Final, peer-reviewed manuscript published in Network: Computation in Neural Systems, 23(4): 212-236.

Comparison of GPU- and CPU-implementations of mean-firing rate neural networks on parallel hardware

Helge Ülo Dinkelbach, Julien Vitay, Frederik Beuth and Fred H. Hamker*

Department of Computer Science, Artificial Intelligence, Chemnitz University of Technology, Germany

*Corresponding author: Fred Hamker (fred.hamker@informatik.tu-chemnitz.de), TU Chemnitz, Straße der Nationen 62, D-09107 Chemnitz, Germany.

Abstract

Modern parallel hardware such as multi-core processors (CPUs) and graphics processing units (GPUs) have a high computational power which can be greatly beneficial to the simulation of large-scale neural networks. Over the past years, a number of efforts have focused on developing parallel algorithms and simulators best suited for the simulation of spiking neural models. In this article, we aim at investigating the advantages and drawbacks of the CPU and GPU parallelization of mean-firing rate neurons, widely used in systems-level computational neuroscience. By comparing OpenMP, CUDA and OpenCL implementations towards a serial CPU implementation, we show that GPUs are better suited than CPUs for the simulation of very large networks, but that smaller networks would benefit more from an OpenMP implementation. As this performance strongly depends on data organization, we analyze the impact of various factors such as data structure, memory alignment and floating precision. We then discuss the suitability of the different hardware depending on the networks' size and connectivity, as random or sparse connectivities in mean-firing rate networks tend to break parallel performance on GPUs due to the violation of coalescence.

Keywords

Neural computation, neural simulator, parallel computing, OpenMP, CUDA, OpenCL, GPU, GPGPU.

1. Introduction

The use of multi-core processors (CPUs) and graphics processing units (GPUs) is a very promising tool to support the development of large-scale neural networks in computational neuroscience. Research has most recently focused on algorithms and simulators best suited for spiking neural models, such as NEST (Gewaltig and Diesmann 2007), Nemo (Fidjeland et al. 2009, Fidjeland and Shanahan 2010) or Brian (Goodman and Brette 2009). The high speedups achieved by these parallel simulators rely partly on specific properties of these networks, such as the rather sparse firing patterns, or the use of specific types of synapses, which allow for a spike propagation mechanism that reduces intercommunication between neurons. However, general purpose simulators should offer the researcher a high degree of flexibility to explore neural function by supporting not only various models of neural response generation but also of synaptic plasticity and the updating of time-dependent variables which are not spike-driven, but follow ordinary differential equations (ODE). Accordingly, we focus in this article on one type of neuron relying solely on the integration of ordinary differential equations.

This type of neuron, mean-firing rate or rate-coded neuron, is widely used in systems-level computational neuroscience (e.g. Schroll et al. 2012; Ziesche and Hamker 2011; Dranias et al. 2008; Frank and O'Reilly 2006). Although rate-coded neurons do not aim at reproducing spiking activity, their rich network dynamics and available powerful learning rules make them an important tool for modeling complex cognitive functions. Only a few neural simulators have addressed recently the parallel simulation of mean-firing rate neural networks, such as NEST (Gewaltig and Diesmann 2007) or Emergent (Asia et al. 2008).

Outside simulators, most research has focused on the GPU acceleration of specialized architectures of static artificial neurons, such as multi-layer perceptrons (Oh and Jung 2004), convolutional networks (Chelapilla et al. 2006, Mutch et al. 2010), radial-basis function networks (Brandstetter and Artusi 2008), Kohonen maps (Zhongwen et al. 2005) or even cortical models (Nere and Lipasti 2010). These studies report faster simulation on GPUs than on CPUs, but the parallel implementation is very specific for the structure of the networks. Prior knowledge on the connectivity of a network allows to optimize efficiently the memory access patterns. In this article, we aim at studying what happen when this structure is *a priori* unknown.

Seifert (2002) provided two conditions for the successful use of parallel computer hardware: first the computational problem must be capable of parallelization, meaning that the major load of the computations have to be local and not rely on global operations such as computing the mean or maximum value of activities in a population; second the underlying hardware must be reflected in the programming paradigm to avoid incorrect thread organization and other potential bottlenecks.

To examine this condition, Hong and Kim (2009) presented a very detailed analytical model to determine performance depending on memory requests and the number of threads used. However, this model does not consider the cost of cache misses and branches. A more recent approach by Sim et al. (2012) addresses additionally the bottlenecks resulting from shared memory and *special function unit* (SFU) accesses. These analytical models are specifically designed for the analysis of a particular implementation of a problem and the results lack generalization, therefore we do not focus on them in this paper.

Nowotny et al. (2011) studied the parallel implementation of mean-firing rate networks using a particular architecture, a three-layer image classifier with different connection patterns between the layers. By comparing OpenMP and CUDA implementations, they show that GPU computing of mean-firing rate networks leads to significant but relatively small speedups when compared to what is observed for spiking networks. In addition to confirming these results, we aim at revealing the influence of several other factors, including the effect of connectivity patterns on coalesced access, by using a more generic framework making no assumption about the structure of the network.

In this article, we highlight some speed limiters in the parallel simulation of large-scale neural networks and propose potential solutions to avoid them. In section 2, we briefly present the existing parallel computation frameworks. In section 3, we present the key concepts in the parallel simulation of neural networks and compare the methods available for spiking and mean-firing-rate neural models. Section 4 will present several considerations on the data structures that have to be used in order to exploit optimally the parallel hardware, regarding current work on data structures and evaluation models (Brette et al. 2011, Siegel et al. 2011). Section 5 presents the methods we used to obtain the results presented in section 6. In this result section, we take a close look on the different speed limiters: floating point precision, thread configuration, data organization and connectivity structure. We show that for networks composed of a huge number of neurons or connections, GPU implementations are significantly faster than a serial CPU implementation, but smaller networks benefit more from a parallel CPU implementation, mostly because of cache effects. We finally investigate the effect of sparse or random connectivity patterns on performance, and discuss the suitability of GPU implementations for generic networks.

2. Background on CPU/ GPU computing

As we present and compare implementations using the parallel computing frameworks CUDA, OpenCL and OpenMP, we first introduce the frameworks and some background to GPU hardware.

2.1. OpenMP programming model

OpenMP (Chapman et al. 2011) is a language extension for C, C++ and Fortran which provides a set of compiler directives for an easier implementation of shared memory parallel programs for CPUs. Most of the currently available compilers are capable of OpenMP and provide it through compiler flags. Already existing serial code is reusable, so in general no explicit parallel programming or attention is required. OpenMP distinguishes two teams: one *master* and multiple *slaves*. The master is in control of the initial thread and executes sequentially the code until a parallel block (e. g. *#pragma omp parallel for*) is reached. If the master thread reaches an OpenMP parallel directive, several slave threads are created. Each thread executes the corresponding code within the parallel block. After the last slave finishes its job, the master destroys the slaves and continues. Next to the number of slave threads, the number of elements computed per thread is controllable, for example each slave thread can be responsible for five or fifty elements in a for-loop over thousand elements.

2.2. GPU architecture

As mentioned in the introduction, a minimal understanding of the underlying hardware, especially on GPUs, is needed for a successful parallel implementation. We now present a short overview on the hardware we used, the NVIDIA Tesla C2050.

Hardware organization

The Tesla C2050 consists of 14 *streaming multiprocessors* (*SMs*), each comprising 32 CUDA cores. Each CUDA core comprises distinct *floating* and *integer arithmetic units*. Additionally, each SM owns 4 *special function units* (SFUs) and 16 load/store units.
The *thread* is the smallest unit the programmer can control. Threads are organized in *blocks* up to 1024 threads. The Fermi architecture supports parallel execution up to 8 blocks per SM, but in practice this number is reduced because of the number of threads used, the register count or, even more importantly, the required shared memory.
Each SM allows the execution of 32 threads, called *warp*, at a single time. In fact, through Fermi's dual warp scheduler, the SM has the ability to run two warps concurrently. The scheduler issues the next instruction of one warp either on the load/store units, the CUDA cores or the SFUs. Since there are less load/store units or SFUs than cores, the cores could be inefficiently waiting for these units. If it is the case, they would start executing instructions from a second warp. Only double precision

instructions do not allow such a dual dispatch, meaning that only one warp is executed at a time.
As each thread possesses an individual instruction address pointer and a register state, they are technically free to branch at every time, therefore every control flow structure (if, else, for, …) or even a memory request represents a possible branch point. However, branches are always to be avoided as they lead to a sequential processing of code paths.

Fast memory access through coalescing

Memory access has a high impact on computation time, as large data structures such as arrays can not be read at once, but rather entry by entry. The memory controller of a GPU can bundle up to 16 read/write operations at once.
The scheduler analyzes the memory requirement of the first active thread in a half-warp, hence for 16 threads. As there are 16 load and store units, it is possible to load 16 int, float or double at a time, resulting in device memory accesses of 64- or 128 byte segments. All threads within a half-warp are executed, if their requested address lies within the loaded segment. After execution, these threads are marked as inactive and the scheduler checks if there are active threads left in the half-warp. If it is the case, the next active thread will be chosen and the corresponding memory transaction issued. In the best case, called *coalesced access*, all requests of a half-warp could be fulfilled at once. In worst case, the requested memory addresses are every time placed outside the loaded segments, leading to sequential processing of threads. In summary, full coalesced access is guaranteed if all 16 threads of a half-warp access a continuous data block of 16 elements.

2.3. OpenCL and CUDA programming model

OpenCL 1.1 and CUDA SDK 4.2 provide two interfaces for GPU computing. CUDA (NVIDIA 2010, NVIDIA 2012a) is only available for NVIDIA graphic boards. In contrast OpenCL, maintained by the Khronos group, aims to provide a hardware-independent framework for parallel computation, as a result it is available to different types of CPUs and GPUs. In both frameworks, the programmer designs different tasks, executed as data-parallel kernel functions on GPU. In order to facilitate programming, they both provide multiple key abstractions: thread model, shared memory and synchronization.

Thread organization

CUDA defines a three level hierarchy for the threads: the top level is the *grid*, which coordinates the kernel functions. A grid comprises multiple *blocks*, each being composed of a number of *threads*. Threads within a block can use shared memory and barriers for synchronization. The number of threads per block and the number of blocks are defined by the programmer, what is also called *kernel configuration*. Configuring an OpenCL kernel requires to define the number of all computable threads (*global work group size*) and the number of threads within a group (*local work group size*), while the number of blocks is implicit.

Shared memory and synchronization

Similarly to CPUs, GPUs also distinguish multiple memory spaces, either by access time, accessibility by threads or size. Each SM has it own shared memory, usable by all its blocks. Note that only threads of the same block will get access to the same shared segment. With 16 kb or 48kb (their exist different software configurations) on the Tesla C2050, this is rather small.

Shared memory provides faster access than global memory and is useful in many cases, first of all for the storage of intermediate results such as the weighted sum, as presented in section 4.2. If the threads of one block need frequent access to small data areas, this data can be preloaded into shared memory before the computation starts, e. g. in case of matrix multiplications. A shared memory segment is only available to threads of the same block. Currently, no functions are provided for synchronization across multiple blocks.

3. Parallel simulation of neural networks

A large variety of approaches for the design of neural networks exists. They range from biologically detailed model of single neurons (compartmental models, conductance-based neurons) to mean-firing rate models (abstracting the dynamics of neural assemblies of several neurons), including the quickly developing field of spiking neurons (Dayan and Abbott 2001). They deal with specific constraints and aim at explaining different biological mechanisms. One common aspect of these networks is that the computation time rises rapidly with the number of neurons and connections between them. This motivates the use of parallel hardware for faster computation.

3.1. Spiking neural networks

Spiking neural networks are an increasingly used tool for studying neural computations in the brain, as they combine a relatively small computational complexity with a fair description of neural dynamics and spiking patterns in populations. There exists many different models of spiking neurons, some being filter-based such as the Spike-Response Model (SRM, Gerstner 1995), others relying only on membrane potential dynamics (often modeled as an ordinary differential equation, ODE), such as the integrate-and-fire (IF) neuron and its variants (see Izhikevitch 2003 for a review). These two types of spiking neurons rely on a relatively simple threshold mechanism to decide when a spike is emitted by a neuron. This spike is then propagated to all neurons connected to it, which in turn modifies their membrane potential. Spiking neurons can therefore be described as an hybrid system (Brette et al. 2007), whose state is described by a number of variables (including at least the membrane potential of the neuron, eventually some internal variables such as rate-adaptation and, depending on the type of synapse considered, some variables describing the postsynaptic potentials - PSPs - induced by incoming spikes) which are governed by specific ODEs and modified by incoming spikes. As most biological neurons typically have low baseline firing rates (between 3 and 10 Hz) and only sporadically fire at higher rates, the propagation of a spike within a spiking neural network can be considered a rare event compared to the dynamics of the neurons (ODEs in such networks are typically numerized with a discretization step of 0.1 or 1 ms).

Modern neural simulators take advantage of this property to limit the communication between the neurons and speed-up the parallel processing. At each simulation step, two phases are performed. In the *neuronal update phase*, the variables of all neurons are updated according to their own dynamics and the arrival of incoming spikes at the synapses. In the subsequent *spike propagation phase*, the membrane potential is compared to a threshold: if a neuron emits a spike, it is propagated to all neurons forming a synapse with it, otherwise nothing happens. For some types of spiking neurons, it is even possible to use event-driven algorithms, where ODEs are not integrated with a fixed clock, but their explicit solution is computed at once after each received spike in order to estimate when the neuron will fire: if no spike is emitted in the network, nothing is computed (Rudolph and Destexhe, 2006). This mechanism, although complex to implement and limited to certain types of networks, achieves significant speed-ups.

SNN (Nageswaran et al. 2009), PCSIM (Pecevski et al. 2009), NEST (Gewaltig and Diesmann 2007), Nemo (Fidjeland et al. 2009, Fidjeland and Shanahan 2010), Brian (Goodman and Brette 2009) or the GeNN simulator (GeNN website) are a few examples of neural simulators which benefit from this spike propagation mechanism in spiking neural networks and exhibit high speed-ups on parallel hardware, focused in most cases on GPUs.

More detailed neural models, such as compartmental or conductance-based models, for which the simulators GENESIS (Bower and Beeman 1998) or NEURON (Carnevale and Hines 2006) are well-suited, can not easily benefit from this research, as they do not have such a threshold mechanism: the whole dynamics of the spike generation is simulated. As an example, Wang et al. (2011) studied more precisely the parallel simulation of Hodgkin-Huxley neurons on GPUs.

3.2. Mean-firing rate neural networks

Another type of neural model widely used in computational neuroscience, especially in systems-level or cognitive approaches, is the mean-firing rate neuron (or rate-coded neuron). This type of neuron is usually described by a single floating variable which represents the instantaneous firing rate of a neuron or group of neurons (such a unit could for example represent the dynamics of a whole cortical microcolumn). This firing rate is governed by an ODE, typically a leaky integrator, and follows a weighted sum of its inputs (which are also firing rates). Contrary to spiking networks, such neurons need to access their inputs at each time step during the numerization of the neurons' ODEs. A spike propagation mechanism is therefore not possible and neurons of this kind of networks require a lot of intercommunication. Fortunately, this is compensated by the fact that much less neurons are required in this framework to perform globally the same computation as a spiking network (Dayan and Abbott 2001). For example, a spiking network with a correct excitation/inhibition balance has to comprise several hundreds of neurons, while the corresponding mean-firing rate networks need only tens of them.

When modeling large-scale neural computations in which synchronization is not thought to be an important process, mean-firing rate networks can therefore be a very useful and fast tool, especially when considering the powerful learning available in this framework, such as covariance learning rules (Dayan and Abbott 2001). Nevertheless, although several neural simulators such as NEST (Gewaltig and Diesmann 2007) or Emergent (Asia et al. 2008) incorporate mean-firing rate neurons, not much effort has been spent recently on their simulation on parallel hardware. Both currently support an implementation on cluster systems via MPI, but there is no support yet of GPUs.

3.3 Neural model

In this article, we aim at investigating the advantages and drawbacks of the parallel computation of mean-firing rate networks on CPUs and GPUs. Typical mean-firing rate neurons have a single output variable (their instantaneous firing rate) which varies as a function of the weighted sum of their inputs through an ODE, usually a leaky integrator. In a population of N neurons, the output of a neuron with index j depends on the weighted sum of its inputs $rate_i$through :

$$\tau \frac{drate_j}{dt} + rate_j = \sum_{i=1}^{N} \quad W_{ij} * rate_i \tag{3.1}$$

where $W_{ij}$is the connection weight between neurons i and j.

For large interconnected networks, the use of an ODE to describe the dynamics of the firing rate does not add significant computational complexity. As an example, we measured computation times in the CUDA framework for a network composed of 25000 neurons receiving 8000 connections each (see section 5 for details on the measurements). The computation of the weighted sum for all neurons took 28.252 ms, while the update of the neural variables took only 12 microseconds for the whole network.

In order to stay as generic as possible, we do not use the ODE in our neural model: the output variable is directly the weighted sum of inputs:

$$rate_j = \sum_{i=1}^{N} W_{ij} * rate_i \qquad (3.2)$$

This simplified model has the advantage of putting the emphasis on the structure of the modeled networks. The results presented in this article will nevertheless represent an upper-limit of the computational costs of such neural networks. More realistic neural models would use more variables or more complex functions, but still the main computational bottleneck would be the integration of inputs. The actual implementation of this network will be described in section 5.

3.4. Parallel evaluation of neurons

Three types of parallelism have been identified for neural networks (Jahnke et al, 1997; Nageswaran et al, 2009): neuron-, synapse- and pattern parallel. In neuron-parallel evaluation, each thread takes care of a single neuron. On the opposite, synapse-parallel evaluation distributes the calculation of the weighted sum over all available threads. Pattern-parallel evaluation can be considered as a combination of neuron- and synapse-parallel evaluations.

Our implementation of the weighted sum (section 5.2) is based on a pattern parallel evaluation: a variable group of threads computes the weighted sum for a single neuron. The other necessary local operations should be computed in a neuron-parallel manner. The only exception would be global operations such as the maximal or the mean firing rate of a population, where a pattern parallel evaluation with the reduction algorithm (Harris, 2008) would be more effective.

Because of the parallel evaluation of neurons, synchronization has to be ensured after each numerical step in order to ensure mathematical exactness. This is especially a problem if the computations differ strongly between neurons, either because of different number of connections (heterogeneous network structure) or of varying effort of local computations in the case of few connections per neuron. The calculation of the weighted sums requires the use of two different arrays to store the firing rate variables: one for the firing rate computed at the last step and one for the storage of the result. After all neurons are updated, the two arrays are switched via pointer swap. Note that an asynchronous computation (where only one array is used and results are immediately written) is faster and even has functional advantages (Rougier and Vitay 2006), but does not ensure mathematical exactness.

4. Data model

One important question is how to represent connection and neural data, more precisely which precision should be used and how should the different elements be stored. The use of single floating point precision is common in the field of GPU computing, but most spiking neural simulators use double precision. Other approaches, such as the NeMo simulator (Fidjeland et al. 2009, Fidjeland and Shanahan 2010), use a fixed-point format, which allows the use of atomic operations on GPUs. Secondly, the storage of the different pieces of information in distinct arrays (called Structure of Arrays, *SoA*, Siegel et al. 2011) is common. But since CUDA SDK 2.0, several optimizations were introduced for object-oriented programming (OOP), hence also an object-oriented approach (Array of Structures, *AoS*) appears to be a valid base for the design of general-purpose neural networks. It has the advantage of using OOP techniques in the neural simulator, for example the framework could provide several basic neuron types, connectivity patterns or learning rules which describe a general functionality. Then these objects could be adapted through derivation by the modeler for his special issues. In addition to the work of Siegel et al. (2011), we will take a closer look at memory alignment and removal of padding space.

4.1. Representation of neurons

In mean-firing rate networks, neural dynamics only depend on locally stored data, minimally the firing rates at the current and previous time steps, but other variables may also be used. It is therefore possible to store the different neural variables either in arrays or in objects regarding the previous considerations. For the studies in this paper, we chose to store all neural data, a value for the firing rate of the last and current time step in two one-dimensional arrays, as neural representations are much smaller than connection data and hence could be neglected for the performance measurements.

4.2. Representation of connections

The data organization of the connections is one of the major points to improve performance. As realistic neural populations are typically sparsely connected, Brette and Goodman (2011) presented different ways to represent these as sparse connection matrices. They used a calculation approach based on arrays, called vectorized operation, to achieved great speedups on networks of single-compartment neuron models. Such vectorized operations are very useful on GPUs and also reasonable for mean-firing rate neuron models. Table 1 shows the comparison between the SoA and AoS representations to the dense, list of a list (LIL) and compressed sparse row (CSR) representations introduced in that article.

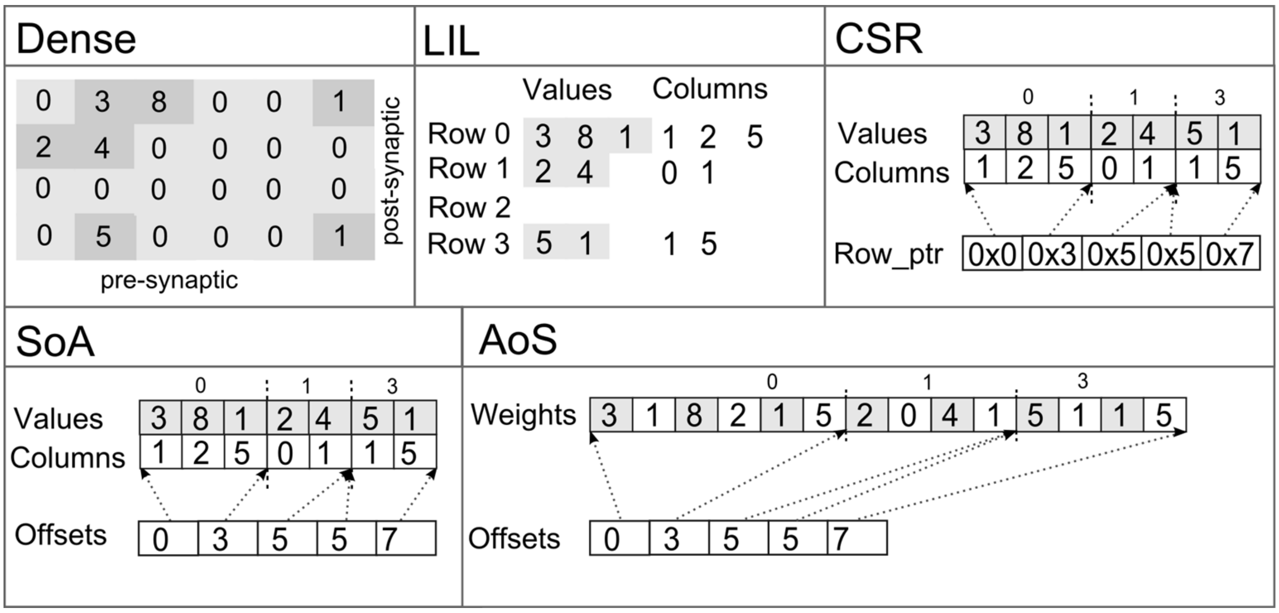

Table 1: Overview of the dense, LIL and CSR connectivity structures presented by Brette and Goodman (2011) and the structures used in this article, SoA and AoS. Dense representations store every possible connection between two populations, even if they do not exist. List of a list (LIL) stores for each postsynaptic neuron the weight value and index of presynaptic neuron for each existing connection. In compressed sparse row (CSR) representation, all values and indices are separately stored in arrays, with an additional pointer array keeping track of the connections belonging to each neuron in the postsynaptic population. Structure of arrays (SoA) simply replaces this pointer array with indices. Array of Structures (AoS) gathers weight values and indices into small structures. In all approaches, the gray items denotes the weight values and the white the indices or pointers.

Dense matrices store all weight values explicitly, even if no connection exists. Obviously more efficient representations are needed in the case of sparse connections. *List of a list (LIL)* stores line by line the column index and weight values of existing connections. Each row represents one neuron in the population. In the case of *compressed sparse row* representation (CSR), all column/value data are stored in one array. For every neuron the pointer to the corresponding data is stored in *row_ptr*. This is very similar to the used SoA representation, except that we replaced the row pointer by an integer offset. This is due to the fact that the data is created at the host side and then transferred to the device: a pointer would become invalid after this transfer. In case of AoS, the connection information (weight and presynaptic index) are stored together in a structure. Altogether, the memory requirements in the case of LIL, SoA and AoS are the same, if we consider the optimized variants of AoS presented in the next section.

4.3. AoS and alignment

When talking about structures, a second point need to be discussed: memory alignment. Structures composed of members with different alignment requirements (through varying data types sizes) possibly lead to higher memory requirements and lower performance. This data structure alignment is normally handled by the language implementation. Nevertheless some language compilers, as for C++, CUDA and OpenCL compiler, provide the programmer with at least a partial control.
In our example, a connection object consists of a *double* value for the weight and an *integer* for the index, theoretically needing 12 bytes. The aligned boundary, multiple of 8 bytes on 64 bits architectures, causes the compiler to allocate 16 bytes, implying that 4 memory padding bytes are allocated for each structure. Because of the large number of synapses per neuron, these 4 padding bytes might be a crucial problem for large models.

struct Weight {
    double weight;
    int index;
};
 	
AoS(1)
	struct Weight {
	    double weight;
	    int index;
	};
	__attribute__((__packed__))
	
	AoS(2)
		struct __align__(4) Weight {
		    double weight;
		    int index;
		};
		__attribute__((__packed__))
		
		AoS(3)

Table 2: Different AoS implementations, where each connection object consists of a weight and an index value. ***AoS(1)*** *The standard implementation which uses 16 bytes on 64bit architectures.* ***AoS(2)*** *With the packed attribute, an object is represented with a minimal space requirement, here 12 bytes.* ***AoS(3)*** *The minimum alignment is set to 4 bytes so that the memory misses are reduced. The size is similar to AoS(2). The influence of these implementations on computation times is presented in section 6.3.*

Table 2 shows the different connection data representations studied in this paper. AoS(1) is the standard definition: allocation of this object takes 16 bytes per object. Usage of the *packed* attribute (AoS(2)) reduces the memory requirement to its minimum: 12 bytes. While increasing performance on CPU, this representation reduces it on GPUs because of misaligned accesses. With the *align* attribute (AoS(3)) the alignment could be changed towards the lowest member of the structure and the performance increases on GPUs (more details on this in section 6.3).

4.4. Memory requirement as a limiting factor

In large-scale models, one limiting factor is the memory required to represent the network: each neuron possesses thousands of connections, represented by 12 bytes at least in our design. In general, CPU-based systems have more memory available than graphic cards. A single graphic card currently stores up to 3 (e.g. Tesla C2050) or 6 GB (e.g. Tesla C2070), to compare with the 48 GB available on our CPU-based system or with the several terabytes available on high-end servers. If ECC (error-correcting code, see section 5.3) is enabled on Tesla cards, the available memory is reduced by 12.5%. To estimate the memory required by a network, we can look at the populations, assuming that all neurons within a population receive the same number of connections. The space requirement *m* in bytes of a population is calculable as shown in equation 4.1.

$$m = 2 \cdot n \cdot d + n \cdot c \cdot (d + i) \qquad (4.1)$$

where n denotes the number of neurons and c the number of connections. The value d represents the data size of all floating point values (in our model the weights and firing rates of the last and current time step), which is either 4 (float) or 8 bytes (double). The *i* represents the size of the integer index value which is constant 4 bytes. The first part of the sum represents the local data (here the 2 firing rate values per neuron) and the second part is the memory requirement of the connection data for all neurons.

5. Methods

In this section the implementations of the different frameworks are presented. In the code snippets, the term *pr* denotes the firing rate of the previous time step. The updated firing rates are stored in *r*. Connection information are represented through an index (*idx*) and a weight value (*w*). The number of neurons in the network is denoted by N and the number of connections for each neuron by C.

5.1. CPU implementation

For OpenMP, a neuron-parallel implementation was realized as shown in Algorithm 1.

```
#pragma omp parallel for schedule(guided, 10)
for(int i=0; i<N; i++) {
```

```
4           for(int j=0;j<C;j++) {
5               r[i] += pr[idx[j]] * w[j];
6           }


7       }
```

Algorithm 1: The code of the multi-threaded CPU implementation with OpenMP. The first for-loop iterates over each neuron and is split over several threads. The assignment of neurons to a certain thread is automatically done by OpenMP (line 1). The second for loop calculates the weighted sum (eq. 3.1) over all presynaptic neurons. The term pr denotes the firing rate at the previous time step, r at the current time step, w the weight value and idx the index of the presynaptic neuron.

5.2. GPU Implementation

Instead of a neuron-parallel evaluation, as in the OpenMP implementation, the GPU implementation uses a pattern-parallel evaluation. The neuron-parallel approach would not be suitable, because each thread within a warp requires different connection data, scattered with a high possibility over different data segments, leading to additional memory requests. Nevertheless this statement only stands for the computation of the weighted sum. The neuron update itself, based on local processing, could be computed in a neuron-parallel manner. A synapse-parallel evaluation of the weighted sum would not be usable for smaller number of connections, as the workload on threads will be to low. Therefore we use a pattern-parallel evaluation, which is a mix of both.

The evaluation (Algorithm 2) is based on the well-known reduction algorithm (Harris, 2008). Reduction is used to compute parallel sums in a hierarchical way without using for-loops. Usage of this algorithm reduces the complexity for the sum from O(T) to O(log(T)), where T is the number of threads per block.

```
extern PREC __shared__ sdata[];
PREC mySum = 0.0;
```

```
 while(i < C) {
         mySum += pr[idx[nIdx*C+i]] * w[nIdx*C+i];
         i+= blockSize;
 }
sdata[tid] = mySum;
__syncthreads();

if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum + sdata[tid + 256]; } __syncthreads(); }
if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum + sdata[tid + 128]; } __syncthreads(); }
if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = mySum + sdata[tid +  64]; } __syncthreads(); }

if (tid < 32)
{
    volatile PREC* smem = sdata;

    if (blockSize >=  64) { smem[tid] = mySum = mySum + smem[tid + 32]; }
    if (blockSize >=  32) { smem[tid] = mySum = mySum + smem[tid + 16]; }
    if (blockSize >=  16) { smem[tid] = mySum = mySum + smem[tid +  8]; }
    if (blockSize >=   8) { smem[tid] = mySum = mySum + smem[tid +  4]; }
    if (blockSize >=   4) { smem[tid] = mySum = mySum + smem[tid +  2]; }
    if (blockSize >=   2) { smem[tid] = mySum = mySum + smem[tid +  1]; }
}

if (tid == 0)
    r[blockIdx.x] = sdata[0];
```

Algorithm 2: As implementations for CUDA and OpenCL work in a similar way, only the CUDA implementation is shown. A neuron with index nIdx is calculated by several threads which each computes one part of the weighted sum stored in mySum. PREC is a template parameter given to the kernel, either double or float. The intermediate results are stored in shared memory, allowing a faster read/write access than global memory for the following reduction step.

5.3. Sample networks and measurement

The article focuses on the computation time for one step of a population, while varying the number of neurons and connections for the three implementations. If not otherwise said, all floating point values use single precision and stored as SoA.

Hardware of the test system

The OpenMP test was run on a dual Intel X5660 hexa-core system, with minimum operating system load and 12 threads. All tests were made on the operating system Windows 7, 64-bit with current drivers and SDKs. The CUDA and OpenCL tests were run on a single Tesla C2050 (CUDA SDK 4.2, Driver version 301.32). Current graphic boards of the NVIDIA Tesla series offer two new features for scientific computation: improved double precision performance (NVIDIA 2010) and ECC (error-correcting code) support. ECC avoids memory errors but reduces the speed of computation. Tests on the Tesla C2050 reports performance impairment with ECC ranging from 0% up to 20% (NVIDIA 2012d). With our CUDA implementation, we observed a speed difference of 19%, with a network consisting of 40000 neurons with 8192 connections each and using single precision. Furthermore the available memory space is reduced (by 12,5% on Tesla C2050). Nevertheless, all tests in section 6 were done with ECC enabled, keeping in mind that it will lower the performance but ensure mathematical correctness.

Time measurement

All mentioned times are in milliseconds and are calculated by a mean over at least 20 runs. The time of single runs are determined by the corresponding framework functions. In CPU/GPU benchmarks, the standard deviation of these measurements is very small, thus we will report only the mean values in the following section. In the example of section 6.2, for SoA we determined 75.87 ms for CPU and 22.27 ms for GPU. The standard deviations were 0.241 ms for CPU and 0.0038 ms for GPU, hence both deviations are negligible.

Performance comparison

Direct comparisons of CPU/GPU implementations via the absolute computation times are strongly dependent on the used hardware and optimization level of the code, and can be seen as highly subjective. The used compilers have different ways to generate and optimize the given codes and it is impossible to say that they are equal to each other. For C++ compilation, we used the compiler option -O3, assuming that the CUDA/OpenCL frameworks also use the highest optimization level.
All implementations are compared to a serial implementation (one thread on the CPU system). In section 6.6, we consider scalability as an alternative metric to evaluate CPU and GPU implementations. We investigate two main cases: neuron and connection scaling. In the first case, the network is composed of 25000 (double precision) or 40000 (float precision) neurons, each receiving a number of connections ranging from 16 to 8192 items, each test doubling the previous one. In the second case, the network had a ranging number of neurons (800 to 409600, each test doubling the previous one), receiving 100 connections each.

In section 6.2, we investigate the influence of the number of threads per block on the performance of GPU implementations. In all other experiments, we simulate each network with a variable number of threads per block (from 4 to 512 in multiples of 2) and retain the configuration with the minimal computation time.

6. Results

The performance of OpenMP, CUDA and OpenCL implementations depends of course on the network size, but several other factors play a significant role. We will first examine these issues: we discuss in section 6.1 the impact of single vs. double precision on the network performance. In section 6.2, we investigate how to choose the optimal number of threads per block in GPU implementations depending on the size of the network. In section 6.3, we compare the different connection representations as presented in section 4.3. After evaluating these influences, we examine the performance depending on the network size for a variable number of neurons (section 6.4) and connections (section 6.5). Section 6.6 investigates the scalability of our implementations. Finally, section 6.7 addresses the impact of the connectivity structure on the performance.

6.1. Floating precision

In this section, we investigate the effect of floating precision on performance. As a test setting, we used a network consisting of 25000 neurons with different number of connections (16 up to 8192) per neuron. Figure 1A shows the computation time as a function of the number of connections per neuron for single and double precision on CUDA and OpenCL. For small networks with less than 256 connections per neuron, the OpenCL implementation is dramatically worse than the CUDA one: between 23 and 47% slower with single precision and between 15 and 62% slower with double precision. For denser networks, the comparison between the CUDA and OpenCL implementation is better: with double precision, the performance decrease goes from 11% with 512 connections per neuron to less than 0.1% with 8192 connections. With single precision, there is a similar decay, but less pronounced: from 22% with 512 connections to 6% with 8192 connections.
Figure 1B shows the ratio of performance of the network when using single instead of double precision for both CUDA and OpenCL implementations. The performance improvement induced by single precision instead of double is rather stable with increasing connectivity for OpenCL, but for small networks single precision does not improve significantly the performance with CUDA. This result could be explained by the fact that on Fermi cards, double precision operations do not allow for dual-warp scheduling. As it is unclear how the dual warp scheduler deals with warps

containing a high number of inactive threads, small networks having underloaded warps may be simulated as fast using double and single precision. A more intensive investigation of this effect is out of the scope of the present article and will be the target of future work.

The relative increase in performance brought by the use of single precision instead of double has an upper limit of 1.5. As we described in section 4, the connection data in the case of single precision requires 8 bytes per connection, while the double precision needs 12 bytes. The use of single precision therefore reduces memory consumption by a factor of 1.5, which is consistent with the observed increase in performance with CUDA. This could suggest that the principle limitation in GPU processing is linked to memory bandwidth. The relatively worse behavior of the OpenCL implementation could therefore be due to problems regarding memory operations on the NVIDIA card used here. This is suggested by the higher percentage of global memory replay overhead from OpenCL in comparison to CUDA in our tests, as we determined using the NVIDIA profiler. Nevertheless this behavior should be part of a further investigation.

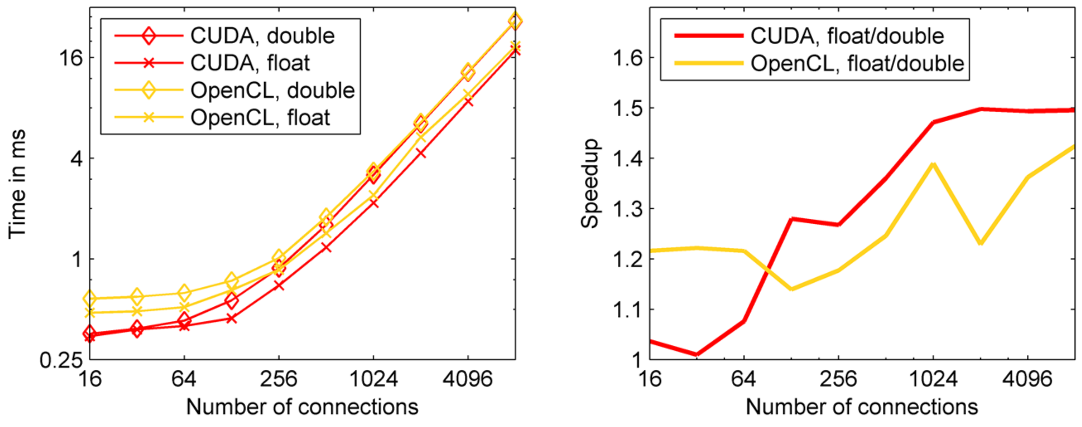

*Figure 1: **A)** Computation times depending on the floating point precision. We tested the CUDA and OpenCL implementation using either single or double precision. The number of connections per neuron were variable in the range 16 to 8192 items, while the number of neurons was set at 25000. The CUDA and OpenCL implementations have nearly the same computation time for large number of connections when using double precision. **B)** The speedup in performance observed when using single instead of double precision.*

6.2 Optimal number of threads per block

On a GPU, the number of the threads and blocks influence the scheduling behavior and therefore the performance. In the CUDA and OpenCL implementations, we used one block for each neuron in the network. The remaining question is how many threads are optimal depending on the number of connections. As test setting, we used a network composed of 40000 neurons with a variable number of connections. We tested thread configurations ranging from 4 threads up to 512 threads per block for both implementations. For the sake of clarity, we show here only the results for CUDA, as the results for the OpenCL implementation were almost identical.

Figure 2 shows the computation times depending on the number of connection per neuron and the number of threads per block. The dotted line describes the optimal number of threads per block as a function of the number of connections, i.e. the configuration which leads to minimal computational times. For weakly connected networks, the optimal number of threads per block is rather low (for 32 connections per neuron, the computation times ranged from 0.34 ms with 8 threads to 2 ms with 512 threads), as too many threads would be idle if the number of threads per block were set high. For higher numbers of connections, 256 or 512 threads per block are the optimal configurations, suggesting that only strongly interconnected networks can fully exploit the intrinsic parallelism of the hardware in our implementation. It is therefore important to analyze the structure of a network before its simulation in order to find the optimal number of threads per block. For the next results, we use the optimal thread configurations, as described in section 5.3.

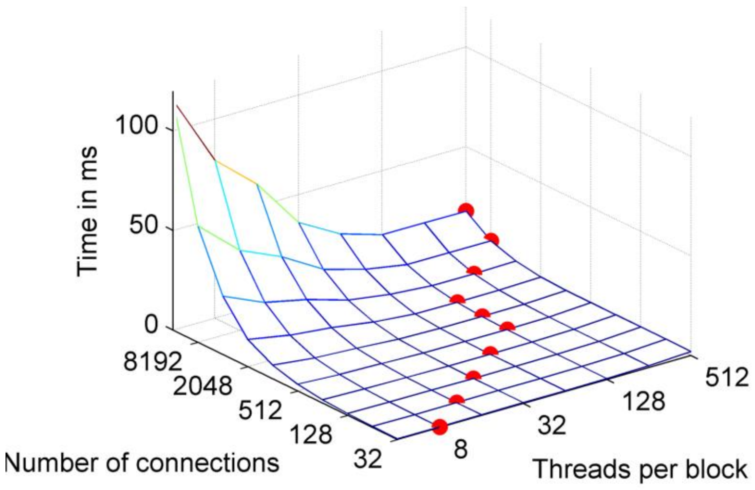

Figure 2: Computation time depending of the number of threads per block and the number of connections per neuron. The network was composed of 40000 neurons. The dotted line depicts the optimal number of threads per block as a function of the network's structure.

6.3. Data organization: SoA versus AoS

We now investigate the influence of connection data representation on performance. Figure 3 shows the comparison between the SoA and the three different AoS representations which were presented in section 4.3. The tests were run for a network consisting of 25000 neurons, each receiving 6000 connections and using double precision. As will be shown later, this huge number of neurons and connections allows us to rule out any cache effect.

First of all, the SoA implementation is faster than the basic AoS implementation (AoS(1)), by a factor of 1.32 on CPU and 1.38 on GPUs. The use of *packed* attribute (AoS(2)) leads on CPUs to nearly the same computation time as SoA. On GPUs, AoS(2) slows down by factor 1.63 compared to SoA which results from the impaired alignment property in this version. Restoring the alignment property in AoS(3) reduces the performance difference between AoS(3) and SoA to a factor of 1.13. The presented results are only valid in case of structures consisting of elements with different memory requirements. This is the case with double precision, where the structure consists of a double value (weight) and an integer value (index). In the case of homogeneous memory requirements (e.g. only integers and single precision floating point values), there will be an effect of the *packed* and *aligned* attributes only if the memory requirement is not a multiple of 8 bytes.

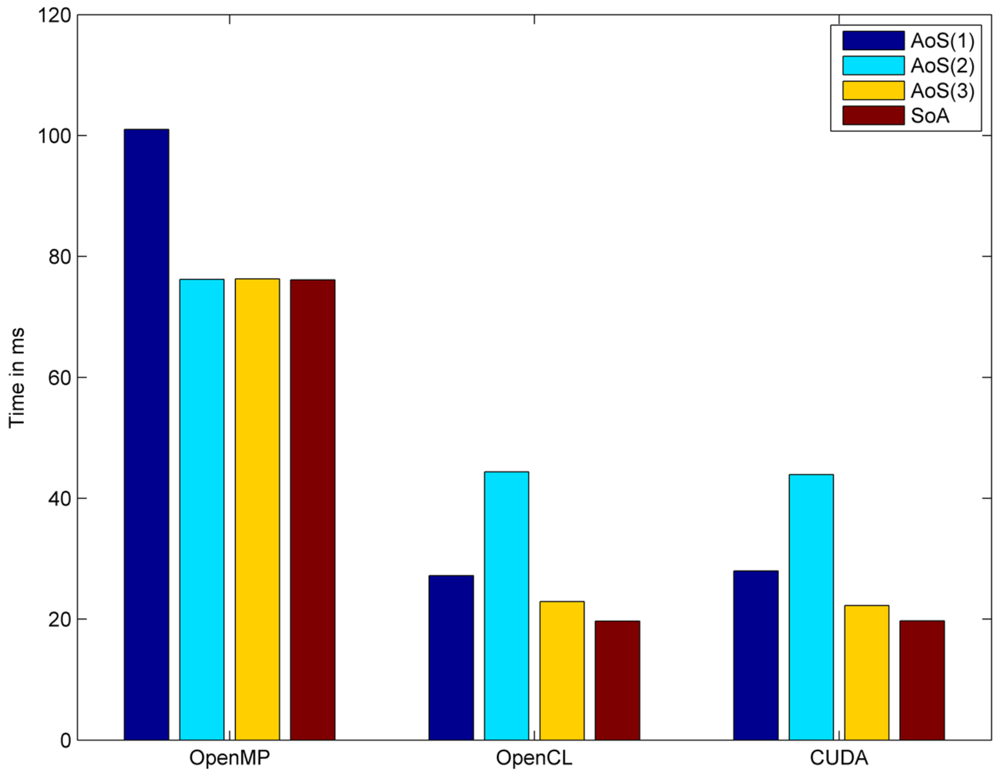

Figure 3: Comparison of the different data representations discussed in section 4.3. AoS(1-3) denote three different structure-based implementation whereby SoA denotes the array-based version. As test setting we used 25000 neurons with each 6000 connections. On CPU the 2nd and 3rd AoS implementation reach the performance of the array-based representation. On the GPU, there still exist a performance difference of 13 percent between structure- and array-based representation. Cache effects on CPUs can be ruled out, because the used memory is larger than the existing caches.

Altogether, the use of arrays instead of structures for the connection data is more efficient. However, with the right optimizations on the connection data, object-oriented representations have similar performances on CPUs and only impairs GPU performance by 13%. If the size of the connection data further increases (e.g. a synapse-specific delay is added to the structure), one would expect anyway a much worse degradation of performance on GPU for structure-based compared to array-based representations.

6.4. Computational time and number of neurons

In this section we investigate the computation time of a network depending on the number of neurons in a homogeneous network (Figure 4). The networks have different numbers of neurons, ranging from 800 to 409600 neurons, receiving 100 connections each.

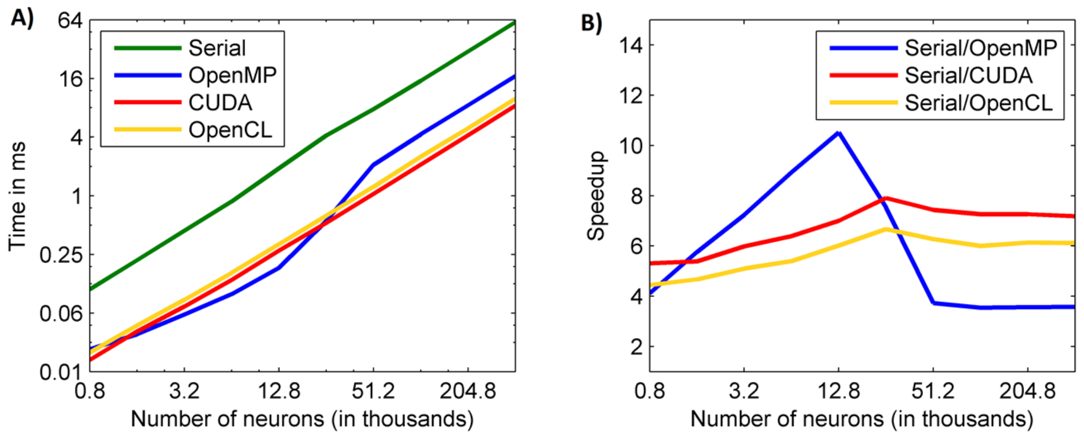

*Figure 4: **A)** The computational time depending on the number of neurons. **B)** The speedup between the GPU and CPU implementations compared to the serial one is shown. In the networks, each neuron has 100 connections while the number of neurons is variable. The OpenMP implementation is faster than OpenCL or CUDA for mid-sized networks thanks to cache effects. For larger network sizes, GPU implementations are faster than OpenMP, with a clear advantage for CUDA over OpenCL.*

Figure 4A shows the obvious fact that the computational time increases linearly with the number of neurons for serial, OpenMP, OpenCL and CUDA implementations. Figure 4B shows the speedups, i.e the performance improvement of the different parallel implementations with respect to the serial one. Very small networks (less than 800 neurons) are slightly faster simulated on GPUs than on CPUs (factor 1.25). A possible explanation would be an inefficient repartition of memory on the caches of the 12 cores used in the OpenMP implementation. As the focus of this article is on large-scale networks, we have not investigated this effect in greater depth.

For large networks (more than 25600 neurons), GPU implementations are faster than the CPU implementation, with speedups of 6 and 7 for OpenCL respectively CUDA and 4 for OpenMP. In the middle-range of network sizes, the CPU caches, which are larger than on a GPU, allow for a much faster memory access and therefore faster computation on the CPU. For these mid-size networks, the OpenMP implementation is faster than the GPU ones (maximal speedup of 11 for 12800 neurons). The exact limit size where networks are better simulated by GPUs instead of CPUs is strongly hardware-dependent, as it depends on the cache sizes on the CPU-based platform. The CPU caches on the test platform have the size of 12x 32kb(L1), 12x 256kb(L2) and 2x 12MB(L3), what should be compared to the GPU caches of 14x 48kb(L1) and 1x 256kb(L2). Theoretically, only networks larger than the size of the L2 cache (3MB or 3900 neurons) or the size of L3 cache (24MB or 31400 neurons) could benefit from GPU acceleration, as shown by our results.

6.5. Computational time and number of connections

A second scalable parameter of a homogeneous network is the number of connections per neuron. In Figure 5, networks of 40000 neurons receiving each different number of connections (from 16 up to 8192 connections) are compared.

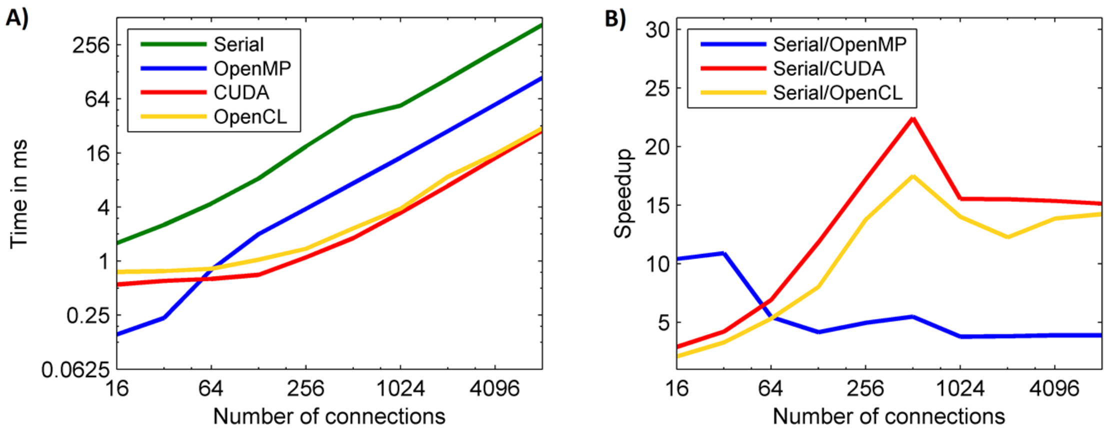

*Figure 5: **A)** The computational time depending on the number of connections in each neuron. **B)** The speed up between the GPU / CPU implementation compared to the serial implementation is shown. The network consists in each test of 40000 neurons. For weakly interconnected networks, the OpenMP implementation obtains higher speedups than OpenCL/CUDA ones because of cache effects. More strongly interconnected networks benefit from a GPU implementation, CUDA being again more efficient than OpenCL.*

When there are more than 128 connections per neuron, the GPU implementations are clearly faster than the CPU implementation in this setup. As with the dependency of computational time on the number of neurons, cache effects are responsible for the fact that only highly interconnected networks would benefit from a GPU implementation. The size of the L2 caches (3MB) could store a network of 9.8 connections in average for each neuron while the L3 caches (24MB) could store 78.6 connections. This is reflected in Figure 5 by the fact that OpenMP is faster than CUDA or OpenCL for less than 64 connections per neuron. As in section 6.4, networks with a lot of interconnections will benefit more than smaller networks from a GPU implementation.

6.6. Scalability

Instead of absolute time measurements, we now focus on the scaling factor of the computational time if we enlarge the network. Computation time analysis as in section 6.5 and 6.6 allows for a comparison of CPU and GPU implementations which is strongly dependent on the used hardware and optimization level of the code. A scalability plot addresses the issue to know if the speedup goes further as it seems or if it comes to a breakdown. The normalized computation time is the ratio between the computational times for the considered network and a reference network.

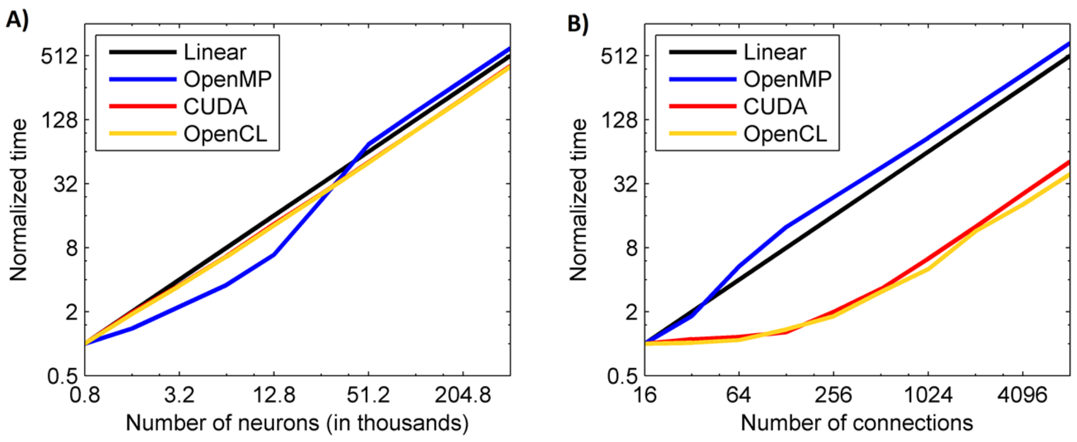

*Figure 6: Scaling of computational time for the implementations towards **A)** neuron respectively **B)** connection numbers. Normalized computation time refers to the ratio between the considered network and a reference network having 800 neurons (A) or 16 connections (B). The networks studied have the constant parameter of 100 connections (A) and 40000 neurons (B).*

On Figure 6A, the reference network has 800 neurons, while on Figure 6B it has 16 connections. All networks in Figure 6A receive 100 connections, while the networks of Figure 6B comprise 40000 neurons. The ideal case, in which the scaling factor is linear, would reflect the fact that if one doubles the number of connections in comparison to the reference network (scaling factor of 2), the computational time will also double. If the scaling factor deviates from this expected linear behavior, some additional mechanisms such as cache effects interfere with the parallelization process.

Figure 6 shows that the CPU and GPU implementations exhibit asymptotically a linear behavior More precisely, the OpenMP implementation performs better for small networks, but is rapidly degraded when cache effects disappear. The GPU implementations tend to have a better scaling behavior than expected. A possible reason is that the GPU is fully used for large networks, but not for small ones such as the reference network, meaning that the reference performance is underestimated.

6.7. Connection structure, sparse connectivity and coalescence

In biological networks, connectivity patterns are usually very sparse (cortical pyramidal neurons have around 6000 synapses, what should be compared to the millions of neurons in a single cortical area) and based on probabilistic distributions (although there often exists a topology in connection patterns, synapses may not always exist). When building realistic models, this means that a neuron will receive connections from neurons that are very probably not in adjacent memory spaces. In the case of GPUs, this will result in a random index pattern which will increase the calculation time.
The indices of the presynaptic neurons determine if the firing rates can be accessed in a coalesced manner or not, what could devastatingly impair the performance. Two extreme cases have to be distinguished, an evaluation with an ascending index pattern and a random one. We assume here the standard condition where the firing rates are stored in continuous arrays. This distinction is not common to spiking neural simulators which store incoming spikes in a local array. In firing rate networks, this reordering of active inputs is not possible.

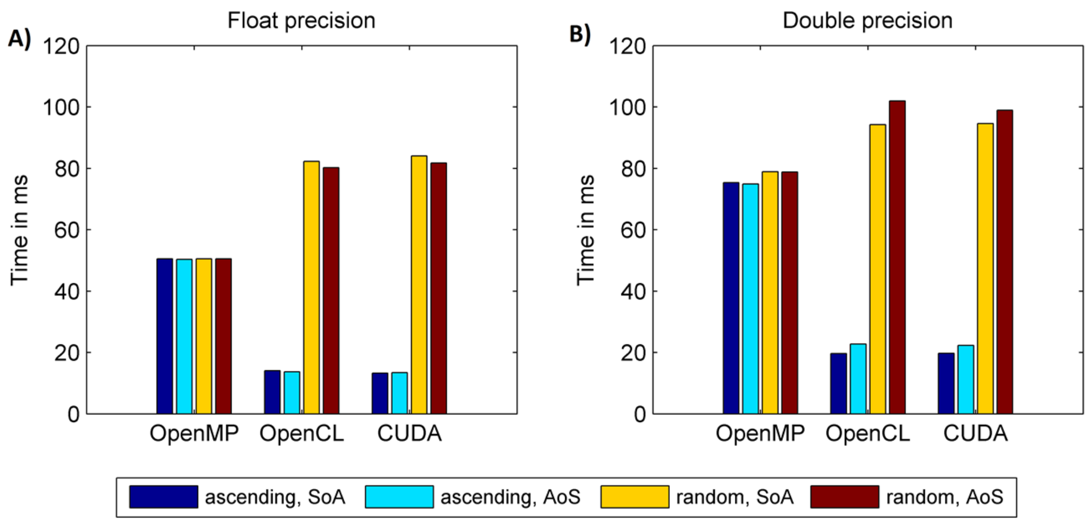

Figure 7: Comparison of random and ascending index of presynaptic neurons on CPU and GPU implementations considering ***A)*** *single precision and* ***B)*** *double precision. The network consists of 25000 neurons with each 6000 connections, what results in a full load for the GPU and CPU systems and rules out any cache effects. Obviously there is a high downfall through random indexing on GPUs by a factor of 5.8 for single and 4.76 for double precision, independently from the AoS or SoA data model. On CPUs instead, the effect is negligible (factor 1.02 respectively 1.05).*

Ascending index occurs if the indices of the neurons are sequentially increasing and hence the weighted sum is calculated over neurons with sequentially increasing number. This in turn implies memory access to firing rates which are stored next to each other in memory and enables coalesced memory access to the presynaptic firing rates, speeding up read access. More precisely, it is only necessary that a group of 16 neurons have an ascending index, because the coalesced memory

access could read at maximum 16 firing rates at once. Considering the example in Table 1, its obvious that this condition is hard to fulfill for sparse connectivity patterns.
The second case is a completely random index pattern, where the weighted sum is calculated over firing rates which are stored separately in memory. Coalesced access to firing rates is therefore not possible for all threads within the group which results in multiple memory loads. In both cases, the other variables (weights, index-array) being already coalesced through SoA or nearly coalesced through AoS(3), as explained in section 5.2.
Sparse connectivity patterns can seriously impair performance on GPUs while having only minor effects on OpenMP implementations, as Figure 7 clearly shows. The OpenMP implementation has nearly identical performance with random and ascending index when using single precision, while performing slightly worse with random indices and double precision. On the contrary, the GPU performance is reduced by a factor of 5.8 resp. 4.76 when using single resp. double precision. The use of SoA instead of AoS representations for the connection data does not ensure full coalescence in the random index case, as firing rates are nevertheless accessed in different memory blocks. As a conclusion, the use of GPU implementations only helps when it is ensured that firing rates of the presynaptic neurons are ordered in an ascending manner, at least in blocks of 16.
In all previous tests, we had used an ascending index, because the random index method would have penalized performance too much and would overflow any further effects, especially for the GPU.

7. Discussion

We have investigated the implementation of mean-firing rate neural networks in different parallel paradigms, such as OpenMP, CUDA and OpenCL. We used a simplified model of mean-firing rate neurons which nevertheless allows to study the constraints imposed by the parallel hardware, as the integration of synaptic inputs is the principal computational bottleneck. Our results show a better suitability of GPUs for large-scale neural networks under certain constraints: large networks will benefit from the GPU if they require more memory than the available CPU caches, otherwise an OpenMP implementation should be preferred.
Data representation is also a critical factor for performance. Since CUDA SDK 2.0, object-oriented programming on GPUs is supported and the performance of heterogeneous structures was massively improved by this release, as shown by Siegel et al. (2011). In addition to this work, we compared various structure representations on our parallel frameworks. We investigated the performance difference between a structure, defined with *packed* and *aligned* attribute (AoS(3)), and a common array-based representation (SoA). We have shown that on CPUs, these two

representations achieve nearly the same computation time, but that on GPUs the performance decreases by a factor of 1.13 for AoS(3). This decrease is solely due to the violations of coalescence induced by heterogeneous structures.

A typical question in the field of scientific computing is the choice of the right precision. Double precision ensures a higher correctness of simulation results but requires more memory, which is a critical issue especially on GPUs. In section 6.1, we showed than the use of single precision is 50% faster on CUDA and 30% on OpenCL when compared to double precision. The data structure we used for the connections requires 50% more memory with double precision as with single precision, what could principally explain the observed speedup with CUDA. If a neural network model could be made tolerant to single floating-point calculations, it would significantly accelerate its simulation on GPU systems.

We also observed that the OpenCL implementation is systematically worse than CUDA, especially for sparsely connected networks and/or in the case of single precision. This may be due to specific problems of OpenCL regarding memory operations on NVIDIA graphics cards. It is also important to mention, that all results are influenced by enabled ECC, whose effect is not further investigated in this article. Until these effects have been further studied, a CUDA implementation should be preferred for mean-firing neural networks.

Direct comparisons between the CPU and GPU implementations should be interpreted carefully. First the hardware is heterogeneous: we compared 448 cores in the GPUs against only 12 in the dual-CPU. The devices strongly differ in terms of computational speed, memory bandwidth, cache organization. Second, the implementations must be equally well optimized at the CPU and the GPU, which is complicated to ensure because of the unknown optimization heuristics of the CUDA and OpenCL compilers. Nevertheless it can be seen as a soft factor for the decision which parallel hardware to use. Like Nageswaran et al. (2009) or Nowotny et al. (2011), we compare the different parallel implementations against a serial one. This allows to investigate how the different implementations cope with increasing complexity in the network size and structure.

The biggest problem arises when the network has a sparse or random connectivity structure: neurons receive connections randomly from other neurons, and not in an ascending order. As we showed, this totally breaks down the performance of GPU implementations, while CPUs are only marginally affected. This is maybe the strongest argument to raise against GPU implementations of mean-firing rate neural networks, as this sparse connectivity pattern is present in biological networks. Mean-firing rate neural simulators which aim at functioning on GPUs definitely have to address this issue. Minimally, they should find a way to group neuronal variables in groups of 16 or multiples of 16 items in order to make use of the coalesced memory access patterns of GPUs. This could be achieved by a structural analysis of the network's connectivity before simulation, in order to attribute indices in the most efficient manner.

The problems investigated in this article are comparable with other works, such as Nowotny et al. (2011). They investigated the parallelization of a similar neural model by focusing on a classifier consisting of three layers (784 input neurons; 50000 hidden neurons; 10 output neurons), with random connections between the input and hidden layers and all-to-all connections between the hidden and output layers. The authors used different kernels for each connection pattern while keeping all computations local. In comparison, we focused on more generic network architectures by varying the networks' size and connectivity. They reported speedups between 2 and 4 for the OpenMP implementation compared to the serial one, and between 7 and 11 for the CUDA version. These rather low speedups are comparable with our results (between 4 and 11 for OpenMP, 5 and 23 for CUDA). The difference may be due to the different hardware, as well as their choice of a specific network for their test.

In summary, both parallel hardware architectures, CPU and GPU are both well suited for parallelization of a neuronal network based on mean-firing rate neurons. A GPU implementation could be preferred for homogeneous large neuronal networks. In this case, the network should use a greater amount of memory than what is available in the CPU caches and the connectivity must ensure a coalesced memory access patterns. Otherwise a multi-threaded CPU should be preferred.

8. Declaration of interest

The authors report no conflict of interest.

9. References

Asia, B., Mingus, B., O'Reilly, R. 2008. The emergent modeling system. *Neural Networks*. 21:1146-52

Bower, J. M., Beeman, D. 1998. The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System (2nd Ed.). New York: Springer-Verlag.

Brette, R. and Goodman, D. F. M. 2011. Vectorised algorithms for spiking neural network simulation. Neural Computation. *Neural Computation.* 23(6):1503-1535

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris F. C. and others. 2007. Simulation of networks of spiking neurons: A review of tools and strategy. *Journal of Computational Neuroscience*, 23:349-398

Brandstetter, A. and Artusi, A. 2008. Radial Basis Function Networks GPU-Based Implementation. *IEEE Transactions on Neural Networks.* 19(12):2150-2154

Carnevale, N. T., Hines, M. L. 2006. The NEURON book. Cambridge University Press.

Chapman BM, Gropp WD, Kumaran K, Müller, MS. 2011. OpenMP in the Petascale Era - 7th International Workshop on OpenMP. *IWOMP2011*.

Chelapilla K., Puri, S. and Simard, P. 2006. High performance convolutional neural networks for document processing. *Proceedings 10th International Workshop Frontiers Handwriting Recognition*.

Dayan, P. and Abbott, L. F. 2001. Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems. MIT Press.

Dranias, M., Grossberg, S. and Bullock D. 2008. Dopaminergic and non-dopaminergic value systems in conditioning and outcome-specific revaluation. Brain Research, 1238, 239-287.

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

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

Frank, M.J. and O'Reilly R. C. 2006. A mechanistic account of striatal dopamine function in human cognition: Psychopharmacological studies with cabergoline and haloperidol. *Behavioral Neuroscience*, 120: 497-517.

GeNN website [Internet]. GeNN simulator. [cited 2012 Oct 09];
Available: http://genn.sourceforge.net/

Gerstner W. 1995. Time structure of the activity in neural network models. *Phys. Rev. E*, 51:738-758

Gewaltig, M.O. and Diesmann, M. 2007. NEST. Scholarpedia, 2(4):1430.

Goodman, D. F. M. and Brette, R. 2009. The Brian simulator. *Frontier of Neuroscience* 2009, 3(2): 192-197

Harris, M. 2008 [Internet]. Optimizing parallel reduction in CUDA. [cited 2012 Oct 9]. Available: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Hong, S. and Kim, H. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. ISCA '09 Proceedings of the 36th annual international symposium on Computer architecture.

Intel 2012 [Internet] Intel Xeon X5660 data sheet. [cited 2012 Oct 9]. Available:
http://ark.intel.com/products/47921/Intel-Xeon-Processor-X5660-%2812M-Cache-2_80-GHz-6_40-GTs-Intel-QPI%29

Jahnke, A., Schönauer, T., Roth, U., Mohraz, K. and Klar, H. 1997. Simulation of Spiking Neural Networks on Different Hardware Platforms. International conference on artificial neural networks. pp. 1187-92

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

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

Nere, A. and Lipasti, M. 2010. Cortical architectures on a GPGPU. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. New York 2010: 12-18

NVIDIA Corporation. 2012a. NVIDIA CUDA C Programming Guide. Version 4.2.

NVIDIA Corporation. 2012b. NVIDIA CUDA C Best Practices Guide. Version 4.2.

NVIDIA Corporation. 2012D [Internet]. Tesla C2050 performance benchmarks. [cited 2012 Oct 09]. Available: http://www.siliconmechanics.com/files/C2050Benchmarks.pdf.

NVIDIA Corporation 2010 [Internet]. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper. [cited 2012 Oct 09]. Available: http://www.NVIDIA.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

McCulloch, WS. and Pitts, WH. 1943. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics,* 5:115-133.

Rougier, N. and Vitay, J. Emergence of Attention within a Neural Population. *Neural Networks,* 19 (5):573-581

Nowotny, T., Muezzinoglu. M. K. and Huerta, R. 2011. Bio-mimetic classification on modern parallel hardware: Realizations in NVidia CUDA and OpenMP. *International Journal of Innovative Computing, Information and Control*, 7:3825-3838

Oh, K. S. and Jung, K. 2004. Gpu implementation of neural networks. *Pattern Recognition*, 37(6): 1311–1314

Pecevski, D., Natschläger, T., Schuch, K. 2009. PCSIM: a parallel simulation environment for neural circuits fully integrated with Python. *Frontiers in Neuroinformatics*. 3:11. doi:10.3389/neuro.11.011.2009

Rudolph, M., Destexhe, A. 2006. Analytical integrate-and-fire neuron models with conductance-based dynamics for event-driven simulation strategies. *Neural Computation.* 18(9):2146-2210.

Seifert U. 2002. Artificial neural networks on massively parallel computer hardware. European Symposium on Artificial Neural Networks; ESANN'2002 proceedings.

Schroll, H., Vitay, J. and Hamker, F. H. 2012. Working memory and response selection: A computational account of interactions among cortico-basalganglio-thalamic loops. *Neural Networks,* 26:59-74.

Siegel, J., Ributzka, J. and Li, X. 2011. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator. *Journal of Algorithms & Computational Technology*. 5(2):432-462.

Sim, J., Dasgupta, A., Kim, H. and Vuduc, R. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pp 11-22.

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

Zhongwen, L., Hongzhi, L., Zhengping, Y. and Xincai, W. 2005. Self organizing maps computing on graphic process unit. *Proceedings European Symposium Artificial Neural Networks*, pp. 557–562.

Ziesche, A., Hamker, F. H. 2011. A Computational Model for the Influence of Corollary Discharge and Proprioception on the Perisaccadic Mislocalization of Briefly Presented Stimuli in Complete Darkness. *Journal of Neuroscience*, 31(48): 17392-17405.