

CARMEN Meeting, Newcastle, 24-05-2011

# Flexible neuronal network simulation framework using code generation for NVIDIA<sup>®</sup> CUDA<sup>™</sup>

T. Nowotny

Informatics, University of Sussex

# Introduction

- Parallel computers go as far back as the 50s and early 60s.
- The desktop revolution however was with serial CPUs (and remains largely such to date)
- Fuelled by the games industry, graphics adaptors have become very powerful and are using increasingly parallel graphical processing units (GPUs)
- This has inspired performance seekers to use GPUs for general purpose computation

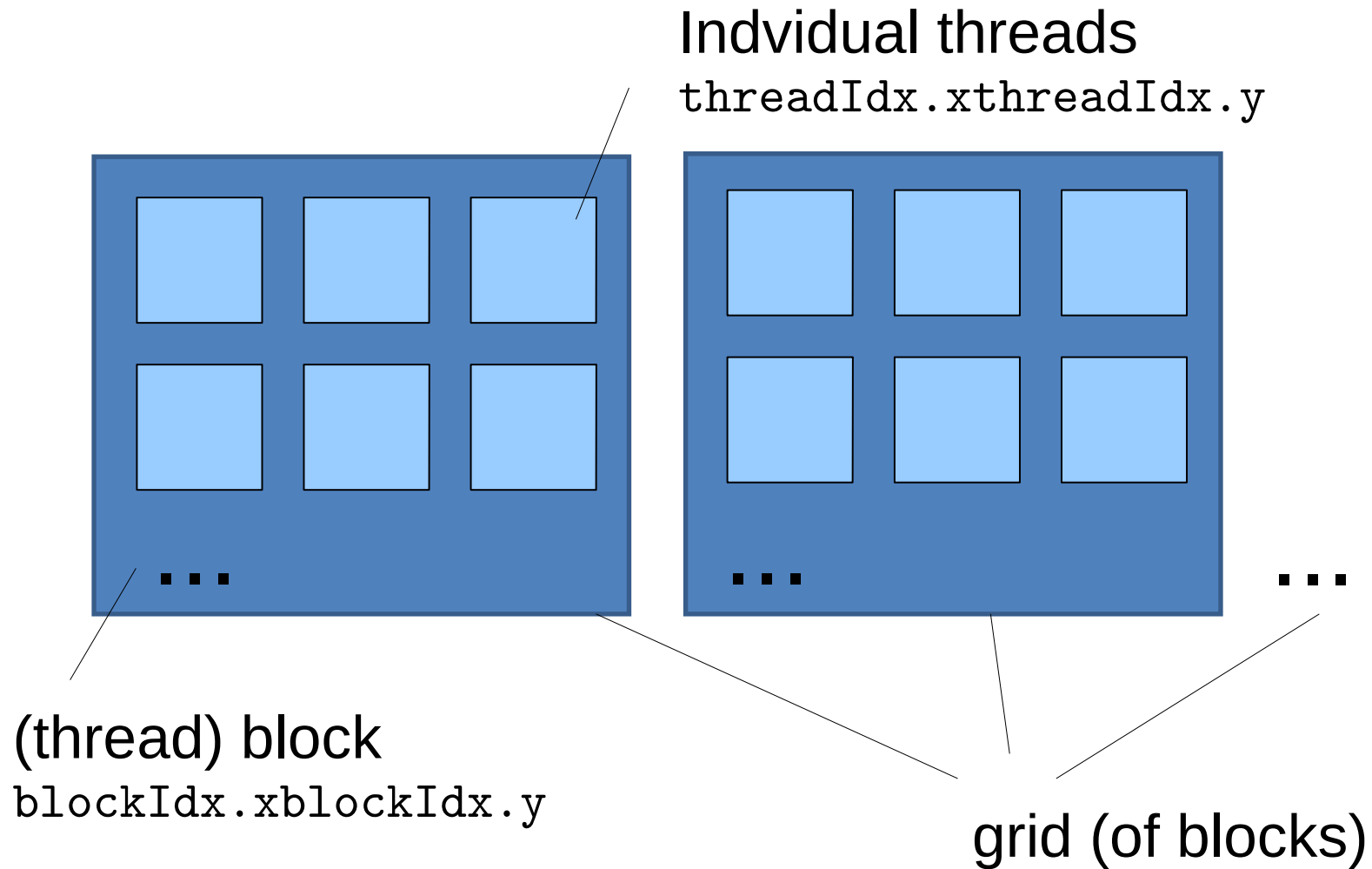
# Neural networks on graphical processing units (GPUs)

- Implementations of neural networks on GPUs appeared around 2004
- They used Cg shader language (OpenGL) or HLSL (Direct X)
- They reported impressive speedups of 20x and more over CPU based simulations
- However, the technology remained a niche because general purpose computing on GPU (GPGPU) using shaders is **very involved**

# Nvidia<sup>®</sup> CUDA<sup>™</sup>

- CUDA<sup>™</sup> = “Common Unified Device Architecture”
- It was introduced by NVidia<sup>®</sup> to allow main stream developers to use massively parallel graphics chips for GPGPU **without the restrictions of shaders**
- CUDA<sup>™</sup> is supported on all newer NVidia cards

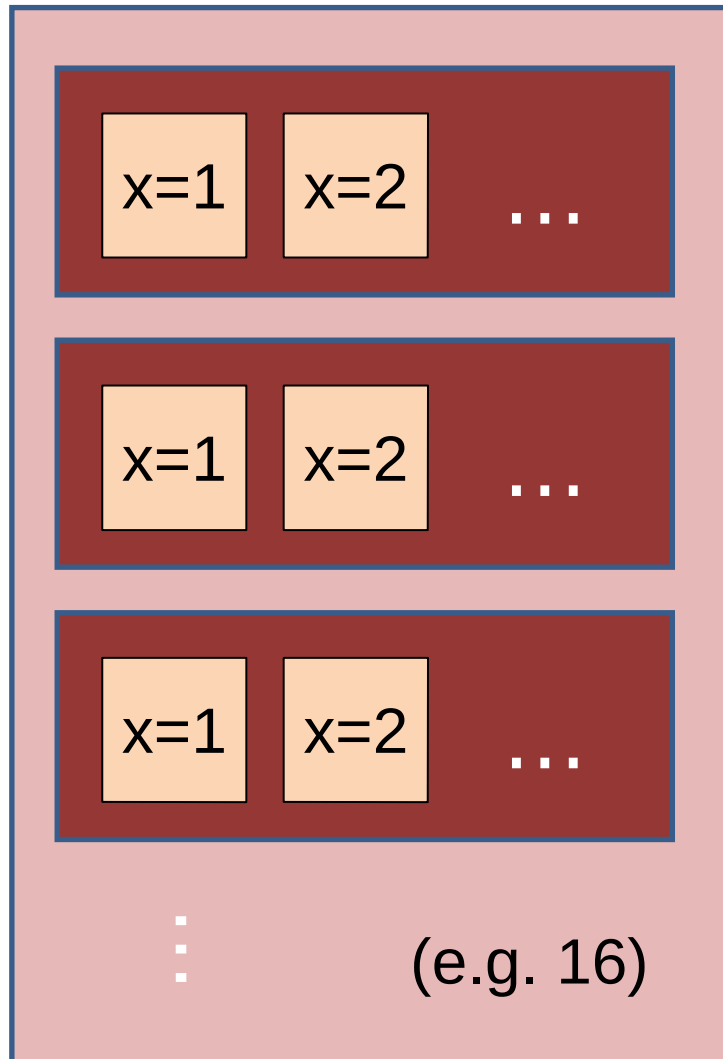
# The CUDA™ API



# The CUDA™ API

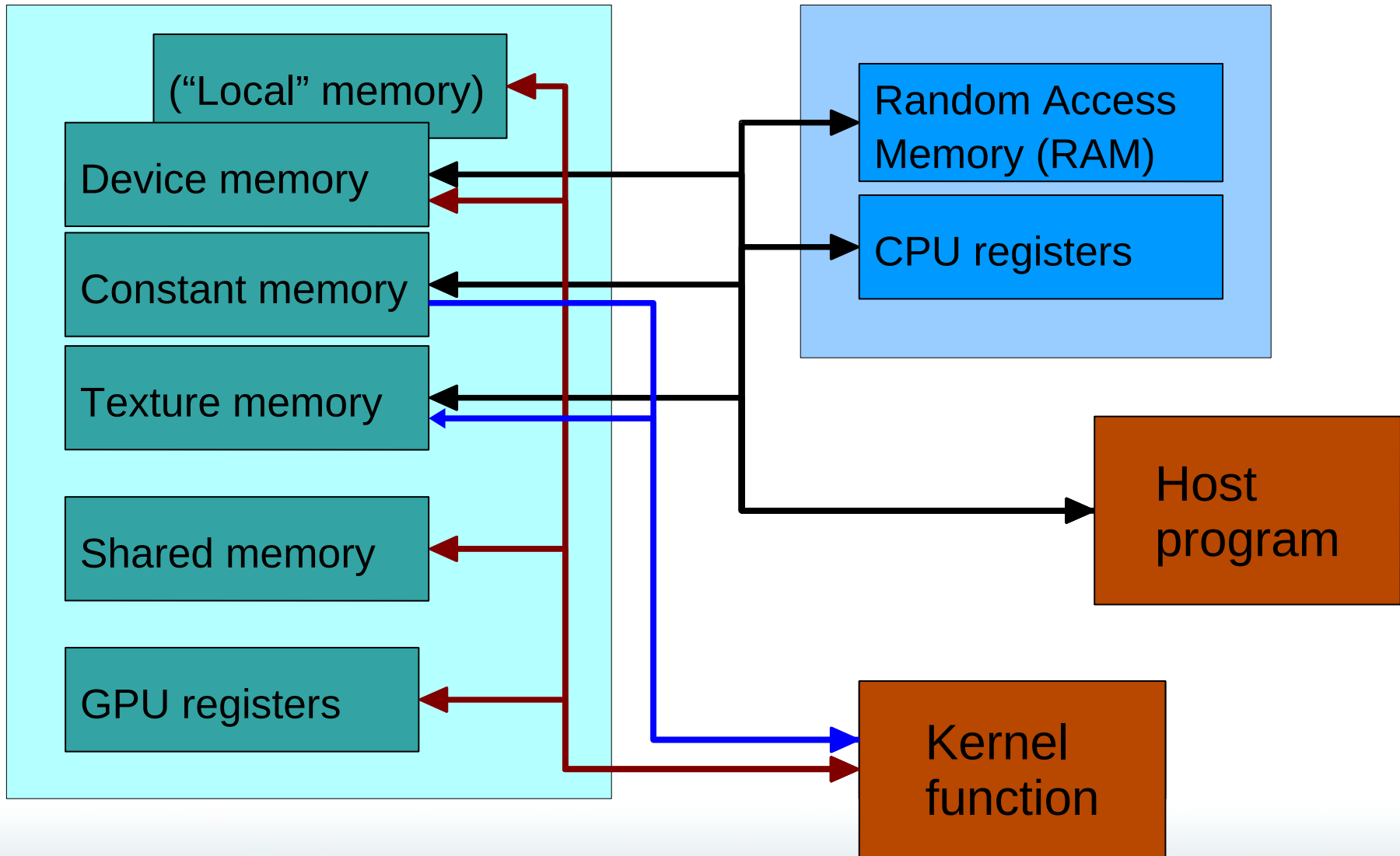
- Each thread executes what is called a “kernel”
- This is a Single-Instruction-Multiple-Data (SIMD) environment
- Blocks of threads can share memory and can be synchronised
- Different blocks may execute in parallel or consecutively
- Maximal block sizes typically 512 (1024) threads
- Grid sizes up to 65535 x 65535 blocks

# What happens on the GPU



- GPUs have several streaming multi-processors (typ. 2-16)
- Each block occupies one multi-processor
- Within the block, threads are executed in (half) warp sizes of (16) 32
- Other thread warps are swapped in and out

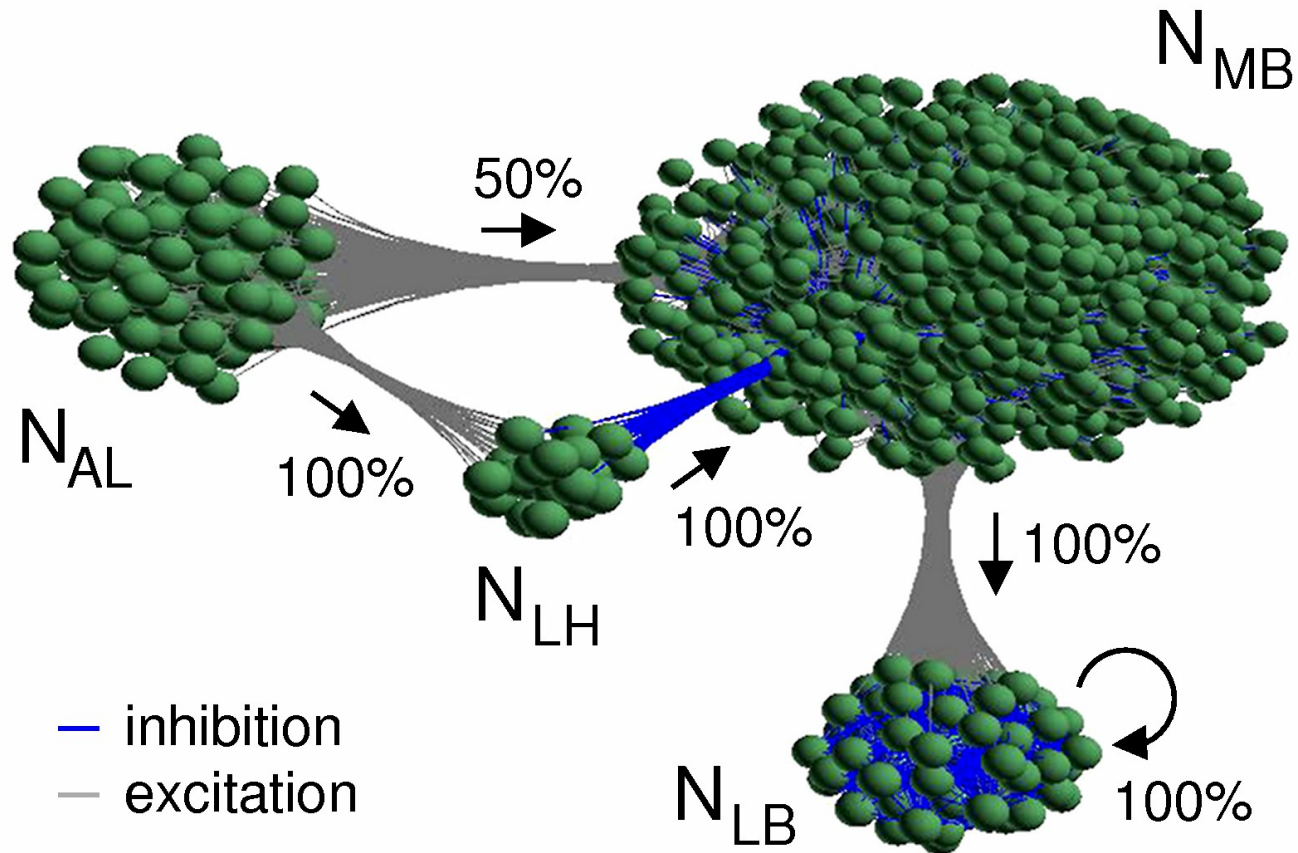
# CUDA memory architecture



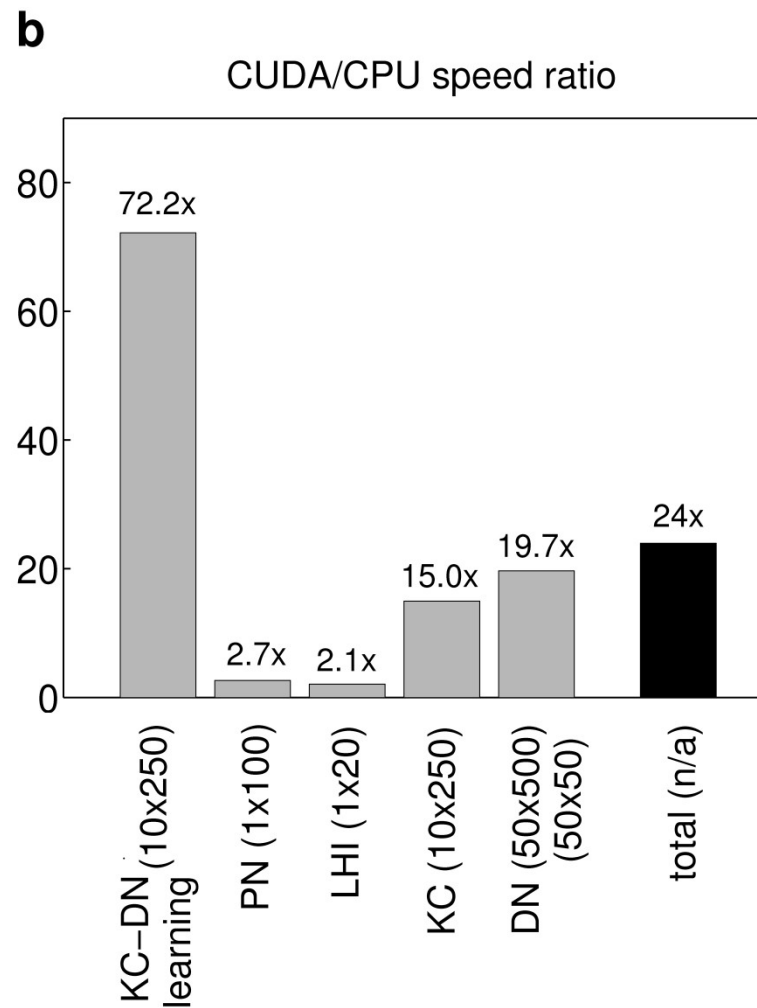
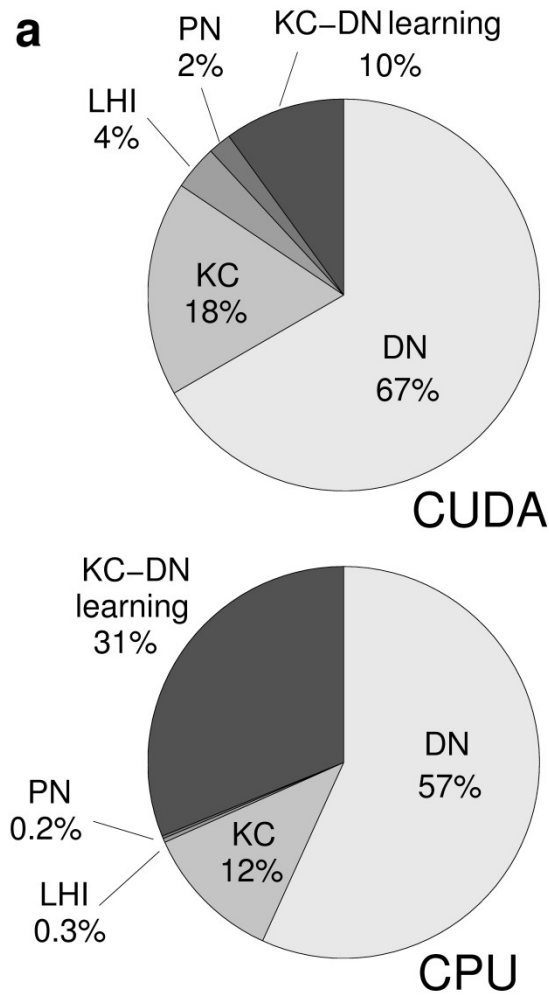
# Why the complex memory hierarchy?

- Texture memory fast for localised lookup
- Constant memory very fast, but no write from kernels
- Shared memory quite fast & shared between kernels, but small and consistency only after explicit synchronisation (bottleneck for performance)
- Registers extremely fast and local but, obviously, not for communication and very few ...
- “local” memory is convenient but as part of device memory quite slow
- Device memory is large and accessible from everywhere but quite slow

# Example: Insect olfaction model



# Hand-tuned neuronal network simulation 2009



T. Nowotny, WCCI 2010 Barcelona

## However ...

- Took me a month to program a previously developed model (this is after learning how to do CUDA)
- The program was optimised for “my” GPU (a Tesla C870)
- It was optimised for one size of the simulation

**Basically this code is useless for any other purpose!**

## Other solutions

- Nageswaran et al. (UC Irvine, 2009): General simulator for Izhikevich neurons with delay, optimized for Izhikevich's thalamo-cortical model (C++ library)
- Fidjeland et al. (Imperial, 2010) - Nemo: General simulator for Izhikevich neurons with delay, optimized for Izhikevich's thalamo-cortical model (C++ library)
- Goodman & Brette (Ecole Normale Sup., 2009): GPU extensions to the Brian simulator (one-timestep grids with partial CPU involvement (massive data transfer bottleneck))
- Mutch et al. (MIT, 2010) CNS simulator: Simulator for layered “cortical networks”, models can be defined by the user (used exclusively through a MatLab interface)

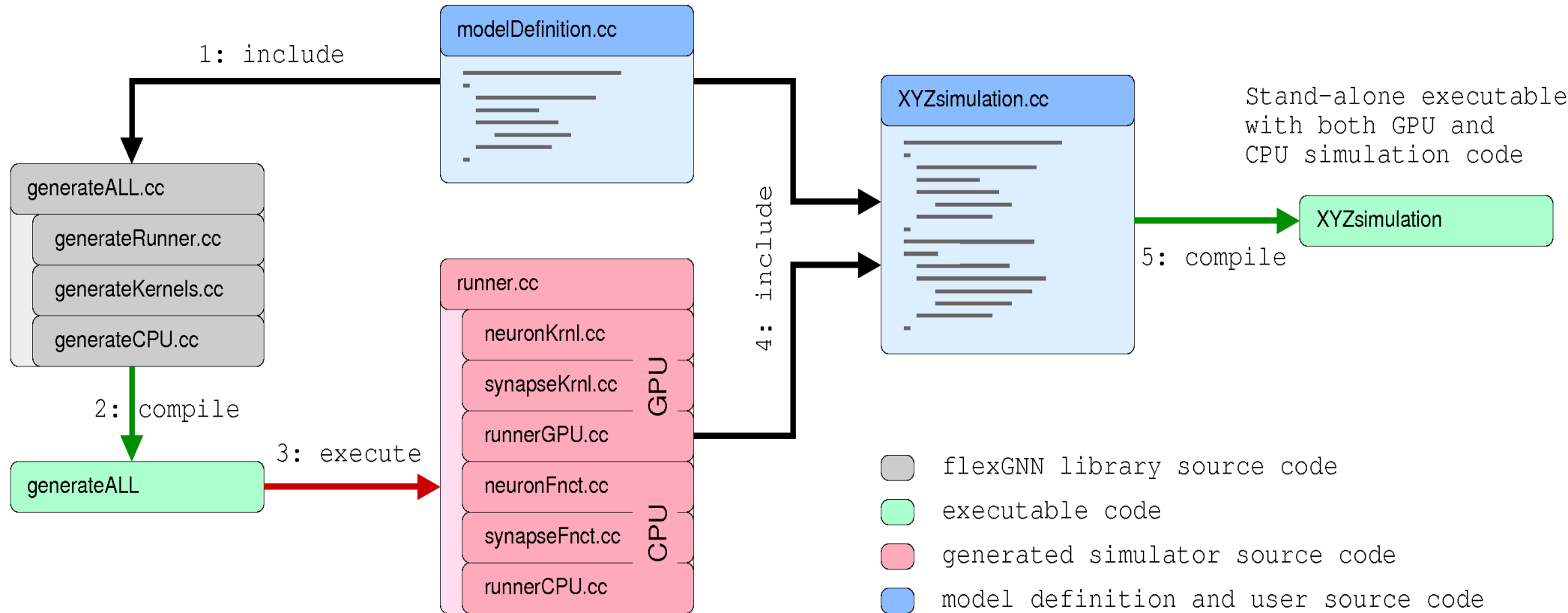
The generality of simulators is limited  
because everything has to be  
optimised to the specific model and  
to the hardware capability

# GeNN: Taking code generation seriously

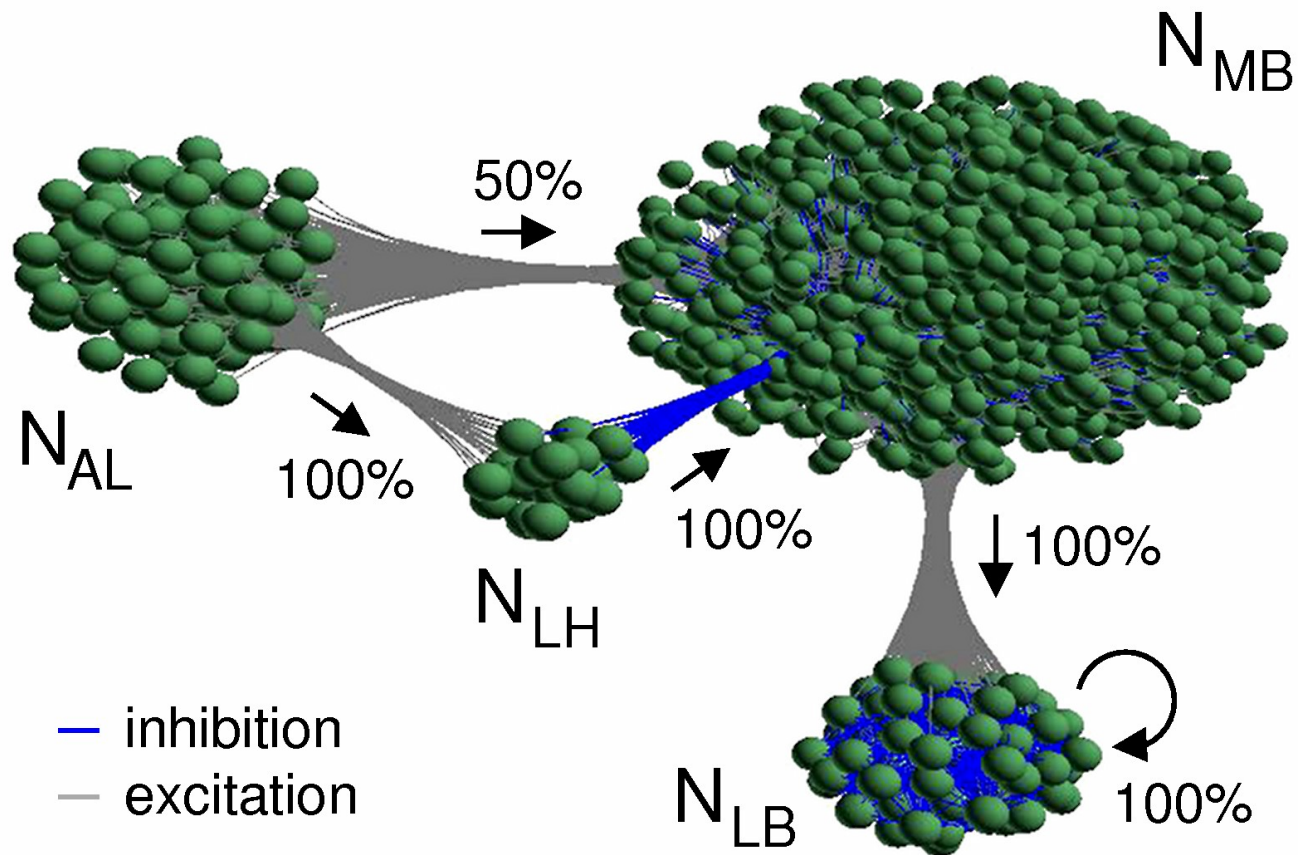
**G**PU **e**nhanced **N**euronal **N**etwork simulator contributes by taking code generation serious:

- Provides a simple C++ API for specifying a neuronal network of interest
- Generates optimised C++ and CUDA code for the model **and for the detected hardware at compile time** (e.g. grid/block organisation, HW capability, model parameters)
- GeNN can offer a large variety of different models – only the used ones actually enter the generated code
- The generated code is compiled with the native NVidia compiler (and all its optimisations).

# GeNN flowchart



# Example: Insect olfaction model



# The API calls for defining a model

```
model.setName("MBody1");
model.addNeuronPopulation("PN", NAL, POISSONNEURON, myPOI_p, myPOI_ini);
model.addNeuronPopulation("KC", NMB, TRAUBMILES, stdTM_p, stdTM_ini);
model.addNeuronPopulation("LHI", NLHI, TRAUBMILES, stdTM_p, stdTM_ini);
model.addNeuronPopulation("DN", NLB, TRAUBMILES, stdTM_p, stdTM_ini);

model.addSynapsePopulation("PNKC", NSYNAPSE, DENSE, INDIVIDUALG, "PN",
"KC", myPNKC_p);
model.addSynapsePopulation("PNLHI", NSYNAPSE, ALLTOALL, INDIVIDUALG, "PN",
"LHI", myPNLHI_p);
model.addSynapsePopulation("LHIKC", NGRADSYNAPSE, ALLTOALL, GLOBALG,
"LHI", "KC", myLHIKC_p);
model.setSynapseG("LHIKC", gLHIKC);
model.addSynapsePopulation("KCDN", LEARN1SYNAPSE, ALLTOALL, INDIVIDUALG,
"KC", "DN", myKCDN_p);
model.addSynapsePopulation("DNDN", NGRADSYNAPSE, ALLTOALL, GLOBALG,
"DN", "DN", myDNDN_p);
model.setSynapseG("DNDN", gDNDN);
```

# API for using the GPU parts

```
// define the connectivity
gpKCDN= ...
gpALKC= ...
...
// copy connectivity and initial values to device
copyGToDevice()
copyStateToDevice();
...
// Run the model on the GPU
For (i=1; i < N; i++) {
    stepTimeGPU(..., t);
}
...
// get results back
copyStateFromDevice();
...
```

# Building it

>> buildmodel Mbody1

...

>> make clean && make

```
# Add source files here
EXECUTABLE := classol_sim
# Cuda source files (compiled with cudacc)
CUFILES_sm_13 := classol_sim.cu
# C/C++ source files (compiled with gcc / c++)
CCFILES :=
CUDACCFLAGS := --ptxas-options=-v
#####
# Rules and targets
include $(NVIDIASDKPATH)/C/common/common.mk
```

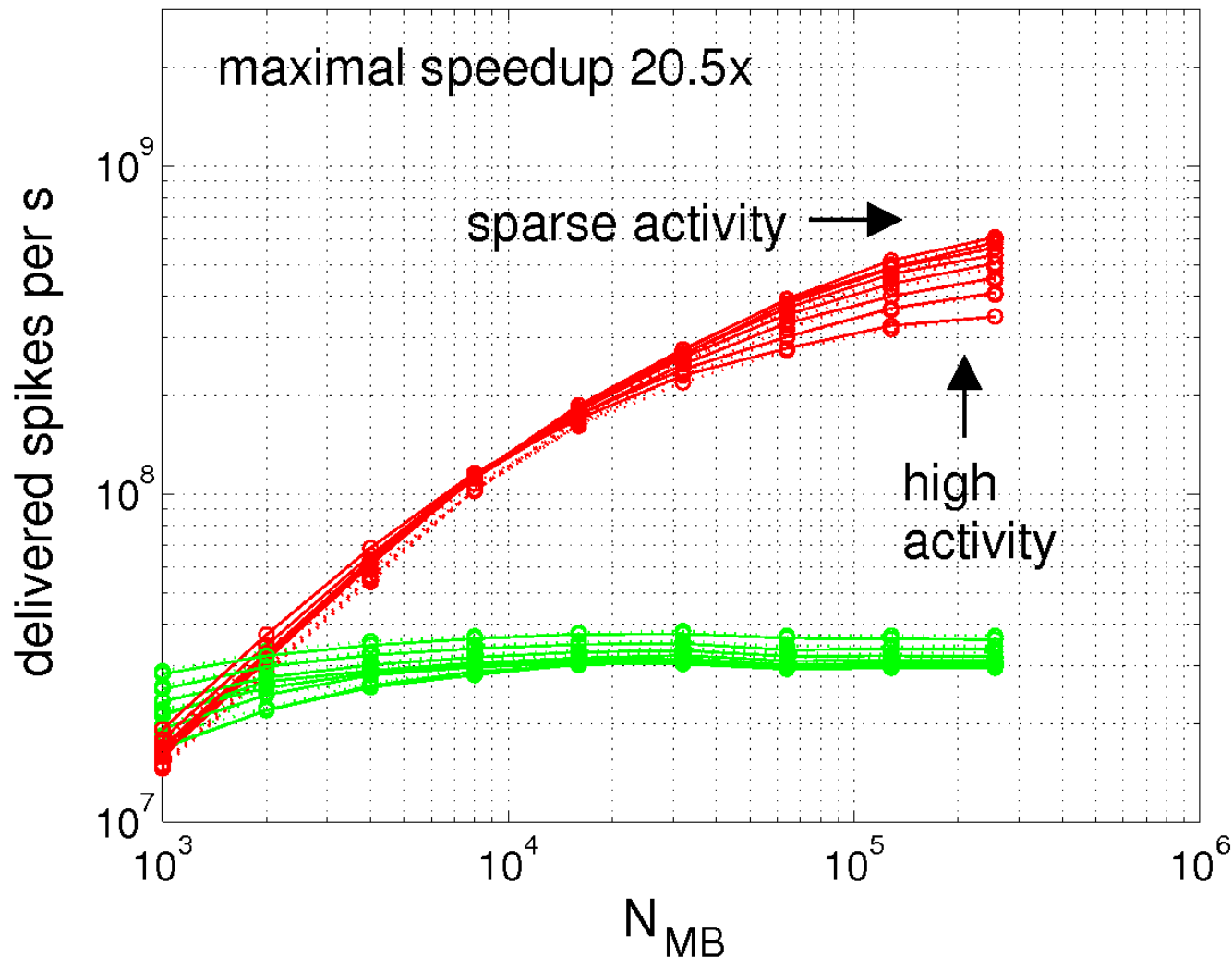
Makefile

# Performance

$$N_{AL} = 1000$$

$$N_{LH} = 20$$

$$N_{LB} = 100$$



AL-MB:  
50 % all-to-all

Individual con-  
ductances

spikes commu-  
nicated to host  
(dotted)

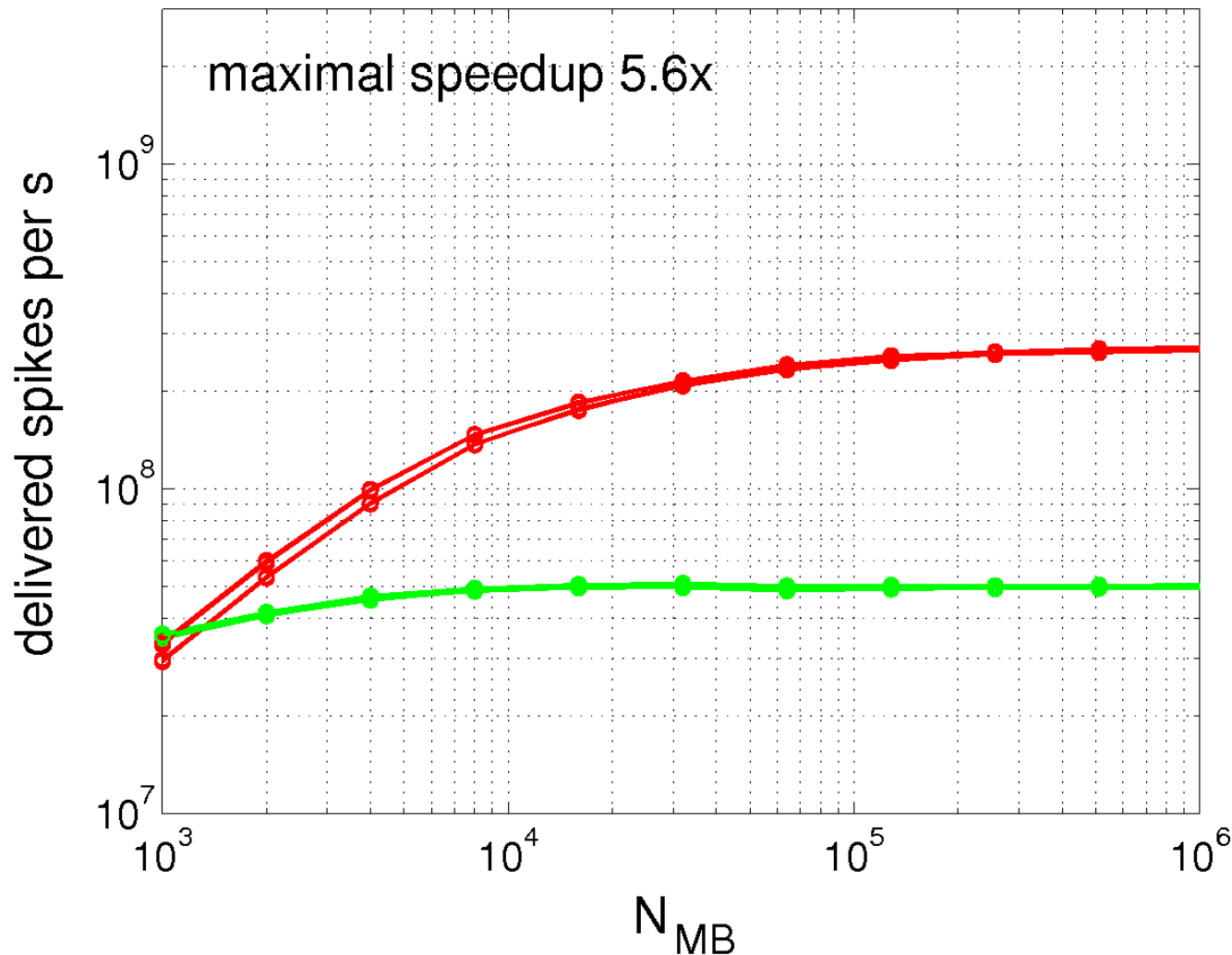
spike # commu-  
nicated to host  
(solid)

# Performance

$$N_{AL} = 1000$$

$$N_{LH} = 20$$

$$N_{LB} = 100$$



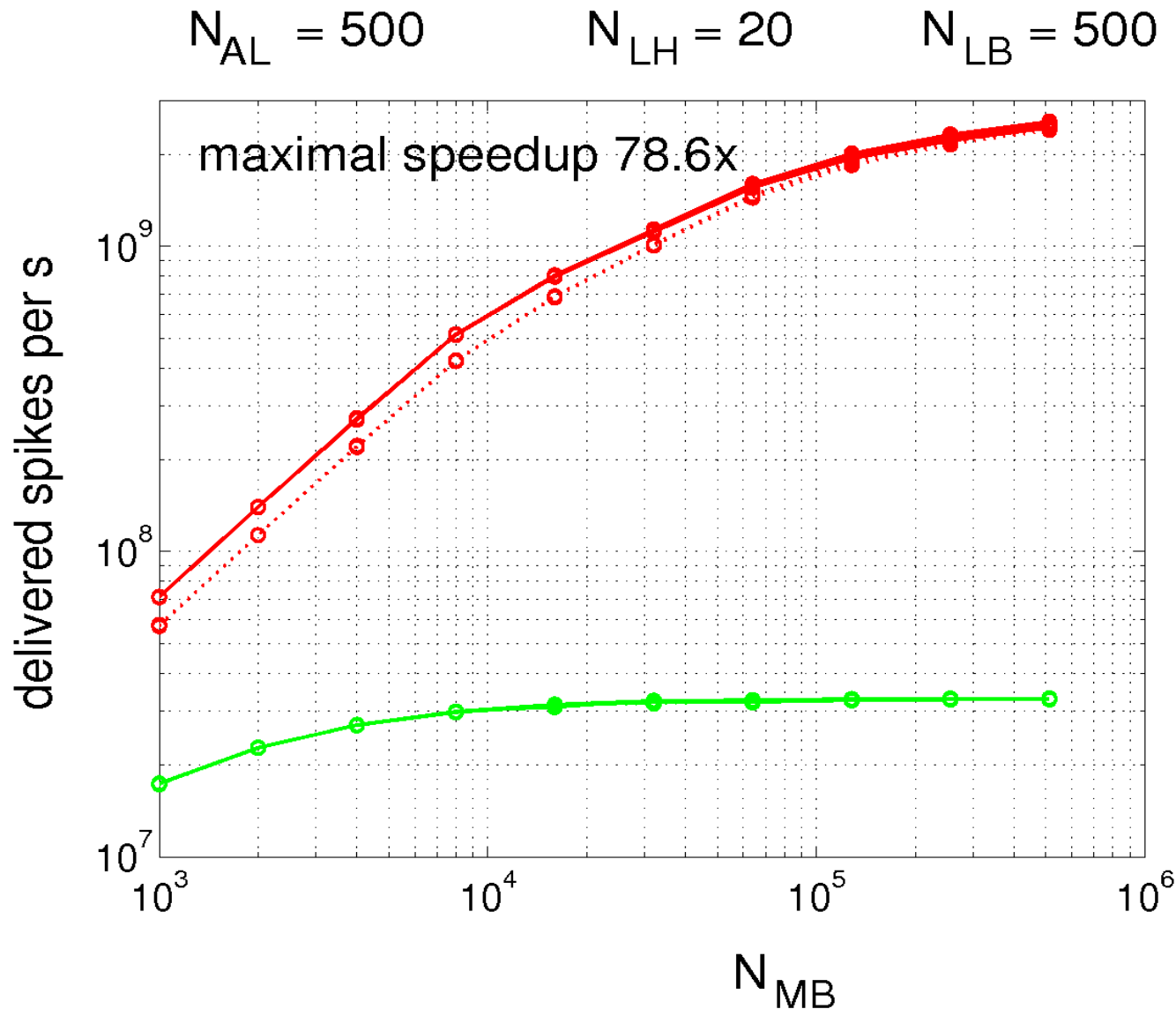
AL-MB:  
50 % all-to-all

Homogeneous  
conductances

spikes commu-  
nicated to host  
(dotted)

spike # commu-  
nicated to host  
(solid)

# Performance



AL-MB:  
50 % all-to-all

Individual  
conductances

spikes commu-  
nicated to host  
(dotted)

spike # commu-  
nicated to host  
(solid)

# Conclusions

- Using a C++/CUDA code generation approach has several advantages:
  - Model specific optimisations at compile time
  - Hardware specific optimisations at compile time
  - Can provide unlimited number of different models but actual simulations stay lean and mean
- GeNN is freely extendible with few constraints
- Low level code is accessible if desired/needed
- New hardware capability can be accommodated

# Outlook

- “Mature” the package
  - Extension to CUDA 4
  - More HW-specific optimisations
  - More connectivity pattern optimisations (e.g. Fidjeland “scatter-gather” strategy)
  - Larger library of predefined model elements, delays
- Find out whether there is a user base & form developer community
- Python API, neuroML API
- Multi-device parallelisation (p-threads)
- Multi-host parallelisation (mpi) ...

# Ambitious Outlook

- Could GeNN applications be services on CARMEN?
  - Need a (good enough) NVidia GPU
  - NVidia (nvcc) compiler on compute host
  - CUDA runtime libraries on compute host
- ... I don't see any particular obstructions ...

# Acknowledgments

- Ramon Huerta
- NVidia professor partnership (2x Quadro FX 5800 cards donated)
- Funders:



<http://www.sourceforge.net/projects/genn>