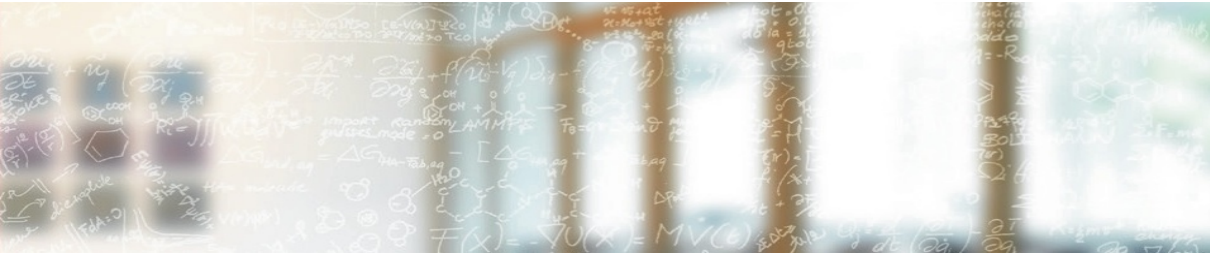
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



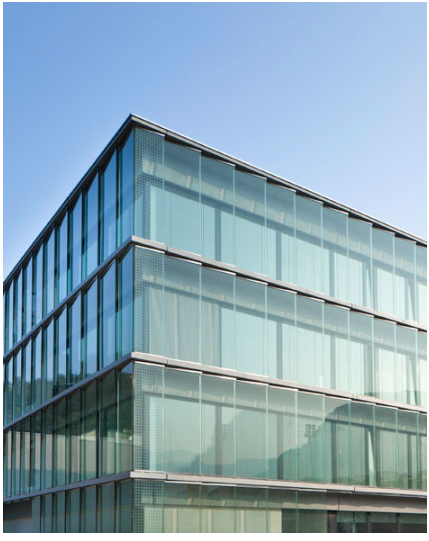
Early experiences on porting NestMC to KNL using intrinsics

Intel KNL: Best practices and experiences

Vasileios Karakasis

April 6, 2017

Outline



- What is NestMC?
  - Goals
  - Design and architecture
  - Critical simulation steps
- Compiling channel and synapse states
  - The KNL backend
- Experiences with intrinsics
  - Helpful tricks
  - Turndowns

What is NestMC?

NestMC is a project to develop

- a new multi-compartmental neuron simulator,
- that is optimized for HPC systems,
- and is easy to integrate into existing workflows.

Who is behind the initiative?

Cross-institutional collaboration

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

JÜLICH
FORSCHUNGSZENTRUM

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

As part of

Human Brain Project

Limitations of current solutions

- There are problems and models that we can't explore with current software and systems
  - Near real-time multi-compartment simulations
  - 'Large' networks
  - Field potential calculations
- New HPC architectures
  - Highly parallel architectures (e.g. Intel KNL)
  - Wide vector operations (e.g. AVX512)
  - Specialized accelerator hardware (GPUs, FPGAs)
- Adapting existing simulators to new architectures is *hard*

NestMC goals

Interoperability

Simulator as library

- Visualization
- Multi-physics
- Multi-scale

Extensibility

Modular internal API

- New integration schemes
- Custom spike communication
- Specialized cells

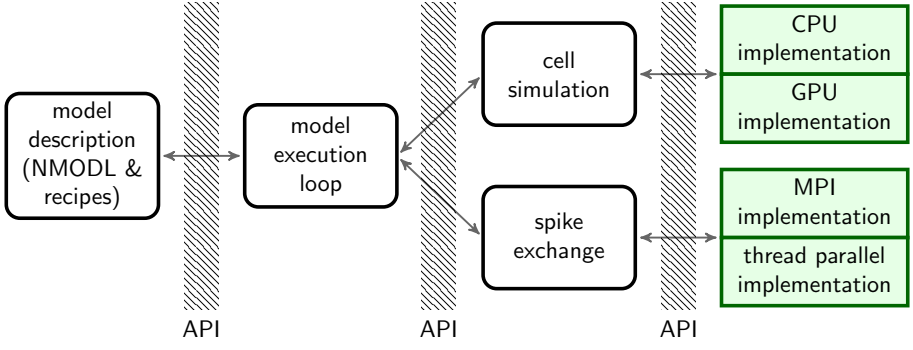
Performance

HPC targeted

- Highly parallel
- GPU and vector targets
- Design for scalability

NestMC design

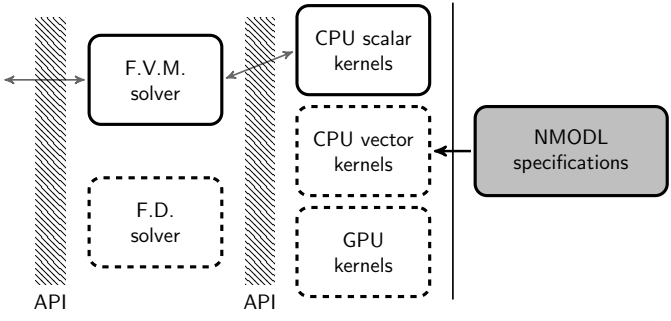
- Modular: components can be substituted according to internal API
- Internal API: ‘thin’ API; type parameterization allows components to determine low-overhead API data structures



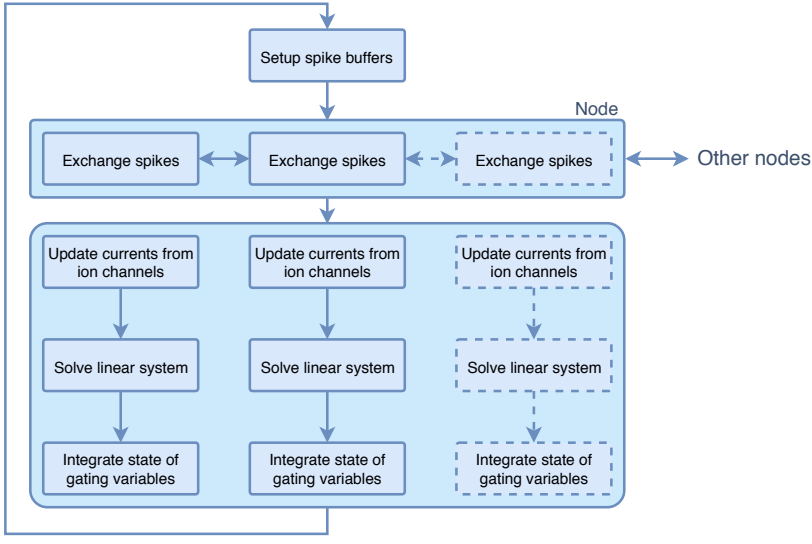
NestMC backends

Cell simulation modules share computational backends for channel and synapse state evolution.

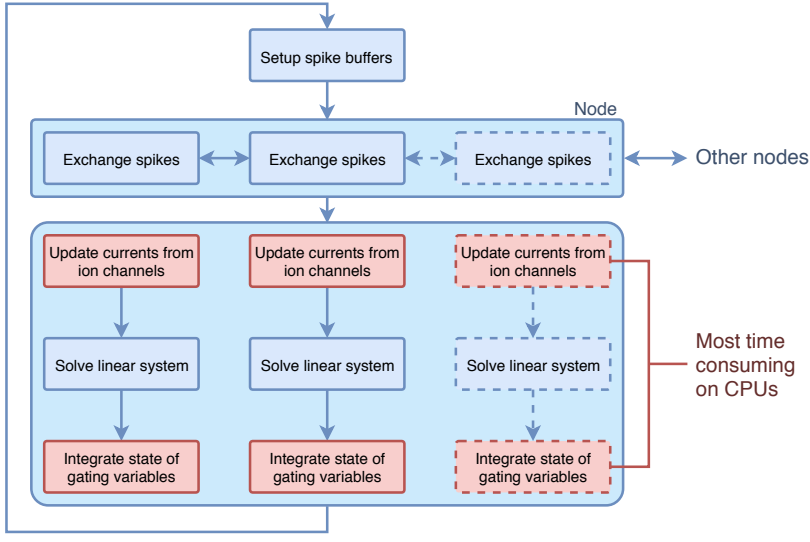
CPU-hosted finite volume cell simulation



Cell simulation timeloop



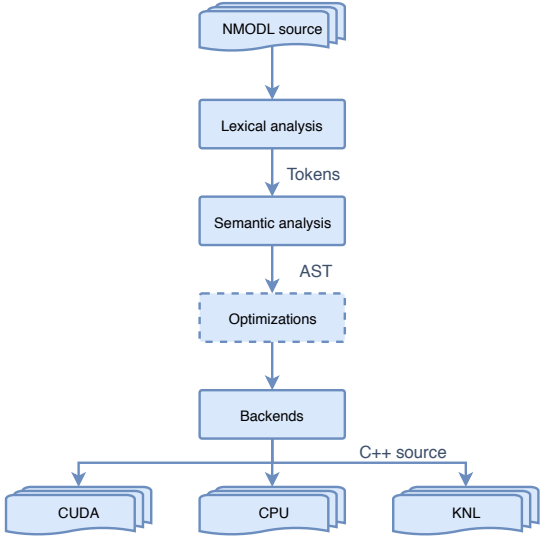
Cell simulation timeloop



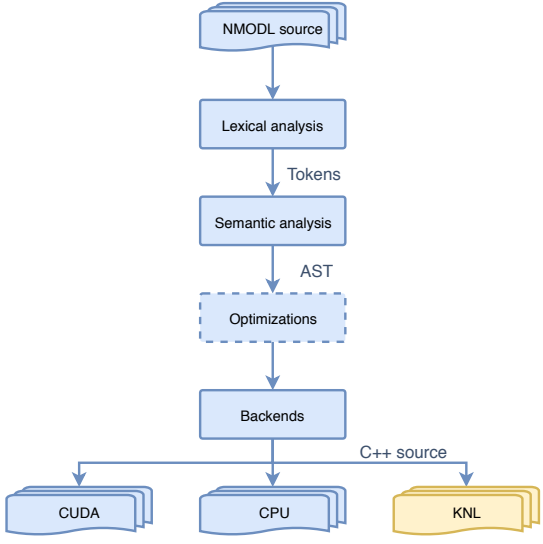
Mechanisms specifications

- Written in NMODL
  - DSL for describing membrane mechanisms of neuron cells
  - Developed for the NEURON simulator
- Translated into C++ with `modcc`
  - Source-to-source compiler from NMODL to C++
  - Generates platform-dependent code for the mechanism

The modcc architecture



The modcc architecture



KNL backend – Baseline

- Standard CPU code, no pragmas
- Rely completely on compiler auto-vectorization capabilities
- Baseline benchmark performance: 49.58s

KNL backend – Guided compiler vectorization

- Proper alignment of data
- Enforce vectorization of critical loops using `#pragma ivdep`
- Benchmark performance: 19.0s (+261%)

KNL backend – Guided compiler vectorization

- Proper alignment of data
- Enforce vectorization of critical loops using `#pragma ivdep`
- Benchmark performance: 19.0s (+261%)

Tricky part:

- Some mechanisms have dependencies between vector elements
- Critical loops rewritten:
  - Blocked on SIMD vector length
  - Inner loop vectorized
  - Manual accumulation of vector elements

KNL backend – Intrinsics

- Original loop rewritten with intrinsics
- Benchmark performance: 19.5s (-2.6%)
  - Out-of-the-box implementation, no tuning yet
  - Unaligned memory access instructions
  - Critical loop is different than the guided version

KNL backend – Intrinsics

Why intrinsics?

- We want to support also non-Intel compilers
- We are writing a compiler backend, so the development overhead of intrinsics is undertaken only once
- Don’t want to depend on another high level C++ vectorization library
- Dive deeper in code optimizations

Helpful tricks with intrinsics

Intrinsics can be chained → prefix notation

g[off_]*(v-e[off_])

```
_mm512_mul_pd(
    _mm512_loadu_pd(&g[off_]),
    _mm512_sub_pd(
        v, _mm512_loadu_pd(&e[off_])
    )
)
```

Helpful tricks with intrinsics

Indirect accesses through gathers and scatters

v = arr[ni[off_]]

```
__m256i vni=_mm256_lddqu_si256(
    (const __m256i *) &ni[off_]);
__m512d v=_mm512_i32gather_pd(vni,arr,8);
// ...
_mm512_i32scatter_pd(arr,vni,v,8);
```

Helpful tricks with intrinsics

Easy access to individual vector elements

- Works around intra-vector dependencies
- `#pragma ivdep` cannot save you here

Access vector elements as a normal array

```
_m512d v = // ...
double *d = (double *) &v;
for (int i=0; i<sizeof(double); ++i) {
    sum += d[i];
}
```

Helpful tricks with intrinsics

- AVX512-ER instructions
  - not IEEE compliant
  - Base-2
  - Not ideal if you need increased accuracy
- Short Vector Math Library (SVML)
  - Look and feel of standard intrinsics, but are actually library calls
  - `_mm512_exp_pd`, `_mm512_log_pd`, `_mm512_pow_pd` etc.
  - Shortcuts of common vector operations (set vector elements, broadcast value to all vector elements etc.)

Intrinsics turndowns

- Can easily get into subtle bugs
- You might not do better than the Intel compiler
- Transcendentals only available by SVML
  - Practically unusable without the Intel compiler
  - Ugly workaround to have GCC link with SVML

Linking GCC with SVML

```
static __m512d _mm512_exp_pd(__m512d v) {
    constexpr int vs = 8;
    double *varr = (double *) &v;
    double ret[vs];
    for (auto i = 0; i < vs; ++i) {
        ret[i] = std::exp(varr[i]);
    }

    return __m512d{ ret[0], ret[1], ret[2], ret[3],
                    ret[4], ret[5], ret[6], ret[7] };
}
```

Compile with `-mveclibabi=svml -mavx512f -Ofast -lsvml`

Conclusions

- Programming with intrinsics is not as difficult as it seems
- The extra effort can pay off with non-Intel compilers
- Finer control on the generated code
- @Intel: please make easily available SVML outside Intel compiler

Conclusions

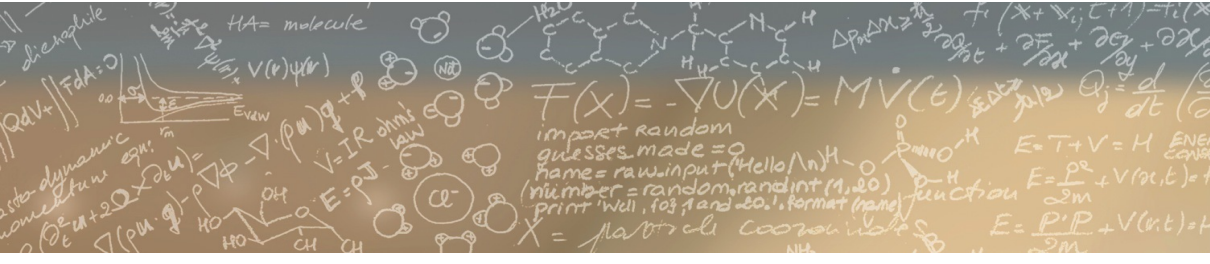
- Programming with intrinsics is not as difficult as it seems
- The extra effort can pay off with non-Intel compilers
- Finer control on the generated code
- @Intel: please make easily available SVML outside Intel compiler

Do it with care and when actually needed

Resources

NestMC prototype:
`https://github.com/eth-cscs/nestmc-proto`

- License: BSD-3
- Compilers: GCC, Clang, Intel
- Backends: CPU, GPU, KNL
- Distributed model: MPI
- On-node parallelism: TBB, Pthreads



Thank you for your attention