An Optimizing Multi-platform Source-to-source Compiler Framework for the NEURON MODeling Language

Pramod Kumbhar[1], Omar Awile[1], Liam Keegan[1], Jorge Blanco Alonso[1], James King[1], Michael Hines[2], and Felix Schürmann[1](✉)

[1] Blue Brain Project, Ecole Polytechnique Fédérale de Lausanne (EPFL), Campus Biotech, 1202 Geneva, Switzerland
felix.schuermann@epfl.ch

[2] Department of Neuroscience, Yale University, New Haven, CT 06510, USA

Abstract. Domain-specific languages (DSLs) play an increasingly important role in the generation of high performing software. They allow the user to exploit domain knowledge for the generation of more efficient code on target architectures. Here, we describe a new code generation framework (NMODL) for an existing DSL in the NEURON framework, a widely used software for massively parallel simulation of biophysically detailed brain tissue models. Existing NMODL DSL transpilers lack either essential features to generate optimized code or capability to parse the diversity of existing models in the user community. Our NMODL framework has been tested against a large number of previously published user models and offers high-level domain-specific optimizations and symbolic algebraic simplifications before target code generation. NMODL implements multiple SIMD and SPMD targets optimized for modern hardware. When comparing NMODL-generated kernels with NEURON we observe a speedup of up to 20×, resulting in overall speedups of two different production simulations by ∼7×. When compared to SIMD optimized kernels that heavily relied on auto-vectorization by the compiler still a speedup of up to ∼2× is observed.

Keywords: NEURON · HPC · DSL · Code generation · Neuroscience

1 Introduction

The use of large scale simulation in modern neuroscience is becoming increasingly important (e.g. [2,24]) and has been enabled by substantial performance progress in neurosimulation engines over the last decade and a half (e.g. [14,17,19,20,27,28]). While excellent scaling has been achieved on a variety of platforms with the conversion to vectorized implementations, domain specific knowledge expressed in the models is not yet optimally used. In other fields, the use of DSLs and subsequent code-to-code translation have been effective in generating efficient codes and allowing easy adaptation to new architectures

© Springer Nature Switzerland AG 2020
V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12137, pp. 45–58, 2020.
https://doi.org/10.1007/978-3-030-50371-0_4

[8,9,30,31]. This is becoming more important as the architectural diversity of hardware is increasing.

Motivated by these observations, we have revisited the widely adopted NEURON simulator [11], which enables simulations of biophysically detailed neuron models on computing platforms ranging from desktop to petascale supercomputers, and which has over 2,000 reported scientific studies using it. One of the key features of the NEURON simulator is extendability via a domain specific language (DSL) layer called the NEURON Model Description Language (NMODL) [12]. NMODL allows the neuroscientist to extend NEURON by incorporating a wide range of membrane and intracellular submodels. The domain scientist can easily express these channel properties in terms of algebraic and ordinary differential equations, kinetic schemes in NMODL without worrying about lower level implementation details.

The rate limiting aspect for performance of NEURON simulations is the execution of channels and synapses written in the NMODL DSL. The code generated from NMODL often accounts for more than 80% of overall execution time. There are more than six thousand NMODL files that are shared by the NEURON user community on the ModelDB platform [29]. As the type and number of mechanisms differ from model to model, hand-tuning of the generated code is not feasible. The goal of our NMODL Framework is to provide a tool that can parse all existing models, and generate optimized code from NMODL DSL code, which is responsible for more than 80% of the total simulation time. Here we present our effort in building a new NMODL source-to-source compiler that generates C++, OpenMP, OpenACC, CUDA and ISPC targeting modern SIMD hardware platforms. We also describe several techniques we employ in the NMODL Framework, which we believe to be useful beyond the immediate scope of the NEURON simulation framework.

2 Related Work

The reference implementation for the NMODL DSL specification is found in *nocmodl* [13], a component in the NEURON simulator. Over the years *nocmodl* underwent several iterations of development and gained support for a number of newer language constructs. One of the major limitations of *nocmodl* is its lack of flexibility. Instead of constructing an intermediate representation, such as an *Abstract Syntax Tree* (AST), it performs many code generation steps on the fly, while parsing the input. This leaves little room for performing global analysis, optimizations, or targeting a different simulator altogether. The CoreNEURON [19] library uses a modified version of *nocmodl* called *mod2c* [3], which duplicates most of the legacy code and has some of the same limitations as *nocmodl*. *Pynmodl* [23] is a Python based parsing and post-processing tool for NMODL. The primary focus of *pynmodl* is to parse and translate NMODL DSL to other computational neuroscience DSLs but does not support code generation for a particular simulator. The *modcc* source-to-source compiler is being developed as part of the Arbor simulator [1]. It is able to generate from NMODL DSL code,

optimized C++/CUDA to be used with the Arbor simulator. It only implements a subset of the NMODL DSL specification and hence is only able to process a modest number of existing models available in ModelDB [29]. For a more comprehensive review of current code-generator techniques in computational neuroscience we refer the reader to Blundell et al. [4]. Other fields have adopted DSLs and code generation techniques [9,25,31] but they are not yet fully exploited in the context of NMODL. We conclude that current NMODL tools either lack support for the full NMODL DSL specification, lack the necessary flexibility to be used as a generic code generation framework, or are unable to adequately take advantage of modern hardware architectures, and thus are missing out on available performance from modern computing architectures.

3 NMODL DSL

In most simple terms, the NEURON simulator deals with two aspects of neuronal tissue simulations: 1) the exchange of spiking events between neuronal cells and 2) the numerical integration of a spatially discretized cable equation that is equipped with additional terms describing transmembrane currents resulting from ion channels and synapses. The NMODL DSL allows the modelers to efficiently express these transmembrane mechanisms. As an example, many models of neurons use a non-linear combination of the following basic ordinary differential equation to describe the kinetics of ion channels first developed by Hodking and Huxley [15]:

$$\frac{dV}{dt} = \left[I - \bar{g}_{Na} m^3 h (V - V_{Na}) - \bar{g}_K n^4 (V - V_K) - g_L (V - V_L)\right] / C \tag{1}$$

$$\frac{dn}{dt} = \alpha_n(V)(1 - n) - \beta_n(V) n \tag{2}$$

$$\frac{dm}{dt} = \alpha_m(V)(1 - m) - \beta_m(V) m \tag{3}$$

$$\frac{dh}{dt} = \alpha_h(V)(1 - h) - \beta_h(V) h \tag{4}$$

where V is the membrane potential, I is the membrane current, g_i is conductance per unit area for ith ion channel, n, m, and h are dimensionless quantities between 0 and 1 associated with channel activation and inactivation, V_i is the reversal potential of the ith ion channel, C is the membrane capacitance, and α_i and β_i are rate constants for the ith ion channel, which depend on voltage but not time.

Figure 1 shows a simplified NMODL DSL fragment of a specific ion channel, a voltage-gated calcium ion channel, published in Traub et al. [32]. Our example highlights the most important language constructs and serves as an example for the DSL-level optimizations presented in the following sections. The NMODL language specification can be found in [12].

At DSL level a lot of information is expressed implicitly that can be used to generate efficient code and expose more parallelism, e.g.:

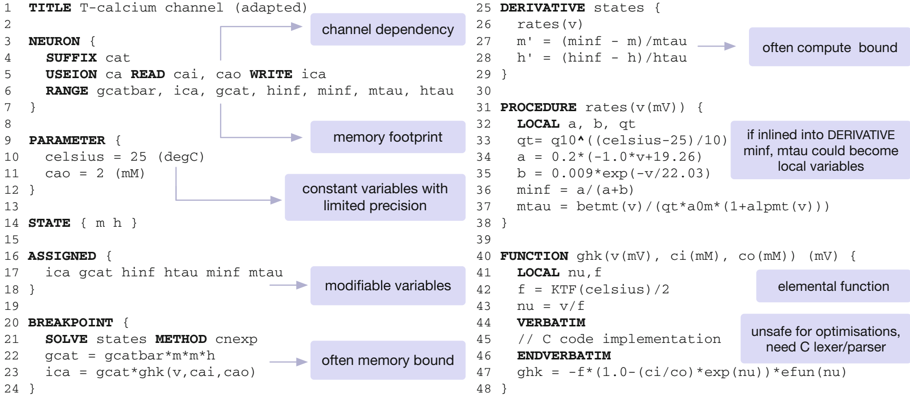

Fig. 1. NMODL example of a simplified model of a voltage-gated calcium channel showing different NMODL constructs and summary of optimization information available at DSL level. Keywords are printed in uppercase and marked with boldface.

- `USEION` statement describes the dependency between channels and can be used to build the runtime dependency graph to exploit micro-parallelism [22]
- `PARAMETER` block describes the variables that are constant, often can be stored with limited precision.
- `ASSIGNED` statement describes modifiable variables and can be allocated in fast memory.
- `DERIVATIVE`, `KINETIC` and `SOLVE` describes ODEs which can be analyzed and solved analytically to improve the performance as well as accuracy.
- `BREAKPOINT` describes current and voltage relation. If this is ohmic then one can use analytical expression instead of numerical derivatives to improve the accuracy as well as performance.
- `PROCEDURE` can be inlined at DSL level to eliminate RANGE variables and thereby significantly reduce memory access cost as well as memory footprint.

To use this information and perform such optimizations, often a global analysis of the NMODL DSL is required. For example, to perform inlining of a `PROCEDURE` one needs to find all function calls and recursively inline the function bodies. As *nocmodl* lacks the intermediate AST representation, this type of analysis is difficult to perform and such optimizations are not implemented. The NMODL Framework is designed to exploit such information from DSL specification and perform optimizations.

4 Design and Implementation

The implementation of the NMODL Framework can be broken down into four main components: lexer/parser implementation, DSL level optimisation passes, ODE solvers, and code generation passes. Figure 2 summarizes the overall architecture of NMODL Framework. As in any compiler framework, lexing and parsing are the

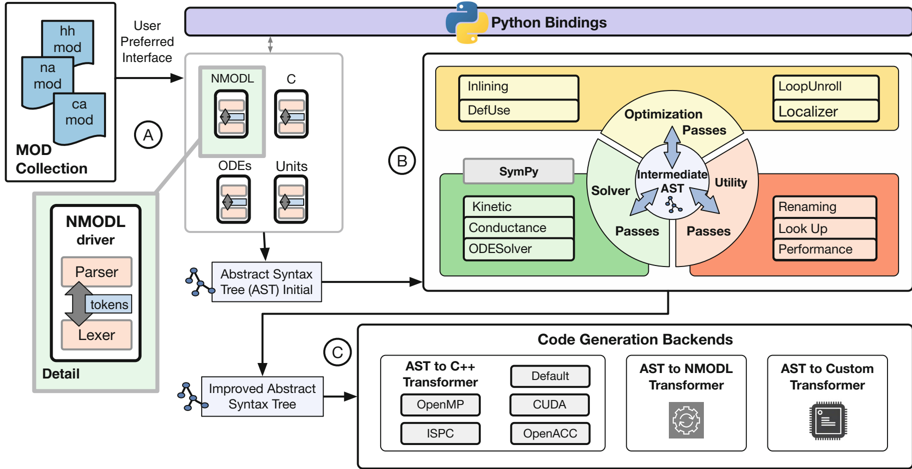

Fig. 2. Architecture of the NMODL Code Generation Framework showing: A) Input NMODL files are processed by different lexers & parsers generating the AST; B) Different analysis and optimisation passes further transform the AST; and C) The optimised AST is then converted to low level C++ code or other custom backends

first two steps performed on an input NMODL. The lexer implementation is based on the popular flex package and bison is used as the parser generator. The ODEs, units and inline C code need extra processing and hence separate lexers and parsers are implemented. DSL level optimizations and code generation are main aspects of this framework and discussed in detail in subsequent sections.

4.1 Optimization Passes

Modern compilers implement various passes for code analysis, code transformation, and optimized code generation. Optimizations such as constant folding, inlining, and loop unrolling are commonly found in all of today's major compilers. For example, the LLVM compiler framework [21] features more than one hundred compiler passes. In the context of the NMODL Framework, we focus on a few optimization passes with very specific objectives. By taking advantage of domain-specific and high-level information that is available in the DSL but later lost in the lower level *C++* or *CUDA* code, we are able to provide additional significant improvements in code performance compared to native compiler optimizations. For example, all NMODL `RANGE`, `ASSIGNED`, and `PARAMETER` variables are translated to *double* type variables in *C++*. Once this transformation is done, *C/C++* compilers can no longer infer these high-level semantics from these variables. Another example is `RANGE` to `LOCAL` transformations with the help of `PROCEDURE` inlining discussed in Sect. 3. All `RANGE` variables in the NMODL DSL are converted to array variables and are dynamically allocated in *C++*. Once this transformation is done, the *C/C++* compiler can only do limited optimizations.

To facilitate the DSL level optimizations summarized in Sect. 3, we have implemented the following optimization passes.

Inlining: To facilitate optimizations such as RANGE to LOCAL conversion and facilitate other code transformations, the *Inlining* pass performs code inlining of PROCEDURE and FUNCTION blocks at their call sites.

Variable Usage Analysis: Different variable types such as RANGE, GLOBAL, ASSIGNED can be analysed to check where and how often they are used. The *Variable Usage Analysis* pass implements *Definition-Use (DU)* chains [18] to perform data flow analysis.

Localiser: Once function inlining is performed, *DU* chains can be used to decide which RANGE variables can be converted to LOCAL variables. The *Localiser* pass is responsible for this optimization.

Constant Folding and Loop Unrolling: The KINETIC and DERIVATIVE blocks can contain coupled ODEs in WHILE or FOR loop statements. In order to analyse these ODEs with SymPy (see Subsect. 4.4), first we need to perform constant folding to know the iteration space of the loop and then perform *loop unrolling* to make all ODE statements explicit.

4.2 Code Generation

Once DSL and symbolic optimizations (see Subsect. 4.3) are performed on the AST, the NMODL Framework is ready to proceed to the code generation step (cf. Fig. 2). The *C++* code generator plays a special role, since it constitutes the base code generator extended by all other implementations. This allows easy implementation of a new target by overriding only necessary constructs of the base code generator.

To better leverage specific hardware platform features such as SIMD, multi-threading or SPMD we have further implemented code generation targets for OpenMP, OpenACC, ISPC (*Intel SPMD Program Compiler*) and experimentally CUDA. We chose ISPC for its performance portability and support for all major vector extensions on x86 (SSE4.2, AVX2, AVX-512), ARM NEON and NVIDIA GPUs (using NVPTX) giving us the ability to generate optimized SIMD code for all major computing platforms.

We have, furthermore, extended the *C++* target with an OpenMP and an OpenACC backend. These two code generators emit code that is largely identical to the *C++* code generator but add appropriate pragma annotations to support OpenMP shared-memory parallelism and OpenACC GPU acceleration. Finally, our code-generation framework supports CUDA as a main backend to target NVIDIA GPUs.

4.3 ODE Solvers

NMODL allows the user to specify the equations that define the system to be simulated in a variety of ways.

- The `KINETIC` block describes the system using a mass action kinetic scheme of reaction equations.
- The `DERIVATIVE` block specifies a system of coupled ODEs (note that any kinetic scheme can also be written as an equivalent system of ODEs.)
- Users can also specify systems of algebraic equations to be solved. The `LINEAR` and `NONLINEAR` blocks respectively specify systems of linear and nonlinear algebraic equations (applying a numerical integration scheme to a system of ODEs typically also results in a system of algebraic equations to solve.)

To reduce duplication of functionality for dealing with these related systems of equations, we implemented a hierarchy of transformations as shown in Fig. 3. First, any `KINETIC` blocks of mass action kinetic reaction statements are translated to `DERIVATIVE` blocks of the equivalent ODE system. Linear and independent ODEs are solved analytically. Otherwise a numerical integration scheme such as implicit Euler is used which results in a system of algebraic equations equivalent to a `LINEAR` or `NONLINEAR` block. If the system is linear and small, it is solved analytically at compile time using symbolic Gaussian elimination. Optionally, Common Subexpression Elimination (CSE) [7] can then be applied.

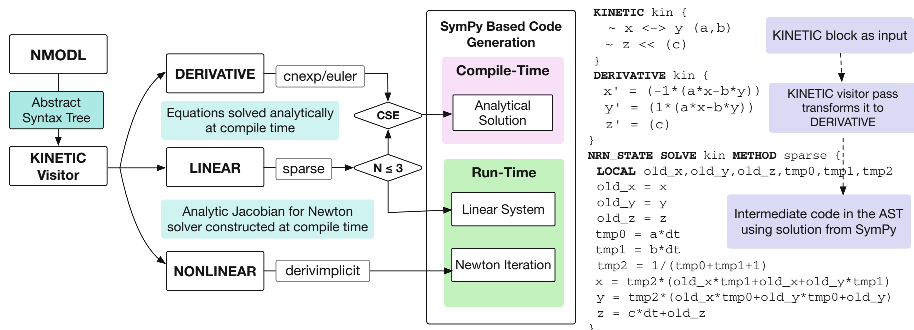

Fig. 3. On the left, unified ODE solver workflow showing ODEs from different NMODL constructs either produces compile–time analytical solutions, or run–time numerical solutions. On the right, example of `KINETIC` block and its transformation to SymPy based solution.

If the system is linear and large, it is solved (at run time) using a lower–upper (LU) matrix decomposition. Non-linear systems of equations are solved at run time by Newton iteration, which makes use of the analytic Jacobian calculated at compile time. The numerical ODE solver uses the Eigen [10] numerical linear algebra *C++* template library, which produces highly optimized and vectorized routines for solving systems of linear equations.

4.4 SymPy

The analytic ODE solver uses SymPy [26], a Python library for symbolic calculations, which can simplify, differentiate and integrate symbolic mathematical expressions. Our analytical solver replaces the purely numerical approach used in other NMODL source-to-source compilers and simulators. It allows us to perform some computations analytically at compile time that were previously carried out at run time at each time step using approximate numerical differentiation.

Linear and independent ODEs have been typically replaced by an analytic solution that treats the coefficients as constant over a time step. NMODL increases the runtime performance by performing algebraic simplification and optionally replacing computationally expensive exponential calculations with the $(1, 1)$ *Pade approximant* [5], consistent with the overall second order correct simulation accuracy (as suggested in [6], and implemented in [1]).

For coupled ODEs, the implicit Euler numerical integration scheme is applied which results in a set of simultaneous algebraic equations. For a linear systems of equations, the `sparse` solver method is used. For non-linear systems, the `derivimplicit` solver method is used. The `sparse` solver chooses from two solution methods, depending on the size of the system to be solved. For small systems (three or less equations), the system is solved by symbolic Gaussian elimination at compile time. The `derivimplicit` solver constructs a system of non–linear equations, which we solve using Newton's method at run time. We therefore compute the system's Jacobian, which is then used in the iterative solver.

5 Benchmarks

To evaluate the achieved performance gains through NMODL, we have performed comprehensive benchmarks on four major production hardware platforms, Intel Skylake 6140, KNL 7230, AMD EPYC 7451 and NVidia Tesla V100. In parallel NEURON and CoreNEURON simulations pure MPI execution expose more parallelism and achieve better performance. To provide a realistic benchmark setup we follow the same approach and perform our measurements on fully loaded nodes (process per physical core).

Benchmarks performed on the Intel platforms to compare auto-vectorization performance with ISPC were compiled with Intel Parallel Studio 2018.1, while all others were compiled with GCC. All benchmarks have been compiled with `-O2 -xHost` and `-O3` flags respectively. For GPU benchmarking we compared performance of Intel Skylake node with a NVidia Tesla V100 GPU.

We selected two brain tissue models: a somatosensory cortex and a hippocampus region model. The first, a somatosensory cortex microcircuit of a young rat published by the Blue Brain Project has 55 layer-specific morphological types and 207 morpho-electrical types [24]. The second, a model of a rat hippocampus CA1 [16] is built as part of the European Human Brain Project and has 13 morphological types and 17 morpho-electrical types. These models are selected

because they are computationally expensive and have a large number of mechanisms which allow us to assess performance benefits for different types of kernels used in production simulations. Based on these two models, we presented results for two benchmarks:

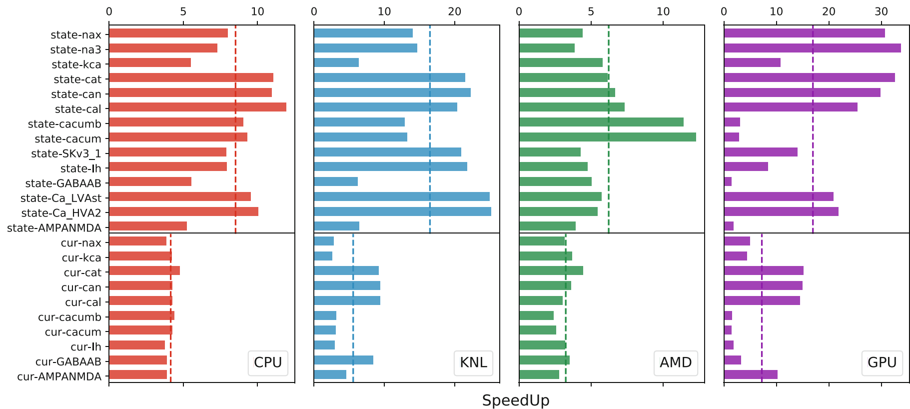

Fig. 4. Speedups of the 14 representative channels from the neocortex and hippocampus models, built using the NMODL Framework with ISPC target and run using CoreNEURON over *nocmodl* and NEURON. The dotted lines denote average speedups of the respective group of kernels

Channel Benchmark: This benchmark consists of 21 different morpho-electrical types selected to include all the unique mechanisms from somatosensory cortex and hippocampus CA1 models. A total of 4,608 cells are created without network connectivity as this benchmark is designed to measure performance of code generation for individual mechanisms. To benchmark GPU performance we created 3,000 artificial cells using mechanisms from this benchmark.

Simulation Benchmark: This benchmark measures the overall performance improvement for whole production simulations. We used 1,000 cells from the somatosensory cortex and hippocampus models simulating one second of biological time using a timestep of 0.025 ms.

6 Results

The code generated from the NMODL DSL accounts for more than 90% of the simulation time in the above-described benchmarks. Each channel is written in the NMODL language and typically contains two kernels: *State Update* (denoted as *state-channel-name*) and *Current Update* (denoted as *cur-channel-name*). For calculating the speedup, we compare the runtimes of these two kernels translated using NMODL Framework with ISPC backend and simulated using CoreNEURON with the same mechanisms compiled with the NEURON

simulator using *nocmodl*. The results of these benchmarks are shown in Fig. 4. We restrict ourselves to most expensive kernels from 14 representative channels. The four columns correspond to the four tested hardware platforms. The dotted lines show the average speedups achieved for the channels. Generally we observe a higher speedup on *State Update* kernels than on *Current Update* kernels. This is due to the *State Update* kernels typically being computationally more expensive, with a higher FLOP per byte ratio than *Current Update* kernels. This is particularly true for GPUs, such as the NVidia V100 platform. The best speedups, particularly for *State Update* kernels, are achieved on Intel KNL. We attribute this to the rather poor performance of the *nocmodl* code generation backend with NEURON. Most of the *Current Update* kernels require atomic operations with indirect memory access, which results in poor kernel performance on all platforms in general, and GPUs in particular. Finally, we notice that especially in the *State Update* kernel the availability of AVX-512 vector units, with optimal memory layout offers a performance advantage as can be seen in the higher performance of the two Intel platforms compared with the AMD EPYC platform, which only offers AVX2.

When looking at top performers we notice that several of the high-level optimizations described in Sects. 4.1 and 4.3 are at least equally if not more important than the generation of vectorized code. We observe that particularly our optimizations on the ODE statements using SymPy based solvers (e.g. *statecacum*) can lead to speedups of more than 12× on AMD EPYC.

Table 1. Absolute runtime in seconds and speedup of 1 s simulated biological time of the hippocampus and somatosensory cortex simulations on Intel Skylake and Intel KNL platform using NEURON with *nocmodl* (NRN-NOCMODL), CoreNEURON with MOD2C (CN-MOD2C) and CoreNEURON with NMODL Framework (CN-NMODL). The total time is further broken down into *State* and *Current Update*, which represent the two main groups of computational kernels generated by the transpiler. Speedup is shown with respect to NEURON.

	Component	Intel Skylake			Intel KNL		
		NRN	CN		NRN	CN	
		NOCMODL	MOD2C	NMODL	NOCMODL	MOD2C	NMODL
Hippocampus	State Update	1089.01 s	310.92 s	145.89 s	3260.16 s	525.89 s	251.81 s
	Current Update	866.81 s	239.52 s	171.99 s	1129.13 s	143.2 s	223.93 s
	Other	157.51 s	84.27 s	67.29 s	869.86 s	348.95 s	266.02 s
	Total	2113.34 s	634.71 s	385.17 s	5259.14 s	1018.04 s	741.76 s
	Speedup	—	3.33×	5.49×	—	5.17×	7.09×
Cortex	State Update	173.29 s	32.81 s	20.39 s	556.63 s	45.09 s	41.73 s
	Current Update	106.86 s	32.38 s	27.34 s	154.29 s	37.18 s	64.56 s
	Other	43.51 s	29.69 s	24.11 s	222.9 s	108.43 s	102.66 s
	Total	323.66 s	94.88 s	71.84 s	933.81 s	190.7 s	208.95 s
	Speedup	—	3.41×	4.51×	—	4.9×	4.47×

Table 1 shows the absolute time and speedup achieved for full simulations of the hippocampus CA1 and somatosensory cortex models. We compare the performance with three different configurations. The first configuration (*NRN-NOCMODL*) uses the *NEURON* simulator with *nocmodl* as the code generation backend. The second configuration (*CN-MOD2C*) uses the *CoreNEURON* library with *MOD2C* as code generation backend. The third configuration (*CN-NMODL*) uses the *CoreNEURON* library with the here-presented NMODL Framework as a code generation backend. Total times are broken down into the above-discussed *State Update* and *Current Update* kernels where the majority of the time is spent in NMODL generated code. The rest of the time is shown as *Other*.

The NMODL Framework shows up to 5.5× speedup on Skylake and up to 7× speedup on the KNL platform. The hippocampus model shows a larger speedup compared to the somatosensory cortex model because it uses *cacum*, *cacumb* and *kca* mechanisms with the *derivimplicit* integration scheme. The Eigen based solver implementation in NMODL offers therefore additional performance improvements. When compared with *CN-MOD2C*, *CN-NMODL* shows a up to 2× performance improvement with NMODL generated *State Update* kernels and up to 1.6× for whole simulation. Note that *CN-MOD2C* is heavily dependent on the auto-vectorization capabilities of the compiler. For example, if the GCC compiler is used instead of Intel, *CN-NMODL* becomes up to 3× and 6× faster compared to *CN-MOD2C* on Intel Skylake and Intel KNL platforms respectively. On the Intel KNL platform the *Current Update* kernels are ∼2× slower in *CN-NMODL* compared to *CN-MOD2C* (highlighted in red). These kernels require indirect addressing due to use of `USEION` constructs. We have found possible performance issues in ISPC when *gather-scatter* instructions and *atomic reductions* are used in the kernels with the target *avx512knl-i32x16*. These issues will be addressed in a future release of the NMODL Framework.

7 Discussion

Many scientific applications do not encode only a single mathematical problem, but scientific users provide the governing equations that need to be integrated by the solvers on a case by case basis. This can impact the success of auto-vectorization and thus strategies are required to allow the user to express the problem at hand on a high level, e.g. through DSLs, that help in producing optimized code.

In this paper we presented a novel NMODL code generation framework for the DSL of the widely used NEURON simulator. The DSL is translated into an AST that lends itself to specific optimization passes before it is handed off to different backends for generation of optimized code for the target platform. We have implemented optimization passes that relate to straight-forward transformation of the DSL code, but also more advanced optimization passes that intercept ODE statements for which an analytical solution can be used instead of having to resort to numerical integration. This functionality is built on top of the SymPy and Eigen libraries.

For code generation we have developed backends for C++ and OpenMP targeting CPUs and ISPC to target a wide variety of CPU architectures providing optimal SIMD performance and reducing the dependency on auto-vectorization capabilities of the compiler. Furthermore, we have developed both a CUDA backend specifically with NVidia GPUs in mind as well as a more generic OpenACC backend.

We have benchmarked kernels from production simulations of two different large-scale brain tissue models on Intel SKL, Intel KNL and AMD EPYC platforms. On those individual kernels, we saw performance improvements from 5× to 20×. In order to test how those kernel improvements translate into speedup of the entire simulations (which use the kernels in different ratios or not at all), we benchmarked production simulations on Intel KNL and Intel SKL platforms. Compared to the regular NEURON simulation environment, we observed a speedup of 6–10×. Compared to an optimized version of the NEURON simulator, CoreNEURON, which heavily relies on auto-vectorization of the compiler, the work presented here nonetheless resulted in a speedup of up to 2×.

Beyond the performance gains, a central goal of our effort is the ability to parse all previously published models. By using the grammar specification from the original NEURON NMODL language, we were able to demonstrate compatibility with 6,370 channels from the public model repository ModelDB. We furthermore took care to maintain the language semantics of the DSL in the AST, providing great flexibility to use NMODL as a generic NMODL parsing framework through its Python API and build new tools on top of it.

More generally, DSL have been used by many other fields to close the gap between the domain scientists and efficient code. Our study brings this capability of automatic generation of efficient and multi-platform code to the computational neuroscience community. Importantly, it is doing so by making extensive use of an abstract intermediate representations to perform optimizations using domain knowledge before it is lost during the translation into a general purpose programming language. This makes it possible to perform symbolic simplifications and equation solving on the level of the intermediate representation. We believe that this approach is applicable to other fields and other DSLs and thus transcends the problem at hand.

Acknowledgements. This study was supported by funding to the Blue Brain Project, a research center of the École Polytechnique Fédérale de Lausanne (EPFL), from the Swiss government's ETH Board of the Swiss Federal Institutes of Technology. The research was also funded by NIH grant number R01NS11613 to the Department of Neuroscience, Yale University, and the European Union's Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreement number 785907 (Human Brain Project SGA2). We would like to thank Antonio Bellotta, Francesco Cremonesi, Ioannis Magkanaris, Matthias Wolf, Samuel Melchior and Tristan Carel for fruitful discussions and their contributions to the NMODL development. The AMD system for benchmarking was provided by Erlangen Regional Computing Center (RRZE).

References

1. Akar, N.A., et al.: Arbor — a morphologically-detailed neural network simulation library for contemporary high-performance computing architectures. In: 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 274–282, February 2019. https://doi.org/10.1109/EMPDP.2019.8671560
2. Arkhipov, A., et al.: Visual physiology of the layer 4 cortical circuit in silico. PLOS Comput. Biol. **14**(11), 1–47 (2018). https://doi.org/10.1371/journal.pcbi.1006535
3. Blue Brain Project: MOD2C - CoreNEURON's converter for mod files to C code (2015). http://github.com/BlueBrain/mod2c
4. Blundell, I., et al.: Code generation in computational neuroscience: a review of tools and techniques. Front. Neuroinformatics **12** (2018). https://doi.org/10.3389/fninf.2018.00068
5. Cambridge, W.H. (ed.): Padé Approximants (Section 5.12): Numerical Recipes: The Art of Scientific Computing, 3rd edn. Cambridge University Press, Cambridge (2007). oCLC: ocn123285342
6. Casalegno, F., et al.: Error analysis and quantification in NEURON simulations. In: Proceedings of the VII European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS Congress 2016), Crete Island, Greece (2016). https://doi.org/10.7712/100016.1892.7366
7. Cocke, J.: Global common subexpression elimination. ACM SIGPLAN Not. **5**(7), 20–24 (1970). https://doi.org/10.1145/390013.808480
8. Deutsch, A., et al.: Morpheus: a user-friendly modeling environment for multiscale and multicellular systems biology. Bioinformatics **30**(9). https://doi.org/10.1093/bioinformatics/btt772
9. DeVito, Z., et al.: Liszt: a domain specific language for building portable mesh-based PDE solvers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011. ACM (2011). https://doi.org/10.1145/2063384.2063396
10. Guennebaud, G., et al.: Eigen v3 (2010). http://eigen.tuxfamily.org
11. Hines, M.L., Carnevale, N.T.: The NEURON simulation environment. Neural Comput. **9**, 1179–1209 (1997)
12. Hines, M.L., Carnevale, N.T.: Expanding NEURON's repertoire of mechanisms with NMODL. Neural Comput. **12**, 995–1007 (2000). https://doi.org/10.1162/089976600300015475
13. Hines, M.: nocmodl - NEURON's converter for mod files to C code (1989). https://github.com/neuronsimulator/nrn/tree/master/src/nmodl
14. Hines, M.: Comparison of neuronal spike exchange methods on a Blue Gene/P supercomputer. Front. Comput. Neurosci. **5** (2011). https://doi.org/10.3389/fncom.2011.00049
15. Hodgkin, A.L., Huxley, A.F.: A quantitative description of membrane current and its application to conduction and excitation in nerve. J. Physiol. **117**(4), 500–544 (1952). https://doi.org/10.1113/jphysiol.1952.sp004764
16. Human Brain Project: Community Models of Hippocampus. https://www.humanbrainproject.eu/en/brain-simulation/hippocampus/
17. Jordan, J., et al.: Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers. Front. Neuroinformatics **12**. https://doi.org/10.3389/fninf.2018.00034

18. Kennedy, K.: Use-definition chains with applications. Comput. Lang. **3**(3), 163–179 (1978). https://doi.org/10.1016/0096-0551(78)90009-7
19. Kumbhar, P., et al.: CoreNEURON: An Optimized Compute Engine for the NEURON Simulator. arXiv:1901.10975 [q-bio], January 2019
20. Kunkel, S., et al.: Spiking network simulation code for petascale computers. Front. Neuroinformatics **8**. https://doi.org/10.3389/fninf.2014.00078
21. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO 2004, p. 75. IEEE Computer Society (2004)
22. Magalhaes, B., Sterling, T., Schuermann, F., Hines, M.: Exploiting flow graph of system of odes to accelerate the simulation of biologically-detailed neural networks, pp. 176–187, May 2019. https://doi.org/10.1109/IPDPS.2019.00028
23. Marin, B.: Pynmodl: Python infrastructure for parsing and generating code from NMODL (2018). https://github.com/borismarin/pynmodl
24. Markram, H., et al.: Reconstruction and simulation of neocortical microcircuitry. Cell **163**(2), 456–492 (2015). https://doi.org/10.1016/j.cell.2015.09.029
25. Membarth, R., Hannig, F., Teich, J., Köstler, H.: Towards domain-specific computing for stencil codes in HPC. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 1133–1138 (2012)
26. Meurer, A., et al.: SymPy: symbolic computing in Python. PeerJ Comput. Sci. **3**, e103 (2017). https://doi.org/10.7717/peerj-cs.103
27. Migliore, M., et al.: Parallel network simulations with NEURON. J. Comput. Neurosci. **21**(2). https://doi.org/10.1007/s10827-006-7949-5
28. Morrison, A., et al.: Advancing the boundaries of high-connectivity network simulation with distributed computing. Neural Comput. **17**(8). https://doi.org/10.1162/0899766054026648
29. Nadkarni, P.M., et al.: ModelDB: an environment for running and storing computational models and their results applied to neuroscience. J. Am. Med. Inform. Assoc. **3**(6), 389–398 (1996). https://doi.org/10.1136/jamia.1996.97084512
30. Rathgeber, F., et al.: PyOP2: a high-level framework for performance-portable simulations on unstructured meshes. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012. IEEE Computer Society (2012). https://doi.org/10.1109/SC.Companion.2012.134
31. Schmitt, C., et al.: ExaSlang: a domain-specific language for highly scalable multigrid solvers. In: 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (2014). https://doi.org/10.1109/WOLFHPC.2014.11
32. Traub, R., Wong, R., Miles, R., Michelson, H.: A model of CA3 hippocampal pyramidal neuron incorporating voltage-clamp data on intrinsic conductances. J. Neurophysiol. **66**, 635–650 (1991). https://doi.org/10.1152/jn.1991.66.2.635