Experiences with Multi-GPU Acceleration in VMD

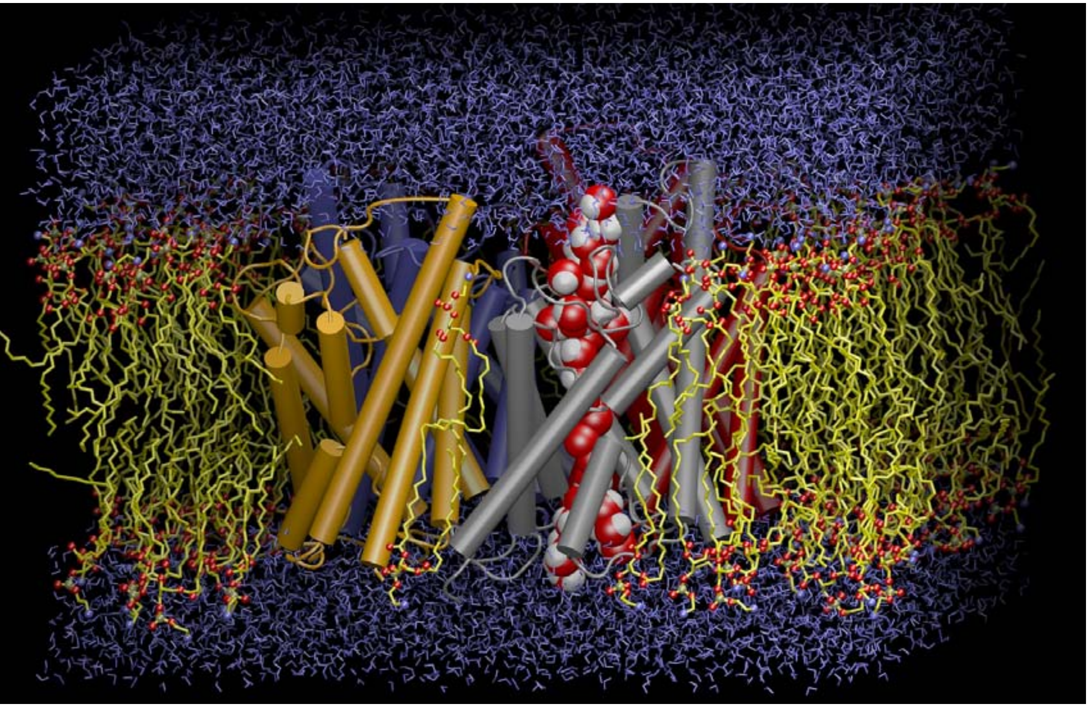
John Stone

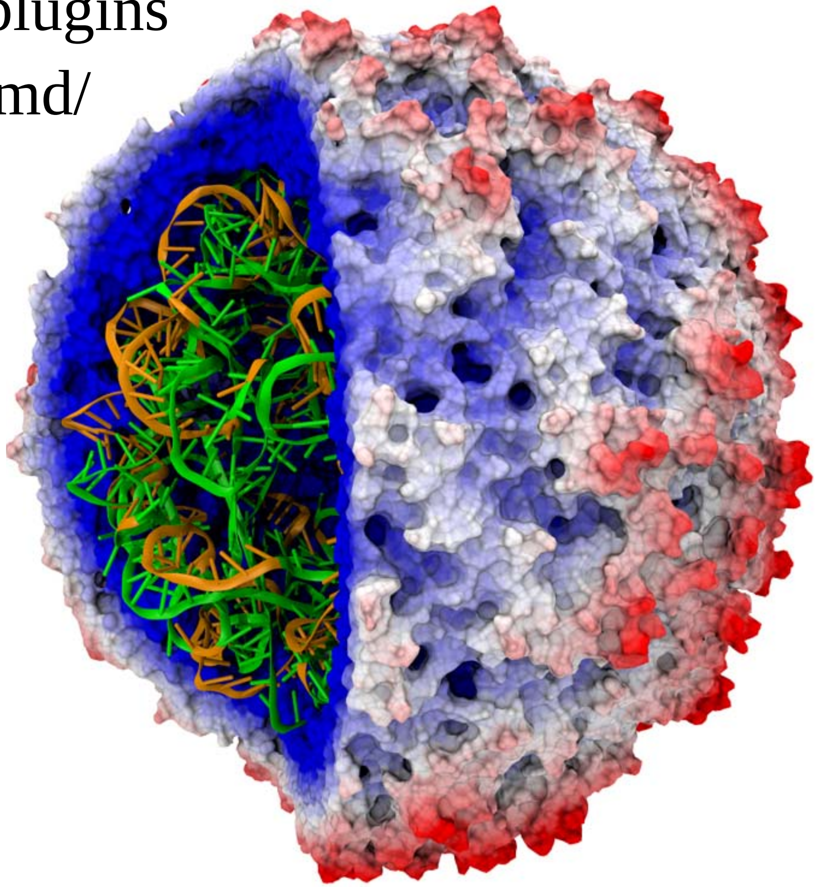
Path to Petascale: Adapting GEO/CHEM/ASTRO Applications for Accelerators and Accelerator Clusters

April 2, 2009

VMD – “Visual Molecular Dynamics”

- Visualization and analysis of molecular dynamics simulations, sequence data, volumetric data, quantum chemistry simulations, particle systems, …
- User extensible with scripting and plugins
- http://www.ks.uiuc.edu/Research/vmd/

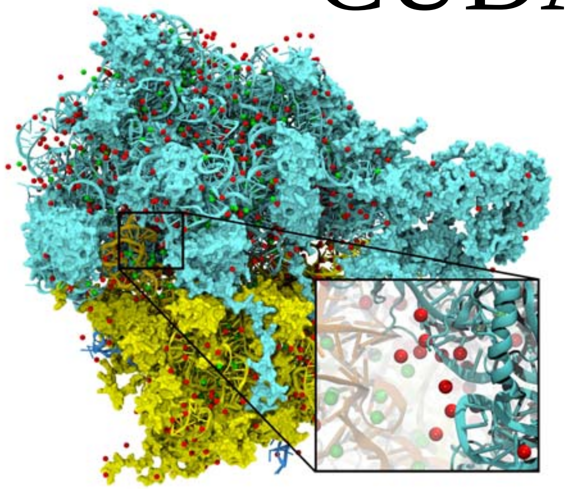




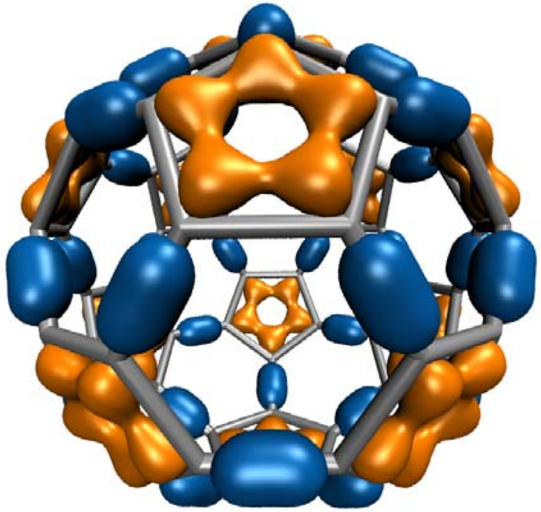
Range of VMD Usage Scenarios

- Users run VMD on a diverse range of hardware: laptops, desktops, clusters, and supercomputers
- Typically used as a desktop science application, for interactive 3D molecular graphics and analysis
- Can also be run in pure text mode for numerically intensive analysis tasks, batch mode movie rendering, etc…
- GPU acceleration provides an opportunity to make some **slow, or batch** calculations capable of being run **interactively, or on-demand…**

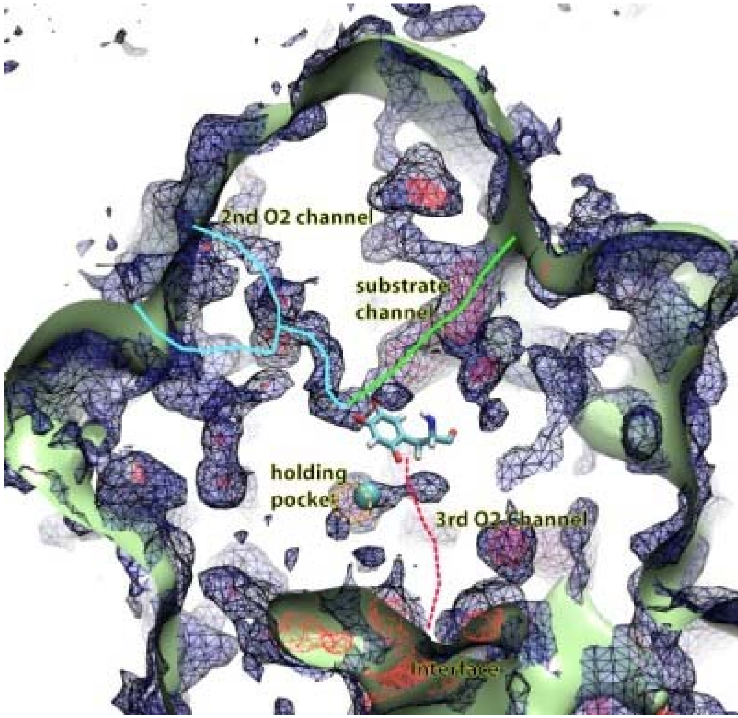
CUDA Acceleration in VMD



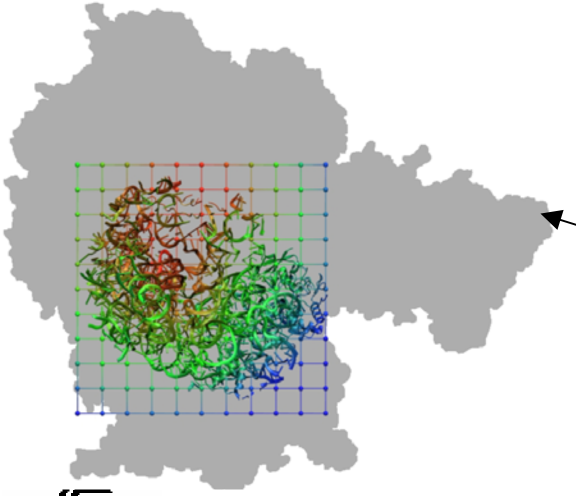
Electrostatic field calculation, ion placement



Molecular orbital calculation and display



Imaging of gas migration pathways in proteins with implicit ligand sampling



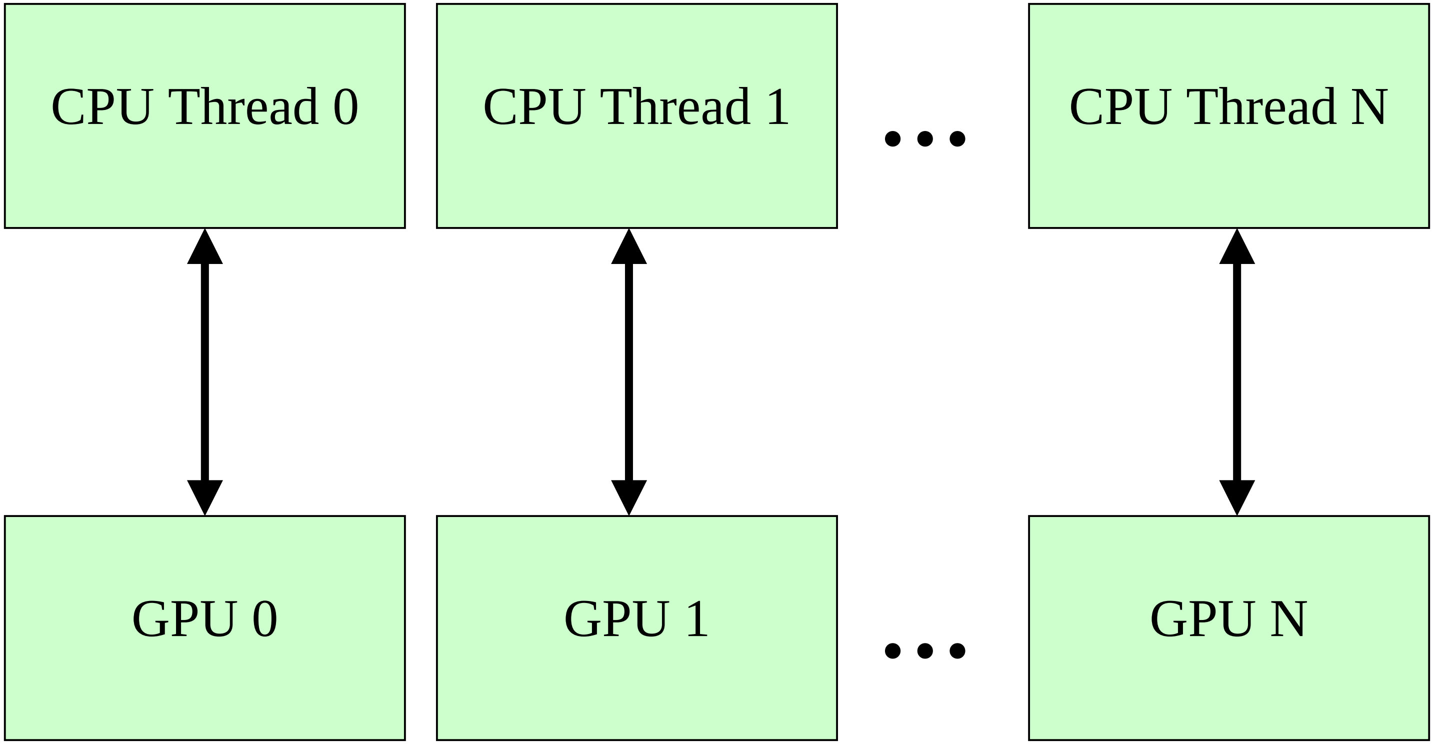
Need for Multi-GPU CUDA Acceleration in VMD

- Ongoing increases in supercomputing resources at NSF centers such as NCSA enable increased simulation complexity, fidelity, and longer time scales…
- Drives need for more visualization and analysis capability at the desktop and on clusters running batch analysis jobs
- Desktop use is the most compute-resource-limited scenario, where **GPUs can make a big impact**…

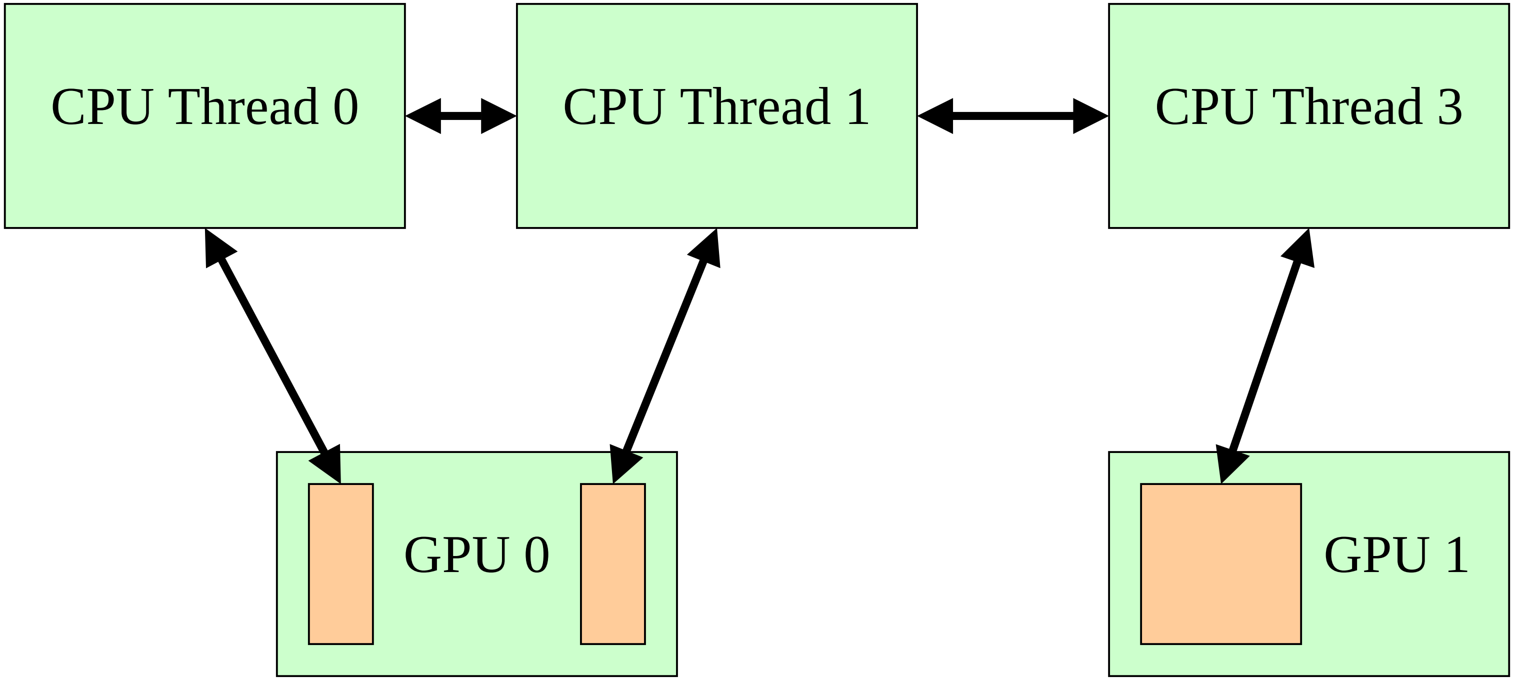
CUDA Runtime API Basics

- A single host thread can attach to and communicate with a single GPU
- A single GPU can be shared by multiple threads/processes, but only one such context is active at a time
- In order to use more than one GPU, multiple host threads or processes must be created

One Host Thread Per GPU (Strategy used by VMD)



Host Thread Contexts Cannot Directly Share GPU Memory, Must Communicate/Share on Host Side



Even threads sharing the same GPU cannot exchange data by reading each other's GPU memory

CUDA Runtime APIs for Enumerating and Selecting GPU Devices

- Query available hardware:
  - cudaGetDeviceCount(), cudaGetDeviceProperties()
- Attach a GPU device to a host thread:
  - cudaSetDevice()
  - This is a permanent binding, once set it cannot be subsequently changed
  - Binding a GPU device to a host thread has overhead:
    - 1st CUDA call after binding takes ~100 milliseconds

Launching/Collecting Host Threads (POSIX Threads)

```
void *cudaworkerthread(void *voidparms); // worker function

…
/* spawn child threads to do the work */
for (i=0; i<numprocs; i++) {
  pthread_create(&threads[i], cudaworkerthread, &parms[i]);
}

/* “join” the threads after work is done */
for (i=0; i<numprocs; i++)
  pthread_join(threads[i], NULL);
}
```

VMD Threading and Work Distribution Abstractions

- Wrap low-level OS threading APIs with convenient abstractions that launch, synchronize, and collect groups of GPU worker threads
- Work distribution routines (shared iterators, akin to a “parallel for loop”, work queues, etc)
- Routines to generate a persistent pool of worker threads that sleep waiting for work to run, amortizing one-time CUDA device initialization, optimizes performance for multi-GPU kernels that have runtimes below 1 second…

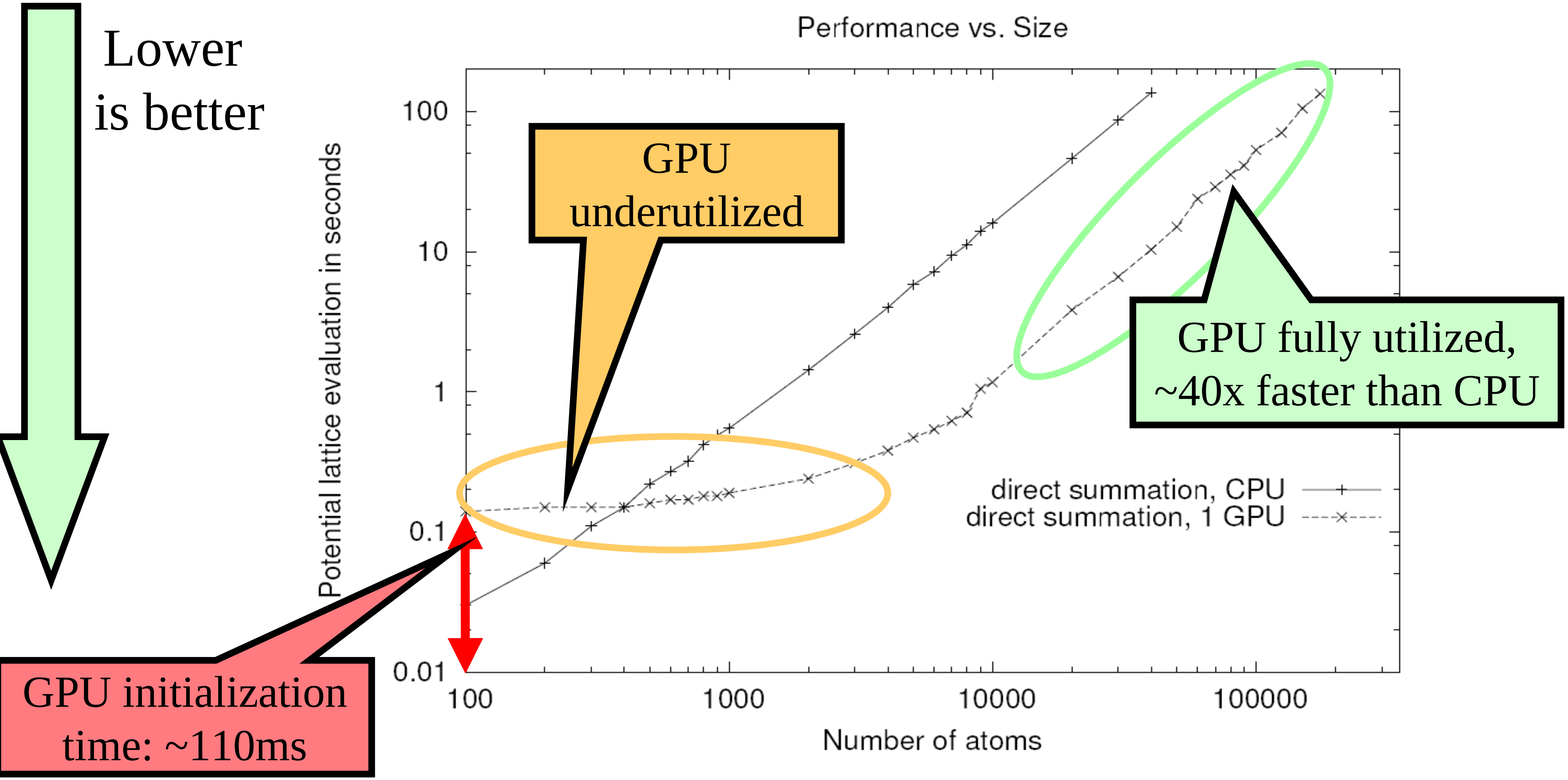
Why Not TBB, Library X, Y, …

- We use the same threading primitives for both the multi-core and CUDA code in VMD, portability is important, minimize dependencies on external libraries
- Intel Threading Building Blocks (TBB) library contains many of the abstractions we want, but…
- Recent versions not (yet?) ported to all platforms/compilers VMD supports
- Uses a cooperative (no preemption) scheduler which is unable to cope with blocking disk I/O, Host-GPU DMA I/O, blocking CUDA calls, etc…

Classification of VMD Workloads

- Analysis computations:
  - Driven by user scripts
  - May run for seconds, minutes, or hours
- Interactive visualization, trajectory animation:
  - Computations used to generate visual representation
  - In all cases, total computation+rendering time should be on the order of 0.1 seconds or less…
  - Sensitive to latency

Direct Coulomb Summation Runtime



Lower is better

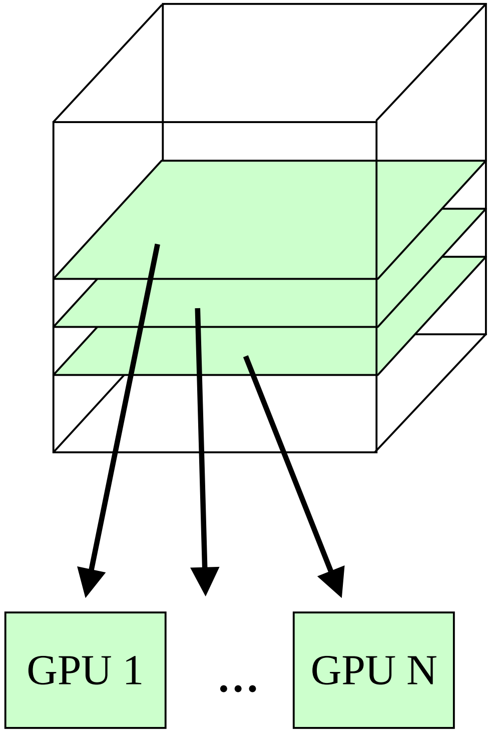
GPU underutilized

GPU fully utilized, ~40x faster than CPU

GPU initialization time: ~110ms

Accelerating molecular modeling applications with graphics processors.
J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten.
J. Comp. Chem., 28:2618-2640, 2007.

Multi-GPU Direct Coulomb Summation



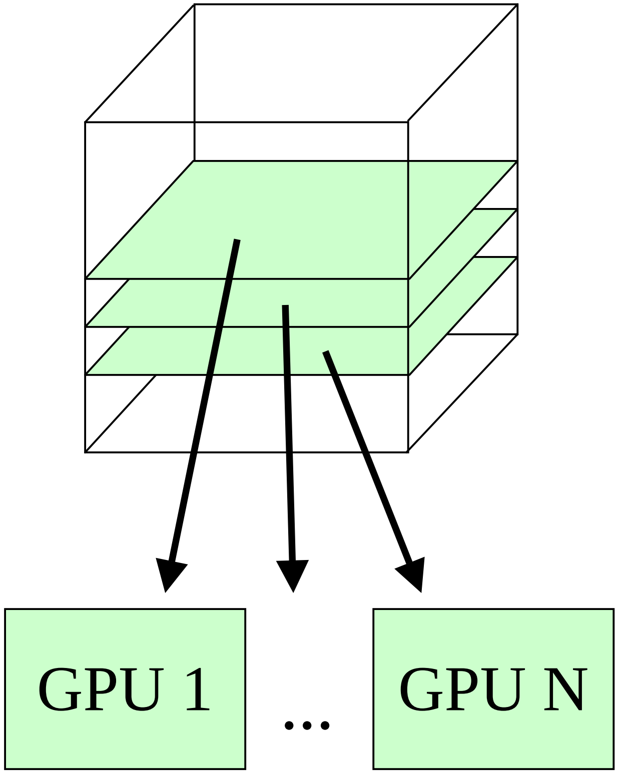
NCSA GPU Cluster
http://www.ncsa.uiuc.edu/Projects/GPUcluster/

	Evals/sec	TFLOPS	Speedup*
4-GPU (2 Quadroplex) Opteron node at NCSA	157 billion	1.16	176
4-GPU GTX 280 (GT200)	241 billion	1.78	271

*Speedups relative to Intel QX6700 CPU core w/ SSE

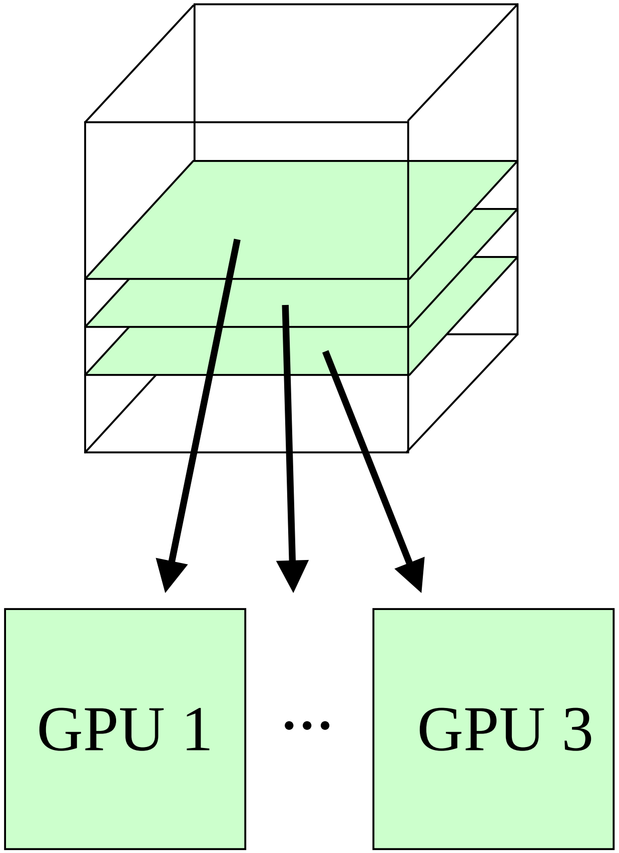
Multi-GPU Data-parallel Decomposition

- Many independent coarse-grain computations farmed out to pool of GPUs
- Work assignment can be explicit in the code, or controlled with a dynamic work scheduler of some sort
- May need to handle load imbalance, GPUs with varying capabilities, runtime errors, etc.



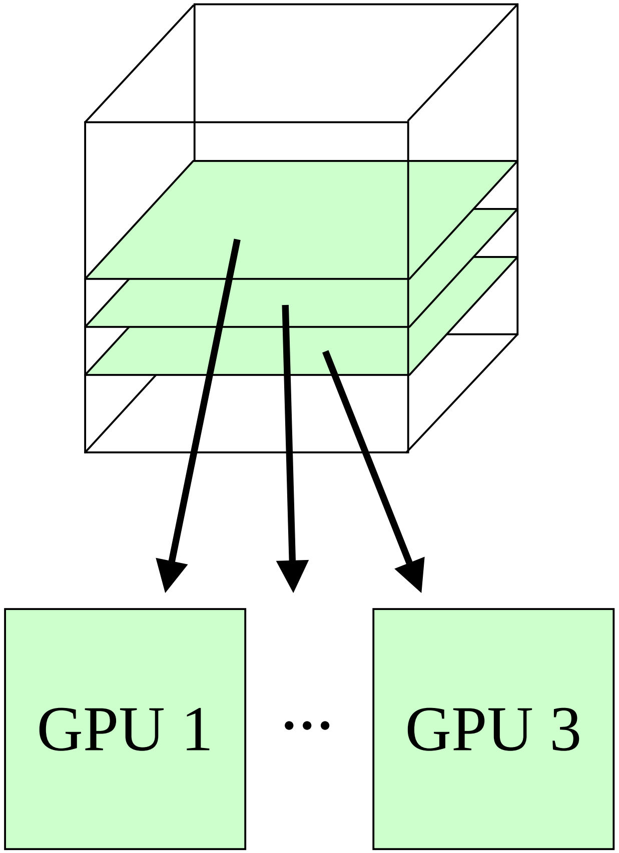
Multi-GPU Static Load Balance, Static Work Decomposition

- Static round-robin load balance:
  - Easy to code, explicit round robin decomposition
  - Low overhead, works well for short calculation runs
  - Can’t reschedule work on error/exception
  - Easy to port to multiple OSs



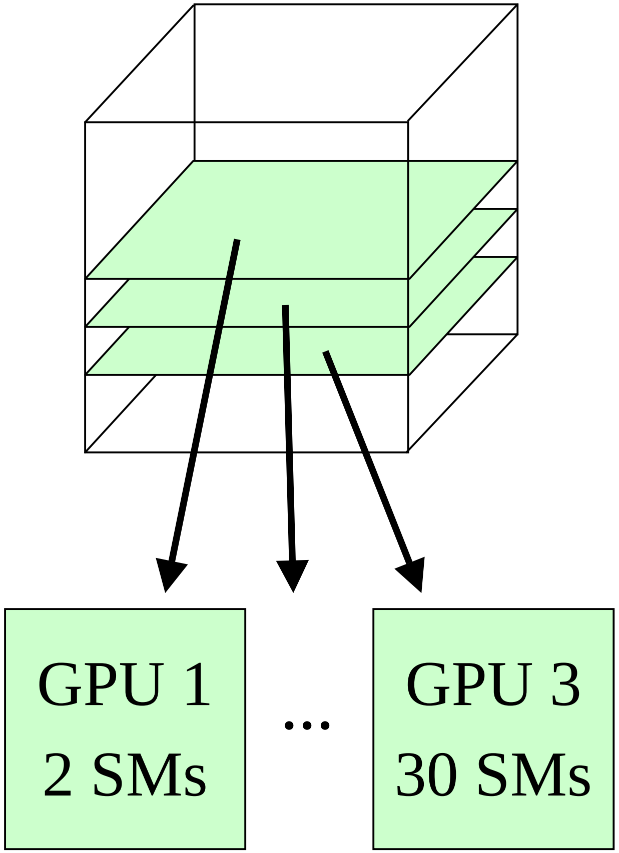
Multi-GPU
Static Work Decomposition

```
// Each GPU worker thread loops over
// subset of 2-D planes in a 3-D cube…
for (k=thrID; k<numplane; k+=thrCount) {
  // Process one plane of work…
  // Launch one CUDA kernel for each
  //   loop iteration taken…
  // Simple scheme, works well when GPUs
  //   and work units are nearly identical…
  // No provision for in-flight error handling
}
```



Multi-GPU Load Balance

- Many early CUDA codes assumed all GPUs were identical
- All new NVIDIA cards support CUDA, so a typical machine may have a diversity of GPUs of varying capability
- Static decomposition works poorly if you have diverse workload, or diverse GPUs, e.g. 2 SM, 16 SM, 30 SM

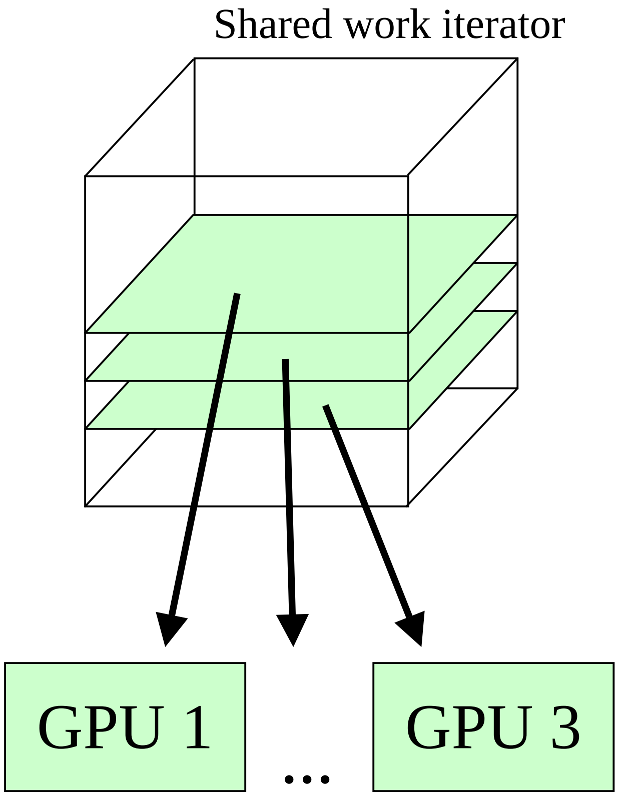


Multi-GPU Dynamic Load Balance, Shared Work Iterator

- Dynamic load balance, single shared iterator assigns slices to workers:
  - Replaces the **for** loop in static decomposition
  - Added overhead from mutex locks or atomic memory operations
  - Can reschedule/retry on error/exception by re-adding to a shared queue or exception stack
  - Still easy to port to multiple OSs

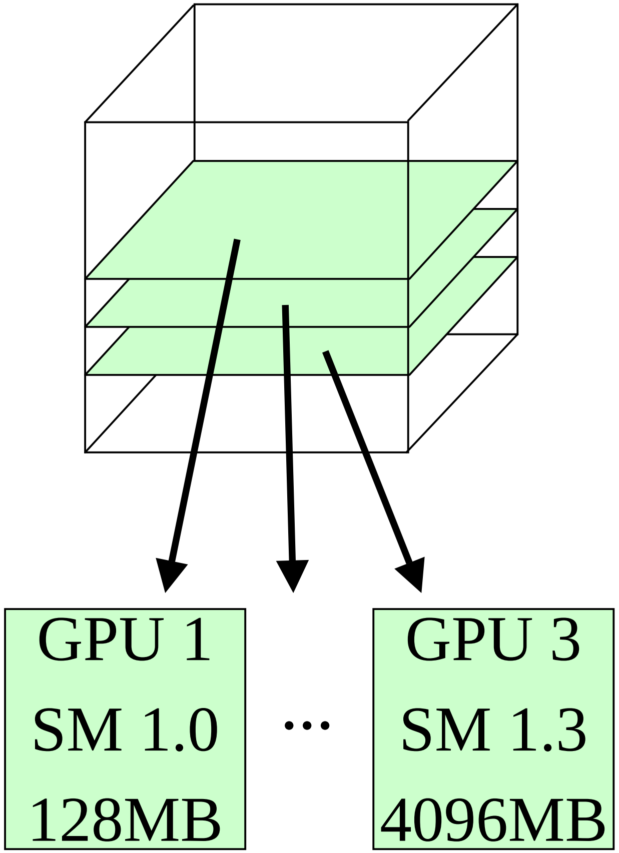
Multi-GPU Shared Work Iterator

```
// Each GPU worker thread loops over
// subset 2-D planes in a 3-D cube…
while (!iterator_next(&parms, &k) {
  // Process one plane of work…
  // Launch one CUDA kernel for each
  //   loop iteration taken…
  // Shared iterator automatically
  //   balances load on GPUs
}
```

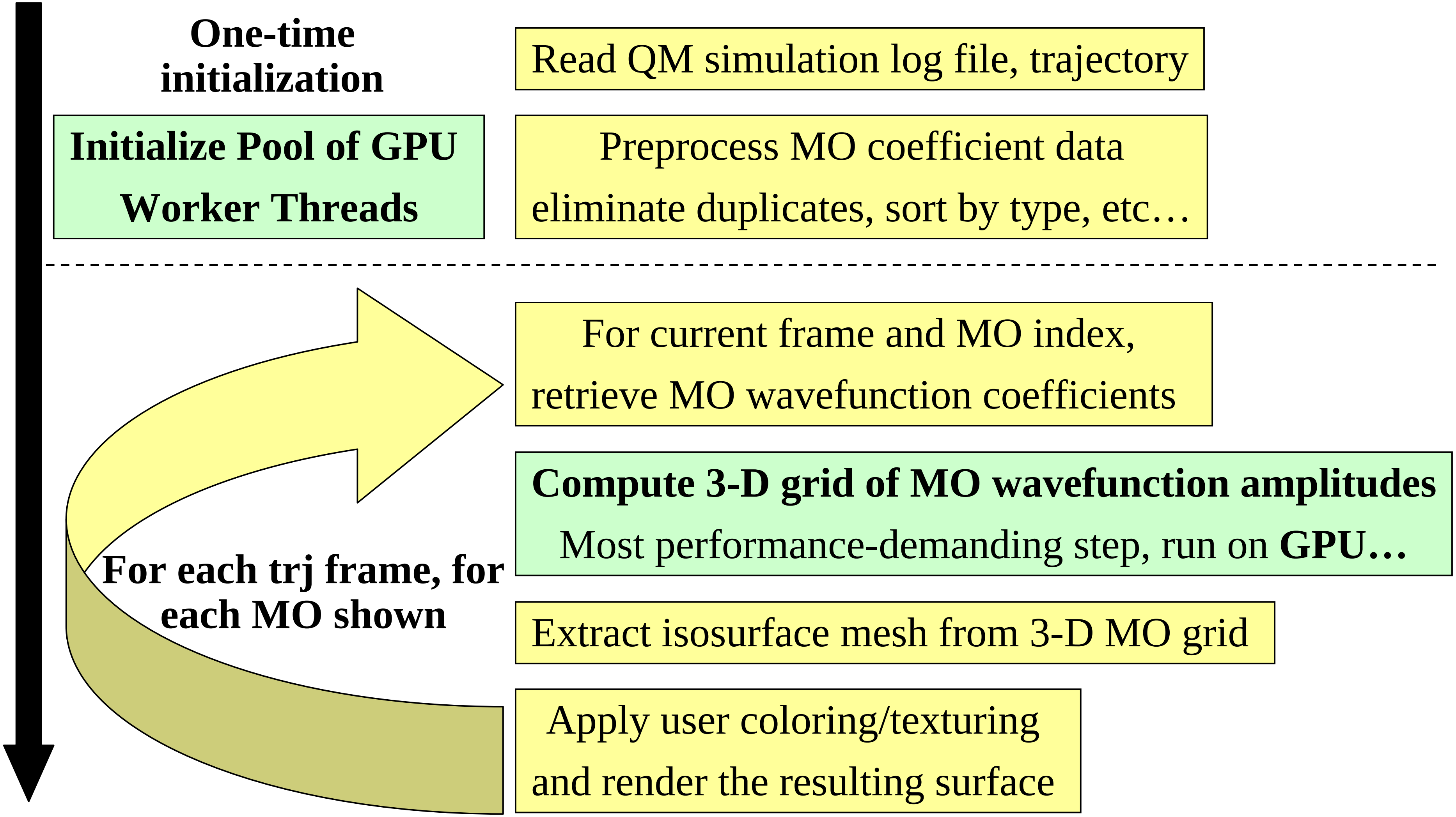


Multi-GPU Runtime Error/Exception Handling

- Competition for resources from other applications or the windowing system can cause runtime failures (e.g. GPU out of memory half way through an algorithm)
- Handling of algorithm exceptions (e.g. convergence failure, NaN result, etc)
- Need to handle and/or reschedule failed tiles of work



Molecular Orbital Computation and Display Process



One-time initialization

Read QM simulation log file, trajectory

Initialize Pool of GPU Worker Threads

Preprocess MO coefficient data eliminate duplicates, sort by type, etc…

For each trj frame, for each MO shown

For current frame and MO index, retrieve MO wavefunction coefficients

Compute 3-D grid of MO wavefunction amplitudes

Most performance-demanding step, run on **GPU…**

Extract isosurface mesh from 3-D MO grid

Apply user coloring/texturing and render the resulting surface

VMD Multi-GPU Molecular Orbital Performance Results for C_{60}

Kernel	Cores/GPUs	Runtime (s)	Speedup
CPU ICC-SSE	1	46.580	1.00
CPU ICC-SSE	4	11.74	3.97
CUDA-const-cache	1	0.400	116.45
CUDA-const-cache	2	0.205	227.21
CUDA-const-cache	3	0.144	323.47

Intel Q6600 CPU,
1x NVIDIA Quadro 5800, 2x Tesla C1060 GPUs,

Uses persistent thread pool to avoid GPU init overhead

Future Work

- Continued focus on low-latency GPU kernel launch/scheduling mechanisms
- Public release of the multi-GPU framework for easy use in other codes
- Add implementations that interoperate with or build on top of libraries like BOOST
- Possibly contribute patches for other libraries like TBB

Acknowledgements

- Theoretical and Computational Biophysics Group, IMPACT group, NVIDIA Center of Excellence, University of Illinois at Urbana-Champaign
- CUDA team at NVIDIA
- NIH support: P41-RR05969