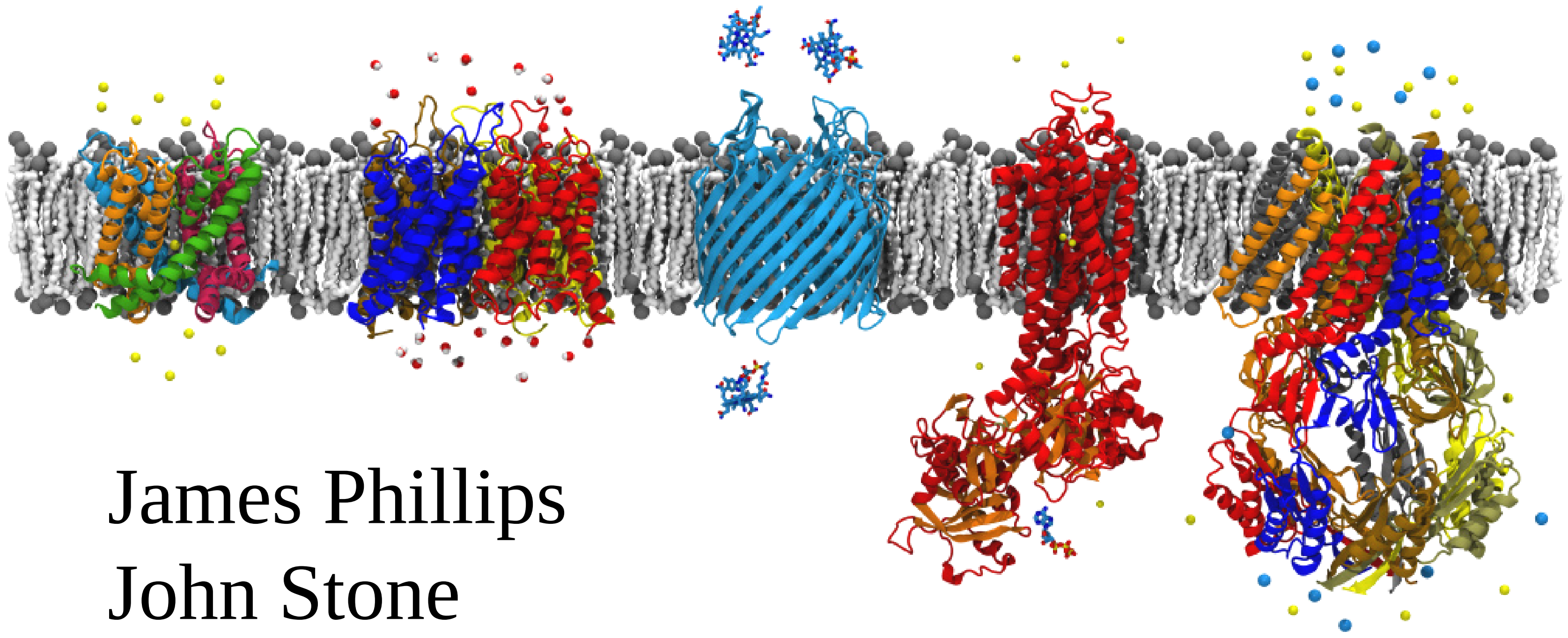
Accelerating NAMD with Graphics Processors



James Phillips
John Stone
Klaus Schulten

http://www.ks.uiuc.edu/Research/namd/

National Center for Research Resources

NAMD: Practical Supercomputing

- 24,000 users can't all be computer experts.
  - 18% are NIH-funded; many in other countries.
  - 4900 have downloaded more than one version.

- User experience is the same on all platforms.
  - No change in input, output, or configuration files.
  - Run any simulation on **any number of processors**.
  - Precompiled binaries available when possible.
- Desktops and laptops – setup and testing
  - x86 and x86-64 Windows, and Macintosh
  - Allow both shared-memory and network-based parallelism.

- Linux clusters – affordable workhorses
  - x86, x86-64, and Itanium processors
  - Gigabit ethernet, Myrinet, InfiniBand, Quadrics, Altix, etc

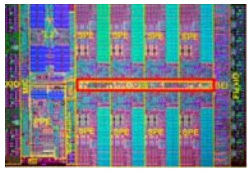
Phillips *et al.*, *J. Comp. Chem.* **26**:1781-1802, 2005.

Our Goal: Practical Acceleration

- Broadly applicable to scientific computing
  - Programmable by domain scientists
  - Scalable from small to large machines
- Broadly available to researchers
  - Price driven by commodity market
  - Low burden on system administration
- Sustainable performance advantage
  - Performance driven by Moore’s law
  - Stable market and supply chain

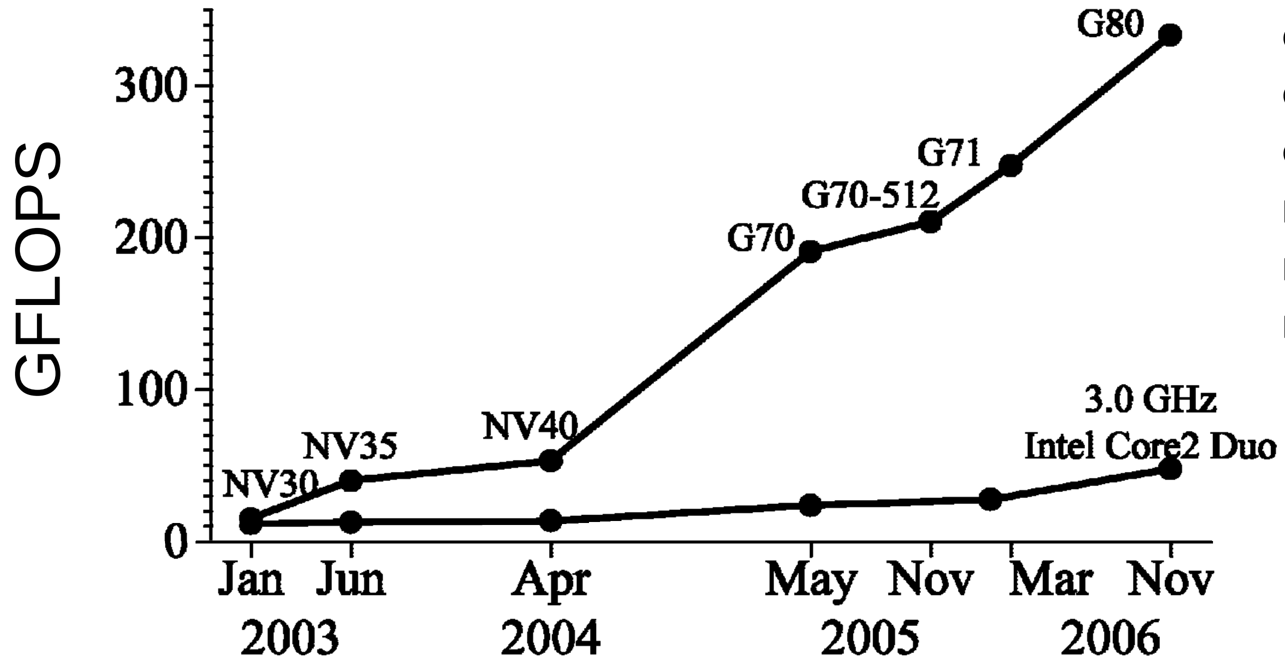
Acceleration Options for NAMD

- Outlook in 2005-2006:
  - FPGA reconfigurable computing (with NCSA)
    - Difficult to program, slow floating point, expensive
  - Cell processor (NCSA hardware)
    - Relatively easy to program, expensive
  - ClearSpeed (direct contact with company)
    - Limited memory and memory bandwidth, expensive
  - MDGRAPE
    - Inflexible and expensive
  - Graphics processor (GPU)
    - Program must be expressed as graphics operations



GPU vs CPU: Raw Performance

– Calculation: 450 GFLOPS vs 32 GFLOPS
– Memory Bandwidth: 80 GB/s vs 8.4 GB/s



G80 = GeForce 8800 GTX

G71 = GeForce 7900 GTX

G70 = GeForce 7800 GTX

NV40 = GeForce 6800 Ultra

NV35 = GeForce FX 5950 Ultra

NV30 = GeForce FX 5800

CUDA: Practical Performance

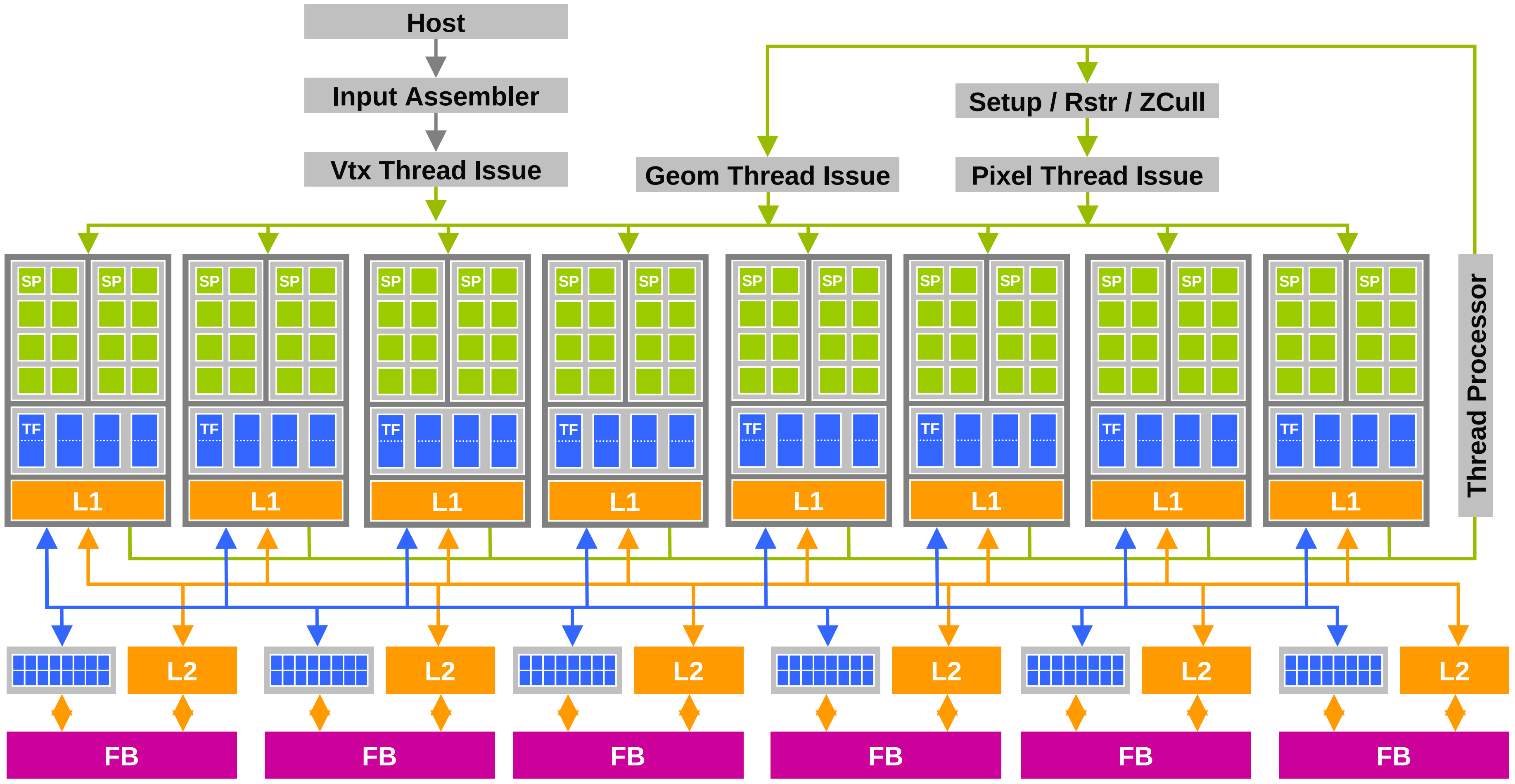
November 2006: NVIDIA announces CUDA for G80 GPU.

- CUDA makes GPU acceleration usable:
  - Developed and supported by NVIDIA.
  - No masquerading as graphics rendering.
  - New shared memory and synchronization.
  - No OpenGL or display device hassles.
  - Multiple processes per card (or vice versa).
- Resource and collaborators make it useful:
  - Experience from VMD development
  - David Kirk (Chief Scientist, NVIDIA)
  - Wen-mei Hwu (ECE Professor, UIUC)

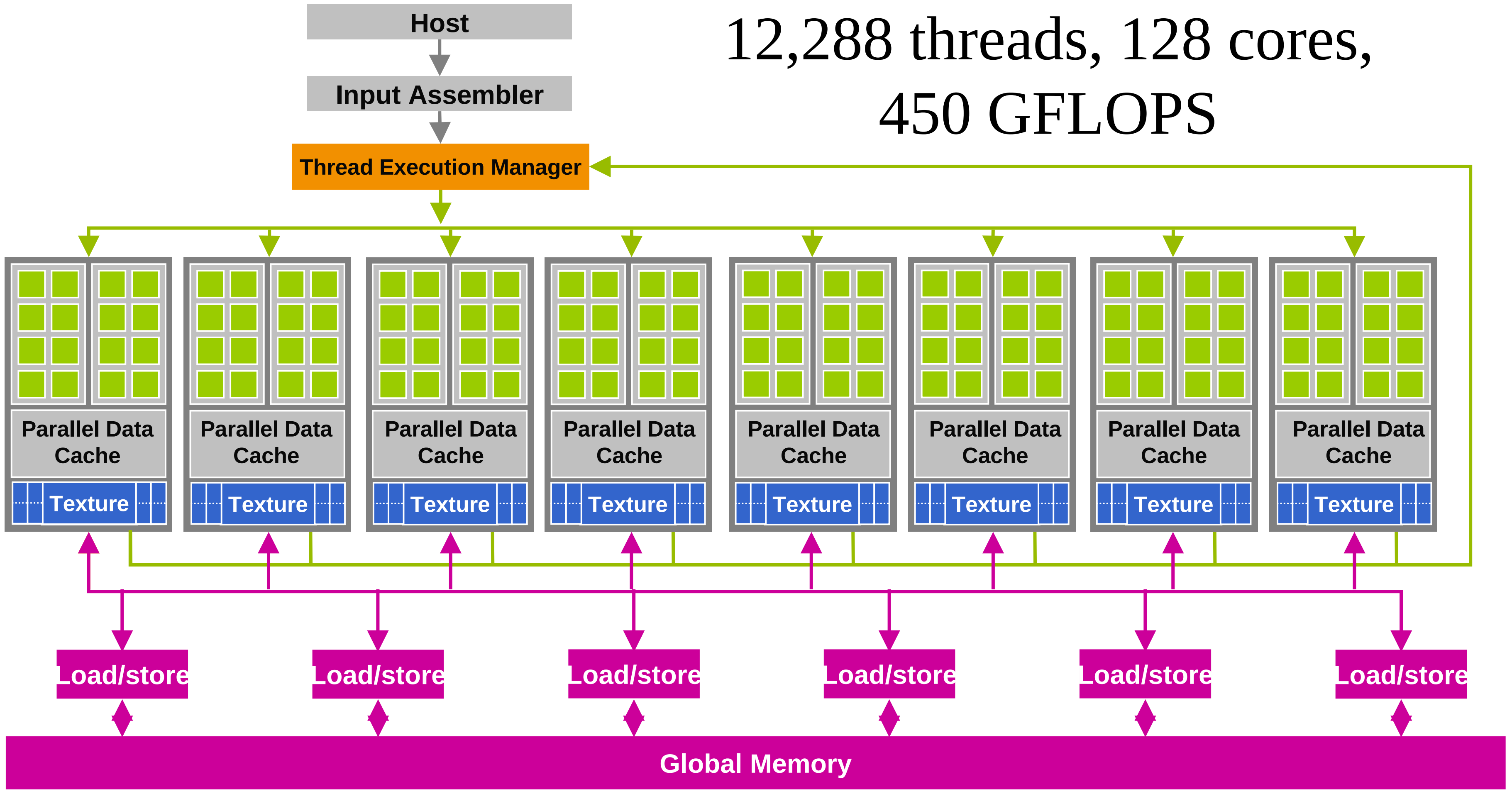
Fun to program (and drive)

Stone *et al., J. Comp. Chem.* **28**:2618-2640, 2007.

GeForce 8800 Graphics Mode



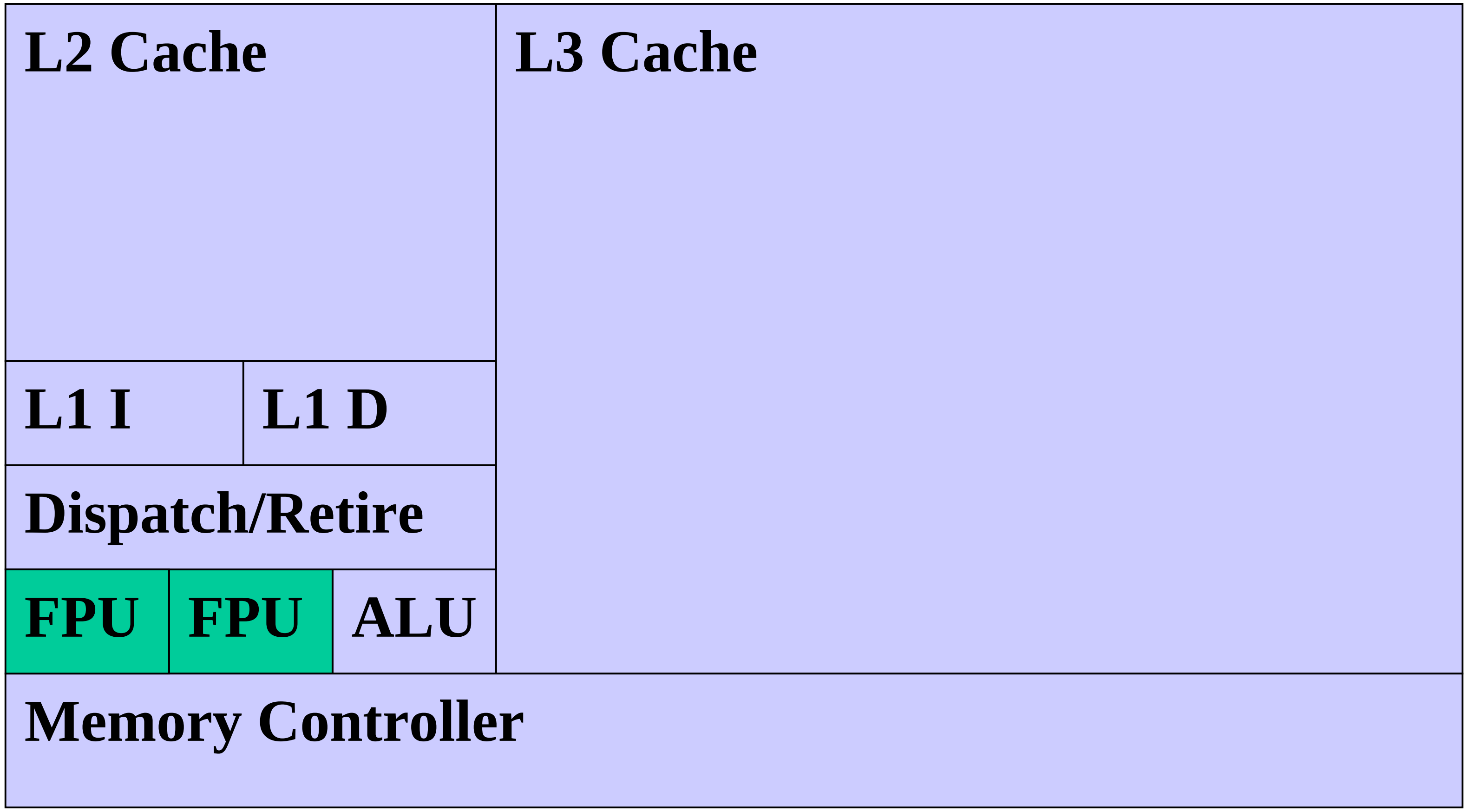
GeForce 8800 General Computing



12,288 threads, 128 cores, 450 GFLOPS

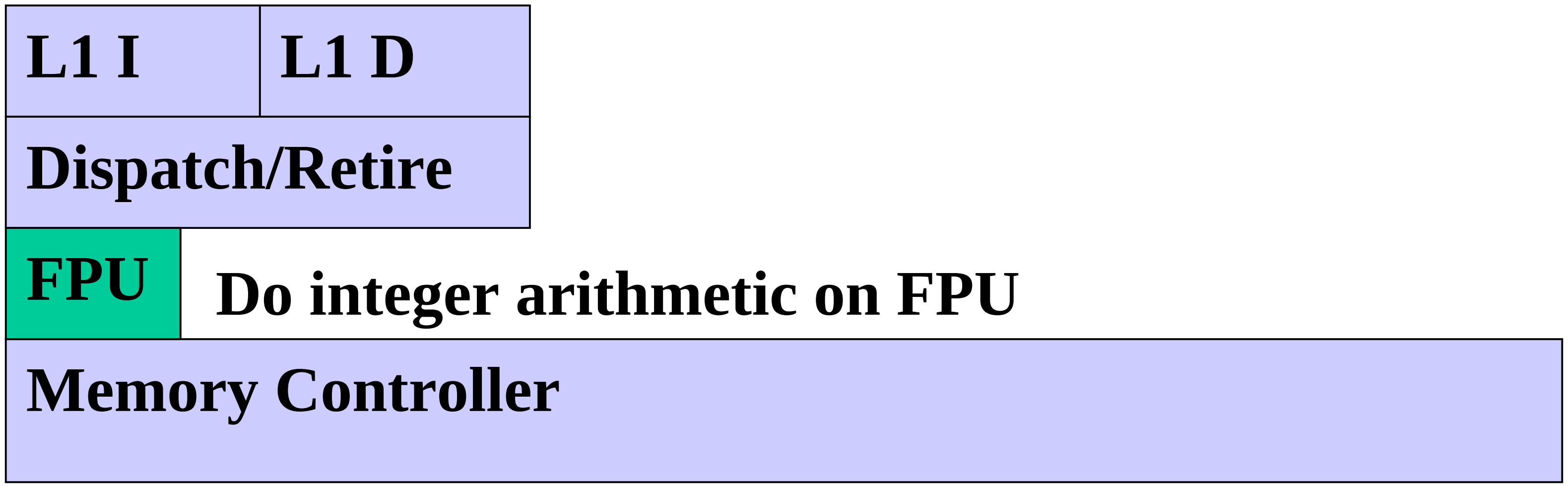
768 MB DRAM, 4GB/S bandwidth to CPU

Typical CPU Architecture



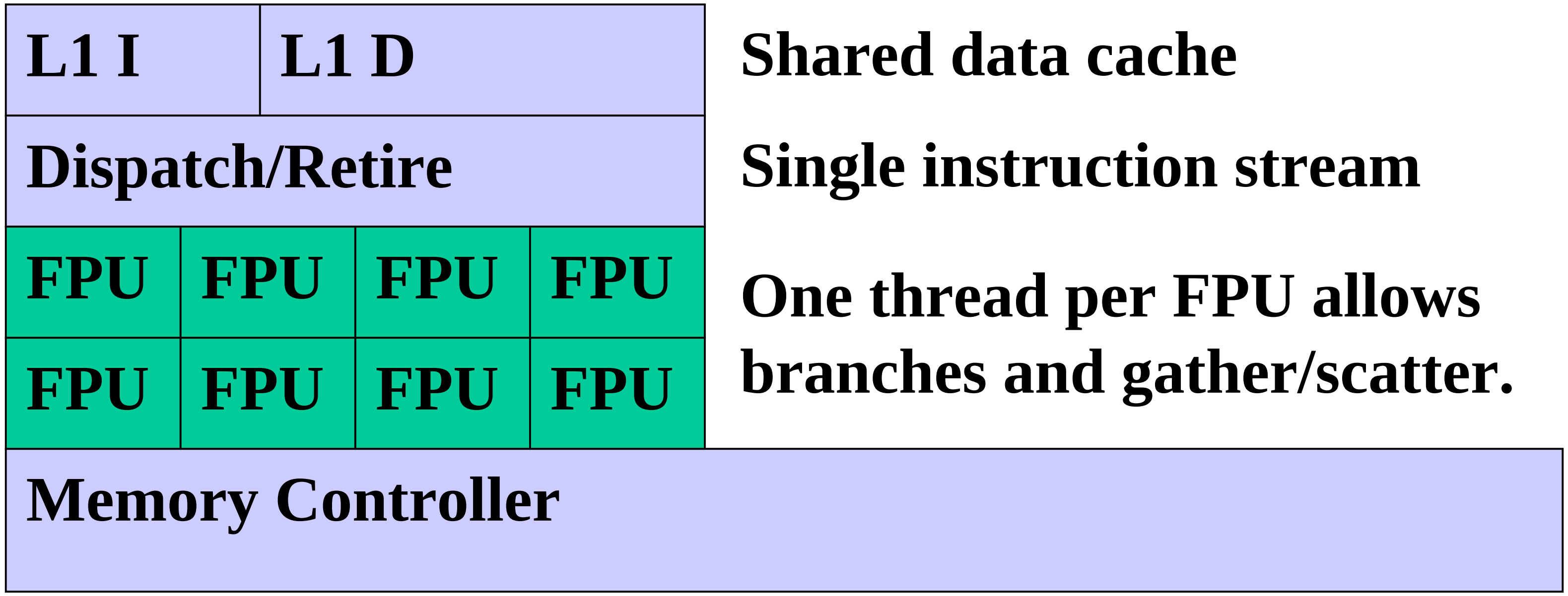
Minimize the Processor

No large caches or multiple execution units



Maximize Floating Point

8 FP pipelines per SIMD unit

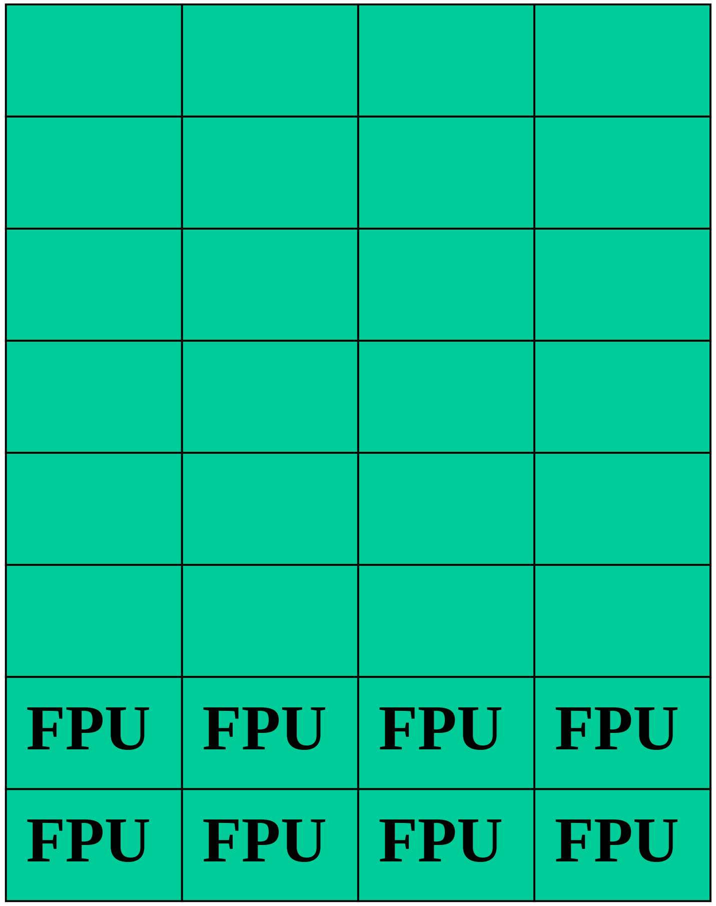


Shared data cache

Single instruction stream

One thread per FPU allows branches and gather/scatter.

Add More Threads



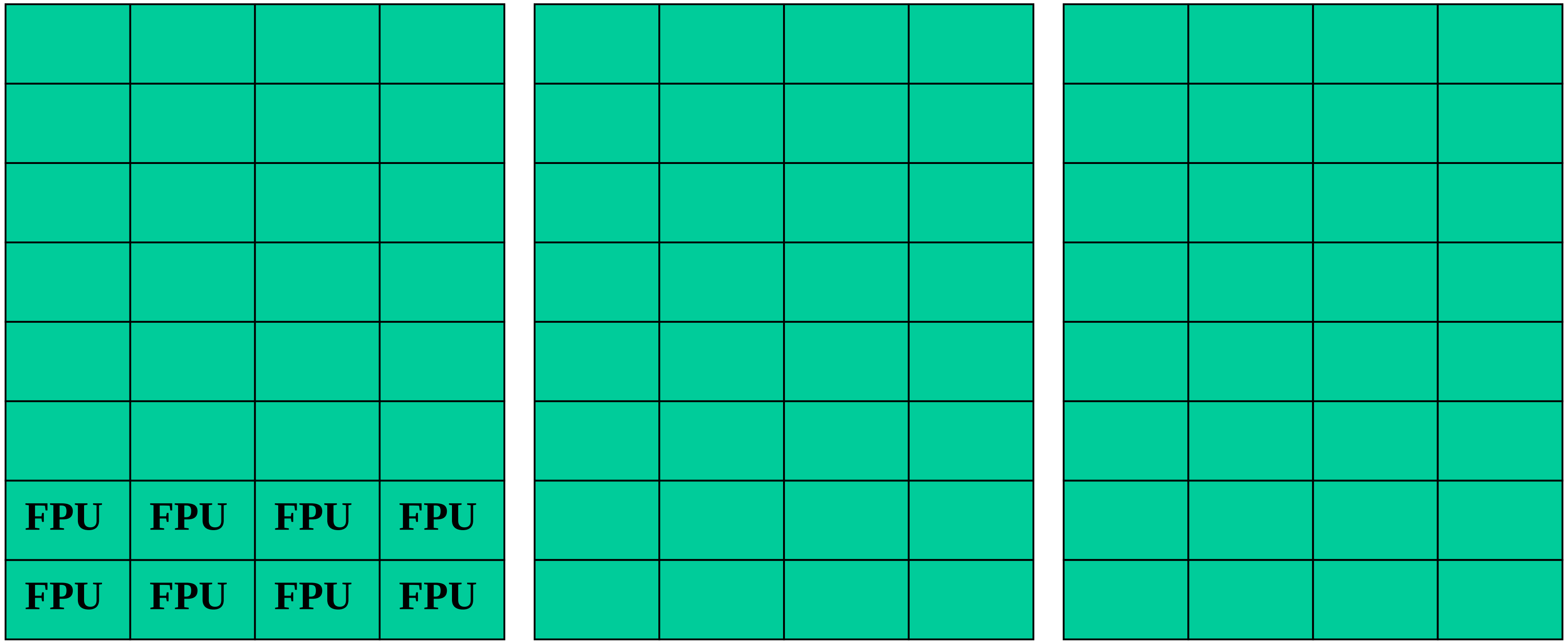
Pipeline 4 threads per FPU to hide 4-cycle instruction latency.

All 32 threads in a "warp" execute the same instruction.

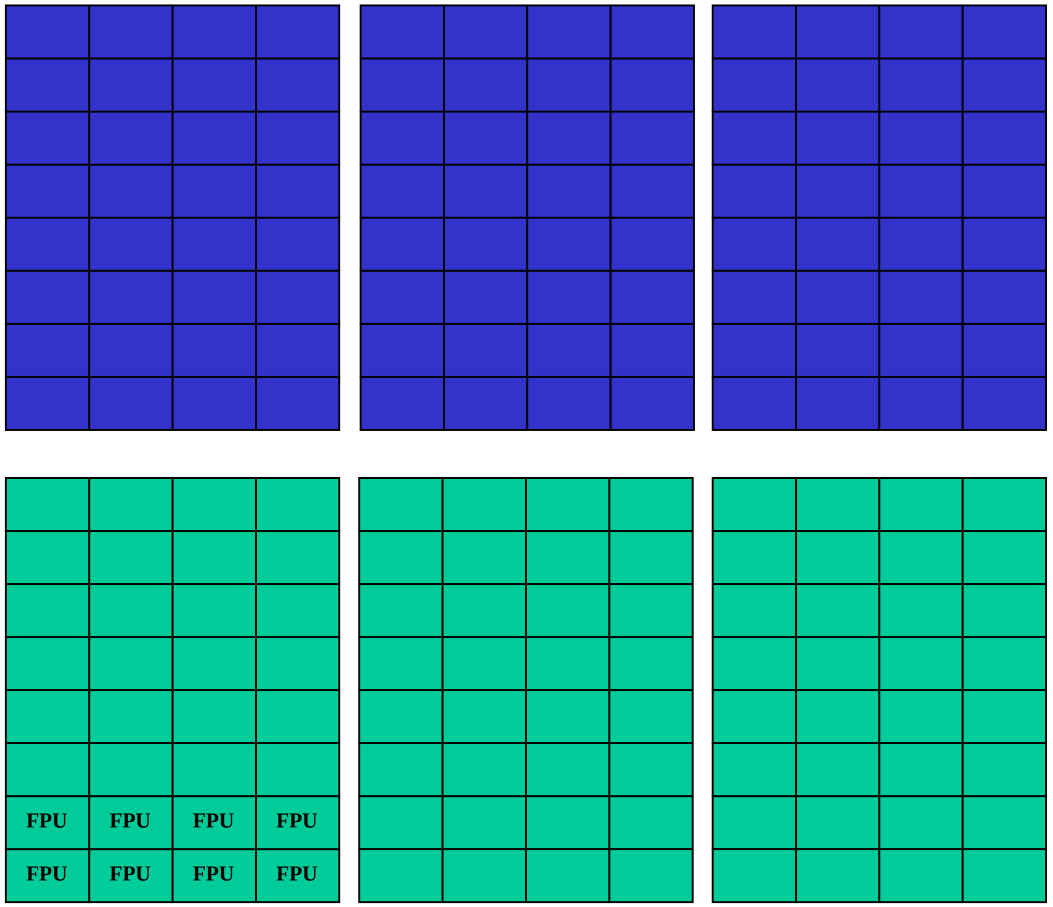
Divergent branches allowed through predication.

Add Even More Threads

Multiple warps in a "block" hide main memory latency and can synchronize to share data.



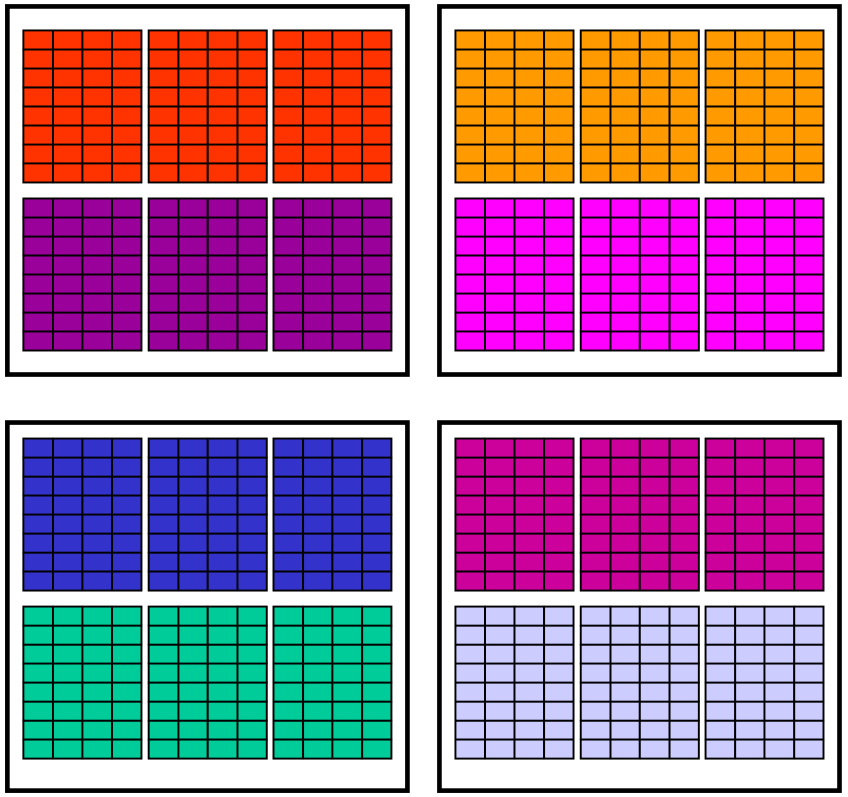
Add More Threads Again



Multiple blocks on a single multiprocessor hide both memory and synchronization latency.

All blocks execute a "kernel" function independently without synchronization or memory coherency.

Add Cores to Suit Customer



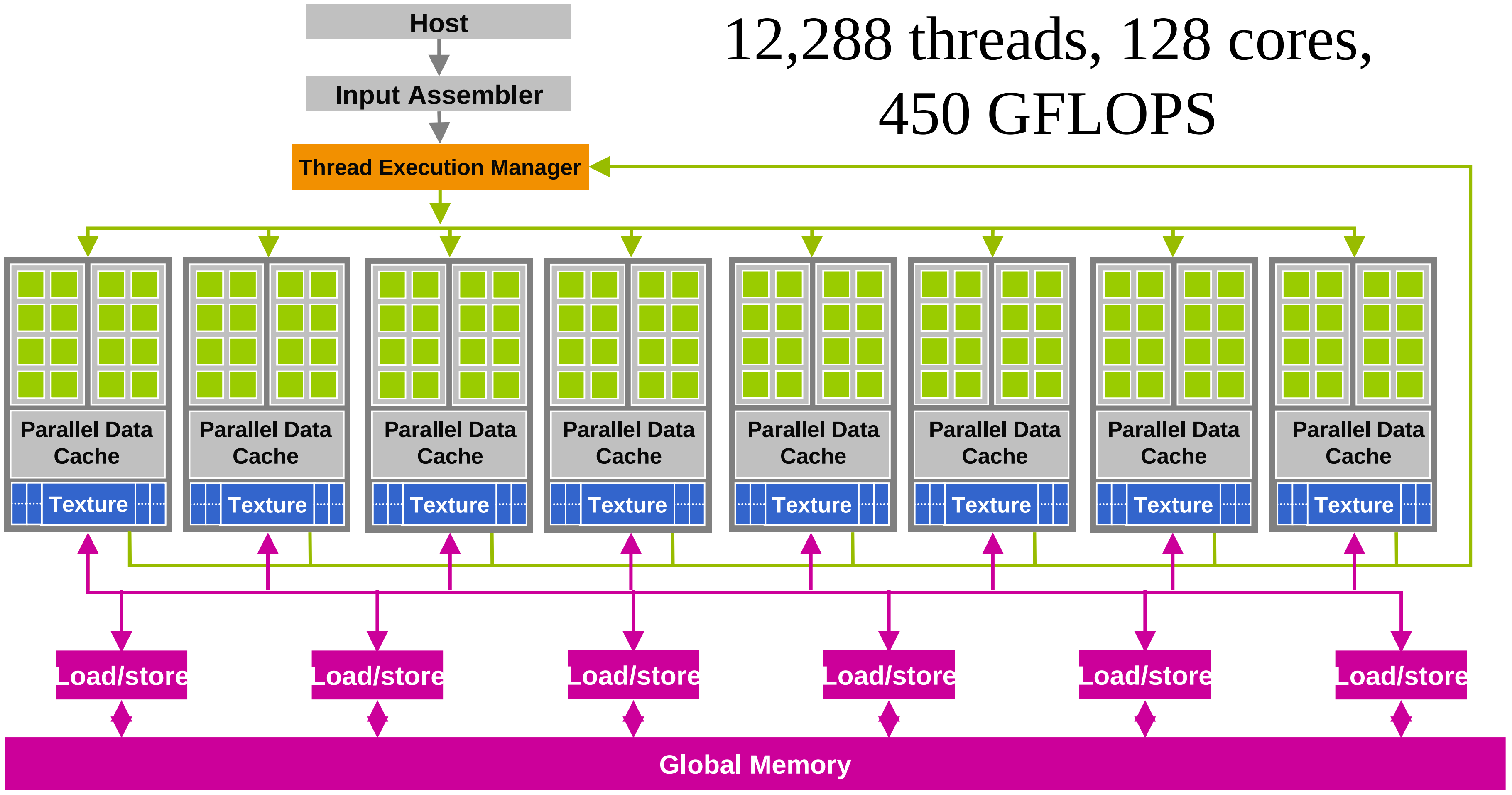
Kernel is invoked on a "grid" of uniform blocks.

Blocks are dynamically assigned to available multiprocessors and run to completion.

Synchronization occurs when all blocks complete.

GeForce 8800 General Computing

12,288 threads, 128 cores, 450 GFLOPS



768 MB DRAM, 4GB/S bandwidth to CPU

Support Fine-Grained Parallelism

- Threads are cheap but desperately needed.
  - How many can *you* give?
  - 512 threads will keep all 128 FPUs busy.
  - 1024 threads will hide some memory latency.
  - 12,288 threads can run simultaneously.
  - Up to 2×10^{12} threads per kernel invocation.

NAMD Parallel Design

Kale *et al.*, *J. Comp. Phys.* **151**:283-312, 1999.

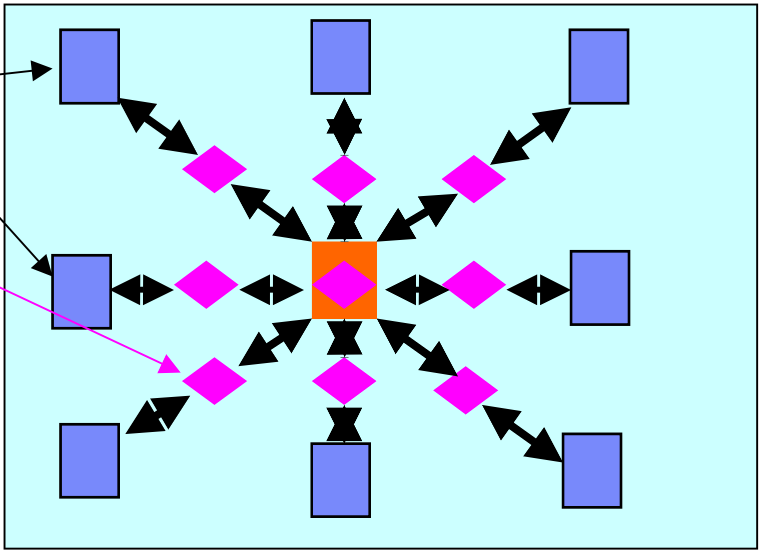
- Designed from the beginning as a parallel program
- Uses the Charm++ idea:
  - Decompose the computation into a large number of objects
  - Have an Intelligent Run-time system (of Charm++) assign objects to processors for dynamic load balancing with minimal communication

Hybrid of spatial and force decomposition:

•Spatial decomposition of atoms into cubes (called patches)

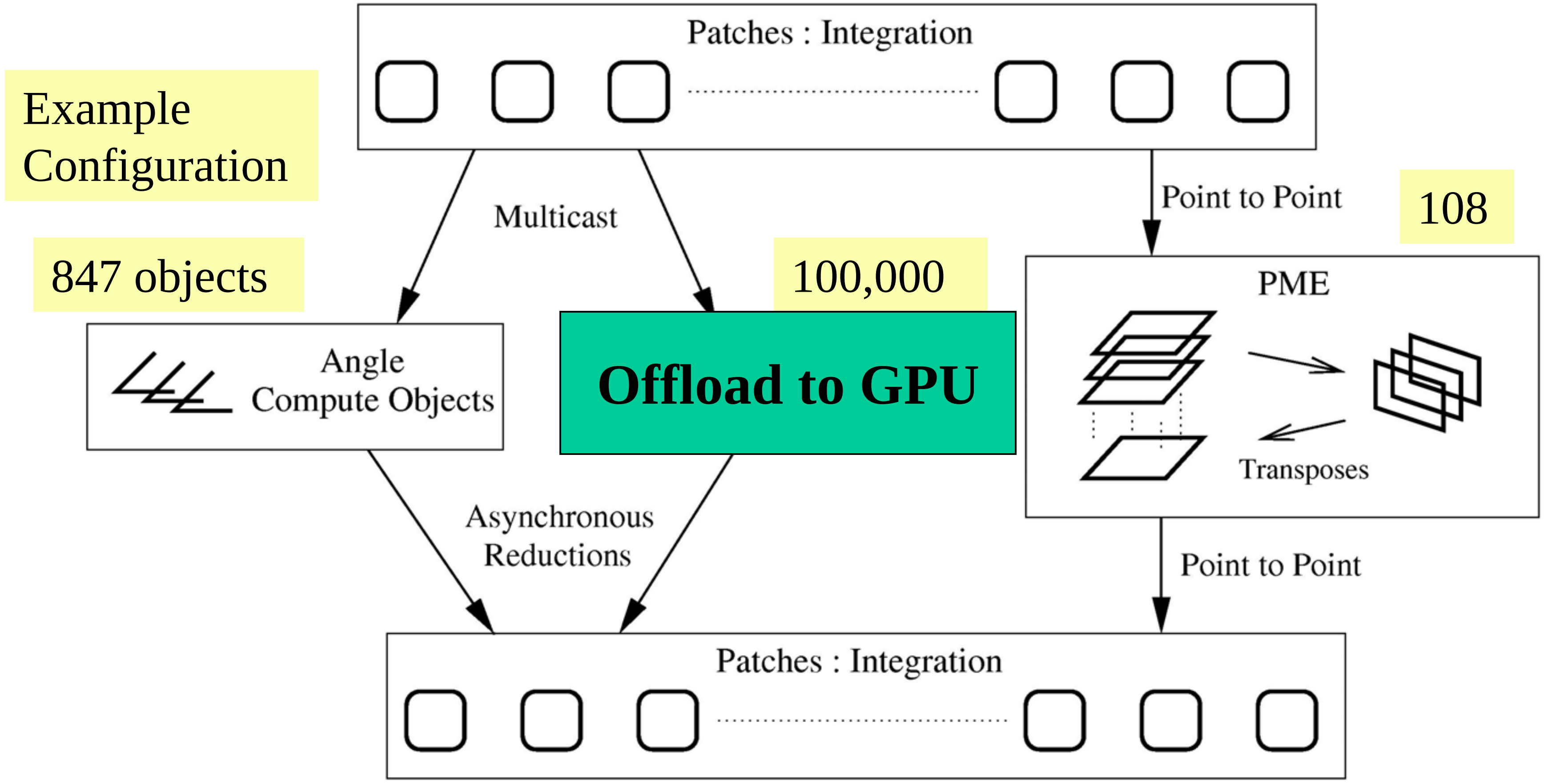
•For every pair of interacting patches, create one object for calculating electrostatic interactions

•Recent: Blue Matter, Desmond, etc. use this idea in some form



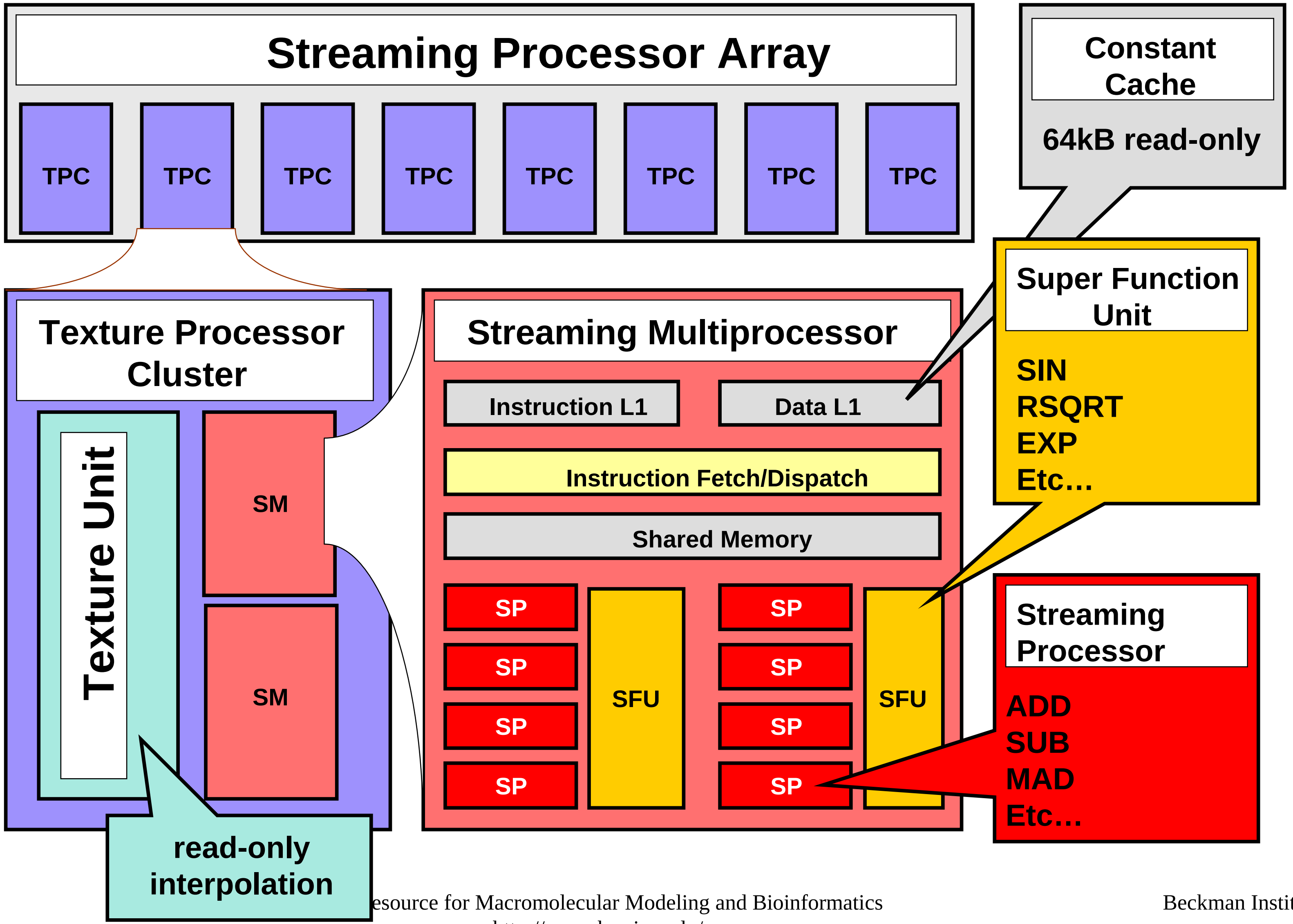
NAMD Overlapping Execution

Phillips *et al.*, *SC2002*.



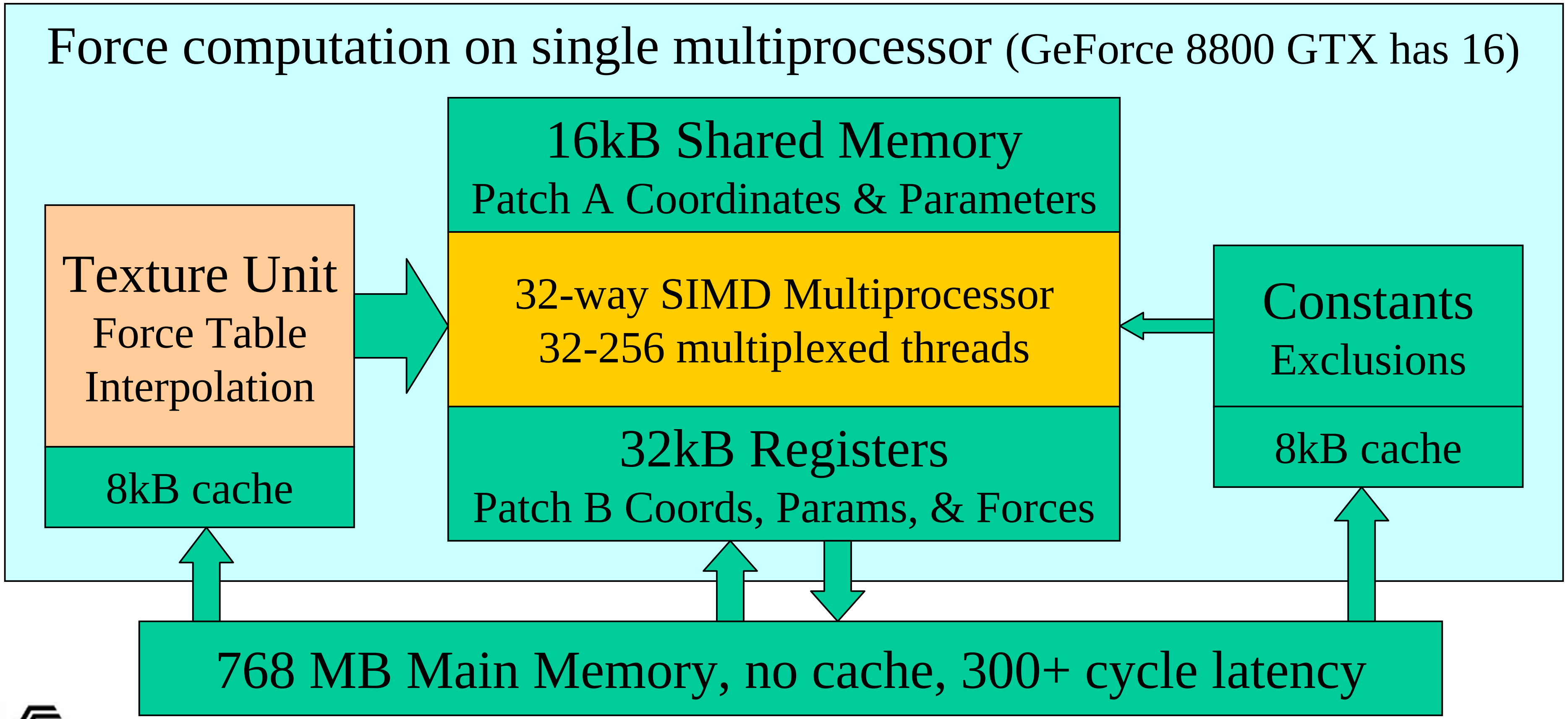
Objects are assigned to processors and queued as data arrives.

GPU Hardware Special Features



Nonbonded Forces on CUDA GPU

- Start with most expensive calculation: direct nonbonded interactions.
- Decompose work into pairs of patches, identical to NAMD structure.
- GPU hardware assigns patch-pairs to multiprocessors dynamically.


Force computation on single multiprocessor (GeForce 8800 GTX has 16)
16kB Shared Memory
Patch A Coordinates & Parameters
Texture Unit
Force Table
Interpolation
8kB cache
32-way SIMD Multiprocessor
32-256 multiplexed threads
Constants
Exclusions
8kB cache
32kB Registers
Patch B Coords, Params, & Forces
768 MB Main Memory, no cache, 300+ cycle latency

Stone *et al.*, *J. Comp. Chem.* **28**:2618-2640, 2007.

Nonbonded Forces CUDA Code

```
texture<float4> force_table;
__constant__ unsigned int exclusions[];
__shared__ atom jatom[];
atom iatom;      // per-thread atom, stored in registers
float4 iforce;   // per-thread force, stored in registers
for ( int j = 0; j < jatom_count; ++j ) {
  float dx = jatom[j].x - iatom.x;   float dy = jatom[j].y - iatom.y;  float dz = jatom[j].z - iatom.z;
  float r2 = dx*dx + dy*dy + dz*dz;
  if ( r2 < cutoff2 ) {
    float4 ft = texfetch(force_table, 1.f/sqrt(r2));
    bool excluded = false;
    int indexdiff = iatom.index - jatom[j].index;
    if ( abs(indexdiff) <= (int) jatom[j].excl_maxdiff ) {
      indexdiff += jatom[j].excl_index;
      excluded = ((exclusions[indexdiff>>5] & (1<<(indexdiff&31))) != 0);
    }
    float f = iatom.half_sigma + jatom[j].half_sigma;  // sigma
    f *= f*f;  // sigma^3
    f *= f;  // sigma^6
    f *= ( f * ft.x + ft.y );  // sigma^12 * fi.x - sigma^6 * fi.y
    f *= iatom.sqrt_epsilon * jatom[j].sqrt_epsilon;
    float qq = iatom.charge * jatom[j].charge;
    if ( excluded ) { f = qq * ft.w; }  // PME correction
    else { f += qq * ft.z; }  // Coulomb
    iforce.x += dx * f;   iforce.y += dy * f;    iforce.z += dz * f;
    iforce.w += 1.f;  // interaction count or energy
  }
}
```

Force Interpolation

Exclusions

Parameters

Accumulation

Stone *et al.*, *J. Comp. Chem.* **28**:2618-2640, 2007.

Why Calculate Each Force Twice?

- Newton's 3rd Law of Motion: $\mathbf{F}_{ij} = \mathbf{F}_{ji}$
  - Could calculate force once and apply to both atoms.
- Floating point operations are cheap:
  - Would save at most a factor of two.
- Almost everything else hurts performance:
  - Warp divergence
  - Memory access
  - Synchronization
  - Extra registers
  - Integer logic

What About Pairlists?

- Generation works well under CUDA
  - Assign atoms to cells
  - Search neighboring cells
  - Write neighbors to lists as they are found
  - Scatter capability essential
  - 10x speedup relative to CPU
- Potential for significant performance boost
  - Eliminate 90% of distance test calculations

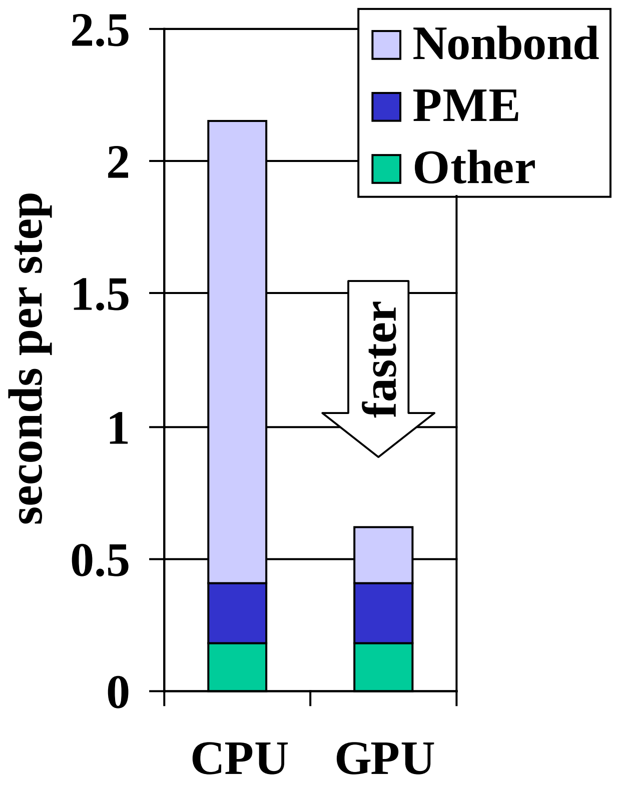
Why Not Pairlists?

- Changes FP-limited to memory limited:
  - – Limited memory to hold pairlists
  - – Limited bandwidth to load pairlists
  - – Random access to coordinates, etc.
  - – FP performance grows faster than memory
- Poor fit to NAMD parallel decomposition:
  - – Number of pairs in single object varies greatly

Initial GPU Performance

- Full NAMD, not test harness
- Useful performance boost
  - 8x speedup for nonbonded
  - 5x speedup overall w/o PME
  - 3.5x speedup overall w/ PME
  - GPU = quad-core CPU
- Plans for better performance
  - Overlap GPU and CPU work.
  - Tune or port remaining work.
    - PME, bonded, integration, etc.

ApoA1 Performance

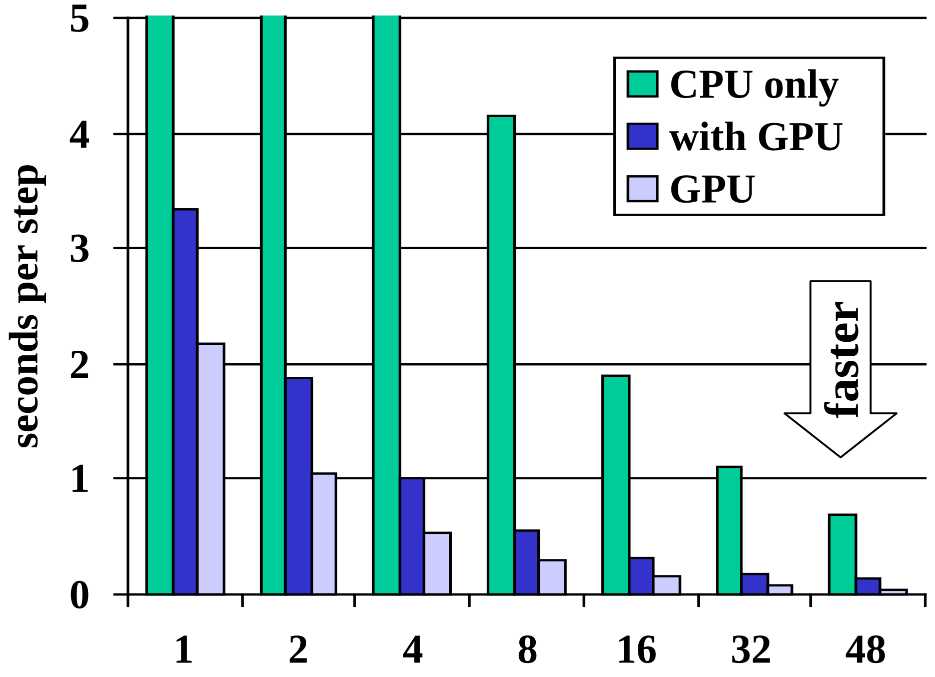


2.67 GHz Core 2 Quad Extreme + GeForce 8800 GTX

New GPU Cluster Performance

- 7x speedup
- 1M atoms for more work
- Overlap with CPU
- Infiniband helps scaling
- Load balancer still disabled
- Plans for better scaling
  - Better initial load balance
  - Balance GPU load

STMV Performance

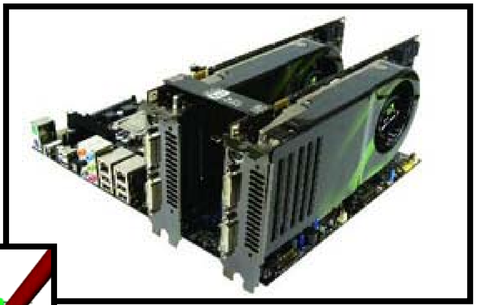


2.4 GHz Opteron + Quadro FX 5600
Thanks to NCSA and NVIDIA

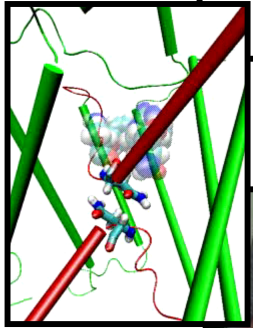
Next Goal: Interactive MD on GPU

- Definite need for faster serial IMD
  - Useful method for tweaking structures.
  - 10x performance yields 100x sensitivity.
  - Needed on-demand clusters are rare.
- AutoIMD available in VMD already
  - Isolates a small subsystem.
  - Specify molten and fixed atoms.
  - Fixed atoms reduce GPU work.
  - Pairlist-based algorithms start to win.
- Limited variety of simulations
  - Few users have multiple GPUs.
  - Move entire MD algorithm to GPU.

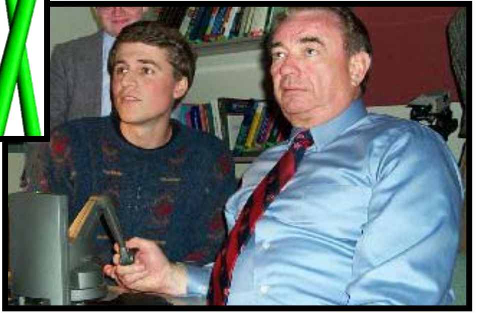
NAMD



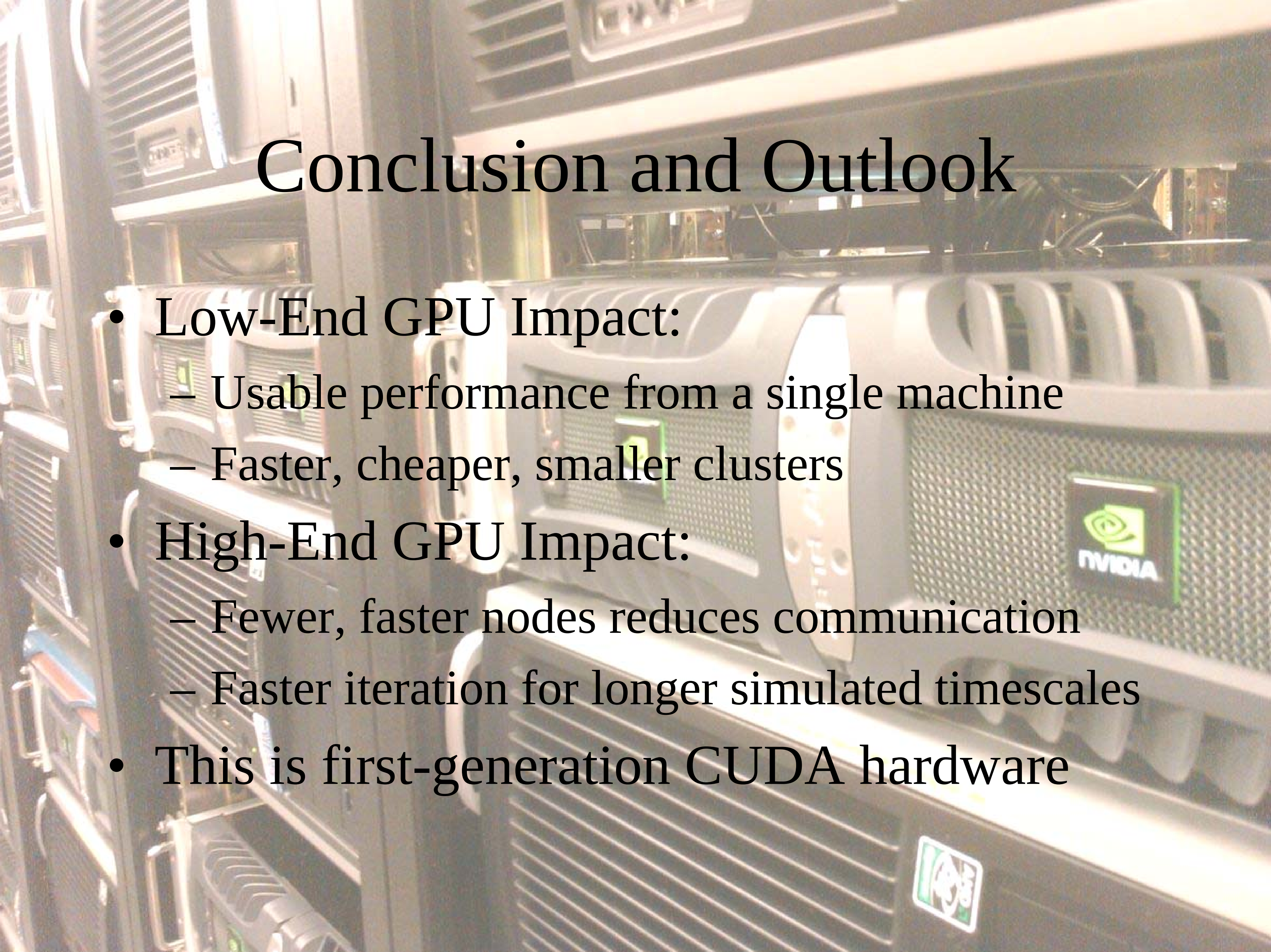
VMD



User



(Former HHS Secretary Thompson)



Conclusion and Outlook

- Low-End GPU Impact:
  - Usable performance from a single machine
  - Faster, cheaper, smaller clusters
- High-End GPU Impact:
  - Fewer, faster nodes reduces communication
  - Faster iteration for longer simulated timescales
- This is first-generation CUDA hardware