BrainGrid Plan File

Updated April 28, 2010

Advisor: Prof. Stiber

Students: Ethan Crawford, Sean McCallum, Fumitaka Kawasaki

# BUGS:

1. When running as a single process on a multi-core machine, the process frequently migrates cores when running single process/single thread. This causes a few seconds of very slow execution. (The Portable Linux Processor Affinity library wraps 2 types of processor affinity system calls and may be the solution to this problem. See <http://www.open-mpi.org/projects/plpa/>). (Probably safe to ignore, as we are not focused on the single-threaded sim.)

# WORK ITEMS:

1. Run sim on MacOS
2. Refactor all time constants to be integer-based and use tick count rather than double-based using fractions of a second. This should address some of the endless loop issues seen when moving between float and double.
3. Re-run sim on Linux multiple times to check RNG influence on results
4. Confirm Matlab final firing rate plot parameters (2007/2008 plots are horked.)
5. Validate refactored codebase against original codebase , using an identical seed to confirm that the output is also identical (Ethan, in progress)
6. Qualitatively validate simulation results (generate plots and compare images to previous versions) (Ethan, in progress)
7. Refactor code to enable measuring error drift between doubles vs. floats (Ethan, in progress)
8. Investigate thread-related seeding issues (delta between single and multi-threaded output)
9. Run long (100 secs, 300 growth cycles) simulations, both double and float. Record output.
10. Integrate GPU v1 (Fumitaka and Sean). Integrate Allan’s single-block CUDA code into GpuSim.cpp.
11. Health: during growth, sort synapses at each coordinate by distance to summation point and dispose of those whose strength has gone to 0.
12. Implement streaming XML writer (required to write large history files one growth cycle at a time.)
13. Save synapse and neuron state to support simulation pause and resume (useful for storing pre-work to get a simulation into an interesting state and required to modify runtime parameters later.)
14. NVIDIA Tesla/CUDA development
    1. Copy C++ collection objects to C-style data structures (we believe this is done; Ethan will confirm). This may not be needed, as some C++ support exists in CUDA. There is also an STL-style library called Thrust that makes this easy.
    2. Create C style functions to perform Advance() functions on C style data structs (we believe this is done; Ethan will confirm)
    3. “CUDA cores” implementation
    4. <http://code.google.com/p/thrust/> is an STL-style library for efficient marshaling of vector types.
15. Documentation:
    1. Document existing feature set
    2. Mention BrainGrid project and status on biocomputing website (http://depts.washington.edu/biocomp/). Use Prof. Fukuda’s site as a template (Prof. Stiber can add more detail here.)
    3. Complete documenting all major methods in Network.cpp.
    4. Resolve remaining formatting inconsistencies in Network.cs
    5. Add Javadoc-style comments to all C++ functions.
    6. Re-run Doxygen on codebase after
16. Refactoring
    1. Review refactoring suggestions with team and incorporate feedback (done on 4/28/2010; see “BrainGrid Class Diagram.vsd”) (Done)
    2. Implement refactor (Scheduled for week of April 26th)
    3. Move unused code into legacy directory

# OVERALL PLAN:

1. Build single neuron model - complete
2. Build grid of neurons - complete
3. Model synapses - complete
4. Model growth - complete
5. Write documentation - in progress
6. Validate - in progress
7. Compare single and and mult-threaded implementation and validate:
8. Is the data the same? (we believe the answer is no, because of RNG calls by multiple threads.)
9. If the data is not the same, does it need to be?
10. Evaluate quantitative and qualitative metrics to confirm or deny simulation “sameness”
11. If yes, pre-generate RNG sequence and/or come up with some other solution
12. Otherwise, define the quantitative and qualitative metrics that imply sameness.
13. Optimize
14. Get CUDA code from Allan.
15. Get basic GPU implementation working, then inspect it for optimization points and do that work.
16. Thread - check. Could improve performance by locating the bug in collapsing each thread's local summationPoint copy. (Allan, what does this mean? We believe you’re saying that the compiler is generating code to reference a single copy of an object, rather than copying it per-thread.)
17. Write functions to copy data from classes into arrays of structs for consumption by the cuda c compiler.
18. Write GPU-compatible C-style functions to advance the network (create “kernels” or “work units” that can execute in parallel.) There shouldn't need to be many modifications to the existing code in Network, DynamicSpikingSynapse and LifNeuron.
19. Write code to copy the necessary simulation data onto the gpu. We believe that not all the synapse state will fit in GPU memory, so there may be some work here to determine which subset is the correct one to have in memory at any one time.
20. Write code to copy the simulation data from the gpu back into the simulator
21. Create a per-neuron RNG thread using a cached seed.
22. Convert the C style functions into kernels.
23. Validate basic GPU implementation
24. Optimize GPU implementation