



**S05: High Performance Computing with CUDA**

**Case Study:  
Molecular Dynamics**

**James Phillips**

**NIH Resource for Macromolecular Modeling and Bioinformatics**

**Outline**



- **Introduction to molecular dynamics**
- **NAMD capabilities and design**
- **Acceleration hardware options**
- **Mapping NAMD to CUDA**
- **Algorithm and code examples**
  - **Loading atoms to registers and shared memory**
  - **Force interpolation with the texture unit**
  - **Fitting exclusions into constant cache**
  - **Inner loop walkthrough**
  - **Failed optimization attempts**
- **Performance results**
- **Take-home lessons**

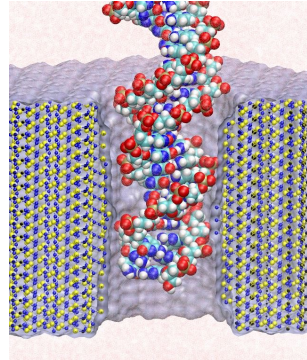
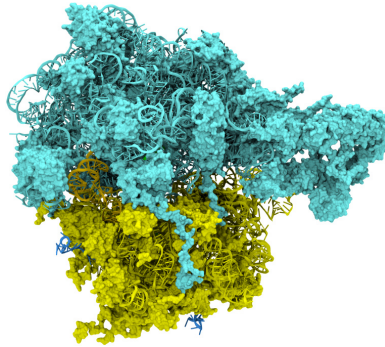
# The Computational Microscope



- Study the molecular machines in living cells

Ribosome: synthesizes proteins from genetic information, target for antibiotics

Silicon nanopore: bionanodevice for sequencing DNA efficiently



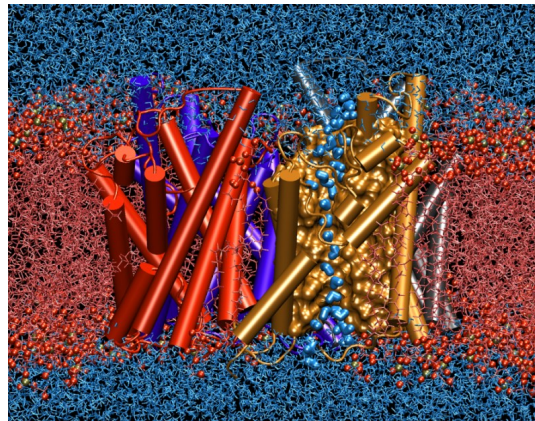
S05: High Performance Computing with CUDA



# How Does the Computational Microscope Work?



- Obtain atomic structure from the Protein Data Bank
- Simulate structure in its native biological environment:
  - Membrane
  - Water
  - Ions
- Display and analyze the prepared system



S05: High Performance Computing with CUDA



## Molecular Mechanics Force Field



$$\begin{aligned}
 U(\vec{R}) = & \underbrace{\sum_{\text{bonds}} k_i^{\text{bond}} (r_i - r_0)^2}_{U_{\text{bond}}} + \underbrace{\sum_{\text{angles}} k_i^{\text{angle}} (\theta_i - \theta_0)^2}_{U_{\text{angle}}} + \\
 & \underbrace{\sum_{\text{dihedrals}} k_i^{\text{dih}} [1 + \cos(n_i \phi_i + \delta_i)]}_{U_{\text{dihedral}}} + \\
 & \underbrace{\sum_i \sum_{j \neq i} 4\epsilon_{ij} \left[ \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left( \frac{\sigma_{ij}}{r_{ij}} \right)^6 \right]}_{U_{\text{nonbond}}} + \sum_i \sum_{j \neq i} \frac{q_i q_j}{\epsilon r_{ij}}
 \end{aligned}$$

S05: High Performance Computing with CUDA



## Classical Molecular Dynamics



Energy function:  $U(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_N) = U(\vec{R})$

used to determine the force on each atom:

$$m_i \frac{d^2 \vec{r}_i}{dt^2} = \vec{F}_i = -\vec{\nabla} U(\vec{R})$$

Newton's equation represents a set of N second order differential equations which are solved numerically via the Verlet integrator at discrete time steps to determine the trajectory of each atom.

$$\vec{r}_i(t + \Delta t) = 2\vec{r}_i(t) - \vec{r}_i(t - \Delta t) + \frac{\Delta t^2}{m_i} \vec{F}_i(t)$$

Small terms added to control temperature and pressure if needed.

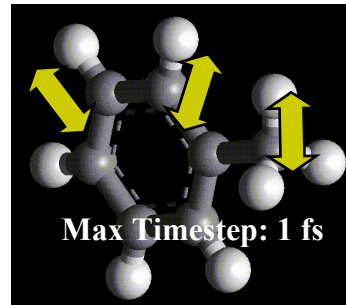
S05: High Performance Computing with CUDA



## Biomolecular Time Scales



Motion	Time Scale (sec)
Bond stretching	$10^{-14}$ to $10^{-13}$
Elastic vibrations	$10^{-12}$ to $10^{-11}$
Rotations of surface sidechains	$10^{-11}$ to $10^{-10}$
Hinge bending	$10^{-11}$ to $10^{-7}$
Rotation of buried side chains	$10^{-4}$ to 1 sec
Allosteric transistions	$10^{-5}$ to 1 sec
Local denaturations	$10^{-5}$ to 10 sec



S05: High Performance Computing with CUDA



## Typical Simulation Statistics



- 100,000 atoms (including water, lipid)
- 10-20 MB of data for entire system
- 100 Å per side periodic cell
- 12 Å cutoff of short-range nonbonded terms
- 10,000,000 timesteps (10 ns)
- 3 s/step on one processor (1 year total!)

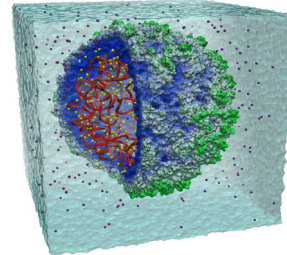
S05: High Performance Computing with CUDA



## NAMD: Scalable All-Atom MD



- CHARMM, AMBER, OPLS force fields
- Efficient PME full electrostatics
- Conjugate-gradient minimization
- Temperature and pressure controls
- Steered molecular dynamics (many methods)
- Interactive molecular dynamics (with VMD)
- Locally enhanced sampling
- Alchemical free energy perturbation
- Adaptive biasing force potential of mean force
- User-extendable in Tcl for forces and algorithms
- All features run in parallel and scale to millions of atoms!



S05: High Performance Computing with CUDA



## NAMD: Practical Supercomputing



- 20,000 users can't all be computer experts.
  - 18% are NIH-funded; many in other countries.
  - 4200 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.
  - Automatically split patches and enable pencil PME.
  - Precompiled binaries available when possible.
- Desktops and laptops – setup and testing
  - x86 and x86-64 Windows, PowerPC and x86 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



S05: High Performance Computing with CUDA



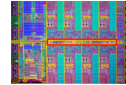
## Hardware Acceleration for NAMD



*Can NAMD offload work to a special-purpose processor?*

### ● Resource studied all the options 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



S05: High Performance Computing with CUDA

11

## 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).
- **We did have some advantages:**
  - Experience from VMD development
  - David Kirk (Chief Scientist, NVIDIA)
  - Wen-mei Hwu (ECE Professor, UIUC)



Fun to program (and drive)



S05: High Performance Computing with CUDA

12

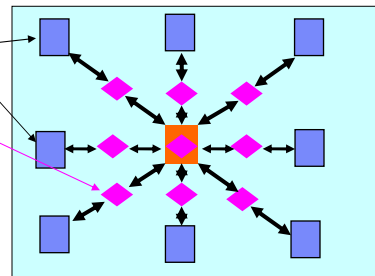
# NAMD Design



- 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

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 Parallelization using Charm++

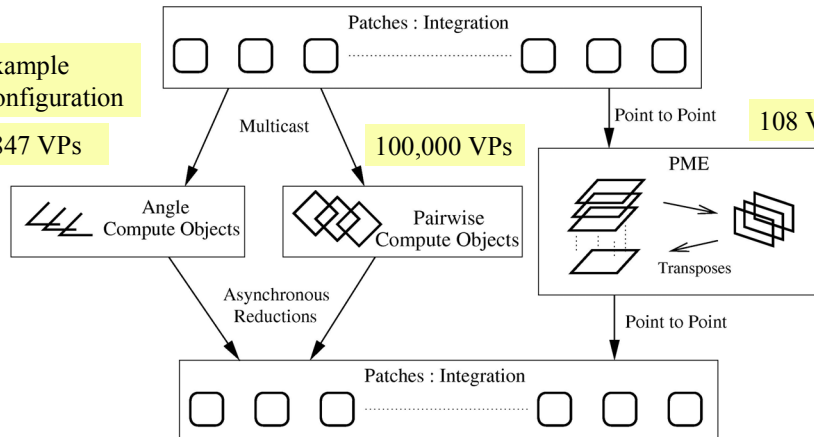


Example Configuration

847 VPs

100,000 VPs

108 VPs



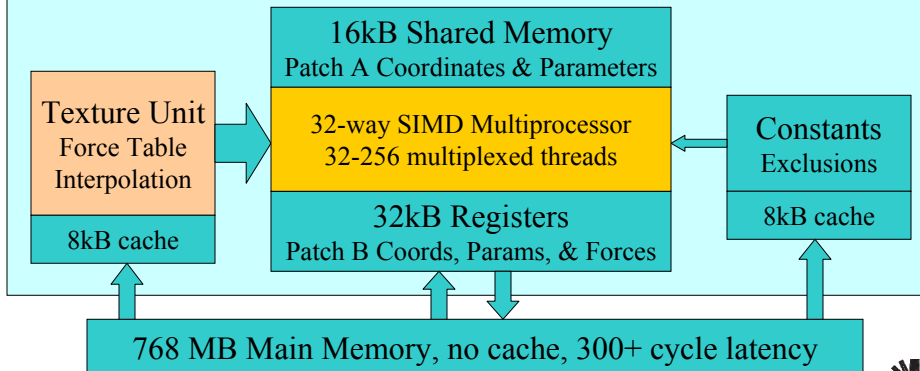
These 100,000 Objects (virtual processors, or VPs) are assigned to real processors by the Charm++ runtime system

## 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)



S05: High Performance Computing with CUDA

15



## Each Block Gets a Pair of Patches



- Block-level constants in shared memory to save registers.
- patch\_pair array is 16-byte aligned.
- To coalesce read have each thread load one int from global memory and write it into a union in shared memory.

```
#define myPatchPair pp.pp
__shared__ union { patch_pair pp; unsigned int i[8]; } pp;
__shared__ bool same_patch;
__shared__ bool self_force;

if ( threadIdx.x < (sizeof(patch_pair)>>2) ) {
    unsigned int tmp = ((unsigned int*)patch_pairs[
        (sizeof(patch_pair)>>2)*blockIdx.x+threadIdx.x]);
    pp.i[threadIdx.x] = tmp;
}
__syncthreads();
// now all threads can access myPatchPair safely
```

S05: High Performance Computing with CUDA

16



## Loading Atoms Is Not Trivial



- **Want to copy two 16-byte structs per thread from global to shared memory.**
- **Global memory access should be aligned on 16-byte boundaries to be coalesced.**
- **16-byte structs in shared memory cause bank conflicts, 36-byte structs do not.**



## Right-Sized Atom Data Structures



```
struct __align__(16) atom { // must be multiple of 16!
    float3 position;
    float charge;
};

struct __align__(16) atom_param { // must be multiple of 16!
    float sqrt_epsilon;
    float half_sigma;
    unsigned int index;
    unsigned short excl_index;
    unsigned short excl_maxdiff;
};

struct shared_atom { // do not align, size 36 to avoid bank conflicts
    float3 position;
    float charge;
    float sqrt_epsilon;
    float half_sigma;
    unsigned int index;
    unsigned int excl_index;
    unsigned int excl_maxdiff;
};
```



## More Problems Loading Atoms



- Global access to mixed-type atom\_param struct won't coalesce! (Only built-in vector types will.)
- Fix it by casting global atom\_param\* to uint4\*.
- Can't take pointer to struct in registers, so copy integers to shared memory.
- Use alias of shared\_atom and uint arrays to finally read patch B into usable struct in registers.
- Use same trick to load patch A, but this time leave the data in shared memory.



## Hack to Coalesce atom\_params



```
extern __shared__ shared_atom jas[]; // atom jas[max_atoms_per_patch]
extern __shared__ unsigned int sh_uint[]; // aliased to jas[]
atom ipq;
atom_param iap;

if ( threadIdx.x < myPatchPair.patch1_size ) {
    int i = myPatchPair.patch1_atom_start + threadIdx.x;
    uint4 tmpa = ((uint4*)atoms)[i]; // coalesced reads from global memory
    uint4 tmpap = ((uint4*)atom_params)[i];
    i = 9*threadIdx.x;
    sh_uint[i] = tmpa.x; // copy to aliased ints in shared memory
    sh_uint[i+1] = tmpa.y;
    sh_uint[i+2] = tmpa.z;
    sh_uint[i+3] = tmpa.w;
    sh_uint[i+4] = tmpap.x;
    sh_uint[i+5] = tmpap.y;
    sh_uint[i+6] = tmpap.z;
    sh_uint[i+7] = ((tmpap.w << 16) >> 16); // split two shorts into shared_atom ints
    sh_uint[i+8] = (tmpap.w >> 16);
    COPY_ATOM(ipq, jas[threadIdx.x]) // macros to copy structs element by element
    COPY_PARAM(iap, jas[threadIdx.x])
}
```



## CPU Force Interpolation



- **Want to avoid calculating erfc(), sqrt(), branches for switching functions.**
- $U(r^2) = \varepsilon(\sigma^{12}A(r^2) + \sigma^6B(r^2)) + qqC(r^2)$
- $F = -2 r U'(r^2)$
- **Piecewise cubic interpolation of A,B,C.**
- **Need more windows at small  $r^2$ , so use exponent and high-order mantissa bits in floating point format to determine window.**



## Texture Unit Force Interpolation



- **Bit manipulation of floats is not possible.**
- **But rsqrt() is implemented in hardware.**
- $F(r^{-1})/r = \varepsilon(\sigma^{12}A(r^{-1}) + \sigma^6B(r^{-1})) + qqC(r^{-1})$
- $F = r F(r^{-1})/r$
- **Piecewise linear interpolation of A,B,C.**
  - $F(r)$  is linear since  $r(a r^{-1} + b) = a + r b$
- **Texture unit hardware is a perfect match.**



## Const Memory Exclusion Tables



- **Need to exclude bonded pairs of atoms.**
  - Also apply correction for PME electrostatics.
- **Exclusions determined by using atom indices to bit flags in exclusion arrays.**
- **Repetitive molecular structures limit unique exclusion arrays.**
- **All exclusion data fits in constant cache.**



## Overview of Inner Loop



- **Calculate forces on atoms in registers due to atoms in shared memory.**
  - Ignore Newton's 3<sup>rd</sup> law (reciprocal forces).
  - Do not sum forces for atoms in shared memory.
- **All threads access the same shared memory atom, allowing shared memory broadcast.**
- **Only calculate forces for atoms within cutoff distance (roughly 10% of pairs).**



```

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
    }
}

```

## Inner Loop



## Writing Forces Is Trivial



- Forces stored in float4, easily coalesced.
- Each block writes to separate output arrays.
- A separate grid sums forces for each patch.



## Swap Shared and Register Atoms



- Need to calculate forces on atoms in shared memory too.
- Declaring a temporary structure increases register count.

```
#define SWAP(TYPE,REG,FIELD) { \
    TYPE tmp = jas[threadIdx.x].FIELD; \
    jas[threadIdx.x].FIELD = REG.FIELD; \
    REG.FIELD = tmp; \
}
```

```
SWAP(float,ipq,position.x);
SWAP(float,ipq,position.y);
SWAP(float,ipq,position.z);
SWAP(float,ipq,charge);
SWAP(float,iap,sqrt_epsilon);
SWAP(float,iap,half_sigma);
SWAP(unsigned int,iap,index)
SWAP(unsigned int,iap,excl_index)
SWAP(unsigned int,iap,excl_maxdiff);
```



## What About Warp Divergence?



- Almost all exclusion checks fail, and the extra work for an excluded pair is minimal.
- Cutoff test isn't completely random.
  - Hydrogens follow their heavy atoms.
  - Atoms in far corners of patches have few neighbors within cutoff distance.
- If cutoff test is removed (calculate all pairs in neighboring patches), total GFLOPS is 10x higher, but the simulation runs slower.



## What About Reciprocal Forces?



- **This was attempted:**
  - On iteration  $j$  thread  $i$  calculates interaction with shared memory atom  $(j+i) \bmod n$ .
  - Don't use mod operator (it's very slow).
  - Syncthreads required between iterations.
- **Performance was worse due to:**
  - Extra cycles needed for control flow calculations.
  - Pipeline stalls for thread synchronization.
  - Increased warp divergence and idle threads.



## What About Pair Lists?



- **Standard method for CPU-based MD:**
  - Atoms move slowly, so build a list of atoms that might move within the cutoff distance and only check those atoms for several steps.
- **Not as useful for CUDA:**
  - Lists are twice as fast (not counting generation time).
  - Random access pattern requires atom data to be loaded through the texture unit.
    - More atoms are required than the texture cache can hold.
  - List is large, limiting size of simulated system.
  - Not known how to generate efficiently on GPU.



## Register Pressure Is Severe



- **Number of threads is limited by registers.**
  - To accommodate larger patches, each thread loads two atoms to registers and shared.
  - Patches can also be subdivided in NAMD.
- **Blocks are not co-scheduled.**
  - Smaller patches would reduce threads per block, but increase total global memory access.

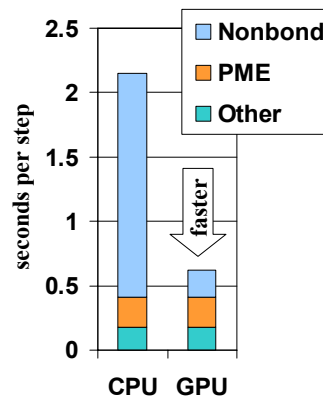


## Initial GPU Performance



- **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



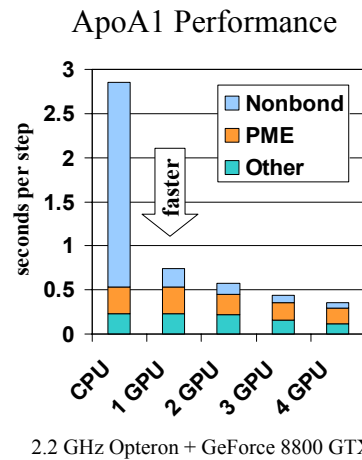
2.67 GHz Core 2 Quad Extreme + GeForce 8800 GTX



## Initial GPU Cluster Performance



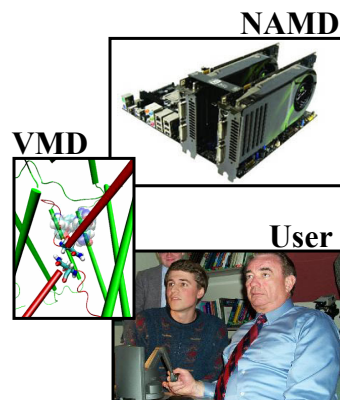
- Poor scaling unsurprising
  - 2x speedup on 4 GPUs
  - Gigabit ethernet
  - Load balancer disabled
- Plans for better scaling
  - InfiniBand network
  - Tune parallel overhead
  - Load balancer changes
    - Balance GPU load.
    - Minimize communication.



## 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.



(Former HHS Secretary Thompson)



## Take-Home Lessons on CUDA



- **Efficient data movement is not automatic.**
  - Look at load instructions in the ptx file!
  - Align global memory access for coalescing.
  - Align shared memory access to avoid bank conflicts.
  - Balance registers and shared memory for maximum occupancy.
- **Use special hardware when possible.**
  - Functions like rsqrt() overlap with floating point.
  - Constant cache useful for rarely-accessed or broadcast data.
  - Texture unit for cached random-access data.
- **Clever algorithms may not help.**
  - There are worse things than warp divergence.
  - Logic steals cycles you could use for floating point.
  - Don't do \_\_syncthreads() in the inner loop.

## Questions?



- **Additional Information and References:**
  - <http://www.ks.uiuc.edu/Research/gpu/>
  - <http://www.ks.uiuc.edu/Research/namd/>
- **Questions, source code requests:**
  - James Phillips [jim@ks.uiuc.edu](mailto:jim@ks.uiuc.edu)
- **Acknowledgements:**
  - Prof. Wen-mei Hwu (UIUC)
  - NVIDIA
  - NIH support: P41-RR05969