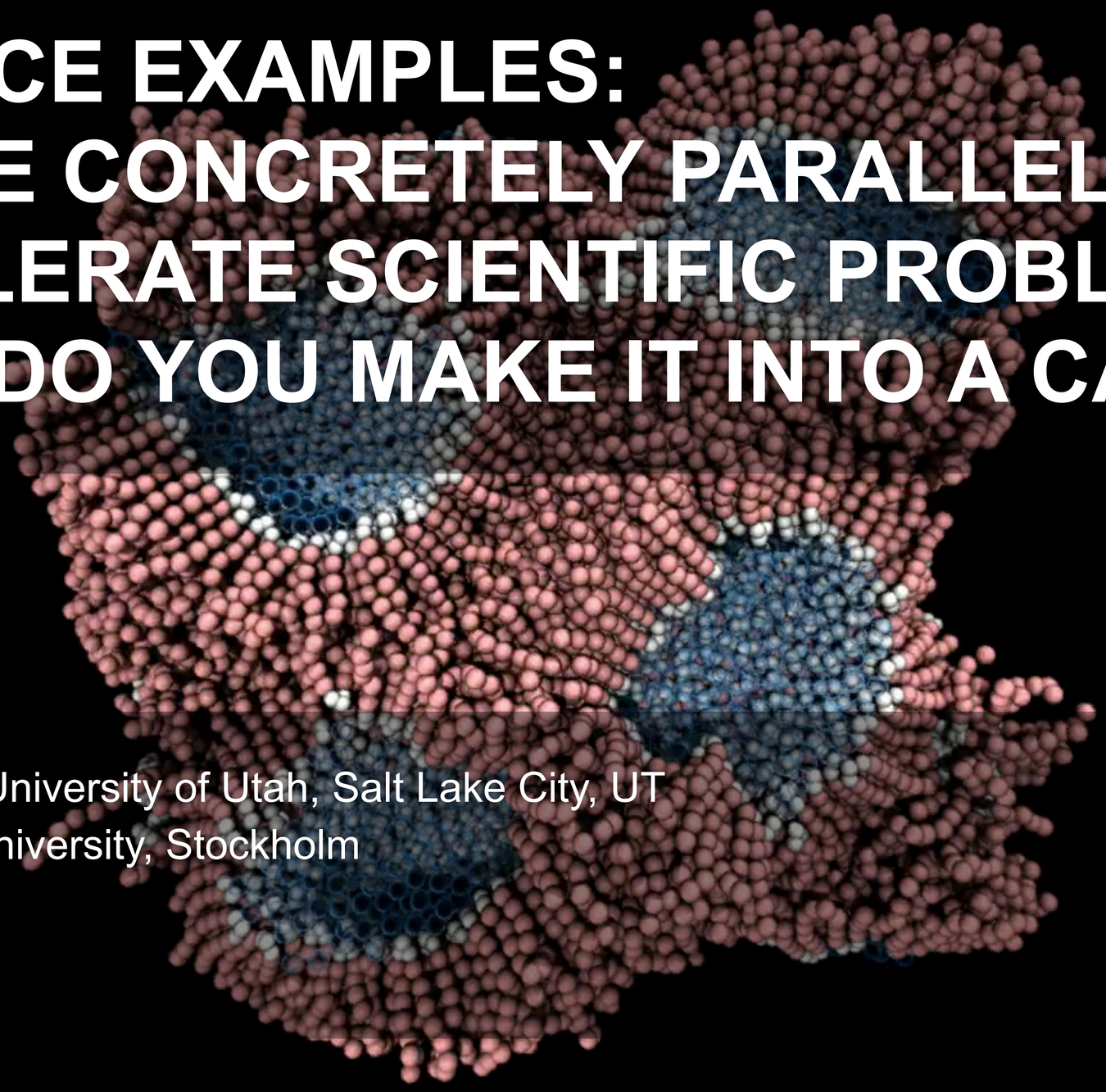
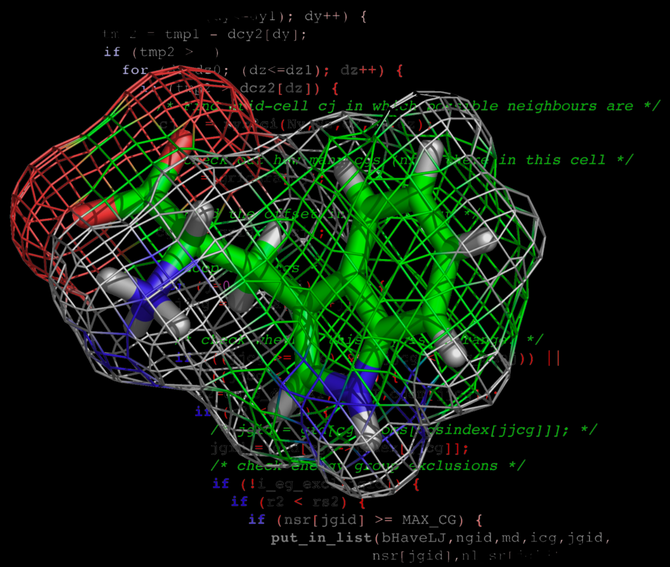
LIFE SCIENCE EXAMPLES: HOW DO WE CONCRETELY PARALLELIZE AND ACCELERATE SCIENTIFIC PROBLEMS? (AND HOW DO YOU MAKE IT INTO A CAREER?)

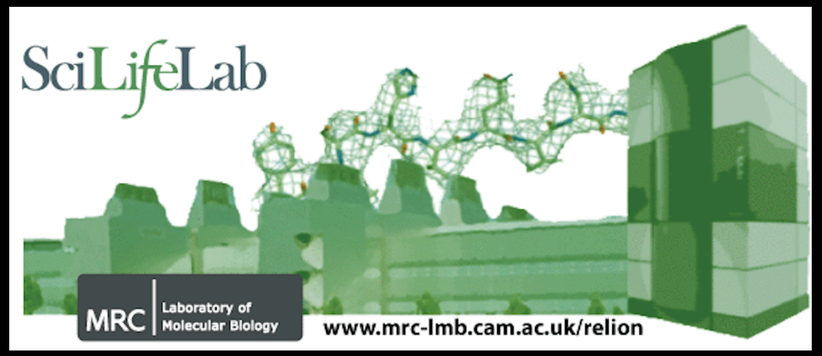


Thomas E Cheatham III, University of Utah, Salt Lake City, UT
Erik Lindahl, Stockholm University, Stockholm
tec3@utah.edu
erik.lindahl@scilifelab.se

Two Stories:

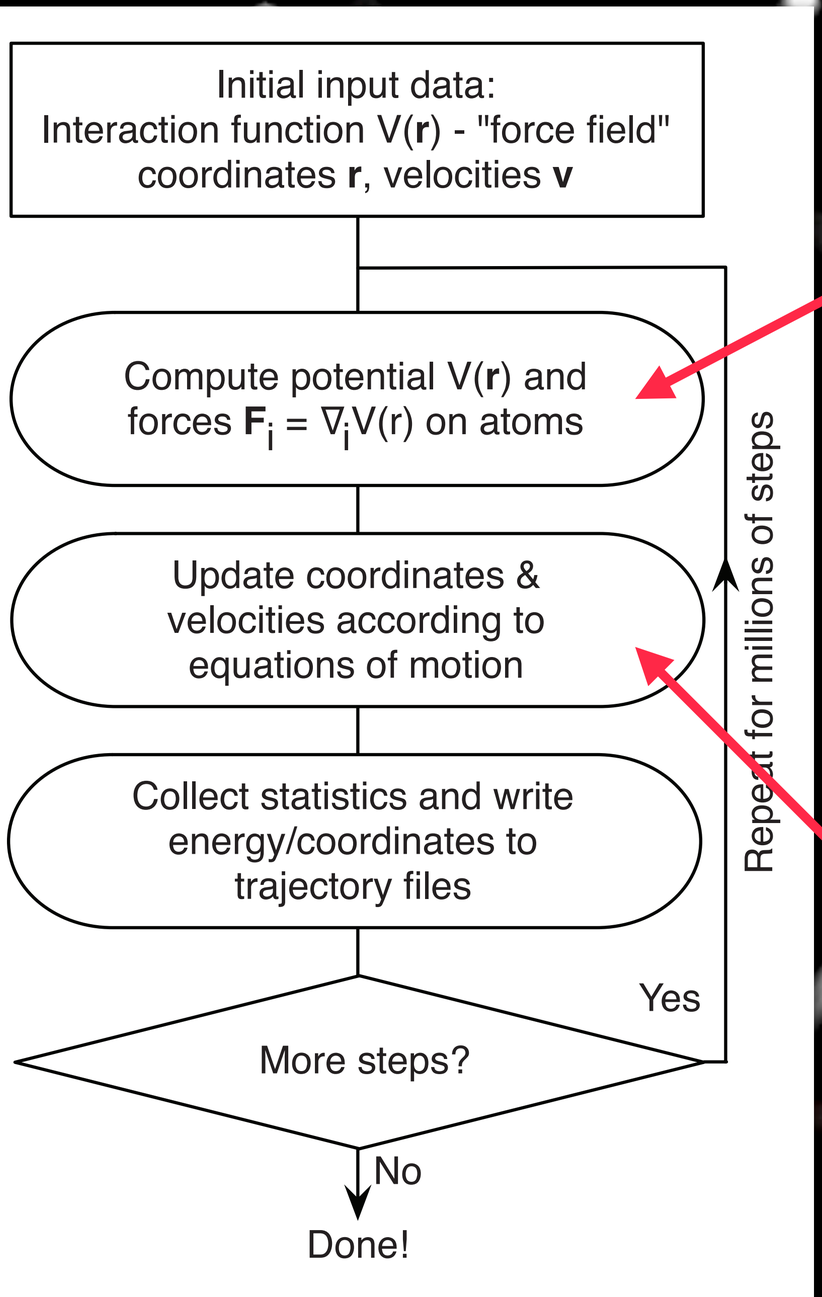


MD:
GROMACS



Cryo-EM:
RELION

Challenge: MD is intrinsically a *sequential* problem



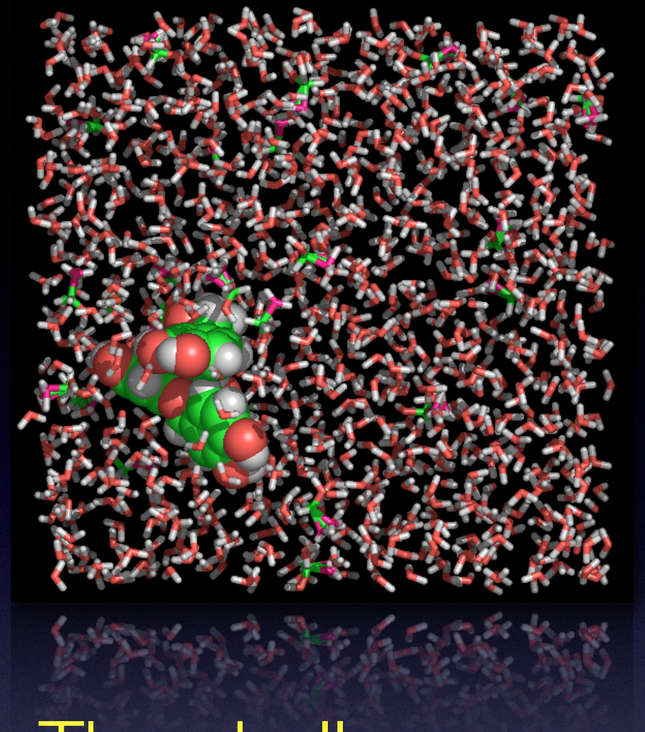
$$V(r) = \sum_{bonds} \frac{1}{2} k_{ij}^{b} \left(r_{ij} - r_{ij}^{0}\right)^2 + \sum_{angles} \frac{1}{2} k_{ijk}^{\theta} \left(\theta_{ijk} - \theta_{ijk}^{0}\right)^2 + \sum_{torsions} \left\{ \sum_{n} k_{\theta} \left[1 + \cos\left(n\phi - \phi_0\right)\right] \right\} + \sum_{impropers} k_{\xi} \left(\xi_{ijkl} - \xi_{ijkl}^{0}\right) + \sum_{i,j} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}} + \sum_{i,j} \left[\frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^{6}} \right]$$

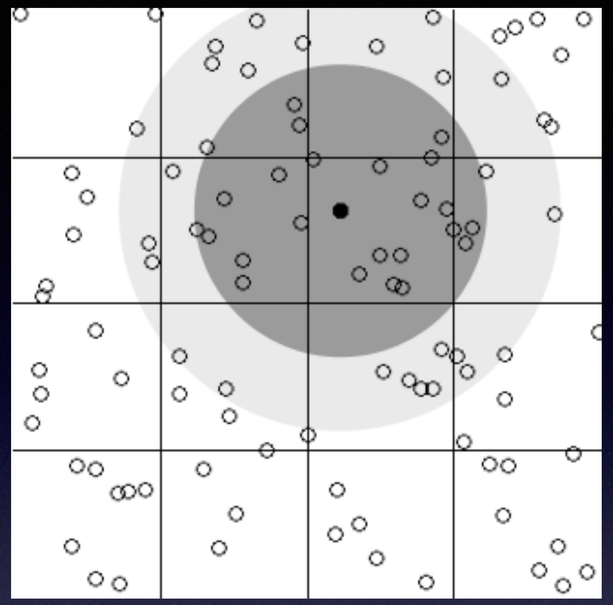
Costly, because these terms involve all pairs

$$m_i \frac{\partial^2 r_i}{\partial t^2} = F_i \qquad i = 1..N$$

$$F_i = -\frac{\partial V(r)}{\partial r_i}$$

With Δt ~ 1fs and μs to s timescales of interest, we need 10^9-10^{15} steps.





The challenge:

- ~100,000 atoms
- Each has ~500 neighbors
  - Maintain a list of them, update ea 10 steps
- ~50M interactions/step
- ~2B FLOPS per step
- ~1ms real time per step

```
for(k=nj0; (k<nj1); k++)
{

    /* Get j neighbor index, and coordinate index */
    jnr             = jjnr[k];
    j3              = 3*jnr;

    /* load j atom coordinates */
    jx1             = pos[j3+0];
    jy1             = pos[j3+1];
    jz1             = pos[j3+2];

    /* Calculate distance */
    dx11            = ix1 - jx1;
    dy11            = iy1 - jy1;
    dz11            = iz1 - jz1;
    rsq11           = dx11*dx11+dy11*dy11+dz11*dz11;

    /* Calculate 1/r and 1/r2 */
    rinv11          = 1.0/sqrt(rsq11);

    /* Load parameters for j atom */
    qq              = iq*charge[jnr];
    tj              = nti+2*type[jnr];
    c6              = vdwparam[tj];
    c12             = vdwparam[tj+1];
    rinvsq          = rinv11*rinv11;

    /* Coulomb interaction */
    vcoul           = qq*rinv11;
    vctot           = vctot+vcoul;

    /* Lennard-Jones interaction */
    rinvsix         = rinvsq*rinvsq*rinvsq;
    Vvdw6           = c6*rinvsix;
    Vvdw12          = c12*rinvsix*rinvsix;
    Vvdwtot         = Vvdwtot+Vvdw12-Vvdw6;
    fscal           = (vcoul+12.0*Vvdw12-6.0*Vvdw6)*rinvsq;

    /* Calculate temporary vectorial force */
    tx              = fscal*dx11;
    ty              = fscal*dy11;
    tz              = fscal*dz11;

    /* Increment i atom force */
    fix1            = fix1 + tx;
    fiy1            = fiy1 + ty;
    fiz1            = fiz1 + tz;

    /* Decrement j atom force */
    faction[j3+0]   = faction[j3+0] - tx;
    faction[j3+1]   = faction[j3+1] - ty;
    faction[j3+2]   = faction[j3+2] - tz;

    /* Inner loop uses 38 flops/iteration */
}
```

I interaction

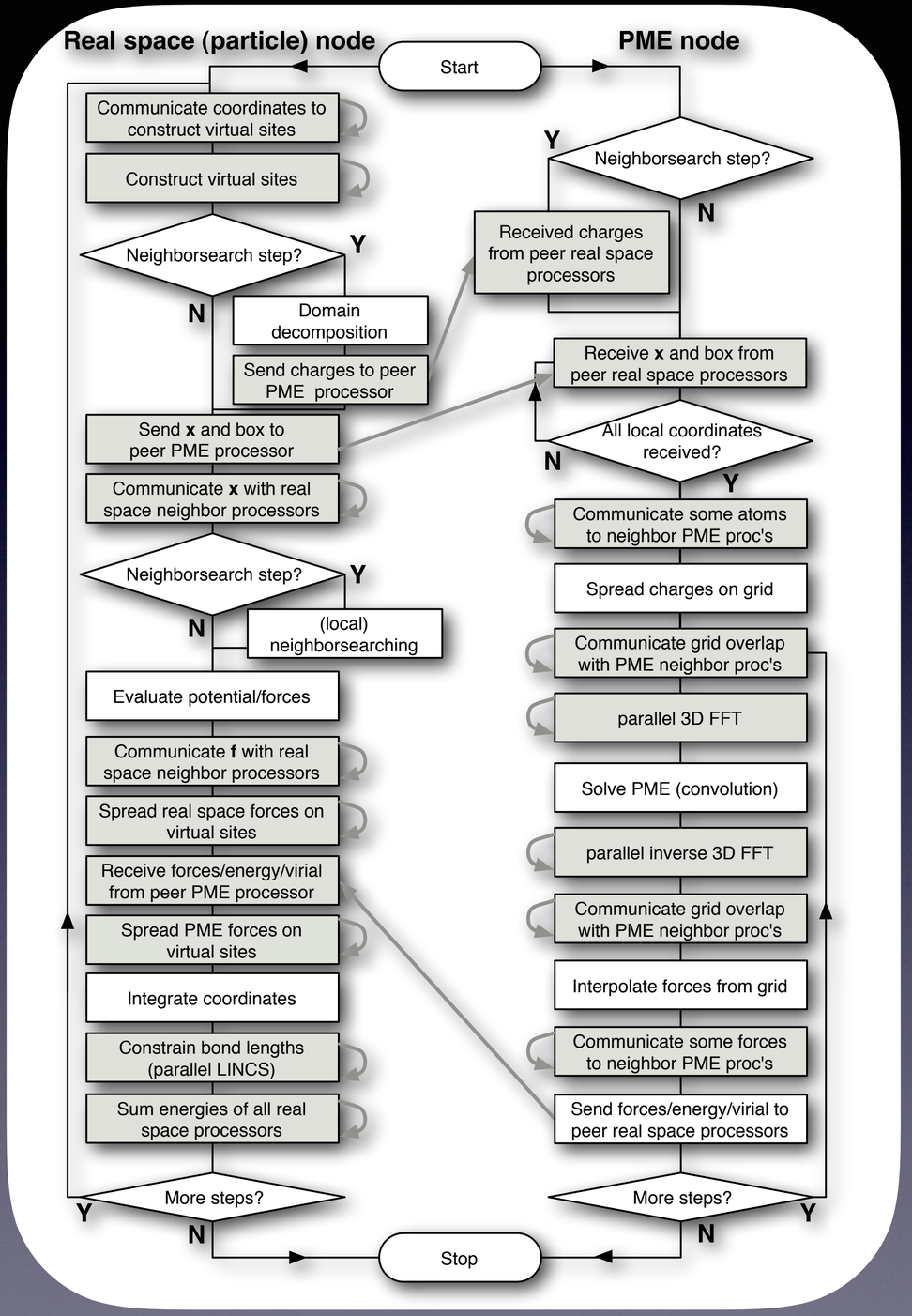
The things we do every ~100 µs

To-Do Thursday 10:15:48.004100 (simple version)

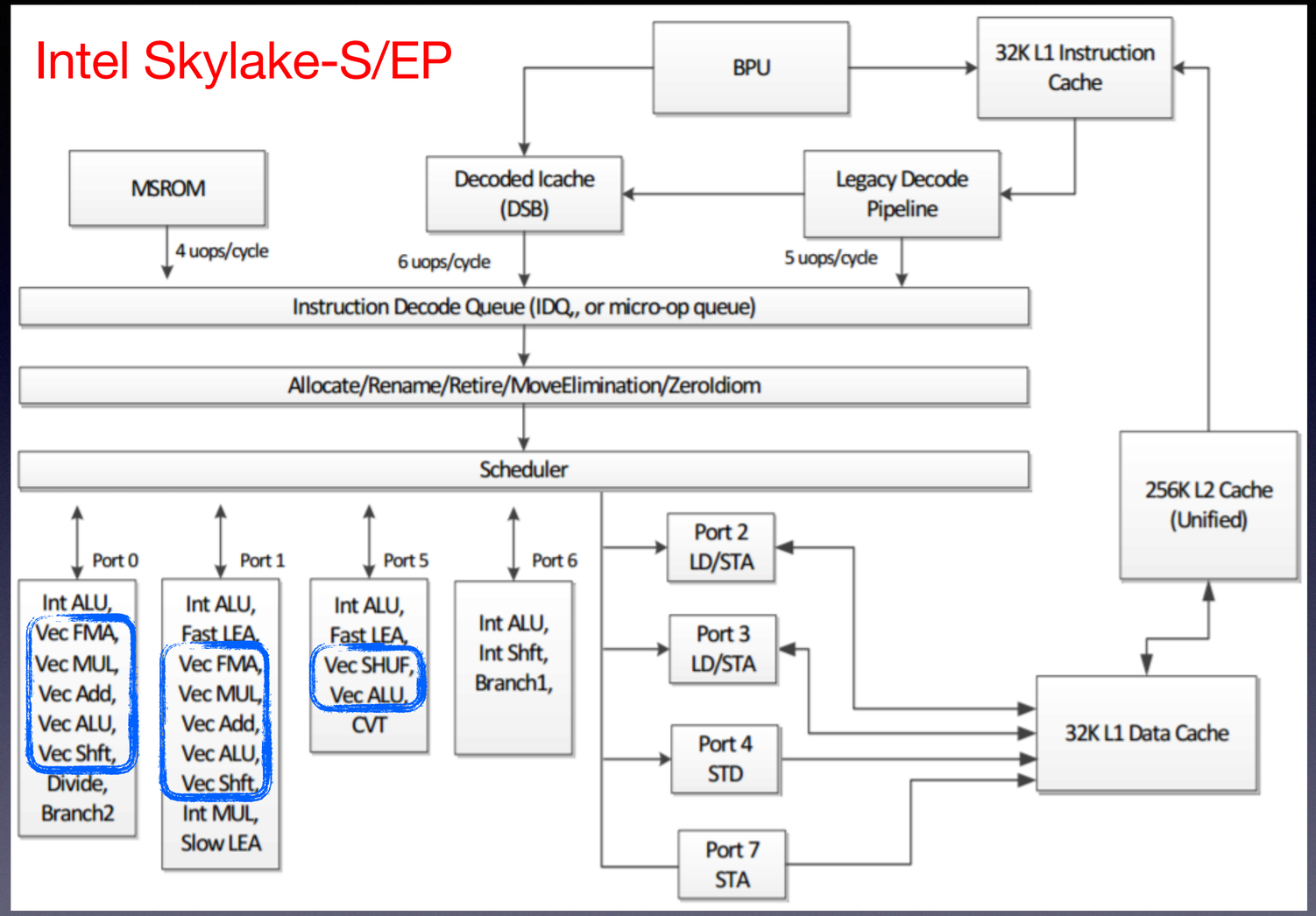
- Adjust domain decomposition [communicate]
- Communicate coordinates to/from 26 neighbor nodes
- Find atoms in proximity [communicate]
- Change charges or parameters for free energy
- Create local virtual particles [communicate]
- Send coordinates to GPU
- Calculate short-range electrostatics & VdW
- Calculate bonds
- Calculate angles
- Calculate torsions
- Perform long-range lattice summation [communicate]
- Apply external fields/forces
- Get forces back from GPU
- Send forces to 26 neighbors [communicate]
- Integrate new positions
- Constrain bonds [communicate]
- Update stats. (temperature, energy) [communicate]
- Write coordinates/forces if necessary

A fairly typical HPC application - complex & fast

Every arrow is communication



What does a modern CPU core look like?



SIMD: 128, 256, or 512 bit vectors

5 µops/core each cycle
SIMD + FMA: 64 flops @ single prec.

Up to 32 cores per chip

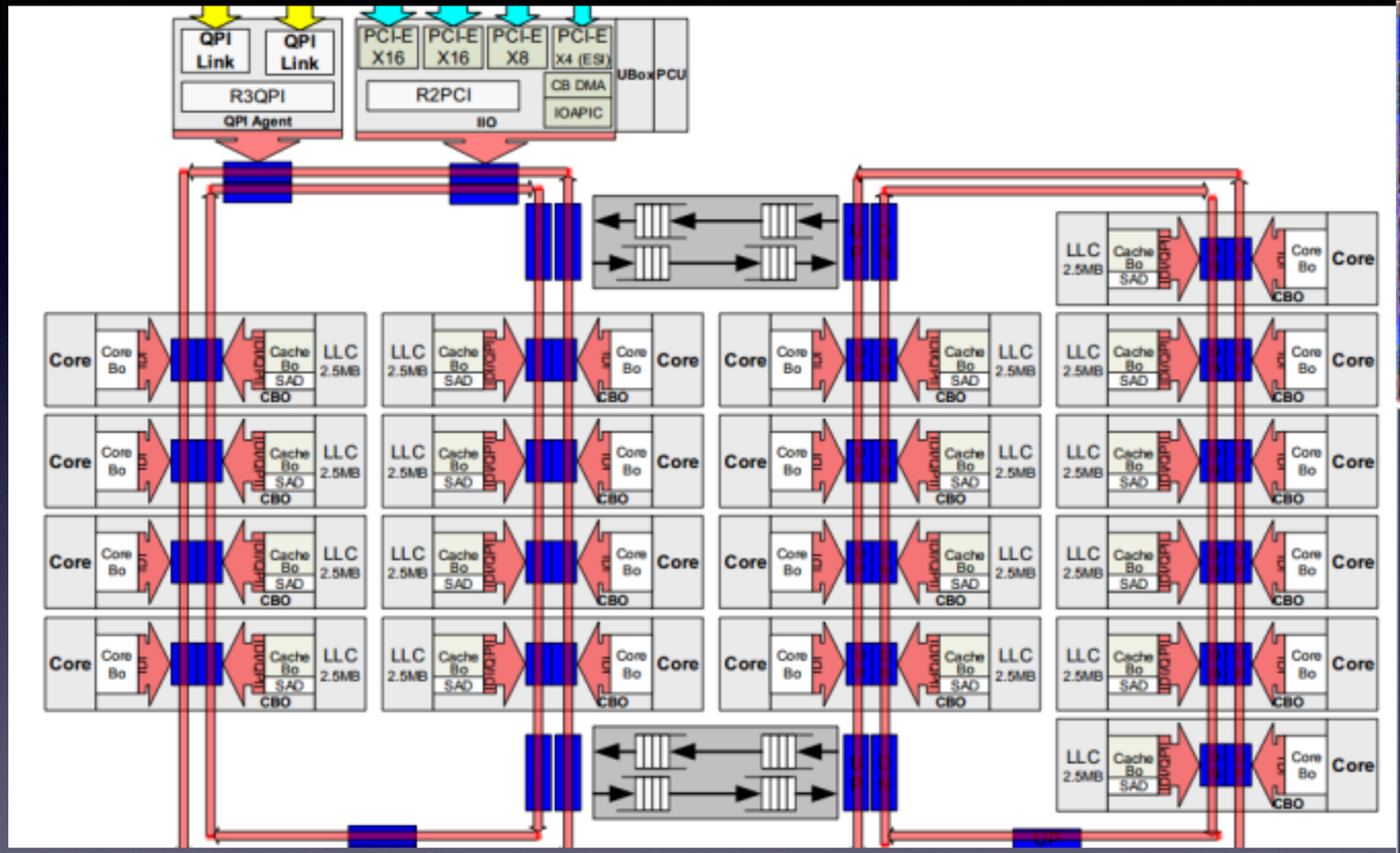
2 sockets per node

Theoretically:
320 instructions or 4096 flops per *cycle* on each node.

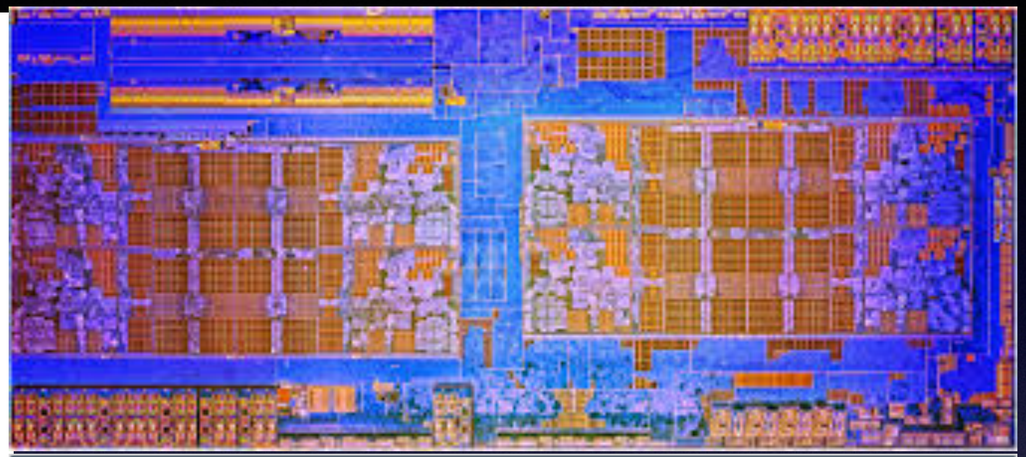
latency is ~4 cycles on Skylake.
You need *256* independent FMA flops (128 FMA operations) to saturate just a single core

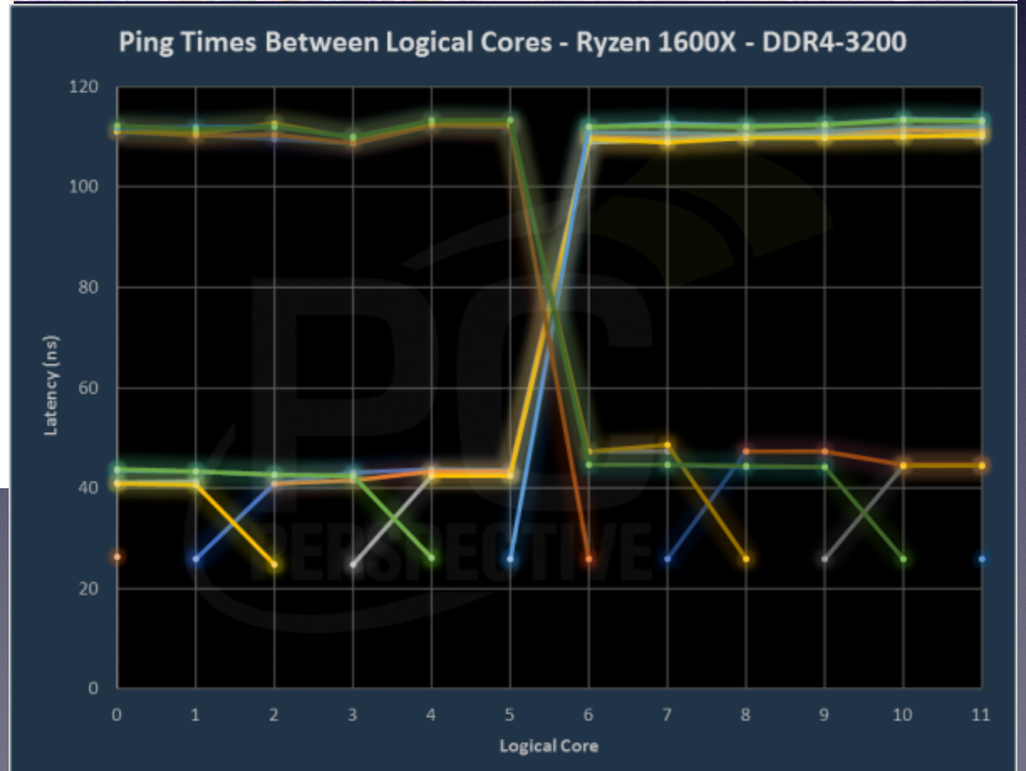
What does a modern CPU *die* look like?

Intel Haswell-EP (18 cores)



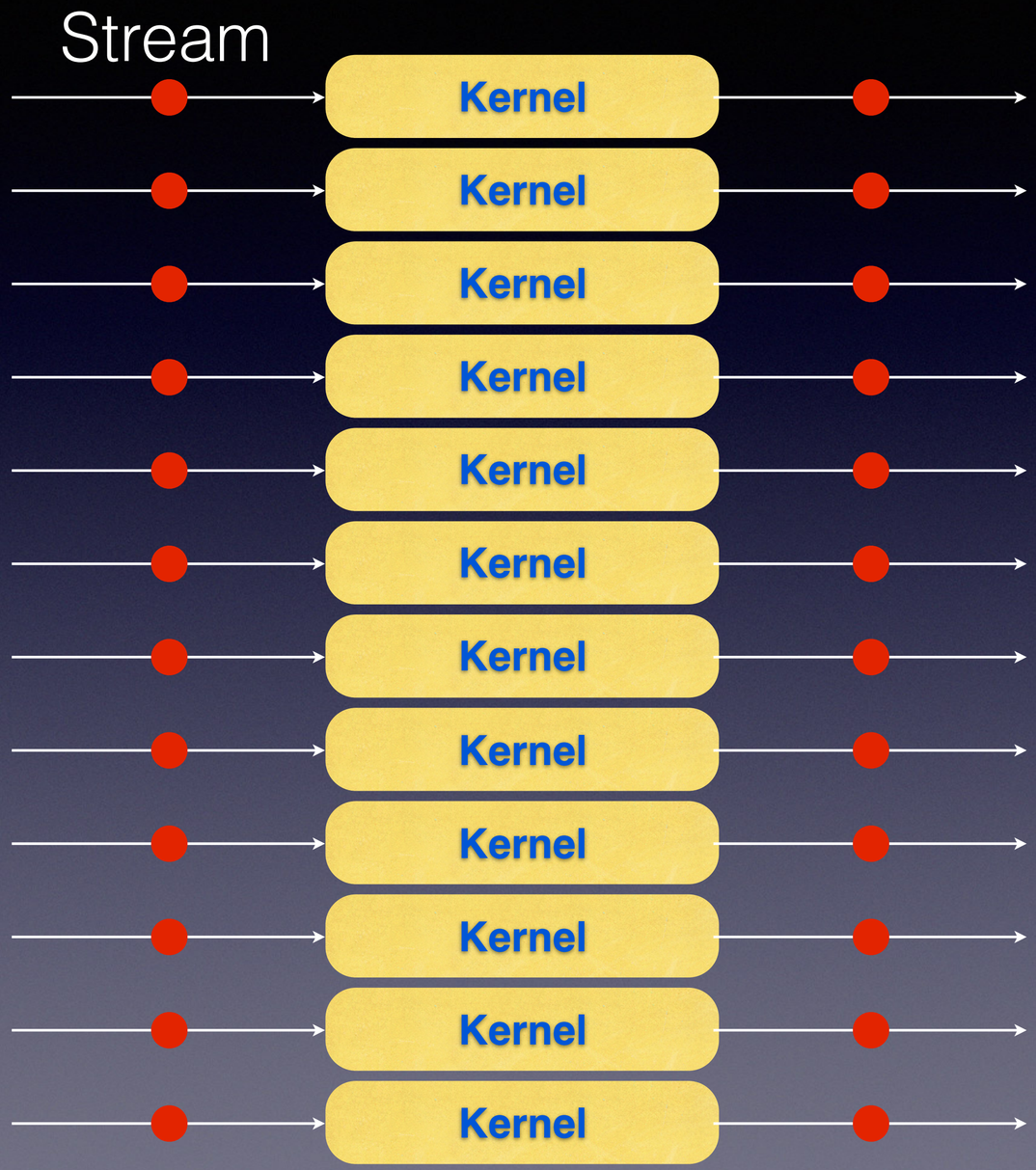
AMD Ryzen 1600x





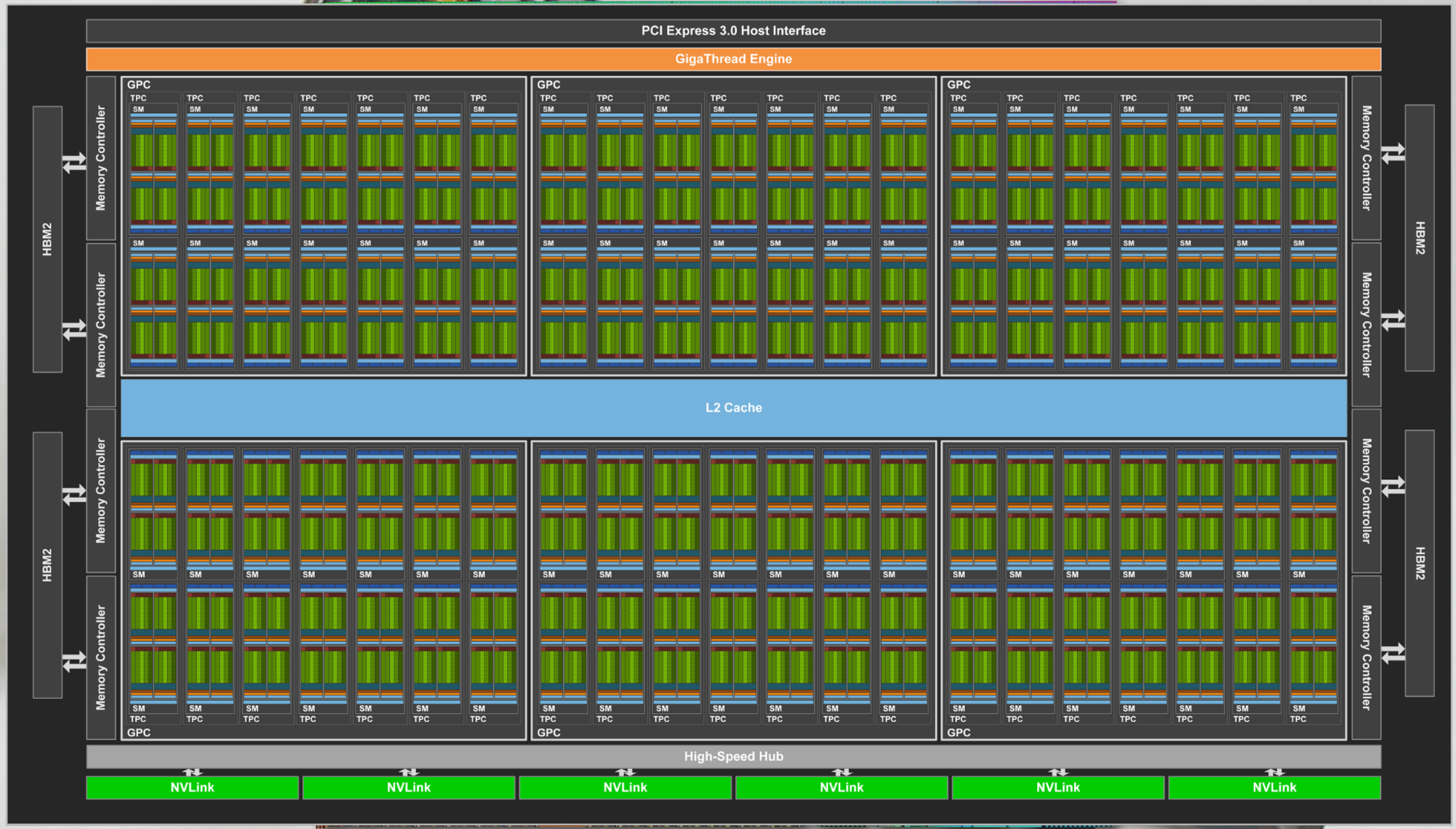
The inside of a modern node looks like a cluster with advanced network topology!

Explicit Data Parallelism



Stream=your data
Kernel=algoritm

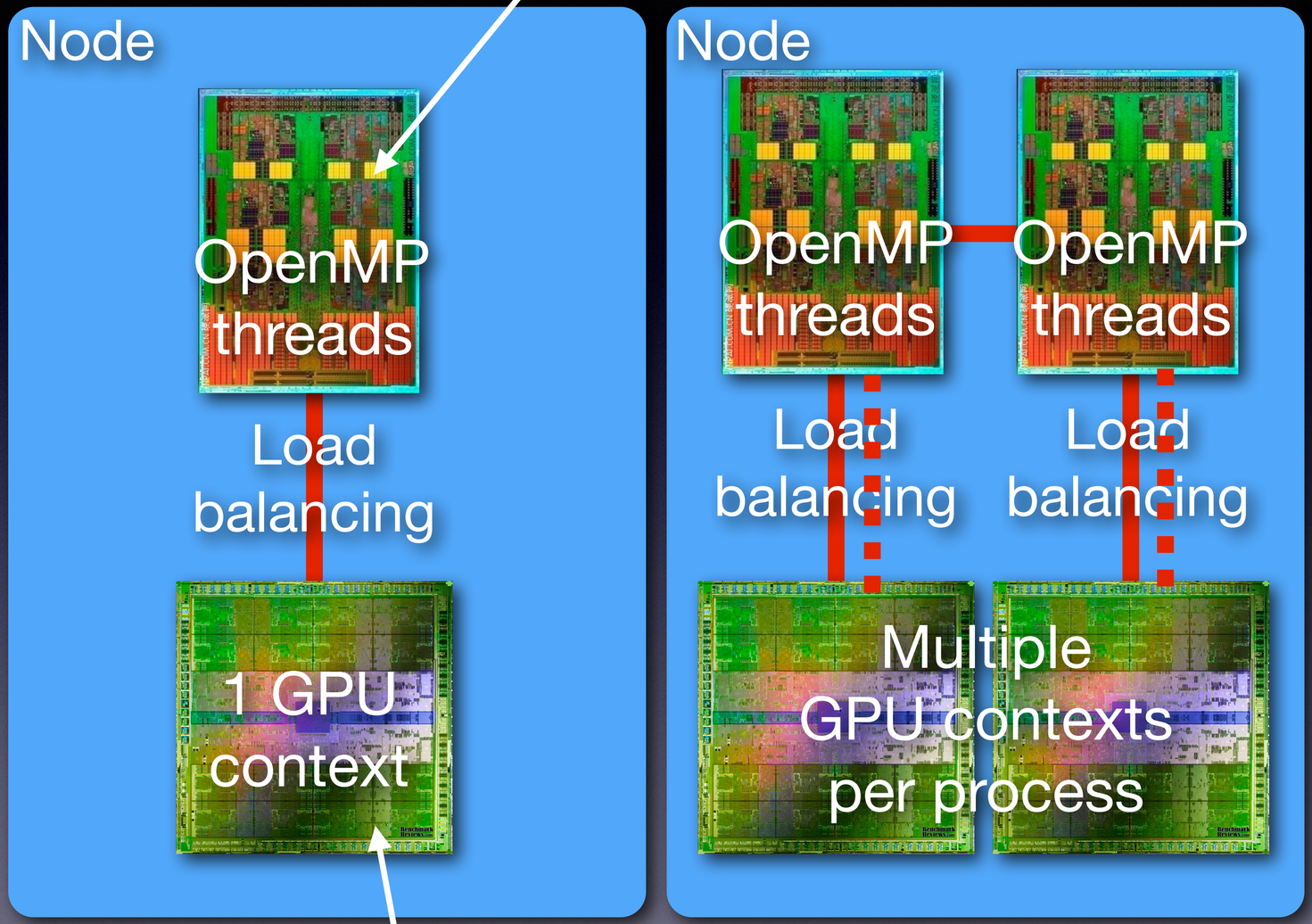
Without dependencies,
all these operations could be
done in parallel if enough
hardware was available!


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
HBM2
GPC
TPC
SM
L2 Cache
High-Speed Hub
NVLink

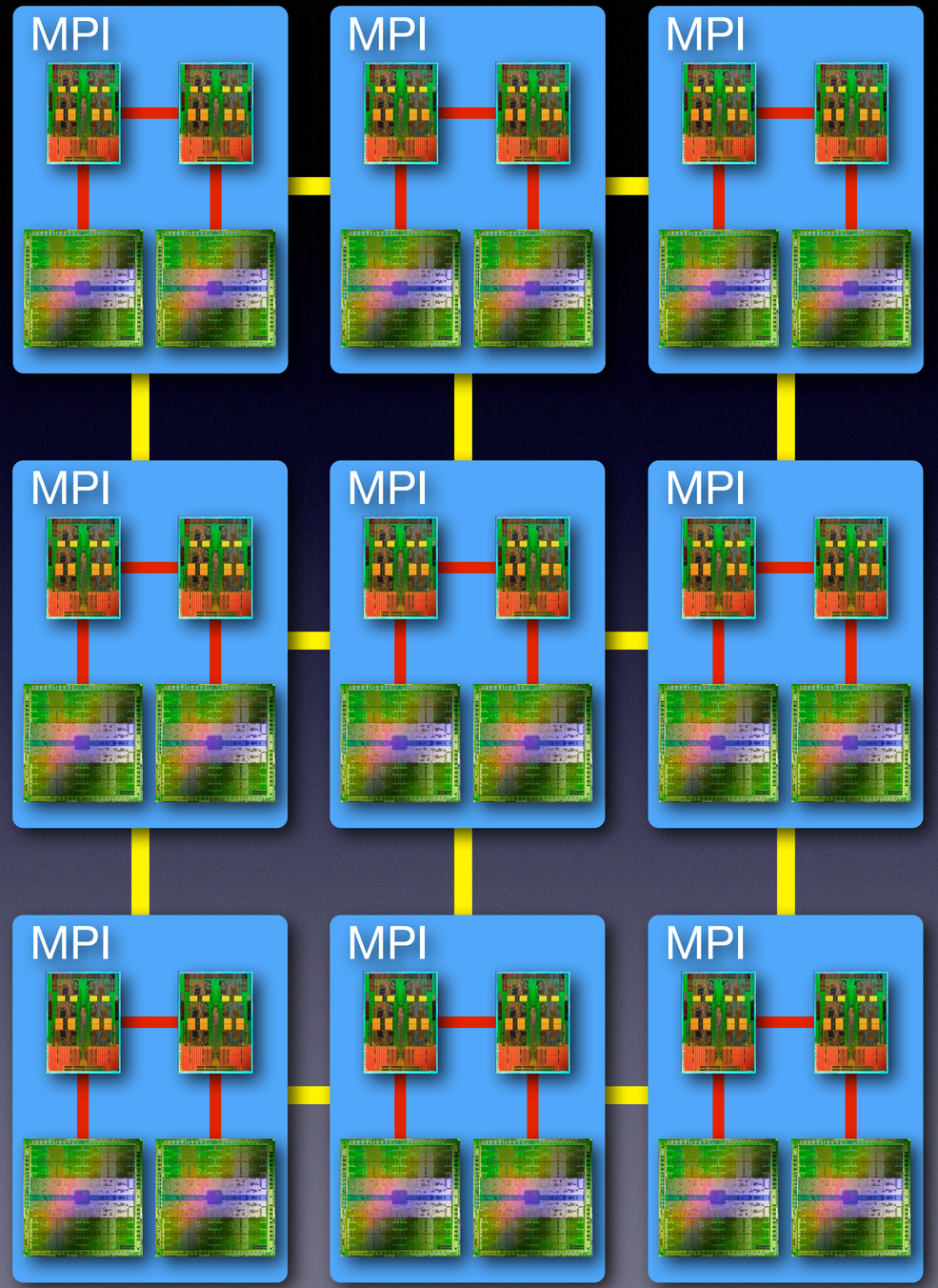
Acceleration Approaches

	GPU libraries	OpenAcc	Pure CUDA	Heterogeneous CPU/GPU
Initial effort / Expertise req.				
Generality / Portability				
Performance				
Code maintainability				
	Works if your code offloads to libraries	Always works, but success depends on you & compiler	Lots of work, assumes impl. can run entirely on GPU	Even more work, less CUDA, can use both CPU & GPU

Explicit SIMD instructions on CPUs & Xeon Phi;
each instruction does up to 32 flops

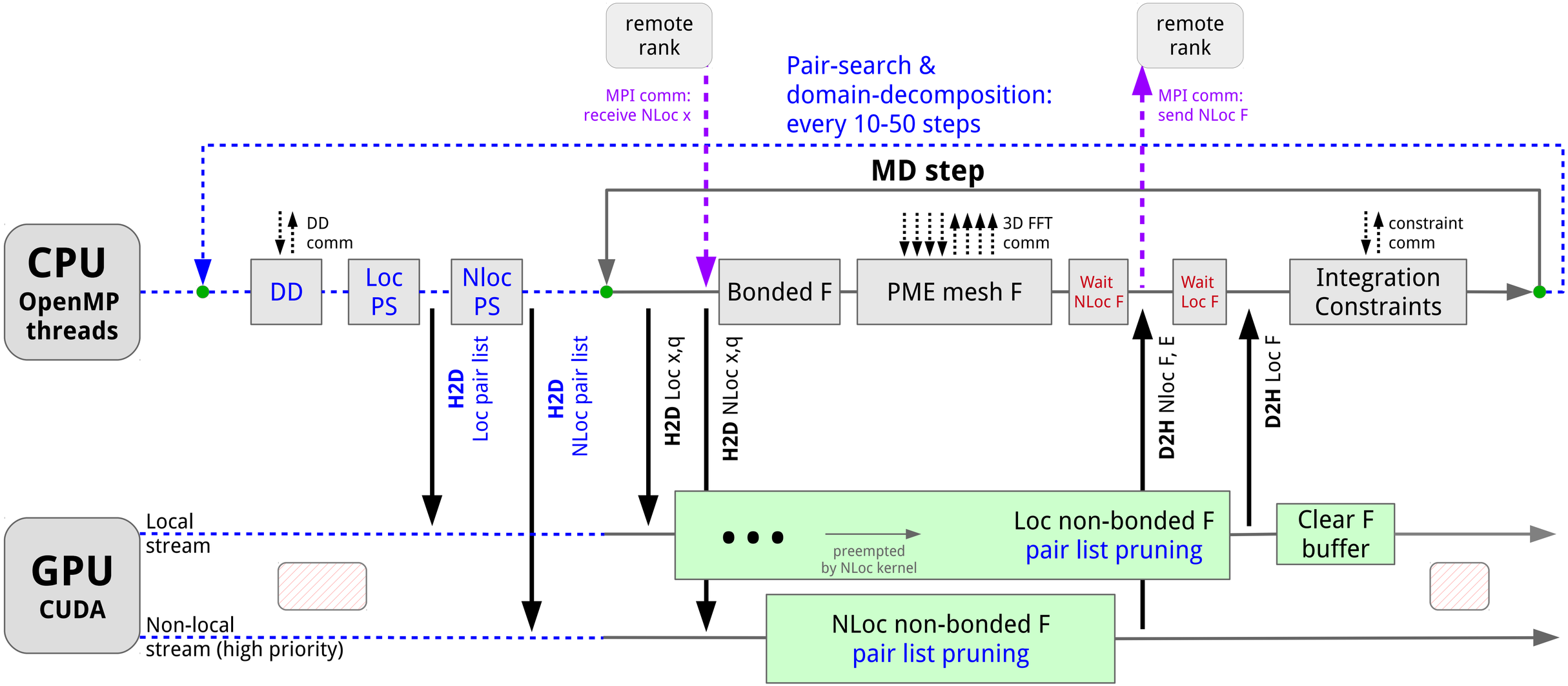


CUDA kernels on NVIDIA GPUs,
OpenCL for AMD/Intel GPUs

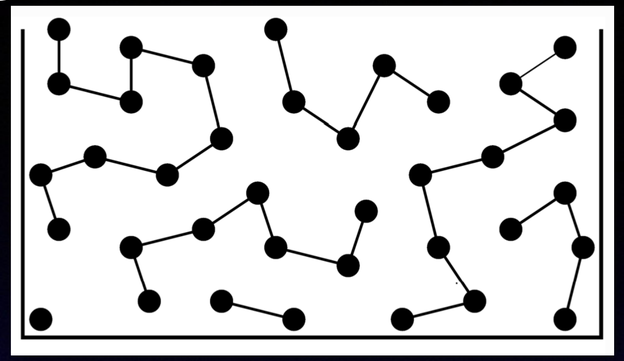


Strategy:

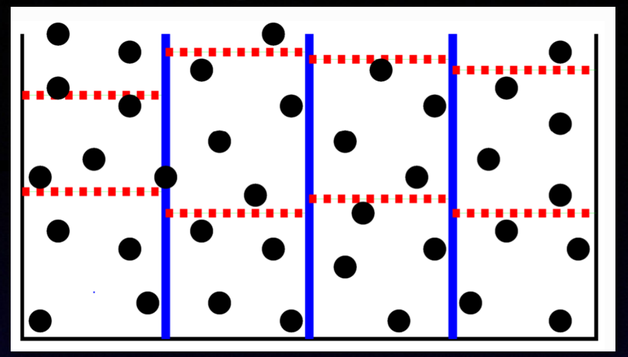
1. Profile
2. Offload largest bottleneck to GPU



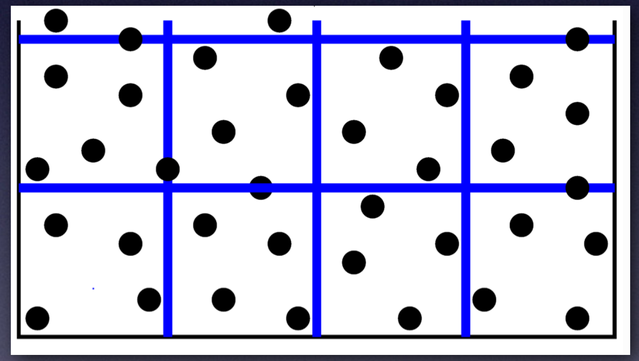
From neighborlists to cluster proximity lists: Revisit everything



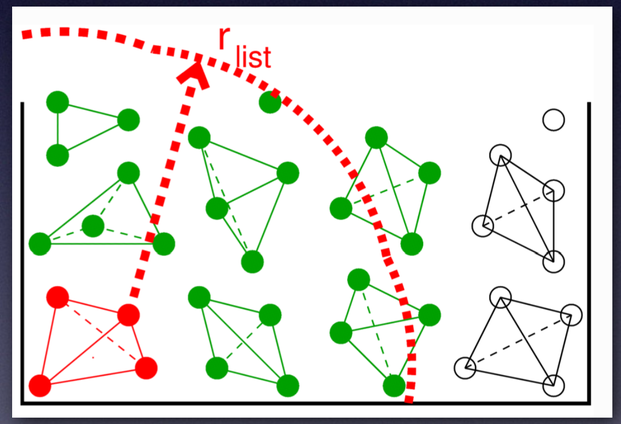
x,y grid
z sort
z bin



x,y,z
gridding



Cluster pairlist



Organize as tiles with all-vs-all interactions:

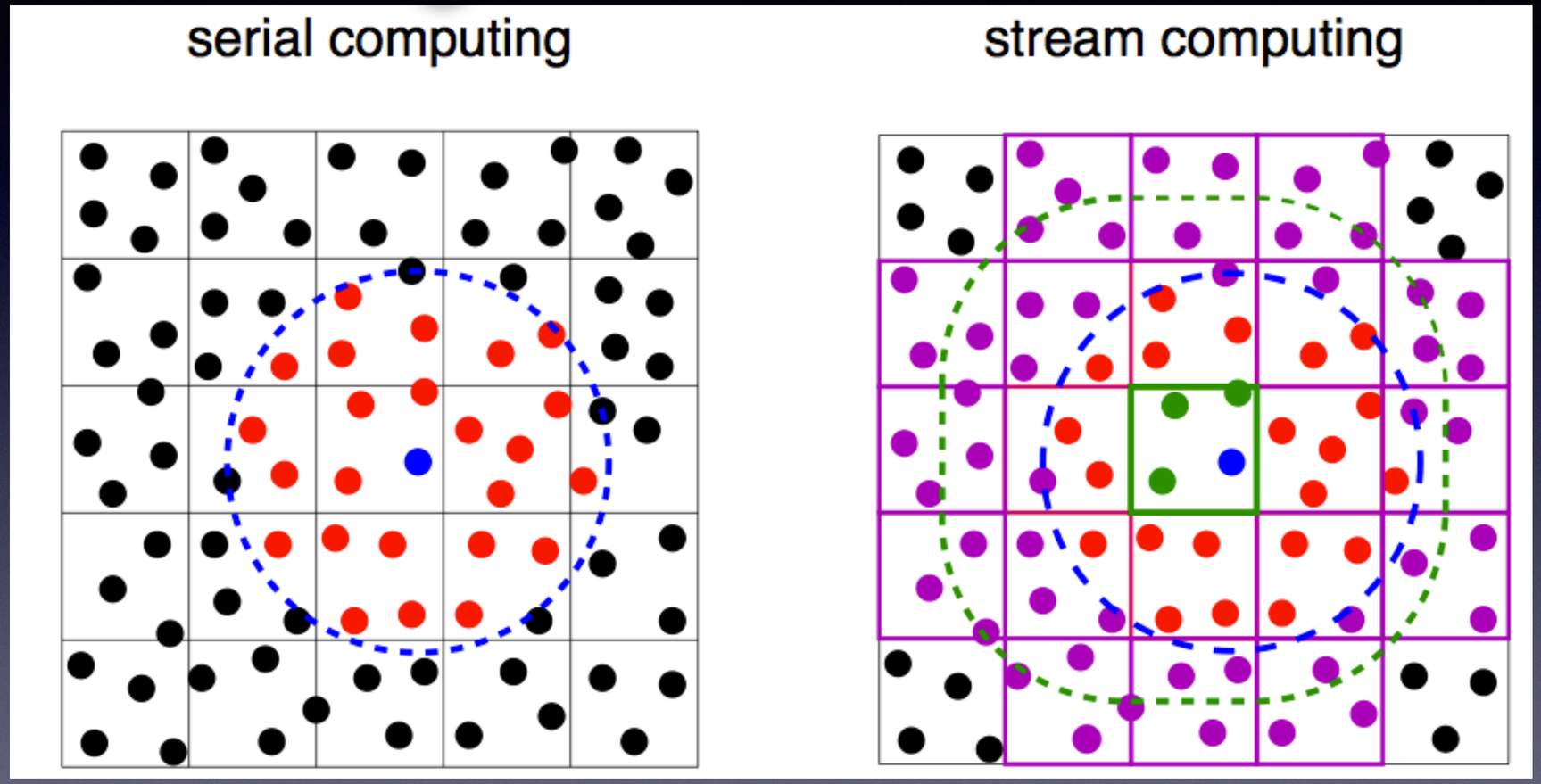
i=3:	5	6	9	12	15	17	18	25	32	…		
i=4:	7	8	9	11	12	15	17	25	32	43	54	…
…	8	9	10	11	12	13	19	20	…			

Tile interaction algorithms:
Load N atoms, compute N^2 forces

The Link-cell algorithm: Load 1 atom, calculate 1 interaction
Verlet, Phys Rev 159, 98-103 (1967)]

This creates a new problem: Tiling circles is difficult

You want to calculate interactions with red neighbors



Lots of wasted FLOPS!

- You need a lot of cubes to cover a sphere
- All interactions beyond cutoff need to be zero

The big gain of heterogeneous acceleration: Very little CUDA required

```
n_cuda lindahl$ ls -ltar

lindahl  staff  13012 Apr 18 15:10 nbnxn_cuda_types.h
lindahl  staff   9155 Apr 18 15:10 nbnxn_cuda_kernels.cuh
lindahl  staff  21576 Apr 18 15:10 nbnxn_cuda_kernel_utils.cuh
lindahl  staff  20945 Apr 18 15:10 nbnxn_cuda_kernel.cuh
lindahl  staff   1965 Apr 18 15:10 CMakeLists.txt
lindahl  staff  39049 Apr 18 15:10 nbnxn_cuda_data_mgmt.cu
lindahl  staff   3667 Apr 18 15:10 nbnxn_cuda.h
lindahl  staff  30920 May 22 09:13 nbnxn_cuda.cu
lindahl  staff   2686 May 22 09:13 ..
lindahl  staff    340 May 22 09:13 .
```

cuda_kernel.c

cuda_kernel.c 〉 No Selection

```
                                {
                                    /* load the rest of the i-atom parameters */
                                    qi      = xqbuf.w;
#ifdef IATYPE_SHMEM
                                    typei   = atib[i * CL_SIZE + tidxi];
#else
                                    typei   = atom_types[ai];
#endif

                                    /* LJ 6*C6 and 12*C12 */
#ifdef USE_TEXOBJ
                                    c6      = tex1Dfetch<float>(nbparam.nbfp_texobj, 2 * (ntypes * typei + typej));
                                    c12     = tex1Dfetch<float>(nbparam.nbfp_texobj, 2 * (ntypes * typei + typej) + 1);
#else
                                    c6      = tex1Dfetch(nbfp_texref, 2 * (ntypes * typei + typej));
                                    c12     = tex1Dfetch(nbfp_texref, 2 * (ntypes * typei + typej) + 1);
#endif                              /* USE_TEXOBJ */


                                    /* avoid NaN for excluded pairs at r=0 */
                                    r2      += (1.0f - int_bit) * NBNXN_AVOID_SING_R2_INC;

                                    inv_r   = rsqrt(r2);
                                    inv_r2  = inv_r * inv_r;
                                    inv_r6  = inv_r2 * inv_r2 * inv_r2;
#if defined EXCLUSION_FORCES
                                    /* We could mask inv_r2, but with Ewald
                                     * masking both inv_r6 and F_invr is faster */
                                    inv_r6  *= int_bit;
#endif                              /* EXCLUSION_FORCES */

                                    F_invr  = inv_r6 * (c12 * inv_r6 - c6) * inv_r2;
#if defined CALC_ENERGIES || defined LJ_POT_SWITCH
                                    E_lj_p  = int_bit * (c12 * (inv_r6 * inv_r6 + nbparam.repulsion_shift.cpot)*ONE_TWELVETH_F -
                                                         c6 * (inv_r6 + nbparam.dispersion_shift.cpot)*ONE_SIXTH_F);
#endif

#ifdef LJ_FORCE_SWITCH
#ifdef CALC_ENERGIES

                                    calculate_force_switch_F_E(nbparam, c6, c12, inv_r, r2, &F_invr, &E_lj_p);

#else

                                    calculate_force_switch_F(nbparam, c6, c12, inv_r, r2, &F_invr);

#endif /* CALC_ENERGIES */
#endif /* LJ_FORCE_SWITCH */
```

A total of ~3000 lines of CUDA, compared to 3M lines of C++

… so we wrote OpenCL kernels too!

Make good use of both GPU & CPU

- Use the CPU to pre-calculate or optimize data structures, so there is less work for the GPU to do in your kernels
- Easier to implement more complex optimization on CPU
- Advanced multi-node domain decomposition easier on CPU
- Run some parts of the algorithm on the CPU (avoid wasting flops)

1. It's important to keep the GPU busy
2. … but it doesn't have to be busy 100% of the time!
3. A CUDA GPU running at 100% will get hot, and clock down
4. NVML "application clocks" effectively overclock the GPU on-the-fly when you have less than 100% utilization

Think of a node as a collection of compute & communication devices - use them all!

Kernel latency is key for heterogeneous acceleration

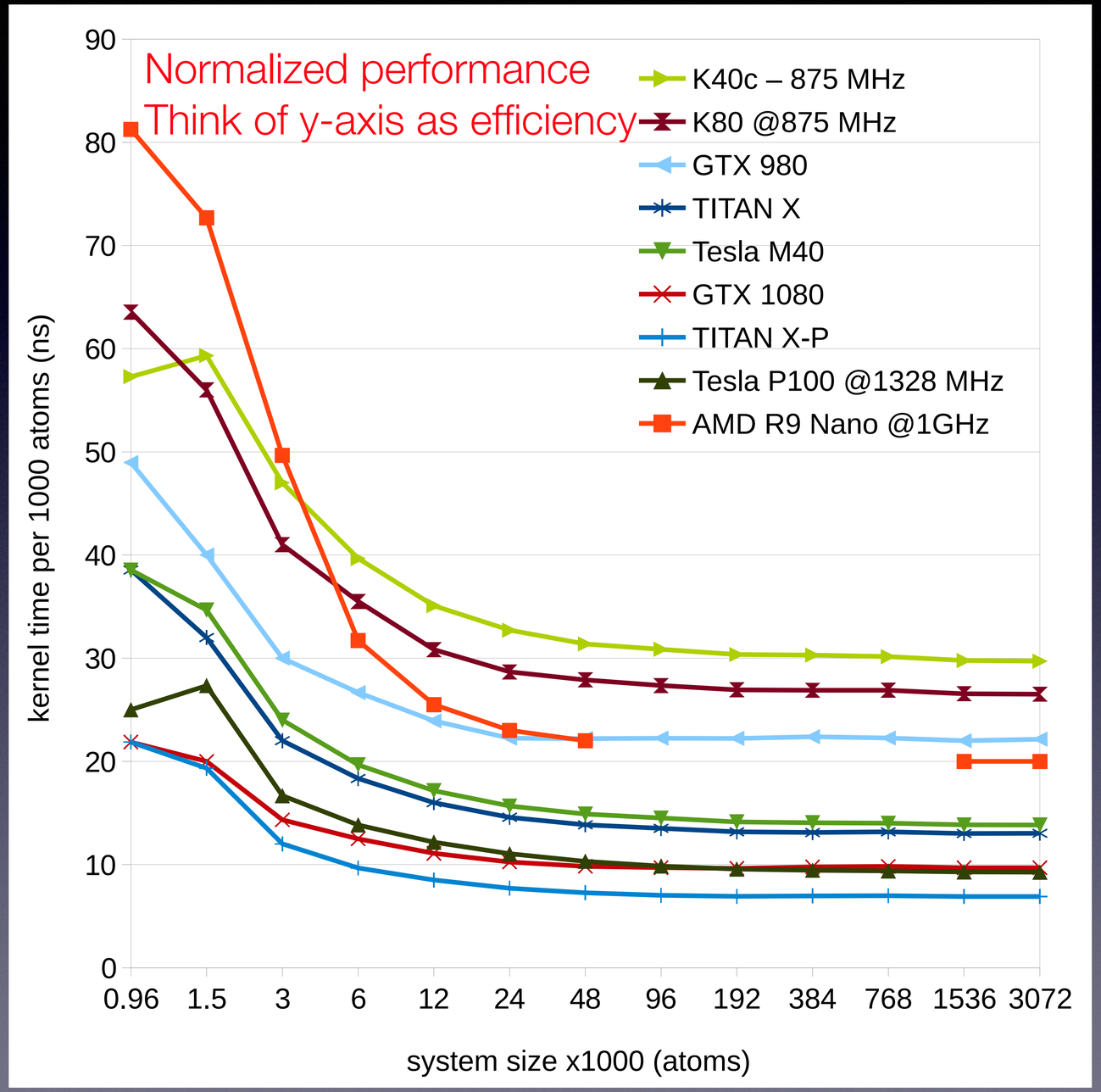
CUDA runtime overheads

- **Horrible ECC + `cudaStreamSynchronize` overhead gone (mostly)!**
- But observed:
  - single stream (without DD): 35-60 us
  - two streams (with DD): 50-100 us
  - `cudaStreamSynchronize` ~30% faster than `cudaEventSynchronize`

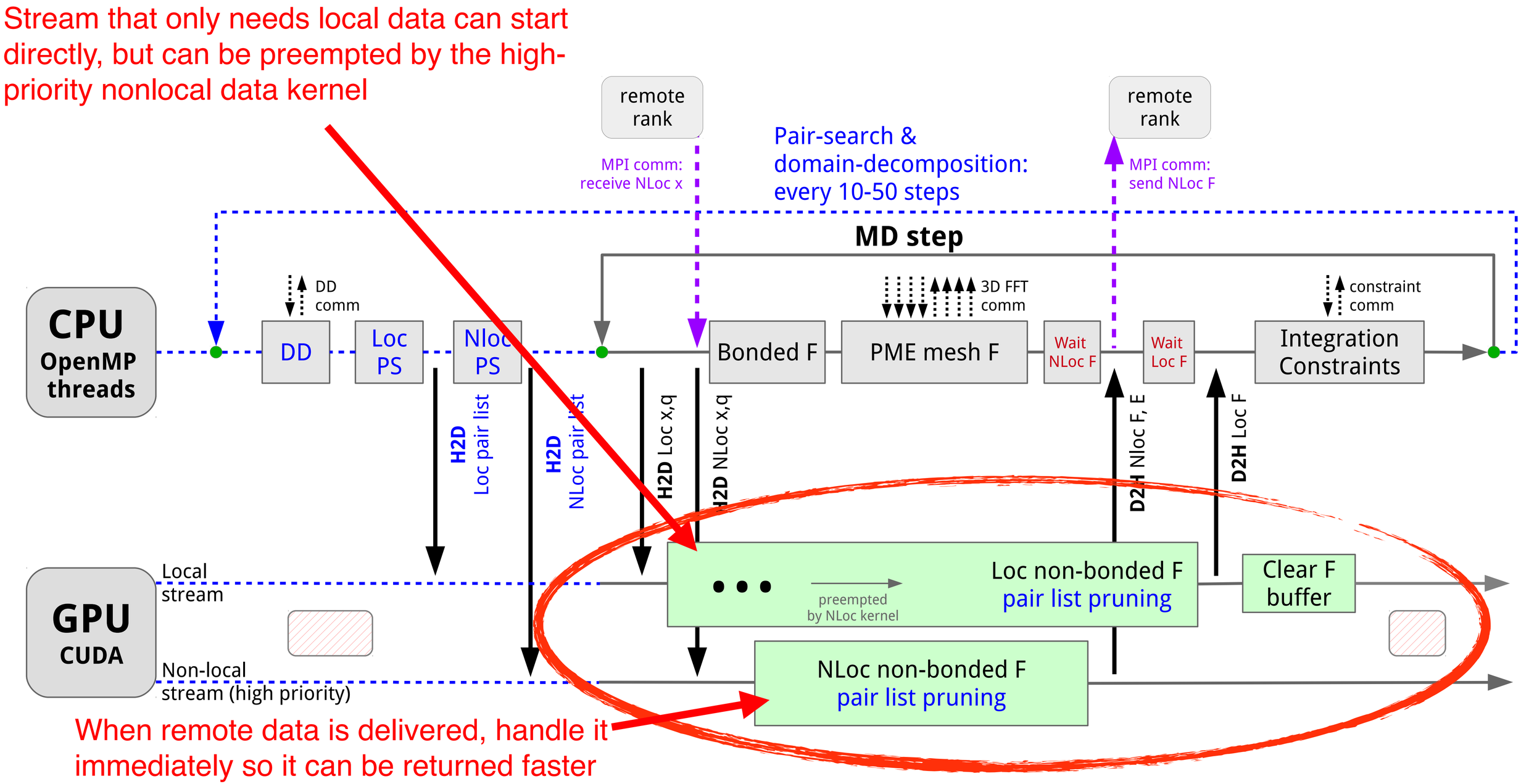
CUDA RT call	single-stream (us/step)	dual-stream (us/step)
H2D pair list	0.3	1.1
H2D x+q	7.2	9.6
NB kernel	9.8	14.8
D2H forces	5.2	9.2
D2H E+shift F	1.4	1.4
cudaStreamSync	3.4	2.3
Clear kernel	9.2	8.8
Total	**37.2**	**49.5**

Hw/sw setup:
- CUDA 5.0 + driver API 5.0: 304.54, 304.60
- Ubuntu 12.04 AMD64
- Intel SB + GTX 680/K20

- On Titan-XP, our force kernel completes in 22ns for 1000 atoms
- But then the efficiency is less than 50%
- From 24k atoms, we have full efficiency
- If we could reach this for the smallest system, that kernel would run 2.5x faster
- This limits parallelization (strong scaling)



Exploiting multiple & high-priority streams



Revisiting Amdahl's law - give GPU more work

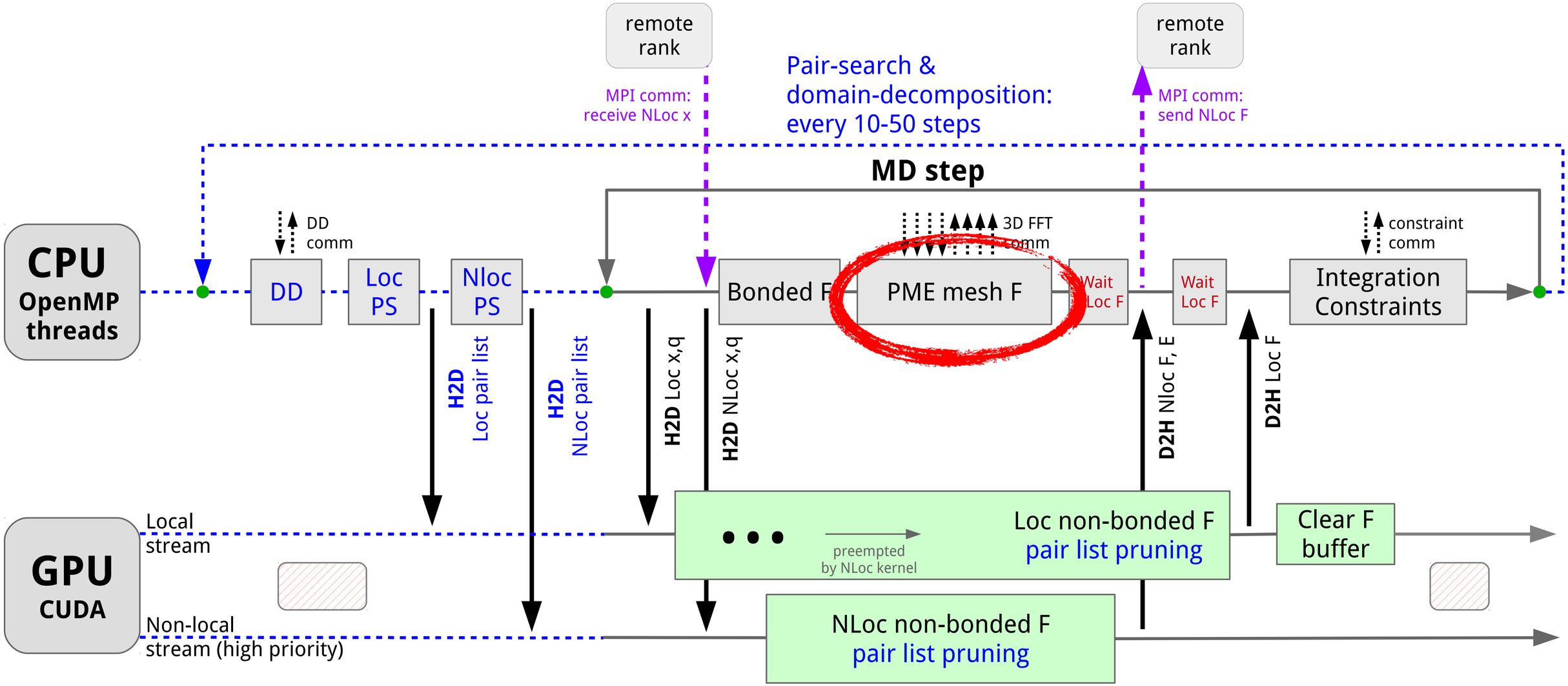
The least parallel part of the code (or at least slowest piece of hardware) will eventually dominate execution completely and limit scaling

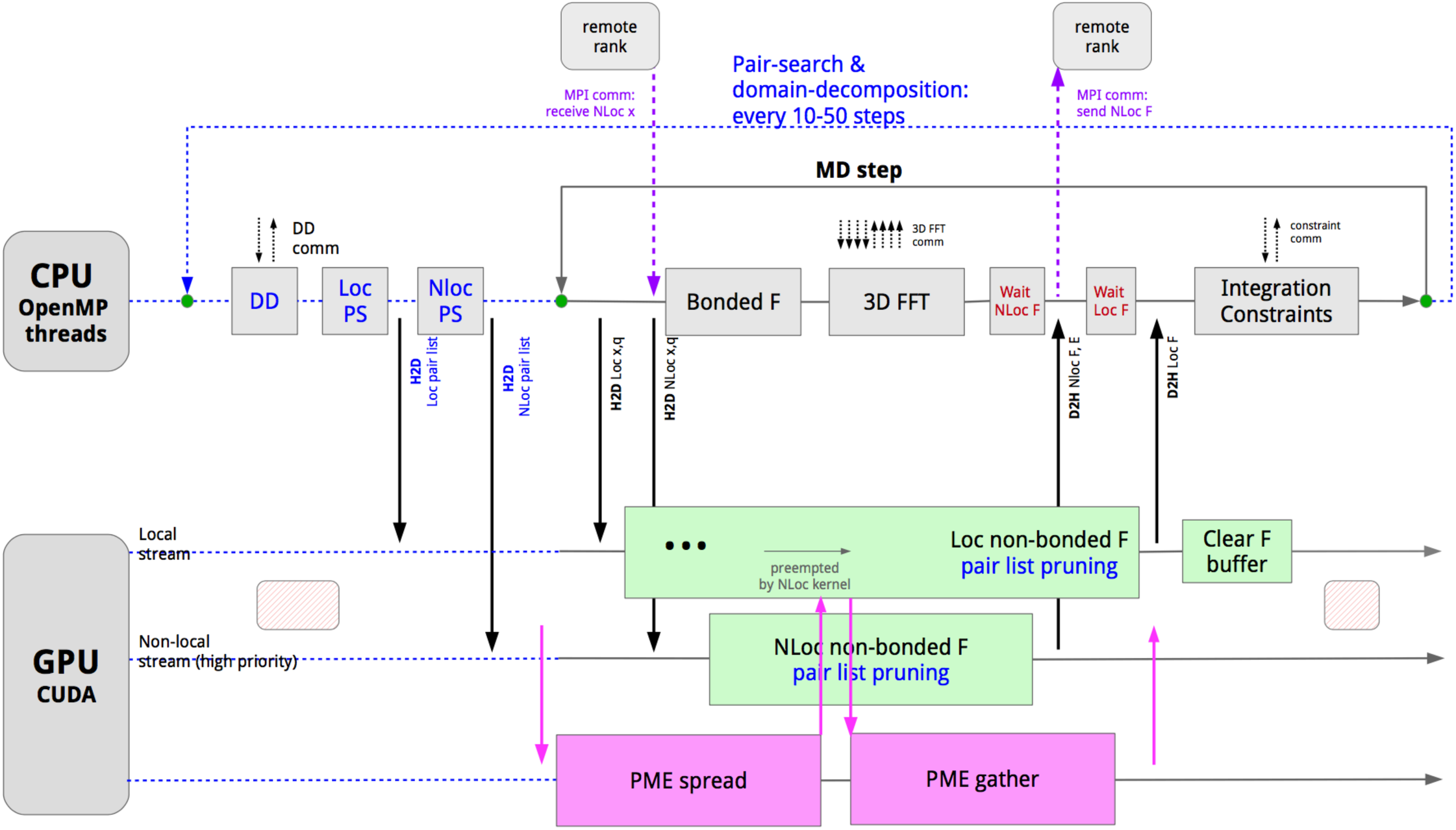
Thanks to heterogeneous parallelism and efficient CPU-side algorithms, GROMACS frequently outperforms GPU-only implementations - and yet we only need a few thousand lines of CUDA.

But… GPU performance grows faster than CPU performance, and sometimes we want to put a high-end GPU in an old low-end CPU box

Our CPUs used to wait for the GPUs, now it's often the opposite

The new bottleneck (for slow CPUs) is the PME algorithm
3D grid spreading, FFTs, convolution, iFFT, interpolation



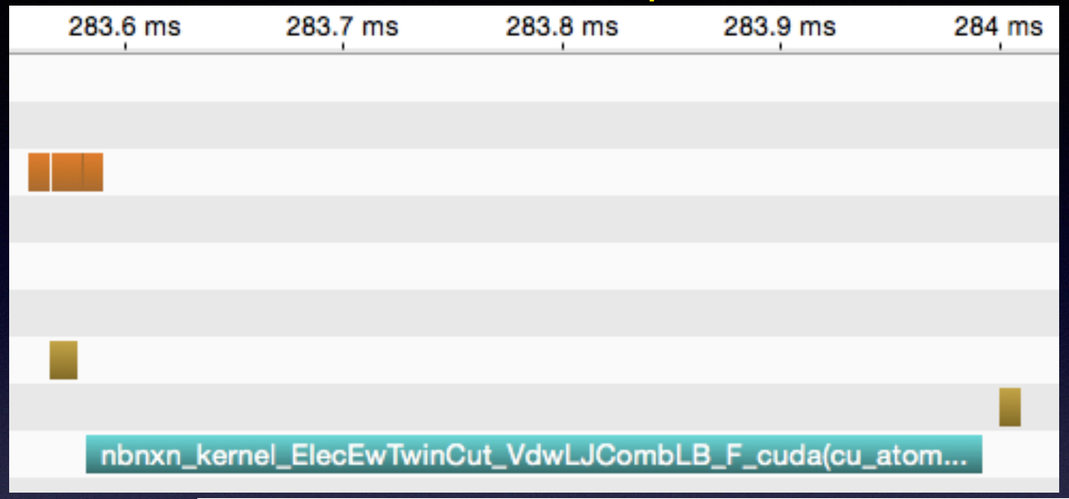


PME offload into separate stream

GPU Timestep profile before/after PME offload (P100)

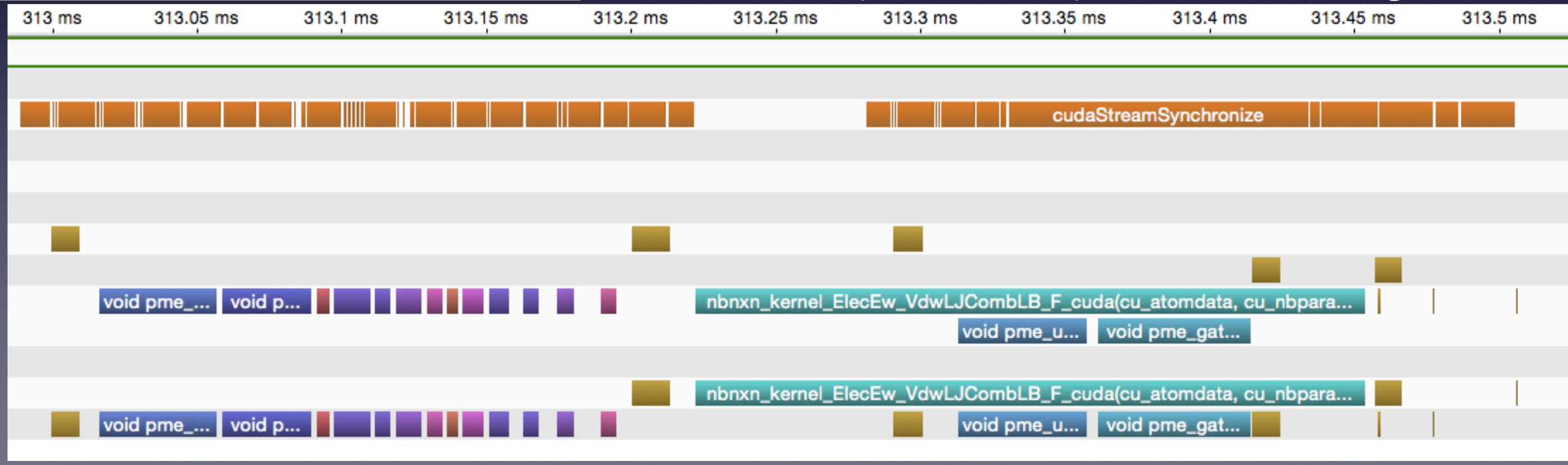
Highly challenging small system (25k atoms), very fast iterations.

Much harder, but important for multi-GPU scaling

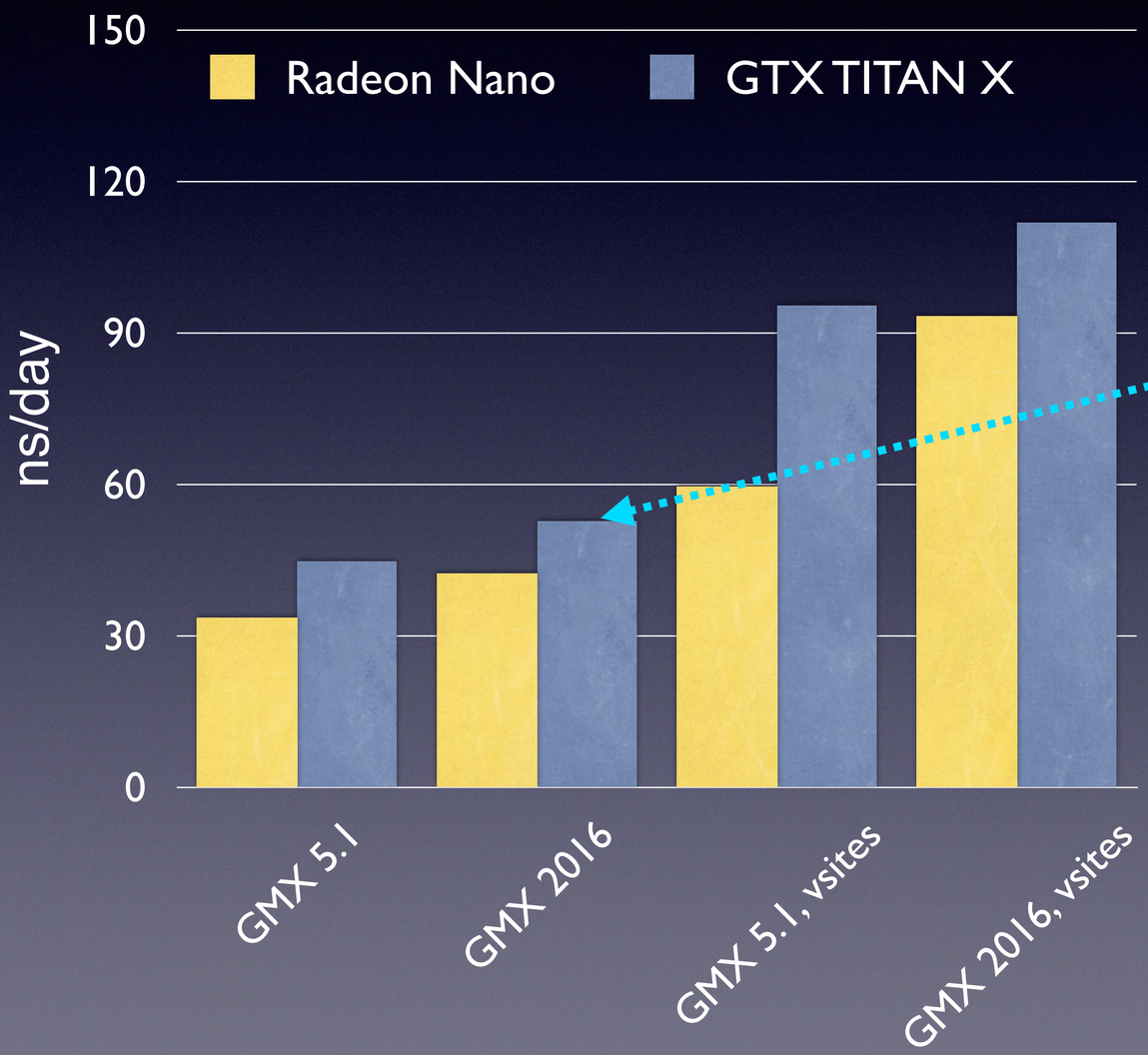


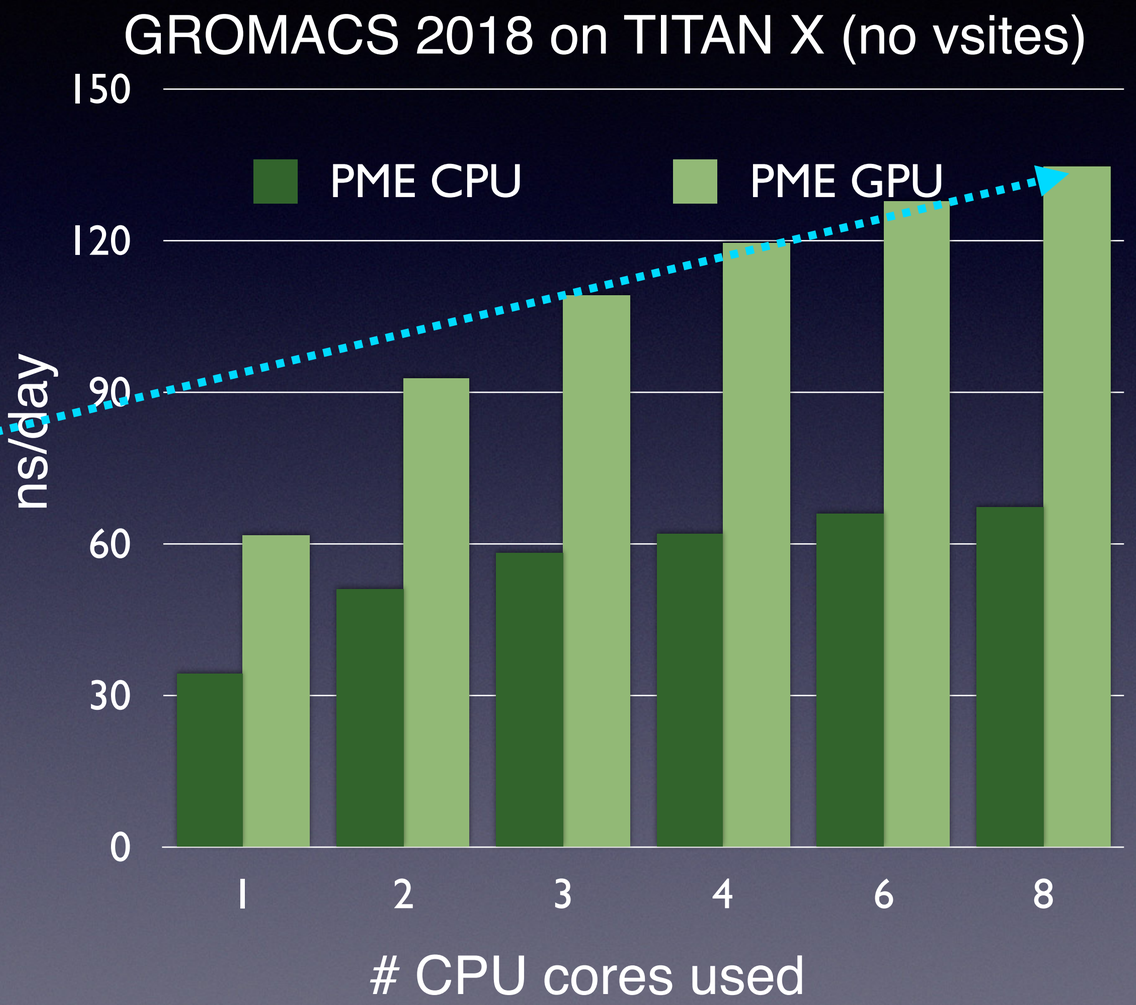
450µs for a complete step - note x scale!

Harder to retain full compute throughput than communication BW when scaling to small sys. Scheduling limited below - kernels should overlap. Lauch operations limiting us.



Multiple streams, neighbor list pruning, PME offload: 2.7X better!
8-10X faster than hand tuned SIMD assembly running on CPU





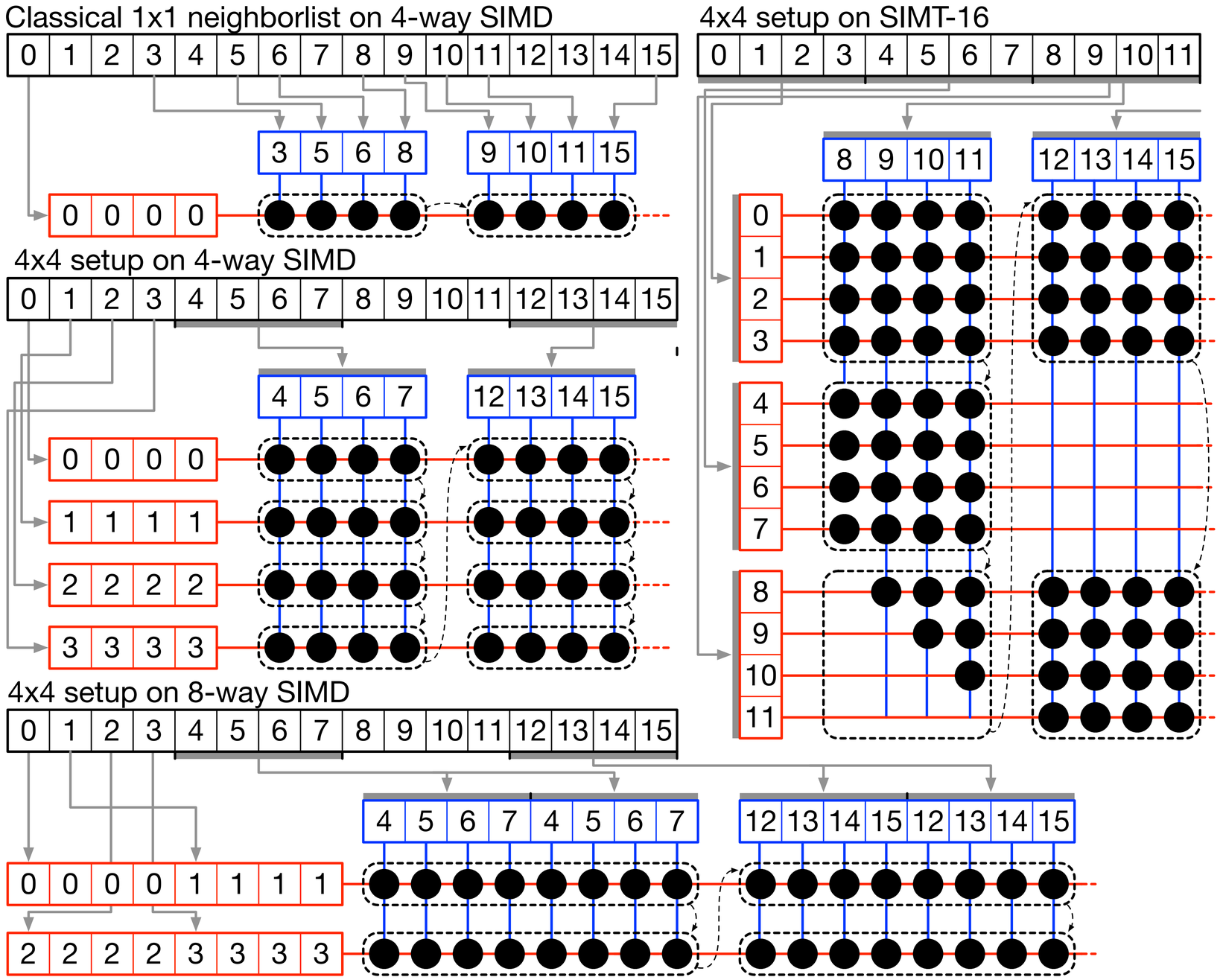
Alcohol dehydrogenase

95.6k atoms in dodecahedron box

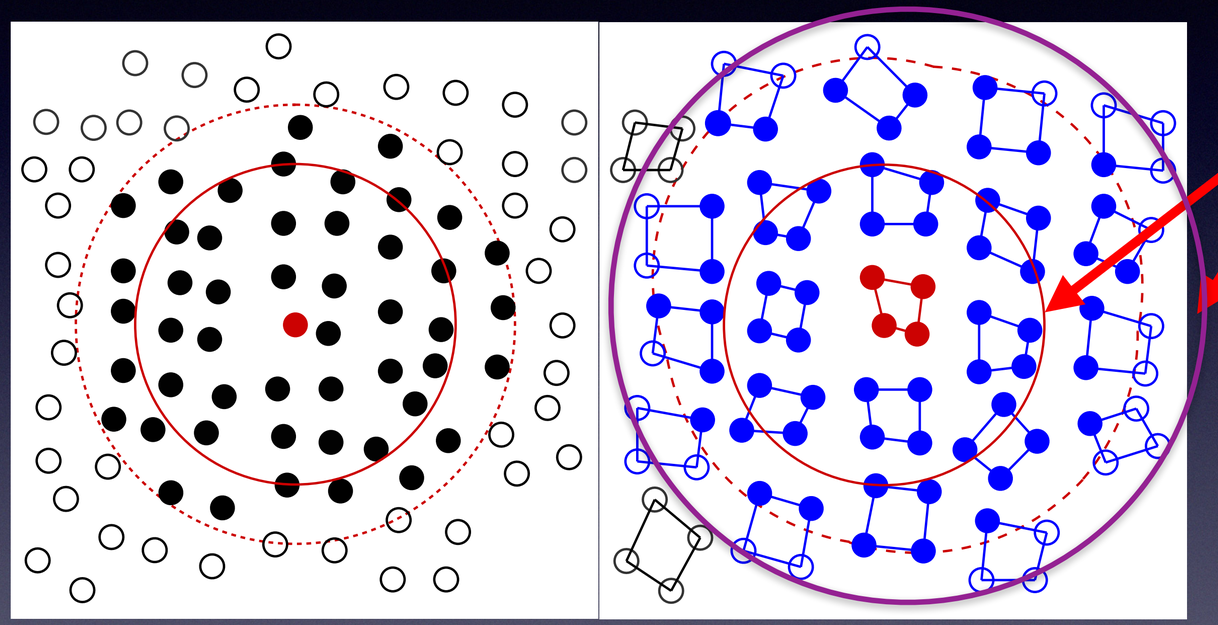
Bringing the Performace back to the CPU

Unified GPU/CPU architecture - completely portable

- CUDA
- OpenCL
- Intel MIC
- x86 SSE2
- x86 SSE4.1
- x86 AVX
- x86 AVX-128-FMA
- x86 AVX2
- x86 AVX2_128
- x86 AVX-512F
- x86 AVX-512ER
- Arm Neon
- Arm64 Asimd
- IBM QPX
- IBM VMX
- IBM VSX
- Fujitsu HPC-ACE



Atom clustering and pair list buffering

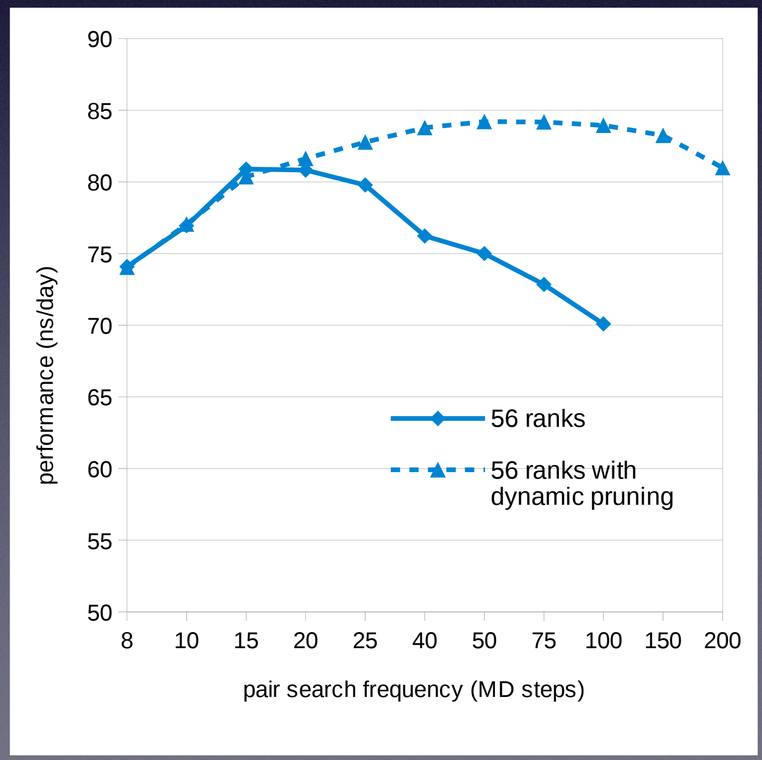


We can
Adjust the
size of this buffer

Larger buffer
means more
calculations, but
we can update
the neighbor list
less frequently

New in GROMACS-2018:
dual-pair list buffer:

- Use very large buffers, and prune it every few steps
- reduces overhead
- less sensitive to parameters



Strong caling issues - challenges at 100µs per iteration

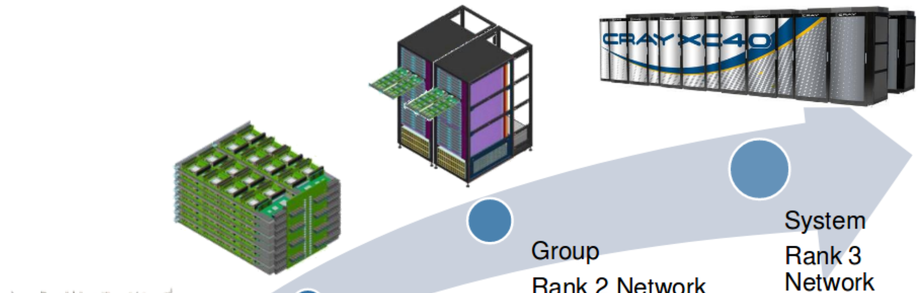
- The 3D-FFT in PME
- MPI overhead - we need MPI_Put_notify()
- OpenMP barriers take significant time
- Load imbalance
- CUDA API overhead can be 50% of CPU time
- Too many GROMACS options to tweak manually

Large performance loss due to imbalance and network speed variation on Cray XC (interference from other jobs on the “smart” network)

<1 millisecond

It would help a lot to have more control over node placement

Cray XC System Building Blocks

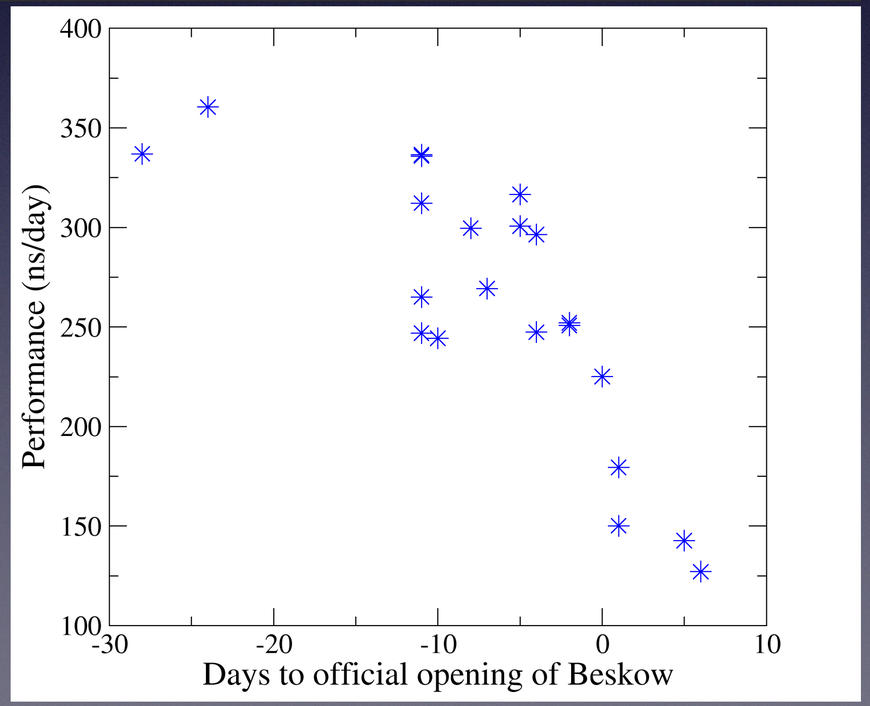


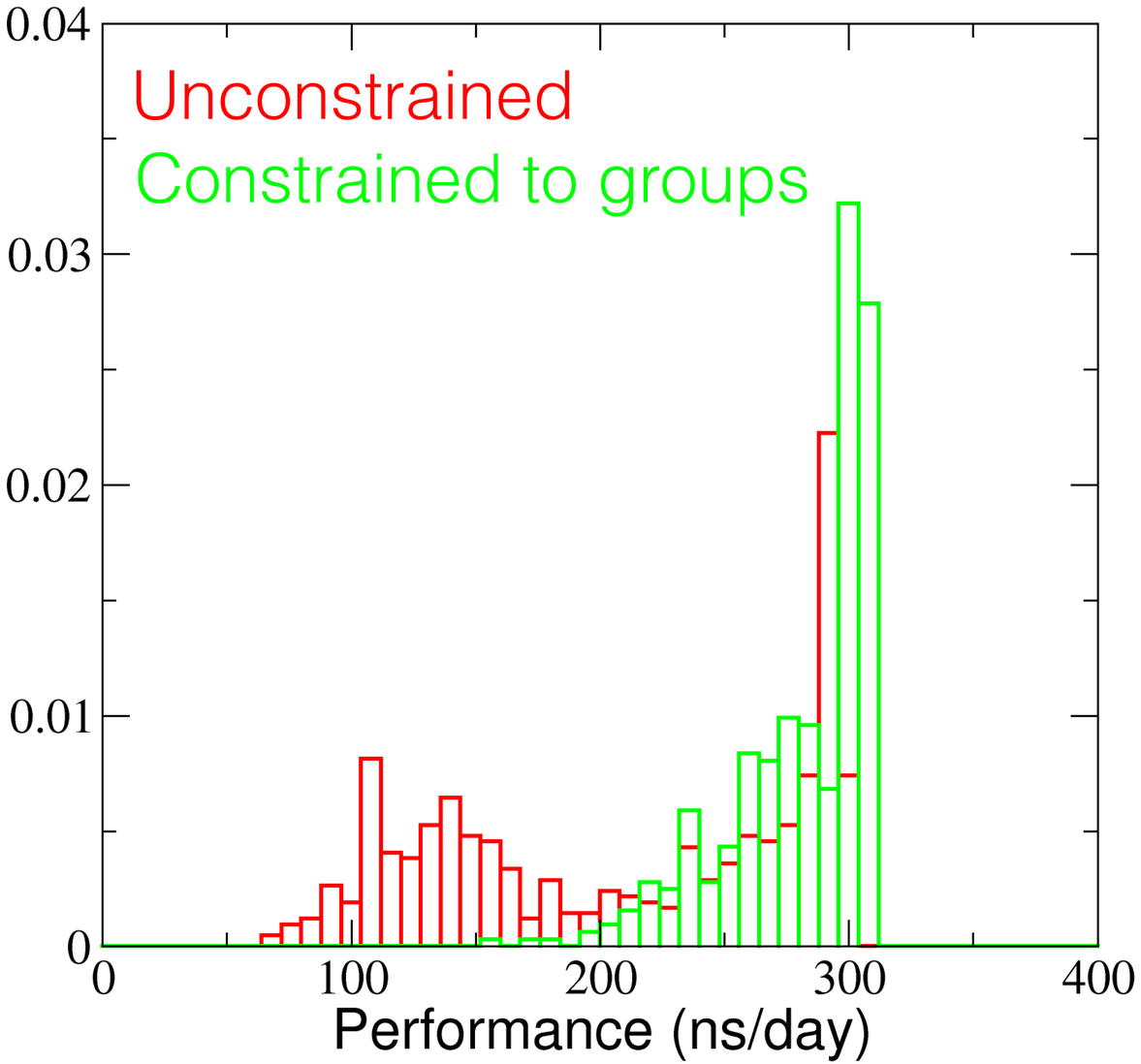
Compute Blade
4 Compute Nodes

Chassis
Rank 1 Network
16 Compute Blades
No Cables
64 Compute Nodes

Group
Rank 2 Network
Passive Electrical Network
2 Cabinets
6 Chassis
384 Compute Nodes

System
Rank 3 Network
Active Optical Network
Hundreds of Cabinets
Up to 10s of thousands of nodes





Intra-rank parallelisation: OpenMP today, future ?

Efficient current parallelization of all algorithms using MPI + OpenMP

OpenMP is (performance) portable, but limited:

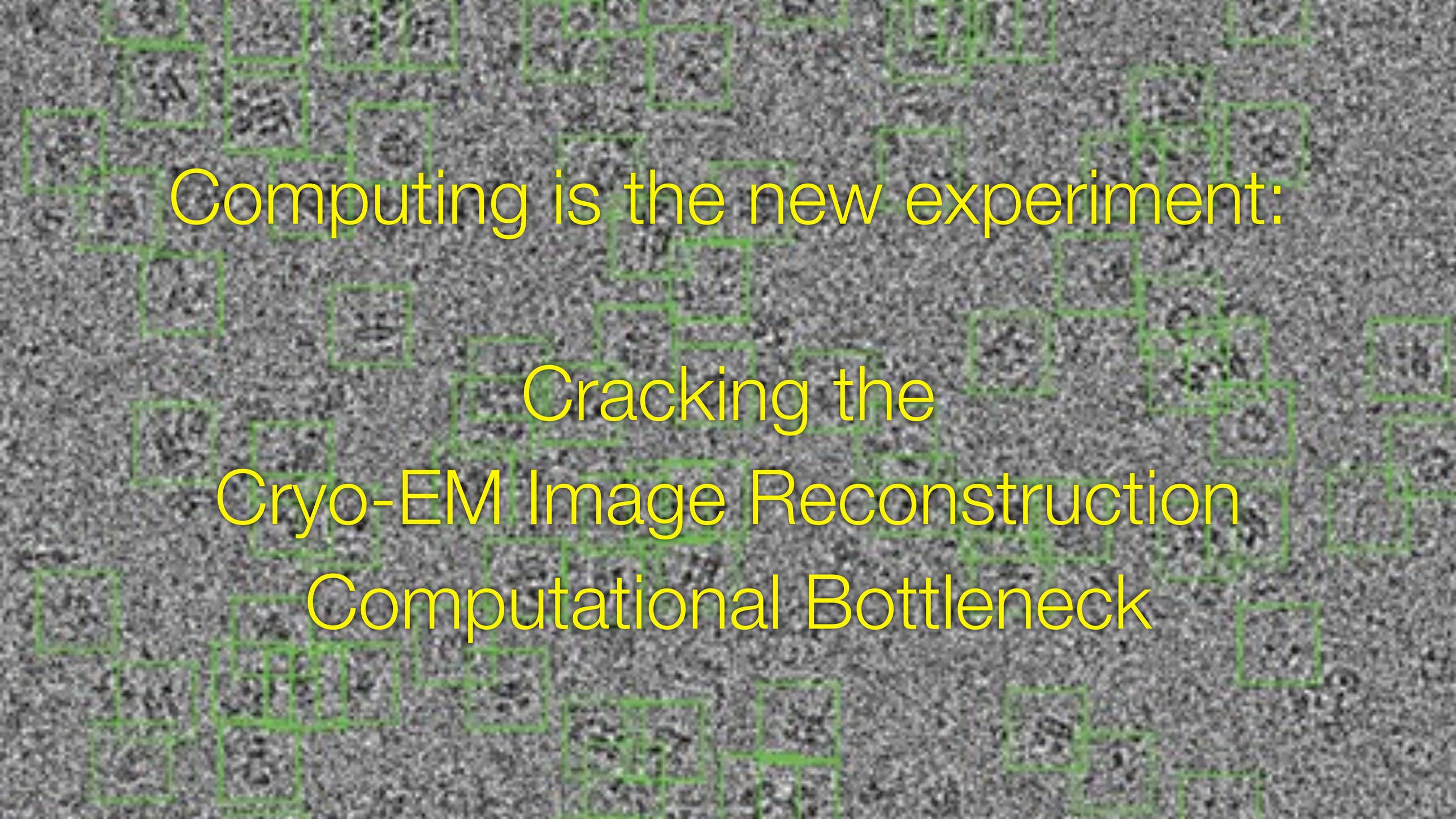
- No way to run parallel tasks next to each other
- No binding of threads to cores (cache locality)

Need for a better threading model, requirements:

- Extremely low overhead barriers (all-all, all-1, 1-all)
- Binding of threads to cores
- Portable

We are convinced we are moving to a world where latency- and throughput-optimized units converge into the same chip - the future is heterogeneous!

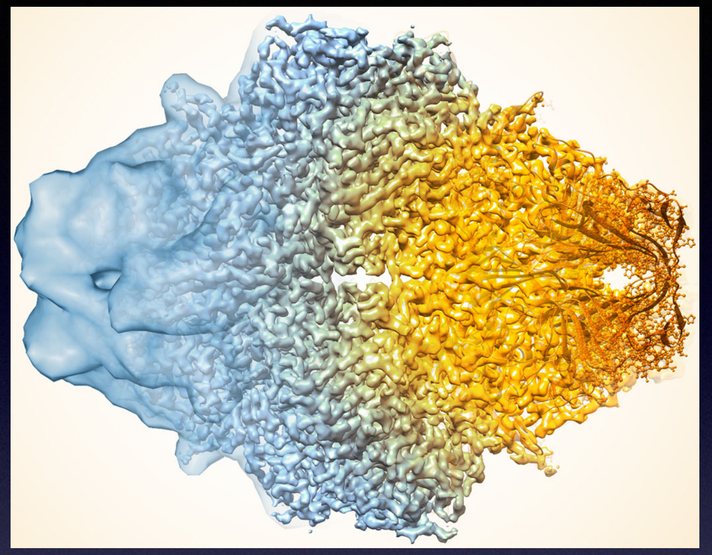
Urgent need for better, standardized and portable HPC-focused task parallelism frameworks. We are looking into both ArgoBots and home-grown solutions.

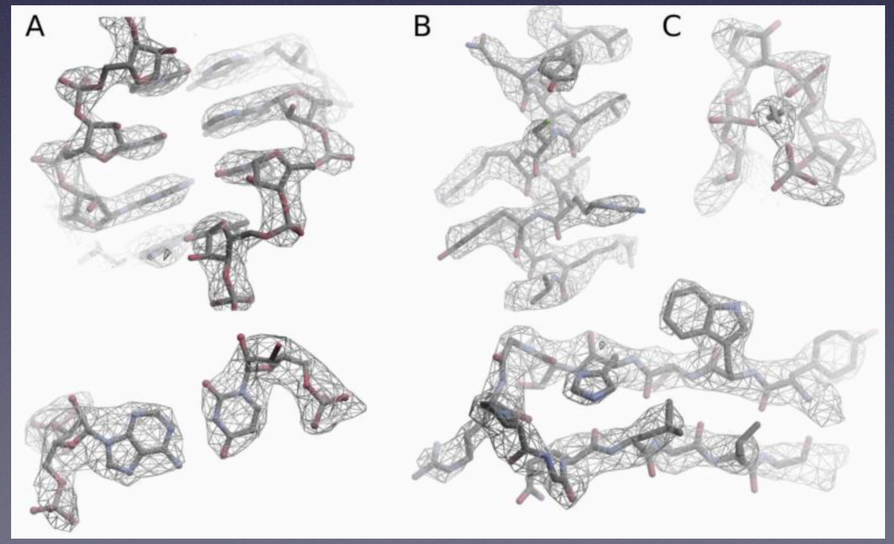

Computing is the new experiment:
Cracking the
Cryo-EM Image Reconstruction
Computational Bottleneck

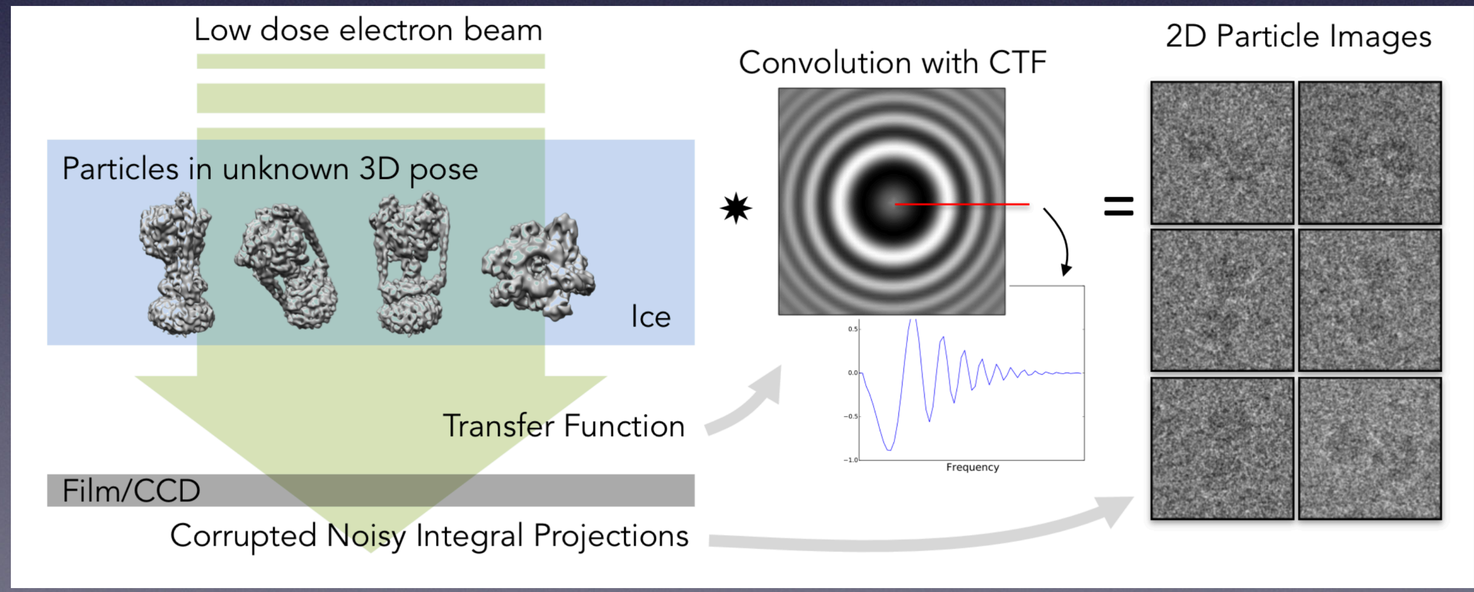
Cryo-EM Image Reconstruction

Structural biology: Determine the structure of biomolecules from electron microscopy.

Take thousands of images each containing 100-1000 particles and use computers to reconstruct 3D electron density







RELION

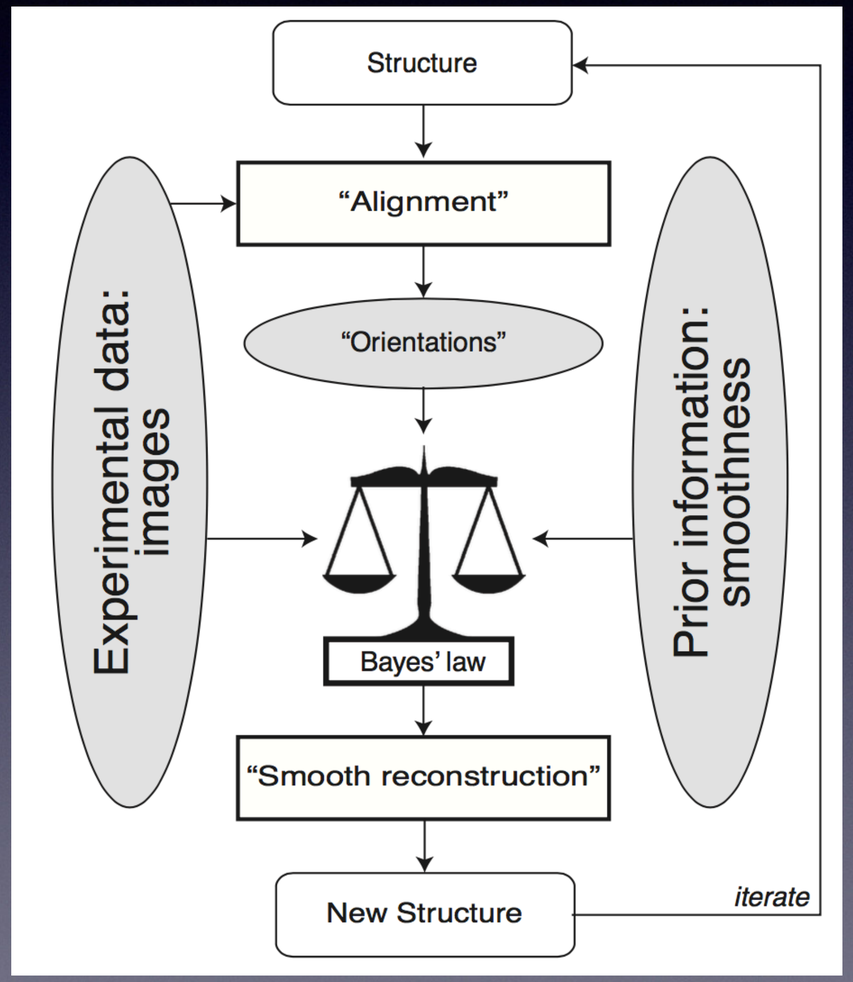
A Bayesian approach

$$p(A|B)p(B) = p(B|A)p(B)$$

$$p(A|B) = \frac{p(B|A)p(B)}{p(B)}$$

Model Image data

This way, we can include all microscope CTF, noise estimates and possible orientational bias in the statistics, as well as the assumption of smoothly varying density



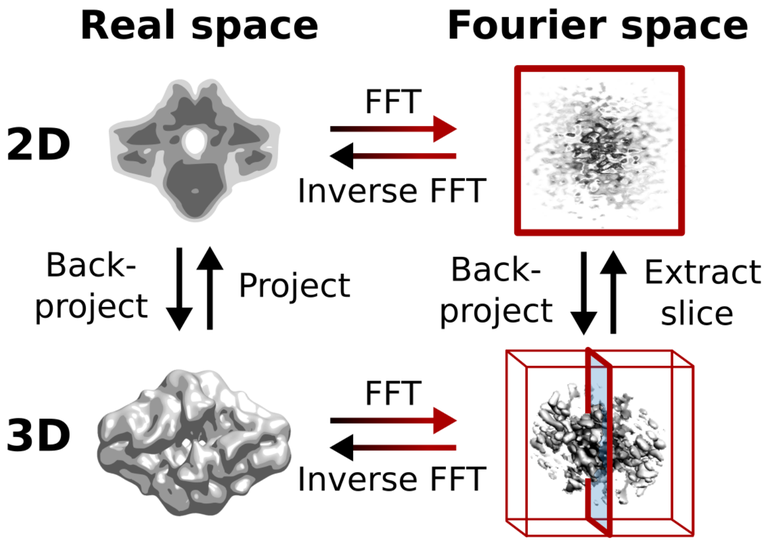
Particle picking

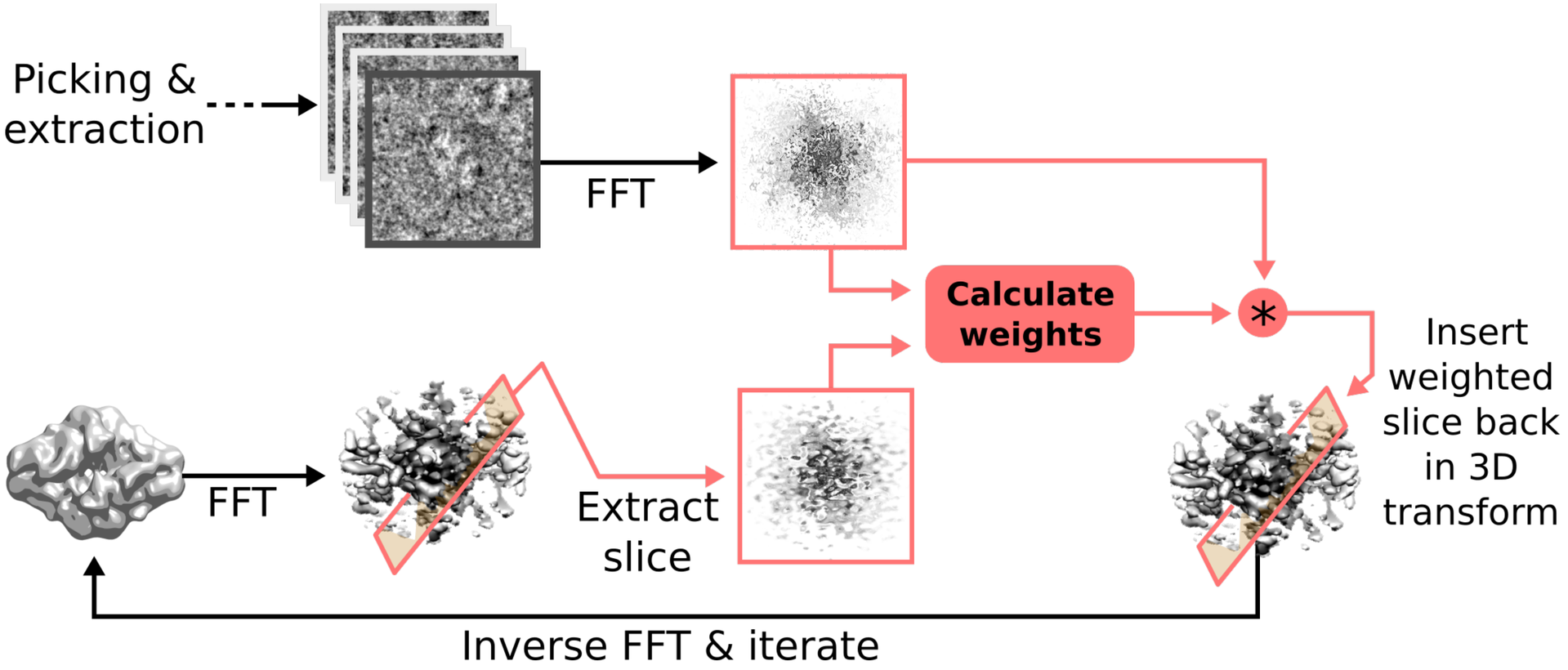
2D classification

3D classification

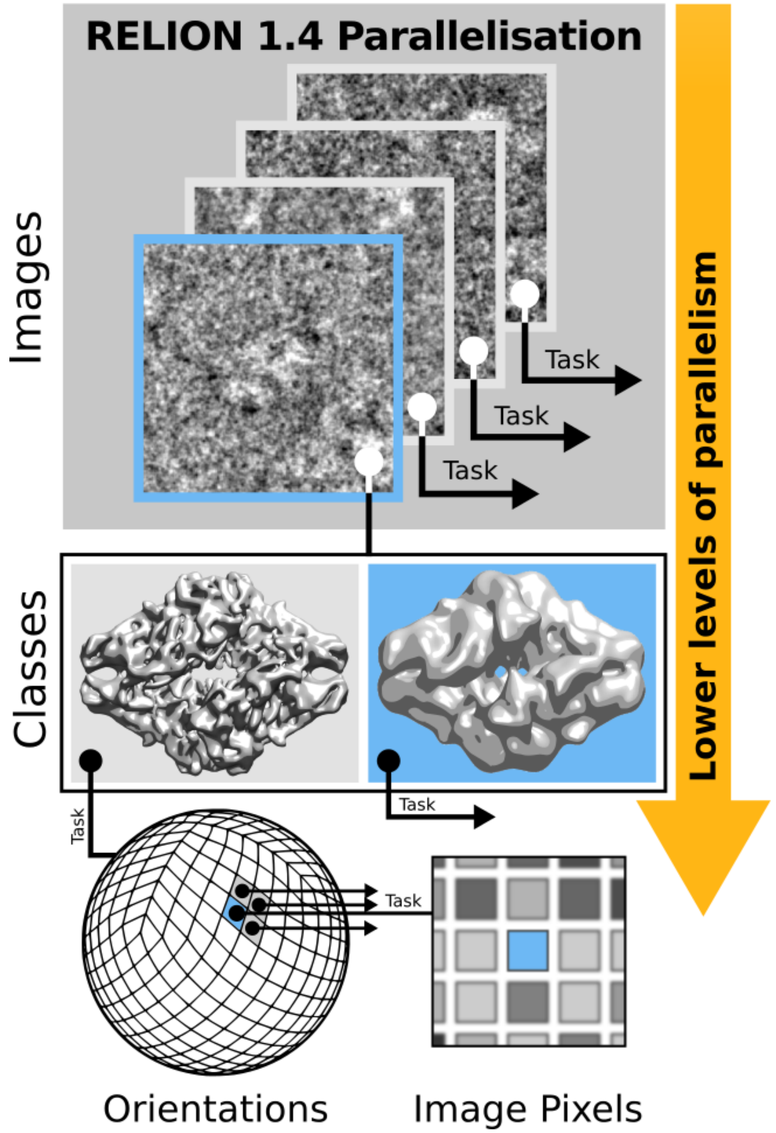
3D refinement ("autorefine")

Alignment & Classification



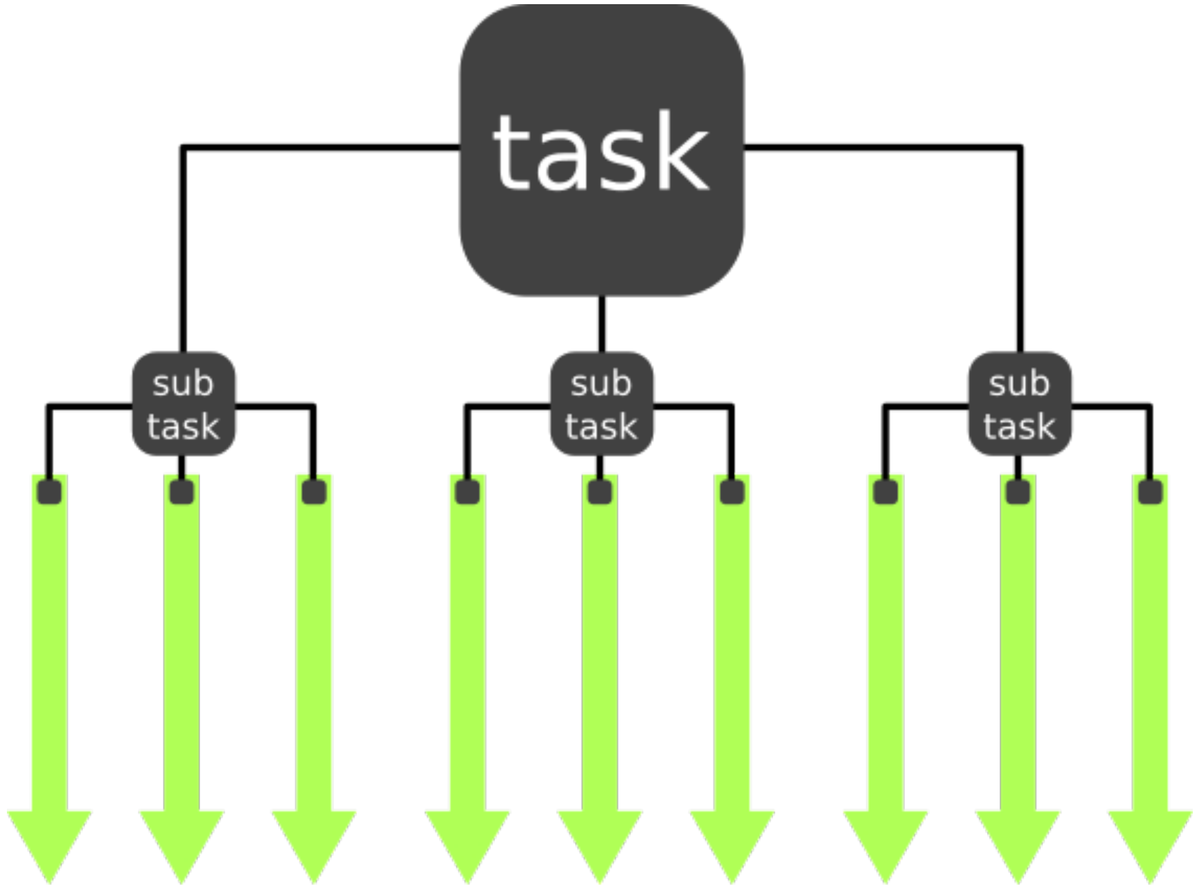


Parallelization

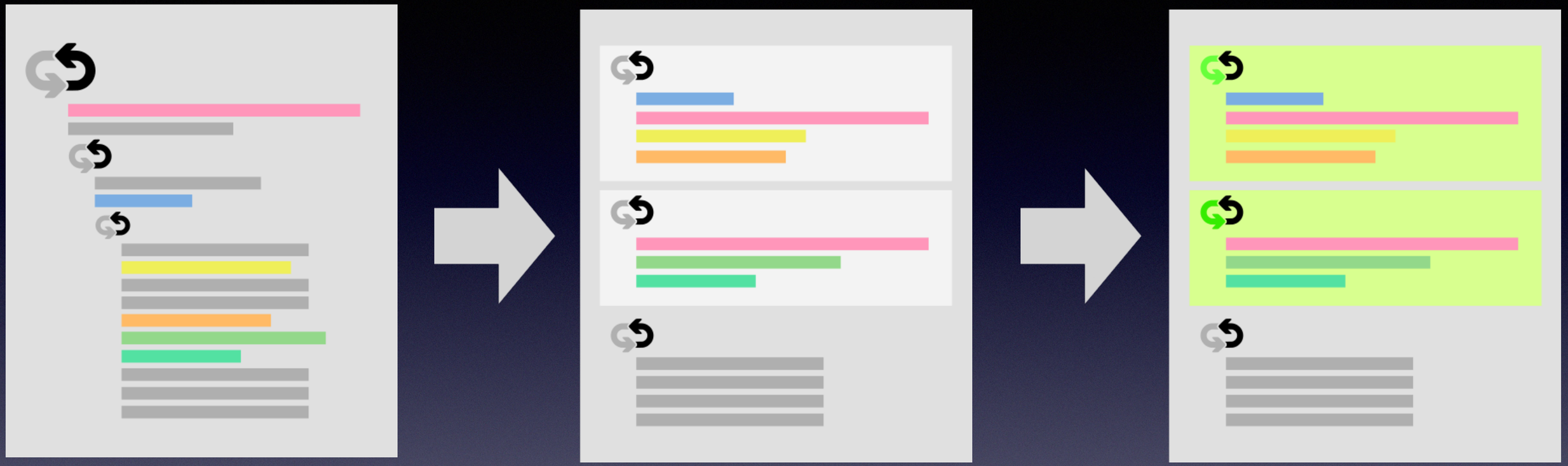


We scan many independent

- **Views (100,000s)**
- **Objects (10)**
- **Pixels (0.25 Mpix)**



Focus on algorithms, not micro-tuning



Plain implementation

Kernels

Tuned kernels

*It is critical that you understand and re-implement the original **physical model algorithm**, rather than just move around or tune code. Being able to stream through data is critical!*

GROMACS, year 2001

```
34535       movaps xmm0, xmm4
34536       movaps xmm1, xmm4
34537
34538       shufps xmm2, xmm6, 0b10001000
34539
34540       shufps xmm0, xmm5, 0b10001000
34541       shufps xmm1, xmm5, 0b11011101
34542
34543       /* move ix0-iz0 to xmm4-xmm6 */
34544       movaps xmm4, [esp + _ix0]
34545       movaps xmm5, [esp + _iy0]
34546       movaps xmm6, [esp + _iz0]
34547
34548       /* calc dr */
34549       subps xmm4, xmm0
34550       subps xmm5, xmm1
34551       subps xmm6, xmm2
34552
34553       /* store dr */
34554       movaps [esp + _dx0], xmm4
```

Good Kernels Express Parallelism, not Architecture

```
template <typename T>
__global__ void cuda_translate3D(   T * g_image_in,
                                    T * g_image_out,
                                    int image_size,
                                    int xdim,
                                    int ydim,
                                    int zdim,
                                    int dx,
                                    int dy,
                                    int dz)

{
   int tid = threadIdx.x;
   int bid =  blockIdx.x;

   int x,y,z,xp,yp,zp,xy;
   int voxel=tid + bid*BLOCK_SIZE;
   int new_voxel;

   int xydim = xdim*ydim;

   if(voxel<image_size)
   {
      z =  voxel / xydim;
      zp = z + dz;

      xy = voxel % xydim;
      y =  xy / xdim;
      yp = y + dy;

      x =  xy % xdim;
      xp = x + dx;

      if( zp>=0 && yp>=0 && xp>=0 && zp<zdim && yp<ydim && xp<xdim)
      {
         new_voxel = zp * xydim + yp * xdim + xp;
         if(new_voxel >= 0 && new_voxel < image_size)
            g_image_out[new_voxel] = g_image_in[voxel];
      }
   }
}
```

```
template <typename T>
void cpu_translate3D(T * g_image_in,
                     T * g_image_out,
                     int       image_size,
                     int       xdim,
                     int       ydim,
                     int       zdim,
                     int       dx,
                     int       dy,
                     int       dz)
{
   int x,y,z,xp,yp,zp,xy;
   int new_voxel;

   for(int voxel=0; voxel<image_size; voxel++)
   {
      int xydim = xdim*ydim;
      z =  voxel / xydim;
      zp = z + dz;

      xy = voxel % xydim;
      y =  xy / xdim;
      yp = y + dy;

      x =  xy % xdim;
      xp = x + dx;

      if( zp>=0 && yp>=0 && xp>=0 && zp<zdim && yp<ydim && xp<xdim)
      {
         new_voxel = zp*xydim + yp*xdim + xp;
         if(new_voxel >= 0 && new_voxel < image_size)
            g_image_out[new_voxel] = g_image_in[voxel];
      }
   }
}
```

“Ninja Coding”: GROMACS vs. RELION

	Lines of raw SIMD code
GROMACS	~600,000
RELION	0

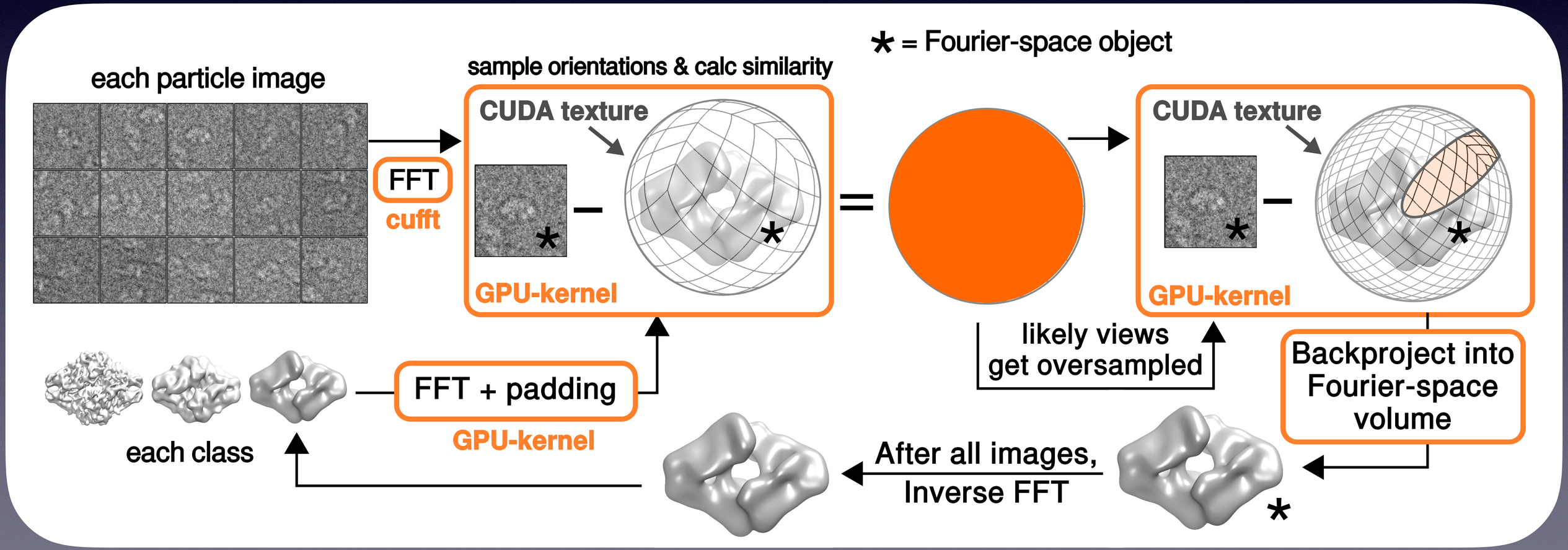
Used to be raw assembly, but now intrinsics

- It is extremely likely that we could get even better speed-ups in RELION with manual tweaking, but if the compiler performs this well with only hints, why bother?
- There is a huge advantage in only having a single code path. All modifications introduced by anybody will now be x86-accelerated, even if they do not understand SIMD

Use GPU-specific features when you can

When we have switched to single precision, we realized we can use GPU textures to handle interpolation during image rotation.

This is an example of something that cannot be done with OpenACC!

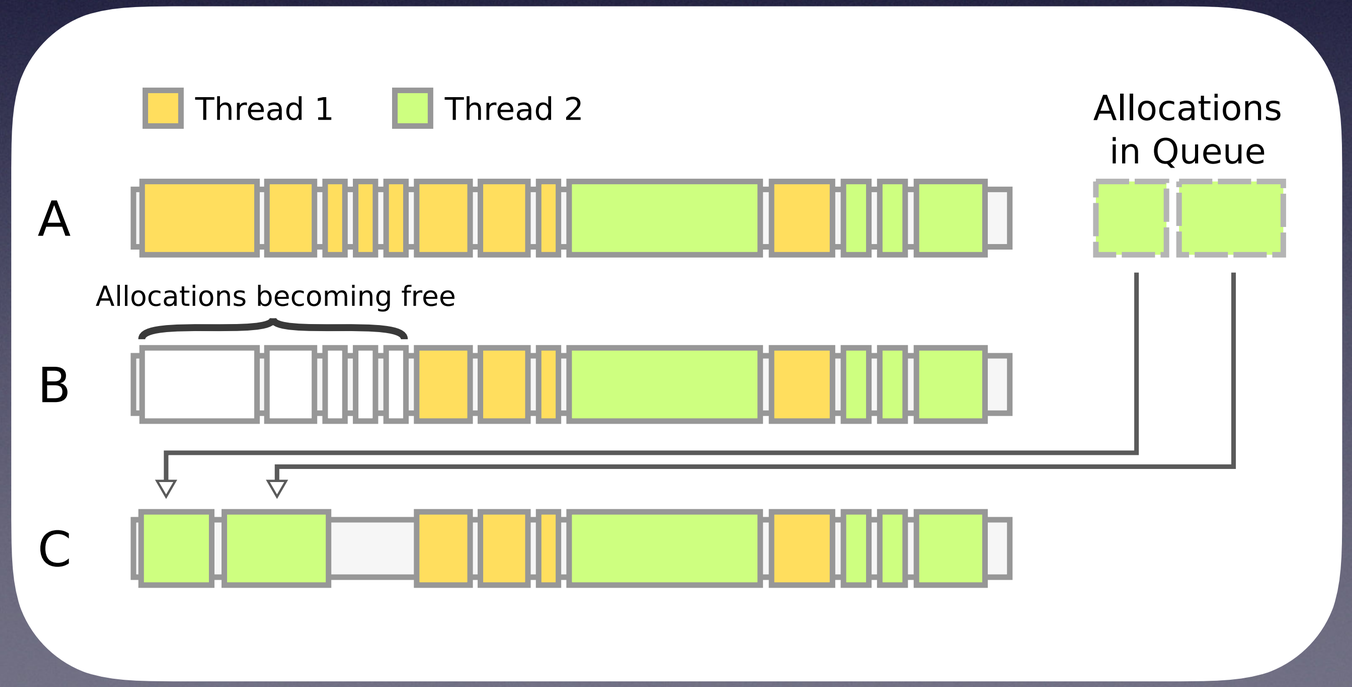


Handling limited GPU memory

If we try to put all data on the GPU, we run out of memory (fast)

When handling things dynamically, cudaMalloc() became bottleneck

We (Dari Kimanius) had to write a custom memory allocator for CUDA



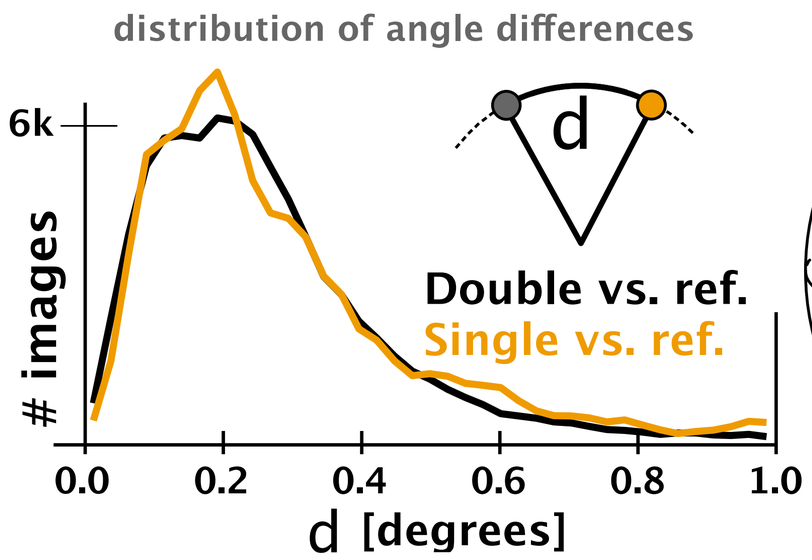
Allocate a large chunk of memory and handle it ourselves

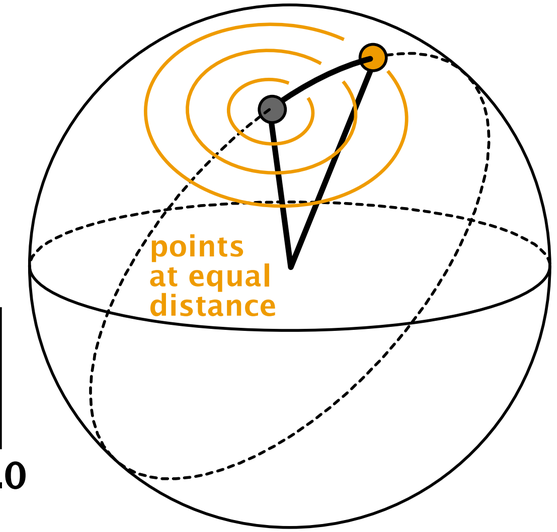
Unified memory could theoretically be an alternative, but presently it is just as slow as CUDA handling memory

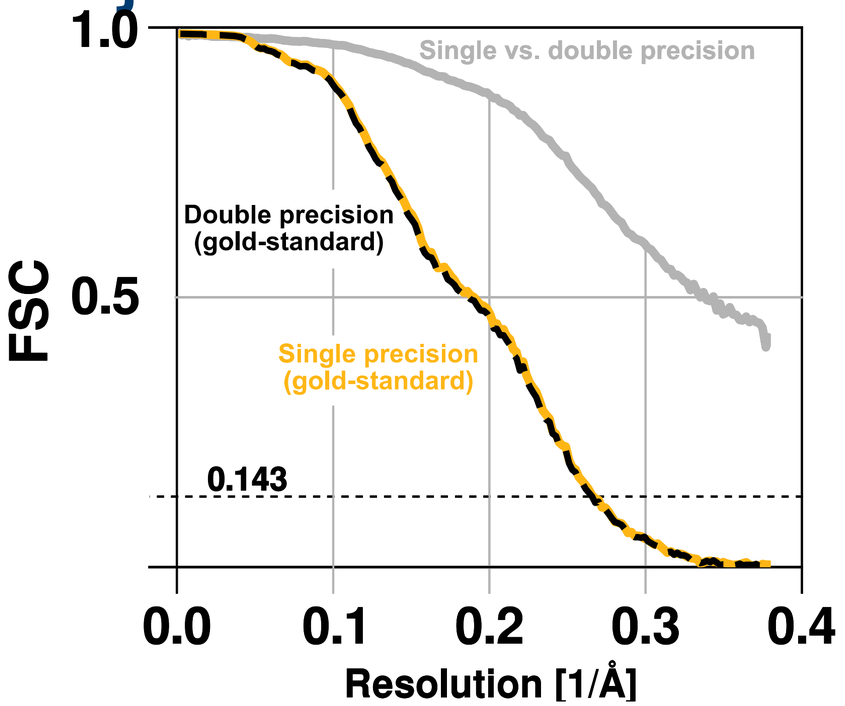
Single precision

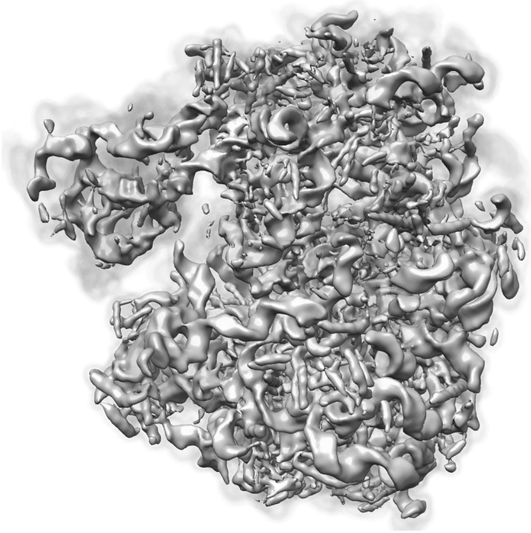
- Most modern CPUs provide *twice* the throughput in single compared to double
  - This also saves cache, memory bandwidth – and RELION is quite memory-hungry
- Do you need more than 7 valid digits in the output?
- Converting scientific code to single-precision is not trivial: Most codes fail if you just do search & replace, but it can be made to work:
  - Identify sensitive parts (maximization step), and leave those in double
  - Only perform a few operations (exponentials) in double
  - Sum small numbers first, or use tree summation instead of linear order
  - Use strength-reduction algorithms (check open source math libraries)
- Beware of double-single-double conversions inside critical code paths

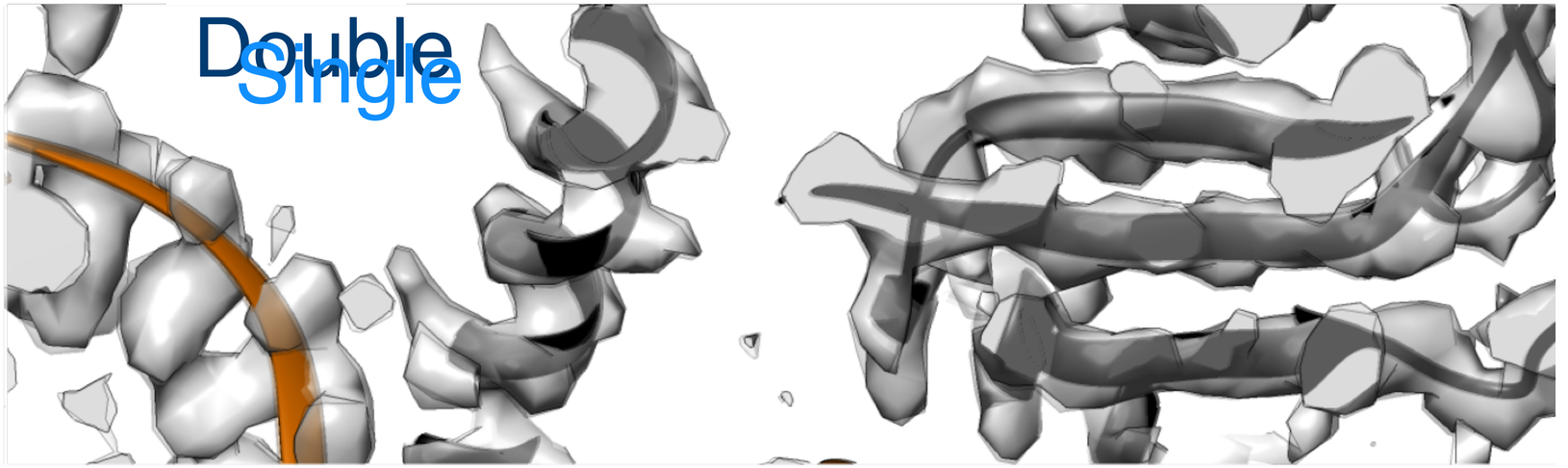
Quality: Single precision RELION is just as accurate









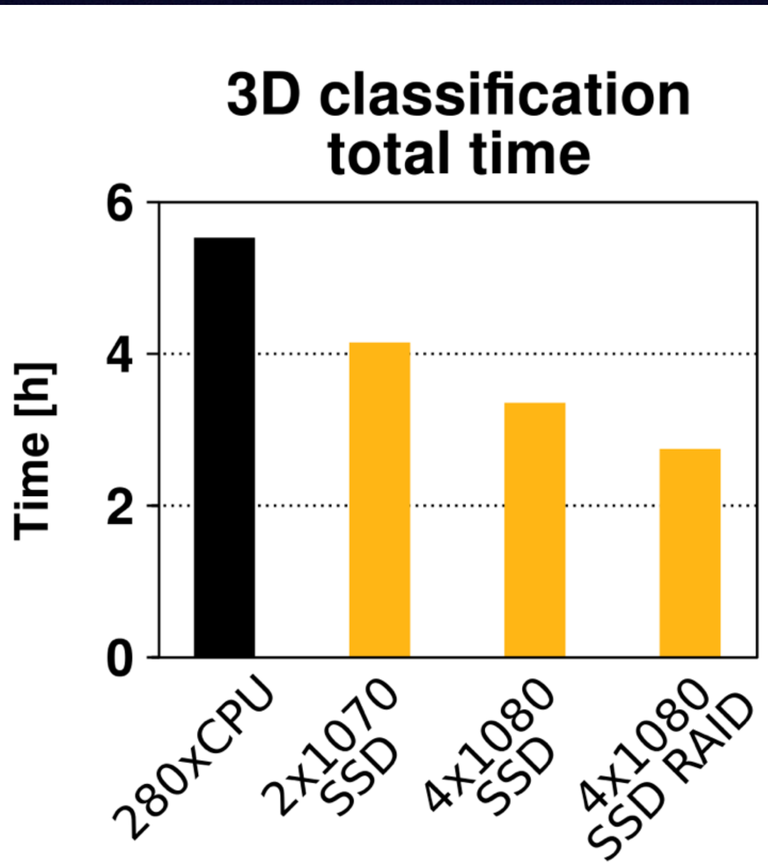


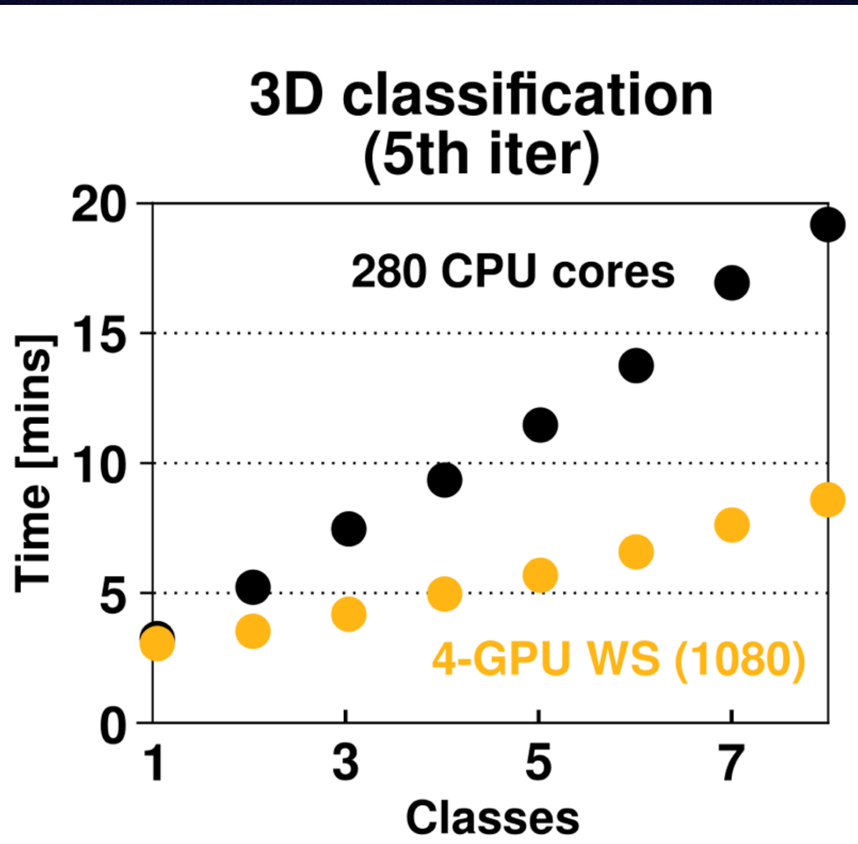
So, how did we do performance-wise?

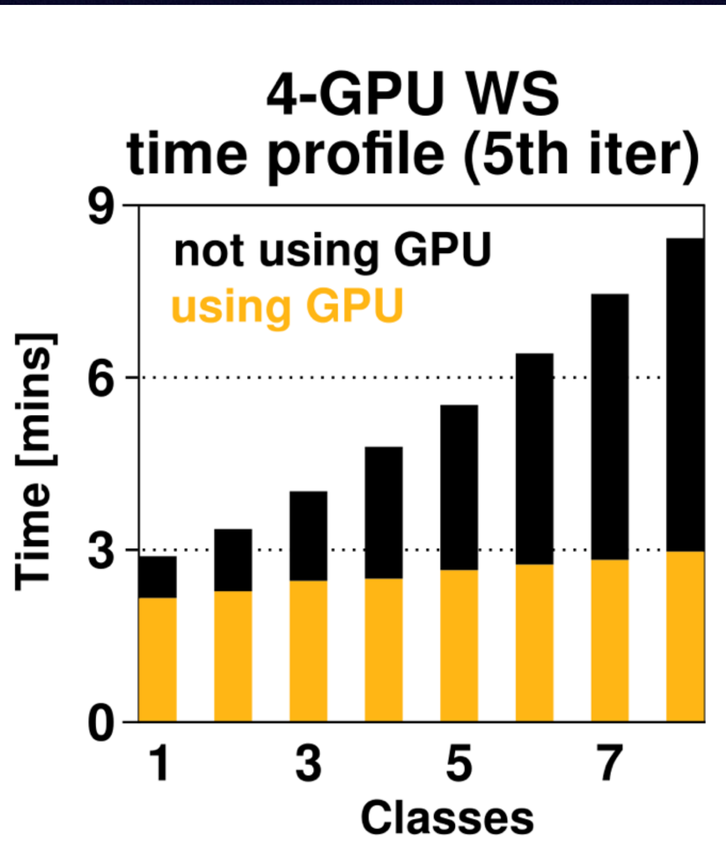
A single quad-GPU workstation was on par with 10 nodes with dual 14-core Xeon E5-2960v4 CPUs (est. cost $85,000)

… and we were still often limited by our CPU code (a bit embarrassing…)

RELION-3: New x86 acceleration makes CPU competitive again!

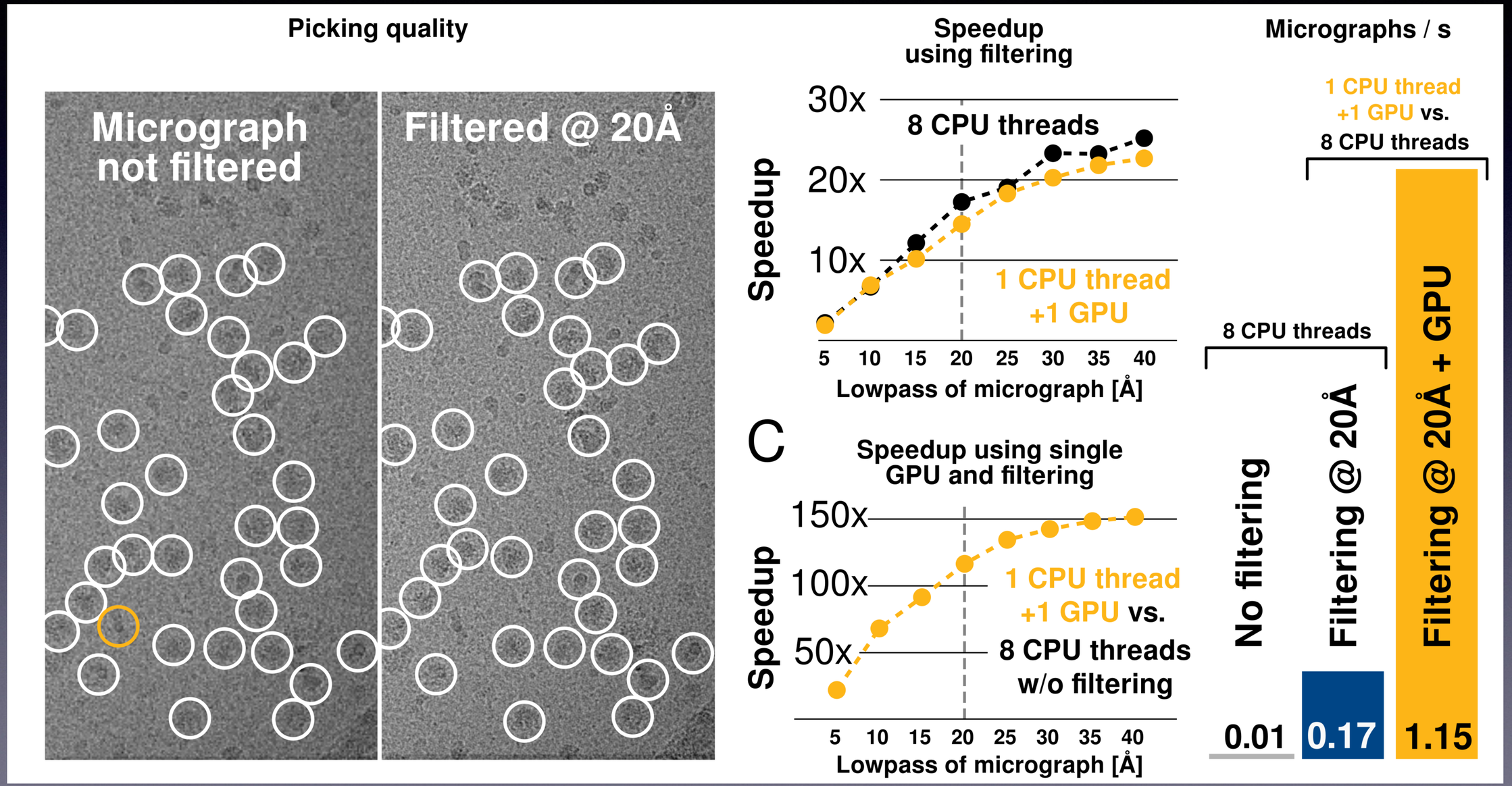


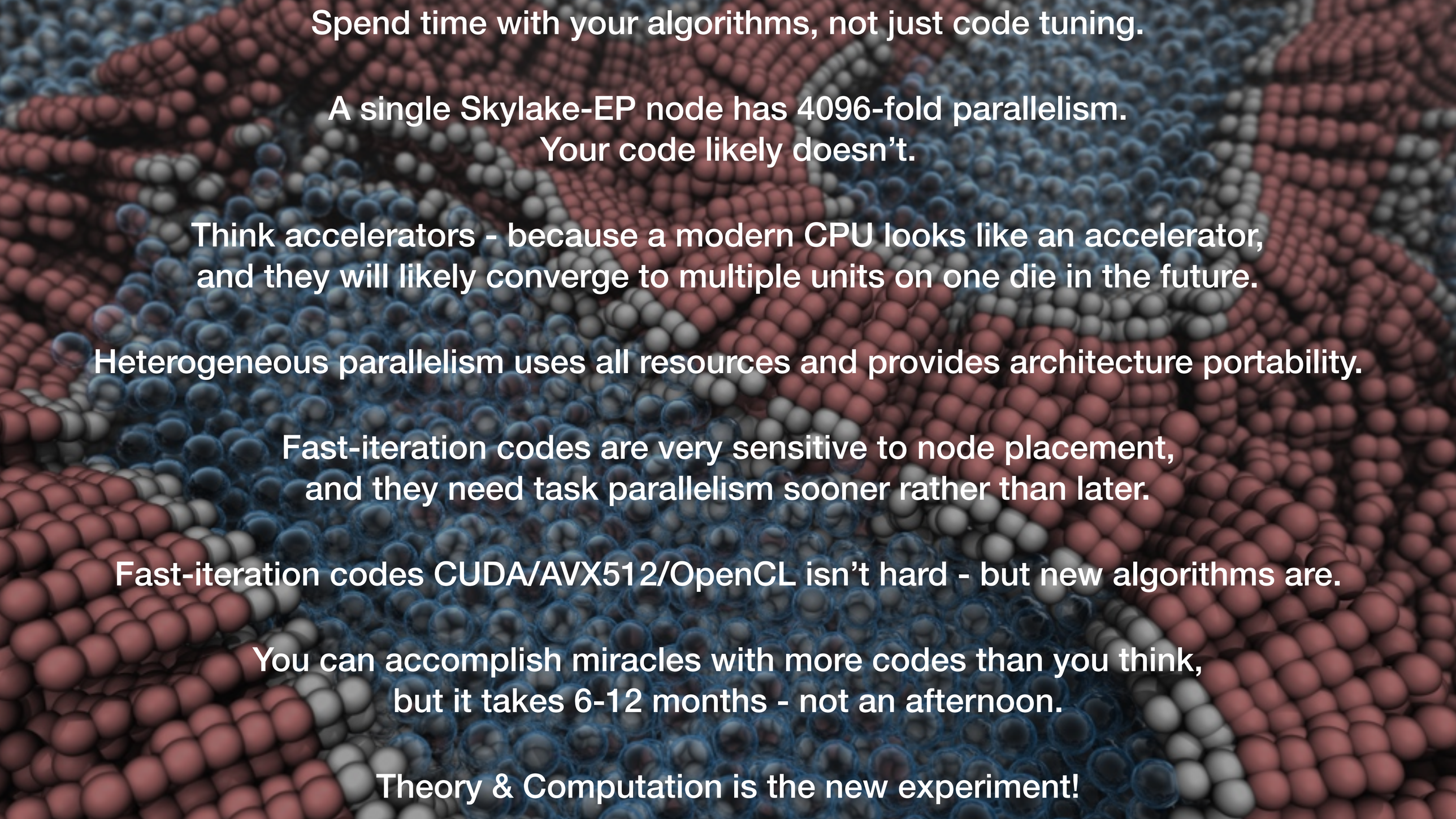




Preprocessing steps are even more impressive - 600x

To find particles in the images, we rely on FFTs where we can use cuFFT




Spend time with your algorithms, not just code tuning.
A single Skylake-EP node has 4096-fold parallelism.
Your code likely doesn’t.
Think accelerators - because a modern CPU looks like an accelerator,
and they will likely converge to multiple units on one die in the future.
Heterogeneous parallelism uses all resources and provides architecture portability.
Fast-iteration codes are very sensitive to node placement,
and they need task parallelism sooner rather than later.
Fast-iteration codes CUDA/AVX512/OpenCL isn’t hard - but new algorithms are.
You can accomplish miracles with more codes than you think,
but it takes 6-12 months - not an afternoon.
Theory & Computation is the new experiment!

Acknowledgments

GROMACS: Berk Hess, Szilard Pall, Mark Abraham, Aleksei Iiupinov,
John Eblen, Roland Shultz, Christian Wennberg, Viveca Lindahl
RELION: Dari Kimanius, Björn Forsberg, Sjors Scheres,
Alexey Amunts, Marta Carroni, Shintaro Aibara
NVIDIA: Mark Berger, Duncan Poole, Julia Levites, Jiri Kraus, Nikolay Markovskiy
INTEL: Charles Congdon, Sheng Fu, Kristina Kermanshahche, Yuping Zhao
CSCS: Thomas Schulthess, Victor Holanda, Prashant Kanduri PDC: Erwin Laure