Simulating spiking neural networks on GPU

Romain Brette[1,2]
Dan F. M. Goodman[1,2]

[1] Laboratoire Psychologie de la Perception, CNRS and Université Paris Descartes, Paris, France, romain.brette@ens.fr

[2] Département d'Etudes Cognitives, Ecole Normale Supérieure, Paris, France, dan.goodman@ens.fr

August 14, 2012

Abstract

Modern graphics cards contain hundreds of cores that can be programmed for intensive calculations. They are beginning to be used for spiking neural network simulations. The goal is to make parallel simulation of spiking neural networks available to a large audience, without the requirements of a cluster. We review the ongoing efforts towards this goal, and we outline the main difficulties.

Keywords: simulation, graphics cards, GPU, spiking neural networks, algorithms, parallel computing

1 Introduction

Graphics processing units (GPUs) are chips available on modern graphics cards. They are inexpensive units designed originally and primarily for computer games, which are increasingly being used for non-graphical parallel computing (Owens et al., 2007). The chips contain multiple processor cores (3072 in the current state of the art designs[1]), which can be programmed with the SIMD paradigm: single instruction, multiple data. Recently, a number of developers have investigated the possibility of simulating spiking neural networks on GPUs (Bernhard and Keriven, 2006; Nageswaran et al., 2009a,b; Fidjeland et al., 2009; Fidjeland and Shanahan, 2010; Nowotny, 2011b; Fernandez et al., 2008; Hoffmann et al., 2010; Bhuiyan et al., 2010; Han and Taha, 2010b; Scorcioni, 2010; Han and Taha, 2010a; Yudanov et al., 2010; Mutch et al., 2010; Ahmadi and Soleimani, 2011; Wang et al., 2011; Igarashi et al., 2011). The hope is that users will then be able to simulate neural networks in parallel, with an efficiency that previously required a cluster.

Some of the authors mentioned above have made their code available as is, or as part of a software package. Notably, the NeMo library by Andreas Fidjeland (Fidjeland et al., 2009) and GeNN by Thomas Nowotny (Nowotny, 2011b) are fully featured C++ libraries for spiking neural network simulation on GPU. NeMo features a fixed set of neuron models to choose from, synaptic delays, an offline, nearest neighbour based STDP learning rule, and a sparse matrix structure allowing for large numbers of synapses. GeNN features code generation for arbitrary neuron models, but no synaptic delays or STDP, and uses a dense matrix structure which puts a memory limit on the maximum model size. The source code for Nageswaran et al. (2009b); Richert et al. (2011) is also available and can be adapted by users. This code features synaptic delays and a standard STDP rule, and makes use of a sparse matrix structure, however it is currently restricted to Izhikevich neurons. Finally, although it does not yet support general simulations on GPU, the Brian simulator (Goodman and Brette, 2008, 2009) supports parallelisation on single or multiple GPU systems in its model fitting toolbox (Rossant et al., 2010, 2011).

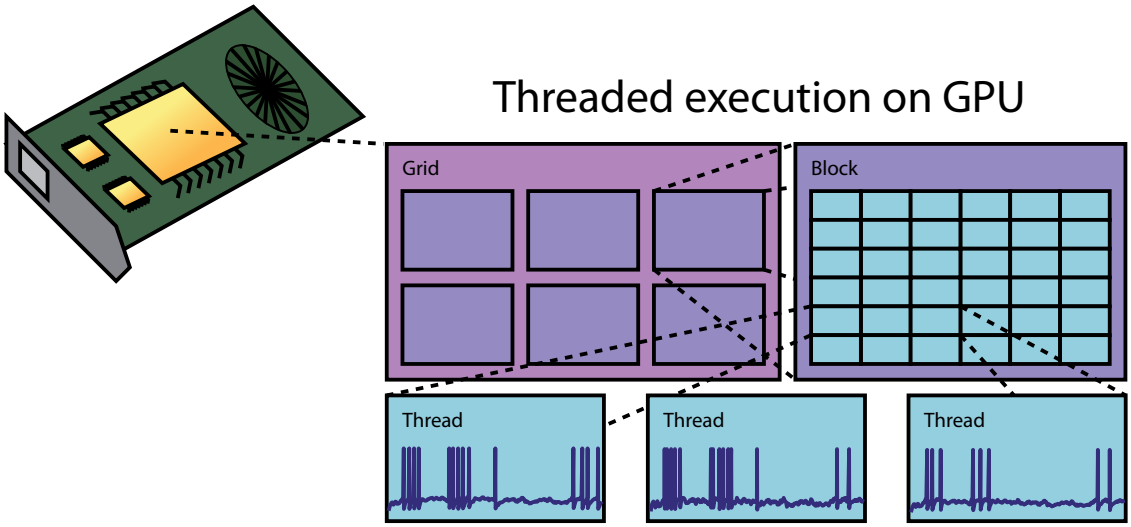

Figure 1: The GPU divides work into a single *grid* of multiple *blocks*, each block consisting of multiple *threads*. Memory consists of two main types, global memory which is large, slow and shared between all blocks, and shared memory which is small, fast and only shared between threads in a block. See section 2.1 for more details.

Programming for a GPU is rather specialised (figure 1, see section 2.1 for more details). Each processor core on a GPU is much simpler than a typical CPU, and this places considerable limitations on what programs can be written for them. Moreover, although recent versions of these chips allow more of the functionality of a full CPU (such as conditional branching), algorithms that do not take into account the architecture of the GPU will not use it efficiently. In particular, the GPU

[1] The GeForce GTX 690 consisting of dual Kepler GK110 GPUs. `http://www.geforce.com/whats-new/articles/article-keynote/`

places constraints on memory access patterns. Although 512 cores may be present in the GPU[2], it is unrealistic in most cases to expect a 512x speed increase over a CPU due to communication overheads (present in all parallel computing scenarios), and in particular on the constraints on memory access patterns (particular to GPU computing). In this note, we outline a number of general issues for neural network simulation on GPUs.

We start by reviewing general algorithmic issues in section 2. In general, the bottleneck for large scale simulations is the propagation of spikes across the network (section 2.3). To turn user-defined models into compilable GPU code, a number of developers are starting to use code generation, which we describe in section 3. In section 4, we review a related issue: tuning GPU algorithms to specific hardware and models.

2 Algorithms

A spiking neuron model is generally described as a hybrid system, that is, a set of differential equations that describe the continuous evolution of a number of state variables (membrane potential, conductances, ...), and discrete events: spikes. When a condition on the state variables is satisfied, for example when the membrane potential exceeds some threshold value, a spike is produced. When a spike is received, it produces a change in one or several variables, for example a synaptic conductance. The propagation of spikes from a presynaptic neuron to a postsynaptic neuron may occur with a conduction delay. Therefore, the simulation of spiking neural networks can be decomposed into three main parts:

1. Integrating the differential equations that describe the neuron models.
2. Propagating the spikes to target neurons.
3. Applying the effect of spikes on target neurons.

Algorithms for CPUs are reviewed in (Brette et al., 2007). Spiking neuron models can also be described in their integrated form, i.e., the membrane potential is a sum of kernels corresponding to each received spike, as in the Spike Response Model (Gerstner and Kistler, 2002). Note that these are not different types of model, only a different description. Here we focus on the differential description mentioned above, but specific simulation algorithms have also been developed for models described in the integrated form, including on GPU (Bohte and Slazynski, 2012). There are also cases in which neurons communicate with continuous signals (e.g. gap junctions), which we address in the discussion.

The number of operations for step 1 scales with the number of neurons. On GPUs, parallelizing the numerical integration (step 1) is straightforward, because it follows the SIMD paradigm (Single Instruction, Multiple Data). That is, the same operations are applied for all neurons that share the same model equations. This is the ideal case scenario for GPUs. We start by describing GPU algorithms for this step in section 2.2. The number of operations for steps 2 and 3 scales with the number of synapses. Therefore, for large networks, these two steps dominate the total computational cost (Brette et al., 2007). In addition, parallelizing spike propagation is more difficult, because it does not follow the SIMD paradigm. Therefore, the main bottleneck for GPU simulation is the propagation of spikes across the network, which we address in section 2.3. Before we address these specific algorithmic issues, we first describe the architecture of GPUs.

2.1 Architecture of GPUs

We only consider the architecture of NVIDIA GPUs in this paper, however other architectures (such as those of AMD) are very similar. The GPU divides work into a single *grid* of multiple *blocks*, each block consisting of multiple *threads* (figure 1). Memory consists of two main types that are relevant here, global memory which is large, slow and shared between all blocks, and shared memory which is small, fast and only shared between threads in a block. On the hardware level, the GPU is divided into multiple streaming multiprocessors (SMs). Each SM contains a fixed amount

[2]For example, the GeForce GTX 580. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580`

of shared memory, and several blocks can be simultaneously assigned to each SM if they do not use all of the available shared memory.

The most basic unit of parallelism is the *warp*, which is a group of 32 threads. Arithmetical instructions are always executed in a warp, and memory accesses are issued in half-warps (the first and last 16 threads of a warp). Threads *diverge* in their execution if different paths are taken at conditionals. Each warp executes the same instructions, and in the case of conditionals each branch is executed serially on a subset of the threads in that warp, and therefore it is important to minimise this divergence for high efficiency.

Global memory is accessed in 32, 64 or 128 byte chunks from half warps. If a single 4 bytes of memory is requested by the threads in a half warp, a 32 byte request would be issued (thereby using only 1/8th of the available bandwidth). However, if each thread in a half warp accesses 4 bytes, and these accesses are contiguous in memory, a single 64 byte memory request would be issued (using all the available bandwidth). The latter type of memory access is *coalesced*, and a key strategy for GPU programming is to layout data and design algorithms that ensure that as many memory transactions as possible are coalesced.

The hardware is optimised for 32-bit data, and single precision arithmetic is usually considerably faster than 64-bit arithmetic (8-24 times faster depending on the model, although this may be irrelevant if the bottleneck is memory access time). For devices of compute capability 2.0 or higher, there is also an L1 and L2 cache (whose size depends on the particular GPU architecture, but the maximum sizes for available NVIDIA GPUs are 48k and 768k respectively[3]).

2.2 Numerical integration

Algorithmically, numerical integration on GPU is straightforward as the problem is both "embarassingly parallel", that is each element to be computed in parallel can be computed independently of all the others, and entirely homogeneous, that is each computation for each neuron is identical. See section 3 for details on how code can automatically be generated for numerical integration on GPU. In addition, if we lay out the data in memory contiguously so that thread i (for neuron i) accesses memory at locations $x + i$, $y + i$, etc., then all memory accesses will be fully coalesced. However, this assumes that the numerical integration scheme uses a fixed step. For a variable time step integration scheme, if neurons i and $i + 1$ use different time steps they will execute different code, and this will cause *divergence* with a significant computational penalty, although there may be ways to work around this (see below). For a review of precision issues with fixed time step methods, see (Brette et al., 2007).

A feature of numerical integration on the GPU is that the limiting factor is often memory access time (latency and bandwidth) rather than compute time. That is, the processor is often waiting in an idle state for data to be transferred to and from global memory. While waiting for memory transfers, the GPU will attempt to execute other operations. However if the ratio of computation to communication is low, that is when performing a simple operation (just a few arithmetic operations) on a large amount of data, then there is not enough computational work to fill this waiting time. This is wasteful, and there are several ways to alleviate the problem. First of all, the amount of wasted processor time is less for more complicated neuron models, and indeed switching to a more complicated model can often have no impact on the computation time on GPU (although if the model is more complicated because it has more variables then this will not be true, as this implies greater data transfers). Another way to make use of wasted processor time is to use sub-steps to improve accuracy. For example, if the main simulation loop has a time step of $\mathrm{d}t$ then integration could be done with a timestep of $\mathrm{d}t/10$. If the processor was idle because it was waiting for memory, it can often be entirely free to use smaller time steps for integration. To some extent, this may even enable variable time step methods to be used, because the divergence will only affect the computation and not the memory access patterns. Alternatively, divergence in a variable time step method could be avoided by always using the smallest time step of each group of neurons in a warp (32 threads, the unit in which computational work is done in the GPU). This will mean that many neurons are integrated at time steps smaller than necessary, however it may well not push

[3]For devices of compute capability 2.0, 2.1 and 3.0, including all Fermi and Kepler architecture GPUs, there is 64 KB on-chip memory which can be allocated to shared memory and L1 cache in a 48/16 or 16/48 arrangement. The maximum L2 cache size for 2.x devices is 768 KB and for 3.0 it is 512 KB (NVIDIA 2012).

the computation time to the point where it is taking longer than the memory access time. Further, if there is some spatial structure to the neurons so that neurons with similar indices are receiving similar inputs, then this may not be such a problem in practice.

An important issue for numerical integration of stochastic differential equations is the parallel generation of independent random numbers. Previously, high quality random number generation on GPU has been a difficult issue, however the latest version (4.1) of the CUDA development toolkit includes random number generation using the Mersenne Twister algorithm as part of the cuRAND library.

2.3 Spike propagation

2.3.1 Parallelisation strategies

The propagation of spikes in the network is the most difficult problem in GPU simulation. A number of algorithms have been designed by several authors, optimized for particular cases (Nageswaran et al., 2009a; Fidjeland and Shanahan, 2010; Nowotny, 2011a).

There are essentially two ways to parallelise spike propagation. The first is to parallelise over neurons (Nageswaran et al., 2009a; Mutch et al., 2010). Each thread updates the total input of one neuron by checking whether any of the presynaptic neurons has spiked. This requires a number of operations at least equal to the total number of synapses in the network, executed at every timestep. The proportion of these operations that are actually useful is $F.\mathrm{d}t$, where F is the mean firing rate in the network and $\mathrm{d}t$ is the timestep. A typical example would be $F = 10$ Hz and $\mathrm{d}t = 0.1$ ms. In this case, only 1% of all neurons are active during any timestep, and therefore most operations executed by the kernel are not useful. In (Nageswaran et al., 2009a), this problem with sparse firing is addressed by a trick that consists in storing the boolean values corresponding to whether a presynaptic neuron has spiked in a bit array. The bit array is scanned by words of 32 bits, and each word is only examined further if it is non-zero. This saves a large number of operations.

The second way to parallelise is over synaptic events, that is, over received spikes. This is the approach taken in Nemo (Fidjeland et al., 2009). Each thread implements the effect of a spike arriving at one synapse. With this approach, there are fewer useless operations when firing is sparse. It was claimed that this strategy is a few times faster than the previous one in practical cases.

2.3.2 Memory issues

In both strategies, GPU kernels need to access a large amount of memory at each timestep, since all synaptic variables (e.g. weights) corresponding to received spikes must be accessed. More importantly, the ratio of numerical operations to memory operations is not high, because synaptic operations are often very simple (an addition). Since the speed of memory access is often the limiting factor in GPU applications, optimizing memory transfers is probably the most critical issue in spike propagation.

Let us consider the general situation, where conduction delays are heterogeneous. When a spike arrives at a synapse, it produces a change in a target variable, for example: $V \to V + w_{ij}$, where V is the membrane potential and w_{ij} is the synaptic weight between neurons i and j. This means loading the value of synaptic weight from memory and applying an operation. On a GPU, threads have access to a fast memory that is shared between all threads of a multiprocessor. However, this shared memory is very limited: up to 48KB in the latest boards. But clearly the number of synapses is much larger in many practical applications. It is possible to reduce the storage requirements when there are repeated patterns in the connectivity matrix (Mutch et al., 2010), but this is not a generic case. Therefore, synaptic weights have to be stored in global memory (up to 8GB). But global memory can be very slow, with a latency of 400-800 clock cycles, and bandwidth is also lower than for shared memory, while there are generally few arithmetical operations per memory access (only one in this example). This simple fact implies that the bottleneck of many neural simulations on GPU is memory, at least in networks with many synapses. More precisely, the bottleneck is the read/write memory access to the values of synaptic and, to a lesser extent, neuronal variables at the times of synaptic events.

How to maximize the speed of memory transfers? The key concept is coalescence (section 2.1). When a specific memory address in global memory is requested, an entire chunk of 32, 64 or 128 bytes is retrieved. This means that memory transfers are much faster if variables that are accessed at the same time are stored contiguously. This applies mainly to the second strategy outlined above, in which spike propagation is parallelised over received spikes. However, since the synaptic variables accessed during one timestep depend on the network model rather than on the specific implementation, maximizing coalescence may also be relevant for all strategies, because it also optimizes cache efficiency. We now outline a possible strategy to maximize coalescence.

2.3.3 Sorting synapses

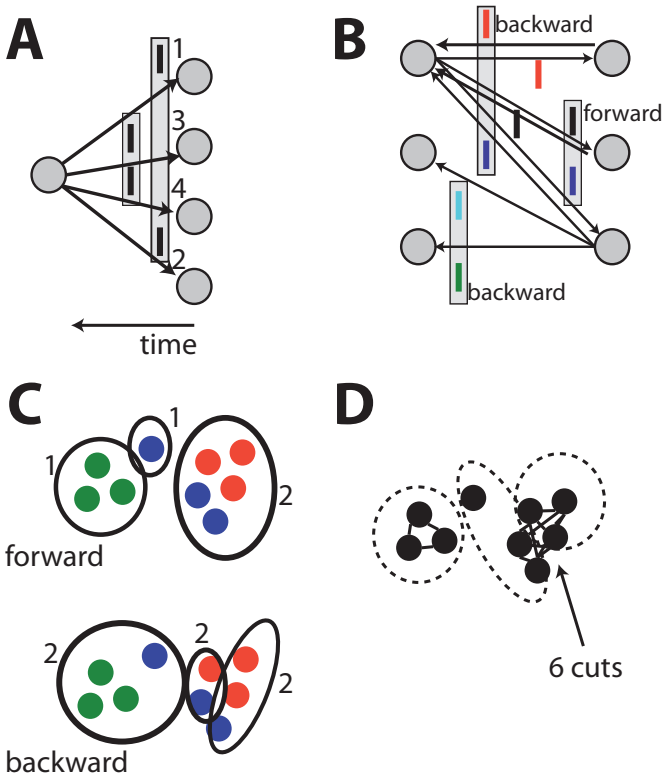

Figure 2: Synaptic propagation. A, A spike is transmitted to four target neurons with different conduction delays. Each rectangle contains spikes that arrive at the same time at the target synapses. Synaptic indices (numbers) are ordered so that synapses that are processed together are consecutively numbered. B, With STDP, spikes are also backpropagated. Spikes with the same color correspond to the same synapse, and therefore to a single synapse index. C, Synapses are colored according to their assigned memory chunk, i.e., all green synapses belong to the same memory chunk, etc. Synapses in the same group (circle) are simultaneously accessed. There is a forward and a backward set of synaptic groups. The number near each circle is the number of memory chunks represented in the synaptic group. D, A graph is built on the set of synapses by connecting synapses belonging to the same group (here, the forward groups). The graph is partitioned into memory chunks (dotted ellipses). A graph cut corresponds to simultaneous access of two memory chunks.

This strategy is exploited in the simulator Nemo (Fidjeland et al., 2009). All synapses made by the same presynaptic neuron with the same conduction delays will receive spikes at exactly the same time (figure 2A). Therefore, all the corresponding synaptic variables will be simultaneously accessed. To maximize coalescence, one simply sorts the synaptic indices so that synapses with the same presynaptic neuron and conduction delay have consecutive indices.

Although this strategy works well in many cases, it fails to generalize to the simulation of plastic synapses, unless synaptic weights are modified offline. In most long term synaptic plasticity rules (spike-timing-dependent plasticity, STDP), both presynaptic and postsynaptic spikes produce

changes on synaptic variables. This implies that each spike has to be propagated forward (from presynaptic neuron to synapses), but also backward (from postsynaptic neuron to synapses). For backward propagation, the aforementioned coalescence condition means that synaptic variables with the same postsynaptic neuron and backward conduction delay should reside in the same memory chunk. In Figure 2B, spikes with the same colour correspond to the same synapse, for forward (right) or backward (left) propagation. Vertically aligned spikes are delivered at the same time and are grouped together. The problem is that the same synapse may belong to two different groups of simultaneously accessed synapses, depending on whether the forward or backward direction is considered. For example, spikes coloured in blue correspond to a synapse that belongs to two different groups. Thus, the optimal indexing for a particular direction may be bad for the reverse direction.

Therefore, coalescence can only be approximate. The problem can be mathematically formulated in the following way (figure 2C). For each direction (forward and backward), we define a partition of the set of synapses into groups of simultaneously accessed synapses. That is, in the forward partition, all synapses with the same presynaptic neuron and the same forward conduction delay are grouped together. In the backward partition, all synapses with the same postsynaptic neuron and the same backward conduction delay are grouped together. Each synapse is assigned to a memory chunk. The cost of accessing the synaptic variables in a synaptic group is proportional to the number of chunks represented in that group. Under some homogeneity assumptions, the average cost is proportional to the sum of the costs of all groups. The goal is to find an assignment of synapses to chunks that minimizes this cost.

In this formulation, the problem is difficult to solve in general. One may express an approximation in terms of graph theory (figure 2D). We build a graph by connecting synapses that are simultaneously accessed (for the forward direction: same presynaptic neuron, same conduction delay). An assignment of synapses to memory chunks is a partition of the graph, with each set representing a chunk. We want synapses that are simultaneously accessed to belong to the same memory chunk. In other words, the partition should avoid *cutting* the edges of the graph. Finding a partition that minimizes the number of cuts in a graph is known as a *graph-cut* problem in computer science. One may build one graph for the forward direction and another one for the backward direction, and minimize the total number of cuts, or one may build a single graph by merging the forward and backward graphs.

3 Code generation

Writing code for GPUs is difficult and time consuming, however a significant fraction of the work is fairly routine and can in principle be automated.

Code generation has been extensively used in the systems biology community (Garny et al. 2008; Miller et al. 2010; see Goodman 2010 for a review). The first neural network simulator to use code generation techniques, for CPU, was Neuron (Carnevale and Hines, 2006). Users enter the definition of a neuron model in a domain specific language (DSL) called NMODL (Hines and Carnevale 2000; based on the earlier MODL language of SCoP, Kootsey et al. 1986). When the user compiles this NMODL definition, it is parsed and converted into C code, which is then compiled using a standard C compiler into a library which is dynamically loaded from the main program. This technique is heavily used in the Brian simulator (Goodman and Brette, 2008, 2009; Goodman, 2010) for generating both Python and C++ code from user-specified models given as differential equations in standard mathematical form, which is transparently generated, compiled and called when the user runs their script. There are currently efforts underway to generate simulation code from the NeuroML (Gleeson et al., 2010) and NineML (Raikov et al., 2011) model description languages.

Code generation has been proposed as particularly useful for GPUs in order to tune a program at runtime, as minor variants on the program can have surprisingly large effects on performance (see section 4, Klöckner et al. 2009 and Pinto and Cox 2011).

For neural network simulation, code generation has already been used in several existing projects. The Brian simulator uses code generation on GPU for its model fitting toolbox, which was able to give a 60x performance boost compared to CPU for this "embarassingly parallel" problem (Rossant et al., 2010, 2011). The GeNN (GPU enhanced Neural Networks) simulator (Nowotny, 2011b) allows the user to specify a neuron model by giving the set of variables, parameters and a numerical

integration step code fragment, and generates a CUDA kernel which is then compiled into a library which is linked to the main program.

In addition to the available software reviewed above, there are several efforts in development for neural network simulation on GPU which make use of code generation. The Brian simulator already contains experimental code for converting model specifications into GPU code which will be available in the next major release, and both GeNN and NeMo (Fidjeland et al., 2009) are working towards integrating the code generation framework from Brian into their simulators.

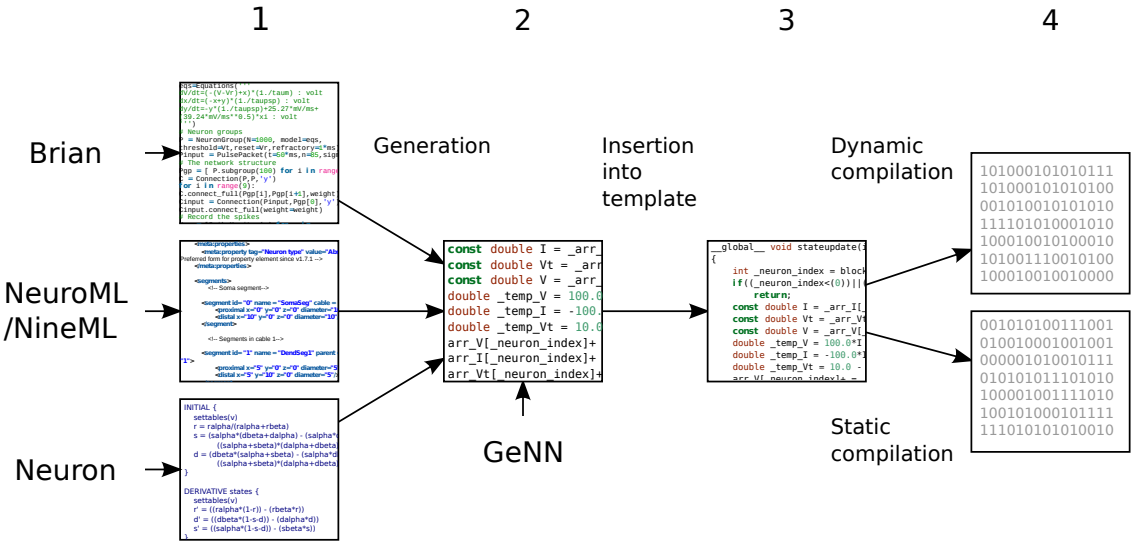

Figure 3: Steps involved in code generation. (1) High level, abstract representation such as equations in Brian, XML description in NeuroML/NineML, or NMODL file in Neuron. (2) Code fragment generated from high level representation, or provided by user (GeNN), to perform a numerical integration step. (3) Code inserted into a template file to generate a compilable source file. (4) Executable object either compiled statically (Neuron, GeNN) or dynamically (Brian).

Neural network simulation primarily consists of two phases, numerical integration of differential equations (section 2.2) and propagation of spikes via synapses (section 2.3). Most of the software mentioned above uses code generation mainly for integration and not for propagation (although Brian and NeMo aim to use code generation for propagation too). There are several different approaches taken, although they are all fairly similar. The process consists of four steps illustrated in figure 3: (1) a model specification is given by the user, (2) this is then converted into code to perform a single numerical integration step, (3) this is inserted into a template to form a complete compilable source file, and (4) the source file is compiled and run. Neuron, Brian, NeuroML and NineML all share the same approach to step (1), namely giving the model specification in a high level manner (although each has a different high level representation). GeNN differs by requiring the user to directly specify the code for the numerical integration step in (2), although an extension is being developed to convert models in Brian's specification format to GeNN format. Neuron and GeNN currently perform step (4) statically, that is generation and compilation of source code for a model is performed in one step, and this is then linked to the main program which can be run in a second step. By comparison, Brian uses runtime code generation, that is the code is generated, compiled and run automatically when the user runs their Brian script. Efficiency is maintained by caching the compiled code and only recompiling when the model specification changes.

4 Tuning algorithms to hardware and model

It is common when programming for GPUs to find that minor changes in the algorithm can have large effects on the running time, and that the optimal algorithm for a given set of data can also vary enormously based on the data in ways which are difficult to predict. Consequently, it is very important to test multiple algorithms and profile them to determine which is the most efficient. The software package PyCUDA (Klöckner et al., 2012) is designed to facilitate this process, allowing

runtime generation, compilation and running of GPU code. This process could also be automated, generating variants of a program at runtime and dynamically adapting the program to the data using runtime profiling. A similar approach has proved successful for the Java language, where Just-In-Time (JIT) compilation techniques have allowed this interpreted language to run almost as fast as or in some cases even faster than compiled code (Bull et al., 2001).

In figure 4 we demonstrate the drastic effect that algorithmic variations can have on a GPU program for different data sets. We present pseudocode for these algorithms, however they are relatively naïve and the results are included here for illustrative purposes only. As can be seen, the optimal algorithm is not easily predictable from the data even in this simple case where only two parameters vary, and the differences between the fastest and slowest algorithms can be substantial.

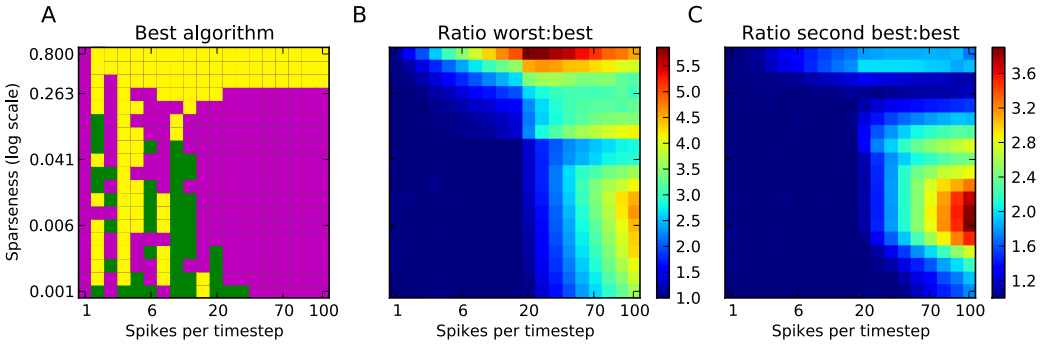

Figure 4: The differences between algorithmic variants for different parameters. The simulated network is a two layer feedforward network with 100 abstract neurons in the source layer which fire spikes with a Poisson distribution with rate set to generate the given total number of spikes per time step of 0.1ms, and 10^5 neurons in the target layer (which consists of null neurons which do nothing except to receive their inputs). This network is designed purely to illustrate the costs of propagating spikes, and not the costs of simulating actual neuron models. The parameters are the sparseness of the connectivity matrix between layers (y-axis) and the number of spikes generated in the first layer per time step (x-axis). Simulations were run on an NVIDIA GeForce GTX 580. (A) The best algorithm indicated by different colours. Algorithms are described in text: VPO (green), VSN (yellow) and VSSN (purple). There are three distinct regions, in the lower left region all algorithms perform roughly equally (see panel B). In the upper region, VSN performs best, and in the lower right region VSSN performs best. The optimal algorithm is not a simple function of these two parameters. (B) The ratio between the computation time of the slowest and fastest algorithms, showing that the choice of algorithm makes a large difference in some cases. (C) The ratio between the second fastest and fastest algorithms. This ratio is high in parts of both the regions where VSN and VSSN are identified as the best algorithm, showing that these two algorithms do not perform similarly well in these areas. This shows that all algorithms perform badly in one region of the parameter space.

Briefly, the algorithms are as follows (pseudocode is given in appendix A):

- Vectorisation over postsynaptic offset (VPO). Each thread loops over the spiking neurons, and thread i propagates the synaptic weight to the ith target neuron of the spiking neuron. The connectivity matrix is stored as a CSR sparse matrix, so reading synaptic targets and weights is coalesced, however writing to the target is uncoalesced and uses an atomic operation which will be slow when each target neuron is receiving many synaptic inputs. The number of threads is equal to the maximum number of postsynaptic targets over all the presynaptic neurons, so in order to make efficient use of the GPU this number should be large, and in order not to waste threads, the number of postsynaptic targets should be fairly homogeneous across the presynaptic neurons. This algorithm performs poorly in general, but does as well as the other two algorithms in the region where uncoalesced memory accesses are unavoidable (i.e. the region where the number of active synapses per timestep is low).

- Vectorisation over synaptic index and neuron index (VSN). This algorithm consists of two phases. In the first phase, the modification of target state variables is computed in a shared

memory buffer. In this stage, the thread index i is used as a synaptic index. Each thread loops over the spiking neurons, and applies the corresponding synaptic operation (typically an addition of a weight). In the second phase, this shared memory buffer is added to the target state variables in global memory in a single coalesced operation. In this phase, the thread index i is used as a neuron index in the target group. These two phases are combined in a single kernel, because the shared memory is lost between kernel launches. The connectivity matrix here is stored as a CSR sparse matrix broken into columnar chunks of width the block size, and so again reading synaptic targets and weights is coalesced. Like the previous algorithm, the first phase also uses an atomic operation but this time it operates on shared memory which is much faster than global memory.

- Vectorisation over spikes, synaptic index and neuron index (VSSN). The basic algorithm is the same as VSN, with the first phase slightly modified. In this phase, if the number of synapses in a given columnar block is substantially lower than the size of the block, then in order to waste fewer threads, several spikes are processed in the same block. This algorithm performs significantly better than the VSN algorithm unless the connectivity matrix is fairly dense in which case the additional overheads of packing the loop into the first phase outweigh the benefits.

The full code, including the code for figure 4 is available at `http://neuralensemble.org/trac/brian/browser/trunk/dev/ideas/cuda/propagation?rev=3213`.

5 Discussion

Because graphics cards are cheaper and easier to maintain than clusters, there is growing interest in the development of neural network simulation on GPUs. In this note, we have reviewed the ongoing efforts in the community and outlined a number of critical issues. From an algorithmic point of view, the main issue is the efficient propagation of spikes across the network. The bottleneck is memory rather than pure computation speed, because global memory access is slow on GPUs. From a usability point of view, the main issue is to turn user-defined models into runnable GPU code. Efficiently programming GPUs is tricky, and this task should not be left to the users. A general strategy is *code generation*, that is, the automatic generation of compilable code from a model description. Since there are many boards with different specifications and limitations, and also models with different types of connectivity, code generation and model inspection may also allow the simulator to pick the most efficient algorithm for a particular combination of hardware and model.

We have restricted this discussion to the simulation of spiking neuron models, where interactions between neurons are mediated by discrete events (spikes). Other technical issues arise if gap junctions (electrical synapses) are considered. In this case, neurons interact continuously rather than at discrete times. This requires different algorithms for GPUs, but these are in principle less challenging than spike propagation algorithms. Indeed, simulating continuous interactions essentially amounts to calculating matrix products, for which there are standard GPU algorithms (Bell and Garland, 2009). Another important point that we have not addressed is compartmental modelling, that is, the simulation of neurons with a spatial extent. Such simulations usually rely on *compartmentalization*, that is, the neuron's morphology is divided into a number of isopotential compartments that are electrically coupled (Mascagni and Sherman, 1989). The same algorithms can be used when the number of compartments is small. However, realistic simulations with hundreds of compartments require specific techniques. Electrical propagation along an axon or a dendrite is described by the cable equation, a non-linear elliptic partial differential equation. With most techniques, the numerical integration is separated in two steps: the integration of the nonlinear equations for the channel variables, and the integration of the linear partial differential equation describing the electrical diffusion across the neuron. Because channel variables are independent, the first step is straightforward to parallelize. The second step is more difficult. Simulating the cable equation on a branch amounts to solving a tridiagonal linear system, for which a number of algorithms have been implemented on GPU (Zhang et al., 2010). Solutions for different branches can be merged with the domain decomposition method (Mascagni, 1991).

Although neural network simulation on GPUs is still in its infancy, we believe that users should

benefit from these techniques in the next few years. It will make parallel simulation of neural networks available to a large audience.

A Pseudocode

Pseudocode for the algorithms listed in section 4 is given below.

Vectorisation over postsynaptic offset (VPO):

```
for synaptic_index = 0 to max_num_synapses-1 in parallel:
    for neuron in spiking_neurons:
        if synaptic_index<num_synapses[neuron]:
            data_offset = row_offset[neuron]+synaptic_index
            target_index = target_indices[data_offset] # coalesced
            weight = weights[data_offset] # coalesced
            atomicAdd(target_variable_ptr+target_index, weight) # uncoalesced
```

Vectorisation over synaptic index and neuron index (VSN):

```
# block_size is the number of threads in a block for this kernel launch
# stage is an array of size block_size in shared memory
for target_index = 0 to num_target_neurons-1 in parallel:
    thread_index = target_index % block_size
    block_index = target_index / block_size
    stage[thread_index] = 0.0
    __syncthreads() # wait until all threads reach here before continuing
    for neuron in spiking_neurons:
        block_offset = neuron*(num_blocks+1)+block_index
        data_offset = row_offset[block_offset]+thread_index
        if data_offset<row_offset[block_offset+1]:
            # coalesced
            stage_index = target_indices[data_offset] % block_size
            weight = weights[data_offset]
            # uncoalesced, but in shared memory
            atomicAdd(stage+stage_index, weight)
    __syncthreads()
    # coalesced
    target_variable[target_index] += stage[thread_index]
```

Vectorisation over spikes, synaptic index and neuron index (VSSN):

```
# block_size is the number of threads in a block for this kernel launch
# stage is an array of size block_size in shared memory
for target_index = 0 to num_target_neurons-1 in parallel:
    thread_index = target_index % block_size
    block_index = target_index / block_size
    spikes_per_block = block_size/max_num_synapses
    stage[thread_index] = 0.0
    __syncthreads()
    for spike_block = 0 to num_spikes/spikes_per_block-1:
        spike_index = spike_block*spikes_per_block+thread_index/max_num_synapses
        neuron = spiking_neurons[spike_index]
        block_offset = neuron*(num_blocks+1)+block_index
        data_offset = row_index[block_offset]+thread_index%max_num_synapses
        if data_offset<row_index[block_offset+1]:
            # coalesced
            stage_index = target_indices[data_offset] % block_size
            weight = weights[data_offset]
            # uncoalesced, but in shared memory
            atomicAdd(stage+stage_index, weight)
```

```
__syncthreads()
# coalesced
target_variable[target_index] += stage[thread_index]
```

Acknowledgments

We thank Marcel Stimberg for his suggestion of formulating the optimisation of coalescence as a graph-cut problem, and the participants of the 2012 Workshop on Neuronal GPU Computing for their input. This work was supported by the European Research Council (ERC StG 240132) and the Agence nationale de la recherche (ANR-11-BSH2-0004).

References

Ahmadi, A. and H. Soleimani (2011, May). A GPU based simulation of multilayer spiking neural networks. In *Electrical Engineering (ICEE), 2011 19th Iranian Conference on*, pp. 1 –5.

Bell, N. and M. Garland (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, pp. 18:1–18:11. ACM.

Bernhard, F. and R. Keriven (2006). Spiking neurons on GPUs. In V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra (Eds.), *Computational Science ICCS 2006*, Volume 3994 of *Lecture Notes in Computer Science*, pp. 236–243. Springer Berlin / Heidelberg.

Bhuiyan, M., V. Pallipuram, and M. Smith (2010, April). Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1 –8.

Bohte, S. and L. Slazynski (2012). Streaming parallel gpu acceleration of large-scale filter-based spiking neural networks. *Network*, in press.

Brette, R., M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. M. Bower, M. Diesmann, A. Morrison, P. H. Goodman, F. C. Harris, M. Zirpe, T. Natschläger, D. Pecevski, B. Ermentrout, M. Djurfeldt, A. Lansner, O. Rochel, T. Vieville, E. Muller, A. P. Davison, S. E. Boustani, and A. Destexhe (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience 23*, 349–98.

Bull, J. M., L. A. Smith, L. Pottage, and R. Freeman (2001). Benchmarking Java against C and Fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, Palo Alto, California, United States, pp. 97–105. ACM.

Carnevale, N. T. and M. L. Hines (2006). *The NEURON Book*. Cambridge University Press.

Fernandez, A., R. San Martin, E. Farguell, and G. Pazienza (2008, July). Cellular neural networks simulation on a parallel graphics processing unit. In *Cellular Neural Networks and Their Applications, 2008. CNNA 2008. 11th International Workshop on*, pp. 208 –212.

Fidjeland, A., E. Roesch, M. Shanahan, and W. Luk (2009, July). NeMo: a platform for neural modelling of spiking neurons using GPUs. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pp. 137 –144.

Fidjeland, A. and M. Shanahan (2010, July). Accelerated simulation of spiking neural networks using GPUs. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1 –8.

Garny, A., D. P. Nickerson, J. Cooper, R. W. dos Santos, A. K. Miller, S. McKeever, P. M. F. Nielsen, and P. J. Hunter (2008, September). CellML and associated tools and techniques. *Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences 366*(1878), 3017–3043. PMID: 18579471.

Gerstner, W. and W. M. Kistler (2002). *Spiking Neuron Models*. Cambridge University Press.

Gleeson, P., S. Crook, R. C. Cannon, M. L. Hines, G. O. Billings, M. Farinella, T. M. Morse, A. P. Davison, S. Ray, U. S. Bhalla, S. R. Barnes, Y. D. Dimitrova, and R. A. Silver (2010, June). NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comput Biol 6*(6), e1000815.

Goodman, D. and R. Brette (2008). Brian: A simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics 2*, 5.

Goodman, D. F. M. (2010, September). Code generation: A strategy for neural network simulators. *Neuroinformatics 8*(3), 183–196.

Goodman, D. F. M. and R. Brette (2009). The Brian simulator. *Frontiers in Neuroscience 3*(2), 192–197.

Han, B. and T. Taha (2010a, July). Neuromorphic models on a GPGPU cluster. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1 –8.

Han, B. and T. M. Taha (2010b, April). Acceleration of spiking neural network based pattern recognition on NVIDIA graphics processors. *Applied Optics 49*(10), B83–B91.

Hines, M. L. and N. T. Carnevale (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation 12*(5), 995–1007.

Hoffmann, J., K. El-Laithy, F. Gttler, and M. Bogdan (2010). Simulating Biological-Inspired spiking neural networks with OpenCL. In K. Diamantaras, W. Duch, and L. Iliadis (Eds.), *Artificial Neural Networks ICANN 2010*, Volume 6352 of *Lecture Notes in Computer Science*, pp. 184–187. Springer Berlin / Heidelberg.

Igarashi, J., O. Shouno, T. Fukai, and H. Tsujino (2011, November). Real-time simulation of a spiking neural network model of the basal ganglia circuitry using general purpose computing on graphics processing units. *Neural networks: the official journal of the International Neural Network Society 24*(9), 950–960. PMID: 21764258.

Klöckner, A., N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih (2009, November). PyCUDA: GPU Run-Time code generation for High-Performance computing. *0911.3456*.

Klöckner, A., N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih (2012, March). PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. *Parallel Computing 38*(3), 157–174.

Kootsey, J. M., M. C. Kohn, M. D. Feezor, G. R. Mitchell, and P. R. Fletcher (1986). SCoP: an interactive simulation control program for micro- and minicomputers. *Bulletin of Mathematical Biology 48*(3-4), 427–441.

Mascagni, M. (1991, January). A parallelizing algorithm for computing solutions to arbitrarily branched cable neuron models. *Journal of Neuroscience Methods 36*(1), 105–114.

Mascagni, M. V. and A. Sherman (1989). Numerical methods for neuronal modeling. In *In Methods in Neuronal Modeling*, pp. 439484. MIT Press.

Miller, A., J. Marsh, A. Reeve, A. Garny, R. Britten, M. Halstead, J. Cooper, D. Nickerson, and P. Nielsen (2010). An overview of the CellML API and its implementation. *BMC Bioinformatics 11*(1), 178.

Mutch, J., U. Knoblich, and T. Poggio (2010, February). CNS: a GPU-based framework for simulating cortically-organized networks. Technical Report MIT-CSAIL-TR-2010-013 / CBCL-286, Massachusetts Institute of Technology, Cambridge, MA.

Nageswaran, J. M., N. Dutt, J. L. Krichmar, A. Nicolau, and A. Veidenbaum (2009a). Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. In *Proceedings of the 2009 international joint conference on Neural Networks*, Atlanta, Georgia, USA, pp. 3201–3208. IEEE Press.

Nageswaran, J. M., N. Dutt, J. L. Krichmar, A. Nicolau, and A. V. Veidenbaum (2009b, August). A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks 22*(56), 791–800.

Nowotny, T. (2011a). Flexible neuronal network simulation framework using code generation for NVidia CUDA. *BMC Neuroscience 12*(Suppl 1), P239.

Nowotny, T. (2011b). GeNN. http://sourceforge.net/projects/genn/.

NVIDIA (2012). CUDA programming guide.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell (2007). A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, Volume 26, pp. 80–113. Citeseer.

Pinto, N. and D. Cox (2011). GPU meta-programming: A case study in biologically-inspired machine vision. In W. H. Wen-mei (Ed.), *GPU Computing Gems Jade Edition*, Volume 2 of *Applications of GPU Computing*. Morgan Kaufmann.

Raikov, I., R. Cannon, R. Clewley, H. Cornelis, A. Davison, E. De Schutter, M. Djurfeldt, P. Gleeson, A. Gorchetchnikov, H. E. Plesser, S. Hill, M. Hines, B. Kriener, Y. Le Franc, C. Lo, A. Morrison, E. Muller, S. Ray, L. Schwabe, and B. Szatmary (2011, July). NineML: the network interchange for neuroscience modeling language. *BMC Neuroscience 12*(Suppl 1), P330. PMID: null PMCID: PMC3240446.

Richert, M., J. M. Nageswaran, N. Dutt, and J. L. Krichmar (2011). An efficient simulation environment for modeling large-scale cortical processing. *Frontiers in Neuroinformatics 5*, 19.

Rossant, C., D. F. M. Goodman, B. Fontaine, J. Platkiewicz, A. K. Magnusson, and R. Brette (2011). Fitting neuron models to spike trains. *Frontiers in Neuroscience 5*, 9.

Rossant, C., D. F. M. Goodman, J. Platkiewicz, and R. Brette (2010). Automatic fitting of spiking neuron models to electrophysiological recordings. *Frontiers in Neuroinformatics*.

Scorcioni, R. (2010, May). GPGPU implementation of a synaptically optimized, anatomically accurate spiking network simulator. In *Biomedical Sciences and Engineering Conference (BSEC), 2010*, pp. 1 –3.

Wang, M., B. Yan, J. Hu, and P. Li (2011, August). Simulation of large neuronal networks with biophysically accurate models on graphics processors. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pp. 3184 –3193.

Yudanov, D., M. Shaaban, R. Melton, and L. Reznik (2010, July). GPU-based simulation of spiking neural networks with real-time performance amp; high accuracy. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1 –8.

Zhang, Y., J. Cohen, and J. D. Owens (2010, January). Fast tridiagonal solvers on the GPU. *SIGPLAN Not. 45*(5), 127136.