GPU-Based Molecular Dynamic Simulations Optimized with CUDA Data Parallel Primitives (CUDPP) and CURAND Libraries

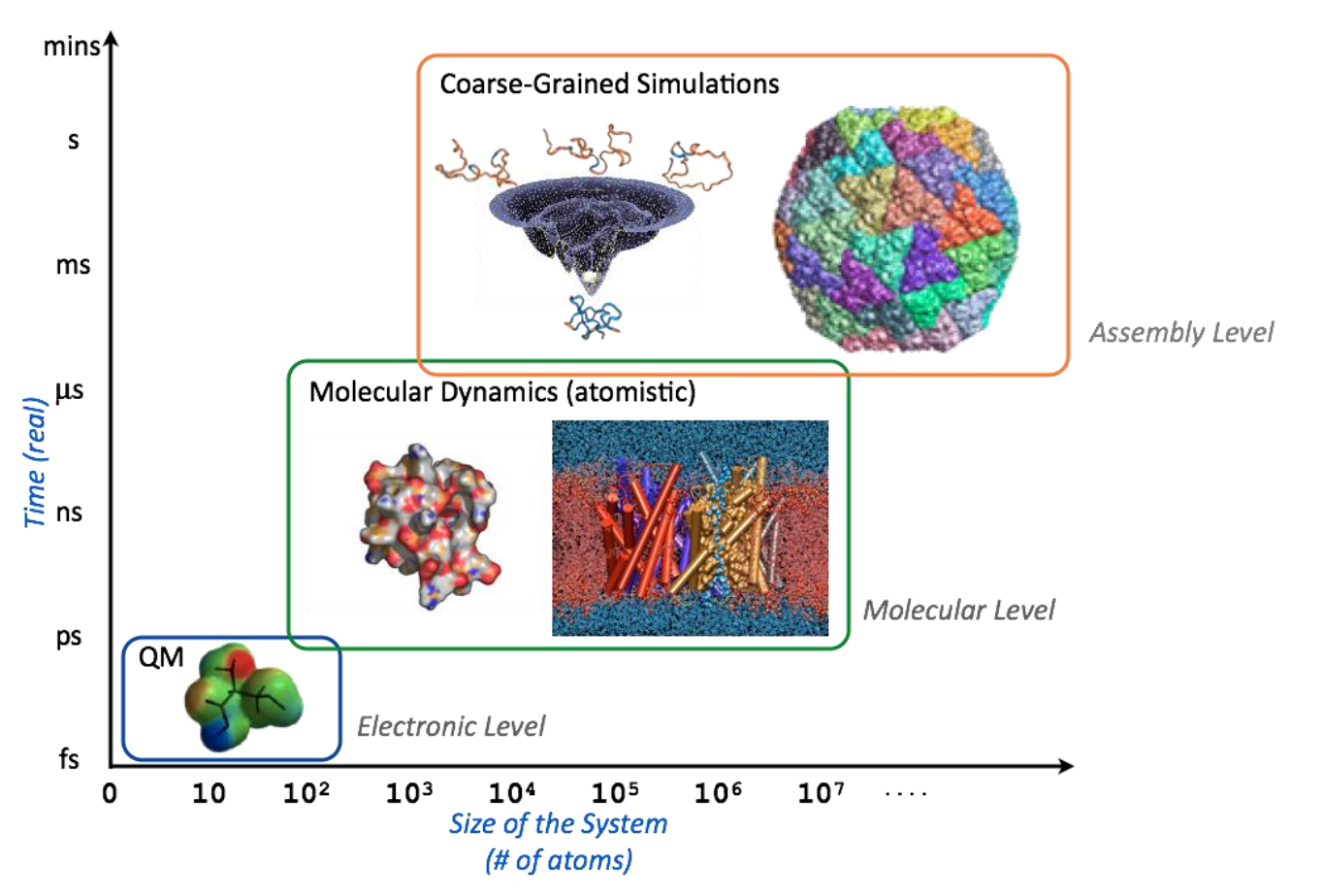
Tyson J. Lipscomb[1] and Samuel S. Cho[1,2]
Wake Forest University
Departments of Computer Science[1] and Physics[2]

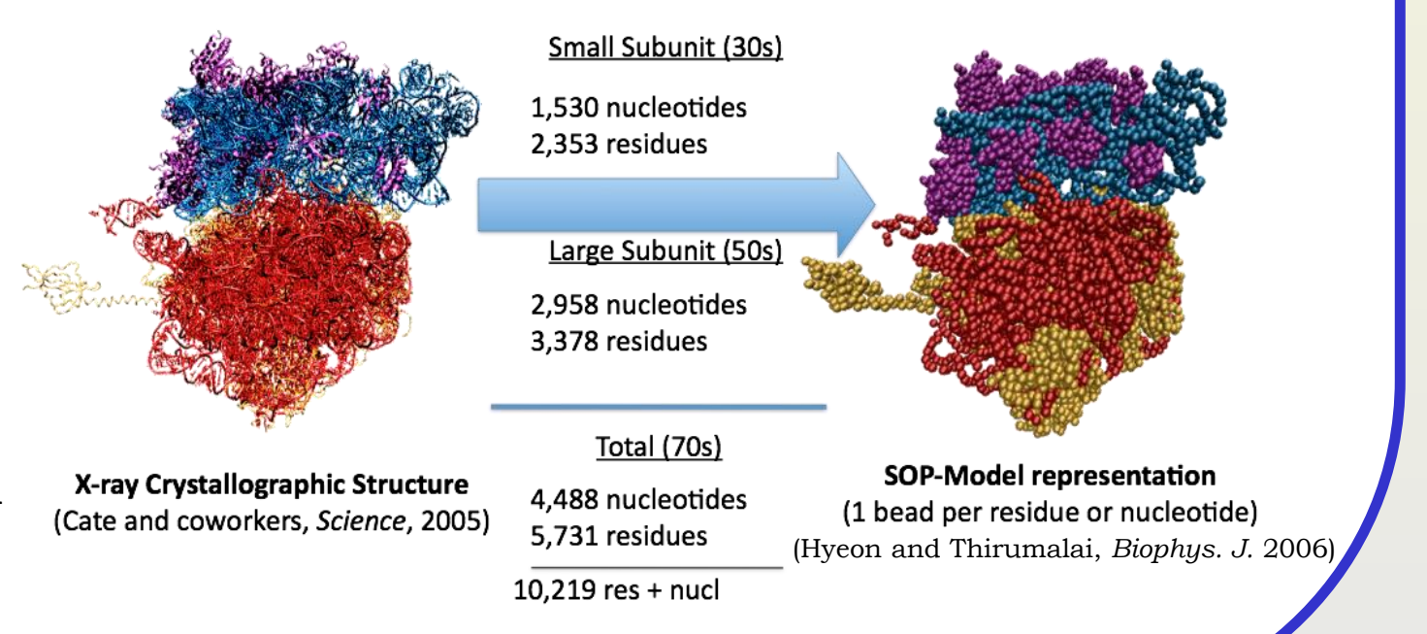
Introduction to Molecular Dynamics Simulations

Biomolecular Simulations are "Molecular Microscopes"
• Lowest energy structure results in cellular functions.
• Simulations help us understand how biomolecules assemble.

Resolutions of Simulations
• There are many classes of simulations ranging from a detailed representation to a coarse representation.
• The more detailed the representation of the biomolecule(s) is, the smaller the timescale of the simulation.

Ribosome:
• A molecular machine whose function is to synthesize proteins (Nobel Prize, 2009)
• Composed of protein and RNA molecules.





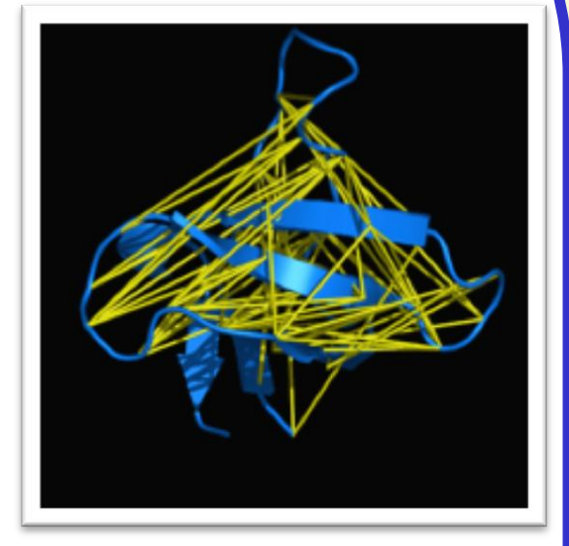
X-ray Crystallographic Structure (Cate and coworkers, *Science*, 2005) — **SOP-Model representation** (1 bead per residue or nucleotide) (Hyeon and Thirumalai, *Biophys. J.* 2006)

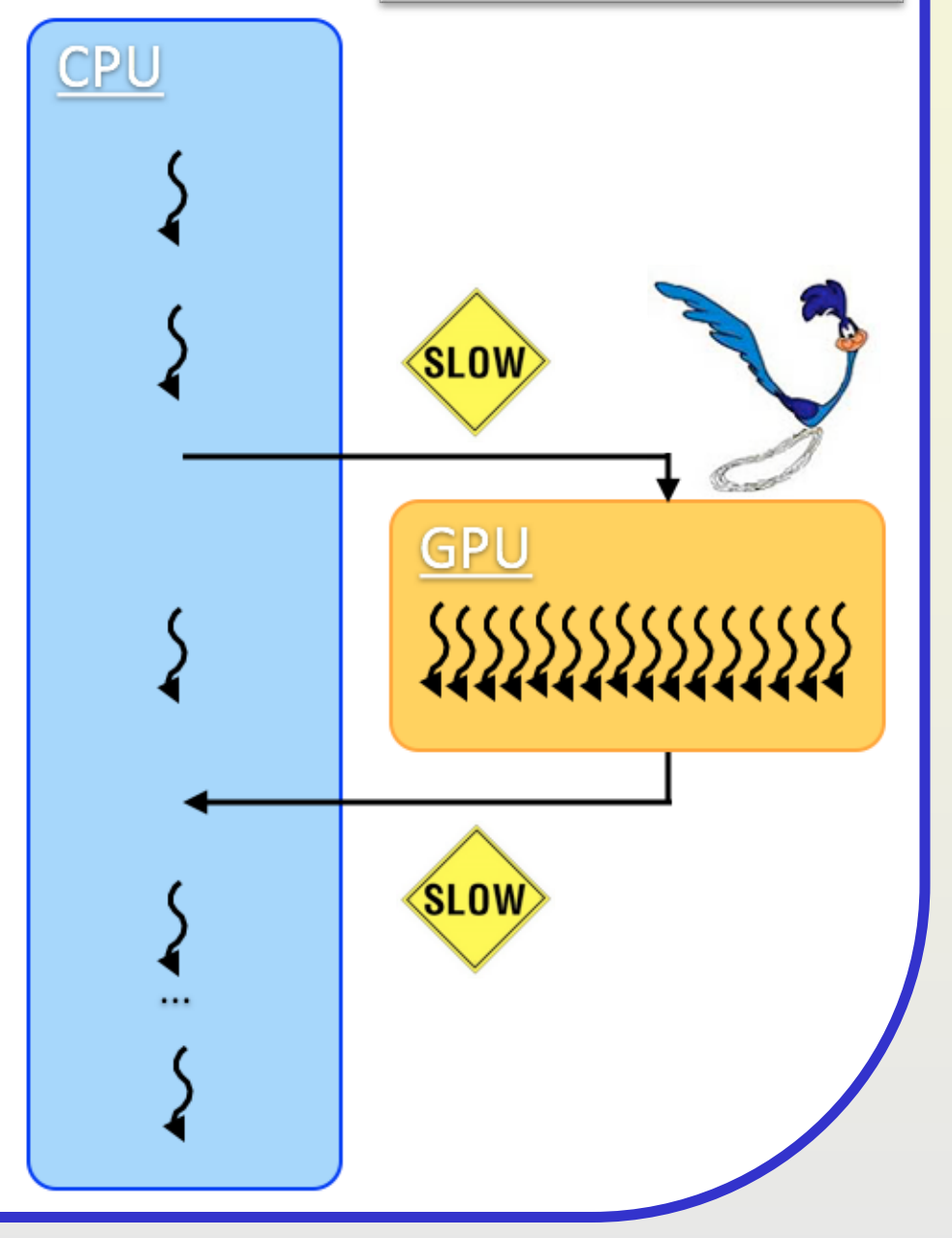
Computational Challenges

MD Simulation Bottleneck
• The average protein is ~400 residues in length. Note: ribosome consists of 10,000+ residues/nucleotides.
• Even in coarse-grained simulations, each residue (represented by a bead) interacts with each other.
• At each timestep, the forces acting on each of the beads must be calculated, which is a $O(N^2)$ calculation that is typically computed sequentially in traditional CPUs.

Algorithm is Highly Parallelizable
• Each bead's position, velocity, and force are independently calculated at each timestep.
• Ideal for parallel computation, e.g., on a GPU architecture.

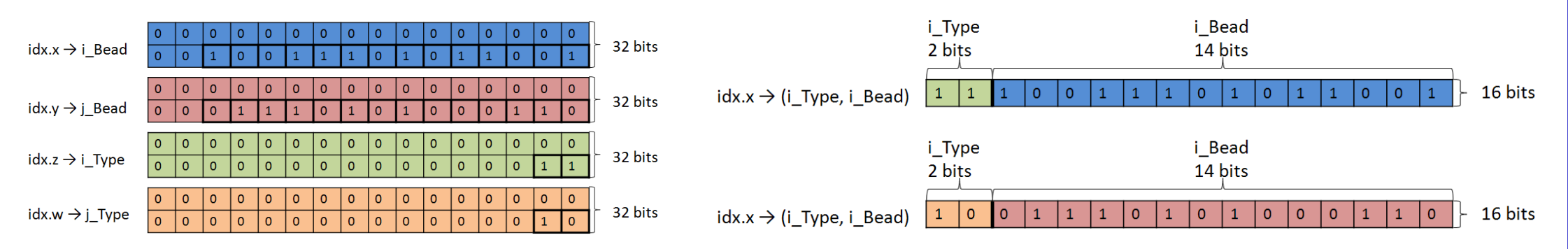
Solution: Assign each interaction to its own individual thread.



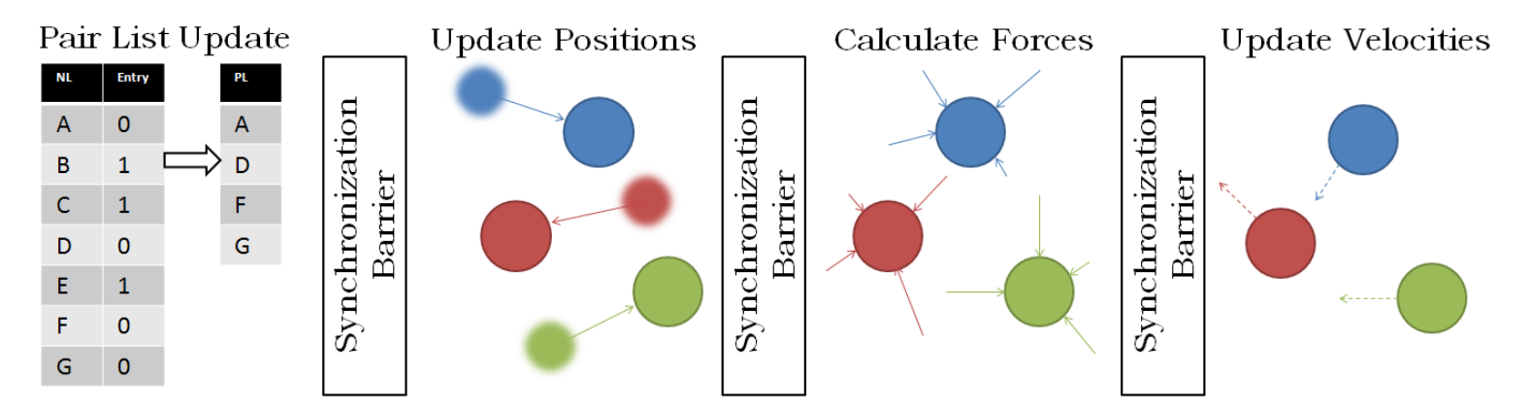


GPU Issues and Limitations

Memory Footprint
• Larger simulations can require multiple gigabytes of storage.
• Most commercially available GPUs have very limited RAM (~3-6 GB)
→ <u>Strategy 1</u>: Move all relevant data transferred to GPU at beginning of program execution an minimize data transfer back to CPU.
→ <u>Strategy 2</u>: Use smallest data types possible; Reduces amount of data transfer form device's global to local memory locations by ~75%.



Barrier Synchronization
• Although each individual force calculation is independent, there are multiple forces acting on a single bead; The assignment of the forces to each bead requires many *dependent, ordered* computations.
• Must launch new kernel for each portion of computation, resulting in the introduction of additional overhead.
• *<u>Difficult to fully utilize GPU.</u>*



GPGPU Libraries Provide Efficient, Reliable Code

CUDPP
(http://code.google.com/p/thrust/)
- Library of GPU-based implementation of many commonly used parallel algorithms.
- Used for parallel scan and key-value sorting.

Thrust
(http://code.google.com/p/thrust/)
- High-level, C++ style interface.
- Provides functionality comparable to CUDPP.

Nadathur Satish, Mark Harris, and Michael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs". In Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, May 2009.

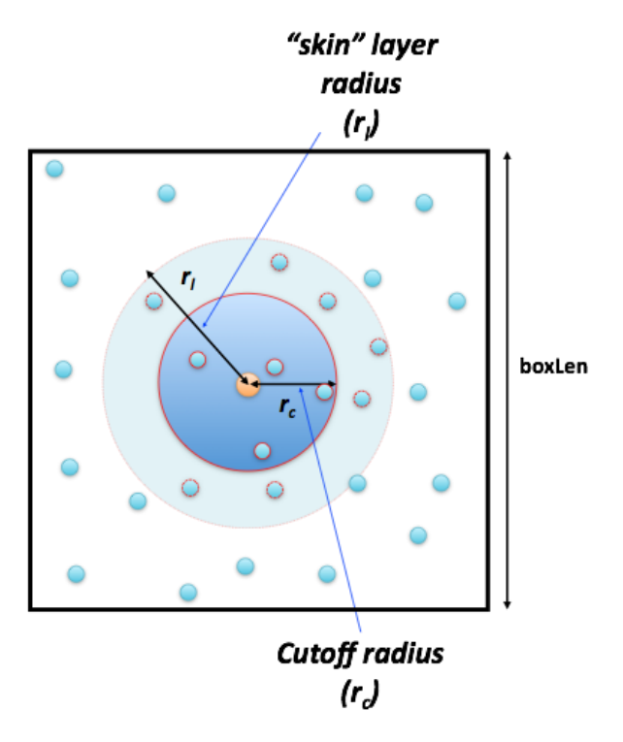
CURAND
- CUDA random number generation library.
- Provides fast random number generation.
- Very low memory demands
  - Only 40 bytes per random generator.
  - Other methods require generating and storing arrays of random numbers.
- Used to generate values for random forces for each bead during every timestep.

Original Neighbor List Algorithm is Unparallelizable

Definition and Background

• **Two major classes of interactions to calculate:**
  – Bonded: bonds, angles, dihedrals
    • $O(N)$ calculation
  – Nonbonded: Lennard-Jones and Electrostatic
    • $O(N^2)$ calculation
    • > 90% of computations in typical MD simulation are of nonbonded interactions.

• **Neighbor List Algorithm:** For each bead, keep track of close beads and evaluate those interactions only.
  • Neighbor List – $r_{ij} < r_l$ (out of all possible pairs)
  • Pair List – $r_{ij} < r_c$ (subset of Neighbor List only)
  • With cutoffs, the computation becomes $O(Nr_c^3) \sim O(N)$



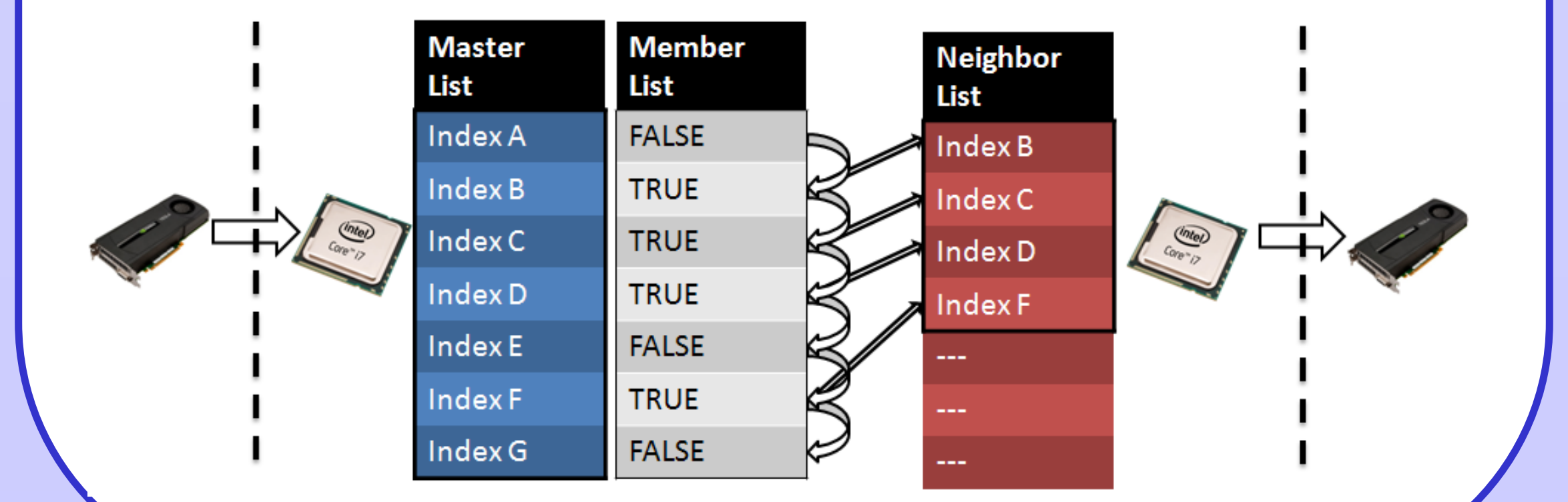
Original Algorithm

```
num_NL = 0;
for(i = 0; i < num_beads; i++)
        if(member_array[i] = TRUE)
                NL[num_NL] = ML[i];
                num_NL++;
```

Position in Neighbor List dependent on number already in list

Number of beads in Neighbor List may change during any iteration

Problems for Parallelization
- Each iteration is dependent upon the results of previous iterations.
- Threads would be dependent upon each other.
- Cannot parallelize!



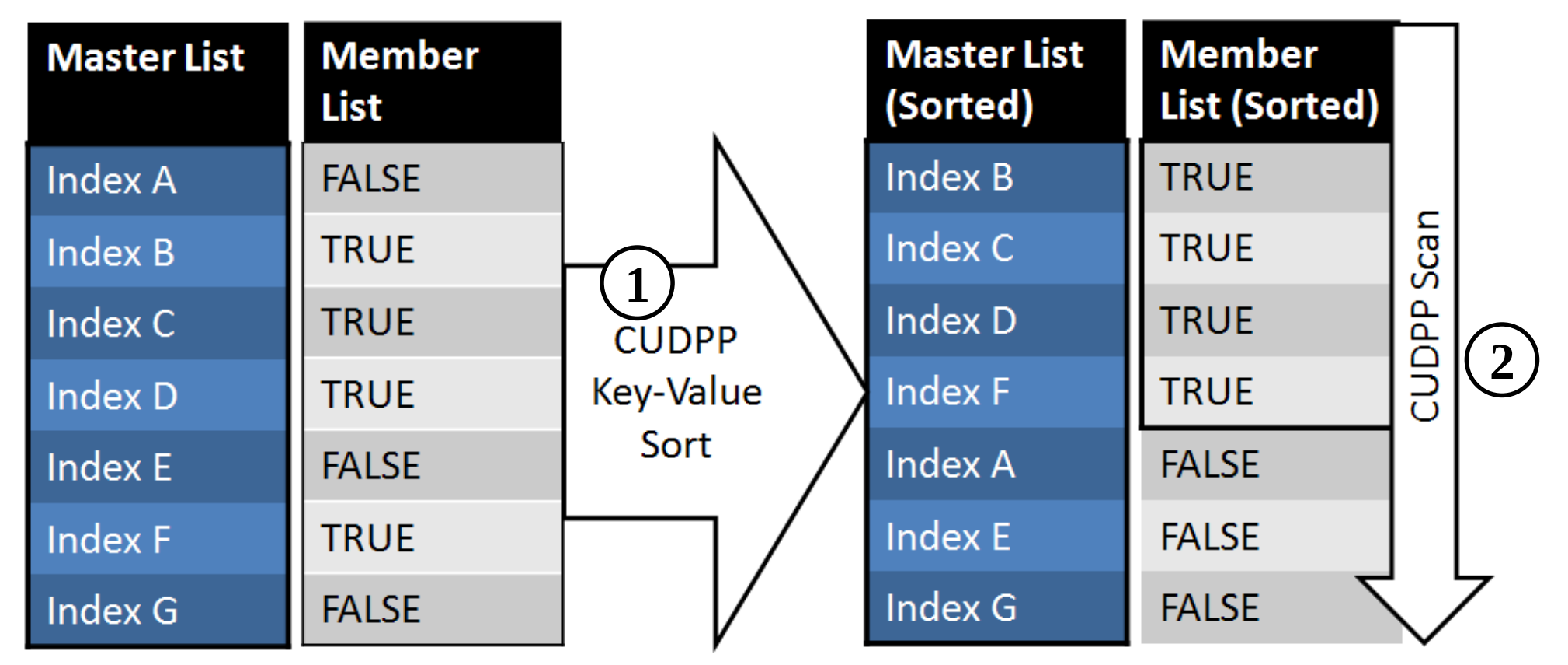
Parallelization of Neighbor List Sorting Optimizes GPU Utilization

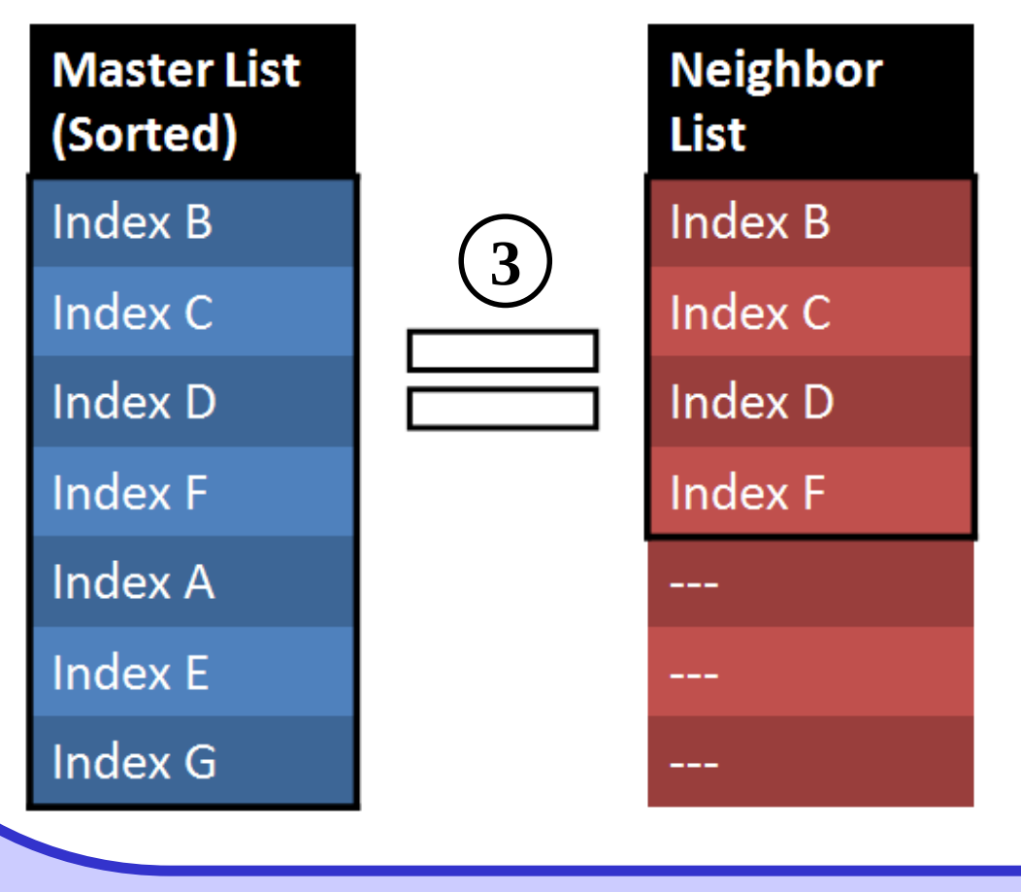
Parallel Algorithm

Step 1: Perform key-value sort on GPU using CUDPP library.
• Member List as keys and Master List as values.
• Groups members of Neighbor List together with others.
• Keys are binary flags, so a 1-bit sort suffices.

Step 2: Perform parallel scan using CUDPP.
• Counts the total number of TRUE values in Member List, determining how many entries are in Neighbor List.

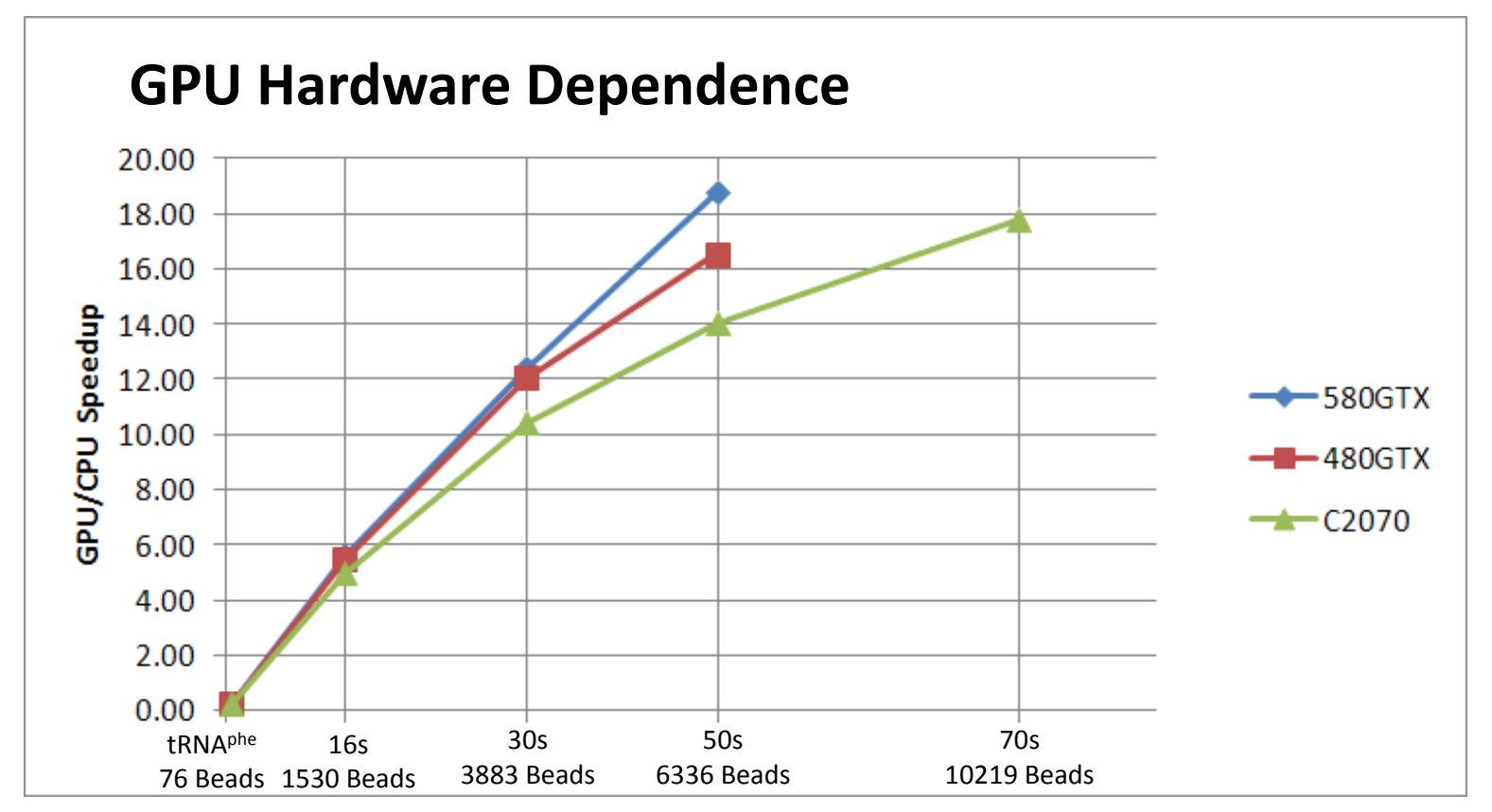
Step 3: Update Neighbor List to point to the first num_NL values of Master List.



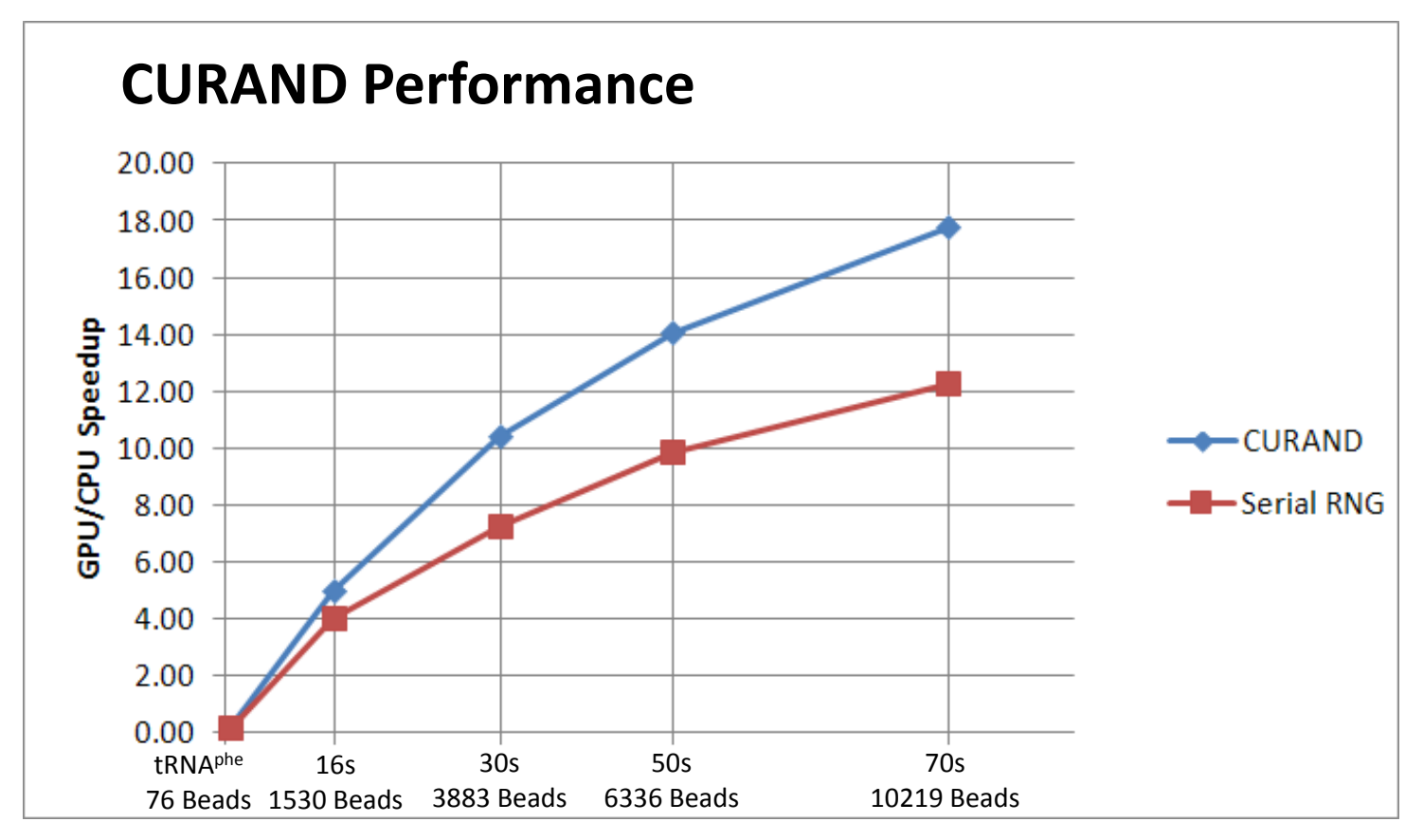


Improvements
- Reduced data transfers
- Reuse of arrays
- Reduces memory footprint

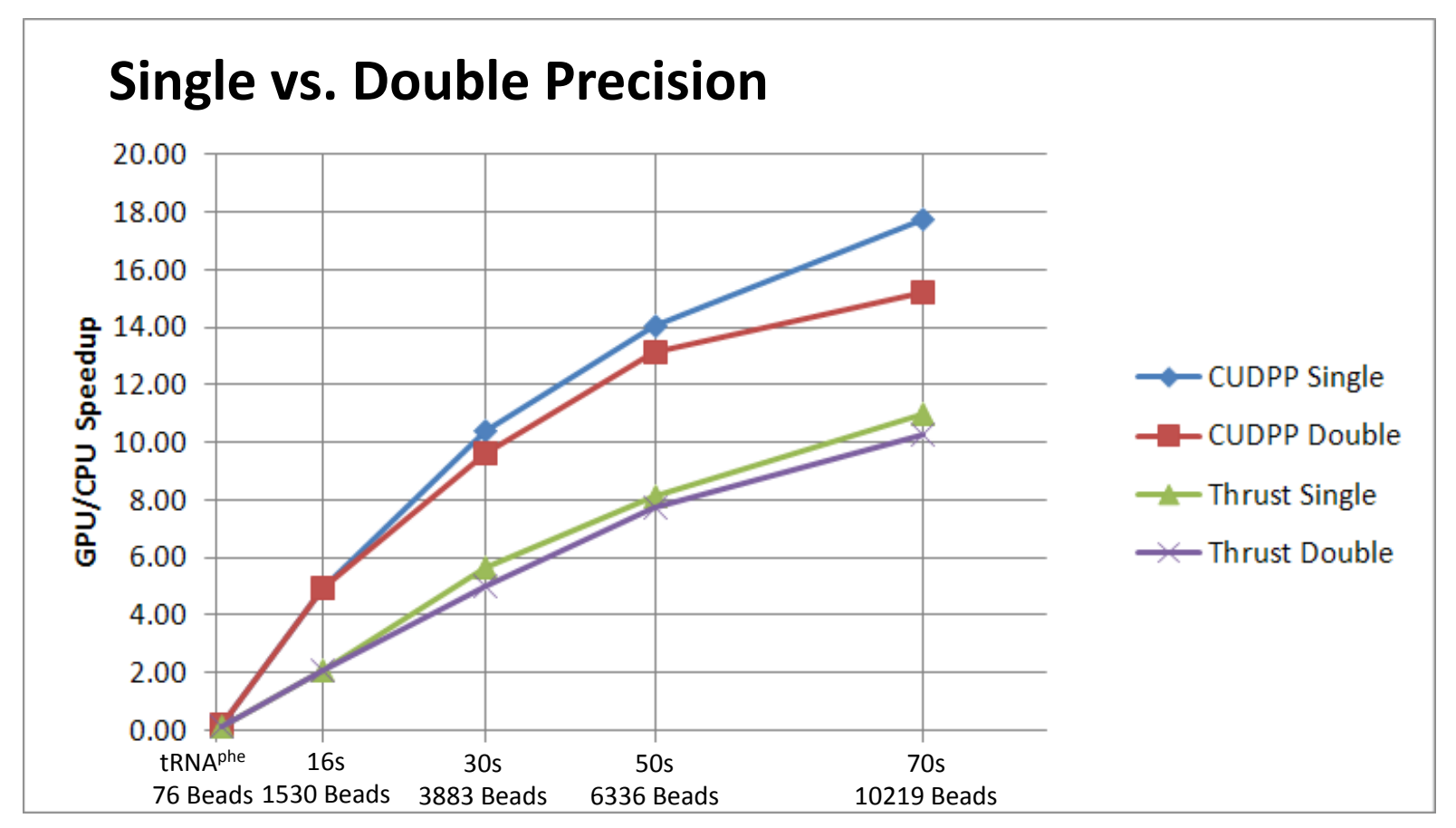
GPU-Based Molecular Dynamics Simulations Performances



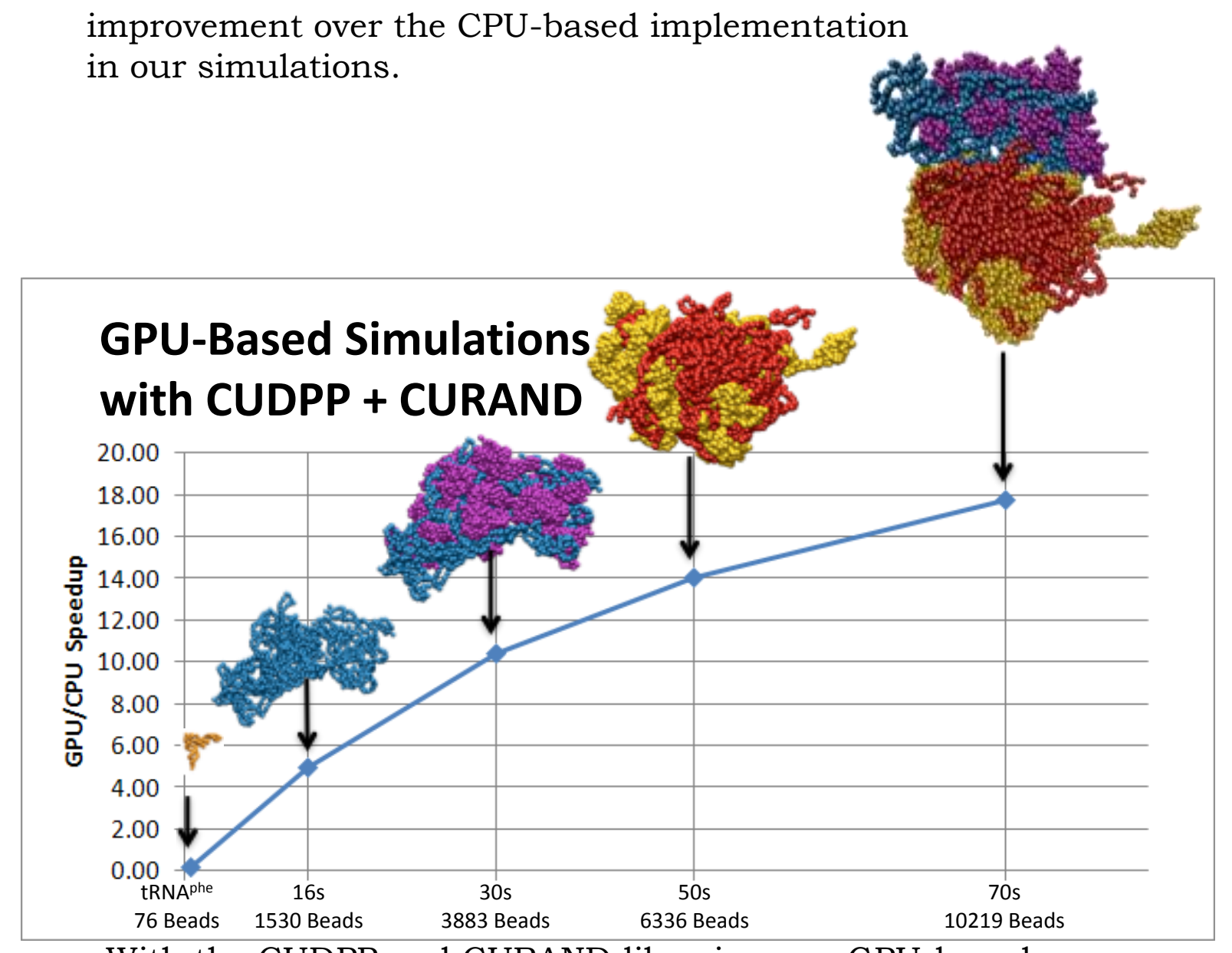
- The performance of the GPU-based simulations depends on the type of GPU.
- The 480/580 GTX are faster than the C2070, but its memory limits the size of the system one can simulate.



- The CURAND libraries showed a marked improvement over the CPU-based implementation in our simulations.



- The simulations performed using the CUDPP libraries with single precision calculations have a noticeable improvement in performance compared to double precision.
- With the Thrust libraries, the performanc was ~1/2 for single precision calculations as compared with the CUDPP libraries.



- With the CUDPP and CURAND libraries, our GPU-based simulations have approximately ~20x improvement over the CPU-based implementation.
- The performances of our simulations is clearly N-dependent.

Conclusions
- Molecular dynamics simulations can be highly optimized using NVIDIA's CUDA API along with the CUDPP and CURAND GPGPU libraries.
- Though memory transfers can cause severe bottlenecks, compression of data can significantly reduce overhead.
- Even non-parallel algorithms can be optimized to a high degree by developing new parallel approaches.
- CUDPP and CURAND libraries provide efficient code that can be quickly and easily implemented.
- There exists an N-dependent GPU vs. CPU performance speed-up (or –down).

Acknowledgements
- Wake Forest University Computer Science Graduate Fellowship for Excellence
- NVIDIA Academic Partnership
- Wake Forest University Science Research Fund