Myriad : a transparently parallel GPU-based simulator for densely integrated biophysical models

Pedro Rittner[1] & Thomas A. Cleland[2]

[1]Dept. Computer Science, [2]Dept. Psychology, Cornell University, Ithaca, NY

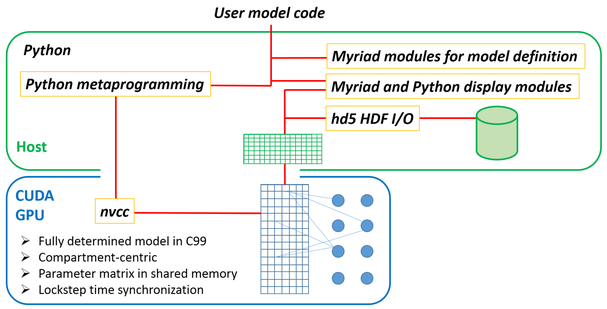
Computational Physiology Laboratory

187.02 TT51

1 Why another simulator?

- Larger network models demand multicore computational tools. However, *densely integrated* network models, in which many nodes must update one another at every timestep (such as many biophysical models), are ill-suited for execution on computational clusters due to slow inter-node communication.
- Current general-purpose implementations are difficult to parallelize and require special coding to achieve limited multiprocessing capabilities. An ideal solution would separate the biophysical problems from the optimization problems.
- GPU hardware is a promising tool for mesoscale parallel simulations, but effective use requires a simulator designed specifically with the strengths and limitations of GPU hardware in mind.

2 Design principles of the *Myriad* simulator



(A) Design a radically granular low-level implementation framework enabling the "trivial parallelization" of any model.

- Myriad removes all hierarchy from compartmental models, recognizing only two computational elements: **compartments** with passive properties and **mechanisms** that connect exactly two compartments in exactly one direction.
- An integral number of these computational elements can be executed on any available core.
- Communication among computational elements is based on a uniform memory access (UMA) shared memory architecture, with each core having equal access to shared memory.

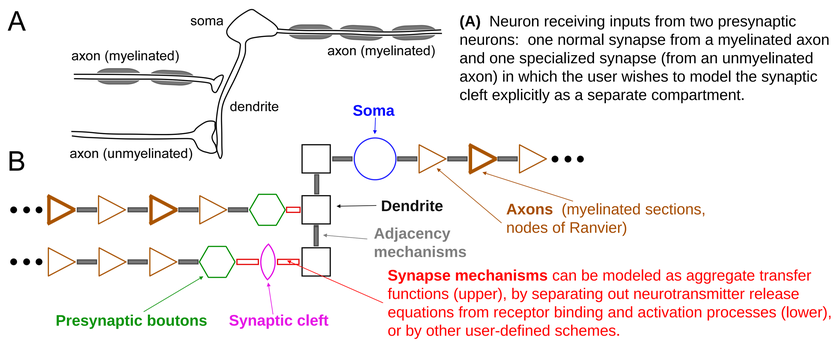
(B) Take maximum advantage of this parallelization capacity by enabling execution on GPUs, including consumer video cards.

- Because of its radically granular architecture, and GPU barrier synchronization, Myriad is fully thread-scalable to any number of available GPU threads without explicit optimization.
- The reduced GPU instruction set limits the extensibility of GPU-enabled applications. We have circumvented this limitation by developing the first object-oriented programming (OOP) model that runs natively on GPUs under CUDA, including on-GPU dynamic type inference and built-in inheritance (Rittner and Cleland, 2014).

(C) Improve ease-of-use with a separate user interface level based on familiar concepts and hierarchies.

- Best practices in model definition include the flexible rescaling of parameters such as model dimensions and numbers of compartments, the abstraction of neuronal regions into *sections* that share properties, the hierarchical segregation of neuron (template) definitions from network definition, and the inclusion of familiar concepts such as an extracellular reference space (Carnevale and Hines, 2006).
- To adhere to these best practices, Myriad incorporates a separate "*user level*", written in Python, in which users construct models using tools from the Python *myriad* module. Python metaprogramming is then used to generate "*implementation level*" C code on the fly for execution on GPU or CPU cores.
- Users retain full flexibility in model definition at the Python level because of the minimal OOP model constructed for the implementation level. This enables the end-user to define wholly novel objects and mechanisms such as ion channels (or even ions), by defining fully extensible, user-defined models to be run on GPU. This innovation also enables models to be written identically for CPU and GPU execution, switching between the two via a compiler option.
- Importantly, the user does not need to write any code specifically to enable parallelization of their model, which can be a substantial barrier to end users.

3 Granularity enables highly flexible model design

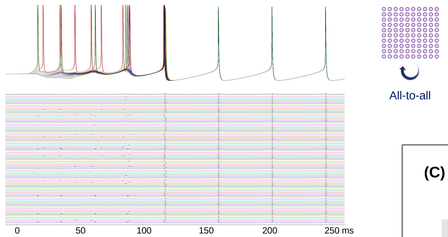


(A) Neuron receiving inputs from two presynaptic neurons: one normal synapse from a myelinated axon and one specialized synapse (from an unmyelinated axon) in which the user wishes to model the synaptic cleft explicitly as a separate compartment.

(B) Schematic example of Myriad compartments and mechanisms. Compartment and mechanism objects incorporate user-defined equations. A single extracellular space compartment here connects to all compartments except the synaptic cleft (multiple extracellular spaces also are supported). *Adjacency* is implemented as paired reciprocal mechanisms.

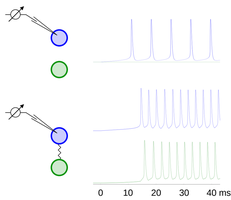
4 Simulation examples

(A) 100 Hodgkin-Huxley neurons coupled with inhibitory synapses to form an *interneuron network gamma* (ING) oscillatory network.



Example simulations coded at the *implementation level* in C

(B) Two Hodgkin-Huxley neurons, separated or connected with a gap junction



(C) Python metaprogramming example and *user level* code prototype

```
import myriad_types
from MyriadObject import MyriadObject
from MHeta import MyriadMeta

class Soma(MyriadObject, metaclass=MyriadMeta):

    capacitance = myriad_types.HDouble
    vm = myriad_types.MyriadScalar("vm", myriad_types.HDouble, pri=True)

    @myriad_method
    def accrete(self,
                a: myriad_types.HInt,
                b: myriad_types.HInt) -> myriad_types.HInt:

        c = 0
        if a > b:
            a = b
            c = a + b
        return c

# ~~~ Resultant Soma.c ~~~~

#ifndef SOMA_H
#define SOMA_H

#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>
#include <math.h>

#include "MyriadObject.h"

typedef int64_t (*saxdot_t)(struct Soma *self, int64_t a, int64_t b);

struct SomaClass;
struct Soma;

extern const void *Soma;
extern const void *SomaClass;

extern int64_t saxdot(struct Soma *self, int64_t a, int64_t b);

extern int64_t super_saxdot(const void *_class, struct Soma *self, int64_t a, int64_t b);

struct Soma
{
    const struct MyriadObject _;
    double capacitance;
    double *vm;
};
struct SomaClass
{
    const struct MyriadClass _;
    saxdot_t my_saxdot_t;
};

#endif

# ~~~ Resultant Soma.c ~~~~

#include "Soma.h"

static void *SomaClass_myriad_ctor(const void *self, va_list *app);
static int64_t Soma_saxdot(struct Soma *self, int64_t a, int64_t b);

const void *Soma;
const void *SomaClass;

static void *SomaClass_myriad_ctor(const void *self, va_list *app)
{
    struct SomaClass* self = (struct SomaClass*) super_ctor(SomaClass, _self, app);

    voidf selector = NULL; selector = va_arg(*app, voidf);

    while (selector)
    {
        const voidf method = va_arg(*app, voidf);

        if (selector == (voidf) saxdot)
        {
            *(voidf *) &self->my_saxdot_t = method;
        }

        selector = va_arg(*app, voidf);
    }

    return self;
}

static int64_t Soma_saxdot(struct Soma *self, int64_t a, int64_t b)
{
    c = 0;
    if (a > b)
    {
        a = b;
        c = a + b;
    }
    return c;
}

int64_t saxdot(struct Soma *self, int64_t a, int64_t b)
{
    const struct MyriadClass* m_class = (const struct MyriadClass*)  myriad_class_of(self);
    assert(m_class->my_saxdot_t);
    return m_class->my_saxdot_t(self,a,b);
}

int64_t super_saxdot(const void *_class, struct Soma *self, int64_t a, int64_t b)
{
    const struct MyriadClass* superclass = (const struct MyriadClass*) myriad_super_(_class);

    assert(superclass->my_saxdot_t);

    return superclass->my_saxdot_t(self,a,b);
}
```

5 Implementational Details

- ❑ **C99 standard-compliant**
  - Extremely portable kernels for x86_64
  - Upgrade path to C11
  - Supports clang, GCC, ICC, possibly MVCC
- ❑ **<pthread.h> on CPU**
  - Portable and low overhead costs
  - Upgrade path to C11 <threads.h>
- ❑ **Python 3.4+ Frontend**
  - Type annotation support
  - "Future-proof" ABI
  - Built-in pip, asyncio
- ❑ **Python 3.x metaclasses**
  - Code creation at parse time, one-time cost
  - No namespace collisions
  - Automatic dependency injection for subclasses
  - Opt-out for advanced users
- ❑ **Incremental compilation**
  - One-time startup cost
  - Uses *make* backend
- ❑ **Efficient analysis IPC**
  - POSIX shared memory
  - Zero copy from simulation kernel to analysis frontend
- ❑ **Zero-cost abstractions**
  - Objects are stored as POD structs with one class pointer.
- ❑ **No memory leaks**
  - Single heap allocation
  - Automatic memory management in kernel.
  - Maximizes stack usage
- ❑ **Barrier synchronization**
  - Supported by pthreads
  - Also in CUDA driver
  - Robust ordering semantics support
- ❑ **Export to NeuroML**
  - Import support currently being investigated.
- ❑ **Data export via Numpy**
  - Supported via automatic attribute conversion.
  - Scipy and Matplotlib support comes "free" as a result.
- ❑ **Automatic Doxygen Documentation**
  - For all C modules
- ❑ **JIT Support for CUDA**
  - Use "fat binary" option to increase compilation time but with faster binaries
- ❑ **Fully-configurable compile-time options**
  - From low-level (e.g. -O optimization level) to high-level (e.g. force heap memory usage).

6 Planned Extensions

- Extend Myriad to a nonuniform memory access architecture to support multiple CUDA cards on a single high-speed bus.
- Implement simulation governor to run multiple instances in series or in parallel (e.g., on distributed-architecture GPU clusters), to support parameter exploration and algorithmic optimization.
- Provide advanced users access to Myriad's code generation API.
- Myriad is an arbitrarily programmable GPU-enabled computational framework that is in principle as appropriate for (e.g.) 3-D spatial diffusion models as for neuronal modeling. Assess Myriad's utility for these different applications, and their synthesis.

7 Interested?

- *Myriad* is an open-source project that soon will be open for community participation.
- If you are interested in **early-stage access as a contributor,** please send a detailed email to both authors describing the reasons for your interest and your relevant skills in Python, C, and GPU coding as well as in neuroscience and related fields.
- If you are interested in **beta testing as an end user,** please send an email to both authors and/or sign up on the provided list.

Author emails: pr273@cornell.edu, tac29@cornell.edu

8 References & Acknowledgments

Rittner P, Cleland TA (2014) The MYRIAD simulator: densely coupled realistic neural networks on GPU. *GPU Technology Conference*, San Jose, CA. http://www.gputechconf.com/

Carnevale NT, Hines ML (2006) The NEURON Book. Cambridge, UK: Cambridge University Press.

This research was supported by an equipment grant from the NVIDIA Corporation.