Automatically generating HPC-optimized code for simulations using neural mass models in The Virtual Brain

Sandra Diaz-Pier[1], Alexander Peyser[1], Marmaduke Woodman[2], Jan Fousek[2], Viktor Jirsa

1. Forschungszentrum Jülich GmbH. Institute for Advanced Simulation, Jülich Supercomputing Centre (JSC), SimLab Neuroscience, JARA, 52425 Jülich, Germany
2. Institut de Neurosciences des Systèmes, Aix Marseille Université, Marseille, France

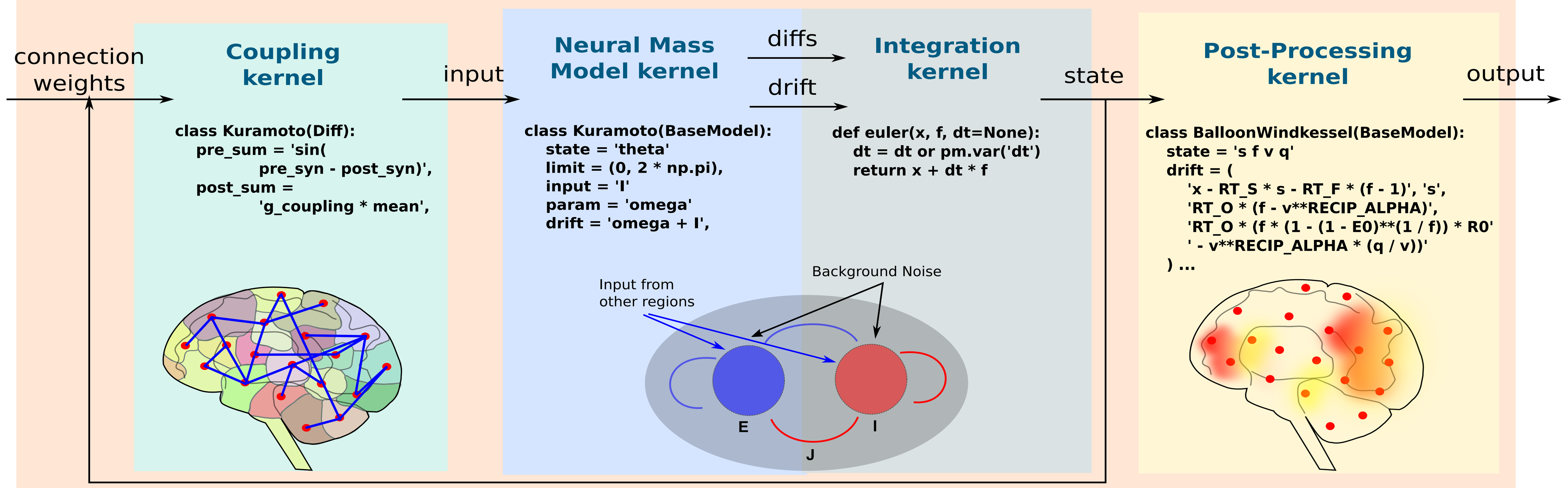
Motivation

- High performance computing is becoming every day a more accessible and desirable concept for researchers in neuroscience.
- Design code to utilize the full power of supercomputers, GPUs and other computational accelerators in a dynamic, maintainable, scalable and robust fashion.

Optimize the workflows and models currently available in The Virtual Brain software (Sanz Leon et al. 2013).

Our approach

- Describe your neural mass model with a high level language.
- Combine it with an integration kernel and a coupling kernel to build a network workflow.
- Define a post-processing kernel.
- The active DSL model representation can be interrogated by different providers to automatically generate and run platform specific code.



Serial loop over time steps

```
osc = model.Kuramoto()
osc.dt = 1.0
osc.const['omega'] = 10.0 * 2.0 * np.pi / 1e3
cfun = coupling.Kuramoto(osc)
scm = scheme.EulerStep(osc.dt)
knl = transforms.network_time_step(osc, cfun, scm)
```

Parallel over parameter sets

```
knl = lp.to_batched(knl, subject, [a, delays],
i_subject, sequential=False)
```

Scales linearly:
10x bigger computer = 10x more data processed in the same time!

Same high level code, multiple target platforms!

CUDA Provider

- High performance utilizing the computational capabilities of GPUs.
- Enables large parallel parameter searches in short time.

OpenCL Provider

- Benefit from different OpenCL platforms like GPUs, CPUs and FPGAs.
- High flexibility, clear code which can be

Numba Provider

- Easy integration to python code.
- Optimized routines which run on any CPU.
- JIT generation of LLVM code.
- Flexibility to move into the CUDA version of numba, which allows seamless GPU usage from python.

Performance results

Numba CUDA speedup against Numba for different load and #of threads

Execution times for different targets and loads

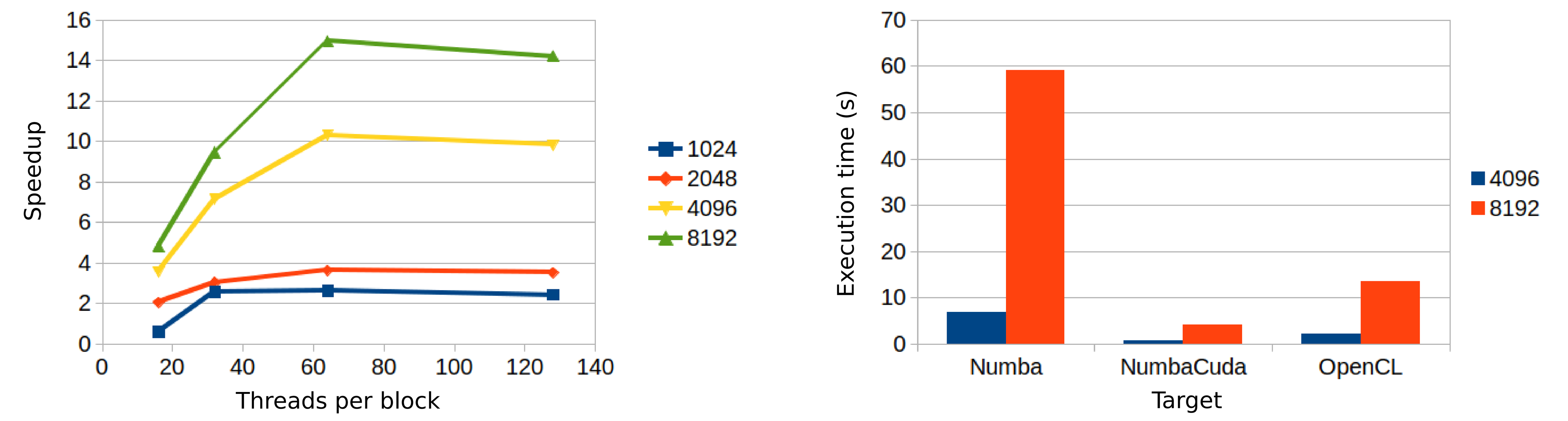


Figure 1: Numba CUDA, Numba and OpenCL runs performed on the Jureca cluster (GPU partition) of the Jülich Supercomputing Centre with a test kernel.

CUDA code performance analysis

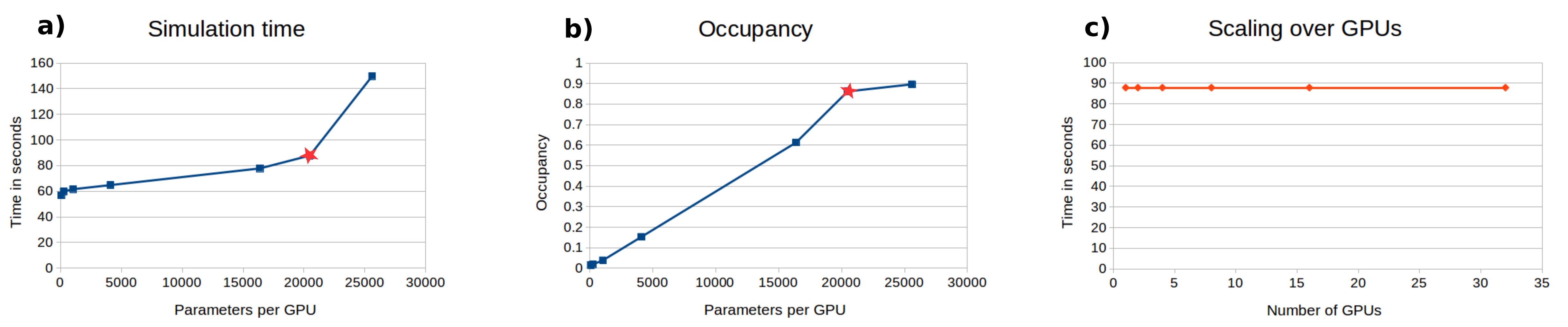


Figure 2: Performance of the CUDA code using the Kuramoto model with changing global coupling and connection speed. a) Simulation time of executing different numbers of parallel simulations on a single GPU. b) GPU occupancy with increasing number of parallel simulations. c) Scaling over different numbers of GPUs with 21600 parallel simulations each. Runs performed on the Jureca cluster (GPU partition) of the Jülich Supercomputing Centre

Example of automatically generated code for a test kernel

Numba + CUDA

```
@ncu.jit
def loopy_kernel_inner(
    n, nnz, row, col, dat, vec, out):
    if -1 + -512*bIdx.y + -1*tIdx.y + n >= 0
and -1 + -512*bIdx.x + -1*tIdx.x + n >= 0:
        acc_j = 0
        jhi = row[1 + tIdx.x + bIdx.x*512]
        jlo = row[tIdx.x + bIdx.x*512]
        for j in range(jlo, -1 + jhi + 1):
            acc_j = acc_j + dat[j]*vec[col[j]]
        out[tIdx.x + bIdx.x*512] =
            (tIdx.y + bIdx.y*512)*acc_j

def loopy_kernel(
    n, nnz, row, col, dat, vec, out):
    loopy_kernel_inner[((511 + n) // 512,
                        (511 + n) // 512),
                        (512, 512)]
              (n, nnz, row, col, dat, vec, out)
```

Numba

```
from __future__ import division, print_function

import numpy as _lpy_np
import numba as _lpy_numba

@_lpy_numba.jit
def loopy_kernel(n, nnz, row, col, dat, vec,
out):
    for i in range(0, -1 + n + 1):
        jhi = row[i + 1]
        jlo = row[i]
        for k in range(0, -1 + n + 1):
            acc_j = 0
            for j in range(jlo, -1 + jhi + 1):
                acc_j = acc_j + dat[j]*vec[col[j]]
            out[i] = k*acc_j
```

Discussion

- Run on different architectures and accelerators like GPUs without changing the top level description of the kernels.
- Hidden complexity to the user, big computational power underneath.
- Great performance boost on GPUs.

Future work

- Further development of the DSL.
- Integration with hyperparameter optimization and interactive visualization frameworks to enhance the parameter space exploration power.
- Integration with other simulation engines such as nest, Arbor and Neuron.

Want to get involved in the development?
Take a look at our code:
https://github.com/the-virtual-brain/tvb-hpc

We are hiring!

We are looking for software developers, PhDs and PostDocs in related areas of computational neuroscience to further develop our HPC tools. If you are interested in joining our project please send your CV to a.peyser@fz-juelich.de

Acknowledgments

We would like to thank our collaborators Lia Domide, Mihai Andrei, Vlad Prunar, Petra Ritter, Michael Schirner and Olaf Sporns. The authors would also like to acknowledge the support by the Excellence Initiative of the German federal and state governments, the JARA and CRCNS grant and the Helmholtz Association through the portfolio theme SMHB and the Initiative and Networking Fund. In addition, this project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 720270 and 785907 (HBP SGA1 & 2).