Ace: An Actively-Typed Compilation Environment for High-Performance Computing

Cyrus Omar, Nathan Fulton, Sharschchandra Bidargaddi and Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
http://www.acelang.org/

***Abstract*—We introduce the Ace compilation environment and demonstrate its practicality as a foundational tool for both high-performance computing research and in practice. Ace consists of a statically-typed programming language with user-extensible semantics controlled by a compile-time metalanguage, Python. Users specify new primitive types and their operations by equipping type definitions, which are first-class objects in the metalanguage, with methods that the compiler selectively invokes when type checking and translating expressions, a mechanism we call *active type-checking and translation (AT&T)*. This mechanism allowed us to express the primitives of the C99 and OpenCL languages as simple modules within Ace. We then give examples of several first-class parallel abstractions that can be defined and used alongside these basic primitives, demonstrating the foundational role that an actively-typed system like Ace can play for researchers and domain experts developing first-class language abstractions, as well as practitioners wishing to use multiple such abstractions within a single program. Active library support in Ace also supports more general forms of type-aware metaprogramming, and Ace functions can be compiled and launched directly from Python, taking its standard numeric data structures as arguments. Using active library support in Ace, we designed a scientific simulation framework that allows users to modularly specify, compile and orchestrate the execution of parameterized families of scientific simulations on clusters of GPUs. This framework has been used to successfully conduct large-scale, high-performance neuroscience simulations, providing initial evidence that Ace can be used, at scale, in practice today.**

I. Introduction

Computer-aided simulation and data analysis techniques have transformed science and engineering. Surveys show that scientists and engineers now spend up to 40% of their time writing software [1, 2]. Most of this software targets desktop hardware and about 20% of scientists target either local clusters or supercomputers for more numerically-intensive computations as well [2]. To fully harness the power of this modern hardware, however, these *professional end-user developers* [3] must write parallel programs.

Professional end-users today generally use high-level scripting languages like MATLAB, Python, R or Perl for tasks that are not performance-intensive, such as small-scale data analysis and plotting [4]. For tasks where these interpretted languages are too slow, they generally call into code written in a statically-typed, low-level language like C and Fortran and use low-level parallel abstractions like pthreads and MPI [5, 6]. Unfortunately, these low-level languages and abstractions are notoriously error-prone and difficult to use, even for expert programmers. CITATION?

The research community has proposed dozens of novel language features and parallel programming abstractions that aim to more equitably balance important concerns relating to performance, portability, verifiability and ease-of-use. Unfortunately, professional end-users have not commonly adopted these approaches, and many have become generally skeptical of new approaches originating in the research community. This viewpoint was perhaps most succinctly expressed by a participant interviewed in a recent study by Basili et al. [6], who stated "I hate MPI, I hate C++. [But] if I had to choose again, I would probably choose the same." Although this sentiment is easy to dismiss as paradoxical, we believe that it demands direct examination by researchers working to advance the practice of scientific and high-performance computing.

One reason for this gap is because new features and abstractions tend to be distributed within new programming languages. Prominent examples include:

- UPC, CAF, X10
- Erlang
- Cilk
- Charm++

This is because they need first-class language and compiler support but monolithic languages cannot be modularly and portably extended (we discuss the difference between language extensions and compiler extensions in Section XX). Unfortunately, writing code in multiple languages poses a host of challenges. Thus, we need a mechanism that allows new primitives to be defined and implemented modularly and distributed with libraries.

Ace is a language-integrated compilation environment that aims to solve this problem. A *compilation environment* is a framework that allows users to programmatically control the compilation of programs. In Ace, each module can interleave two kinds of program logic: compile-time and run-time. The compile-time logic, written in the *metalanguage*, is the top-level logic in each file. Here, users can define and import new primitive types and operators, generate functions programmatically, perform arbitrary code transformations and emit functions for later execution. Ace also permits a hybrid mode of use whereby the functions can also be executed immediately after being defined.

Paragraph describing how active typechecking and translation work. Generalization of active libraries.

The overall design of the Ace language as a gestalt thing that satisfies a number of design and adoption criteria imposed by professional end-users. We demonstrate its practicality by describing an Ace-based simulator library in Section ??. Ace also supports researchers by allowing them to modularly specify new abstractions, which we demonstrate with examples like OpenCL and several others in Section ??.

In addition to the general introduction of Ace as a foundational tool for research and practice and generalizable lessons relating to language design that we note below, specific novel contributions in this paper:

- ATT, a generalization of active libraries, where types are metalanguage objects and programs are given semantics by type-associated methods called by the compiler.
- A method for eliminating type annotations by using a form of type inference within functions and type propagation between functions.
- A method to check the correctness of generated code, by checking for satisfaction of representational consistency constraints associated with types, defined in Section ??.
- Several type-aware code generation and simulation orchestration techniques used in the clegans library, described in Section ??.

II. ACE: AN ACTIVELY-TYPED COMPILATION ENVIRONMENT

Ace is based on the Python programming language and implemented entirely as a Python package. The metalanguage is Python itself, the object language uses Python's grammar directly, and the compiler is implemented in Python as well. Write some more here. Why Python? Popularity and to aid in adoption: can be installed just like any other Python module, uses Python's module system and distribution infrastructure, piggybacks on existing Python toolchain.

We now describe a programming language designed with many of the criteria described in the previous sections in mind, cl.oquence. cl.oquence is built around a minimal core and nearly every construct in the language is implemented as a library. Rather than beginning directly with this extension mechanism, however, let us begin with a concrete example of the language in use.

```
from ace.OpenCL import OpenCL, get_global_id

@OpenCL.fn
def map(in, out, fn):
    gid = get_global_id(0)
    out[gid] = fn(in[gid])
```

Figure 1. A generic data-parallel map function written using the user-defined Ace OpenCL module.

```
from figure1 import map
from ace.OpenCL import global_ptr, double, sin

T = global_ptr(double)
map_sin_dbl = map.compile(T, T, sin.cl_type)
```

Figure 2. The generic map function compiled to map the sin function over global arrays of double-precision floating-point numbers. The `get_global_id` and `sin` functions are OpenCL primitives.

A. Example: Higher-Order Map for OpenCL

Figure 1 shows the standard data-parallel map function written using an extension that implements the full OpenCL language as a library. Each thread, indexed by `gid`, applies the transfer function, `fn`, to the corresponding element of the input array, `in`, writing the result into the corresponding location in the output array, `out`.

B. Syntax

Readers familiar with the Python programming language will recognize the style of syntax used in Figure 1. In fact, cl.oquence uses the Python grammar and parsing facilities directly. Several factors motivated this design decision. First, Python's syntax is widely credited as being particularly simple and readable, due to its use of significant whitespace and conventional mathematical notation. Python is one of the most widely-used languages in scientific computing, so its syntax is already familiar to much of the field. And significantly, a large ecosystem of tools already exist that work with Python files, such as code editors, syntax highlighters, style checkers and documentation generators. These can be used without modification to work with cl.oquence files. Therefore, by re-using an existing, widely-used grammar, we are able to satisfy many of the design criteria described in Section **??** and the adoption criteria described in Section **??** without significant development effort.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void map_sin_dbl(
  __global double* in,
  __global double* out)
{
    size_t gid;
    gid = get_global_id(0);
    out[gid] = sin(in[gid]);
}
```

Figure 3. The OpenCL kernel function generated by running `acecc figure2.py`, or by executing the `print map_sin_dbl.code` statement.

C. Semantics

cl.oquence is a compiled, statically-typed language. The extension used in Figure 1 implements the full OpenCL type system, which includes features inherited from C99 like pointer arithmetic and OpenCL-specific constructs like vector types. In fact, compilation of Figure 1 will produce code semantically equivalent, and with the same performance profile, as manually-written OpenCL code. Compilation produces OpenCL source code, so cl.oquence, when used with this extension, inherits OpenCL's portability profile as well, as described in Section **??**.

In OpenCL itself, however, there is no way to write a function like `map`. As described in Section **??**, OpenCL lacks support for higher-order functions or polymorphism (over the types of `in` and `out` in this case.) To map a function over an array, developers must create a separate function for every combination of argument types and for every transfer function. Figure 2 shows an OpenCL kernel that implements a map specifically for the `sin` function applied to arrays of doubles in global memory. Note that because OpenCL does not support templates or function pointers, this is the most general solution possible without explicit code generation. However, even in CUDA, which does support templates, there is greater syntactic overhead for specializing a function with particular types than in cl.oquence, where it happens implicitly.

To support this much more concise form for specifying kernels, cl.oquence uses techniques pioneered by functional programming languages. We briefly describe these below.

1) Type, Kernel and Extension Inference: As described in Section **??**, professional end-users tend to prefer languages that do not require explicit type annotations for variables. These languages accomplish this by using dynamic typing, so that types are associated with values instead. Statically-typed functional programming languages have also eliminated type annotations in many cases, however, using a technique known as *type inference* [8].

Type inference is a technique for giving types to variables by examining how these variables are being used. As a simple example, in Figure 1, the variable `gid` is being used to hold the result of calling the `get_global_id` function so the type of `gid` must be consistent with the return type of that function, `size_t` (a device-dependent integer type.) Note, however, that in C99, many possible integer types can be given to `gid` (in practice, a 32-bit integer is often used.) Picking a specific type may require further examining how `gid` is used in the remainder of the function. To deal with this non-locality, type inference is typically cast as a constraint solving problem – the type of each program variable is treated as an unknown and each use of the variable introduces a new constraint. A constraint-solving algorithm, generally a variant of unification, can be used to find a type assignment that satisfying all constraints.

cl.oquence uses a variant of this approach (modified to admit extensions to the type system, as we will discuss shortly) to conduct whole-function type inference. This approach differs from the HM algorithm, however, which conducts whole-program (or whole-file) inference and assigns a single type to function arguments. In cl.oquence, types are *propagated* to functions from their calling sites (to enable structural polymorphism, described below) and inference occurs only within a function body.

In addition to type annotations, OpenCL introduces additional syntactic burdens. Developers must annotate functions that meet the requirements to be called from the host with the `__kernel` attribute, and several types (notably, `double`) and specialized functions require that an OpenCL extension be enabled with a `#pragma` (as can be seen in Figure 3.) The OpenCL extension we have developed automatically infers many of these annotations as well, consistent with the principle of conciseness described in Section **??**.

2) Structural Polymorphism: In Section **??**, we discussed several strategies for achieving *polymorphism* – the ability to create functions and data structures that operate over more than a single type. In cl.oquence, all functions are implicitly polymorphic and can be called with arguments of *any type that supports the operations used by the function*. For example, in Figure 1, `in` can be any type that supports indexing by a variable of type `size_t` to produce a value of a type that can be passed into `fn`, which must then produce a value consistent with indexing into `out`. OpenCL pointer types are consistent with these constraints, for example. Although powerful, this also demonstrates a caveat of this approach – that it is more difficult to give a function a concise signature, because arguments are constrained by capability, rather than to a single type [9].

Structural typing can be compared to the approach taken by dynamically-typed languages that rely on "duck typing". It is more flexible than the parametric polymorphism found in many functional languages and in languages like Java (which only allow polymorphic functions that are valid for *all* possible types), but is of comparable strength to the template system found in C++. It can be helpful to think of each function as being preceded by an implicit template header that assigns each argument its own unique type parameter. At function call sites, these parameters are implicitly specialized with the types of the provided arguments. This choice is again motivated by the criteria of conciseness given in Section **??**.

3) Higher-Order Functions: Section **??** also discusses the value of higher-order functions. The OpenCL extension supports higher-order functions, despite the absence of function pointers from OpenCL itself, by explicitly passing functions as constants at compile-time. In other words, each function *uniquely inhabits* a type corresponding to it.

Functions are not, however, *first-class* – references to functions cannot be stored in memory, for example. Most use

```
from figure1 import map
from clq.backends.opencl import double, sin

map_sin_double = map.specialize(
    double.global_ptr, double.global_ptr,
    sin.clq_type)
```

Figure 4. Programmatically specializing the `map` function from Figure 1 for use by the standalone compiler to produce code equivalent to Figure 3.

cases that we have considered thus far have never needed to do so, however. Compared to the function pointer approach used in other C-based languages, this approach is more efficient. It can, however, result in "code bloat", since a new copy of the higher-order function is created for each unique function argument passed to it. First-class functions can theoretically be simulated using an auxiliary lookup function in OpenCL, and we leave clean support for this in cl.oquence as open future work.

D. Active Libraries in cl.oquence

In addition to adopting Python's grammar, cl.oquence uses the Python language itself as a *compile-time metalanguage*. Every cl.oquence file is, at the top-level, a Python script. This script is executed during compilation and has access to a number of powerful capabilities that allow developers to influence the compilation process and manipulate programs directly. Libraries that have such capabilities has been called *active libraries* in prior proposals [7]. A number of projects, such as Blitz++, have taken advantage of the C++ preprocessor and template-based metaprogramming system to implement domain-specific optimizations. In cl.oquence, we replace these brittle mini-languages with a general-purpose language. This allows for several interesting uses that we discuss in the following sections.

E. Module System

In Figure 1, the only compile-time operations are library imports and a declaration of a cl.oquence function. However, this already demonstrates an important benefit of this approach: cl.oquence uses Python's simple but flexible module system. As a consequence, libraries written using cl.oquence can directly utilize the module distribution infrastructure available for Python, partially relieving the issues discussed in Section **??**.

F. Compilation and Invocation

cl.oquence functions can be invoked in one of two ways: from any host language supporting OpenCL by using the standalone compiler, or directly from Python itself.

1) Programmatic Specialization and Compilation: The standalone cl.oquence compiler, `clqcc`, generates source code from cl.oquence source files. To do so, however, users must *specialize* any *externally callable* functions with their specific types. Figure 4 shows how the specific OpenCL function given in Figure 3 can be recovered from the generic specification of `map` given in Figure 1. The `specialize` method of a generic function like `map` creates a new function that is concrete – that is, with concrete argument types.

The standalone compiler operates by first executing the module, then iterating over the externally available variables in the module's environment to extract the OpenCL functions and any other functions and constructs that they depend on. This allows developers who wish to use the host language of their choice, rather than Python, to continue writing library functions generically, in one place, only writing small stub modules to specialize them as needed.

2) Direct Invocation from Python: As discussed in the introduction, a common usage scenario in scientific computing involves the use of a dynamic, high-level language for workflow orchestration and exploratory visualization and analysis, paired with a low-level language for performance-critical sections. Although the extension mechanism we describe shortly will permit the development of a broad array high-level language features in the future, current extensions, such as the OpenCL extension we have been describing, have been designed to make low-level programming simpler. For this reason, the OpenCL extension includes support for directly invoking cl.oquence functions from Python scripts with minimal overhead.

This feature builds on top of an established library, `pyopencl`, that wraps the OpenCL host API and integrates it with the popular `numpy` numerics library in Python. Buffers (arrays) in `pyopencl` do not retain type information, so the cl.oquence extension adds type information, mirroring the API of `numpy` arrays. It also hides many of the most verbose components of the OpenCL interface by providing reasonable defaults, such as an implicit global context and a default queue associated with each context.

Using these simplifications, generic cl.oquence OpenCL functions like `map` can be called directly, without specialization as in the previous section. Figure 5 shows a full OpenCL program written using these bindings. By way of comparison, the same program written using OpenCL directly is two orders of magnitude larger and correspondingly more complex. Not shown are several additional conveniences, such as delegated kernel sizing and In and Out constructs that can reduce the size and improve the clarity of this code further – due to space constraints, the reader is referred to the language documentation for additional details.

G. Code Generation in cl.oquence

Metaprogramming refers to the practice of writing programs that manipulate other programs. There are a number of use cases for this technique, including domain-specific optimizations and code generation for programs with a repetitive structure that cannot easily be packaged using available abstractions. OpenCL in particular relies on code

```
from figure1 import map
import numpy as np
import clq.opencl.pyopencl as cl

ctx = cl.Context.for_device(0, 0)
d_in = ctx.to_device(np.ones(1024))
d_out = ctx.alloc(like=d_x)
map(d_in, d_out, clq.opencl.sin,
    global_size=d_in.shape, local_size=(128,))
out = ctx.from_device(d_out)
```

Figure 5. A full OpenCL program using the cl.oquence Python bindings, including data transfer and kernel invocation.

```
class PtrType(Type):
  def __init__(self, target_type, addr_space):
    self.target_type = target_type
    self.addr_space = addr_space

  def verify_Subscript(self, context, node):
    slice_type = context.validate(node.slice)
    if isinstance(slice_type, IntegerType):
      return self.target_type
    else:
      raise TypeError('<error message here>')

  def translate_Subscript(self, context, node):
    value = context.translate(node.value)
    slice = context.translate(node.slice)
    return copy_node(node,
      value = value,
      slice = slice,
      code = value.code + '['+slice.code+']')
```

Figure 6. A portion of the implementation of OpenCL pointer types implementing subscripting logic using the cl.oquence extension mechanism.

generation as a fundamental mechanism, partially justifying the lack of support for higher-order programming.

cl.oquence supports code generation from directly within the metalanguage. A cl.oquence function can be constructed from a string containing its source using the `clq.from_source` function. It can also be constructed directly from a Python abstract syntax tree, available via the standard `ast` package, using the `clq.from_ast` function. We discuss a use case for code generation for simulation orchestration in cl.oquence in Section 4.

H. Active Language Extensions

We now turn our attention to the extension mechanism used by cl.oquence, as motivated by section ??. The cl.oquence type checker and compiler interacts directly with active libraries over an interface that gives substantial control over type checking and translation of an operation to a *type definition* associated with one of its operands according to a fixed dispatch protocol.

1) Dispatch Protocol: When the compiler encounters an expression, it must first verify that it type checks, then generate code for it. Rather than containing fixed logic for this, however, the compiler defers this responsibility to the *type* of a subexpression whenever possible. Below are example from the cl.oquence dispatch protocol. Due to space constraints, we do not list the entire dispatch protocol.

- Responsibility over a **unary operation** like `-x` is handed to the type assigned to the operand, `x`.
- Responsibility over **binary operations** is first handed to the type assigned to the left operand. If it indicates that it does not understand the operation, the type assigned to the right operand is handed responsibility, with a different method call[1].
- Responsibility over **attribute access**, `obj.attr`, and **subscript access**, `obj[idx]`, is handed to the type assigned to `obj`.

2) Verification and Translation: A type in cl.oquence is a metalanguage *instance* of a Python class deriving from `clq.Type`. The compiler hands control over an expression to a type by calling a method corresponding to the syntactic form of the expression according to the above dispatch protocol. Figure 6 gives the verification and translation logic governing subscript access for pointer types in OpenCL.

During the verification phase, the type of the primary operand, as determined by the dispatch protocol, is responsible for assigning a type to the expression as a whole. In this case, it recursively assigns a type to the slice operand. If it is assigned an integer type, the pointer's target type is assigned to the expression as a whole. Otherwise, a type error is raised with the provided error message (an ability that can be used by extension authors to satisfy the error message criteria described in Section **??**.)

If verification succeeds, the compiler must subsequently translate the cl.oquence expression into an expression in the output language, here OpenCL. It does so by again applying the dispatch protocol to call a method prefixed with `translate_`. This method is responsible for returning a copy of the expression's ast node with an additional attribute, `code`, containing the source code of the translation. In this case, it is simply a direct translation to the corresponding OpenCL attribute access, using the recursively-determined translations of the operands. More sophisticated abstractions may insert arbitrarily complex statements and expressions. The context also provides some support for non-local effects, such as new top-level declarations (not shown.)

3) Modular Backends: Thus far, we have discussed using OpenCL as a backend with cl.oquence. The OpenCL extension is the most mature as of this writing. However, cl.oquence supports the introduction of new backends in a manner similar to the introduction of new types, by extending the `clq.Backend` base class. Backends are provided as the first argument to the `@clq.fn` decorator, as can be seen in Figure 1. Backends are responsible for some aspects of

[1]Note that this operates similarly to Python's operator overloading protocol.

the grammar that do not admit simple dispatch to the type of a subterm, such as number and string literals or basic statements like `while`.

In addition to the OpenCL backend, preliminary C99 and CUDA backends are available (with the caveat that they have not been as fully developed or tested as of this writing.) Backends not based on the C family are also possible, but we leave such developments for future work.

4) Use Cases: The development of the full OpenCL language using only the extension mechanisms described above provides evidence of the power of this approach. Nothing about the core language was designed specifically for OpenCL. However, to be truly useful, as described in Sections **??** and **??**, the language must be able to support a wide array of primitive abstractions. We briefly describe a number of other abstractions that may be possible using this mechanism. Many of these are currently available either via inconvenient libraries or in standalone languages. With the cl.oquence extension mechanism, we hope to achieve robust, natural implementations of many of these mechanisms within the same language.

Partitioned Global Address Spaces: A number of recent languages in high-performance computing have been centered around a partitioned global address space model, including UPC, Chapel, X10 and others. These languages provide first-class support for accessing data transparently across a massively parallel cluster, which is verbose and poorly supported by standard C. The extension mechanism of cl.oquence allows inelegant library-based approaches such as the Global Arrays library to be hidden behind natural wrappers that can use compile-time information to optimize performance and verify correctness. We have developed a prototype of this approach using the C backend and hope to expand upon it in future work.

Other Parallel Abstractions: A number of other parallel abstractions, some of which are listed in **??**, also suffer from inelegant C-based implementations that spurred the creation of standalone languages. A study comparing a language-based concurrency solution for Java with an equivalent, though less clean, library-based solution found that language support is preferable but leads to many of the issues we have described [10]. The extension mechanism is designed to enable library-based solutions that operate as first-class language-based solutions, barring the need for particularly exotic syntactic extensions.

Domain-Specific Type Systems: cl.oquence is a statically-typed language, so a number of domain-specific abstractions that promise to improve verifiability using types, as discussed in Section **??**, can be implemented using the extension mechanism. We hope that this will allow advances from the functional programming community to make their way into the professional end-user community more quickly, particularly those focused on scientific domains (e.g. [11]).

Specialized Optimizations: In many cases, code optimization requires domain-specific knowledge or sophisticated, parametrizable heuristics. Existing compilers make implementing and distribution such optimizations difficult. With active libraries in cl.oquence, optimizations can be distributed directly with the libraries that they work with. For instance, we have implemented substantial portions of the NVidia GPU-specific optimizations described in [12] as a library that uses the extension mechanism to track affine transformations of the thread index used to access arrays, in order to construct a summary of the memory access patterns of the kernel, which can be used both for single-kernel optimization (as in [12]) and for future research on cross-kernel fusion and other optimizations.

Instrumentation: Several sophisticated feedback-directed optimizations and adaptive run-time protocols require instrumenting code in other ways. The extension mechanism enables granular instrumentation based on the form of an operation as well as its constituent types, easing the implementation of such tools. This ability could also be used to collect data useful for more rigorous usability and usage studies of languages and abstractions, and we plan on following up on this line of research going forward.

I. Availability

cl.oquence is available under the LGPL license and is developed openly and collaboratively using the popular Github platform[2]. Documentation and other learning material is being made available at http://cl.oquence.org/.

III. Case Study: Neurobiological Circuit Simulation

An important criteria that practitioners use to evaluate a language or abstraction, as discussed in Section **??**, is whether significant case studies have been conducted with it. In this section, we briefly (due to space limitations) discuss an application of the cl.oquence OpenCL library, Python host bindings and code generation features for developing a modular, high-performance scientific simulation library used to simulate thousands of parallel realizations of a spiking neurobiological circuit on a GPU.

A. Background

A neural circuit can be modeled as a network of coupled differential equations, where each node corresponds to a single neuron. Each neuron is modeled using one or more ordinary differential equations. These equations capture the dynamics of physically important quantities like the cell's membrane potential or the conductance across various kinds of ion channels and can take many forms [**?**]. Single simulations can contain from hundreds to tens of millions of neurons each, depending on the specific problem being studied. In some cases, such as when studying the effects of

[2]https://github.com/cyrus-/cl.oquence

noise on network dynamics or to sweep a parameter space, hundreds or thousands of realizations must be generated. In these cases, care must be taken to only probe the simulation for relevant data and process portions of it as the simulation progresses, because the amount of data generated is often too large to store in its entirety for later analysis.

The research group we discuss here (of which the first author was a member) was studying a problem that required running up to 1,000 realizations of a network of between 4,000 and 10,000 neurons each. An initial solution to this problem used the Brian framework, written in Python, to conduct these simulations on a CPU. Brian was selected because it allowed the structure of the simulation to be specified in modular and straightforward manner. This solution required between 60 and 70 minutes to conduct the simulations and up to 8 hours to analyze the data each time a parameter of the simulation was modified.

Unsatisfied with the performance of this approach, the group developed an accelerated variant of the simulation using C++ and CUDA. Although this produced significant speedups, reducing the time for a simulation by a factor of 40 and the runtime of the slowest analyses by a factor of 200, the overall workflow was also significantly disrupted. In order to support the many variants of models, parameter sets, and probing protocols, C preprocessor flags were necessary to selectively include or exclude code snippets. This quickly led to an incomprehensible and difficult to maintain file structure. Moreover, much of the simpler data analysis and visualization was conducted using Python, so marshalling the relevant data between processes also became an issue.

B. *The* cl.egans *Simulation Library*

In order to eliminate these issues while retaining the performance profile of the GPU-accelerated code, the project was ported to cl.oquence. Rather than using preprocessor directives to control the code contained in the final GPU kernels used to execute the simulation and data analyses, the group was able to develop a more modular library called cl.egans[3] based on the language's compile-time code generation mechanism and Python and OpenCL bindings.

cl.egans leverages Python's object-oriented features to enable modular, hierarchical simulation specifications. For example, Figure 7 shows an example where a neuron model (`ReducedLIF`) is added to the root of the simulation, a synapse model (`ExponentialSynapse`) is then added to it, and its conductance is probed in the same way, by adding a probe model as a child of the synapse model. If interleaved analysis needed to be conducted as well, it would be specified in the same way.

Implementations of these classes do not evaluate the simulation logic directly, but rather contain methods that generate cl.oquence source code for insertion at various

[3]...after *c. elegans*, a model organism in neuroscience

```
# Create the root node of the simulation
sim = Simulation(ctx, n_timesteps=10000)
neurons = ReducedLIF(sim, count=N, tau=20.0)
e_synapse = ExponentialSynapse(neurons, 'ge',
    tau=5.0, reversal=60.0)
probe = StateVariableProbeCopyback(e_synapse)
```

Figure 7. An example of a nested simulation tree, showing that specifying a simulation is both simple and modular.

points, called *hooks*, in the final simulation kernel. The hook that code is inserted into is determined by the method name, and code can be inserted into any hook defined anywhere upstream in the simulation tree. New hooks can also be defined in these methods and these become available for use by child nodes. Figure 8 shows an example of a class that inserts code in the `model_code` hook and defines several new hooks. This protocol is closely related to the notion of *frame-oriented programming*. Although highly modular, this strategy avoids the performance penalties associated with standard object-oriented methodologies via code generation.

Compared to a similar protocol targeting OpenCL directly, the required code generation logic is significantly simpler because it enables classes like `StateVariable` to be written generically for all types of state variables, without carrying extra parameters and *ad hoc* logic to extract and compute the result types of generated expressions. Moreover, because types are first-class objects in the metalanguage, they can be examined during the memory allocation step to enable features like fully-automatic parallelization of multiple realizations across one or more devices, a major feature of cl.egans that competing frameworks cannot easily offer.

Once the kernel has been generated and memory has been allocated, the simulation can be executed directly from Python using the bindings described in Section II-F2. The results of this simulation are immediately available to the Python code following the simulation and can be visualized and further analyzed using standard tools. Once the computations are complete, the Python garbage collector is able to handle deallocation of GPU memory automatically (a feature of the underlying `pyopencl` library [13].)

Using this cl.oquence-based framework, the benefits of the Brian-based workflow were recovered without the corresponding decrease in performance relative to the previous CUDA-based solution, leading ultimately to a satisfying solution for the group conducting this research.

IV. Related Work

V. Conclusion

Professional end-users demand much from new languages and abstractions. In this paper, we began by generating a concrete, detailed set of design and adoption criteria that we hope will be of broad interest and utility to the research community. Based on these constraints, we designed a new

```
class SpikingModel(Model):
  """Base class for spiking neuron models."""
  def in_model_code(self, g):
    """
    idx_state = idx_model + (realization_num -
        realization_start)*count
    """ << g  # << appends to code generator, g
    self.insert_cg_hook("read_incoming", g)
    self.insert_cg_hook("read_state", g)
    self.insert_cg_hook("calculate_inputs", g)
    self.insert_cg_hook("state_calculations", g)
    self.insert_cg_hook("spike_processing", g)
  # ...
```

Figure 8. An example of a hook that inserts code and also inserts new, nested hooks for downstream simulation nodes below that.

language, cl.oquence, making several pragmatic design decisions and utilizing advanced techniques, including type inference, structural typing, compile-time metaprogramming and active libraries, to uniquely satisfy many of the criteria we discuss, particularly those related to extensibility. We validated the extension mechanism with a mature implementation of the entirety of the OpenCL type system, as well as preliminary implementations of some other features. Finally, we demonstrated that this language was useful in practice, drastically improving performance without negatively impacting the high-level scientific workflow of a large-scale neurobiological circuit simulation project. Going forward, we hope that cl.oquence (or simply the key techniques it proposes, by some other vehicle) will be developed further by the community to strengthen the foundations upon which new abstractions are implemented and deployed into professional end-user development communities.

References

[1] J. Howison and J. Herbsleb, "Scientific software production: incentives and collaboration," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 2011, pp. 513–522.

[2] J. Hannay, C. MacLeod, J. Singer, H. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. IEEE Computer Society, 2009, pp. 1–8.

[3] J. Segal, "Some problems of professional end user developers," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, 2007, pp. 111–118.

[4] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana, "A survey of scientific software development," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010, p. 12.

[5] J. Carver, R. Kendall, S. Squires, and D. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, may 2007, pp. 550 –559.

[6] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *Software, IEEE*, vol. 25, no. 4, pp. 29–36, 2008.

[7] T. L. Veldhuizen and D. Gannon, "Active libraries: Rethinking the roles of compilers and libraries," in *Proc. 1998 SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, 1998. [Online]. Available: http://arxiv.org/abs/math/9810022

[8] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[9] D. Malayeri and J. Aldrich, "Is structural subtyping useful? an empirical study," *Programming Languages and Systems*, pp. 95–111, 2009.

[10] V. Cavé, Z. Budimlić, and V. Sarkar, "Comparing the usability of library vs. language approaches to task parallelism," in *Evaluation and Usability of Programming Languages and Tools*. ACM, 2010, p. 9.

[11] A. Kennedy, "Types for units-of-measure: Theory and practice," in *CEFP*, ser. Lecture Notes in Computer Science, Z. Horváth, R. Plasmeijer, and V. Zsók, Eds., vol. 6299. Springer, 2009, pp. 268–305. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17685-2

[12] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *ACM SIGPLAN Notices*, vol. 45, no. 6. ACM, 2010, pp. 86–97.

[13] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Computing*, 2011.