MOD2IR: High-Performance Code Generation for a Biophysically Detailed Neuronal Simulation DSL

George Mitenkov*
Imperial College London
UK

Ioannis Magkanaris*
École Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

Omar Awile
École Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

Pramod Kumbhar
École Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

Felix Schürmann
École Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

Alastair F. Donaldson
Imperial College London
UK

Abstract

Advances in computational capabilities and large volumes of experimental data have established computer simulations of brain tissue models as an important pillar in modern neuroscience. Alongside, a variety of domain specific languages (DSLs) have been developed to succinctly express properties of these models, ensure their portability to different platforms, and provide an abstraction that allows scientists to work in their comfort zone of mathematical equations, delegating concerns about performance optimizations to downstream compilers. One of the popular DSLs in modern neuroscience is the NEURON MODeling Language (NMODL). Until now, its compilation process has been split into first transpiling NMODL to C++ and then using a C++ toolchain to emit the efficient machine code. This approach has several drawbacks including the reliance on different programming models to target heterogeneous hardware, maintainability of multiple compiler back-ends and the lack of flexibility to use the domain information for C++ code optimization. To overcome these limitations, we present MOD2IR, a new open-source code generation pipeline for NMODL. MOD2IR leverages the LLVM toolchain to target multiple CPU and GPU hardware platforms. Generating LLVM IR allows the vector extensions of modern CPU architectures to be targeted directly, producing optimized SIMD code. Additionally, this gives MOD2IR significant potential for further optimizations based on the domain information available when LLVM IR code is generated. We present experiments showing that MOD2IR is able to produce on-par execution performance using a single compiler back-end implementation compared to code generated via state-of-the-art C++ compilers, and can even surpass them by up to 1.26×. Moreover, MOD2IR supports JIT-execution of NMODL, yielding both efficient code and an on-the-fly execution workflow.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Applied computing** → Computational biology.

Keywords: DSL, NEURON, NMODL, Compiler, LLVM, SIMD

ACM Reference Format:
George Mitenkov, Ioannis Magkanaris, Omar Awile, Pramod Kumbhar, Felix Schürmann, and Alastair F. Donaldson. 2023. MOD2IR: High-Performance Code Generation for a Biophysically Detailed Neuronal Simulation DSL. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3578360.3580268

1 Introduction

The last decade has seen a cambrian explosion of compute architectures fueled by the breakdown of Dennard scaling [8] and the insatiable appetite for compute resources [49]. Computer architecture designs have proliferated, catering to both mainstream applications and specialized domains. This trend is expected to continue with more application-specialized architectures [21]. It would be, however, difficult to write software that is able to take advantage of these hardware technologies without the help of modern compiler frameworks. Compilers thus are constantly evolving to keep up with evolving programming language standards, and incorporate optimizations techniques for new instruction set architectures. Still, writing efficient, large-scale and portable HPC applications poses unique challenges to the developer in today's environment of constant innovation in hardware architectures. Parallel programming models like OpenMP, OpenACC, CUDA, HIP, OneAPI, SYCL [19], and performance-portability frameworks like RAJA [5] and Kokkos [17] are trying to bridge the gap between new hardware architectures and the needs of HPC software. But, there is no silver

*These authors share first authorship.

CC '23, February 25–26, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0088-0/23/02.
https://doi.org/10.1145/3578360.3580268

bullet and developers need to retune their programs with each new generation of hardware.

Domain-specific languages (DSLs) play an important role in reducing the performance and knowledge gaps to write efficient programs. Here we focus on the field of computational neuroscience, where DSLs have become particularly useful. Examples include SBML [26], MorphML [15], LEMS [10], and NESTML [43]. These DSLs allow scientists to express the often complex neuron and synapse models in the most appropriate form and leave the work of generating efficient and hardware-optimized simulation code to a transpiler framework [7]. This is particularly important for the study of biophysical models, where simulations have seen an enormous increase in scale and complexity over the last two decades. For example, researchers are building morphologically detailed models of regions of the rodent brain to understand important questions like synaptic plasticity [12], others explore possibilities to run human brain-scale simulations [55]. As demands for scalability increase, neuroscience simulation frameworks undergo constant modernization [4, 11, 45], often to take advantage of the evolving hardware landscape.

The NEURON simulation framework [23] is widely used in the computational neuroscience community for simulating detailed neuronal models. A core feature of NEURON is its DSL, NEURON MOdel Description Language (NMODL) [24], which allows the neuroscientist to describe membrane and synapse mechanisms to model the diverse cell dynamics present in neuronal tissue. While over the years several NMODL transpilers have been proposed, all have in common that they generate C++, which must be then compiled into a library used in the simulation. The NMODL framework [33] is one such code-generation framework. It ensures that NMODL programs run efficiently on modern architectures, by employing a combination of ordinary differential equation (ODE) simplifications, various AST transformations, and generation of C++ code that interfaces with programming models including ISPC, OpenMP, OpenACC, and CUDA. The AST transformations include standard optimizations like inlining, constant folding and loop-unrolling at the DSL level, as discussed in Kumbhar et al. [33, Section 4.1].

This has helped to achieve speedups of up to 10× in production simulations and up to 2× when compared to programs hand-tuned for SIMD processors [24]. Even though NEURON is able to provide an extensible and optimized simulation environment, there are still multiple challenges for developers as well as end-users. First, end-users need to have a fully functional compiler toolchain even when using binary distribution of NEURON. Second, limited auto-vectorization capabilities and lack of SIMD vector math libraries in open-source compilers add a dependency on vendor compilers for achieving the best possible performance. Third, generating C++ code with various back-ends for hardware optimizations means that the NMODL framework contains in effect several code generators, which adds additional complexity and maintenance efforts. Fourth, the ahead-of-time (AOT) compilation used by the NMODL framework limits potential optimizations when certain information about the model is only available at runtime. In addition, it makes the compilation workflow less interactive and attractive for end-users.

In order to address these challenges, in this paper we introduce MOD2IR, an extensible multi-platform code generation framework for NMODL, based on LLVM [34] The four main contributions of the framework are:

1. Reducing the development cost and maintenance burden by leveraging the LLVM infrastructure to target existing and emerging hardware architectures, without sacrificing the performance compared to existing back-ends and compiler toolchains.
2. Building an end-to-end code generation pipeline by using LLVM's code generation capabilities and integrating open-source vector math libraries like SLEEF [50].
3. Mapping NMODL's abstract representation directly into LLVM IR to generate as efficient or better code for CPUs, including scalar and SIMD code, and GPUs.
4. Supporting dynamic code generation and on-the-fly execution using LLVM's JIT-execution infrastructure.

Thus, our approach lends itself to interactive yet high performance use as other projects in other domains have established, such as Julia [6] or JAX [9]. In our experiments, MOD2IR achieves on-par performance when compared to state-of-art compilers such as GCC, Clang, and Intel or NVHPC C++ compilers. For certain benchmarks, MOD2IR outperforms existing solutions showing a speedup of more than 1.2× for both CPU and GPU platforms.

2 Background

2.1 NEURON and NMODL DSL

NEURON is a simulation environment which allows users to represent the diversity of electrical and biophysical properties of nerve cells, and to model the dynamics of individual membrane channels and synapses. Developed over the last four decades, NEURON is one of the most popular software simulators in computational neuroscience, as evidenced by citations from more than 2,500 scientific works, collectively featuring more than 750 models that range from single-cell studies to large networks simulating entire brain regions.

In NEURON, individual nerve cells are treated as a tree of branched cables and the electrical activity of cells is modeled using a discretization of the cable equation [54], giving rise to a set of coupled ODEs. To model the ion channels in membrane and synapses, one needs to consider additional ODEs that the simulator must solve at every time integration step. These ion channel and synapse types including their differential equations are specified in NMODL. This abstraction via a DSL is crucial as it allows domain experts to focus on the scientific problem at hand rather than low-level programming details. Figure 1 shows an example of such a

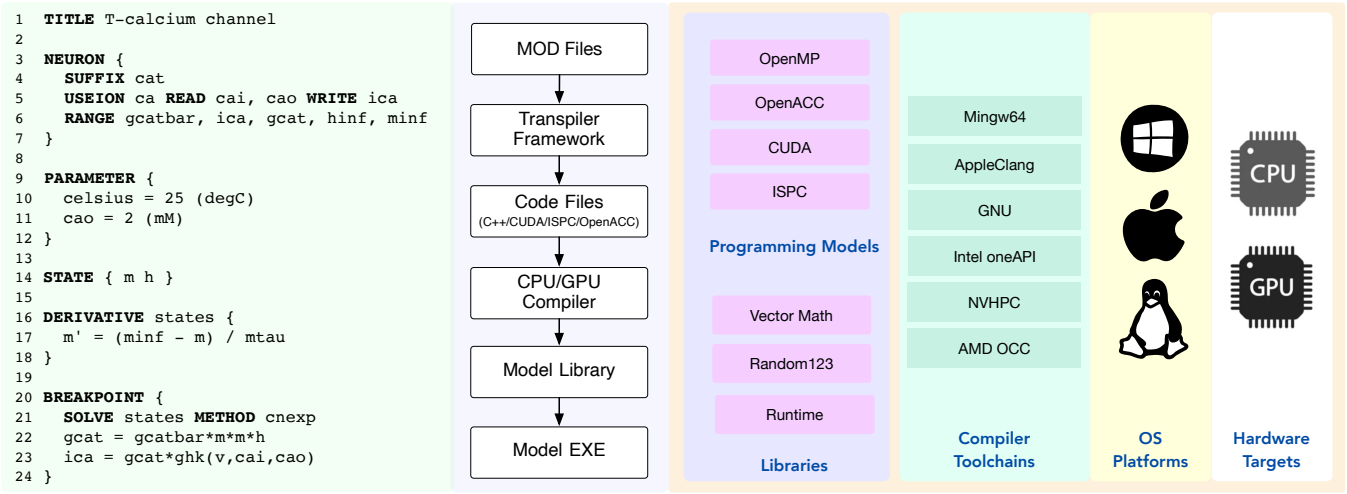

Figure 1. On the left, a simplified example of an NMODL program describing an ion-channel. In the middle, the code generation workflow translating the model to C++ and then creating a model-specific binary that is used for the simulation. On the right, the complexity of supporting various CPU and GPU architectures with different compiler toolchains and programming models. Software portability and maintenance are an additional challenge when supporting different operating systems.

channel specification using NMODL. This specific example and NMODL specification are already discussed in [33].

From a code generation and simulator performance perspective the runtime execution profile of NEURON simulations is typically dominated by compute kernels of the ion channel and synapse models generated from NMODL. The computational cost and performance properties of such kernels and simulations have been extensively studied in [13, 14]. In this work we are interested in the `DERIVATIVE` and `BREAKPOINT` blocks in the Figure 1 that are converted to a compute-intensive state update kernel (*nrn_state*) and a memory-bound current update kernel (*nrn_cur*). The first calculates voltage updates, while the latter calculates the changes in various ionic currents and their contribution to the total current within each discretization element of the neuron. The cumulative runtime of all *nrn_state* and *nrn_cur* kernels typically accounts for more than 90% of the total simulation time and hence are the targets for optimal code generation in our work.

2.2 Code Generation Workflow and Complexity

To run the simulation, a model written in NMODL is transpiled to create a model-specific library (see Figure 1). A channel or synapse model is written in NMODL as a collection of `MOD` files, which are transpiled into `C++` files, and then compiled into a model-specific library. Depending on the target hardware, this involves compiling for a host architecture (CPU) and additionally an accelerator (e.g. GPU). When simulating a model, this model-specific library is loaded by the NEURON distribution that is installed by a user.

Figure 1 also depicts the complexity arising from supporting such a DSL code generation workflow on different OS platforms, compiler toolchains, programming models to target CPU/GPU architectures, and underlying dependencies with vendor-specific libraries. As discussed in Section 1, compiler toolchains like GCC and Clang are commonly available and used by end-users. But until now the NMODL transpiler framework has largely relied on vendor compilers for efficient SIMD code generation and GPU offload support. This leads to a significant maintenance burden to support such a complex software stack.

3 Design and Implementation of MOD2IR

We now describe the architecture of MOD2IR, implementation of code generation pipelines, and integration with open source and vendor vector-math libraries. We also discuss how, by leveraging LLVM's JIT infrastructure, MOD2IR provides a Python API that enables on-the-fly model execution.

3.1 Architecture

MOD2IR follows a conventional modular design with three major components, depicted in Figure 2: 1) an AST visitor that prepares the NMODL AST for code generation; 2) an AST visitor that produces LLVM IR; and 3) multiple passes that specialize and optimize the IR. The optimized IR can then be compiled ahead-of-time, or JIT-compiled at runtime. Lexing and parsing to yield the AST for an NMODL program is performed by the NMODL framework on which MOD2IR is built, and hence are not shown in the architecture diagram.

As the first step, MOD2IR populates a `CodegenConfig` object (1) which holds information about the target architecture and guides the code generation workflow. MOD2IR then takes the AST generated by the NMODL framework and passes it to the `Lowering` visitor (2), which yields code in a

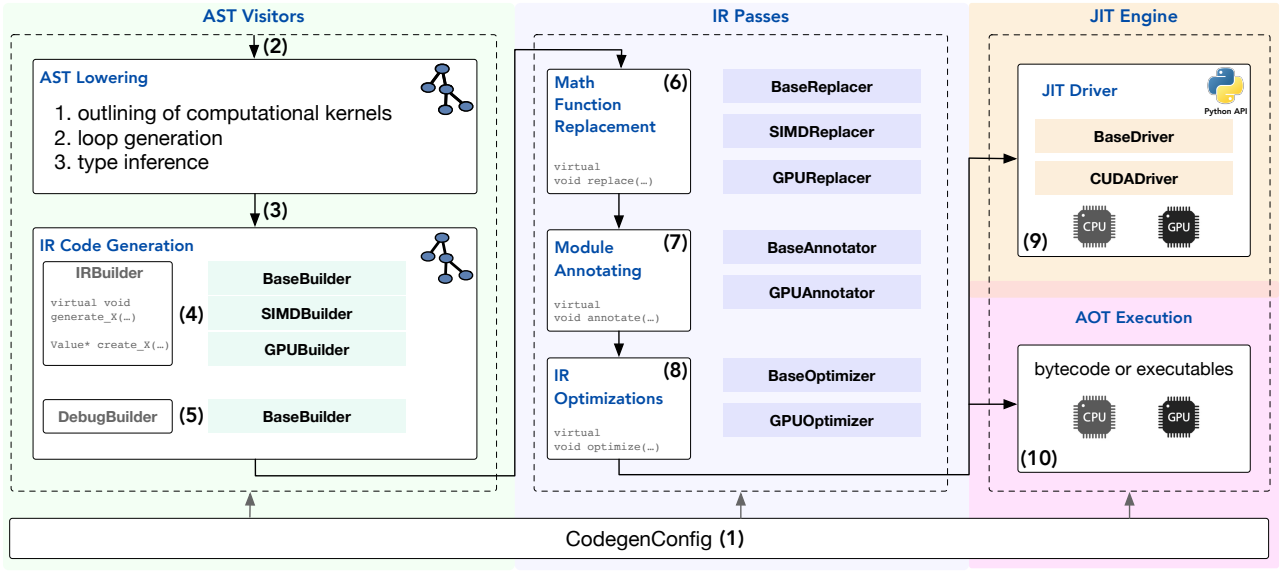

Figure 2. Architecture of MOD2IR: On the left, AST visitors which are used to prepare the NMODL AST for code generation and to emit LLVM IR for various back-ends. In the middle, IR passes to further specialize and optimize the generated IR. On the right, two execution workflows for JIT- and AOT-compilation.

C-like target-agnostic abstract representation ready for code generation (see Section 3.2.1) and includes all computational kernels (*nrn_state*, *nrn_cur*). We refer to this as the *lowered NMODL AST*. Next, the lowered NMODL AST is passed to `IRCodegen` visitor (3), which is responsible for generating LLVM IR for computational kernels. IR generation logic resides in builders: classes with a common code generation interface but aimed at different target platforms (4). Currently, MOD2IR provides three code generation builders: 1) a default builder that generates serial, non-vectorized code; 2) a SIMD builder tailored for vector code generation; and 3) a GPU builder that targets NVIDIA GPUs. MOD2IR also allows adding debug information to the generated IR (5).

Once the IR is generated, it undergoes further transformations via three major MOD2IR passes. The first pass replaces transcendental math functions with appropriate library calls from vector- or GPU-math libraries (6) (see Section 3.3). The second pass annotates the IR module and functions with metadata (7). These annotations help subsequent LLVM passes with alias analysis or decide which functions will be offloaded to the GPU, and are inserted based on the domain information. Finally, the optimization pass is scheduled (8), which runs different existing LLVM optimization passes.

In the end, the optimized IR can be consumed in two ways: direct JIT-compilation for on-the-fly execution (9) or conversion to bytecode/assembly code for AOT-compilation (10) (see Section 3.4). The JIT engine is designed to be easily extensible, and MOD2IR currently provides an implementation suitable for executing LLVM IR on CPUs, or NVIDIA GPU platforms.

3.2 Code Generation

3.2.1 Target-Agnostic AST Lowering. The NMODL AST does not explicitly define computational kernels, nor the underlying data structure that describes nerve cells including physical properties such as compartments, their areas, ion concentrations, voltage, etc. Instead, many AST nodes implicitly define the semantics of updating the state of a nerve cell. Mapping such AST nodes to corresponding IR operations would make low-level IR generation difficult, with a single AST node mapping to hundreds of LLVM IR instructions.

To avoid this, MOD2IR introduces a small set of new AST nodes specific to code generation, allowing to progressively lower high-level domain-specific NMODL AST nodes to newly-introduced IR-friendly abstractions and finally to IR itself. MOD2IR adds new AST nodes to represent: 1) a function, which is used to construct *nrn_state* and *nrn_cur* kernels from `DERIVATIVE` and `BREAKPOINT` blocks; 2) a `Mechanism` object storing physical properties of the nerve cell; and 3) a `for` loop which encapsulates all computations. Figure 3 shows an example of how these new nodes can be used to lower the AST for a simple state update kernel.

In order to make target-specific code generation from target-agnostic AST scalable and extensible, MOD2IR annotates expressions with *traits*, which describe the behavior of an expression with respect to vectorization (shown as `{{trait-type}}` in Figure 3).

- `induction` – Expression must remain scalar, even in presence of vector instructions. Most common use case of this trait is to annotate induction variables.

- `broadcast` – Expression is broadcasted when vectorized. This helps to specify how to vectorize constants.
- `direct` (default) – Expression with a memory access can be vectorized directly. This is helpful to specify how standard loads and stores are handled.
- `indirect` – Expression with a memory access has to be gathered/scattered and may have write conflicts on parallel access.

Availability of traits and a `for` loop AST node greatly simplifies code generation. Figure 4 shows C++ which is a direct translation from LLVM IR generated by MOD2IR for a range of platforms (1) – (3). We observe that it is sufficient to specialize the code generation for the `for` loop and the expression traits, leaving other code generation logic intact.

It is worth pointing out that some models, such as synapses, may issue multiple writes to the same memory location. When generating vectorized or GPU code, these writes must be atomic to maintain program correctness. Many downstream C++ compilers are too conservative and do not vectorize the code in presence of such atomic operations. MOD2IR exploits domain knowledge and trait annotations to identify conflicting writes, and has an additional AST node to represent such atomic operations to facilitate vectorization.

3.2.2 Target-Aware Code Generation.

Generation of IR code is performed by the `IRCodegen` AST visitor (Figure 2), which uses an `IRBuilder` instance to generate target-specific IR. The `IRCodegen` visitor dictates the overall structure of the generated IR, in terms of basic blocks and high-level logic, and the `IRBuilder` provides concrete implementations of IR-generating functionality. This allows IR for different platforms to be generated with minimal effort; e.g., targeting a vector architecture merely requires an `IRBuilder` instance capable of emitting LLVM instructions on vector types.

By default, MOD2IR uses the `BaseBuilder` class to generate scalar code. It allows MOD2IR users to generate code for CPUs as well as extend it to support custom platforms. The builder has a set of *creation* methods that provide services such as accessing mechanism data members and looking up values in the symbol table, and are accessible to the subclasses of `BaseBuilder`. Also, the builder defines a set of code *generation* methods – virtual C++ functions allowing subclasses of `BaseBuilder` to customise the code generation rules, e.g. for atomic statements.

In addition to scalar code generation, we provide the functionality to generate SIMD code and target GPUs (only NVIDIA at the moment) via subclasses of `BaseBuilder`: `SIMDBuilder` and `GPUBuilder` respectively. As SIMD and GPU code generation requires specific considerations, here we summarise key implementation details:

`SIMDBuilder`: facilitates generation of vectorized code, overriding generation methods to ensure that: 1) the stopping condition and increment of a `for` loop are adjusted to account for the vector width; 2) scalar constants with `broadcast` trait

```
STRUCT Mechanism {
  INTEGER* node_index
  DOUBLE* voltage
  DOUBLE* m
  INTEGER size
}
VOID state_update(Mechanism& data){
  {{induction}} INTEGER id
  INTEGER node_id
  DOUBLE v
  FOR (id = START; id < END; id += INC) {
    node_id = data.node_index[id]
    v = {{indirect}} data.voltage[node_id]
    data.m[id] = {{broadcast}} 2 + v * data.m[id]
  }
}
```

Figure 3. The NMODL AST generated by the Lowering visitor for a simple state update kernel. All expression traits are automatically generated. In red – target-agnostic AST nodes specializing the `for` loop, which can be converted into different IR based on the target platform.

```
// This struct is included in all code snippets below
struct Mechanism {
  int* index;
  double* voltage;
  double* m;
  int size;
};
```

(1)

```
void state_update(Mechanism& data){
  int id;
  int node_id;
  double v;
  for (id = 0; id < m.size; id += 1) {
    node_id = data.index[id];
    v = data.voltage[node_id];
    data.m[id] = 2 + v * data.m[id];
  }
}
```

(2)

```
void state_update(Mechanism& data){
  int id;
  __mm256i node_id;
  __mm512d v;
  for (id = 0; id < m.size - 7; id += 8) {
    node_id = _mm256_load_si256(&data.index[id]);
    v = _mm512_i32gather_pd(node_id,
          &data.voltage[node_id], 8);
    __mm512d t1 = __mm512_load_pd(&data.m[id]);
    __mm512d t2 = __mm512_set1_pd(2);
    __mm512d t3 = __mm512_fmadd_pd(t1, v, t2);
    __mm512_store_pd(&data.m[id], t3);
  }
}
```

(3)

```
__global__
void state_update(Mechanism& data){
  int id;
  int node_id;
  double v;
  for (id = blockIdx.x * blockDim.x + threadIdx.x;
       id < m.size;
       id += blockDim.x * gridDim.x) {
    node_id = data.index[id];
    v = data.voltage[node_id];
    data.m[id] = 2 + v * data.m[id];
  }
}
```

Figure 4. Direct translation from the LLVM IR generated by MOD2IR into C++ for different target platforms: 1) CPUs; 2) CPUs with AVX-512 support; and 3) NVIDIA GPUs. Note that C++ is presented here for brevity only.

```python
import numpy as np
import nmodl.dsl as nmodl
from neuron import h
# two models represented as strings
hh     = "NEURON { SUFFIX hh …"
expsyn = "NEURON { SUFFIX expsyn …"
models = [hh, expsyn]
# parse NMODL mechanisms and create ASTs
models_ast = []
driver = nmodl.NmodlDriver()
for mod in models:
 models_ast.append(driver.parse_string(mod))
# create model to simulate
h.load_file('model.ses')
# model setup [code]
sim_model = h.get_model()
# copy model to target device [code]
# code generation and JIT configuration
cfg = nmodl.CodeGenConfig()
cfg.llvm_gpu_target = "nvptx64"
cfg.llvm_gpu_target_arch = "sm_70"
jit = nmodl.Jit(cfg)
# set dt, tstop [code]
# representative simulator loop
for t in np.arange(0, tstop, dt):
  # timestep setup [code]
  for ast in models_ast:
    jit.run(ast, sim_model)
  # solver and communication [code]
# copy results back to host [code]
```

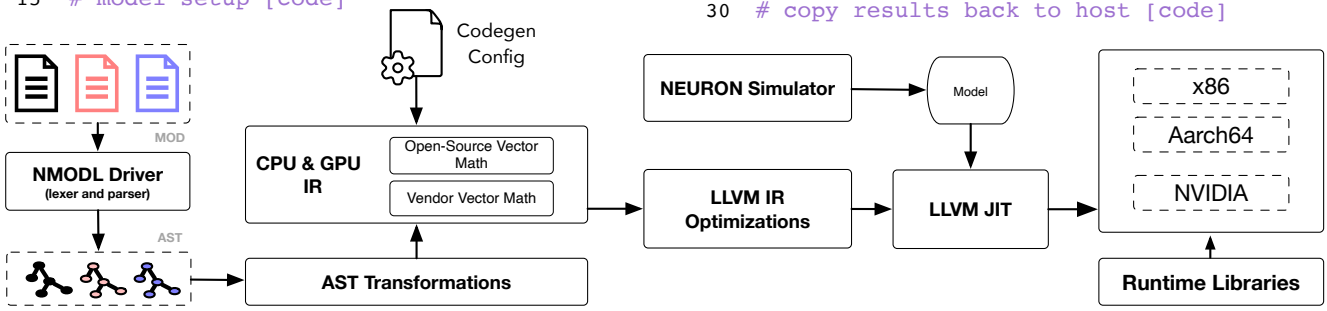

Figure 5. An end-to-end example using MOD2IR's Python API with JIT Driver. At the top, a representative example to demonstrate on-the-fly execution using the Python API of MOD2IR. It shows how mechanisms (`hh`, `expsyn`) written in NMODL DSL are instantiated and executed using JIT engine. Comments with `[code]` markers indicate the implementation details that are not shown for brevity. At the bottom, the underlying workflow in MOD2IR showing full pipeline from the lexer, parser, AST creation, AST transformations and IR generation to JIT execution on different hardware targets.

are indeed broadcasted; 3) gather and scatter instructions are issued in presence of `indirect` traits; 4) reading and writing with `direct` trait involves masked instructions to account for non-uniform execution of conditional statements; and finally 5) code generated for atomic statements issues a sequence of scalar writes to avoid *write-write* conflicts.

`GPUBuilder`: targets NVIDIA platforms, largely reusing the scalar builder functionality. The start and stop conditions of the `for` loop are adjusted based on thread, block and grid ids. Also, code generation for atomic statements is changed to issue GPU atomics (e.g. *atomicAdd*). In addition, when performing the annotations pass, we mark certain LLVM IR functions as *kernels* which are offloaded to GPU, enabling the reuse of LLVM NVPTX back-end[1]. This allows MOD2IR to use NVPTX-specific optimizations, as well as infer where in the GPU memory hierarchy the data must reside.

3.3 Integration of Vector Math Libraries

The NMODL DSL can use many transcendental math functions such as `exp` and `log`. For the open-source compilers such as GCC, the lack of a cross-platform vector-math library has been a limiting factor for the auto-vectorization performance of NMODL. As discussed in [13], the use of SIMD-optimized math functions is important. To interface with vector-math libraries, MOD2IR features a *Math Function Replacement Pass* (MFRP). Once the builder generates LLVM's math intrinsics, the pass replaces them with appropriate function calls from the vector-math library. Currently MFRP supports five CPU libraries and one GPU library covering all hardware and OS platforms that NEURON is most commonly used today: Accelerate [3], Libdevice [53], Libmvec [44], MASSV [27], SVML [28] and SLEEF. Internally, MFRP reuses an existing LLVM pass[2] and only extends it to support more replacement patterns, e.g. for Libdevice. Also, the use of an open-source library like SLEEF allows us to integrate it as part of the binary distribution of NEURON and avoid dependency on external, vendor-specific libraries but also take advantage of them when they are present.

3.4 JIT Execution and Interfacing With Python APIs

MOD2IR offers two compilation workflows: ahead-of-time and just-in-time. With AOT compilation, MOD2IR converts NMODL into LLVM IR, optimizes it, and then converts it into either LLVM bytecode or assembly file. The generated

[1] https://llvm.org/docs/NVPTXUsage.html

[2] https://llvm.org/doxygen/ReplaceWithVeclib_8h_source.html

code can then be used to create a model-specific executable by the regular code generation workflow discussed in Section 2.2. With JIT-compilation, instead of creating assembly or bytecode files, the optimized IR is directly converted to machine code for the specified target hardware and executed just-in-time. To achieve this, we have implemented a JIT-compilation engine based on the ORC JIT[3]. The availability of this JIT compilation in MOD2IR paves the way for a new type of simulation workflow in the NEURON , and enables implementations of easy-to-use programmatic interfaces for MOD2IR. We have extended the NMODL Python API to provide access to the LLVM code-generation visitors and the JIT engine, allowing dynamic execution of mechanisms.

Figure 5 shows a representative example of such a new workflow using MOD2IR's Python API and JIT-based execution. First, using the `nmodl` Python module, the mechanisms (`hh`, `expsyn`) are parsed and their AST objects are created. Using the `neuron` Python module we instantiate a simulation object. Then, we can configure the code generation target and execute all mechanisms using MOD2IR's JIT engine. While this workflow is not yet fully implemented in the production toolchain as some integration between NEURON and NMODL is still missing, the MO2DIR's Python API is an important step towards the final goal of simplifying the code generation and making it more interactive.

4 Benchmarks

4.1 Benchmarking Methodology

We have set up a number of benchmarks to compare the performance of MOD2IR with the latest vendor compilers and math libraries on CPU and GPU platforms. The main objective of MOD2IR is to eliminate the need for maintaining multiple compiler back-ends with different programming models and third-party compilers while maintaining performance competitiveness. Hence, the goal of these benchmarks is to attain performance equal to or better than the code generated from the state-of-the-art compilers used today. Note that the AST transformations from Kumbhar et al. [33, Section 4.1] are implemented in NMODL framework and hence they equally benefit MOD2IR as well as existing back-ends.

For our benchmarks, we use the two standard neuronal mechanisms shipped with NEURON: the Hodgkin-Huxley (`hh`) and the ExpSyn (`expsyn`) models. The Hodgkin-Huxley model [25] is a set of nonlinear differential equations that describes the electrical properties of neurons and is commonly found in electrical models of neurons. The ExpSyn [22] is an exponential decay synapse that is deterministic and commonly used to model synaptic dynamics. As discussed in Section 2.1, when these mechanisms are transpiled, there are two kernels that account for more than 90% of simulation time: *nrn_state* and *nrn_cur*. These two kernels are laid out in such a way that the innermost loop iterates over all discretized elements in the neuronal model and the mechanisms of the same type stored together. This makes performance optimizations via loop vectorization possible.

The three compute kernels from the `hh` and `expsyn` mechanisms benchmarked here are representative of the vast majority of mechanisms found in NEURON simulations in terms of their computational characteristics and vectorization patterns: 1) the state update kernel of the `hh` mechanism (`nrn_state_hh`) has high arithmetic intensity due to use of division operations and transcendental math functions like `exp`. Hence, for vectorization performance the use of vector math libraries is important. 2) the current update kernel of `hh` (`nrn_cur_hh`) is a memory streaming kernel, having a low arithmetic intensity. In these kernels, the use of prefetch instructions or memory bandwidth saturation impacts the performance. 3) the current update kernel of the `expsyn` synapse is similar to `nrn_cur_hh` but it has the peculiarity of involving a reduction step in the main compute loop. This is necessary to accumulate synaptic currents within the same discretization element as there could be more than one synapse located at a specific location of a nerve cell. For these types of kernels, the auto-vectorization performed by general-purpose C++ compilers is unsuccessful as loop iterations are not independent.

In order to run our benchmarks in a realistic setting, we instantiate many instances of a mechanism with random parameters, execute them and measure the runtime. We make sure to allocate > 6 GBs of memory for each execution to avoid caching effects. Each benchmark is executed five times and the average runtime is reported, while the variance in all cases is very small. All benchmarks are run in double precision. Since MOD2IR supports on the fly execution via JIT-compilation, our benchmarks include results for both the AOT- and JIT-compilation workflows in the case of MOD2IR but only the AOT workflow for other compilers.

4.2 Benchmarking Platforms

All CPU benchmarks were performed on an Intel Xeon Cascade Lake CPU (see Table 1). Furthermore, we compare our implementation against four popular state-of-the-art compilers: Intel C++ Compiler Classic (icpc), GCC, NVHPC C++ Compiler (nvc++) and Clang. The use of mathematical libraries is essential to achieving optimal code performance, especially due to the use of transcendental math functions, as their implementation can be highly sensitive to hardware platform details. Because of this, we use Intel's SVML library and the open-source SLEEF library. We also run benchmarks on an NVIDIA GPU, comparing MOD2IR performance with OpenACC code compiled with NVHPC Compiler. Table 1 summarizes all the details of our benchmarking setup including hardware, compiler toolchains, and math libraries.

The compiler flags used for various benchmarking configurations discussed in Section 4.3 are based on production build

[3] https://llvm.org/docs/ORCv2.html

Table 1. Benchmarking System and Compiler Toolchains.

CPU	Intel Xeon Gold 6248, L2 20MiB, L3 27.5MiB, 2.5 GHz (Cascade Lake)
GPU	NVIDIA V100 32GB HBM2
Compilers	icpc 2021.4.0, g++ 11.2.0, clang++ 13.0.0, nvc++ 22.3, nvcc 11.6.1
Math libraries	SVML 2021.4.0, SLEEF 3.5.1, libdevice 11.6.1

configurations and ensure typical target specific optimization are enabled (e.g. *-march=skylake-avx512 -mtune=skylake -mavx512f*). In every case, we additionally enable OpenMP SIMD for explicit vectorization via the *omp simd* pragmas, pass the most beneficial optimization level flag for each compiler (level 2 for Intel due to aggressive math library optimizations that generate numerically different results and level 3 for others), and enable *fastmath* and *tree-vectorize* flags when applicable. The obtained results with the optimization flags are within acceptable accuracy range.

4.3 Benchmarking Results

4.3.1 CPU Results. Figure 6 presents the performance evaluation of the `hh` and `expsyn` benchmarks on the x86 CPU platform. We compare the runtime of AOT compiled kernels. As MOD2IR supports JIT execution, additionally we measure performance with JIT workflow. We use Intel compiler with SVML math library as our baseline (speedup = 1) as it is often the optimal configuration used in production simulations. We show relative speedup (> 1) or degradation (< 1) compared to this baseline.

For the AOT workflow, we first generate C++ code using the NMODL framework, which is then compiled and, if specified, linked against the SVML math library (*_svml* suffix in the legends). The resulting shared library is then loaded at runtime by the MOD2IR benchmarking tool. MOD2IR can use either SVML (*_svml* suffix in the legends) or SLEEF (*_sleef* suffix in the legends) math libraries and then can be executed via JIT. The *mod2ir_svml* and *mod2ir_sleef* follow the AOT compilation workflow as well, using the Clang compiler to compile the LLVM IR code generated by MOD2IR and then creating a shared library. Finally, *mod2ir_jit_svml* and *mod2ir_jit_sleef* follow the JIT compilation workflow.

First, we observe the importance of using a math library, specifically in the state update kernel, which makes heavy use of transcendental math functions and benefits from optimized and vectorized implementations. This can be easily observed in the `nrn_state_hh` kernel that uses transcendental functions in which the performance of GCC as well as Clang is about 6.5× slower. But when the SVML math library is used, we see that Clang is able to generate better code and outperforms Intel compiler by ∼1.12×. The performance of MOD2IR with SVML is on par or faster than the Intel compiler baseline. For the `nrn_state_hh` kernel, MOD2IR performs ∼1.1× faster.

Second, even though using the SLEEF library leads to a ∼1.28× slow-down compared to the baseline, this is an important result as it provides significantly better performance than toolchains like GCC and Clang without having to rely on a vendor specific, proprietary math library (e.g. SVML).

Third, in the case of the `nrn_cur_hh` kernel, Intel compiler, GCC and MOD2IR have similar performance independent of the math library used. This is because current update kernels typically have memory streaming characteristics. Notably, we see that Clang is about 1.43× slower and NVHPC is about 3.8× slower than the baseline. For Clang, we looked into generated LLVM IR and assembly code and found that Clang generates more memory-related instructions. In the case of NVHPC compiler, even though necessary compiler flags and loop annotations are used, it reports that the vectorization is not profitable and does not vectorize the loop [29].

Fourth, even more interestingly, for the `nrn_cur_expsyn` kernel MOD2IR achieves an improvement of ∼1.26× compared to other compilers. This performance advantage in MOD2IR is due to the fact that the MOD2IR framework is able to use the domain knowledge of the needed reduction operation in AST for synapse mechanisms, while the generated C++ code for the general purpose compilers must rely on an additional reduction loop in order to enable vectorization. In many neuronal models, synapse mechanisms are a dominant part of simulation time and hence this is a significant performance improvement using MOD2IR.

Finally, we note that the JIT workflow has on average no runtime overhead over the AOT workflow and hence using the JIT is to be preferred since it brings additional flexibility at no performance cost.

4.3.2 GPU Results. Figure 7 presents the performance evaluation on an NVIDIA GPU platform. We compare the performance of MOD2IR to the NVHPC compiled C++ OpenACC code. We use the same strategy as in Section 4.3.1: for AOT workflow, a shared library is created using NVHPC compiler and for JIT workflow, CUDA JIT back-end of LLVM is used to execute the LLVM IR. The OpenACC implementation is used as a baseline because it is the default programming model in NEURON for GPU execution and has a similar performance to CUDA implementation [32]. As MOD2IR does not add significant overhead with JIT, MOD2IR with JIT workflow was used for simplicity of benchmarking. For all benchmarks, the *libdevice* library that implements math primitives for NVIDIA GPU devices is used. The presented benchmarks only take into account the execution time of kernels on GPU and do not include the data transfer. This is because NEURON transfers the in-memory model to GPU during the initialization step and subsequent timestep evaluations do not need to move data between CPU and GPU

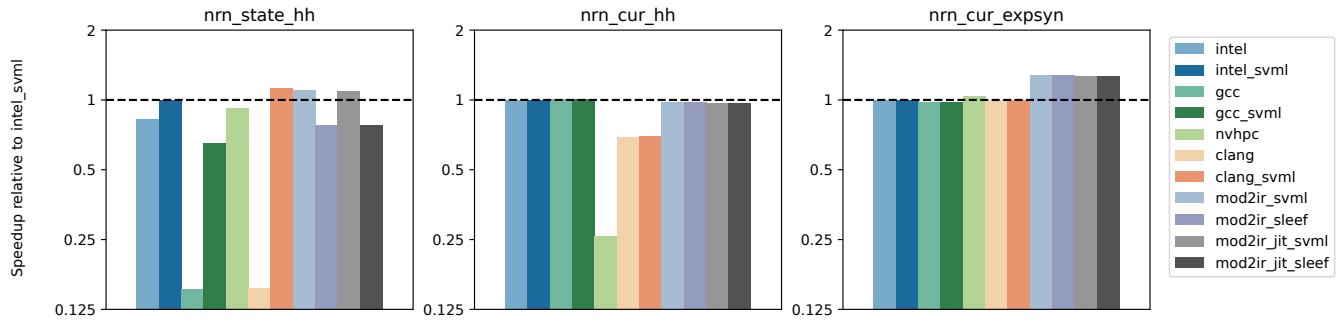

Figure 6. Performance comparison of MOD2IR on Intel Cascade Lake CPU (Intel compiler with SVML is used as baseline i.e. speedup = 1), plotted using a logarithmic scale. The color legend is ordered left to right; top to bottom. MOD2IR performs on par or faster than the baseline for state and current update kernel of hh, while it outperforms by 1.26× all other compilers for the current update kernel of expsyn.

memories. Also, this data transfer is handled outside of the code generated from NMODL DSL.

The GPU performance results show a similar trend to CPU benchmarks discussed in the previous section. First, the nrn_state_hh kernel shows the same performance with both MOD2IR and NVHPC compiler. Second, the nrn_cur_hh shows ~1.07× better performance when using MOD2IR compared to NVHPC compiler. The detailed profiling analysis shows that the MOD2IR code has larger warp occupancy due to fewer registers used by the generated code. Third, the nrn_cur_expsyn of for expsyn synapse mechanism achieves ~1.2× better performance with MOD2IR. The profiling analysis shows that this is due to the higher arithmetic intensity, fewer loads from global memory, and more cache hits for the MOD2IR generated code than NVHPC compiler. In summary, MOD2IR performs on par or faster than the baseline performance of NVHPC compiler. Additionally, using JIT execution workflow MOD2IR provides better flexibility than the AOT compilation workflow with NVHPC compiler.

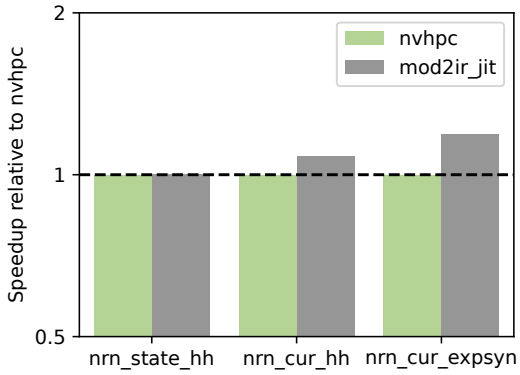

Figure 7. Performance comparison of MOD2IR on NVIDIA V100 GPU (NVHPC compiler is used as baseline i.e. speedup = 1), plotted using a logarithmic scale. The color legend is ordered left to right; top to bottom. MOD2IR performs on par with the baseline for state update kernel and ~1.07× better for the current update kernel of hh mechanism. It achieves ~1.2× better performance for current update kernel of expsyn.

4.4 Discussion

With the code generation support of CPU as well as GPU platforms, we believe that MOD2IR serves as a good example for future domain-specific compilers in the field of neuroscience. The progressive lowering approach of the NMODL AST helps with the code maintainability and allows the support of new platforms in the future more easily.

We also believe that targeting intermediate representations like LLVM IR opens many new doors for code generation. In particular, MOD2IR can target almost all back-ends supported by LLVM, including GPUs, as well as enjoy many LLVM-specific tools like llvm-mca [2] or explore conversions to other representations like SPIR-V [30] or MLIR [35]. One other important aspect of MOD2IR is explicit vectorization. In the previous attempts with an embedded DSL library like Cyme [18], one needed to implement support for different SIMD instruction set extensions for each ISA. Using IR vector instructions helps us to avoid vendor-specific vector intrinsics, once again improving the maintainability of the code while achieving on-par performance.

Another important aspect of code vectorization is the availability of vector math libraries. MOD2IR supports a range of high-performance SIMD libraries with fast implementation of transcendental math functions, including both vendor and open-source libraries. Also, integration of libraries like SLEEF will help us to create binary distributions without dependency on vendor libraries like SVML.

For MOD2IR, performance is also an important consideration. In the benchmarks section, we have shown that MOD2IR achieves on par performance with the state-of-the-art compilers like Intel, GCC, NVHPC or Clang on most

of the computational benchmark kernels, and sometimes even outperforms them by up to 1.26×. On NVIDIA GPUs MOD2IR exhibits a similar performance trend. We believe that in the future, directly generating intermediate representation will allow MOD2IR to exploit the knowledge at the DSL level to generate more optimized code, leading to greater speedups.

5 Related Work

In computational science, DSLs have found many uses to generate optimized solver codes. Liszt [16] is an early example of a DSL enabling performance portability of mesh-based partial differential equation (PDE) solvers. FEniCS [37] and Firedrake [47] are examples of frameworks which automatically generate PDE solvers using the finite-element method. The Open Earth Compiler [20] is a recent example of multi-level IR generation in weather and climate modeling. Other examples of stencil computation DSLs include Pochoir [52], SDSLc [48] and Devito [38, 39]. Dawn [42] is a LLVM-based code generation library for geophysical fluid dynamics models. Halide [46] and Polymage [41] are examples of DSLs allowing the user to tune image processing pipelines. Implementing DSLs often involves writing the same components such as a lexer, a parser, and AST visitors. To simplify the design and implementation of DSLs, frameworks such as MontiCore [31] and AnyDSL [36] have been proposed.

Numerous examples of DSLs and code-generation frameworks can be found in computational neuroscience. The NMODL framework, on which this work is based, is a transpiler for the homonymous DSL being developed within the NEURON simulator. It supports different CPU and GPU back-ends but uses a source-to-source translation technique and relies on different programming models like C++, ISPC, CUDA, and OpenACC. Arbor [1] is another simulator, which supports the NMODL language with its own transpiler – `modcc` [4]. Similar to the NMODL framework, it has different back-ends like C++, CUDA, and HIP to target different hardware architectures. For SIMD execution on CPUs, instead of relying on auto-vectorization capabilities of compilers, it implements various SIMD classes to support back-ends like AVX, AVX-512, NEON, and SVE. Other computational neuroscience DSLs like NESTML, NeuroML2 and NineML use similar source-to-source translation techniques to convert higher-level DSL specifications to Python, C++, or CUDA code. The code generation techniques of these neuroscience DSLs are discussed in detail in Blundell et al. [7]. All these DSL frameworks require constant maintenance and tuning with the evolving programming models and hardware architectures. To our knowledge, MOD2IR is the first example of an LLVM-based code generation framework for detailed brain modeling in the computational neuroscience domain.

[4] https://github.com/arbor-sim/arbor/tree/v0.7/modcc

6 Conclusion

In this work, we presented MOD2IR, an LLVM-based code generation framework with optional JIT-compilation support for detailed neurosimulations. Our approach builds on top of the NEURON simulator and the NMODL DSL, a declarative language allowing domain scientists to model the biophysical processes in neurons. We show how our framework allows to generate LLVM IR or compile to machine code rather than first generating a high-level language, such as C++, as has been done in previous implementations. This allows us to simplify the code-generation process while not sacrificing the performance – our approach generates code that is up to 1.26× faster than state-of-the-art compilers in terms of execution speed for performance-critical kernels. Thanks to the JIT infrastructure we offer a high-level Python interface with the potential to simplify the simulation workflow.

The progressive NMODL AST lowering method and JIT-based execution capabilities of MOD2IR, open up many optimization opportunities that were not possible with an AOT compilation. For example, runtime introspection capabilities for mechanism types and their locality will help to better expose parallelism, reduce the need for reductions or atomic instructions and generate optimized code.

In its current state, MOD2IR will require additional development to fully integrate with the NEURON simulator. Furthermore, we would like to support additional hardware platforms such as GPUs from other vendors and platforms with scalable vector width ISA, such as ARM Scalable Vector Extension [51] (SVE). While different in nature from traditional vector instruction sets, SVE can be easily supported by extending MOD2IR's builders and passes.

Based on the results achieved by MOD2IR, we believe that compiler frameworks such as LLVM or MLIR can become the new standard in developing simulators for computational neuroscience. This will enable making the compilation workflow more interactive and friendly for domain scientists, while running simulations of brain models faster.

7 Data Availability Statement

The results presented in this paper (Figures 6 and 7) can be reproduced using provided artifact package [40].

Acknowledgments

This work was supported by funding to the Blue Brain Project (a research center of EPFL) from the Swiss government, National Institutes of Health (NIH) under the Grant No. R01NS11613, the European Union's Horizon 2020 Framework Programme Grant Agreement No. 785907 (HBP SGA2) and IRIS EPSRC Programme Grant (EP/R006865/1). We would like to thank James Gonzalo King, Nicolas Cornu, Giacomo Castiglioni and NMODL developers for fruitful discussions and contributions. We would also like to thank George Bisbas for providing feedback on an earlier draft of this work.

References

[1] Nora Abi Akar, Ben Cumming, Vasileios Karakasis, Anne Kusters, Wouter Klijn, Alexander Peyser, and Stuart Yates. 2019. 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)*. IEEE, Pavia, Italy, 274–282. https://doi.org/10.1109/EMPDP.2019.8671560

[2] Andrea Di Biagio. 2018. llvm-mca - LLVM Machine Code Analyzer. Available at https://llvm.org/docs/CommandGuide/llvm-mca.html.

[3] Apple. 2018. Accelerate. Available at https://developer.apple.com/accelerate/.

[4] Omar Awile, Pramod Kumbhar, Nicolas Cornu, Salvador Dura-Bernal, James Gonzalo King, Olli Lupton, Ioannis Magkanaris, Robert A. McDougal, Adam J. H. Newton, Fernando Pereira, Alexandru Săvulescu, Nicholas T. Carnevale, William W. Lytton, Michael L. Hines, and Felix Schürmann. 2022. Modernizing the NEURON Simulator for Sustainability, Portability, and Performance. , 2022.03.03.482816 pages. https://doi.org/10.1101/2022.03.03.482816

[5] David A. Beckingsale, Thomas RW Scogland, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, and Brian S. Ryujin. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, Denver, CO, USA, 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (Jan. 2017), 65–98. https://doi.org/10.1137/141000671

[7] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. 2018. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics* 12 (Nov. 2018), 68. https://doi.org/10.3389/fninf.2018.00068

[8] Mark Bohr. 2007. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Newsletter* 12, 1 (2007), 11–13. https://doi.org/10.1109/N-SSC.2007.4785534

[9] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: Composable Transformations of Python+NumPy Programs.

[10] Robert C. Cannon, Padraig Gleeson, Sharon Crook, Gautham Ganapathy, Boris Marin, Eugenio Piasini, and R. Angus Silver. 2014. LEMS: A Language for Expressing Complex Biological Models in Concise and Hierarchical Form and Its Use in Underpinning NeuroML 2. *Frontiers in Neuroinformatics* 8 (Sept. 2014). https://doi.org/10.3389/fninf.2014.00079

[11] Weiliang Chen, Tristan Carel, Omar Awile, Nicola Cantarutti, Giacomo Castiglioni, Alessandro Cattabiani, Baudouin Del Marmol, Iain Hepburn, James G. King, Christos Kotsalos, Pramod Kumbhar, Jules Lallouette, Samuel Melchior, Felix Schürmann, and Erik De Schutter. 2022. STEPS 4.0: Fast and Memory-Efficient Molecular Simulations of Neurons at the Nanoscale. , 2022.03.28.485880 pages. https://doi.org/10.1101/2022.03.28.485880

[12] Giuseppe Chindemi, Marwan Abdellah, Oren Amsalem, Ruth Benavides-Piccione, Vincent Delattre, Michael Doron, András Ecker, Aurélien T. Jaquier, James King, Pramod Kumbhar, Caitlin Monney, Rodrigo Perin, Christian Rössert, Anil M. Tuncel, Werner Van Geit, Javier DeFelipe, Michael Graupner, Idan Segev, Henry Markram, and Eilif B. Muller. 2022. A Calcium-Based Plasticity Model for Predicting Long-Term Potentiation and Depression in the Neocortex. *Nature Communications* 13, 1 (Dec. 2022), 3038. https://doi.org/10.1038/s41467-022-30214-w

[13] Francesco Cremonesi, Georg Hager, Gerhard Wellein, and Felix Schürmann. 2020. Analytic Performance Modeling and Analysis of Detailed Neuron Simulations. *The International Journal of High Performance Computing Applications* 34, 4 (July 2020), 428–449. https://doi.org/10.1177/1094342020912528

[14] Francesco Cremonesi and Felix Schürmann. 2020. Understanding Computational Costs of Cellular-Level Brain Tissue Simulations Through Analytical Performance Models. *Neuroinformatics* (Feb. 2020). https://doi.org/10.1007/s12021-019-09451-w

[15] Sharon Crook, Padraig Gleeson, Fred Howell, Joseph Svitak, and R. Angus Silver. 2007. MorphML: Level 1 of the NeuroML Standards for Neuronal Morphology Data and Model Specification. *Neuroinformatics* 5, 2 (April 2007), 96–104. https://doi.org/10.1007/s12021-007-0003-6

[16] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2063384.2063396

[17] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (Xsw 2013)*. IEEE, Boulder, CO, USA, 18–24. https://doi.org/10.1109/XSW.2013.7

[18] Timothée Ewart, Fabien Delalondre, and Felix Schürmann. 2014. Cyme: A Library Maximizing SIMD Computation on User-Defined Containers. In *Supercomputing*, Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer (Eds.). Vol. 8488. Springer International Publishing, Cham, 440–449. https://doi.org/10.1007/978-3-319-07518-1_29

[19] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. 2020. Parallel Programming Models for Heterogeneous Many-Cores: A Comprehensive Survey. *CCF Transactions on High Performance Computing* 2, 4 (Dec. 2020), 382–400. https://doi.org/10.1007/s42514-020-00039-4

[20] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2021. Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation. *ACM Transactions on Architecture and Code Optimization* 18, 4 (Dec. 2021), 1–23. https://doi.org/10.1145/3469030

[21] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

[22] Michael L. Hines. 2007. NEURON: Point Processes and Artificial Cells.

[23] M. L. Hines and N. T. Carnevale. 1997. The NEURON Simulation Environment. *Neural Computation* 9, 6 (Aug. 1997), 1179–1209. https://doi.org/10.1162/neco.1997.9.6.1179

[24] M. L. Hines and N. T. Carnevale. 2000. Expanding NEURON's Repertoire of Mechanisms with NMODL. *Neural Computation* 12, 5 (May 2000), 995–1007. https://doi.org/10.1162/089976600300015475

[25] A. L. Hodgkin and A. F. Huxley. 1952. A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve. *The Journal of Physiology* 117, 4 (Aug. 1952), 500–544. https://doi.org/10.1113/jphysiol.1952.sp004764

[26] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, and the rest of the SBML Forum:, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le Novere, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness,

Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. 2003. The Systems Biology Markup Language (SBML): A Medium for Representation and Exchange of Biochemical Network Models. *Bioinformatics* 19, 4 (March 2003), 524–531. https://doi.org/10.1093/bioinformatics/btg015

[27] IBM. 2018. Mathematical Acceleration Subsystem (MASS) Libraries for Linux. Available at https://www.ibm.com/support/pages/mathematical-acceleration-subsystem-mass-libraries-linux-big-endian-latest-version.

[28] Intel. 2019. Intel Short Vector Math Library (SVML). Available at https://software.intel.com/en-us/node/523613.

[29] Ioannis Magkanaris. 2022. Forcing Loop Vectorization with NVHPC Compiler. Available at https://forums.developer.nvidia.com/t/force-a-loop-to-vectorize/134231/3.

[30] John Kessenich and Boaz Ouriel. 2018. SPIR-V Specification (The Khronos Group). Available at https://registry.khronos.org/SPIR-V/specs/1.0/SPIRV.pdf.

[31] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer* 12, 5 (Sept. 2010), 353–372. https://doi.org/10.1007/s10009-010-0142-1

[32] Pramod Kumbhar. 2016. CoreNeuron: Morphologically Detailed Neuron Simulations. Available at https://on-demand.gputechconf.com/gtc/2016/presentation/s6213-pramod-kumbhar-coreneuron.pdf.

[33] Pramod Kumbhar, Omar Awile, Liam Keegan, Jorge Blanco Alonso, James King, Michael Hines, and Felix Schürmann. 2020. An Optimizing Multi-platform Source-to-source Compiler Framework for the NEURON MODeling Language. In *Computational Science – ICCS 2020 (Lecture Notes in Computer Science)*, Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira (Eds.). Springer International Publishing, Cham, 45–58. https://doi.org/10.1007/978-3-030-50371-0_4

[34] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[35] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[36] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 1–30. https://doi.org/10.1145/3276489

[37] Anders Logg, Kent-Andre Mardal, and Garth Wells (Eds.). 2012. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Number 84 in Lecture Notes in Computational Science and Engineering. Springer, Heidelberg.

[38] Mathias Louboutin, Michael Lange, Fabio Luporini, Navjot Kukreja, Philipp A. Witte, Felix J. Herrmann, Paulius Velesko, and Gerard J. Gorman. 2019. Devito (v3.1.0): An Embedded Domain-Specific Language for Finite Differences and Geophysical Exploration. *Geoscientific Model Development* 12, 3 (March 2019), 1165–1187. https://doi.org/10.5194/gmd-12-1165-2019

[39] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. 2020. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Software* 46, 1 (March 2020), 1–28. https://doi.org/10.1145/3374916

[40] George Mitenkov, Ioannis Magkanaris, Omar Awile, Pramod Kumbhar, Felix Schürmann, and Alastair Donaldson. 2023. MOD2IR: High-Performance Code Generation for a Biophysically Detailed Neuronal Simulation DSL : Artifact. Zenodo. https://doi.org/10.5281/ZENODO.7521260

[41] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Istanbul Turkey, 429–443. https://doi.org/10.1145/2694344.2694364

[42] Carlos Osuna. 2020. Dawn: A High Level Domain-Specific Language Compiler Toolchain for Weather and Climate Ap Plications. *Supercomputing Frontiers and Innovations* 7, 2 (June 2020). https://doi.org/10.14529/jsfi200205

[43] Dimitri Plotnikov, Bernhard Rumpe, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, and Abigail Morrison. 2016. Nestml: A Modeling Language For Spiking Neurons. (March 2016). https://doi.org/10.5281/ZENODO.1412345

[44] GNU Project. 2015. Libmvec in Glibc. Available at https://sourceware.org/glibc/wiki/libmvec.

[45] Jari Pronold, Jakob Jordan, Brian J. N. Wylie, Itaru Kitayama, Markus Diesmann, and Susanne Kunkel. 2022. Routing Brain Traffic Through the Von Neumann Bottleneck: Parallel Sorting and Refactoring. *Frontiers in Neuroinformatics* 15 (2022).

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notices* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[47] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2017. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Software* 43, 3 (Jan. 2017), 1–27. https://doi.org/10.1145/2998441

[48] Prashant Rawat, Martin Kong, Tom Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. SDSLc: A Multi-Target Domain-Specific Compiler for Stencil Computations. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. ACM, Austin Texas, 1–10. https://doi.org/10.1145/2830018.2830025

[49] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022. Compute Trends Across Three Eras of Machine Learning. (2022). https://doi.org/10.48550/ARXIV.2202.05924

[50] Naoki Shibata and Francesco Petrogalli. 2020. SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (June 2020), 1316–1327. https://doi.org/10.1109/TPDS.2019.2960333

[51] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (March 2017), 26–39. https://doi.org/10.1109/MM.2017.35

[52] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures - SPAA '11*. ACM Press, San Jose, California, USA, 117. https://doi.org/10.1145/1989493.1989508

[53] CUDA Toolkit. 2007. Libdevice User's Guide. Available at https://docs.nvidia.com/cuda/libdevice-users-guide/index.html.

[54] Henry C. Tuckwell. 2005. *Introduction to Theoretical Neurobiology. 2: Nonlinear and Stochastic Theories.* Number 8,2 in Cambridge Studies in Mathematical Biology. Cambridge Univ. Pr, Cambridge.

[55] Tadashi Yamazaki, Jun Igarashi, and Hiroshi Yamaura. 2021. Human-Scale Brain Simulation via Supercomputer: A Case Study on the Cerebellum. *Neuroscience* 462 (May 2021), 235–246. https://doi.org/10.1016/j.neuroscience.2021.01.014

Received 2022-11-10; accepted 2022-12-19