ECE 498AL

Lecture 18:
Performance Case Studies:
Ion Placement Tool, VMD

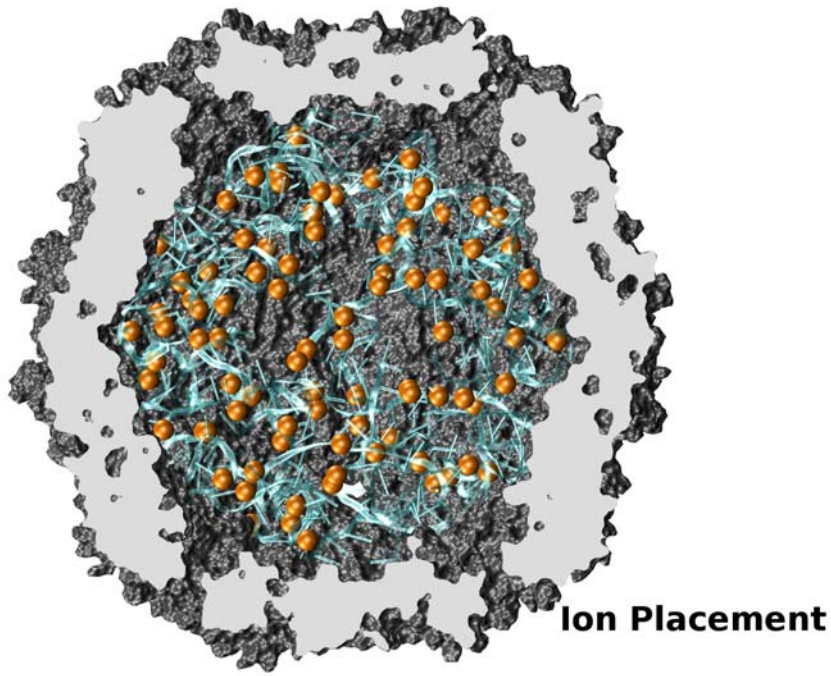
Guest Lecture by John Stone
Theoretical and Computational Biophysics Group
NIH Resource for Macromolecular Modeling and Bioinformatics
Beckman Institute for Advanced Science and Technology

Objective

- To learn design, implementation, and testing strategies for GPU acceleration of existing software using CUDA
  - Identify performance-critical software modules
  - Decompose identified modules into kernels which may benefit from GPU acceleration
  - Detailed examination of Coulombic potential map code
  - Abstract the implementation of the computational kernel so that caller need not worry about the low level details
  - Software structures supporting GPU acceleration

Molecular Modeling: Ion Placement

- Biomolecular simulations attempt to replicate *in vivo* conditions *in silico*.
- Model structures are initially constructed in vacuum
- Solvent (water) and ions are added as necessary for the required biological conditions
- Computational requirements scale with the size of the simulated structure



Evolution of Ion Placement Code

- First implementation was sequential
- Repeated scientific methodological revisions improved results
- As the size of simulated structures increased, the performance of the code became much more important
- Virus structure with 10^6 atoms would require 10 CPU days
- Tuned for Intel C/C++ vectorization+SSE, ~20x speedup
- Parallelized /w pthreads: high data parallelism = linear speedup
- Parallelized GPU accelerated implementation: Three GeForce 8800GTX cards outrun ~300 CPUs!
- Virus structure now runs in 25 seconds on 3 GPUs!
- Seems impossible until one considers how much faster GPUs are for graphics than a general purpose CPU…
- Further speedups should still be possible…

Ion Placement Algorithm

- Calculate initial Coulombic electrostatic potential map around the simulated structure:
  - For each voxel, sum potential contributions for all atoms in the simulated structure: potential += charge[i] / (distance to atom[i])
- Place ions one at a time:
  - Find the voxel containing the minimum potential value
  - Add a new ion atom centered on the minimum voxel position
  - Update the potential map adding the potential contribution of the newly placed ion
  - Repeat until the required number of ions have been added

Computational Profile of the Algorithm

- Over 99% of the run time of the algorithm is consumed in the initial potential map calculation, since the number of ions is always tiny compared to the size of the simulated system.
- Direct summation of electrostatic potentials is “safe” in terms of numerical accuracy, and is highly data parallel
- Interesting GPU test case since coulombic potential maps are useful for many other calculations
- Faster approximation algorithms currently in testing…

Coulombic Potential Map Slice: Simplest C Version GFLOPS? Don't ask…

```
void cenergy(float *energygrid, dim3 grid,, float gridspacing, float z, const float *atoms,  int numatoms) {
  int i,j,n;
  int atomarrdim = numatoms * 4;
  for (j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f;
      for (n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each atom
        float dx = x - atoms[n    ];
        float dy = y - atoms[n+1];
        float dz = z - atoms[n+2];
        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
    }
  }
}
```

Algorithm Design Observations

- Voxel coordinates are generated on-the-fly
- Atom coordinates are translated to the map origin in advance, eliminating redundant work
- Ion placement maps require ~20 potential voxels/atom
- Atom list has the smallest memory footprint, best choice for the inner loop (both CPU and GPU)
- Arithmetic can be reduced by creating a new atom list containing X, Q, and a precalculation of dy^2 + dz^2, updated for each row (CPU)

Observations and Challenges for GPU Implementation

- The innermost loop will consume operands VERY quickly
- Straightforward implementation has a low ratio of floating point arithmetic operations to memory transactions (for a GPU)
- Since atoms are read-only calculation, they are ideal candidates for texture memory or const memory
- GPU implementation must avoid bank conflicts and overlap computations with memory latency
- Map is padded out to a multiple of the thread block size, eliminating the need for conditional handling at the edges

Plan for CUDA
Coulombic Potential Map Code

- Allocate and initialize potential map memory on host
- Allocate potential map slice buffer on GPU
- Preprocess atom coordinates and charges
- Loop over slices:
  - Copy slice from host to GPU
  - Loop over groups of atoms: (if necessary)
    - Copy atom data to GPU
    - Run CUDA Kernel on atoms and slice resident on GPU
  - Copy slice from GPU to host
- Free resources

CUDA Block/Grid Decomposition

- 16x16 thread blocks are a nice starting size with a good number of threads
- Small enough that there's not much waste if we pad out the map array to an even number of thread blocks
- Kernel variations that unroll the inner loop calculate more than one voxel per thread
  - Thread count per block must be decreased to retain 16x16 block size
  - Or, block size gets bigger as threads do more than one voxel

Version 1: Tex Memory
90 GFLOPS, 9 Billion Atom Evals/Sec

- Pros:
  - Texture memory is large, enough capacity to hold millions of atoms
  - Most map slices could be computed in a single pass
- Cons
  - Texture fetches aren't as fast as shared memory or const memory
- For this algorithm, it has proven a better strategy to take advantage of broadcasting reads provided by const memory or shared memory, since all threads reference the same atom at the same time…

Version 1 Inner Loop Structure

Full source for CUDA potential map slice kernels:

http://www.ks.uiuc.edu/Research/vmd/projects/ece498/

```
…….
  float energyval=0.0f;
  for (atomid=0,tx=0,ty=0; atomid < numatoms; ty++) {
    for (tx=0; tx < TEXROWSIZE && atomid < numatoms; tx++, atomid++) {
      float4 atominfo = texfetch(tex, tx, ty);    // Bad, no latency hiding, not enough
      float dx = coor.x - atominfo.x;             // FP ops done per texfetch(),
      float dy = coor.y - atominfo.y;             // not taking any advantage of 2-D
      float dz = coor.z - atominfo.z;
      energyval += atominfo.w * (1.0f / sqrtf(dx*dx + dy*dy + dz*dz));
    }
  }
……
```

Version 2: Const+Precalc
150 GFLOPS, 16.7 Billion Atom Evals/Sec

- Pros:
  - Less addressing arithmetic (compared to texture version)
  - Pre-compute dz^2 for entire slice
  - Inner loop over read-only atoms, const memory ideal
  - If all threads read the same const data at the same time, performance is similar to reading a register
- Cons:
  - Const memory only holds ~4000 atom coordinates and charges
  - Potential summation must be done in multiple kernel invocations per slice, with const atom data updated for each invocation
  - Host code has a lot more book keeping to do, but not too big of an issue

Version 2: Kernel Structure

```
…
  float curenergy = energygrid[outaddr];  // start global mem read very early
  float coorx = gridspacing * xindex;
  float coory = gridspacing * yindex;
  int atomid;
  float energyval=0.0f;
  /* Main loop: 9 floating point ops, 4 FP loads per iteration */
  for (atomid=0; atomid<numatoms; atomid++) {
    float dx = coorx - atominfo[atomid].x;
    float dy = coory - atominfo[atomid].y;
    energyval += atominfo[atomid].w *
                          (1.0f / sqrtf(dx*dx + dy*dy + atominfo[atomid].z));
  }
  energygrid[outaddr] = curenergy + energyval;
```

Version 3: Const+Precalc+Loop Unrolling 226 GFLOPS, 33 Billion Atom Evals/Sec

- Pros:
  - Although const memory is very fast, loading values into registers costs instruction slots
  - We can reduce the number of loads by reusing atom coordinate values for multiple voxels, by storing in regs
  - By unrolling the X loop by 4, we can compute dy^2+dz^2 once and use it multiple times, much like the CPU version of the code does
- Cons:
  - Compiler won't do this type of unrolling for us (yet)
  - Uses more registers, one of several finite resources
  - Increases effective tile size, or decreases thread count in a block

Version 3: Inner Loop

```
…
  for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
    float dx1 = coorx1 - atominfo[atomid].x;
    float dx2 = coorx2 - atominfo[atomid].x;
    float dx3 = coorx3 - atominfo[atomid].x;
    float dx4 = coorx4 - atominfo[atomid].x;
    energyvalx1 += atominfo[atomid].w * (1.0f / sqrtf(dx1*dx1 + dysqpdzsq));
    energyvalx2 += atominfo[atomid].w * (1.0f / sqrtf(dx2*dx2 + dysqpdzsq));
    energyvalx3 += atominfo[atomid].w * (1.0f / sqrtf(dx3*dx3 + dysqpdzsq));
    energyvalx4 += atominfo[atomid].w * (1.0f / sqrtf(dx4*dx4 + dysqpdzsq));
  }
…
```

Version 4: Const+Shared+Loop Unrolling+Precalc 235 GFLOPS, 34.8 Billion Atom Evals/Sec

- Pros:
  - Loading prior potential values from global memory into shared memory frees up several registers, so we can afford to unroll by 8 instead of 4
  - Using fewer registers allows more blocks, increasing GPU “occupancy”
- Cons:
  - Even with shared memory, still uses 21 registers
  - Only a net performance gain of ~5% over version 3
  - Higher performance should still be possible
  - Bumping against hardware limits (uses all const memory, most shared memory, and a largish number of registers)
- Need more experience or a different strategy in order to go beyond this level of performance

Version 4: Kernel Structure

- Loads 8 potential map voxels from global memory at startup, and immediately stores them into shared memory before going into inner loop
- Processes 8 X voxels at a time in the inner loop
- Sums previously loaded potential values and stores back to global memory
- Code is too long (and ugly) to show even in a snippet due to the large amount of manual unrolling of loads into registers
- Various attempts to further reduce register usage didn't yield any benefits, so a different approach is required for further performance gains on a single GPU
- See full source example "cuenergyshared"

Calculating Potential Maps in Parallel

- Both CPU and GPU versions of the code are easily parallelized by decomposing the 3-D potential map into slices, and computing them concurrently
- For the ion placement tool, maps often have 200-500 slices in the Z direction, so there's plenty of coarse grained parallelism still available even for a big machine with hundreds of CPUs/GPUs

Parallel GPUs with Multithreading: 705 GFLOPS /w 3 GPUs

- One host thread is created for each CUDA GPU
- Threads are spawned and attach to their GPU based on their host thread ID
  - First CUDA call binds that thread's CUDA context to that GPU for life
  - Handling error conditions within child threads is dependent on the thread library and, makes dealing with any CUDA errors somewhat tricky, left as an exercise to the reader…. ☺
- Map slices are computed cyclically by the GPUs
- Want to avoid false sharing on the host memory system
  - map slices are usually much bigger than the host memory page size, so this is usually not a problem for this application
- Performance of 3 GPUs is stunning!
- Power: 3 GPU test box consumes 700 watts running flat out

Multi-GPU CUDA
Coulombic Potential Map Performance

- Host: Intel Core 2 Quad, 8GB RAM, ~$3,000
- 3 GPUs: NVIDIA GeForce 8800GTX, ~$550 each
- 32-bit RHEL4 Linux (want 64-bit CUDA!!)
- 235 GFLOPS per GPU for current versoin of coulombic potential map kernel
- 705 GFLOPS total for multithreaded multi-GPU version

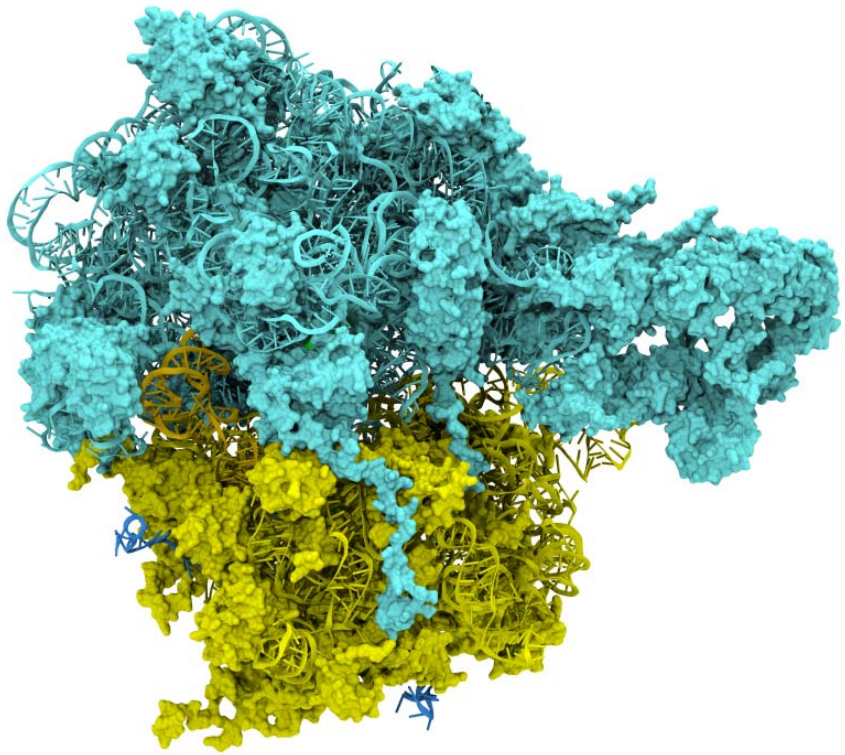
Three GeForce 8800GTX GPUs in a single machine, cost ~$4,650

Never Trust Compilers
(With apologies to Wen-mei and David)

- When performance really matters, it is wise to distrust compilers by default and to read their assembly output to see if you're getting what you had hoped for
  - Compilers often miss "easy" optimizations for various reasons
  - By reading intermediate output, e.g. PTX, you can find ways to coax the compiler into doing what you want
  - PTX still isn't the final word, as it gets run through another optimization pass, but it's the first place to look until better tools are available
- Test on microbenchmarks representative of inner loops before integrating into real code
  - Small benchmark codes facilitate focused experimentation
  - MUCH easier to isolate bugs and performance issues in a small code than a large one

Early Experiences Integrating CUDA Kernels Into VMD

- VMD: molecular visualization and analysis
- State-of-the-art simulations require more viz/analysis power than ever before
- For some algorithms, CUDA can bring what was previously supercomputer class performance to an appropriately equipped desktop workstation
- Early results from a variation on the work already done for the ion placement tool



Ribosome: 260,790 atoms before adding solvent/ions

VMD/CUDA Integration Observations

- Single VMD binary must run on all hardware, whether CUDA accelerators are installed or not
  - Must maintain both CPU and CUDA versions of kernels
  - High performance requirements mean that the CPU kernel may use a different memory layout and algorithm strategy than CUDA, so they could be entirely different bodies of code to maintain
  - Further complicated by the need to handle both single-threaded and multithreaded compilations, support for many platforms, etc…

VMD/CUDA Integration Observations (2)

- Graceful behavior under errors or resource exhaustion conditions becomes trickier to deal with:
  - CPU kernel becomes the fallback
  - What to do when the CPU version is 100x slower than CUDA on the GPU?!?
- All of these software design problems already existed:
  - Not specific to CUDA
  - CUDA just adds another ply to the existing situation for codes like VMD that employ multiple computation strategies

VMD/CUDA Resource Management

- Must choose the best kernel/strategy at runtime, depending on availability of CPU/GPU resources, combined with user preferences and system policies
- Examples:
  - Good for VMD to use all CPUs and CUDA GPUs on a workstation not shared by multiple users
  - Bad for VMD to use all 1024 processors on a shared supercomputer by default (e.g. running remotely in text mode for batch analysis)

VMD/CUDA Resource Management (2)

- Dynamically changing load on CPUs/GPUs:
  - Interference from other apps multitasking on the same set of CPUs/GPUs
  - A “benchmark” run at startup can become invalid for selection of kernel strategy if CPU/GPU load changes during the course of a long-running execution (e.g. overnight analysis job running at the same time as an interactive visualization, both vying for the CPUs/GPUs…)
  - Perhaps the computation strategy should be periodically re-tested/evaluated as load conditions change

VMD/CUDA Code Organization

- Single header file containing all the CUDA kernel function prototypes, easy inclusion in other src files
- Separate .cu files for each kernel:
  - each in their compilation unit
  - no need to worry about multiple kernels defining const buffers etc…
- As new CUDA kernels augment existing CPU kernels, the original class/function becomes a wrapper that dynamically invokes the CPU/GPU version at runtime

VMD/CUDA Code Organization (2)

- A C++ wrapper class to hold data needed for execution strategy, CPU/GPU load balancing, etc. (much is still unimplemented and only exists in my head)
- First CUDA GPU kernels are so much faster than the CPU that the existing VMD runtime strategy is nearly as simple as:

```
int err = 1; // force CPU execution if CUDA is not compiled in
#if defined(VMDCUDA)
if (cudagpucount > 0)
  err=CUDAKernel(); // try CUDA kernel if GPUs are available
#endif
if (err)
  err=CPUKernel(); // if no CUDA GPUs or an error occurred, try on CPU
…
```