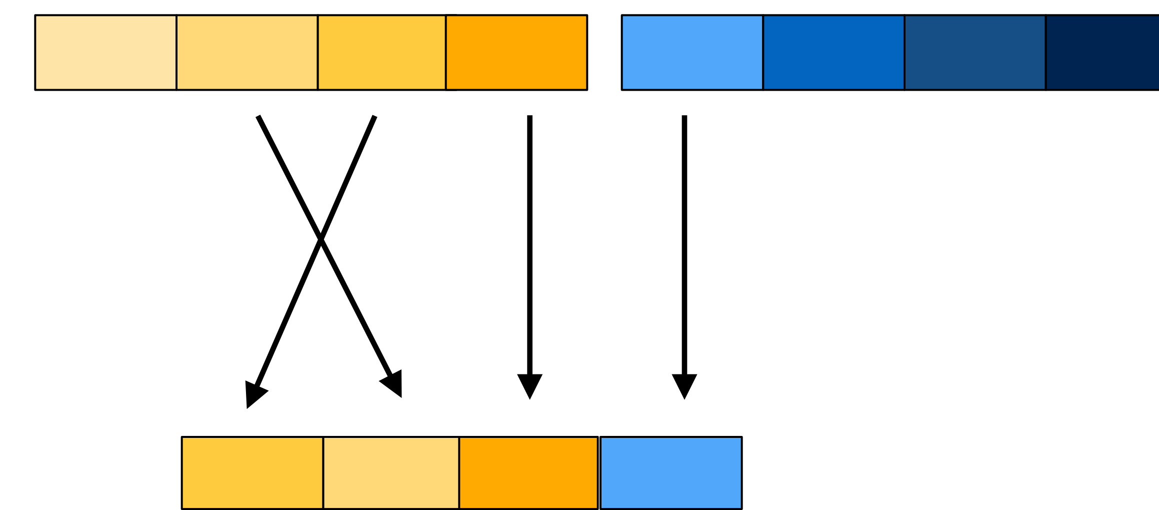
Add/Multiply
MAdd



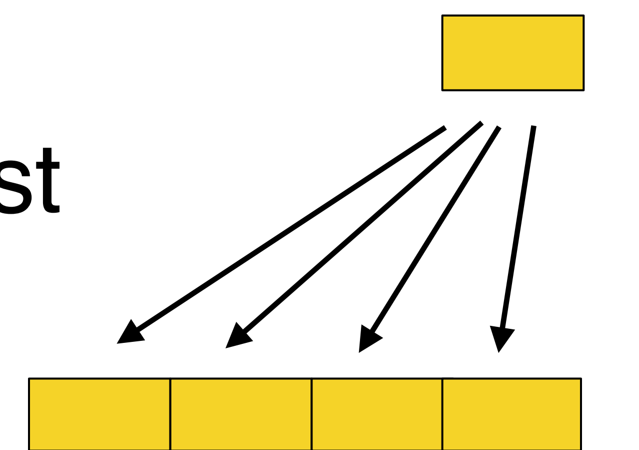
Masked
Variants



Blends
Swizzles
Permutes

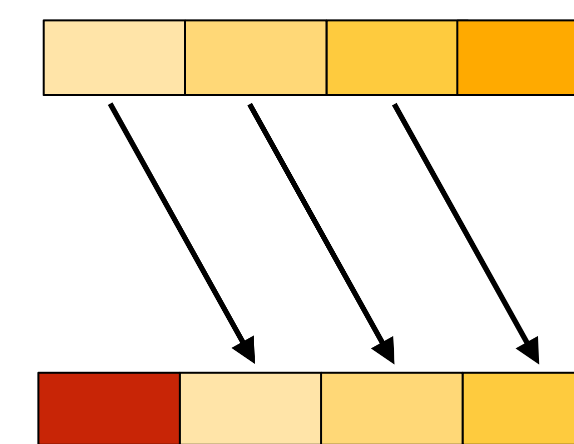


Broadcast

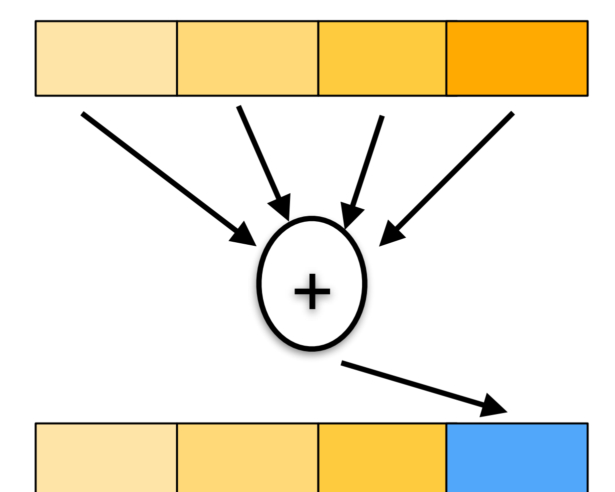


- Variety of vector operations
- Not standard on different hardware
  - e.g. masking in hardware
- Some standards for ‘simple’ vectorization
- To get all the features of a particular system, one may need to turn to compiler intrinsics/assembly

Shifts
&
Rotates



Horizontal
Sum
Reduction

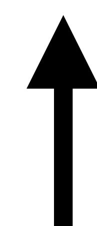


Vectorized AXPY 3 Ways

```
#define N_LARGE 128*1024*1024
#define PREFDIST 128

__declspec(align(16)) float x[N_LARGE];
__declspec(align(16)) float y[N_LARGE];
__declspec(align(16)) float a;
```

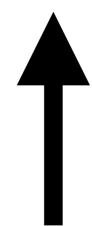
Intel Compiler Intrinsics

```
void axpy_Cilk()
{
  y[:] = a*x[:]+y[:];
}
```



Cilk Extended Array Notation

```
void axpy_OpenMP4()
{
#pragma omp parallel for simd
  for(int i=0; i < N_LARGE;i++) {
    y[i] = a*x[i]+y[i];
  }
}
```



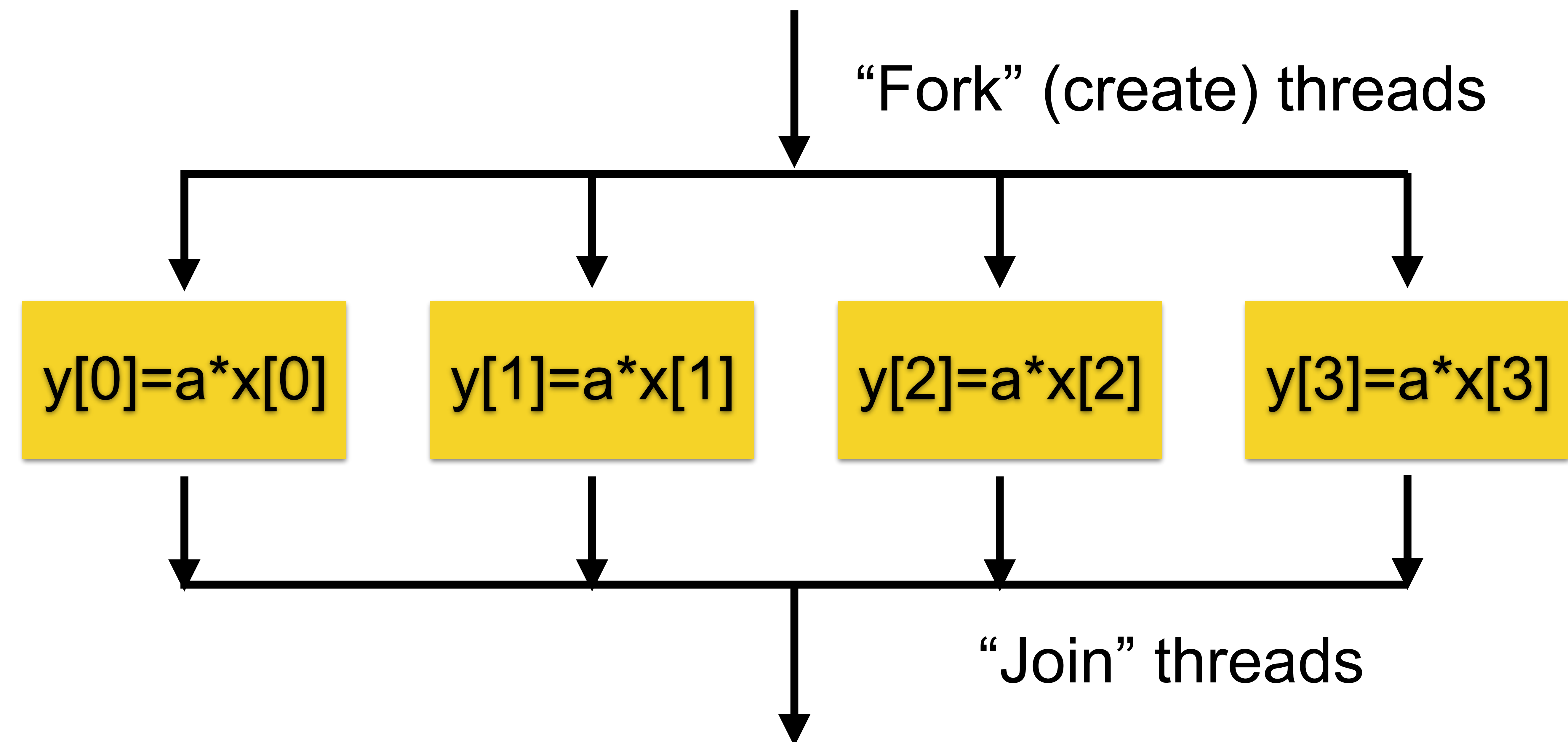
OpenMP 4

```
void axpy_Intrinsic()
{
  __m256 avec=_mm256_broadcast_ss(&a);

  for(int i=0; i < N_LARGE; i+=8) {
    _mm256_store_ps(y+i,
      _mm256_add_ps(
        _mm256_mul_ps(avec,
                      _mm256_load_ps(x+i)),
       _mm256_load_ps(y+i) ) );
  }
}
```

On Node Thread Parallelism

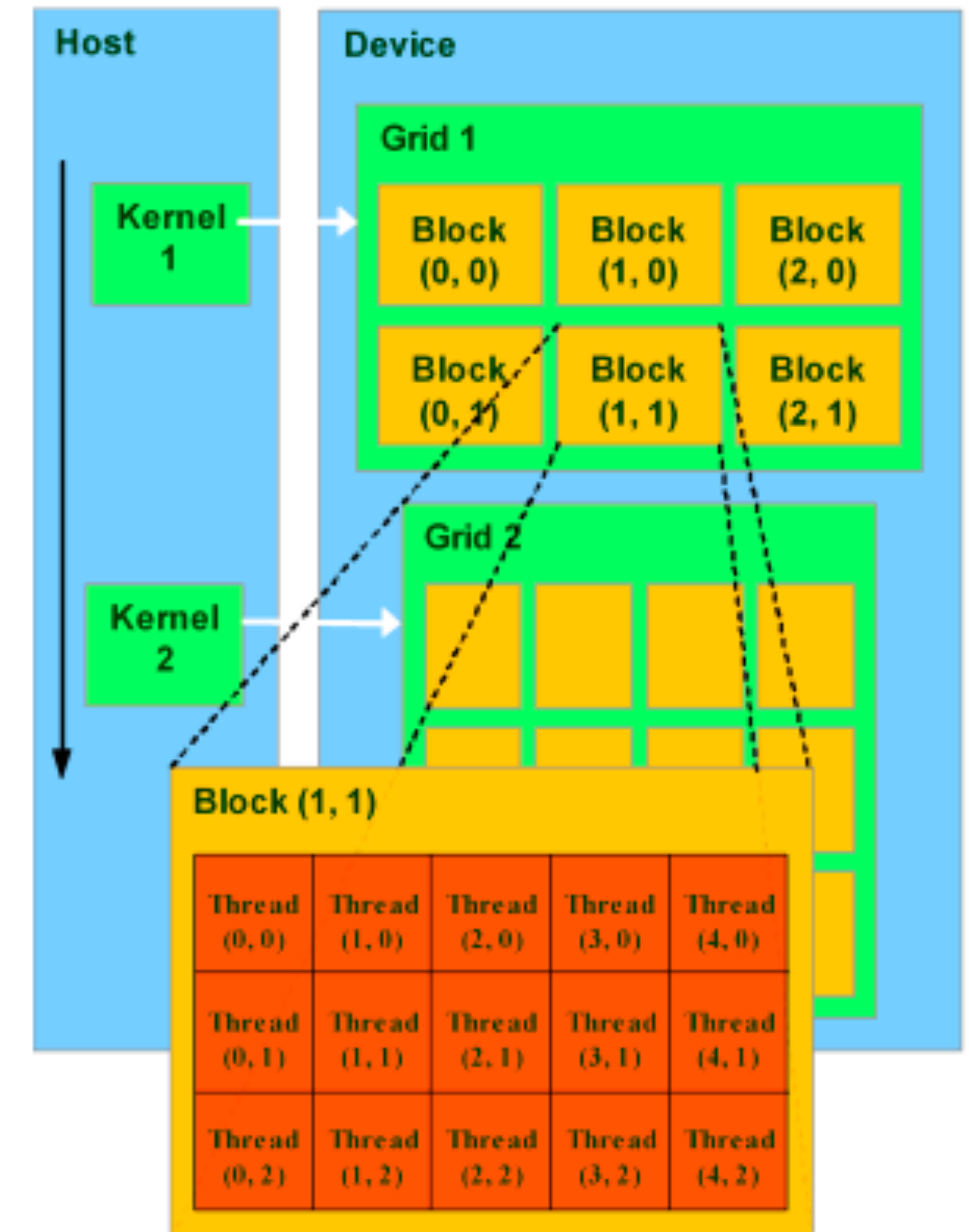
- Concept: several streams of execution ocurring simultaneously
- Common example: fork-join model
- Software Implementation
  - library / language/ compiler support
  - e.g. OpenMP, Pthreads, TBB
- Thread / processor mapping
  - can map threads to separate cores
  - or separate H/W threads in a core
  - usually done by O/S, runtime or driver

```
float y[4];
float x[4] = {0.0,0.1,0.2,0.3};
float a = 2.0;

#pragma omp parallel for
for(int i=0; i < 3; i++) { y[i]= a*x[i]; }
```



Threading on a GPU

- Programmer defines Kernels (units of computing) to run on GPU
- Kernels are launched from the host CPU
- The kernels are defined over ‘blocks’ of threads
- The ‘blocks’ for a kernel are collected into a ‘grid’
  - blocks can have fast synchronization amongst their threads
  - different blocks in a grid cannot synchronize amongst themselves — must be done via the host
  - blocks are assigned to SMX-s
- Arrays for the blocks must be ‘copied’ to GPU



AXPY on a GPU: Way 1

OpenACC

```
void saxpy(int n, float a, float *x, float *y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
```

← Regular Function Definition

← OpenACC #pragma marks code as code for GPU and causes compiler to generate code for the kernel and to copy data on and off the GPU

← Regular Function Call

See: http://devblogs.nvidia.com/parallelforall/six-ways-saxpy/

AXPY on a GPU: Way 2

NVIDIA CUDA

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
…
int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);
cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

__global__ marks this as a CUDA kernel

blockIdx, threadIdx are thread coordinates
blockDim are thread block sizes
CUDA defines these when the kernel is called

copy the data to the GPU from the host

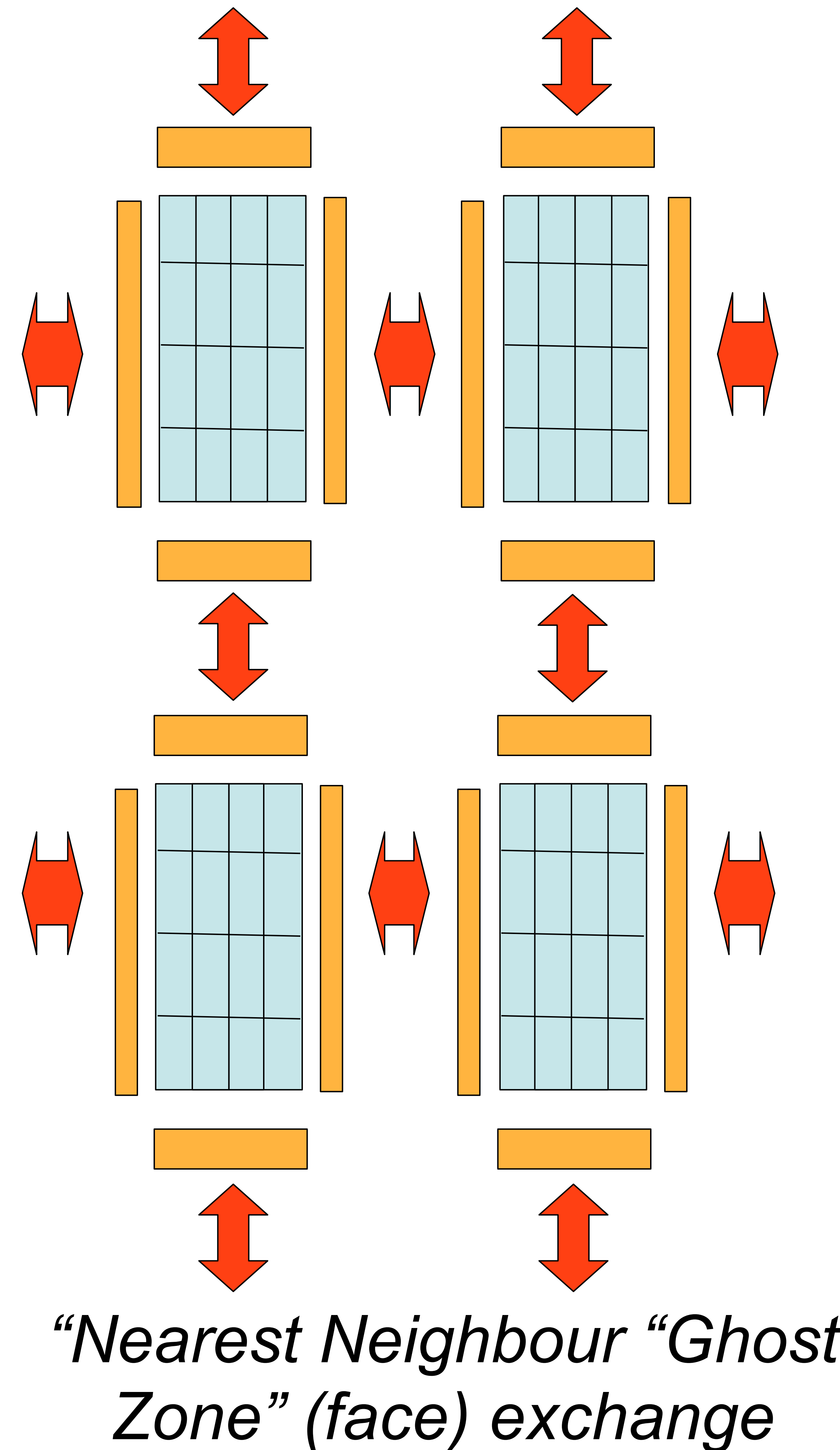
Launch CUDA Kernel: 4096 blocks, 256 threads/block

copy the result from the GPU to the host

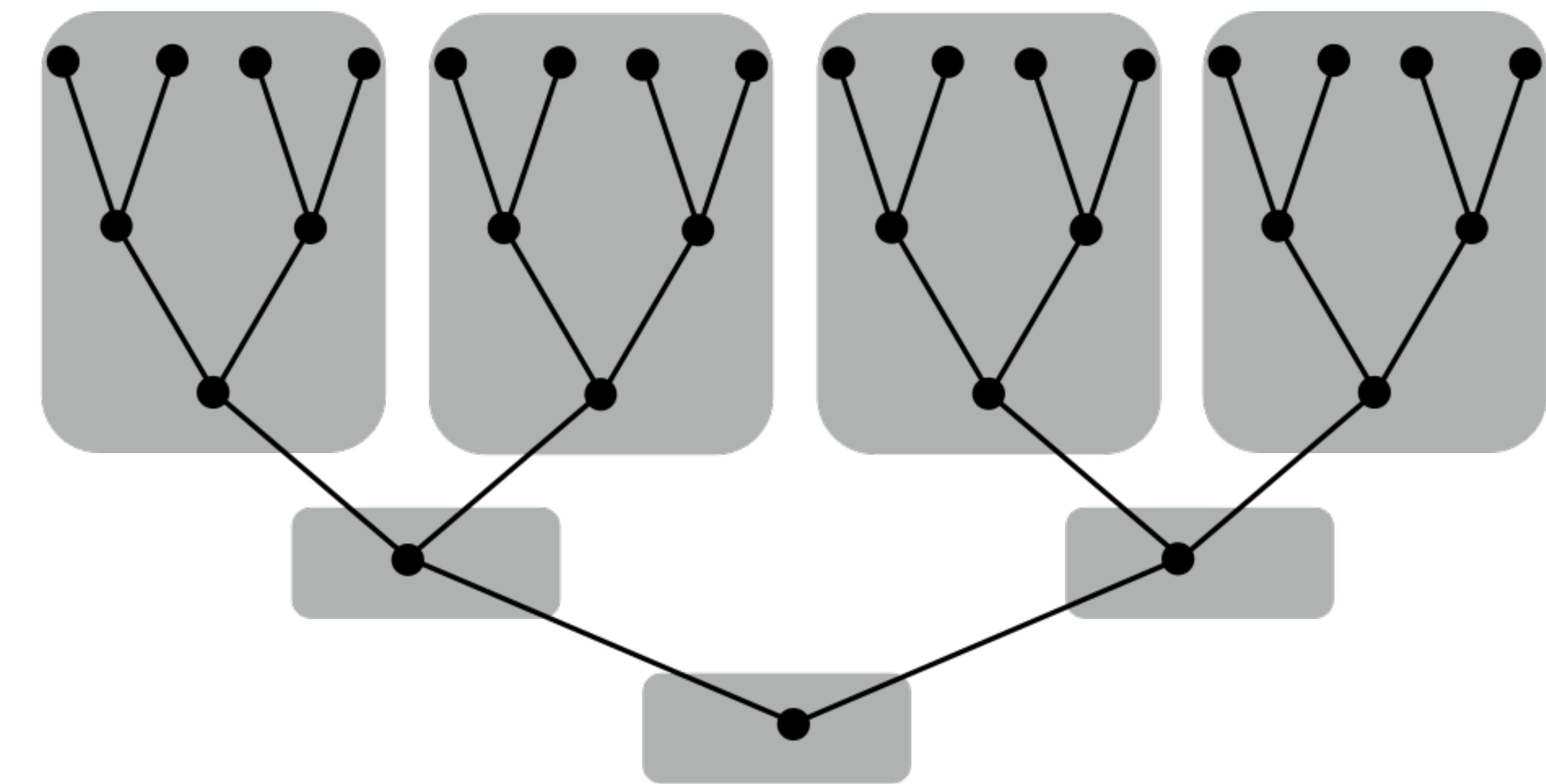
See: http://devblogs.nvidia.com/parallelforall/six-ways-saxpy/

Node Level Parallelism

- Often need to communicate between nodes
  - point-to-point
    - boundary exchange for nearest neighbours
  - global sums/inner products
    - Krylov solvers
  - all-to-all communiations
- In some cases, think of communication as a ‘remote memory access’



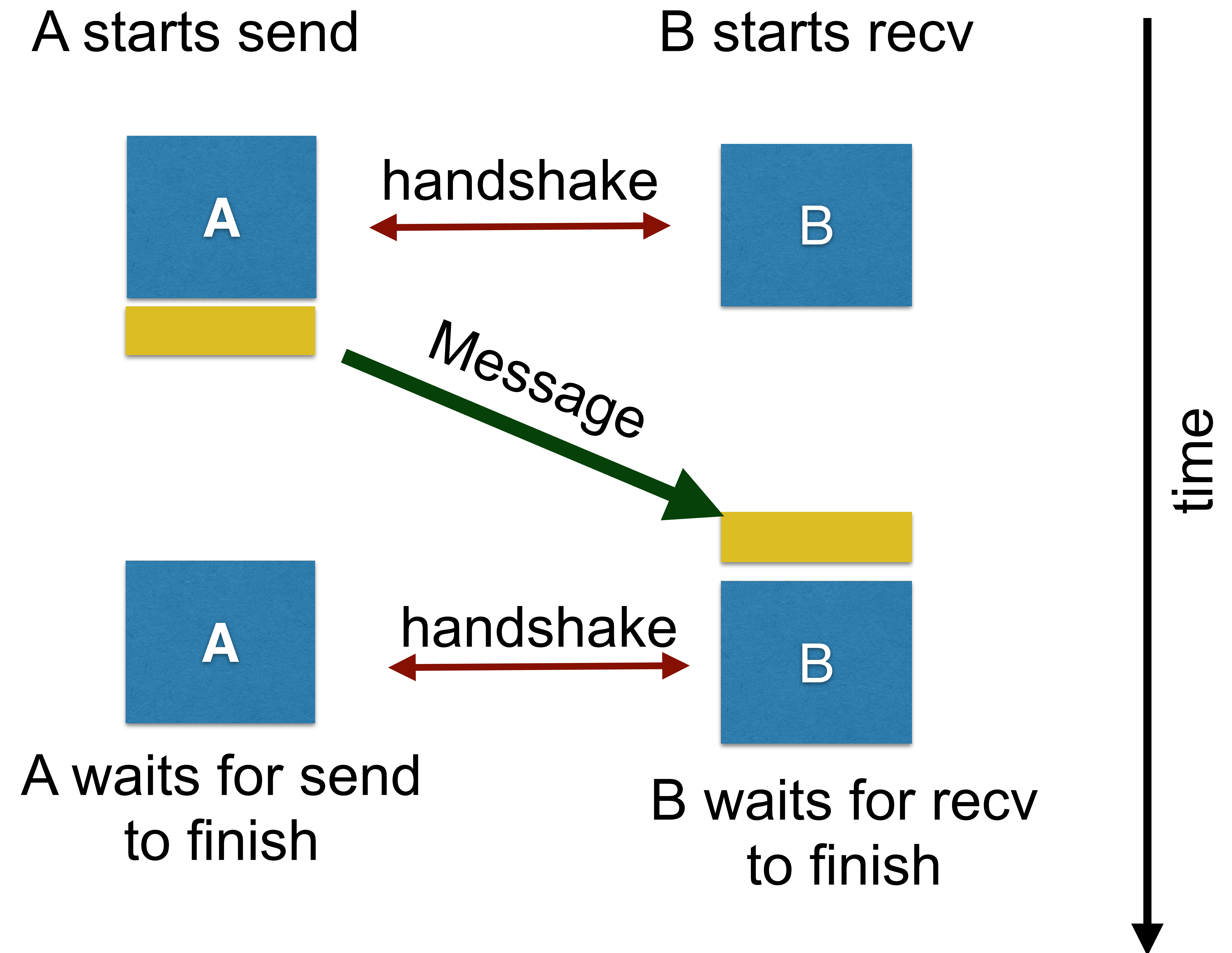
“Nearest Neighbour “Ghost Zone” (face) exchange



Tree Reduction (e.g. sum)

Message Passing

- Exchange Data between a pair of proceses (e.g. on different nodes)
- A sends data and B receives data
- Synchronous
  - Both A and B wait for the send to complete.
  - Analogy: A phone call between A & B



Message Passing

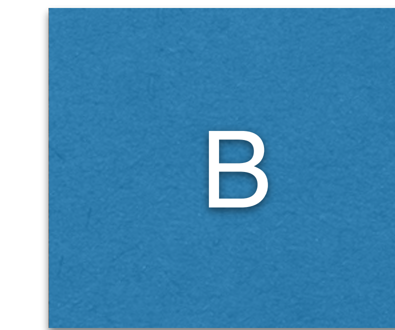
- Exchange Data between a pair of proceses (e.g. on different nodes)
- A sends data and B receives data
- Asynchronous
  - A sends and carries on with other work.
  - B expresses an intent to receive and does other work
  - Eventually B checks/is alerted that a message arrived
  - Analogy: (e)mail between A & B
  - Advantage over synchronous: potentially less time spent waiting

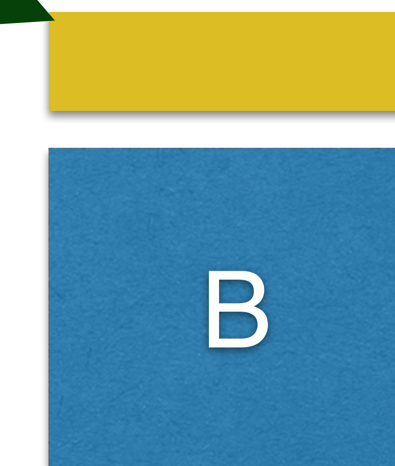
A starts send

A carries on with work

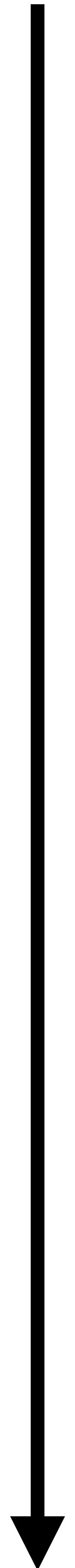
Message

B posts recv and carries on working





B waits for recv to finish



Message Passing with MPI

```
#include <mpi.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  int rank;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int message;
  if (rank==0) {
    message=5;
    std::cout << "Sending Message" << std::endl;
    MPI_Send((const void*)&message, 1, MPI_INT,1,0,MPI_COMM_WORLD);
  }

  if (rank==1) {
    message = 0;
    MPI_Status status;
    MPI_Recv((void *)&message,1,MPI_INT,0,0,MPI_COMM_WORLD, &status);
    std::cout << "Received Message="<< message << std::endl;
  }
  MPI_Finalize();
}
```

Setup: Initialize, get process number

Process 0: Send ‘5’ to process 1

Process 1: Recv Message from process 0

Asynchronous Message with MPI

```
{
  int rank, size;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Request req; MPI_Status status;
  int message;
  if (rank==1) {
    message = 0;
    MPI_Irecv((void *)&message,1,MPI_INT,0,0,MPI_COMM_WORLD, &req);
  }

  if (rank==0) {
    message=5;
    std::cout << "Sending Message" << std::endl;
    MPI_Isend((const void*)&message, 1, MPI_INT,1,0,MPI_COMM_WORLD,&req);
  }

  // -- BOTH PROCESSES CAN DO SOME USEFUL WORK HERE …

  MPI_Wait(&req,&status);
  if( rank == 1) {
    std::cout << "Received: " << message << std::endl;
  }

  MPI_Finalize();
}
```

Setup: Initialize, get process number

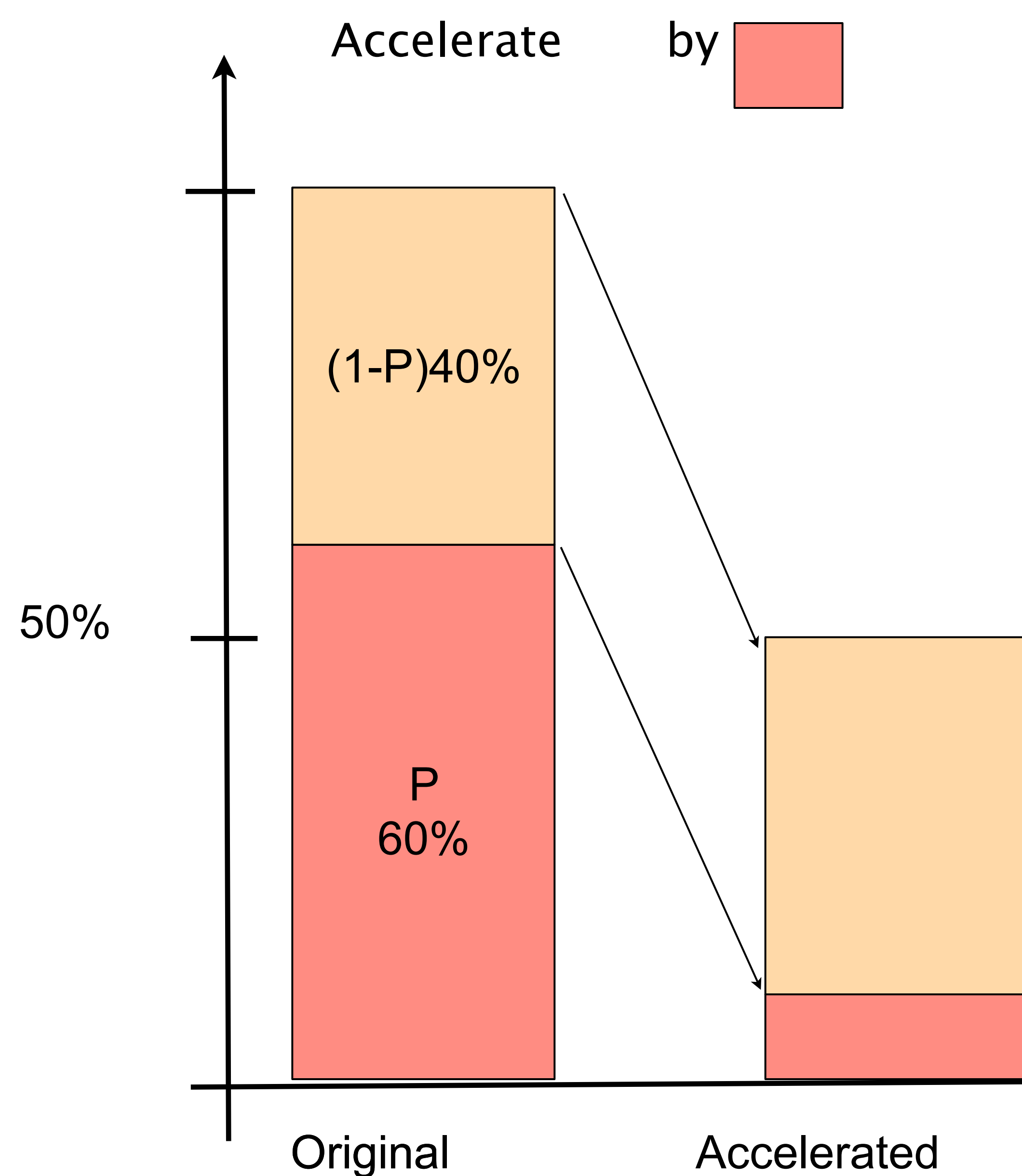
Process 1: Post Intent to receive

Process 0: Post a send

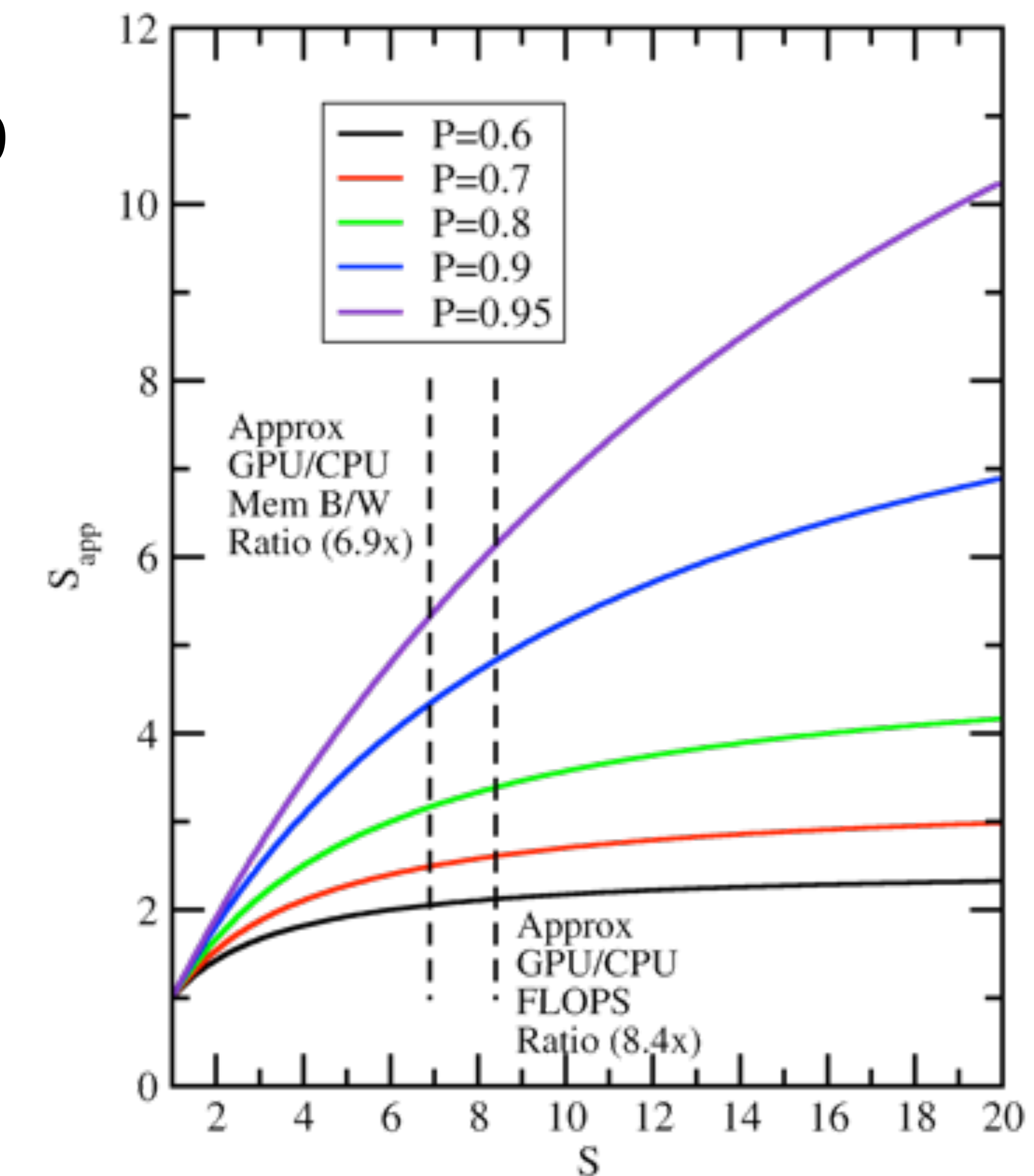
Both Processes wait for their send
or receive to complete

Amdahl's Law

$$S_{\text{app}} = \frac{1}{(1-P) + \frac{P}{S}}$$



- Amdahl's Law
  - How much can parallel programming speed up a problem?
  - Speedup = Optimized Run Time / Original Unoptimized Runtime
  - if you speed up (parallelize/optimize) portion P of your code, overall speedup limited by 1-P portion
  - "what you don't speed up will become your next bottleneck."



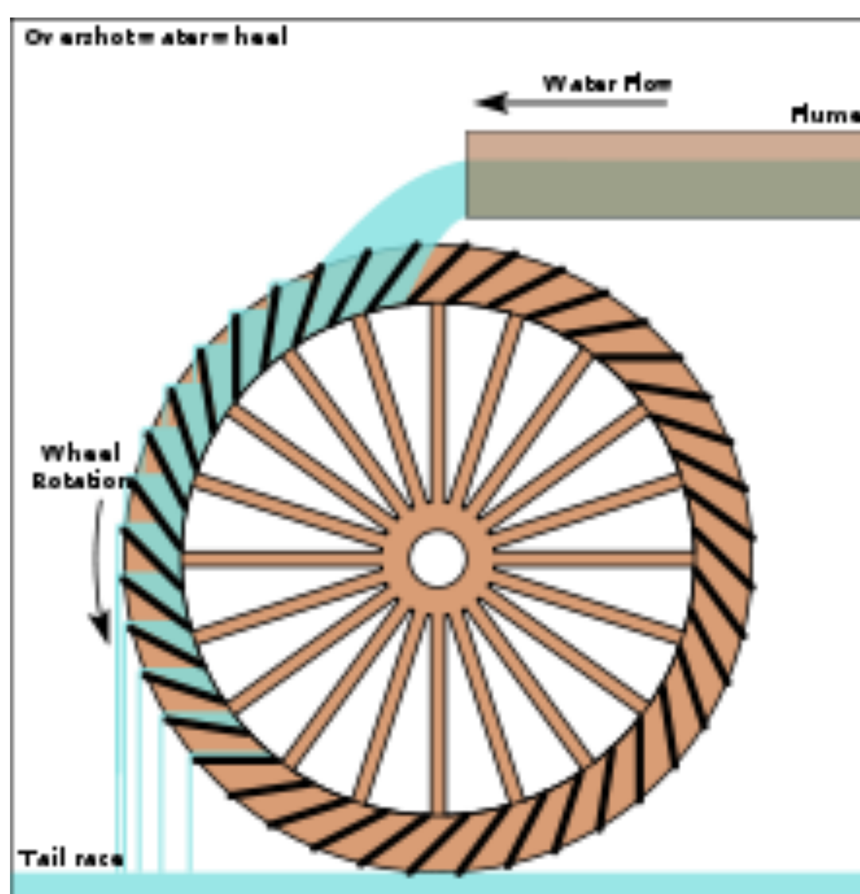
Summary for Part 1

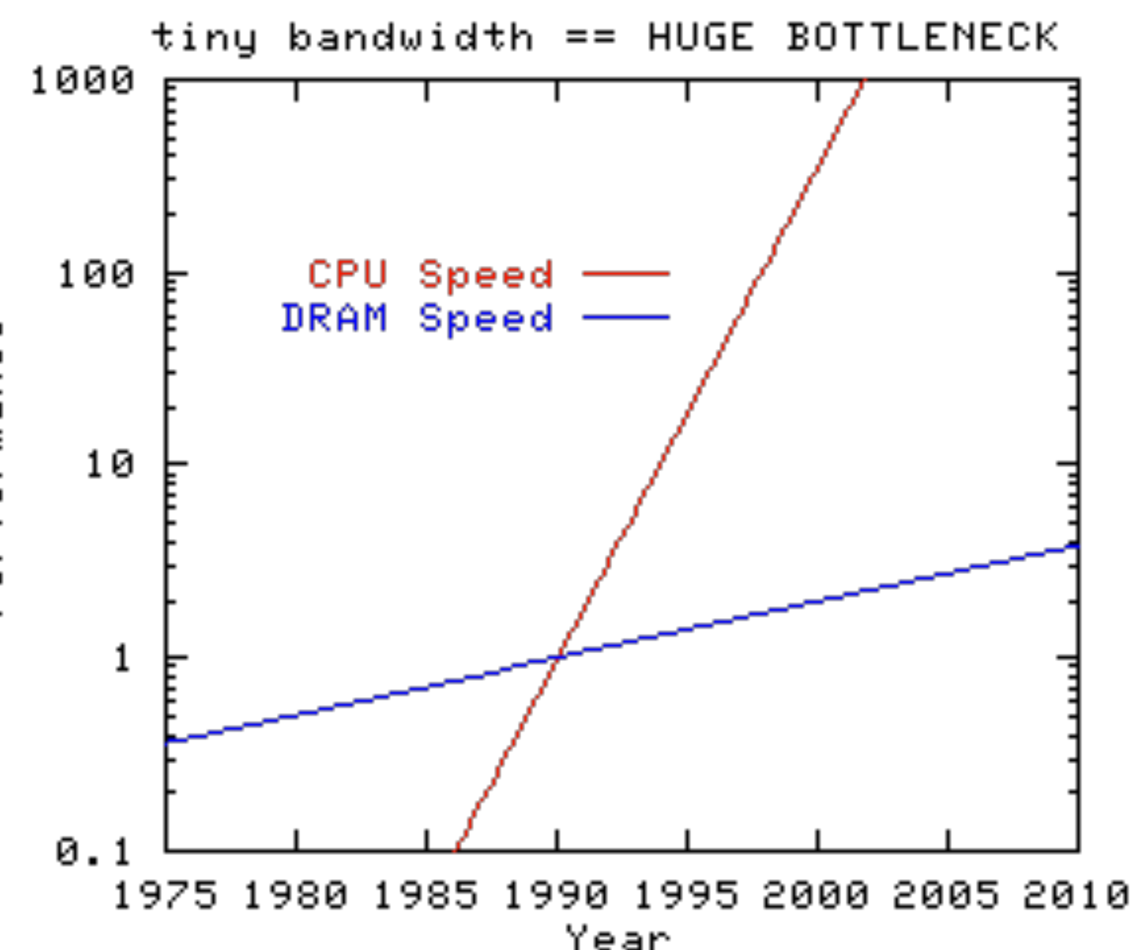
- Discussed High Performance Computing and Computational Science
- Looked at aspects of parallel programming
  - Hardware trends, and parallelism in hardware (multi-core, GPU, Xeon Phi)
  - Building an HPC System
  - vector level parallelism: Intrinsics, OpenMP4 vectorization #pragmas, Cilk Array Notation
  - thread level parallelism: OpenMP, CUDA
  - Internode parallelism: Message Passing and MPI
- Next Lecture: Performance aspects

Part 2

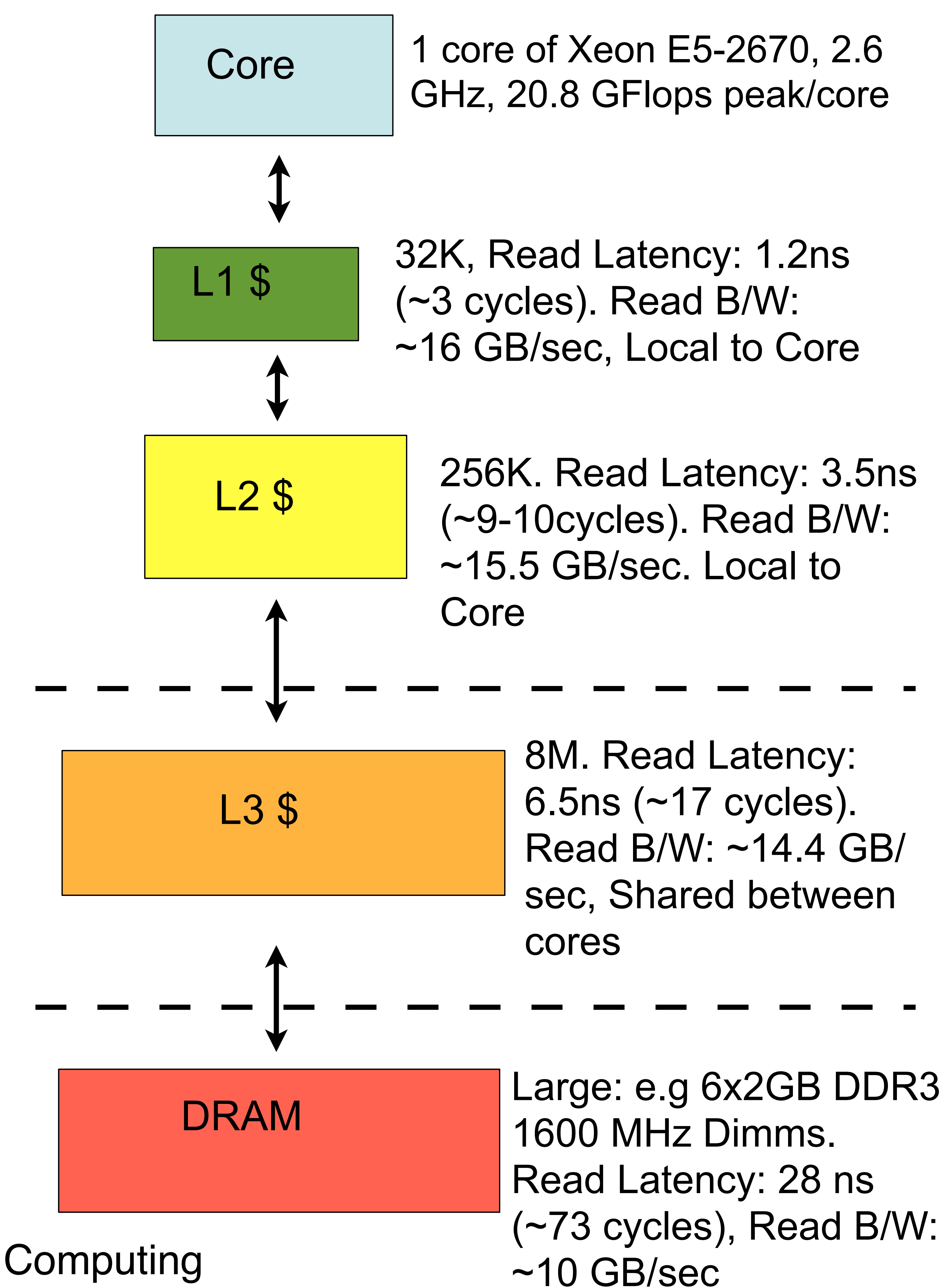
- Memory
- Thinking about performance & bottlenecks

Memory





- Data for the computation needs to come from memory
- Memory speed has not been keeping up with CPU speed historically
- Manage this with a system of caches/ scratchpad memories:
  - high bandwidth low latency memory, to store working set/temporary results
  - for efficiency: organize computation to perform max. no of operations as possible on cached data (reuse)
- Next generation: On-Chip stacked memory
  - dramatic improvements in Bandwidth expected.



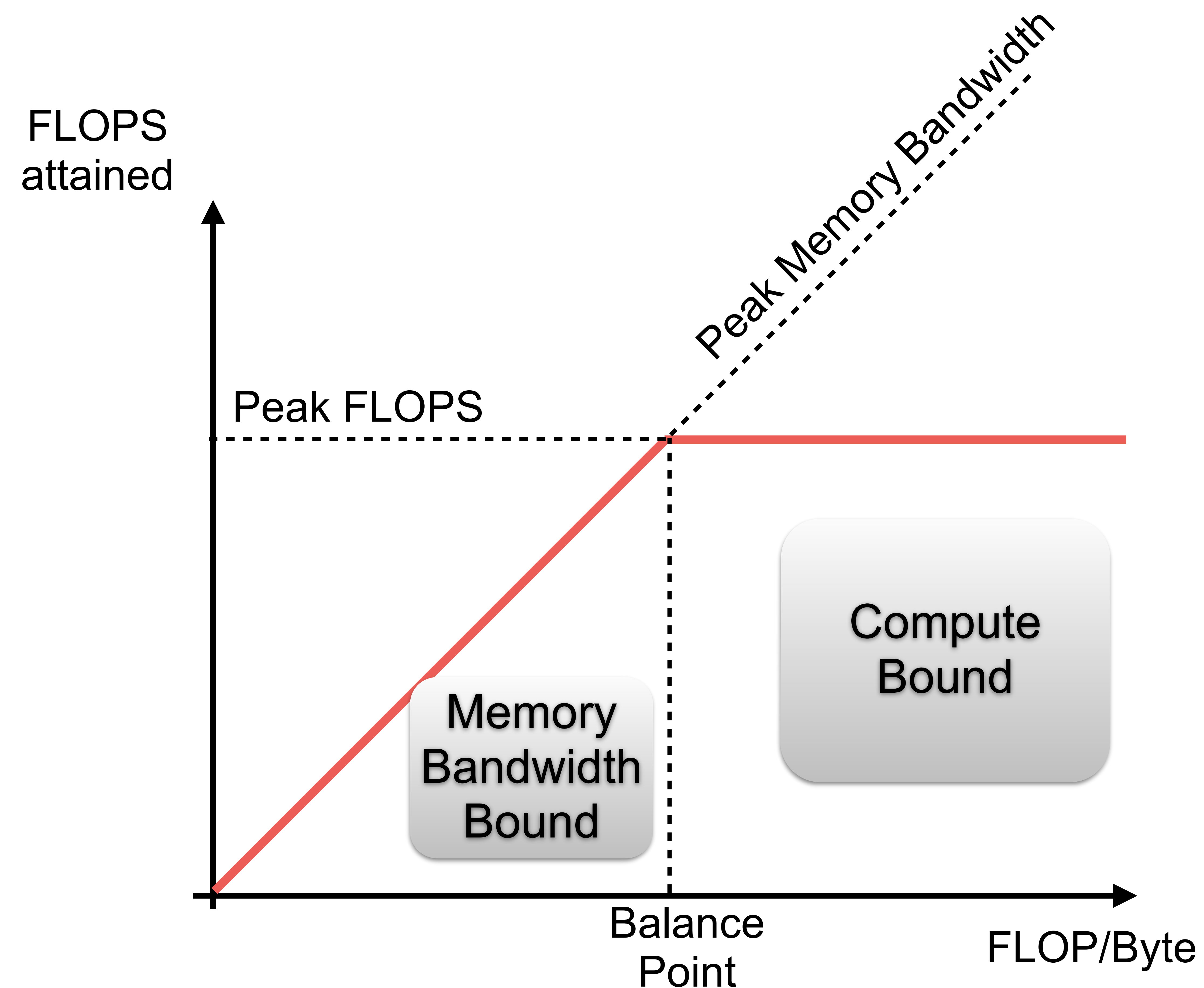
Latencies and bandwidths from: S. Saini, J.Chang, H. Jin, High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation, Lecture Notes in Computer Science Volume 8551, 2014, pp 25-51

Micro-architecture

- Modern CPUs are complex beasts
- They can do many things at once
  - a floating point multiply
  - a floating point add
  - memory loads
  - memory stores
- Exactly what can be done is dictated by microarchitecture.
- “Peak performance” assumes the microarchitcture is working optimally
- Microarchitectural “mishaps” can cause performance degradation
  - pipeline stalls, branch misprediction
  - imbalance in terms of floating point and addition operations etc.

Thinking about on-Chip performance

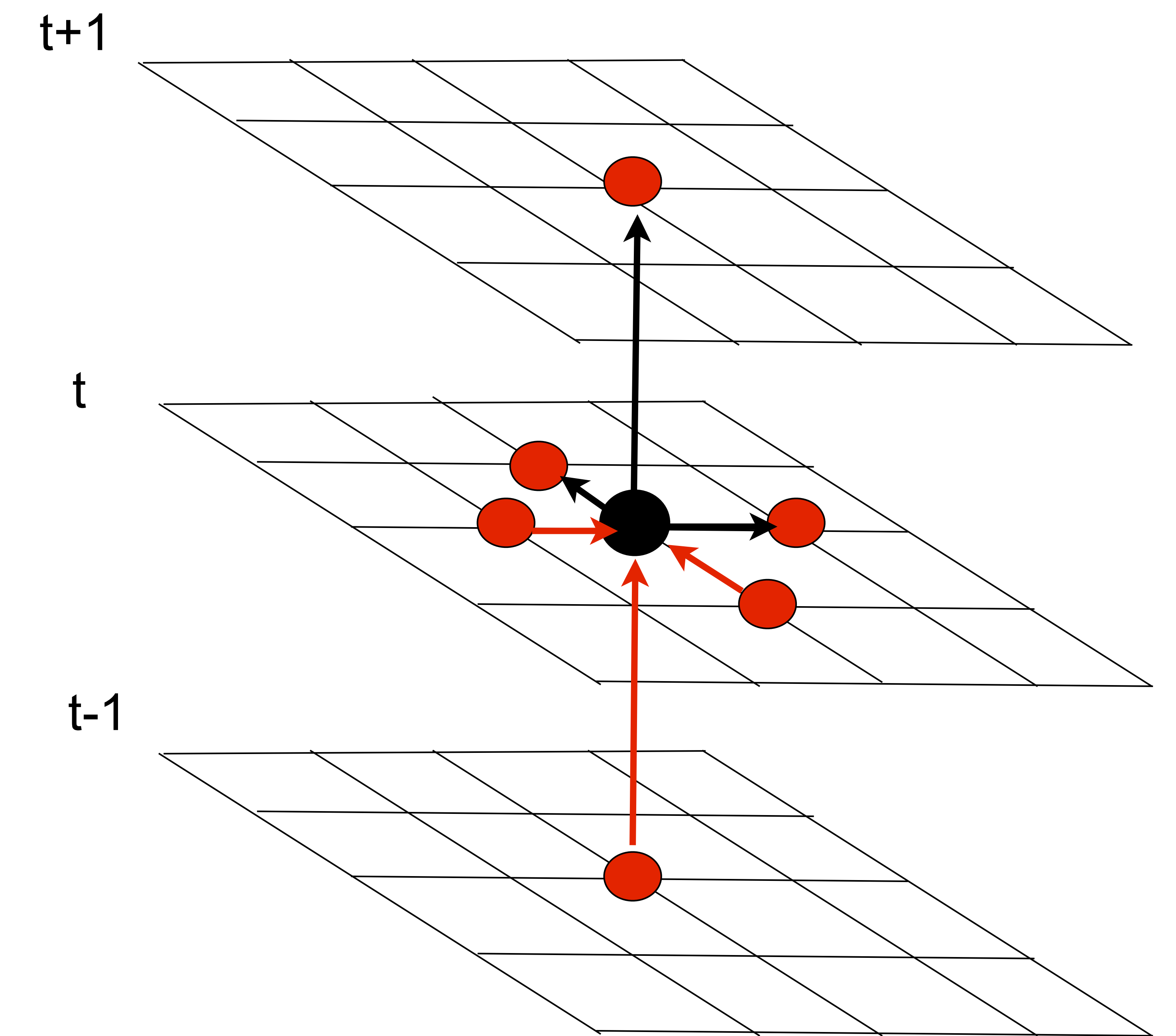
- To solve a problem needs
  - some number of FLOPs (or IOPs)
  - a certain amount of data to work on: Bytes
  - Arithmetic Intensity of problem AI = FLOP/Bytes
- Hardware is capable of supplying
  - some number of FLOP'/s per cycle
  - some memory bandwidth Bytes'/s
  - Balance point of machine B = FLOP'/Bytes'
- If AI < B, problem is memory bound
  - Optimize to maximize memory bandwidth attained
- If AI >= B, problem is compute bound
  - Optimize to maximize FP throughput



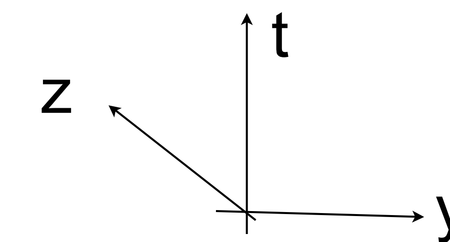
S. Williams, A. Waterman, D. Patterson, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures", Communications of the ACM (CACM), April 2009, doi: 10.1145/1498765.1498785

Example: Wilson Dslash Operator

$$D_{x,y} = \sum_{\mu} \left[(1-\gamma_\mu) U_{x,\mu} \delta_{x+\hat{\mu},y} + (1+\gamma_\mu) U^\dagger_{x-\hat{\mu},\mu} \delta_{x-\hat{\mu},y} \right]$$



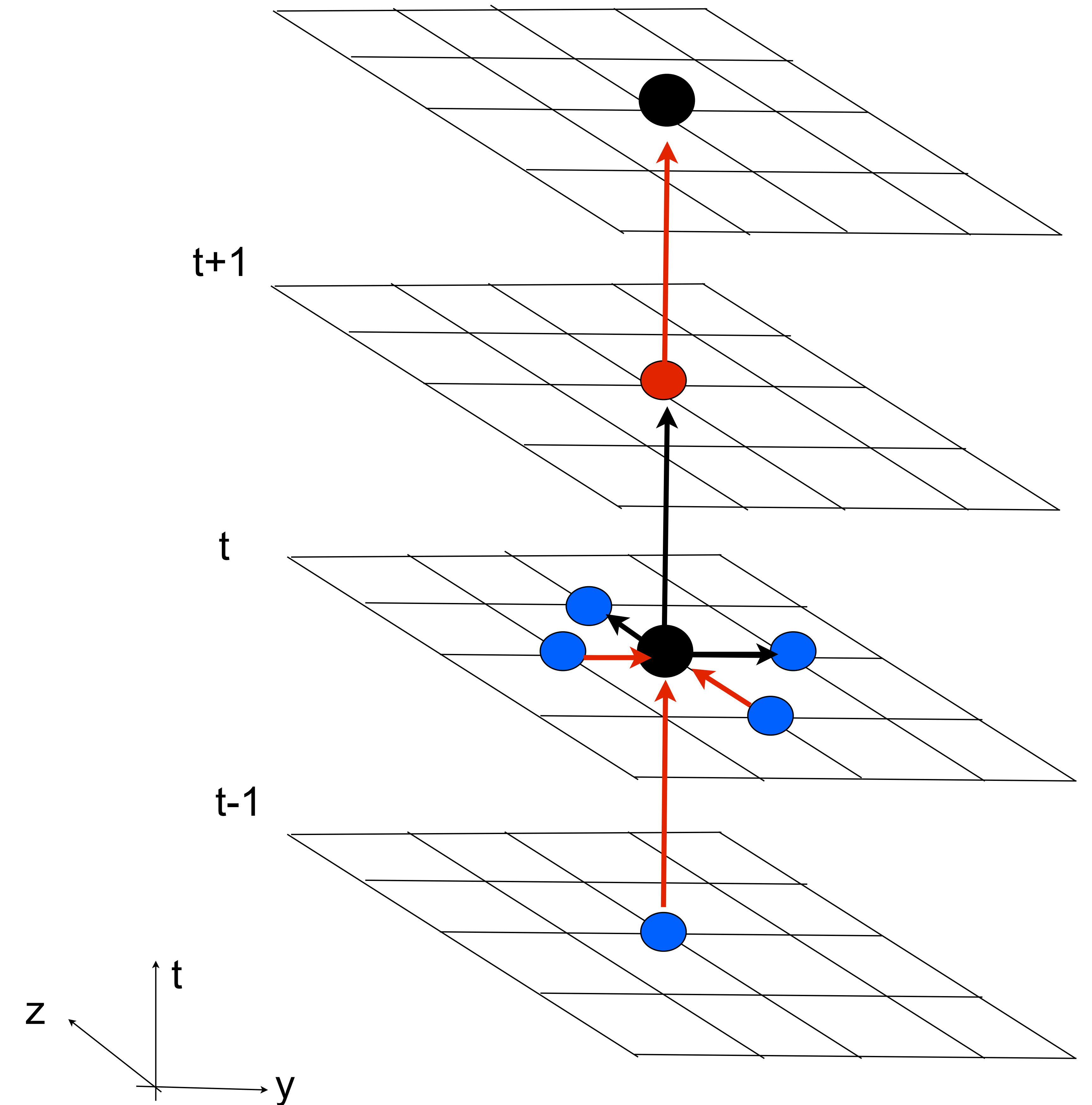
- Key LQCD Kernel: Wilson Dslash Operator
  - U matrices on links. 3x3 Unitary Matrices (complex)
  - spinors on sites: 3x4 complex matrices
  - 9 point stencil in 4-dimensions
    - read 8 neighbours, 8 Us, multiply, write central value
  - Naive Intensity: 0.92 flop/byte (SP), 0.46 flop/byte (DP)



Reuse Potential of D-slash

$$D_{x,y} = \sum_{\mu} \left[(1-\gamma_\mu) U_{x,\mu} \delta_{x+\hat{\mu},y} + (1+\gamma_\mu) U^{\dagger}_{x-\hat{\mu},\mu} \delta_{x-\hat{\mu},y} \right]$$

- By streaming along a direction (e.g. T)
- reuse 7 of 8 neighbouring spinors
- due to even-odd coloring: no reuse of links
  - unless multiple Dslash applications: temporal blocking



M. Smelyanskiy, K. Vaidyanathan, J. Choi, B. Joo, J. Chhugani, M. A. Clark, P. Dubey, "High-Performance Lattice QCD for Multi-core Based Parallel Systems Using a Cache-Friendly Hybrid Threaded-MPI Approach", SC'11

Basic Performance Bound for Dslash

- R = # of reused input spinors
- r = 0 for streaming store
  = 1 for 'read-for-write'.
- B_r = read bandwidth
- B_w = write bandwidth
- G = size of Gauge Link matrix (bytes)
- S = size of Spinor (bytes)

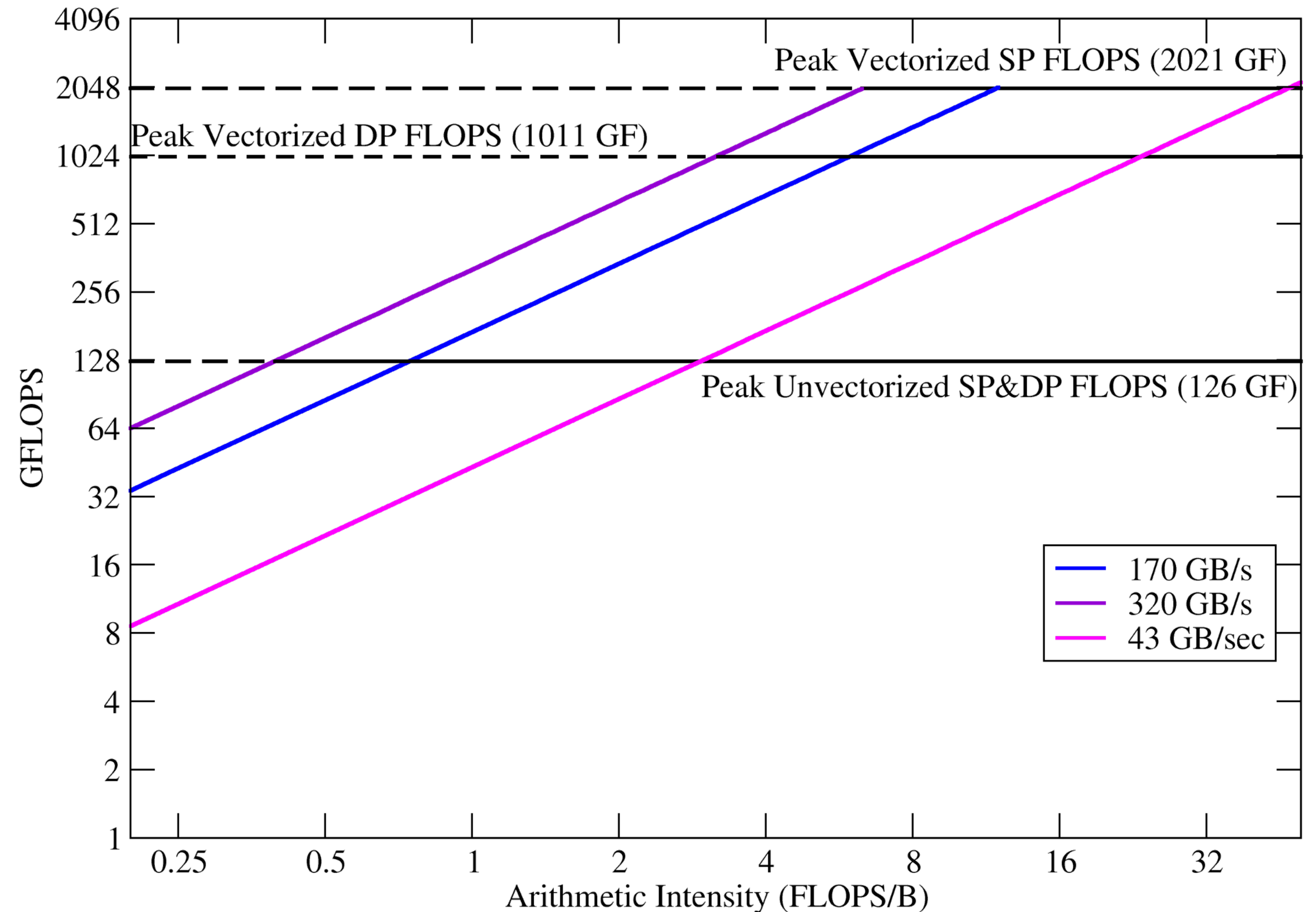
$$F = \frac{1320}{8G/B_r + (8 - R + r)/B_r + S/B_w}$$

Reuse (R)	Streaming Store	Compress	SP FLOPS/B
0	No	No	0.86
0	Yes	No	0.92
0	Yes	Yes	1.06
7	Yes	No	1.72
7	Yes	Yes	2.29

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ \mathrm{x} & \mathrm{x} & \mathrm{x} \end{pmatrix} \xrightarrow{\begin{aligned} \mathbf{a} &= (a_1, a_2, a_3) \\ \mathbf{b} &= (b_1, b_2, b_3) \\ \mathbf{c} &= (\mathbf{a} \times \mathbf{b})^* \end{aligned}} \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix}$$

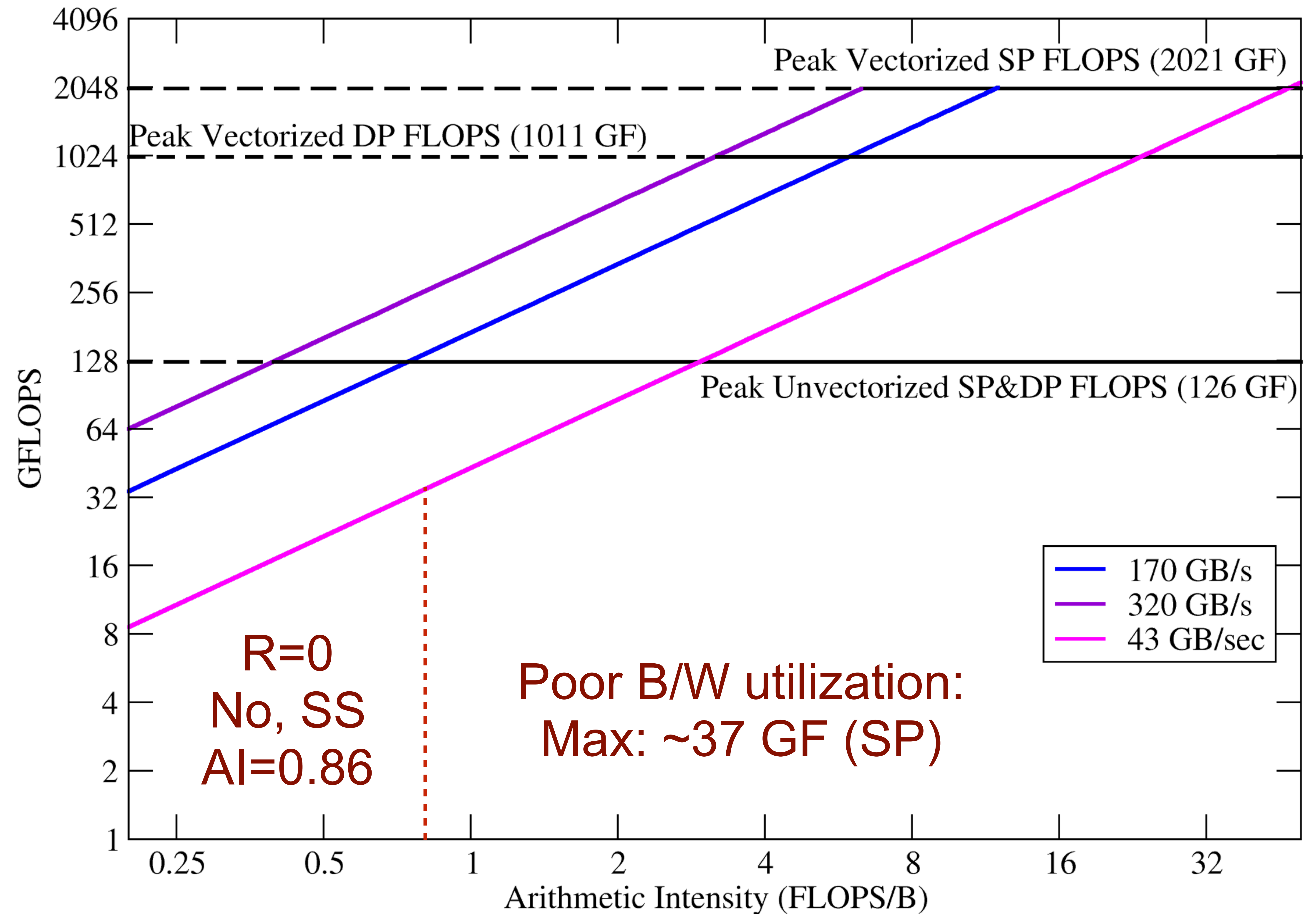
Performance limits for Xeon Phi KNC

- Peak Mem Bandwidth is quoted as 320 GB/s
- In reality streams achieves around 150-170 GB/sec
- “Unvectorized” peak assumes Vector Unit is used (but only 1 lane)
  - ie. 2 FLOP/cycle
  - same ‘unvectorized peak’ for DP as SP



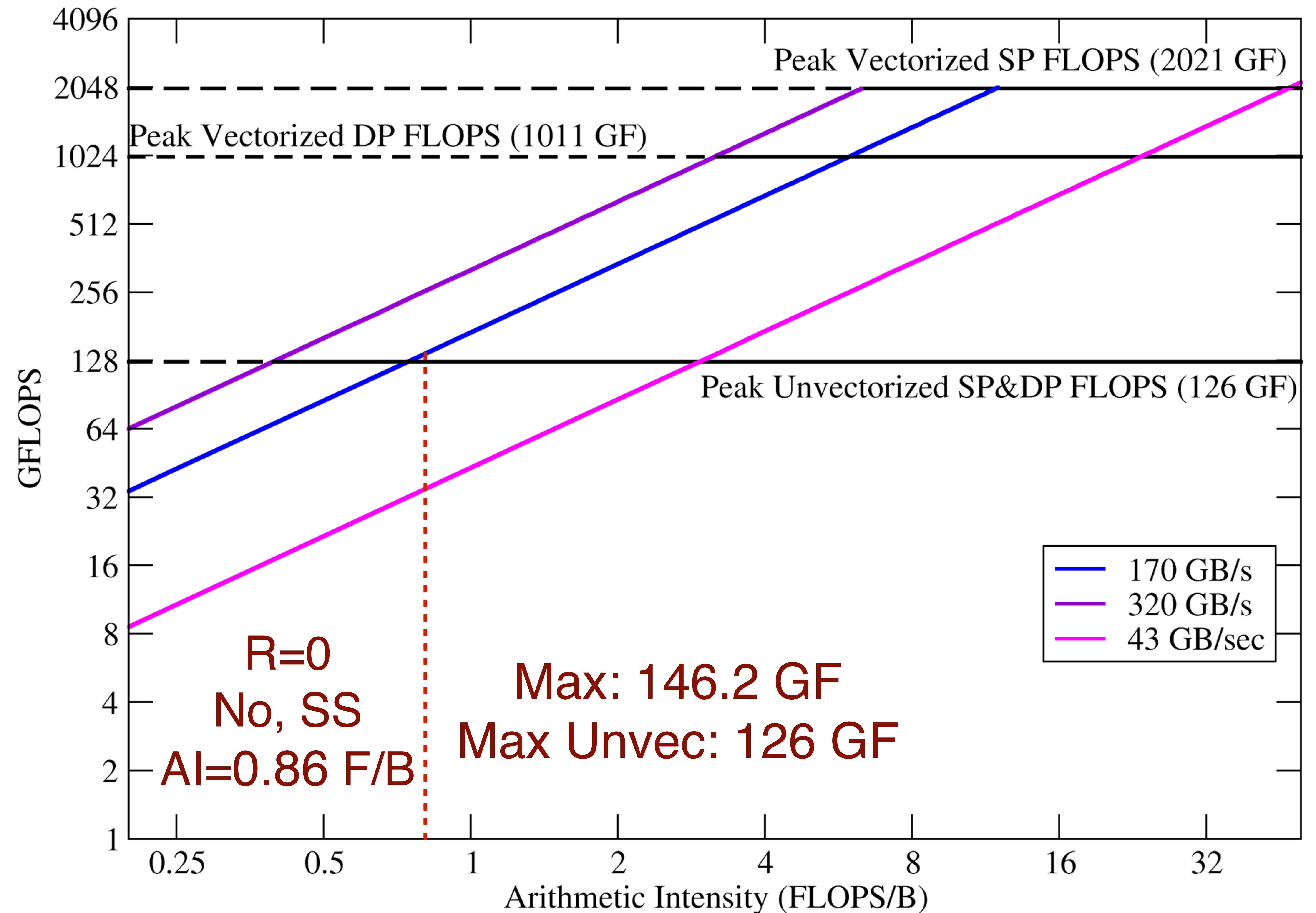
Rooflines - SP

- Poor Bandwith Use Example:
  - E.g. if 43 GB/sec sustained
  - Completely memory bound, vectorization won't help



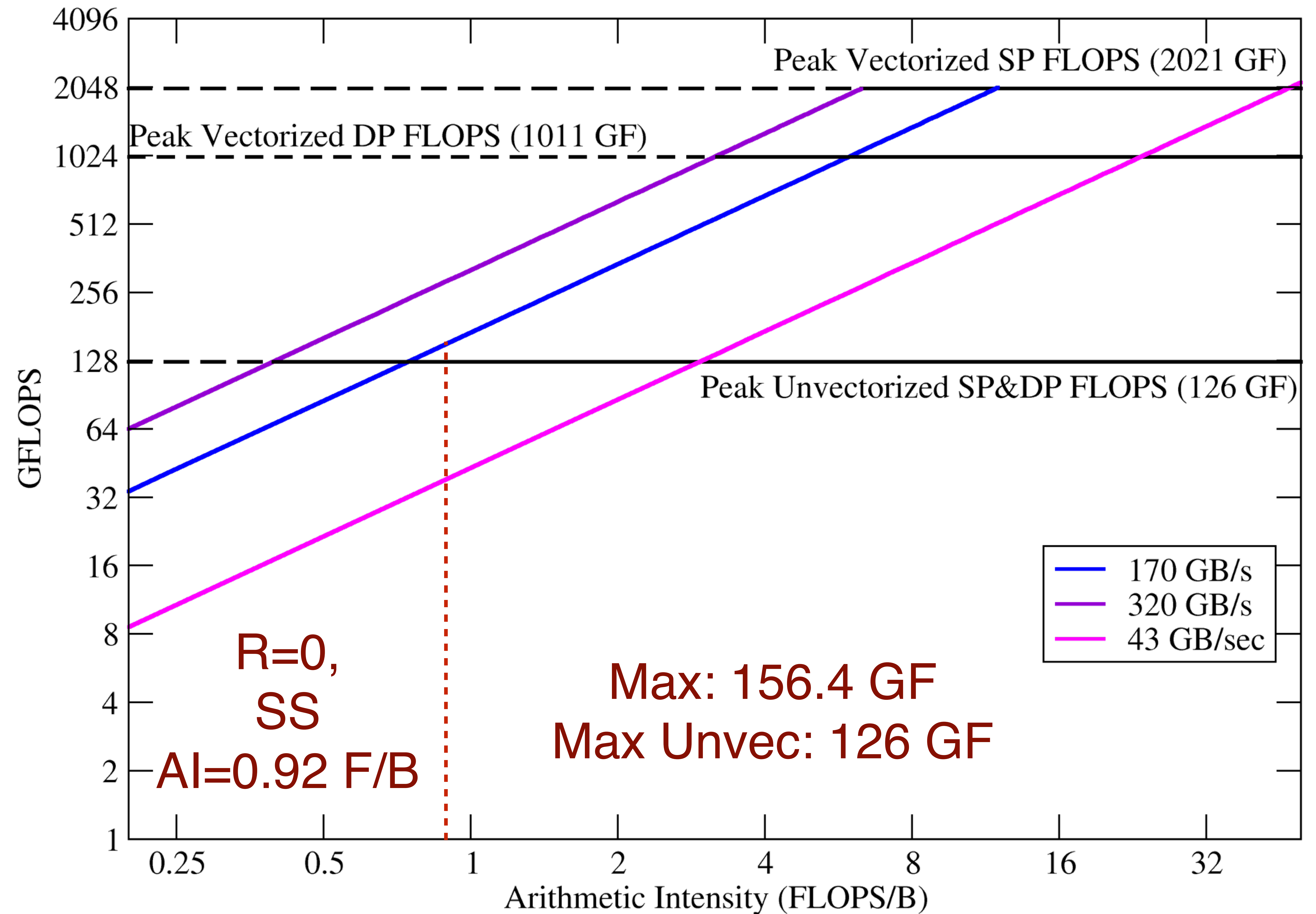
Rooflines - SP

- If you could exhaust B/W:
  - alignment
  - L2 (&L1) prefetching
  - large memory pages(?)
- Naive case (no reuse etc)
  - Compute bound for unvectorized arithmetic
  - Still bandwidth bound for vectorized arithmetic
  - use further bandwidth saving tricks…



Rooflines - SP

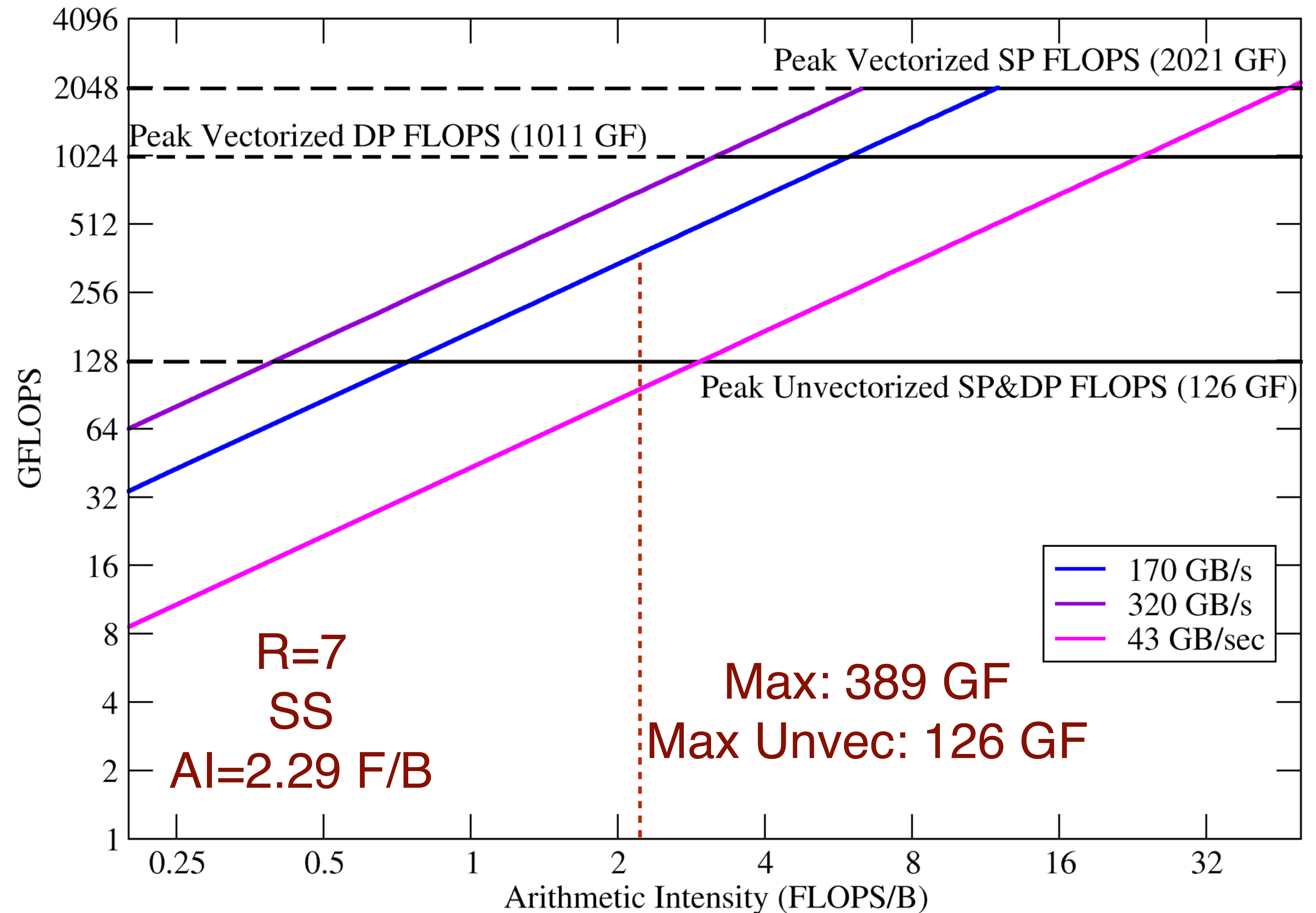
- Add Streaming Stores



Rooflines - SP

- Add Streaming Stores
- Add Cache reuse
- Add 2-row compression
  - B/W bound, free FLOPS

- Max ~ 3x Max Unvec.
  - vectorization is desirable

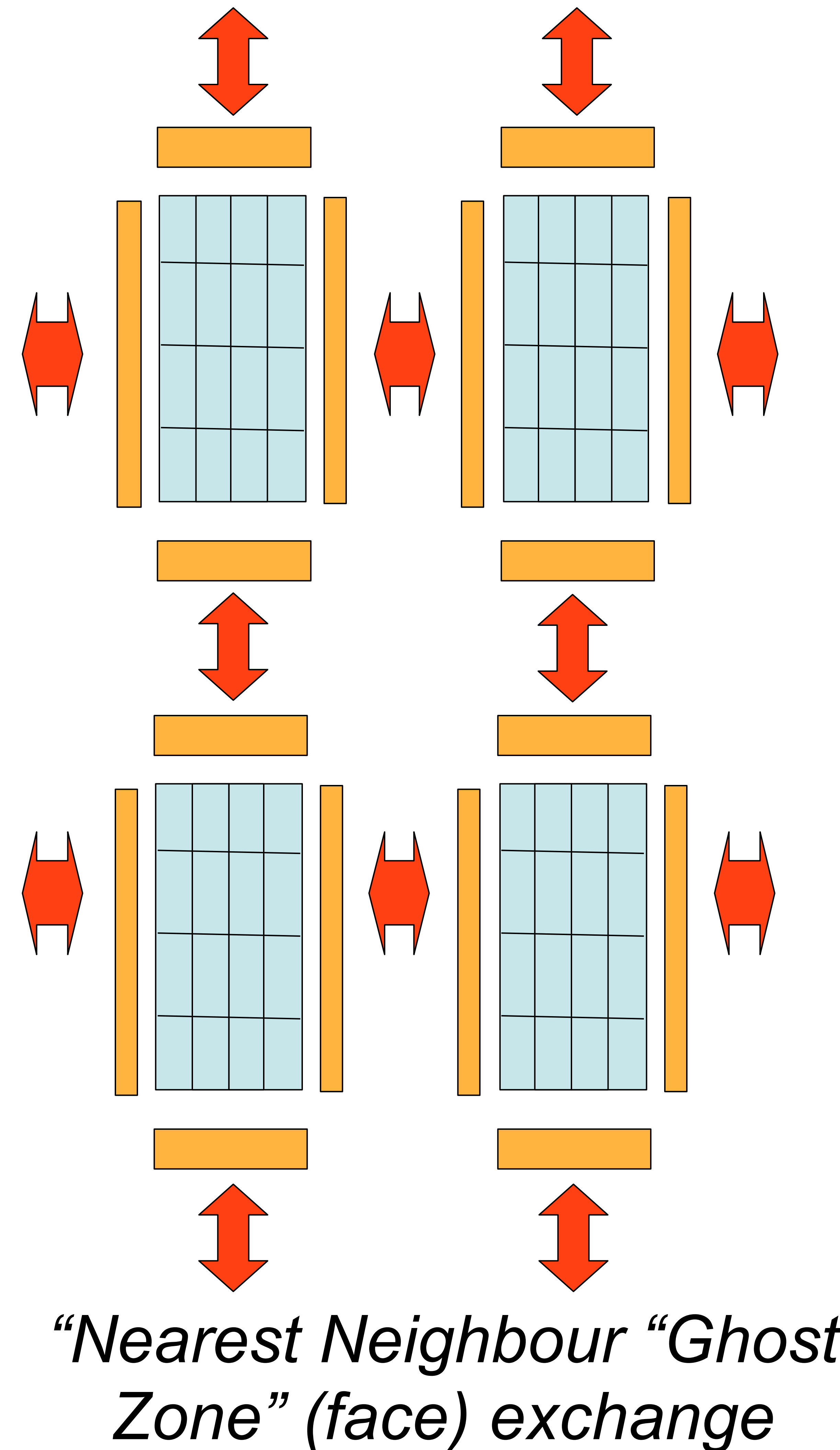


Enemies of performance

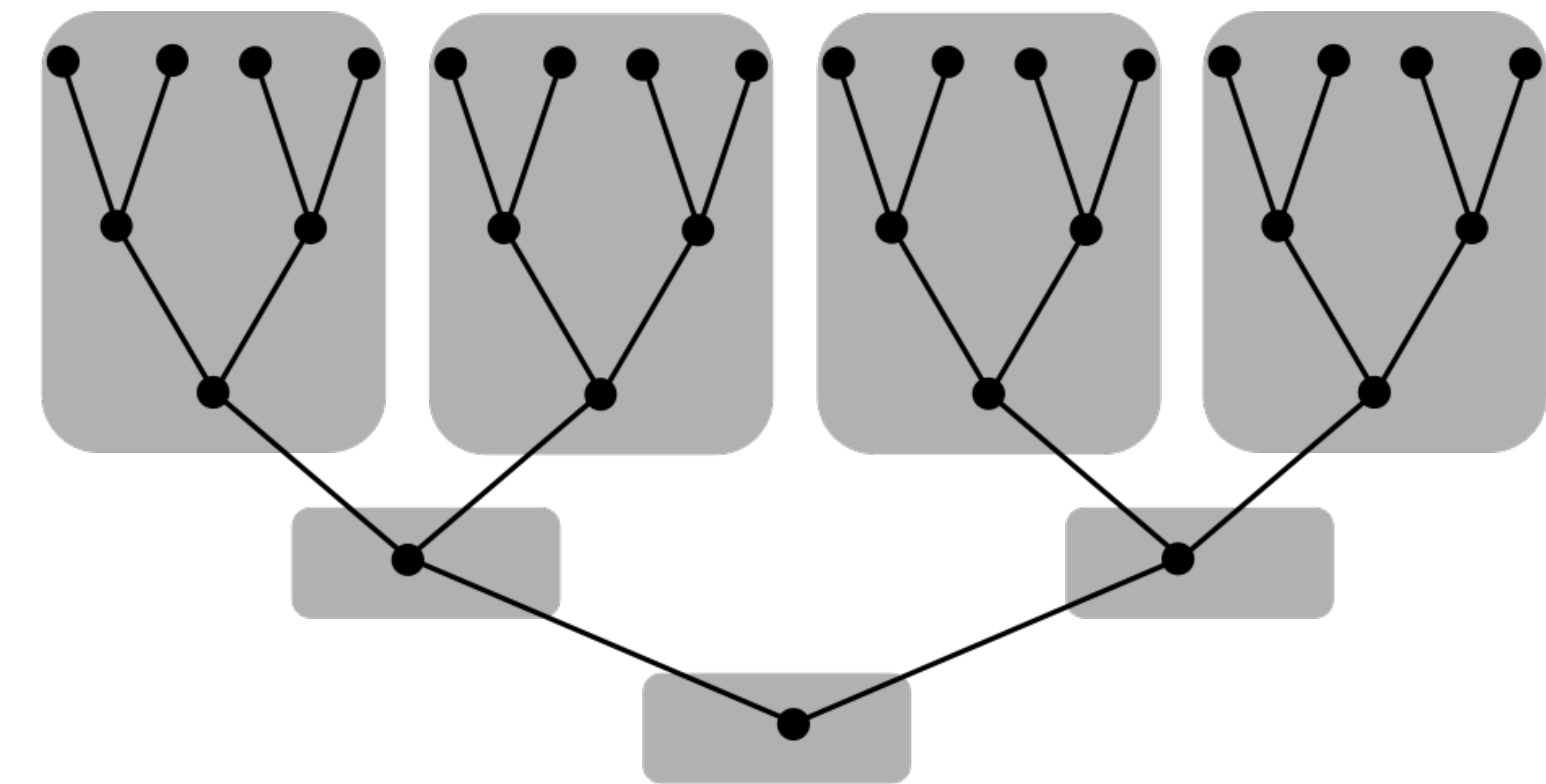
- Idleness, Dependency, Inefficiency and Overheads
  - Lowest performance is when the chip is idle
    - using only a single core from a multi-core node (rest are idle)
    - microarchitectural issues (pipeline stalls, etc)
    - waiting for message to arrive from other node, data to arrive from memory
    - waiting for a hardware resource (register, or FP Unit) to become available
    - waiting for other threads to reach a certain point in a calculation (barrier)
  - Inefficient use of a resource
    - e.g. not maximizing memory bandwidth, by reading unaligned data or by not achieving ‘coalesced’ reads (on a GPU).
  - Some operations just take a “long” time, overhead on useful computation
    - e.g. Create and Join threads, thread barriers, synchronization

Communicating Between Nodes

- Often need to communicate between nodes
  - point-to-point
    - boundary exchange for nearest neighbours
  - global sums/inner products
    - Krylov solvers
  - all-to-all communiations
- In some cases, think of communication as a 'remote memory access'



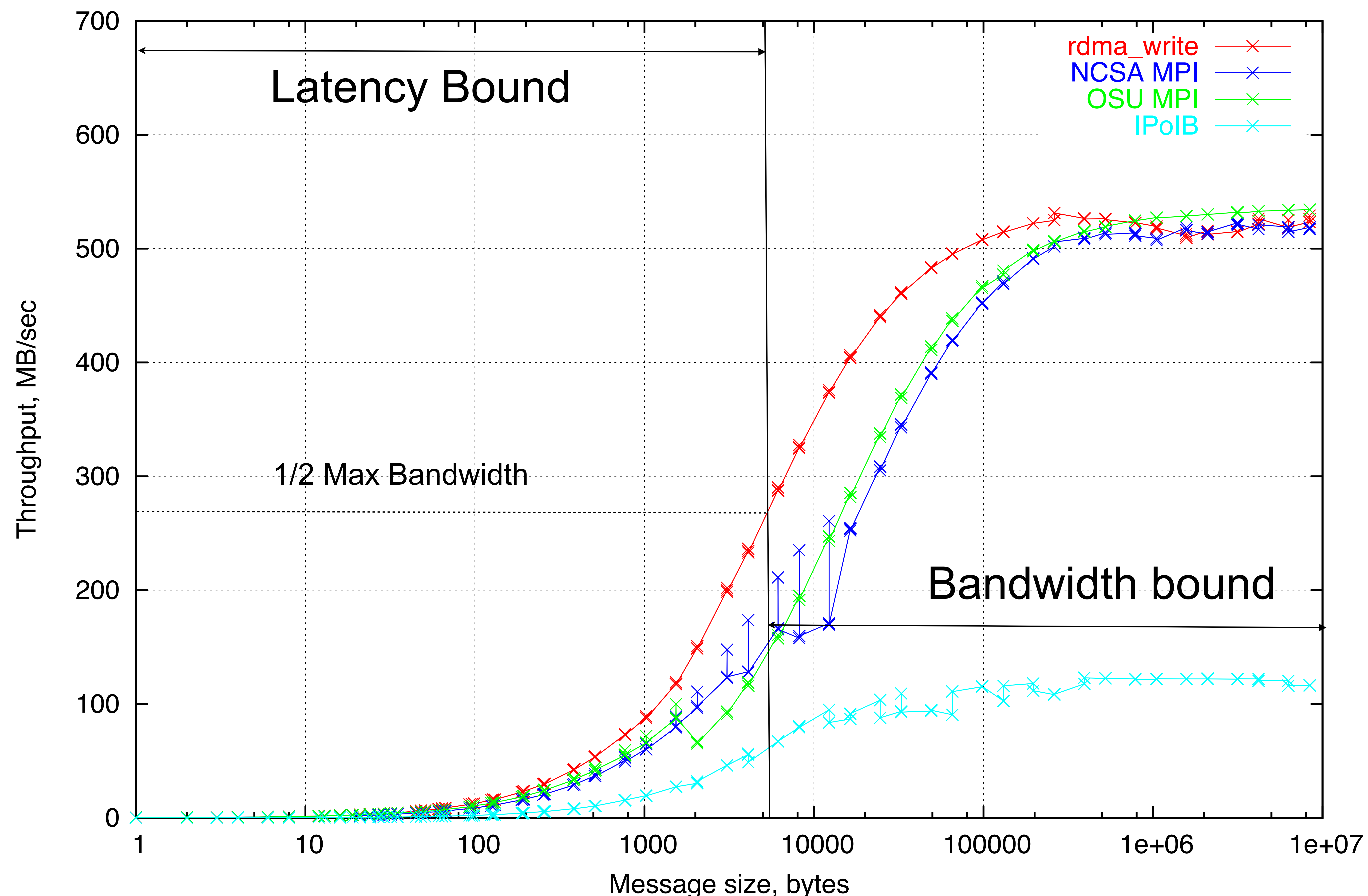
"Nearest Neighbour "Ghost Zone" (face) exchange



Tree Reduction (e.g. sum)

Message passing constraints

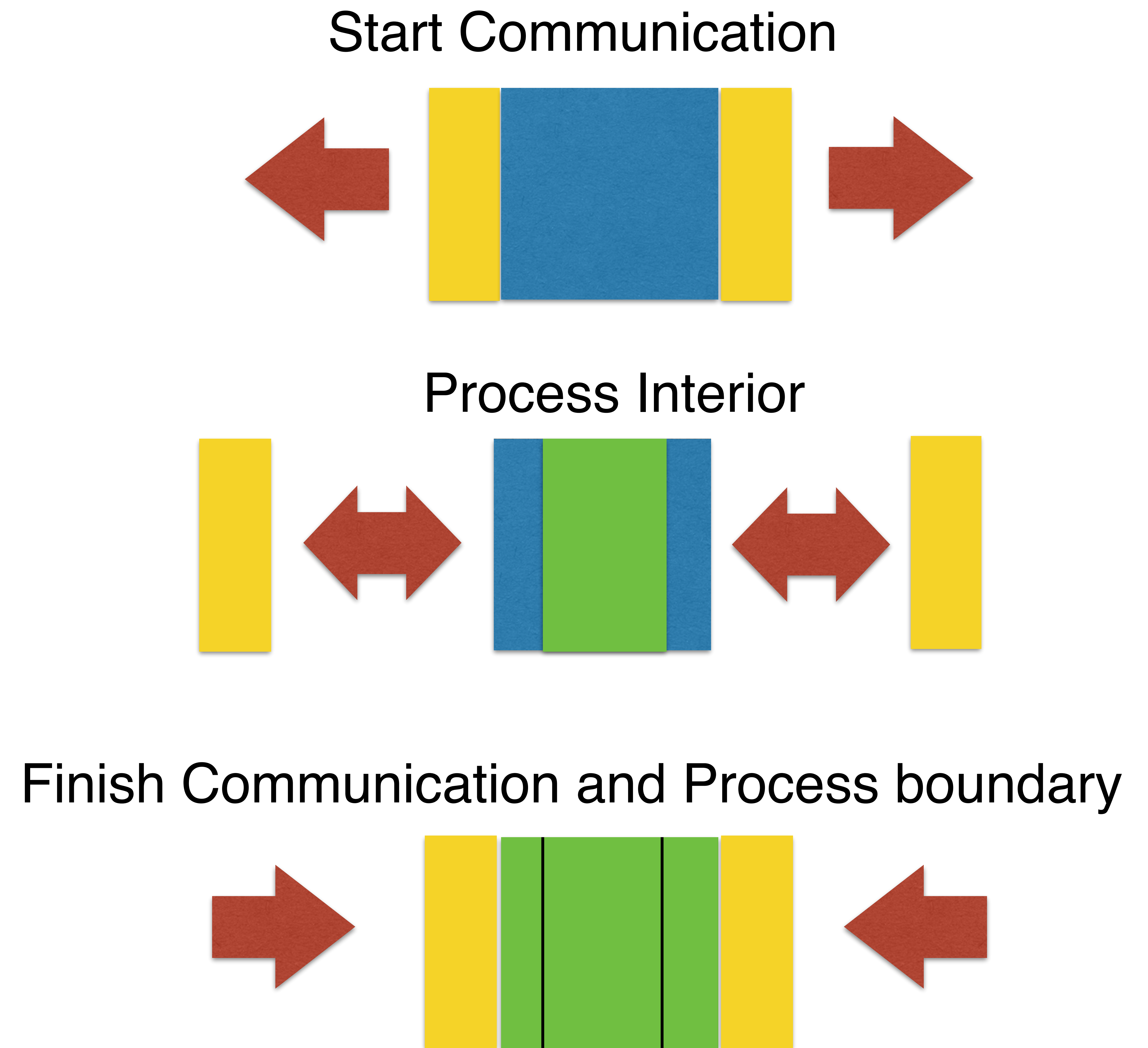
- Sending message has a start-up time (latency) and a flow-rate (bandwidth)
- Similar in some sense to DRAM
- Message can be latency or bandwidth bound given its size.
- Can use asynchronous message passing to hide communication (overlap compute with comms)



Performance of MPI over infiniband: http://lqcd.fnal.gov/benchmarks/newib/index.html

Mitigating Bottlenecks

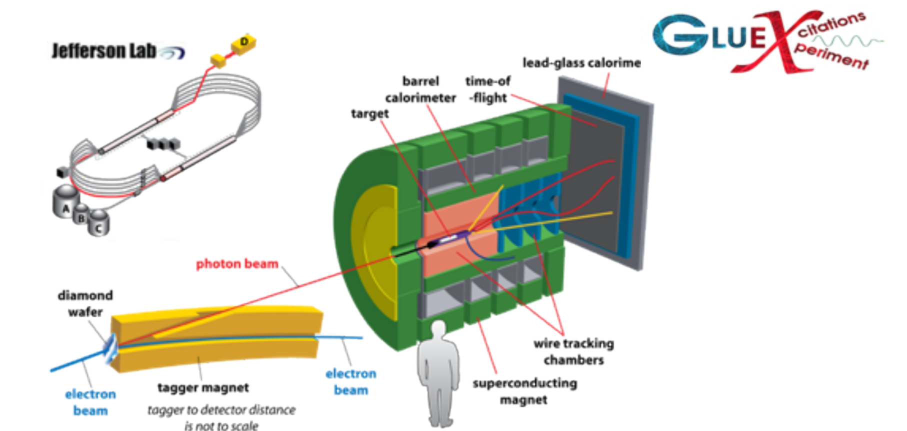
- Some bottlenecks can be mitigated
- Overlap Communication with Computation
  - use asynchronous communication
  - process interior while boundary data is in flight
  - breaks down if interior is too small
- Hide latency with parallelism
  - If a thread is waiting on latency, switch to working on another — GPUs do this in hardware
- Reduce overheads if possible
  - e.g. if thread fork/join is expensive collapse multiple #pragma omp parallel regions into one (may need to use other synchronization)



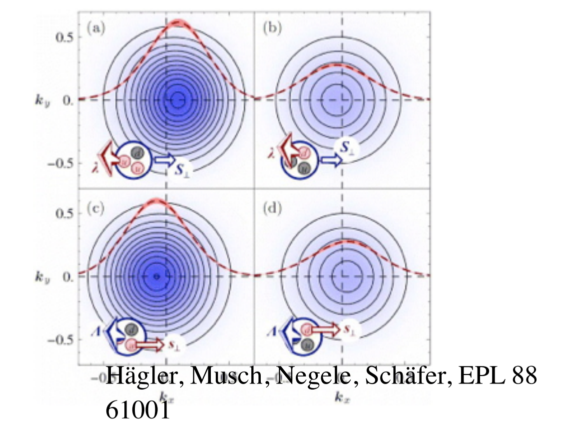
Art of Developing For High Perf.

- As you can see Developing for Performance can be a complex endeavor
- Many, hierarchical levels of concurrency to exploit:
  - On Chip: Vectors (Intrinsics, language extensions) & Threads (OpenMP, pthreads, TBB)
  - Inter-node: Message passing (MPI), Remote Memory Access (PGAS)
- Complex Memory Hiearachy
  - Caches, Scratch-pads, DRAM etc
- Bottlenecks, inefficiencies, overheads
  - Speeds & Feeds: Interconnect/System Bus/Memory/Instruction Latencies & Bandwidths
  - Hardware Parallelism vs. Problem Parallelism mismatches
- Didn't even discuss I/O, file transfer and other day-to-day minutiae
- Good news: You have help to stay productive
  - Frameworks, Optimized Libraries, Tools, People

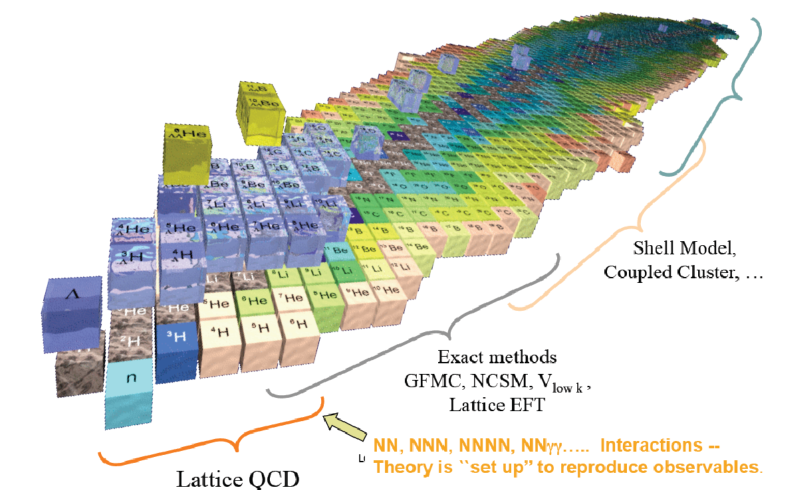
Case Study: Lattice QCD for NP



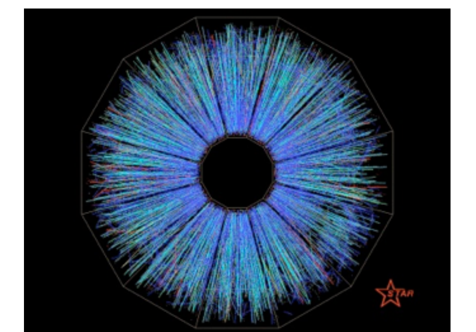
- What kind of matter can QCD make?


Hägler, Musch, Negele, Schäfer, EPL 88 61001

- How does QCD make protons, neutrons?
  - what are the distribution of quarks, gluons, etc in a proton or neutron ?



- QCD must predict properties of light nuclei
  - how to make helium, tritium etc

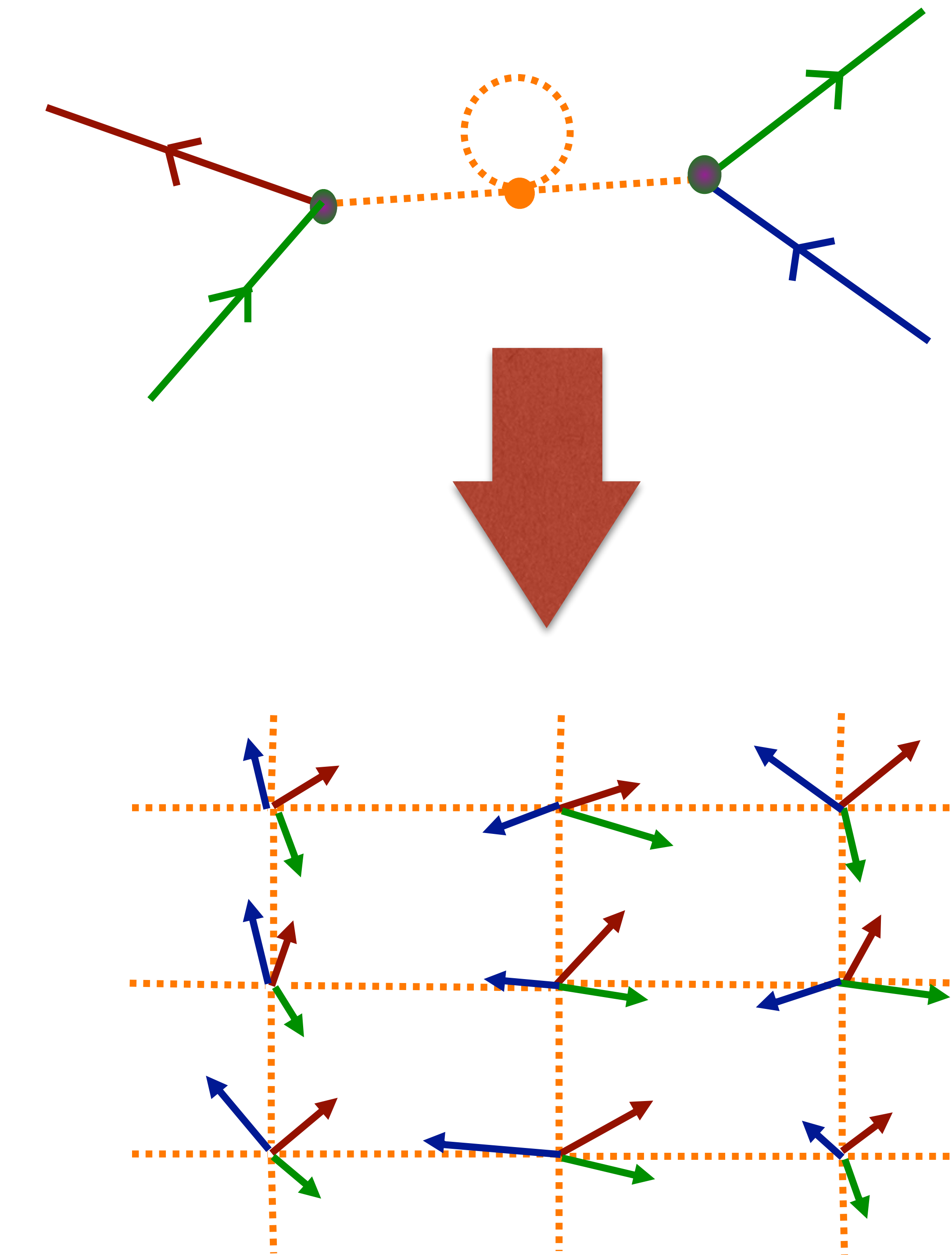


- How does QCD behave under extreme temperatures & pressures such as in exploding starts or shortly after the Big-Bang.

Methodology

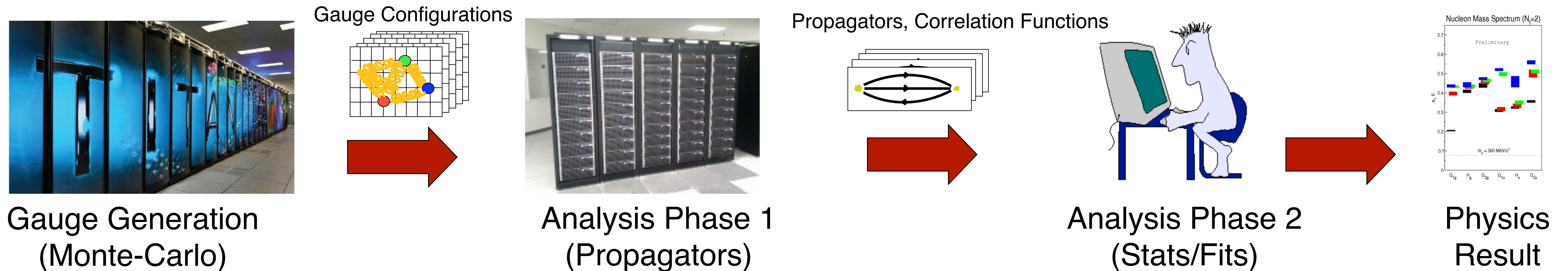
- Lattice QCD - a formulation of QCD amenable to non-perturbative calculation
  - quarks live on lattice sites
  - gluons move to lattice links
    - move from su(3) Lie Algebra to SU(3) Lie Group
    - parallel transporters
  - Euclidean time
  - Path integrals become products of integrals
  - Lattice Actions
    - e.g. Wilson Gauge action $O(a^2)$ discretization error
    - Fermion Formulation: $O(a)$ or $O(a^2)$ discretization typically

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}A \, \mathcal{D}\bar{\psi} \, \mathcal{D}\psi \, \mathcal{O} \, e^{-S(A,\bar{\psi},\psi)}$$



$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \prod_{\text{all links}} dU \prod_{\text{all sites}} d[\bar{\psi},\psi] \, \mathcal{O} \, e^{-S(U,\bar{\psi},\psi)}$$

LQCD Calculation Workflow



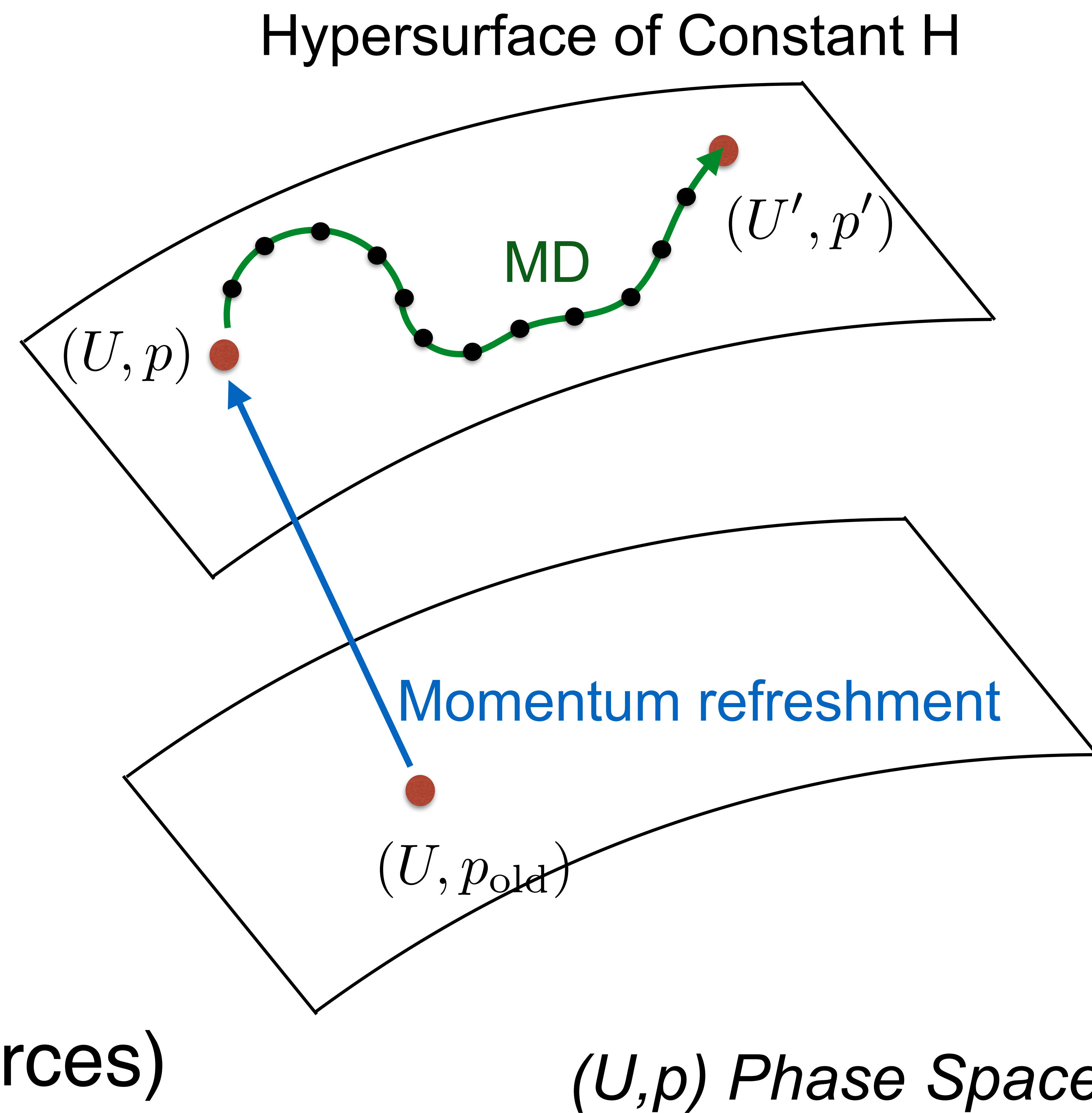
- Gauge Generation: Capability Computing on Leadership Facilities
  - configurations generated in sequence using Markov Chain Monte Carlo technique
  - focus the power of leadership computing onto single task exploiting data parallelism
- Analysis: Capacity computing, cost effective on Clusters
  - task parallelize over gauge configurations in addition to data parallelism
  - can use clusters, but also Leadership Facilities in throughput (ensemble) mode.

Hybrid Monte Carlo (HMC)

1. Refresh momenta from Gaussian Heatbath
   - generate (U,p) from (U,p_{old})
2. Compute H = H(U,p)
3. Perform Molecular Dynamics (MD) trajectory
   - generate (U',p')
   - MD must be reversible and 'area preserving'
4. Compute H' = H(U',p')
5. Accept with Metropolis probability

$$P = \min\left(1, e^{-H(U',p')+H(U,p)}\right)$$

6. If rejected new state is (U,p)

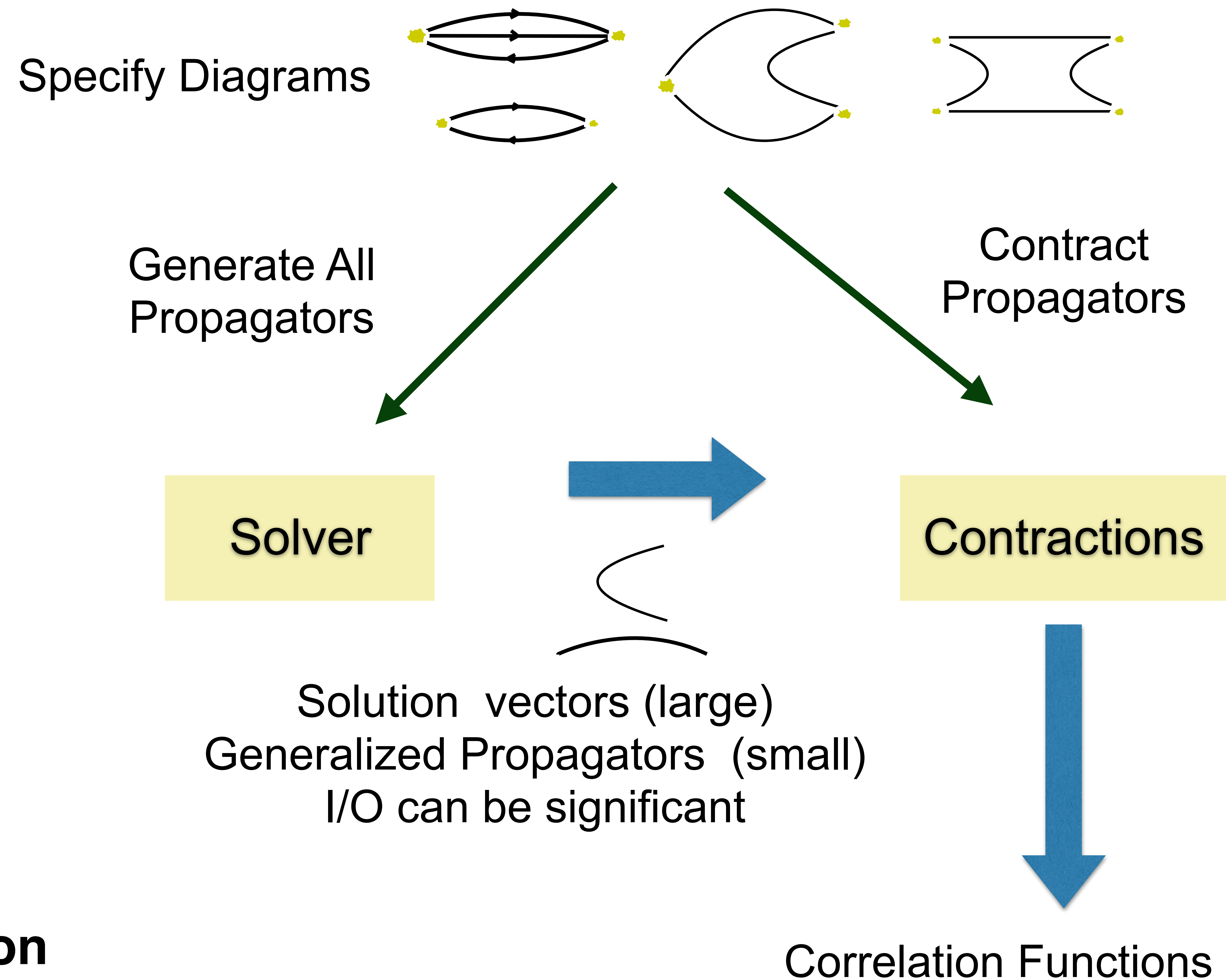


- **O(10000)** trajectories per ensemble
- 60-80% of work in Linear Solvers (Quark MD Forces)

(U,p) Phase Space

Method for Computing Corr. Fn-s

- Distilllation: Two main components
  - propagator calculations (solver)
  - contraction calculations
- Contractions use dense matrix mutliply
  - matrix dimension is O(100) (# sources)
- Many solves needed on single configuration:
  - #spin x #timeslice x #source x #quarks
- Typical Example
  - 4 spins, 256 timeslices, 386 source vectors and light + strange quarks
  - **790,528 individual solves per configuration**



Solvers

- Traditionally we solve the Linear Systems with iterative Krylov Subspace solvers
- These can
  - work as black boxes
  - typically need only L1 BLAS and MV operations
  - typical candidates: Conjugate Gradients, BiCGStab
- Convergence depends on condition number of M
- As quark mass approaches the physical mass, M becomes more and more ill conditioned
- Critical Slowing Down in the Solver.

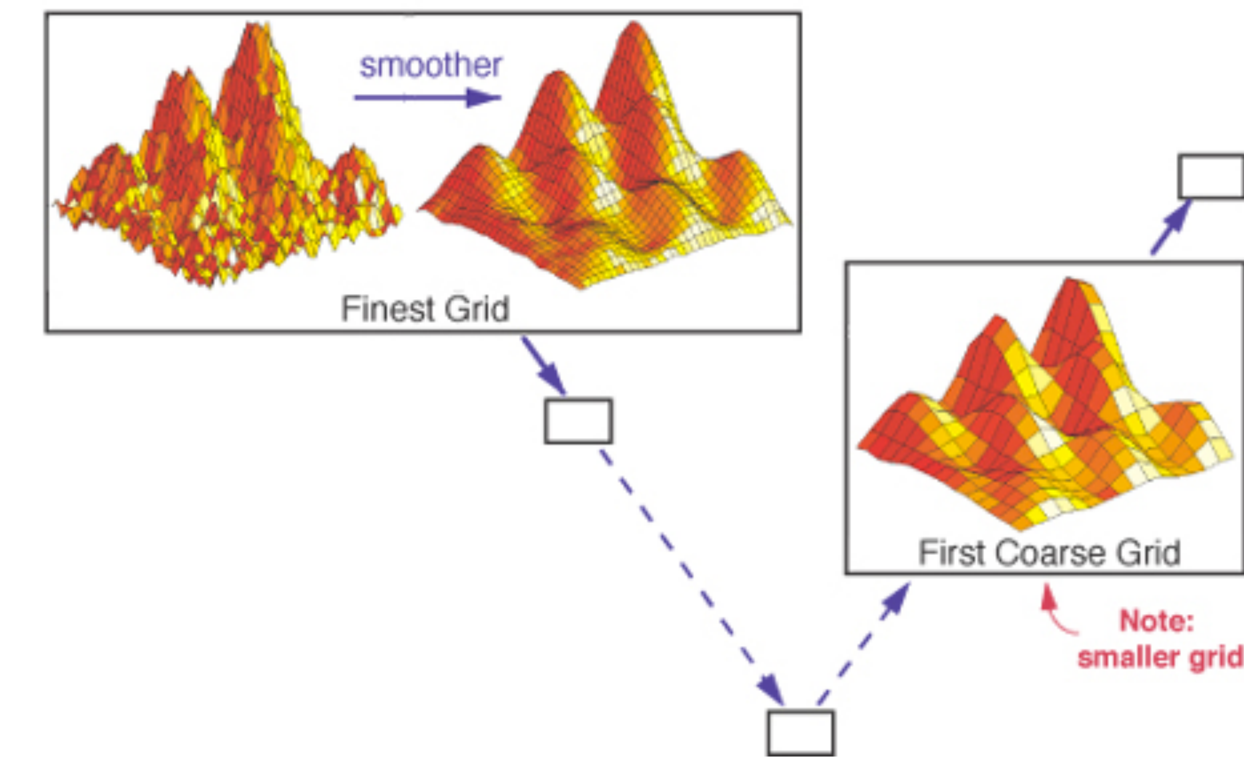
($\phi_0 = \phi$ is an Initial Guess)

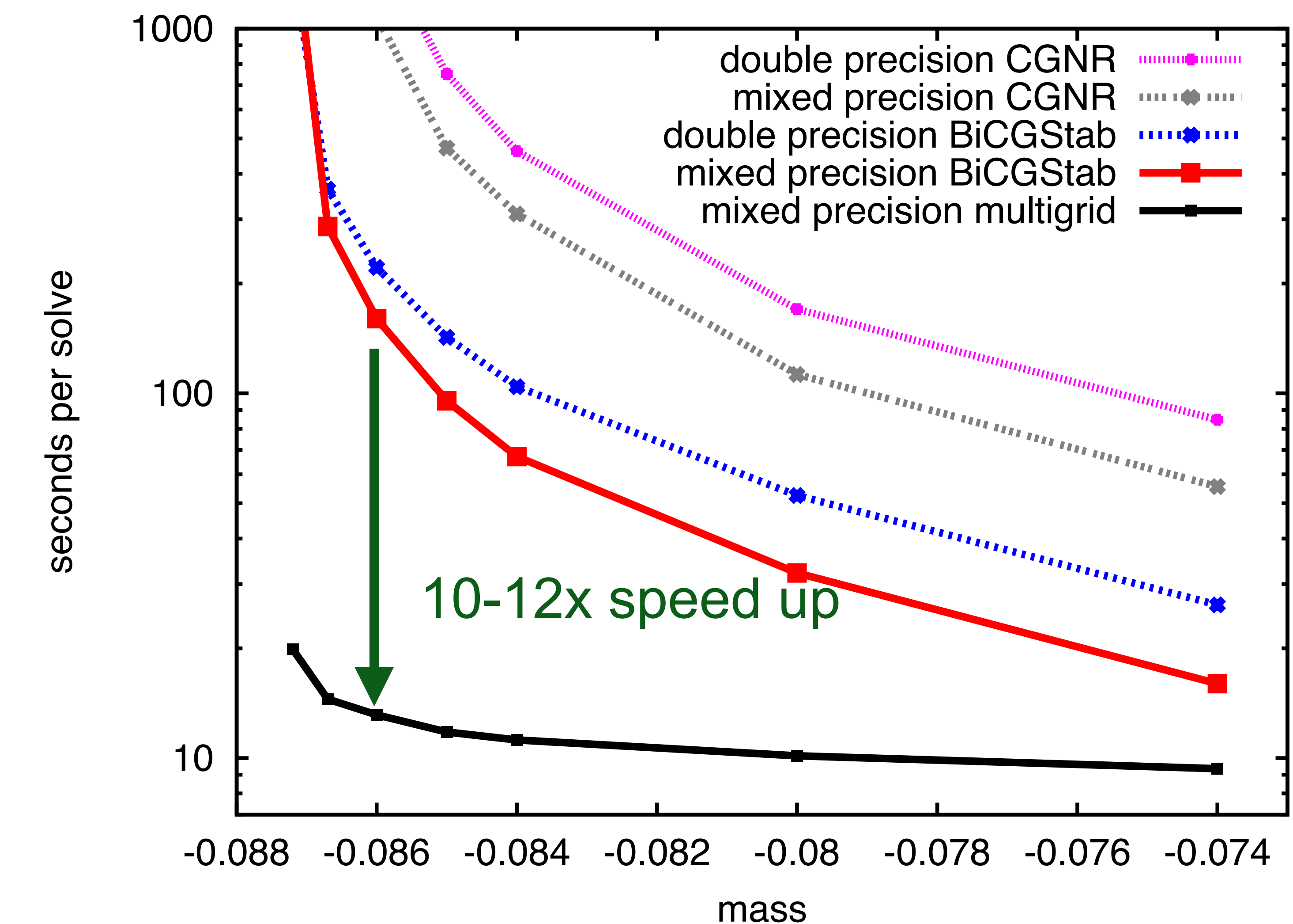
1. Compute $r_0 = \chi - M^\dagger M \phi_0, \; p_0 = r_0$
2. For $j = 0, 1, \ldots$ until convergence:
3. $\alpha_j = \dfrac{\langle r_j, r_j \rangle}{\langle M p_j, M p_j \rangle}$
4. $\phi_{j+1} = \phi_j + \alpha_j p_j$
5. $r_{j+1} = r_j - \alpha_j \left(M^\dagger M \right) p_j$
6. $\beta_j = \dfrac{\langle r_{j+1}, r_{j+1} \rangle}{\langle r_j, r_j \rangle}$
7. $p_{j+1} = r_{j+1} + \beta_j p_j$
8. End For

Algebraic Multi Grid

- Critical Slowing down is caused by ‘near zero’ modes of M
- Multi-Grid method
  - separate (project) low lying and high lying modes
  - solve for high lying modes with “smoother”
  - solve for low modes on coarse grid with reduced dimensional operator
  - Gauge field is ‘stochastic’, so no geometric smoothess on low modes => algebraic multigrid
  - Setting up restriction/prolongation operators is costly
  - Easily amortized in Analysis with O(100,000) solves

Image From: http://computation.llnl.gov/casc/sc2001_fliers/SLS/SLS01.html
Credit: LLNL, CASC

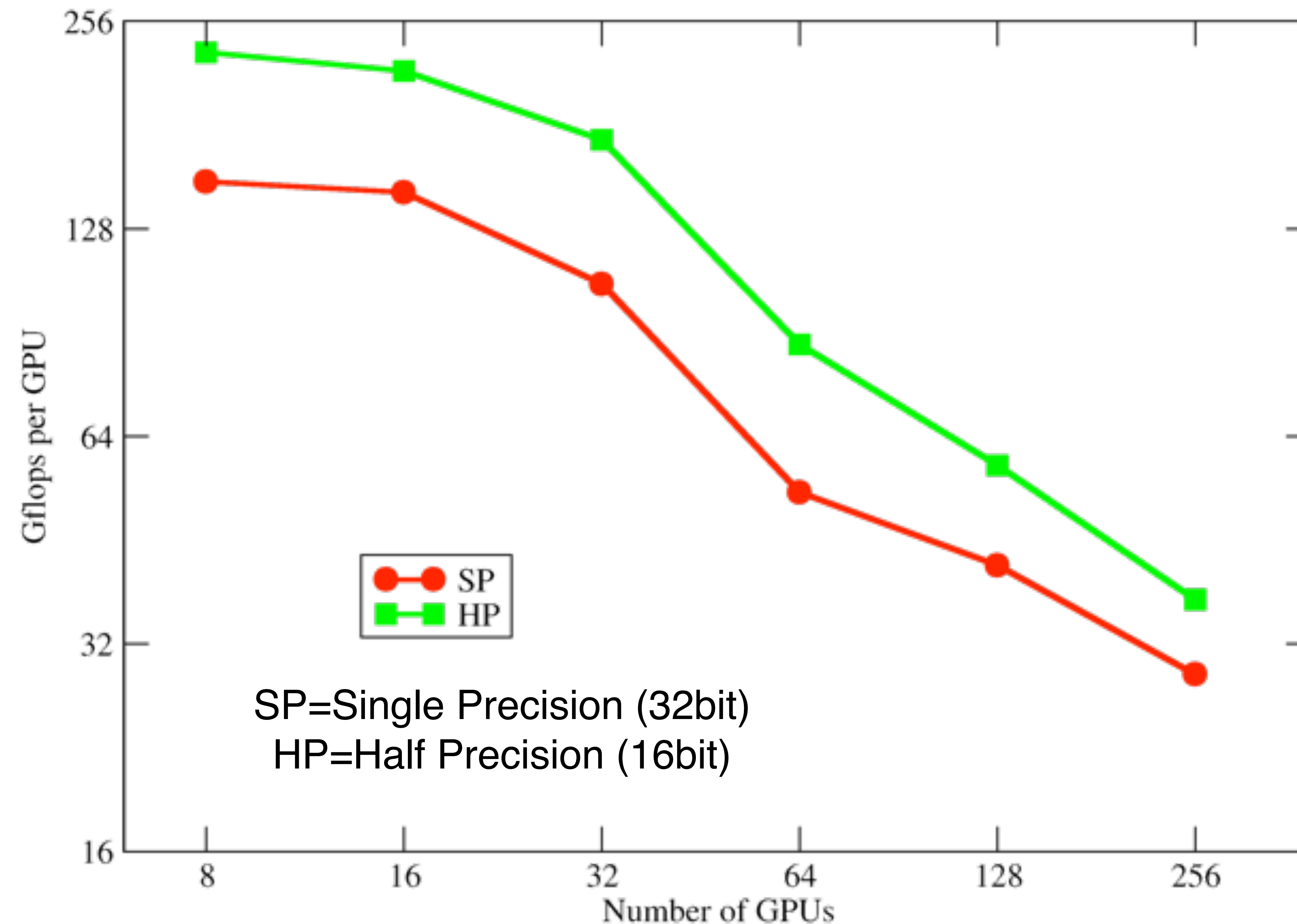




Multi-Grid. figure from J. C. Osborn et. al. PoS Lattice 2010:037,2010, R. Babich et. al. Phys. Rev. Lett, 105:201602,2010

Scaling Bottleneck Example:

R.Babich, M. A. Clark, B. Joo, G. Shi, R. C. Brower, S. Gottlieb. "Scaling Lattice QCD Beyond 100 GPUs"
Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)
page 70:1-70:11, New York, NY, USA, ACM (2011)



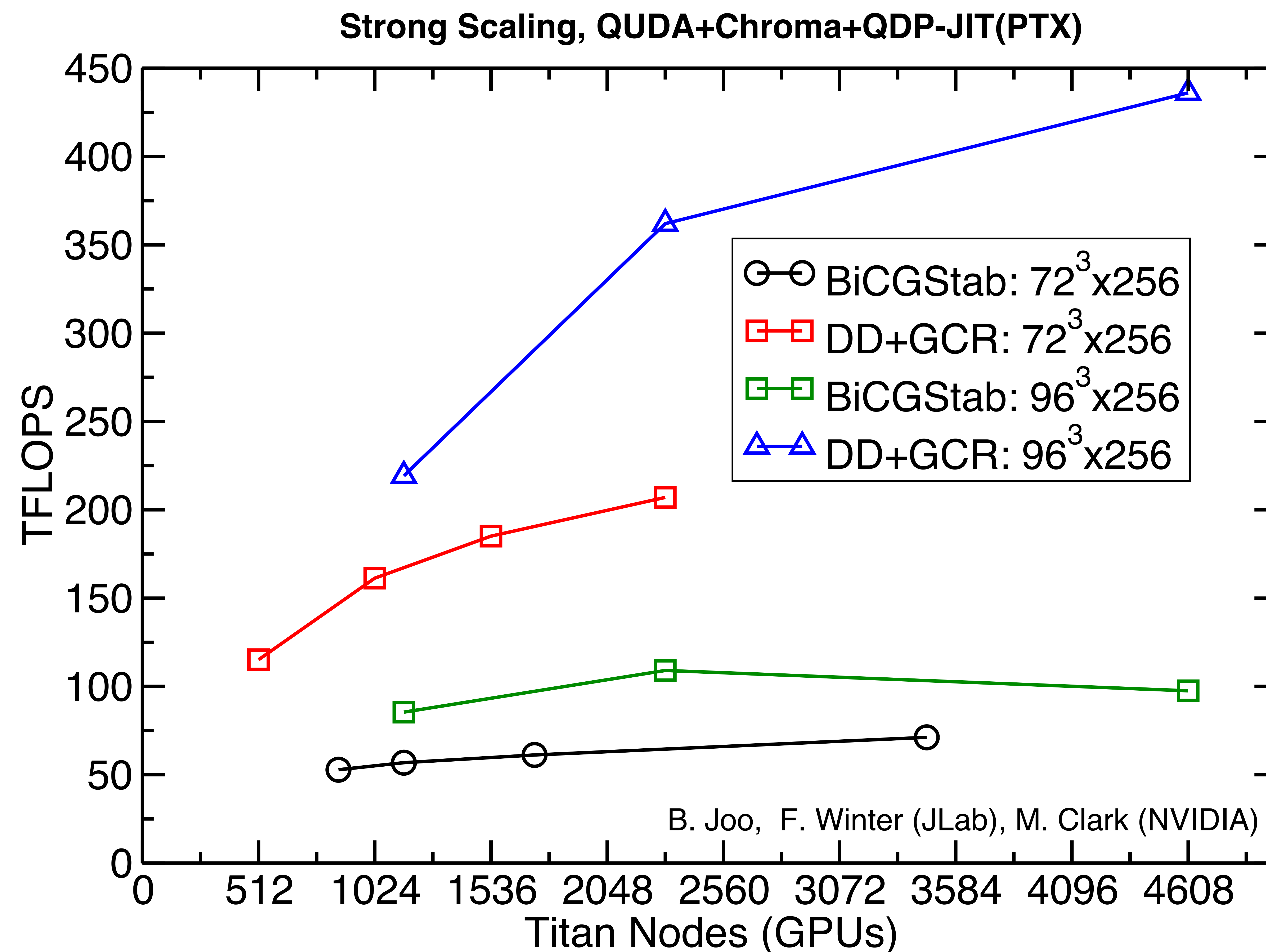
- One of the original findings was that strong scaling was difficult with accelerators
- Inter-device communications was considered to be the main bottleneck
- Mismatch of bandwidths
  - 8+8 GiB on PCIe Gen2
  - ~150-170 GB/sec on device
- Spurred the development of Domain decomposed solvers…

Architecture Awareness

- Attempt to deal with communications bottleneck:
  - don’t communiate at all
- Use a block-diagonal operator as a ‘preconditioner’ in the solver
  - inner-outer scheme
- Arrange to spend most time in the preconditioner.
- But be aware:
  - block diagonal operator is a ‘wavelength filter’
  - outer scheme still needs to deal with long wavelength modes
- Example of interplay of architecture, algorithm, applied maths and physics.

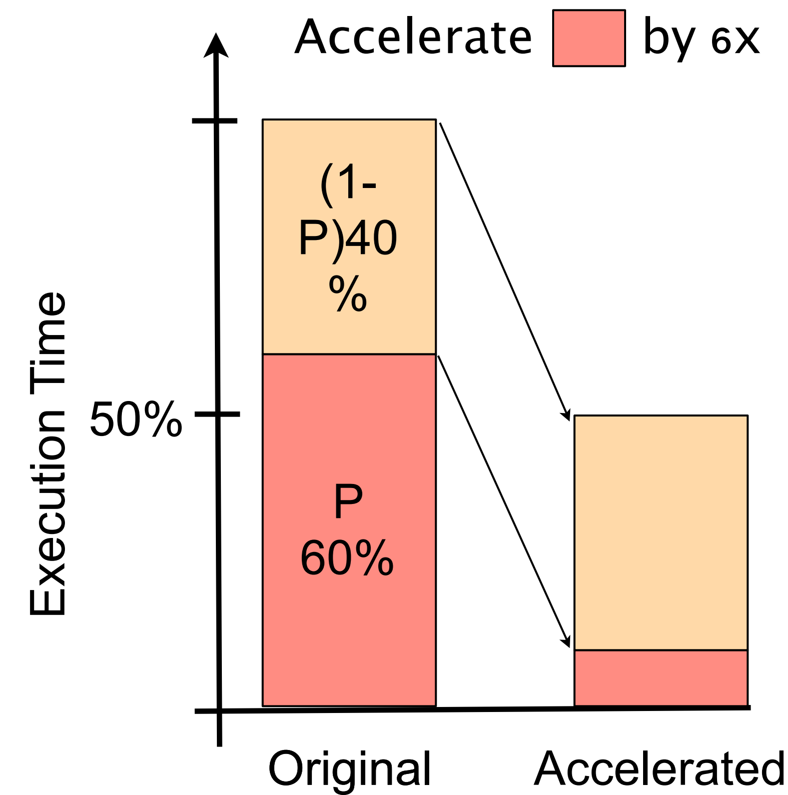
Solver Performance

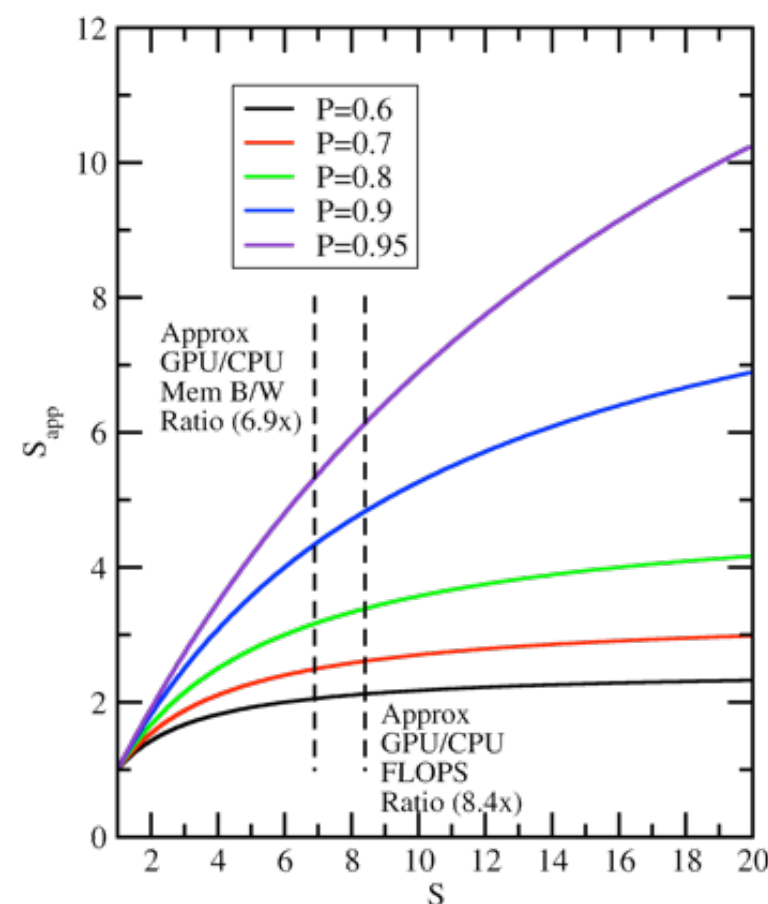
- QUDA Solver performance on Titan
  - Cray XK7 system
  - 1 NVIDIA K20X GPU per node
  - Gemini Interconnect
- The DD+GCR solver does considerably better than the standard BiCGStab
- But even DD+GCR is affected by strong scaling effects



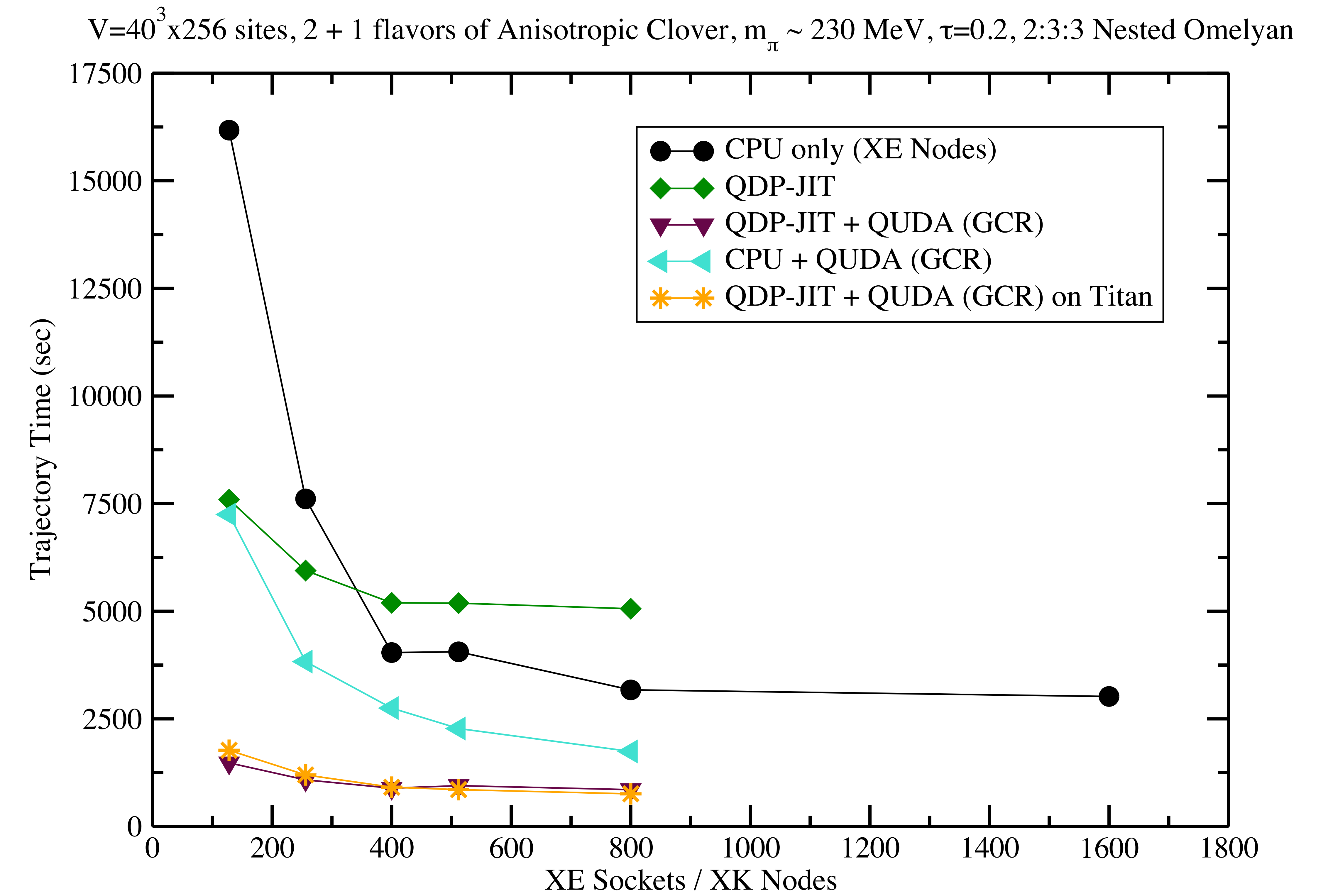
Non-Solver Performance

$$S_{\text{app}} = \frac{1}{(1-P) + \frac{P}{S}}$$





- Amdahl's Law
  - if you speed up (parallelize/ optimize) portion P of your code, overall speedup limited by 1-P portion
  - "what you don't speed up will become your next bottleneck."
  - E.g. HMC on GPUs: after solver is optimized, non-solver part becomes the bottleneck.



F. T. Winter, R. G. Edwards, M. A. Clark, B. Joo, IPDPS'14

Part 2: Summary

- In Part 1 we were thinking about the forms of parallelism and how to program them
  - threads, vectors, processes, etc.
- In Part 2 we thought more about performance and what it means
  - how to think about performance — roofline models, case study with Wilson Dslash
  - what are the obvious enemies of performance — and how to attempt to mitigate them
  - finally we took a case study of a lattice QCD campaing
    - Stages of the computation
    - The key algorithms
    - Architectural factors such as GPUs
    - How software and algorithms can surmount some of these challenges.

Overall Summary

- High Performance Computing is a ‘3rd pillar ’ of Science
  - Attempts to fill the gap between experiment/observation and theory
- High Performance Computers are multi-faceted complex systems
  - performance these days is delivered through parallelism/concurrency
  - power is the key limiter: Efficiency was: FLOPS/$, now it is FLOPS/W x W/$.
- Interplay between Architecture, Algorithm Choice, and Performance Optimization
  - natural for HPC projects to be multi-disciplinary (e.g. SciDAC)
- High Performance doesn’t have to be ‘Big Iron’ - tho some of us like that :)
- Performance is always relative. If you choose your algorithms, libraries, and develop your code, to best exploit your hardware, you’re doing HPC