Accelerating Large-Scale Simulations of Cortical Neuronal Network Development

Fumitaka Kawasaki
University of Washington, Bothell
fumik@u.washington.edu

Michael Stiber
University of Washington, Bothell
stiber@u.washington.edu

March 2012
Accelerating large-scale simulations of cortical neuronal network development

Fumitaka Kawasaki     Michael Stiber
Computing & Software Systems Program
University of Washington Bothell
18115 Campus Way NE
Bothell, WA 98011 USA
fumik@u.washington.edu
stiber@u.washington.edu

CSS Technical Report# UWB-CSS-12-01
February 2012

Abstract

Cultured dissociated cortical cells grown into networks on multi-electrode arrays are used to investigate neuronal network development, activity, plasticity, response to stimuli, the effects of pharmacological agents, etc. We made a computational model of such a neuronal network and studied the interplay of individual neuron activity, cell culture development, and network behavior. For small networks (100 neurons in a $10 \times 10$ arrangement), we concluded that our simulations’ behaviors were dominated by their limited size. However, increasing network size required huge computational resources: for a single-threaded simulator, a $100 \times 100$ neuron simulation would take at least 2,000 hours (83 days). To tackle this problem, we ported the network simulator to the GPU. A first, naive implementation performed about 2.4 times faster than the single thread simulator. By progressively modifying the simulator structure, we achieved about 23 times performance gain compared with the single threaded simulator, bringing large-scale simulations into the realm of feasibility. Through this experience, we identified factors that limit performance improvement for this type of application.

keywords: computational neuroscience, neural development, parallel simulation, GPU algorithms

1 Background

How activities and connections of individual neurons contribute to the development and computation of cerebral cortex is one of the central questions in neuroscience. A powerful experimental approach for investigating these questions is growing networks of cultured dissociated cortical cells on multi-electrode arrays. Such preparations allow investigation of network development, activity, plasticity [5], responses to stimuli [4], the effects of pharmacological agents [3], etc.

In such experiments, network behavior commonly converges to whole-culture pathological (in the sense that it does not occur in vivo) bursting that generally interferes with the experimental goals [8]. This bursting is interesting from both a theoretical point of view [9], as well as a clinical one. Understanding the mechanisms that underlie bursting could allow creation of more useful cell cultures and have medical applications [11].

The work described here is part of an ongoing computational study of the interplay of individual neuron activity, cell culture development, and network behavior. While theoretical studies have revealed that networks of randomly-connected model neurons can produce bursting behavior [9], we have investigated whether such behavior can occur in an initially unconnected network that develops connections according to a model of cell culture connectivity growth. This model includes descriptions of neuron, synapse (connection), and development (connection formation).
The neuron model is of the integrate-and-fire type and includes synaptic, constant bias, and noise currents:

\[ C_m \frac{dV_m}{dt} = \frac{1}{R_m} (V_{\text{rest}} - V_m) + I_{\text{syn}} + I_{\text{inj}} + I_{\text{noise}} \]  

(1)

When \( V_m \) exceeds the threshold voltage \( V_{\text{thresh}} \), it is reset to \( V_{\text{reset}} \) and held there for the length of the absolute refractory period, \( T_{\text{refract}} \).

Synapses exhibit dynamics that include activity-dependent facilitation and depression. Their model has four state variables: three that govern the fraction of synaptic resources in particular states — \( x \) (recovered state), \( y \) (active state), and \( z \) (inactive state) — and one, \( u \), that represents synaptic efficiency,

\[
\begin{align*}
\frac{dx}{dt} &= \frac{z}{\tau_{\text{rec}}} - ux\delta(t-t_{\text{sp}}) \\
\frac{dy}{dt} &= -\frac{y}{\tau_I} + Ux\delta(t-t_{\text{sp}}) \\
\frac{dz}{dt} &= \frac{y}{\tau_I} - \frac{z}{\tau_{\text{rec}}} \\
\frac{du}{dt} &= \frac{u}{\tau_{\text{facil}}} + U(1-u)\delta(t-t_{\text{sp}})
\end{align*}
\]

(2)  

(3)  

(4)  

(5)

where \( \delta(t-t_{\text{sp}}) \) is the unit impulse at time \( t_{\text{sp}} \), the arriving spike time. The three time constants \( \tau_I, \tau_{\text{rec}}, \text{and } \tau_{\text{facil}} \) govern inactivation after an arriving spike, recovery from inactivation, and facilitation after a spike, respectively. The synaptic current produced by an arriving spike is \( I_{\text{syn}} = W_y \), where \( W \) is the strength of the synaptic connection [6].

Synaptic strength, \( W \), was determined dynamically by a model of neurite (cell input and output region) growth and synapse formation [10]. In this, a cell’s region of connectivity is modeled as a circle with radius that changes at a rate inversely proportional to a sigmoidal function of cell firing rate:

\[
\frac{dR_i}{dt} = \rho G(F_i)
\]

(6)

\[
G(F_i) = 1 - \frac{2}{1 + \exp((\epsilon - F_i)/\beta)}
\]

(7)

where \( R_i \) is the radius of connectivity of neuron \( i \), \( F_i \) is neuron \( i \)’s firing rate (normalized to be in the range \([0,1])\), \( \rho \) is a rate constant, \( \epsilon \) is a constant that sets the “null point” for outgrowth (the firing rate in spikes/sec that causes no outgrowth or retraction), and \( \beta \) determines the slope of \( G(\cdot) \). One divergence in these simulations from strict modeling of the living

preparation was that \( \rho \) was increased to reduce simulated development times from the weeks that the living preparation takes to 60,000s (approximately 16 simulated hours). Extensive analysis and simulation was performed to determine the maximum \( \rho \) that would not interfere with network dynamics (the increased value of \( \rho \) was still orders of magnitude slower than the slowest of the neuron or synapse time constants).

Synaptic strengths were computed for all pairs of neurons that had overlapping connectivity regions as the area of their circle’s overlap. The bulk of the neurons in the network were excitatory; a small number were chosen to be inhibitory. Similarly, most neurons were not spontaneously active, but a few had their firing threshold, \( V_{\text{thresh}} \), lowered from 15mV to \( 13.6 \leq V_{\text{thresh}} \leq 13.7 \) to produce spontaneous firing at a rate of between 0.02 and 6 spikes/sec.

As shown in figure 1, each simulation proceeded as a sequence of 100 second segments (activity epochs) with 0.1 millisecond time step, during which the connectivity was kept constant. The average firing rate of each neuron during the preceding activity epoch was used to adjust its neurite outgrowth, according to equation (6), for the next. Simulations were 300–600 activity epochs long (30,000–60,000 seconds).

Before GPU implementation, the simulator supported parallelization using OpenMP, in addition to a single-threaded version. This architecture is illus-
Figure 2: Simulator architecture. An abstract interface (ISimulation) was used to allow swapping of different underlying simulator implementations.
tated in figure 2.

Initial simulations consisted of networks of 100 neurons in a $10 \times 10$ arrangement, with each simulation taking around 20 hours on computers with 2-3GHz microprocessors. Preliminary results from these simulations indicated that behavior was dominated by small network effects. Larger simulations, say $100 \times 100$ networks, would help distinguish between the effects of network size and inherent behavior. However, increasing network size has computational consequences that must be addressed: in the single-threaded form, a 60,000s simulation of a $100 \times 100$ network would take at least 2,000 hours (83 days). Parallelization using OpenMP could not feasibly lead to large enough speedups to make such simulations practical, and thus we turned to a GPU implementation.

2 Match between Algorithm Structure and GPU Architecture

As described in figure 1, the simulation algorithm consisted of updates to neuron and synapse state that occurred every 0.1ms time step. For a $100 \times 100$ network, the number of synapses, determined by neuron connectivity radius ($R_i$), was much larger than the 10,000 neurons. Preliminary simulations indicated that the common equilibrium value of neuron radius would yield a ratio between the number of neurons and the number of synapses of $1:46.4$. Therefore, the states of 10,000 neurons and about 464,000 synapses were required to be updated every time step, using the exponential Euler method for numerical integration of equations (1)–(5). Consequently, a 60,000s simulation time of a $100 \times 100$ arrangement represents $6 \times 10^8$ time steps, or approximately $2.8 \times 10^{14}$ state updates.

The massive parallelism of a GPU seemed to suit the inherent parallelism of the simulation algorithm very well. In the single threaded implementation, the calculation for each neuron and synapse was done by two inner loops, one for neurons and one for synapses. A calculation for each neuron or synapse is independent of other neurons or synapses, so those calculations can be done in parallel. Therefore, the naive conversion of the loops into GPU kernel functions — `advanceNeuronDevice()` and `advanceSynapseDevice()` — was straightforward. Each thread of the kernel functions implemented one iteration of the original loops of the single threaded C++ code. Within the kernels, each thread could identify its neuron or synapse using its `blockIdx` and `threadIdx` values.

3 GPU Implementation

We implemented the simulator on a GPU and performed simulations with 10,000 neurons and 464,000 synapses in a $100 \times 100$ arrangement. We performed one 100s activity epoch, including growth update at its conclusion, on each revision of the GPU implementation, which includes the whole execution path and allows us to evaluate the performance of each component. We set all starting neuron radii to 2.0 because that was a common equilibrium value of neuron radius in previous simulations. This allowed for evaluation of simulator performance at that state. Simulations were run on a 2.8GHz AMD Phenom II X4 920 processor with 512KB L2 cache and an NVIDIA Tesla C1060 device. Execution times of kernel functions were determined using GPU timers. The CUDA event API call `cudaEventRecord()` was placed before and after each kernel function to record time stamps and the `cudaEventElapsedTime()` API call returned elapsed time. Elapsed time was accumulated under iteration and resulted in cumulative execution times.

Referring to the simulator architecture described in figure 2, we implemented a `GpuSim` sub-class of `ISimulation` with a GPU version of the `advancedUntilGrowth()` method, where the process of one activity epoch was performed. For each time step, the state update of each neuron and each synapse was performed by the kernel functions `advanceNeuronDevice()` and `advanceSynapseDevice()` concurrently, in contrast to their sequential nature in the single-threaded version. Figure 3 compares execution times of the single-threaded version and GPU version. As seen in the figure, this naive GPU implementation achieved a speedup about 2.4 times against the single thread version — not too dissimilar to the multi-core OpenMP version.

One inherent bottleneck in the model’s structure lies in the convergence of synapse input onto each individual neuron. This occurs at each neuron’s summation point, where synapse responses are summed to produce a net input for the connected neuron. These summation points might be written into by more than one synapse, modified simultaneously by `advanceSynapseDevice()` kernel function. Therefore, we used atomic operations — read-modify-write without interruption from other threads — to avoid conflict.
However, these atomic operations were very expensive, so the `advanceSynapseDevice()` kernel function occupied the majority of the execution time. To eliminate the atomic locks, the code was modified as follows:

1. The summation operation was separated from the kernel function `advanceSynapseDevice()`.

2. An *inverse map* was created. This was a table indexed by neuron number that mapped to the synapses (by synapse ID) that provided input to that neuron.

3. A new kernel function, `calcSummationMap()`, was implemented. One thread of this executed for each neuron, using the inverse map to sum up the responses of the synapses that provided input to that neuron.

Next, we applied various performance-enhancement techniques to the kernel functions, which included:

1. Reduced global memory access by using registers.

2. Combined kernel functions. The naive implementation mirrored the single-threaded version by using device functions, called from the GPU kernels, that matched subroutines to handle such things as spike arrivals to synapses and neuron spike generation. These were merged into the `advanceNeuronDevice()` function to eliminate the significant overhead associated with the function calls.

3. Used constant memory. Synapse constants, $U$, $D$, $F$, were moved into constant memory.

4. Each synapse object had a delay queue: a queue to store incoming spike events scheduled to arrive in the future, with its own pointer to indicate current position in the queue. We made the delay queue pointer global to all synapses; therefore, each synapse didn’t need to update the pointer.

5. The neuron and synapse data were originally stored as arrays of `LifNeuron_struct` and `DynamicSpikingSynapse_struct`, respectively. This mirrored the object-oriented nature of the single-threaded simulator. However, this configuration prevented favorable global memory access patterns because the GPU hardware achieves high global memory access efficiency when all threads in a warp access consecutive global memory locations (coalesced memory access). To correct this, the arrays of neuron and the synapse data structures were reorganized into global neuron and synapse structures containing multiple, homogeneous arrays of data.

6. Neuron network connectivity update is done once each 100 seconds of simulated time (once each activity epoch). Therefore, the ratio of connectivity update to simulation time step is $1 : 1,000,000$ (remember that the simulation time step is 0.1ms). However, the execution time of adjusting synapses, which is the major portion of `updateNetwork()`, still took about 9.4% of total execution time. This was because updating was done by a function on the host CPU, so there was room for performance improvement by moving that process to the GPU. We refactored the simulator architecture so that the ISimulation
interface included an `updateNetwork()` method, and implemented a GPU version of this method. Updating connectivity on the GPU also eliminated the need to copy neuron and synapse data between host and GPU memory each activity epoch. As a result, the only communication necessary between host and GPU was initialization, kernel sequencing, once-per-epoch communication of results from GPU to host, and final communication of simulation state from GPU to host (to enable simulation resumption at a later time).

7. A random number generator was used to produce each neuron’s noise current, $I_{noise}$. Because a set of random numbers is needed for each time step, random number generation occupied about 55% of the total execution time. Random numbers were originally generated by a single thread on the host CPU; we implemented a GPU version of the random number generator, using a Mersenne Twister algorithm.

The successive improvements brought by these optimizations are shown in figure 4. The final combination of performance tuning techniques produced an approximately 23 times speedup compared with the single threaded version. The final simulator version runs almost entirely on the GPU, requiring 4–5 days to complete a 60,000s simulation.

4 Detailed Performance Results

Three of the kernels comprising the final GPU implementation account for most of its execution time: `advanceNeuronDevice()`, `advanceSynapseDevice()`, and `calcSummationMap()`. Detailed performance data was collected and analyzed for these to enable a more precise understanding of which limitations were inherent in the algorithmic structure of this type of simulator, which were peculiar to the particular simulator, and which might be amenable to further optimization. Table 1 presents basic configuration information and execution time for these three major kernel functions. We also examined multiprocessor occupancy, thread divergence, effective memory bandwidth, and effective MIPS.

4.1 Multiprocessor Occupancy

*Occupancy* is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. The number of active warps per streaming multiprocessor (SM) is limited by several CUDA hardware resources, such as register availability [2]. Table 1 indicates occupancy for the three major kernels. Because of their larger use of register memory, `advanceNeuronDevice()` and `advanceSynapseDevice()` were not able to achieve 100% occupancy.

4.2 Thread Divergence

*Thread divergence* occurs when threads in the same warp follow different paths of control flow, such may occur in if-then-else or for-loop constructs. When divergent, parts of a warp take different execution paths sequentially — the threads for one pathway are executed (until the divergent paths rejoin), then those for the next, etc., until all pathways have been processed. This has an important effect on performance. Table 2 summarizes thread divergence data for the three major GPU kernels.

In table 2, the divergent paths 1, 2, and 3 of the `advanceNeuronDevice()` kernel function correspond to the three neuron states (refractory, firing, and normal), respectively. Each probability was calculated as follows.

From simulation data, the average firing rate of a single neuron was 19.4085Hz; in other words, on average each neuron fired that many times per second. An excitatory neuron’s refractory period was 3ms; an inhibitory neuron’s was 2ms. The percentages of excitatory and inhibitory neurons was 98% and 2%, respectively. Therefore the average refractory period (for each time it fired) was 2.98 milliseconds. Firing events themselves took one simulation step, 0.1ms. Thus, the average probabilities of a thread taking each of the three paths was:

$$P(\text{refractory}) = p_1 = (19.4085)(2.98)/1,000 \approx 0.058$$  \hspace{1cm} (8)

$$P(\text{firing}) = p_2 = 19.4085/10,000 \approx 0.002$$ \hspace{1cm} (9)

$$P(\text{normal}) = p_3 = 1 - (0.058 + 0.002) = 0.94$$ \hspace{1cm} (10)

The divergence paths 1 and 2 of the `advanceSynapseDevice()` kernel function correspond to the states of a synapse (firing and normal), respectively, and each probability was calculated from the average
Figure 4: Summary of speedups produced by different techniques. Execution times of a single activity epoch, plus network connectivity update, are shown in comparison to the single-threaded CPU version (bottom). From bottom to top, times are for the naive GPU implementation and successively optimized versions.

<table>
<thead>
<tr>
<th>Kernel function</th>
<th>$\text{AdvanceNeuronDevice}$</th>
<th>$\text{AdvanceSynapseDevice}$</th>
<th>$\text{calcSummationMap}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per block</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Registers per thread</td>
<td>28</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>Physical max warps/SM</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td># of warps/SM</td>
<td>16</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Occupancy</td>
<td>50%</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td># of iteration</td>
<td>1,000,000</td>
<td>1,000,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td># of threads</td>
<td>10,000</td>
<td>464,108</td>
<td>10,000</td>
</tr>
<tr>
<td>Cumulative execution time (ms)</td>
<td>162,631</td>
<td>635,624</td>
<td>529,130</td>
</tr>
</tbody>
</table>

Table 1: Measured performance of three kernels. Basic data and cumulative execution time for each kernel function.

<table>
<thead>
<tr>
<th>Kernel function</th>
<th>$\text{AdvanceNeuronDevice}$</th>
<th>$\text{AdvanceSynapseDevice}$</th>
<th>$\text{calcSummationMap}$</th>
</tr>
</thead>
<tbody>
<tr>
<td># of paths</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Probability of divergent path 1</td>
<td>0.058</td>
<td>0.002</td>
<td>1.000</td>
</tr>
<tr>
<td>Probability of divergent path 2</td>
<td>0.002</td>
<td>0.998</td>
<td>0.000</td>
</tr>
<tr>
<td>Probability of divergent path 3</td>
<td>0.940</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Table 2: Measured performance of three kernels. Thread divergence data.
Kernel functions  | *AdvanceNeuronDevice* | *AdvanceSynapseDevice* | *calcSummationMap*
--- | --- | --- | ---
# bytes read in path 1 | 36  | 113  | 588  |
# bytes written in path 1 | 16  | 44   | 8    |
# bytes read in path 2 | 73  | 25   | 0    |
# bytes written in path 2 | 37  | 12   | 0    |
# bytes read in path 3 | 80  | 0    | 0    |
# bytes written in path 3 | 16  | 0    | 0    |
Average # read/written bytes | 93  | 37   | 596  |
Effective bandwidth (GB/s) | 5.7 | 27.2 | 11.3 |

Table 3: Measured performance of three kernels. Number of bytes read and written is show for each kernel’s divergent pathways; probabilities of each pathway from table 2 was used to compute average number of bytes and effective memory bandwidth.

firing rate in a manner similar to that for the neurons. The *calcSummationMap()* kernel was not divergent.

### 4.3 Effective Memory Bandwidth

Table 3 shows the number of read/written bytes and effective bandwidth of each kernel function. The average number of bytes read/written, $B_{rw}$, was calculated as:

$$B_{rw} = \sum_{i=1}^{D} p_i B_{rw,i}$$  \hspace{1cm} (11)

where, $p_i$ was the probability of divergence path $i$, $D$ the total number of divergent paths for that kernel, and $B_{rw,i}$ the number of bytes read/write in divergent path $i$. The effective bandwidth was:

$$B_e = \frac{B_{rw} n_{ite} n_{thread}}{t_{cum}}$$  \hspace{1cm} (12)

where, $n_{ite}$ was the number of iterations, $n_{thread}$ was the number of threads, and $t_{cum}$ was the cumulative execution time.

### 4.4 Effective MIPS

Table 4 shows the number of instructions and effective MIPS of each kernel function. The average number of instructions, $I$, was calculated in a manner analogous to that for $B_{rw}$ in equation (11). The effective MIPS, $\text{MIPS}_e$, for each kernel was calculated as:

$$\text{MIPS}_e = \frac{I n_{ite} n_{thread}}{t_{cum}}$$  \hspace{1cm} (13)

### 5 Discussion

Theoretical peak performance of a device is a combination of memory bandwidth and computing resources. The Tesla C1060 is theoretically capable of 933.12GFLOPs (single precision) of processing performance and 102.4GB/s bandwidth, which can be calculated from its hardware specifications [1]. For example, the Tesla C1060 uses DDR (double data rate) RAM with a memory clock rate of 800 MHz and a 512-bit wide memory interface, so the peak theoretical memory bandwidth is:

$$B_p = \frac{800 \times 10^6 \times \left( \frac{512}{8} \right) \times 2}{10^9} = 102.4\,$$ GB/s \hspace{1cm} (14)

The maximum throughput of a streaming processor (SP) is one instruction per clock when the pipeline of the SP is full [7]. Therefore, the peak theoretical MIPS is:

$$\text{MIPS}_p = \frac{n_{mp} n_{sp} r}{10^6} = \frac{30 \times 8 \times 1.296}{10^6} = 311,040\,$$ MIPS \hspace{1cm} (15)

where $n_{mp}$ is the number of multiprocessors, $n_{sp}$ is the number of streaming processors per multiprocessor, and $r$ is the clock rate in GHz.

Because the C1060 card can dual-issue a multiply concurrent with a multiply-add, giving a 50% boost to the theoretical max, the peak theoretical FLOPS (single precision) speed is:

$$\text{FLOPS}_p = 2 \times 1.5 \times \text{MIPS}_p = 932.12\,$$ GFLOPS \hspace{1cm} (16)

The ratio of effective MIPS to theoretical peak MIPS in table 5 reflects the utilization efficiency of the streaming multiprocessor (SM) execution units. When all the execution units in the CUDA device are fully utilized, the device can achieve its peak performance. There are some factors that limit execution unit utilization efficiency:

1. There need to be enough instructions between memory accesses, and enough warps issuing
those instructions, to hide the long memory access latency, and therefore to maximize the utilization of execution units. When this is not the case, execution units sit idle while awaiting memory access completion. Therefore, the global memory to instruction cycle ratio measures the degree of memory access frequency. A lower ratio will result in higher utilization efficiency.

2. Occupancy determines the number of warps per streaming multiprocessor (SM), and therefore the number of threads that can issue instructions “simultaneously”. Low occupancy decreases utilization efficiency.

3. Thread divergence, where the execution of threads in a warp takes different paths sequentially, lowers utilization efficiency.

4. The kernel functions `advanceNeuronDevice()` and `calcSummationMap()` don’t utilize all streaming multiprocessors (SMs) due to network size. The kernel function `advanceNeuronDevice()` only uses 20 streaming multiprocessors (SMs) out of 30 SMs (2/3 utilization ratio), and the kernel function `calcSummationMap()` only uses 10 streaming multiprocessors (1/3 utilization ratio).

5. Furthermore, there are some other factors that seem to lower actual utilization efficiency more than our analysis above:

   (a) We generated PTX files and counted the number of instructions in the assembly level code. However, the PTX code is not a sufficiently precise representation of the actual machine instructions executed on the GPU, as it does not precisely correspond to its actual machine instructions.

   (b) Some instructions are executed by special function units (SFUs) or double-precision units (DPUs), which are lower throughput devices than SPs.

A number of these conclusions are relevant to any simulation of this type (networks of simple interacting components with behavior simulated via iteration in time). In particular, we can consider occupancy, instruction mix, and serial dependency.

The neuron and synapse models simulated here are not terribly complex when compared to other neural simulations. In particular, the leaky integrator neuron model is just about the simplest that one might use. This would seem to suggest that large numbers of neurons could be simulated at once, thereby increasing the number of threads and, as a result, increasing utilization efficiency. However, despite this relative model simplicity, register usage by `advanceNeuronDevice()` and `advanceSynapseDevice()` still precludes 100% occupancy due to resource limits.

The simplicity of these models leads to a low ratio of computation to control and communications. This prevents full utilization of highly pipelined SP execution units and causes SPs to sit idle while threads wait for memory accesses to complete. One common approach to dealing with this would be to make each thread responsible for more than one neuron or synapse. However, this would increase register usage by each thread, resulting in decreased occupancy. Reduction in number of threads would also decrease occupancy. Therefore, it seems likely that this would...
be a self-defeating approach.

The computational complexity of this class of algorithms derives both from the large number of objects (neurons, synapses) to be simulated and the large number of iterations (here hundreds of millions) to be performed. The former is amenable to parallelization speedup. However, because neural simulations stereotypically involve very simple computational elements, the latter imposes an upper limit on overall performance.

To sum up, despite convenient libraries that enable rapid development of GPU-enabled code, detailed knowledge of GPU architecture is still essential for achieving maximum performance. One important concept is utilization efficiency: making all execution units as busy as possible, and many performance-improving techniques of GPU programming are related to this idea. As seen in the table 5, the utilization efficiency of each kernel function varies widely. It is heavily dependent on the nature of the function, such that memory bound functions may show lower utilization efficiency. However, understanding the detailed architecture of the GPU and focusing on utilization efficiency does lead to distinctly better solutions.

References


