GPU Histogramming: Radial Distribution Functions

John Stone

Theoretical and Computational Biophysics Group

Beckman Institute for Advanced Science and Technology

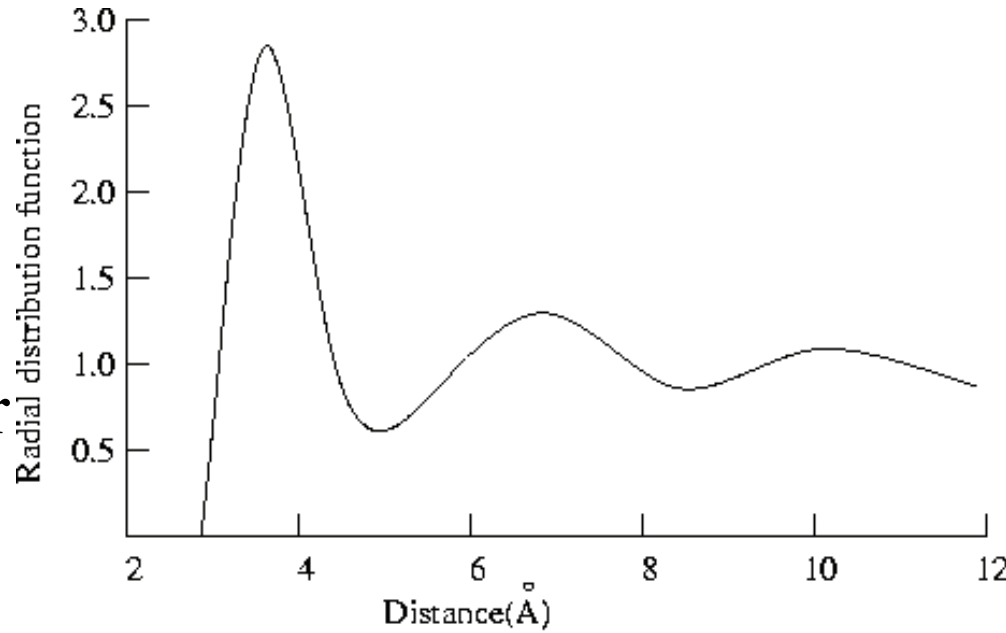
University of Illinois at Urbana-Champaign

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

Workshop on GPU Programming for Molecular Modeling , Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign, August 7, 2010

Radial Distribution Function

- Describes how atom density varies with distance
- Decays toward unity with increasing distance, for liquids
- Sharp peaks appear for solids, according to crystal structure, etc.
- Quadratic time complexity $O(N^2)$



Computing RDFs

- Compute distances for all pairs of atoms between groups A and B (A and B may be the same, or different)
- Use nearest image convention for periodic systems
- Each atom pair distance is inserted into a histogram
- Histogram is normalized one of several ways depending on use, but usually according to the volume of the spherical shells associated with each histogram bin

Histograms

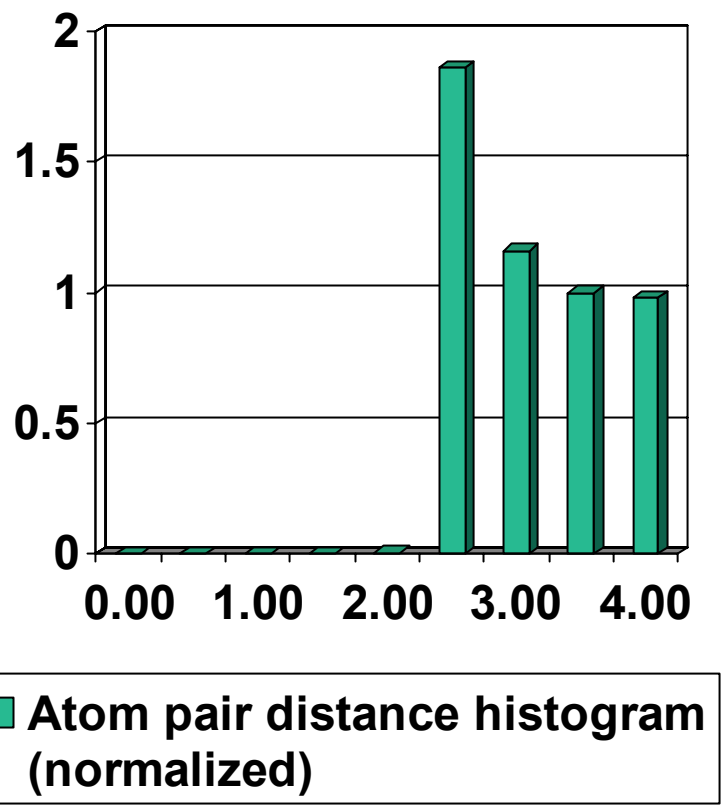
- Since the RDF algorithm contains histogramming as a key component, it may be interesting for other applications
- Volumetric density map processing tools for crystallography and cryo-EM often make use of histograms
- Useful for rapidly estimating quantiles, eliminating “outlier” data from visualizations, etc.

Computing RDFs on CPUs

- Atom data can be traversed in a strictly consecutive access pattern, yielding good cache utilization
- Since RDF histograms are usually small to moderate in size, they normally fit entirely in L2 cache
- Performance tends to be limited primarily by performance of histogram update step

Histogramming

- Partition population of data values into discrete bins
- Compute by traversing input population and incrementing bin counters



Histogramming on the CPU (slow-and-simple C)

```
memset(histogram, 0, sizeof(histogram));
for (i=0; i<numdata; i++) {
  float val = data[i];
  if (val >= minval  && val <= maxval) {
    int bin = (val - minval) / bindelta;
    histogram[bin]++;
  }
}
```

Random access updates to histogram array…

What About x86 SSE for RDF Histogramming?

- Atom pair distances can be computed four-at-a-time without too much difficulty
- Current generation x86 CPUs don't provide SSE instructions to allow individual SIMD units to scatter results to arbitrary memory locations
- Since the scatter operation must be done with scalar code, the histogram updates become the performance bottleneck for the CPU

Parallel Histogramming on Multi-core CPUs

- Parallel updates to a single histogram updates creates a potential output conflict
- CPUs have atomic increment instructions, but they often take hundreds of clock cycles, not intended for this purpose
- For small numbers of CPU cores, it is best to replicate the histogram for each CPU thread, compute them independently, and combine the separate histograms in a final reduction step

VMD Multi-core CPU RDF Implementation

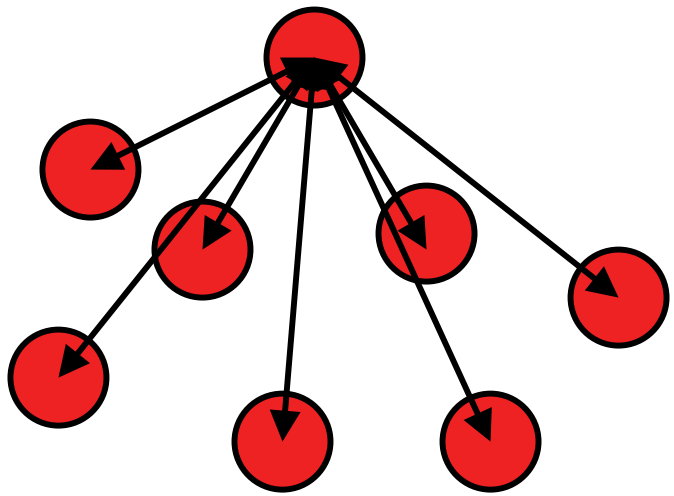
- Each CPU worker thread processes a subset of atom pair distances, maintaining its own histogram
- Threads acquire “tiles” of work from a dynamic work scheduler built into VMD
- When all threads have completed their histograms, the main thread combines the independently computed histograms into a final result histogram
- CPUs compute the entire histogram in a single pass, regardless of the problem size or number of histogram bins

Computing RDFs on the GPU

- Need tens of thousands of independent threads
- Each GPU thread computes one or more atom pair distances
- Histograms are best stored in fast on-chip shared memory
- Size of shared memory severely constrains the range of viable histogram update techniques
- Performance is limited by the speed of histogramming
- Fast CUDA implementation ~70x faster then CPU

GPU Work Decomposition

- Process atoms in group A by loading blocks of atoms into shared memory and processing them…
- Each thread computes all atom pair distances between current atom and all atoms in constant memory, inserting each distance into histogram



Computing Atom Pair Distances on the GPU

- Distances are computed using nearest image convention in the case of periodic boundary conditions
- Since all atom pair combinations will ultimately be computed, the memory access pattern is relatively simple
- Primary consideration is amplification of effective memory bandwidth, through use of GPU on-chip shared memory, caches, and broadcast of data to multiple or all threads in a thread block

GPU Atom Pair Distance Calculation

- Divide A and B atom selections into fixed size blocks
- Load a large block of A into constant memory
- Load small block of B into thread block's registers
- Each thread in a thread block computes atom pair distances between its atom and all atoms in constant memory, incrementing appropriate histogram bins until all A/B block atom pairs are processed
- Next block(s) are loaded, repeating until done…

GPU Histogramming

- Tens of thousands of threads concurrently computing atom distance pairs…
- Far too many threads for a simple per-thread histogram approach like CPU…
- Viable approach: per-warp histograms
- Fixed size shared memory limits maximum histogram size that can be computed in a single pass
- Large histograms require multiple passes

Intra-warp Atomic Update for Old NVIDIA GPUs (Compute 1.0, 1.1)

```
__device__ void addData(volatile unsigned int *s_WarpHist,
    unsigned int data, unsigned int threadTag) {
  unsigned int count;
  do {
    count = s_WarpHist[data] & 0x07FFFFFUL;
    count = threadTag | (count + 1);
    s_WarpHist[data] = count;
  } while(s_WarpHist[data] != count);
}
```

Loop until we were the "winning" thread to update value – store a 5-bit thread ID tag upper bits of counter

Per-warp Histogram Approach

- Each warp maintains its own independent histogram in on-chip shared memory
- Each thread in the warp computes an atom pair distance and updates a histogram bin in parallel
- Conflicting histogram bin updates are resolved using one of two schemes:
  - Shared memory write combining with thread-tagging technique (older hardware)
  - atomicAdd() to shared memory (new hardware)

RDF Inner Loops (Abbrev.)

```
// loop over all atoms in constant memory
for (iblock=0; iblock<loopmax2; iblock+=3*NCUDABLOCKS*NBLOCK) {
   __syncthreads();
   for (i=0; i<3; i++) xyzi[threadIdx.x + i*NBLOCK]=pxi[iblock + i*NBLOCK]; // load coords…
   __syncthreads();
   for (joffset=0; joffset<loopmax; joffset+=3) {
      rxij=fabsf(xyzi[idxt3  ] - xyzj[joffset  ]);  // compute distance, PBC min image convention
      rxij2=celld.x - rxij;
      rxij=fminf(rxij, rxij2);
      rij=rxij*rxij;
      […other distance components…]
      rij=sqrtf(rij + rxij*rxij);
      ibin=__float2int_rd((rij-rmin)*delr_inv);
      if (ibin<nbins && ibin>=0 && rij>rmin2) {
        atomicAdd(llhists1+ibin, 1U);
      }
   } //joffset
} //iblock
```

Writing/Updating Histogram in Global Memory

- When thread block completes, add independent per-warp histograms together, and write to per-thread-block histogram in global memory
- Final reduction of all of the per-thread-block histograms stored in global memory

Preventing Integer Overflows

- Since all-pairs RDF calculation computes many billions of pair distances, we have to prevent integer overflow for the 32-bit histogram bin counters (supported by the atomicAdd() routine)
- We compute full RDF calculation in multiple kernel launches, so each kernel launch computes partial histogram
- Host routines read GPUs and increment 64-bit (e.g. long, or double) histogram counters in host memory after each kernel completes

Acknowledgements

- Ben Levine and Axel Kohlmeyer at Temple University
- Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign
- NVIDIA CUDA Center of Excellence, University of Illinois at Urbana-Champaign
- NCSA Innovative Systems Lab
- The CUDA team at NVIDIA
- NIH support: P41-RR05969