Parallelization schemes & GPU Acceleration

Erik Lindahl / Szilárd Páll

GROMACS USA workshop
September 13, 2013
Outline

- MD algorithmic overview
- Motivation: why parallelize/accelerate MD?
- Acceleration on CPUs
- Parallelization schemes
- Heterogeneous/GPU acceleration
Important technical terms

• Hardware:
  – CPU core: physical, logical, "Bulldozer" module
  – Node: compute machine (!= mdrun “node”)
  – Accelerators: GPUs, Intel MIC

• Algorithmic/parallelization:
  – Accelerated code, compute kernels: CPU SIMD: SSE, AVX; GPU SIMT: CUDA
  – multi-threading: OpenMP (thread-MPI)
  – SPMD, MPMD: MPI
Molecular dynamics: algorithm overview

MD step

- NS, DD
- Bonded F
- Non-bonded F
- PME
- Integration
- Constraints

Neighbor search/DD step: every 10-50 iterations

~ milliseconds
Why parallelize/accelerate?

- **Need for speed:**
  - MD is computationally demanding
  - but a fast\(^*\) simulation needs:
    - **short time-step**
    - **strong scaling**

**Goal:** making it as short as possible currently at peak: ~100s of microsec.
Motivation: hardware evolution

- Hardware is increasingly:
  - **parallel** on multiple levels
  - **heterogenous**: accelerators

SIMD & SIMT memory & cache

multicore, NUMA accelerator, PCI-E, topology

network: topology, bandwidth, latency

GROMACS USA workshop
Motivation: need to keep up!

- Multiple levels of hardware parallelism
  - need to address each level with suitable parallelization

- Memory bottleneck
  - hierarchical memory, caches

- Zoo of accelerator architectures:
  - NVIDIA GPUs, project Denver (ARM + GPU)
  - Intel MIC
  - AMD GPUs & Fusion
  - Adapteva Parallela, Intel GPUs, etc.
Acceleration on CPUs

• Goal: maximize single-threaded performance
• Exploit CPU architecture-specific features
  – utilize the SIMD instruction set
  – SIMD: data-parallelization within a core

• Raw assembly up to GROMACS v4.5
• Switch to compiler intrinsics in 4.6
  → compiler matters more:
    • use a recent compiler (reason: performance & correctness)
    • always run regression tests (make check)
Acceleration on CPUs: supported platforms

- x86 (Intel, AMD):
  - SSE2, SS4.1, 128/256-bit AVX (Intel), 128-bit AVX + FMA4 (AMD Bulldozer/Piledriver)
  - Soon: AVX2 (Intel)

- In development:
  - Fuji Sparc (K computer)
  - Blue Gene P/Q

- Note: always use the “highest” possible acceleration supported by the platform
Acceleration on CPUs: practical info

- Using the right acceleration: build system picks most suitable setting for the build machine
  - need cross-compile e.g. if cluster compute & head node differ
  - to check/change: GMX_CPU_ACCELERATION CMake variable

- Checking what acceleration you're using:
  - mdrun -version header:
    CPU acceleration:   SSE2
  - run-time check whether binary matches hardware, see the log:
    Detecting CPU-specific acceleration.
    Vendor: AuthenticAMD
    Brand:  AMD Phenom(tm) II X6 1090T Processor
    [...]
    Acceleration most likely to fit this hardware: SSE2
    Acceleration selected at GROMACS compile time: SSE2
    Note: crash with “illegal instruction” error will occur when the instruction set is not supported (e.g. AVX binary on SSE4.1 hardware).
Multi-node parallel MD

- Particle decomposition
  - distribute particles across compute units
  - not efficient: scales bad
  - will be (partially?) deprecated in v5.0

- Domain decomposition
  - volume-based
  - more efficient: better scaling
  - load balancing
Parallelization schemes

• MPI message passing
  – parallelization across multiple processors
  – SPMD, MPMD: single/multiple program multiple data

• Multi-threading
  – OpenMP: “true” SMP (no DD needed)
  – thread-MPI: domain decomposition-based
    • efficient multi-threaded MPI implementation
    • advantage: single-node runs don't need MPI!

• MPI + OpenMP multi-threading
  – multiple threads within a rank
    → multiple cores per rank/domain
OpenMP multi-threading

- Supported in `mdrun` (and a few tools):
  - verlet scheme: fully
  - group scheme: in PME
- No need for DD ⇒ more efficient than thread-MPI on:
  - up to 4-6 AMD cores
  - up to 12-16 Intel cores
- Default with verlet scheme with moderate thread count
SPMD: MPI hardware mapping

- single rank per core
- \#domains = \#cores
- parallelization limitation posed by \texttt{minimum cell size}
SPMD: MPI hardware mapping & communication

- single rank per core
- #domains = #cores
- parallelization limitation posed by minimum cell size
- every rank communicates → communication bottleneck (especially in PME!)
MPMD: separate PME ranks

- Problem: 4x MPI_Alltoall/step in PME
- Solution: coarse task-parallelization
  
  → dedicate some of the CPUs for PME

- **Advantage:** less communication

- **Disadvantage:**
  
  - PME ranks idle during integration, DD, NS
  
  - often need to tweak the amount of resources for PME, i.e. the number of PME ranks/threads (`g_tune_pme`)
MPMD:
PME ranks – reduction in communication

- Switch to separate PME ranks automated
- Number of ranks estimated
- Typical PP/PME ratio 3:1
  - 4x less ranks do PME
  - 4*4=16x less communication calls
Hybrid parallelization: MPI + OpenMP

- Multi-threading within each MPI rank \(\rightarrow\) multiple cores/rank

- Advantages:
  - larger domains/rank
    \(\rightarrow\) can scale further
  - with GPUs

- Disadvantages:
  - combined MPI + OpenMP overhead
  - often faster only at the scaling limit
Multi-threaded runs: practical info

- **mdrun command line options:**
  - `-ntomp`: number of OpenMP threads; `OMP_NUM_THREADS` also respected, `GMX_PME_NUM_THREADS` for PME ranks
  - `-ntmpi`: total number MPI threads
  - `-nt`: total number of threads (thread-MPI x OpenMP)

- Numbers reported on the command line (and log):
  - Using 8 OpenMP threads

- Log: performance accounting table 3\textsuperscript{rd} column
MPI+OpenMP parallelization: practical info

• Typically slower than pure MPI or pure OpenMP

• Advantageous:
  – at high parallelization:
    • low atoms/core count
    • high node count
  – multi-sim - each simulation can use OpenMP only
  – GPU accelerated runs

• Numbers reported on the command line (and log):
  
  Using 2 MPI threads
  Using 4 OpenMP threads per tMPI thread

• See also in the log:
  – performance accounting table 2nd/3rd column
  – Time summary field (%): (Core t/Wall t)/(Nranks x Nthreads) = core efficiency
Thread pinning/affinities

- Thread migration causes performance issues

  → **thread pinning/affinity**: lock threads to a CPU core (more precisely hardware thread)

    -pin on/off enable/disable pinning; on by default only if all cores in a node are used (otherwise warning is issued)
    -pinoffset N lock threads to cores starting from core N+1
    -pinstripe N lock threads with a stride N (useful with SMT e.g. with HyperThreading N=1 to use HT, N=2 to not use)

- **Always use pinning with OpenMP!**
GPU acceleration: background

• First generation support in v4.5: OpenMM
  • Advantage: fast implicit solvent simulations
  • Disadvantages: single-GPU, limited feature support and compatibility with other features (“black-box”)
  • **Unmaintained in v4.6, planned deprecation in v5.0**

• Second generation:
  – native support in v4.6
  – new algorithms (Verlet scheme and NxN kernels)
GPU acceleration design principles

- **Future-proof algorithm**
  - hardware is evolving fast
  → avoid having to re-design for new hardware

- **Keep supporting for the rich GROMACS feature set**

- **Allow multi-node scaling**
  → treat GPU as an accelerator: offload compute-expensive calculation

- **Maximize both CPU and GPU utilization**
  - idle hardware not useful
  - challenge: GROMACS is fast (<1ms/step on CPU!)
GPU acceleration strategy

Pair search/domain-decomposition: every 10-50 iterations

MD step

Pair search, DD → Bonded F → Non-bonded F → PME → Integration → Constraints

offload

10-100s of microseconds?
GPU acceleration design principles

• Support existing **features**
  - **offload** most compute intensive task, keep the rest on the CPU

• Maximize **both** CPU and GPU utilization
  - idle hardware is not useful
  - GROMACS is fast, accelerating it is not easy

• **Future-proof** algorithm
  - we don’t want to rewrite the algorithms for new hardware
Particle-based non-bonded algorithm

- Traditional algorithm:
  - particle-based **neighbor list**
    (+ linked cell list)

→ Not efficient for SIMD acceleration
  - shuffle overhead (~50%)
  - low data reuse: memory/cache pressure
  - irregular data: ill-suited for GPUs
Cluster-based non-bonded algorithm

• Need an algorithm that lends itself to efficient SIMD parallelization:
  • maps naturally to SIMD width
  • accelerators e.g. SIMT
  • emphasizes data reuse

→ Algorithmic work unit:

**spatial cluster of N atoms**

• neighbor pair list: clusters of M vs N atoms
Cluster pair search

Standard cell grid:
spatially uniform

Clusters:
#atoms uniform
Cluster pair algorithm

- CPU:
  - 4-wide (SSE, 128-bit AVX): 4x4
  - 8-wide (256-bit AVX): 4x(4+4)

- GPUs need more parallelism
  - super-cluster: group of 8 i-clusters
  - high data reuse: >75 ops/byte

- NVIDIA GPUs, CUDA:
  - search: 8x8
  - kernel: (8x) 8x4 + 8x4 (2 independent warps)
GPU kernel work efficiency

- Source of 0-s calculated:
  - Verlet buffer
  - cluster setup
    - effective buffering: 0 buffer with PME and short cut-off
- Work efficiency (#zeros wrt Verlet): 0.5-0.7
CUDA kernel performance

- Performance depends on:
  - **system size** - tail effect: performance deteriorates
  - **GPU hardware**
    - generation: Fermi vs Kepler
    - \#multiprocessors
  - buffered cut-off: \textbf{rlist} (rcut and nstlist)

![Performance graph showing iteration time per 1000 atoms for different system sizes and GPUs: Tesla C2050, GeForce GTX 580, GeForce GTX 680, Tesla K20. PME, rc=1.0 nm, nstlist=20.]
Cluster scheme characteristics

- Efficient SIMD parallelization
  - emphasizes data reuse
  - various SIMD widths
- Automated buffering
  -> free to pick pair list update frequency
- Existing particle-based pair-list optimizations apply

- Disadvantages:
  - work-efficiency tradeoff
  -> not advantageous for expensive interactions
Heterogeneous parallelization: data & control flow

Avg. CPU/GPU overlap: 60-80% per step

Pair search: every 10-50 iterations

MD step
Heterogeneous parallelization: data & control flow – MPI/multi-node case

Pair search/DD step every 10-50 iterations

MD step

MPI receive non-local x
MPI send non-local F

CPU
OpenMP threads

DD
Local pair search
Non-local pair search

Bonded F
PME
Wait for non-local F
Wait for local F
Integration Constraints

MD step

CPU
CUDA

Local stream
Non-local stream

Local non-bonded F pair-list pruning

Non-local non-bonded F pair-list pruning

Clear F

Stream priorities in CUDA 5.5:
→ allow local kernel preemption
→ start non-local kernel when coordinates arrive through MPI
Heterogenous parallelization: hardware mapping

• Each domain maps to an MPI rank
  – currently one GPU/rank
  – ranks can share GPUs (with MPI)

• Partition CPU cores:
  – #threads = #cores / #GPUs
  – OpenMP scaling limitations:
    typically max. 4-8 threads/rank
    → need to “oversubscribe” GPUs (-
gpu_id 0011)
GPU acceleration: requirements & features

• Extra requirements:
  – CUDA v3.2+ (strongly recommended: 5.0+)
  – NVIDIA GPU with compute capability 2.0+ (Fermi or later)
  – OpenMP (MPI + GPU sharing often much slower)

• Features:
  – everything supported by the Verlet scheme but energy groups (for details check the GROMACS wiki)
Heterogeneous/GPU acceleration: practical info

• mdrun command line options:

   -nb gpu/cpu/gpu_cpu: non-bonded calc. on GPU or CPU (or both)
   (gpu_cpu: local forces on GPU non-local on CPU)
   -gpu_id XYZ: manual GPU selection; the order gives mapping to PP ranks (per node);
   GMX_GPU_ID env. var. useful e.g. with inhomogeneous nodes;
   -nt: total number of threads (thread-MPI x OpenMP)

• GPUs used by default, reported on the command line:

  2 GPUs detected on host tcbs28:
  
  #0: NVIDIA GeForce GTX TITAN, compute cap.: 3.5, ECC: no, stat: compatible
  #1: NVIDIA GeForce GTX TITAN, compute cap.: 3.5, ECC: no, stat: compatible

  2 GPUs user-selected for this run: #0, #1
Heterogenous/GPU acceleration: practical info (cont.)

- OpenMP scaling limits (with DD):
  - Intel: max 6-8 threads/rank
  - AMD: max 2-4 threads/rank

Given a 2x 16-core AMD node + 2 GPUs try:

```bash
mpirun -np 4 mdrun -gpu_id 0011
mpirun -np 8 mdrun -gpu_id 00001111
```

- Multiple independent simulations on the same GPU:
  - with multi-sim:
    ```bash
    mpirun -np 4 mdrun -multi 4 -gpu_id 0011
    ```
  - manually (assume 8 cores):
    ```bash
    mdrun -ntomp 4 -gpu_id 0 -pin on &
    mdrun -ntomp 4 -gpu_id 0 -pin on -pinoffset 4
    ```
Load imbalance

• Uneven load between the different computational units:
  CPU threads, CPU-GPU, MPI ranks
  → task completion at different times
  → global synchronization before integration: late tasks delay all CPUs
  \[\Rightarrow \text{idling = waste of resources}\]

• Caused by **inhomogeneity in**:
  - computational cost:
    • inhomogeneous particle distribution
    • inhomogeneous interaction cost distribution: water/non-water (group scheme), bonded interactions
  - hardware:
    • different CPUs/GPUs, thermal throttling, network delays/congestion, other processes interfering
  - (statistical fluctuation with small number of particles)

• Solution: **load balancing**
Automated multi-level load balancing

- **Domain decomposition:**
  - dynamic load-balancing
  - equalize load among ranks by resizing domains
  - consequences:
    - DD limits parallelization: minimum cell size
    - multi-threading limitations
    - bonded imbalance: amplified by GPUs

- **PP-PME load balancing**
  - scale the coulomb cut-off
    → shift workload from PME to PP
  - with GPUs and/or separate PME ranks
  - disadvantage: PP vs PME scaling

DD load balancing effect:
group scheme water vs non-water kernel
causing imbalance
(cell started out as equal in size!)
CPU-GPU load balancing

- Increasing cut-off & grid spacing
  - shifts load from CPU to GPU when the GPU is “too fast” (i.e. complete computing before the CPU)
    - maximizes CPU-GPU overlap
    - improves GPU utilization & minimize CPU idling

- Fully automated:
  - a few hundred load balancing steps in the beginning
  - use provided cut-off as minimum
Load balancing: practical info

- **Domain decomposition dynamic load-balancing**
  
  mdrun -dlb auto/no/yes
  - automated: turns on when load imbalance is detected
  - dynamic: keeps balancing continuously

- **PP-PME load balancing**
  
  mdrun -[no]tunepme
  - automated: always checks when PP-PME tasks are separate
  - static: uses the selected cut-off/grid throughout the run
    - can go slightly wrong: external factors, fluctuation, interference with DD load balancing
    
    → **check for consistency**
CPU-GPU load imbalance: low PP load

<table>
<thead>
<tr>
<th>Computing:</th>
<th>Nodes</th>
<th>Th.</th>
<th>Count</th>
<th>Wall t (s)</th>
<th>G-Cycles</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbor search</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>1.647</td>
<td>13.180</td>
<td>5.2</td>
</tr>
<tr>
<td>Launch GPU ops.</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>0.086</td>
<td>0.689</td>
<td>0.3</td>
</tr>
<tr>
<td>Force</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>6.357</td>
<td>50.863</td>
<td>19.9</td>
</tr>
<tr>
<td>PME mesh</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>18.204</td>
<td>145.640</td>
<td>56.9</td>
</tr>
<tr>
<td>Wait GPU local</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>0.024</td>
<td>0.193</td>
<td>0.1</td>
</tr>
<tr>
<td>NB X/F buffer ops.</td>
<td>1</td>
<td>3</td>
<td>2002</td>
<td>0.647</td>
<td>5.174</td>
<td>2.0</td>
</tr>
<tr>
<td>Write traj.</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.323</td>
<td>2.588</td>
<td>1.0</td>
</tr>
<tr>
<td>Update</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>0.619</td>
<td>4.950</td>
<td>1.9</td>
</tr>
<tr>
<td>Constraints</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>3.753</td>
<td>30.025</td>
<td>11.7</td>
</tr>
<tr>
<td>Rest</td>
<td>1</td>
<td></td>
<td></td>
<td>0.311</td>
<td>2.485</td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td>31.971</td>
<td>255.786</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Force evaluation time GPU/CPU: 14.358 ms/24.537 ms = 0.585 (ns/day) (hour/ns)
Performance: 5.410 13.180

GPU "too fast" (wrt CPU) → low CPU-GPU overlap → GPU idles

Note: load balancing is manually turned off above (notunepme).
CPU-GPU load imbalance: too much PP load

<table>
<thead>
<tr>
<th>Computing:</th>
<th>Nodes</th>
<th>Th.</th>
<th>Count</th>
<th>Wall t (s)</th>
<th>G-Cycles</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbor search</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>2.062</td>
<td>16.493</td>
<td>5.6</td>
</tr>
<tr>
<td>Launch GPU ops.</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>0.082</td>
<td>0.655</td>
<td>0.2</td>
</tr>
<tr>
<td>Force</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>6.209</td>
<td>49.673</td>
<td>16.8</td>
</tr>
<tr>
<td>PME mesh</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>10.762</td>
<td>86.104</td>
<td>29.1</td>
</tr>
<tr>
<td><strong>Wait GPU local</strong></td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>12.185</td>
<td>97.483</td>
<td>33.0</td>
</tr>
<tr>
<td>NB X/F buffer ops.</td>
<td>1</td>
<td>3</td>
<td>2002</td>
<td>0.661</td>
<td>5.289</td>
<td>1.8</td>
</tr>
<tr>
<td>Write traj.</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.321</td>
<td>2.565</td>
<td>0.9</td>
</tr>
<tr>
<td>Update</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>0.606</td>
<td>4.846</td>
<td>1.6</td>
</tr>
<tr>
<td>Constraints</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>3.702</td>
<td>29.616</td>
<td>10.0</td>
</tr>
<tr>
<td>Rest</td>
<td>1</td>
<td></td>
<td></td>
<td>0.361</td>
<td>2.886</td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td>36.949</td>
<td>295.610</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Force evaluation time GPU/CPU: 29.187 ms/16.954 ms = 1.722

Performance: 4.681 (ns/day) 15.147 (hour/ns)

Pocket

- How to avoid it?
  - use less CPU cores? → won't improve absolute performance
  - use faster GPU → not always possible...

GPU “too slow” (wrt CPU) → CPU idles while waiting for forces
CPU-GPU load balancing: in action

- Time consecutive cut-off settings, **pick fastest**
  - need to adjust PME grid ⇒ discrete steps

- Robust:
  - discard first timings
  - re-try if fluctuation is noticed
# CPU-GPU load balancing: the result

<table>
<thead>
<tr>
<th>Computing:</th>
<th>Nodes</th>
<th>Th.</th>
<th>Count</th>
<th>Wall t (s)</th>
<th>G-Cycles</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neighbor search</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>1.647</td>
<td>13.180</td>
<td>5.2</td>
</tr>
<tr>
<td>Launch GPU ops.</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>0.086</td>
<td>0.689</td>
<td>0.3</td>
</tr>
<tr>
<td>Force</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>6.357</td>
<td>50.863</td>
<td>19.9</td>
</tr>
<tr>
<td>PME mesh</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>18.204</td>
<td>145.640</td>
<td>56.9</td>
</tr>
<tr>
<td>Wait GPU local</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>0.024</td>
<td>0.193</td>
<td>0.1</td>
</tr>
<tr>
<td>NB X/F buffer ops.</td>
<td>1</td>
<td>3</td>
<td>2002</td>
<td>0.647</td>
<td>5.174</td>
<td>2.0</td>
</tr>
<tr>
<td>Write traj.</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.323</td>
<td>2.588</td>
<td>1.0</td>
</tr>
<tr>
<td>Update</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>0.619</td>
<td>4.950</td>
<td>1.9</td>
</tr>
<tr>
<td>Constraints</td>
<td>1</td>
<td>3</td>
<td>1001</td>
<td>3.753</td>
<td>30.025</td>
<td>11.7</td>
</tr>
<tr>
<td>Rest</td>
<td>1</td>
<td></td>
<td></td>
<td>0.311</td>
<td>2.485</td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1</td>
<td></td>
<td></td>
<td><strong>31.971</strong></td>
<td><strong>255.786</strong></td>
<td><strong>100.0</strong></td>
</tr>
</tbody>
</table>

Force evaluation time GPU/CPU: 19.732 ms/19.727 ms = 1.000

(ns/day) *(hour/ns)*

Performance: 6.279 11.350
CUDA overhead & scaling issues

Kernel scaling deteriorating: the GPU can’t keep up with the CPU => CPU waiting

Overhead with small systems: launching the GPU operations takes up to 15%!

GPUs not designed for \(~0.2 \text{ ms/step} = 5000 \text{ FPS}\)

Overhead with small systems: launching the GPU operations takes up to 15%!

Straight from the log file performance summary:

Runtime breakdown of GPU accelerated runs with:

- Hardware: Intel Core i7-3930 12T + GeForce GTX 680
- Settings: PME, rc>=0.9, nstlist=20

Overhead with small systems: launching the GPU operations takes up to 15%!

GPUs not designed for \(~0.2 \text{ ms/step} = 5000 \text{ FPS}\)

Kernel scaling deteriorating: the GPU can’t keep up with the CPU => CPU waiting
What to expect?

single-node performance

- Systems:
  - RNase: 24k/16.8k atoms
  - GLIC: 144k atoms
  - villin: 8k atoms

- Settings:
  - PME, rcoul $\geq$ 0.9 nm, LJ cut-off 0.9 nm
  - constraints: all bonds
  - 2 fs/5 fs (with vsites)
What to expect?  
strong scaling on Cray XK7

- System: ADH solvated protein, 134k
- Settings:
  - PME, rc $\geq 0.9$ nm, dt = 2 fs, all-bond constraints
- Notes:
  - **K20X: 3.5x speedup**
  - **Peak: 130 atoms/core**
  - PME tasks faster above 4 nodes
  - Optimal: 2-8 ranks/node & GPU
  - Limiting factors:
    - load imbalance
    - CUDA RT overhead
      (>15% at 64 nodes)
Further reading

- GROMACS wiki page on acceleration & parallelization
  http://www.gromacs.org/Documentation/Acceleration_and_parallelization

- GROMACS wiki page on cut-off scheme
  http://www.gromacs.org/Documentation/Cut-off_schemes

- GROMACS manual
Additional material
Pair list pruning with GPUs

- All-vs-all atom distance check expensive

- Solution:
  - pair search: only cluster bounding box distance
  - prune on GPU using **warp vote**:
    
    ![None](image)
    of the atom-pairs within range if:
    
    ```none
    !__any(r2 < rlist_sq)
    ```

- 20-30% overhead
  - prunes 30-50% of the pairs
  - only every pair-search step!
# Ewald kernel pair force rate

<table>
<thead>
<tr>
<th>PU</th>
<th>SIMD width</th>
<th>Ewald</th>
<th>$M \times N$</th>
<th>pairs/kcycle</th>
<th>1 thread</th>
<th>IPC</th>
<th>flops/pair</th>
<th>flops/cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>4</td>
<td>ana.</td>
<td>$1 \times 1$</td>
<td>51</td>
<td>-31%</td>
<td>2.20</td>
<td>66</td>
<td>3.4</td>
</tr>
<tr>
<td>SB</td>
<td>8</td>
<td>ana.</td>
<td>$1 \times 1$</td>
<td>63</td>
<td>-16%</td>
<td>1.98</td>
<td>66</td>
<td>4.2</td>
</tr>
<tr>
<td>SB</td>
<td>4</td>
<td>tab.</td>
<td>$4 \times 4$</td>
<td>111</td>
<td>-15%</td>
<td>2.42</td>
<td>43</td>
<td>4.8</td>
</tr>
<tr>
<td>SB</td>
<td>8</td>
<td>tab.</td>
<td>$4 \times 4$</td>
<td>147</td>
<td>-26%</td>
<td>2.26</td>
<td>43</td>
<td>6.3</td>
</tr>
<tr>
<td>SB</td>
<td>8</td>
<td>tab.</td>
<td>$4 \times 8$</td>
<td>134</td>
<td>-6%</td>
<td>1.88</td>
<td>43</td>
<td>5.8</td>
</tr>
<tr>
<td>SB</td>
<td>4</td>
<td>ana.</td>
<td>$4 \times 4$</td>
<td>110</td>
<td>-14%</td>
<td>2.40</td>
<td>68</td>
<td>7.5</td>
</tr>
<tr>
<td>SB</td>
<td>8</td>
<td>ana.</td>
<td>$4 \times 4$</td>
<td>139</td>
<td>-11%</td>
<td>1.76</td>
<td>68</td>
<td>9.5</td>
</tr>
<tr>
<td>SB</td>
<td>8</td>
<td>ana.</td>
<td>$4 \times 8$</td>
<td>137</td>
<td>+1%</td>
<td>1.52</td>
<td>68</td>
<td>9.3</td>
</tr>
<tr>
<td>BD</td>
<td>4</td>
<td>ana.</td>
<td>$4 \times 4$</td>
<td>114</td>
<td></td>
<td>2.16</td>
<td>68</td>
<td>7.8</td>
</tr>
<tr>
<td>Fermi</td>
<td>32</td>
<td>tab.</td>
<td>$8 \times 4$</td>
<td>549</td>
<td></td>
<td>1.66</td>
<td>41</td>
<td>24</td>
</tr>
<tr>
<td>Kepler2</td>
<td>32</td>
<td>tab.</td>
<td>$8 \times 4$</td>
<td>1130</td>
<td></td>
<td>3.2</td>
<td>41</td>
<td>46</td>
</tr>
<tr>
<td>Kepler2</td>
<td>32</td>
<td>ana.</td>
<td>$8 \times 4$</td>
<td>1151</td>
<td></td>
<td>3.7</td>
<td>69</td>
<td>85</td>
</tr>
</tbody>
</table>

SB: Intel Sandy Bridge  
BD: Bulldozer  
Fermi: GF100  
Kepler2: GK110  

cluster sizes  
1x1: traditional neighbor list-based scheme
## Ewald kernel effective pair force rate

<table>
<thead>
<tr>
<th>PU</th>
<th>elect.</th>
<th>$M \times N$</th>
<th>pairs/kcycle</th>
<th>list upd. (steps)</th>
<th>buffer (nm)</th>
<th>effective pair ratio</th>
<th>ratio vs 1x1</th>
<th>effective pairs/kcycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>SB</td>
<td>RF</td>
<td>1x1</td>
<td>76</td>
<td>10</td>
<td>0.09</td>
<td>0.77</td>
<td></td>
<td>59</td>
</tr>
<tr>
<td>SB</td>
<td>RF</td>
<td>4x4</td>
<td>223</td>
<td>10</td>
<td>0.07</td>
<td>0.48</td>
<td>0.62</td>
<td>107</td>
</tr>
<tr>
<td>Kepler2</td>
<td>RF</td>
<td>8x4</td>
<td>1351</td>
<td>10</td>
<td>0.07</td>
<td>0.40</td>
<td>0.52</td>
<td>544</td>
</tr>
<tr>
<td>Kepler2</td>
<td>RF</td>
<td>8x4</td>
<td>1386</td>
<td>20</td>
<td>0.10</td>
<td>0.37</td>
<td>0.48</td>
<td>519</td>
</tr>
<tr>
<td>SB</td>
<td>Ewald</td>
<td>1x1</td>
<td>63</td>
<td>10</td>
<td>0.05</td>
<td>0.86</td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>SB</td>
<td>Ewald</td>
<td>4x4</td>
<td>139</td>
<td>10</td>
<td>0.00</td>
<td>0.53</td>
<td>0.61</td>
<td>74</td>
</tr>
<tr>
<td>Kepler2</td>
<td>Ewald</td>
<td>8x4</td>
<td>1151</td>
<td>10</td>
<td>0.00</td>
<td>0.47</td>
<td>0.55</td>
<td>545</td>
</tr>
<tr>
<td>Kepler2</td>
<td>Ewald</td>
<td>8x4</td>
<td>1181</td>
<td>20</td>
<td>0.03</td>
<td>0.44</td>
<td>0.51</td>
<td>521</td>
</tr>
</tbody>
</table>

SB: Intel Sandy Bridge
Fermi: GF100
Kepler2: GK110

1x1: traditional neighbor list-based scheme
Multi-GPU weak scaling

![Graph showing multi-GPU weak scaling with different system sizes and iteration times per 1000 atoms (ms/step). The graph includes lines for 1x, 2x, and 4x GPUs, with separate lines for CUDA and CPU totals.]
Multi-GPU weak scaling cont.

![Graph showing iteration time per 1,000 atoms (ns/step) vs. system size per GPU (1,000s of atoms) for different configurations.

- **PME**
  - 1xC2075 CUDA F kernel
  - 1xC2075 CPU total
  - 2xC2075 CPU total
  - 4xC2075 CPU total

The graph illustrates the performance of different GPU configurations across varying system sizes. The iteration time decreases as the system size increases, with notable differences between CPU and GPU configurations. The 1xC2075 CUDA F kernel shows the best performance across all sizes, followed by 1xC2075 CPU total, 2xC2075 CPU total, and 4xC2075 CPU total configurations, which show slight increases in iteration time with larger system sizes.
Strong scaling

- **Benchmark system**: water box with 1.5M particles.
- **Settings**: electrostatics cut-off auto-tuned >0.9 nm for PME and 0.9 nm for reaction-field, LJ cut-off 0.9 nm, 2 fs time steps.
- **Hardware**: Bullx cluster nodes with 2x Intel Xeon E5649 (6C), 2x NVIDIA Tesla M2090, 2x QDR Infiniband 40 Gb/s.