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

Cyrus Omar, Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
{comar,jonathan.aldrich}@cs.cmu.edu

***Abstract*—In this paper, we introduce the Ace programming language and demonstrate its practicality as a foundational tool for both high-performance computing researchers and practitioners. Ace is a statically-typed language that shares a syntax with Python and uses the Python language itself as a multi-purpose compile-time metalanguage. Ace has a minimal core that delegates control over many aspects of compilation to user modules via a novel extension mechanism called *active type-checking and translation (AT&T)*. Specifically, users specify new primitive types and operations by equipping type definitions – first-class objects in the metalanguage – with compile-time methods. The compiler invokes these methods when type checking and translating expressions, according to a fixed type dispatch protocol associated with each syntactic form.**

I. Introduction

Researchers design and implement novel parallel programming abstractions on a regular basis. Each abstraction aims to make some class of developers more productive by making appropriate trade-offs between performance, portability, verifiability and ease-of-use. It can be observed that the most widely-adopted abstractions to date are those that been implemented as libraries that can be easily deployed within existing, widely-used languages. Abstractions that require introducing new primitive constructs, changing the semantics of existing constructs in particular contexts, or controlling compilation in a fine-grained manner cannot take this approach. This has led to a proliferation of specialized programming languages, each designed around a few privileged abstractions. Unfortunately, such languages have not been widely adopted by practitioners.

Computer-aided simulations 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, generally individually or in small groups [**?**][**?**]. Most of this software targets conventional desktop hardware, but about 20% of scientists also target either local clusters or supercomputers for more numerically-intensive computations [**?**]. In recent years, GPUs and other kinds of coprocessors have emerged as powerful platforms for high-performance computing as well.

Many scientists and engineers prefer high-level languages like MATLAB, Python, R and Perl [**?**], due to their powerful domain-specific libraries and their focus on productivity and flexibility. These languages are typically interpreted rather than compiled and generally do not feature static type systems, relying instead on run-time ("dynamic") type lookup and indirection. Although this can increase the flexibility of the language, these features also negatively impact performance. Along code paths where performance appears to be a bottleneck, developers typically turn to traditional statically-typed low-level languages like C and Fortran [**?**]. These languages require explicit type annotations on variables and give users explicit control over data layout and heap allocation. Although this makes writing programs more tedious and error-prone, it can also significantly improve performance, particularly if attributes of the target architecture (e.g. caching behavior) are considered.

While a sequential reimplementation of a critical code path using a low-level language will produce a modest speedup, more significant speedups on modern hardware require parallelizing over many processor cores and, on massively-parallel machines, many interconnected nodes. Despite well-known difficulties, low-level approaches that use a form of shared-memory multithreading (e.g. pthreads, OpenMP) paired with explicit message passing between nodes (typically using MPI) remain widely used [**?**][**?**].

Although researchers often propose higher-level language features and parallel programming abstractions that aim to strike a better balance between raw performance, productivity, code portability and verifiability (see Section 2), end-users remain skeptical of new approaches. This viewpoint was perhaps most succinctly expressed by a participant interviewed in a recent study [**?**], 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 those in the research community working to advance the practice of scientific and high-performance computing. We begin in Section 2 by developing a set of design and adoption criteria for new languages and abstractions based on prior empirical studies of these *professional end-user developers*

as well as our observations of characteristics common to successful projects in the past.

We then introduce a new programming language named cl.oquence (pronounced "C eloquence") in Section 3. cl.oquence initially targets code paths where low-level languages, particularly OpenCL, would be used today. By employing a collection of powerful front-end language and compiler design techniques, making pragmatic design choices, and building on top of established infrastructure, we argue that cl.oquence may uniquely satisfy many of the criteria established in Section 2, particularly those related to ease of use, familiarity, performance and tool support.

Unlike many existing languages, cl.oquence is designed to be highly extensible from inception, employing a powerful compile-time language extension and code generation mechanism based around the concept of *active libraries* [**?**]. We argue that this feature may make cl.oquence particularly suitable as a platform for researchers developing new parallel abstractions and other specialized, high-level language features, as well as development tools.

In Section 4 we briefly describe a case study where the language was used to conduct large-scale spiking neural circuit simulations on GPU hardware, demonstrating the feasibility of this approach on a non-trivial problem. Finally, conclude in Section 6 with future directions for both the cl.oquence language and the research community.

II. Design and Adoption Criteria

The primary purpose of this paper is to describe the design of a programming language, cl.oquence. Unfortunately, as in many areas of design, it can be difficult to objectively evaluate the merit of such efforts. To better support such evaluations, researchers in design disciplines typically develop a set of high-level design criteria that serve as a rubric for evaluating and guiding their design efforts. In programming language design, particularly for scientific and high-performance computing, there have been few concerted efforts to develop a coherent set of design criteria that capture the needs of the targeted developer communities. Similarly, there have been few treatments of adoption criteria for new languages and abstractions in practice, an important issue given the slow rates of adoption today. We have purposefully separated these criteria from the specific language we describe in Section 3 in the hopes of starting a productive discussion within the community.

A. Professional End-User Developers

An important first step is to identify the developer communities targeted by our design efforts. Researchers in software engineering typically distinguish between *end-user developers* and *professional developers*. End-user developers have little formal training relevant to the tools that they use, and the tools themselves are relatively simple (spreadsheets, for example). Professional developers, on the other hand, have explicit training or extensive experience with software engineering and software development techniques. Scientists and engineers do not cleanly fall into either of these groups, however – although formal training in software development is rare [**?**], members of these groups are more technically literate and demanding than typical end-users. For this reason, researchers have proposed a third group, *professional end-user developers*, to describe "people working in highly technical, knowledge rich professions, such as financial mathematicians, scientists and engineers, who develop their own software in order to advance their own professional goals. [**?**]" As such, we consider the literature on novice and end-user programming (cf. [**?**]) in addition to that on programming language design for professional developers to justify the criteria we develop below. Language usability and adoption are not commonly studied topics, so many of the hypotheses we present below also rely on informal observations about languages and tools that have been successful in the past.

B. Design Criteria

We define design criteria as aspects of a language's design, rather than its implementation, that help developers express their intent naturally and correctly.

1) Concise, Familiar and Readable Syntax: Despite a considerable amount of evidence pointing toward the value of a well-designed syntax, particularly for end-users in specialized domains [**?**], the issue is sometimes marginalized. It is likely the case, however, that some libraries and languages experience low adoption due in part due to the use of a verbose, unfamiliar or unreadable syntactic style.

A simple and concise syntax is common to most of the high-level languages that are widely used in scientific computing. Cordy identifies the principle of conciseness with elimination of redundancy and the availability of reasonable defaults [**?**]. The high-level languages listed in the introduction have all either made optional, or removed entirely, much of the syntactic overhead characteristic of low-level languages, such as explicit variable declarations and extensive headers or preambles that contain information that can be inferred from the body of the program in most cases. They also typically feature simple array indexing syntax and concise literal forms for common data structures like arrays, sets and maps. A concise and minimal syntax eliminates unnecessary keystrokes and keeps more code visible on screen at a time. Studies have shown that there may be a correlation between lines of code entered and overall error rate, independent of other factors [**?**].

However, one can also go too far, so these benefits must be balanced with concerns about readability and familiarity. A number of principles have been proposed to operationalize the notion of code readability. The most widely-used collection of principles are Green's cognitive dimensions [**?**]. Of particular relevance is the notion of self-consistency, which

serves to ensure that similar forms have similar meaning and that there are few subtle or context-dependent distinctions that users must be mindful of. Another important principle has been called *closeness of mapping*, expressing the value of a close correspondence between mental models and the formal model as expressed concretely using the language.

Most researchers become familiar with formal notation by studying mathematics. In nearly all commonly used languages in scientific computing, the common mathematical notation is used whenever possible. In contrast, many academic languages have settled on alternative notational styles. For example, the LISP family of languages uses a highly uniform list-based notation, while most functional languages typically borrow notation for function invocation from the lambda calculus, using the form `f x` rather than `f(x)` for function invocation. Although both of these styles have certain benefits, they can impose mental burdens on users who continue to mentally translate them into more familiar notation [**?**]. Although these and related difficulties can decrease with experience, they remain an important barrier for new users to these languages.

Syntactic cues like whitespace and typography that do not have a formal meaning but rather exist to assist developers are called secondary notation [**?**]. Most languages are whitespace-insensitive, while others (notably, Python) enforce consistent uses of whitespace, both in pursuit of consistency and as a technique to eliminate the need for block delimiters. Additionally, a few languages, such as Mathematica, have support for more typographically-rich mathematical notation via a structural editing interface. These techniques appear to be helpful, although we do not know of formal studies that provide evidence for this claim.

2) Support for Multiple Paradigms and Abstractions: Although the difficulties of low-level and parallel programming are widely acknowledged, there is little consensus on which high-level abstractions are most appropriate for easing this burden. Indeed, it appears likely that no one abstraction will emerge triumphant over the others. Although library-based abstractions are useful, they often suffer from limitations of the language, particularly if they require compile-time support. Primitive language support for a parallel abstraction has been shown to be more usable than a library-based implementation in at least one case [**?**]. Languages that have been designed specifically to explore a single abstraction as a core language feature often see more limited adoption than library-based implementations because they remain difficult to use in circumstances for which an alternative abstraction is more appropriate. Examples of abstractions where language-based implementations remain less frequently used than library-based implementations include the actor model (e.g. Erlang), software transactional memory [**?**], and global arrays (e.g. UPC). As such, it is important that a language support several modes of operation, ideally without excessively favoring one over another.

3) Extensibility: Although support for multiple paradigms and primitive abstractions can be built into a language design, this leaves control in the hands of the language designer. Novel or domain-specific constructs that may be useful to a small number of users or in rare situations can be difficult to develop and distribute for this reason, leading to the proliferation of new languages as described above. Language extensibility mechanisms support these use cases by giving users the ability to develop new abstractions and constructs that behave as if they were primitive constructs. If a mechanism is powerful enough, nearly all language constructs may be implemented using it, greatly simplifying the core semantics of a language.

Dynamic languages commonly rely on mechanisms like operator overloading and metaobject protocols [**?**] to provide extensibility via indirection. For example, the Python language allows objects to overload nearly every operator and as well as operations like attribute lookup (`obj.attr`) and assignment. This mechanism is used by a number of libraries to create a more natural interface to a low-level API (e.g. `pycuda`). More open-ended dynamic mechanisms, such as programmatic macros, have also been widely studied but as of yet have seen little adoption in languages used by professional end-users.

Language extensibility for statically-typed languages, on the other hand, remains an active research area. Compile-time metaprogramming systems, which are related to programmatic macros, have been developed for a number of languages (e.g. Template Haskell [**?**]) but these too have seen relatively limited development or adoption thus far. In Section 3, we introduce a novel static language extension mechanism that uses type-based dispatch rather than direct manipulation of syntax trees.

Some researchers have advocated the use of compiler extension mechanisms, rather than mechanisms built into a language itself (cf. [**?**]). Modern compilers now offer some support for front-end language extensions, although often these can be quite difficult to use. Although potentially powerful, this approach can also lead to issues when extensions are not easily composable or when multiple compilers exist for a language. Back-end extensions (that is, extensions that preserve the semantics of the language, improving only performance) are better supported in modern compilers.

Domain-specific language frameworks are a related approach that can serve many of the same goals as language-based extension mechanisms [**?**]. These tools ease the development of "little languages" and allow for the development and distribution of language features as modules. Domain experts use these tools to develop highly specialized languages that capture domain requirements precisely and allow for natural and concise specifications of scientific models and other structures. A major issue with this approach arises at language boundaries – interoperability between different domain-specific languages is difficult due to feature

mismatches. Another issue is that domain-specific languages often outgrow their initial implementations and begin to need increasingly powerful general-purpose features.

All extensibility mechanisms must be used carefully. Indeed, inexperienced developers can abuse mechanisms like operator overloading to create inscrutable interfaces that cause more problems than they solve. Some languages (notably Java) have taken up the philosophy that even simple language extension mechanisms should not be in the hands of end-users for this reason. Although this continues to be debated, we argue that extensibility is crucial for parallel and scientific programming in particular due to the significant levels of ongoing research into new parallel abstractions and the diverse set of other domain-specific use cases.

4) Support for Higher-Order Constructs: A number of parallel programming data structures and algorithms are of higher-order, meaning that they take other functions or types as arguments or parameters. We argue that support for higher-order programming is critical to language usability in our target domains.

Well-known examples of functions that operate using other functions include fundamental parallel primitives like map, reduce and scan. Fortunately, most modern languages now support using functions as values. Languages like C and Fortran implement this using function pointers, although at considerable syntactic expense. However, unlike functional languages and most dynamic languages, they do not allow anonymous functions (that is, functions that are defined as inline expressions rather than statements), nor functions that close over variables in the surrounding scope. The most recent revision of C++ now has support for closures and closures are also being considered for a future revision of Java. OpenCL does not come with any support for higher-order functions, an issue we explore further in Section 3.

Functions and types that are parameterized by types are often referred to as *polymorphic* or *generic*. C++ supports this using its template system. Java, ML and Haskell support a simpler form of *parametric polymorphism*, and Haskell also supports a more flexible *type class* system. C++, Java and Fortran also support *function overloading*, allowing multiple versions of a function that differ only according to their argument types. C and OpenCL do not support any form of polymorphism or user-defined function overloading.

Dynamic languages do not require that variables be assigned a type, so generic functions are written using run-time checks that ensure that a particular value supports the specific interface that a function expects. This is sometimes known as "duck typing" and is generally considered as a useful feature, due to its considerable flexibility. A promising approach that resembles duck typing, while operating statically, is known as *structural typing* [**?**].

5) Modularity and Packaging: Modularity is a broad term that refers to mechanisms useful for combining and reusing independently developed libraries of code. Languages with good support for modularity promote information hiding, thus localizing the effect of code changes, and allow modules to communicate over well-defined interfaces. There remains widespread disagreement and considerable ongoing research on language support for modularity. Object-oriented methodologies, although common in industrial projects, are used less frequently in scientific codes due to a perceived loss of control and performance [**?**]. Many dynamic languages support modularity only implicitly using duck typing. Functional languages, on the other hand, often have module systems that enforce correct usage of an interface at compile-time, albeit at some notational cost [**?**].

The packaging and linking mechanisms available in a language can also significantly impact its usability. Languages like C and C++ use a fragile preprocessor-based packaging mechanism and require separate header files, which often leads to subtle errors and compilation inefficiencies. More recent high-level languages have developed a varied set of mechanisms that are significantly simpler.

6) Verifiability: Verifying that a program does not contain errors and that it will operate according to specification (if a specification exists, which can be rare in this domain [**?**]) is a critical concern across all areas of software development. Errors in scientific programs may arise due to problems in basic program logic, as in other domains, but also due to the accumulation of numerical approximation errors or by violation of domain-specific constraints (e.g. inconsistent scientific units.)

A number of design decisions can influence the difficulty of formal program verification. Broadly stated, unconstrained support for dynamic indirection and direct access to memory have made program verification more difficult in many commonly-used languages. As a result, many classes of errors are only caught at run-time, often due to edge cases that are difficult to test for.

Advanced type systems, matched with appropriately constrained data structures and control constructs, can dramatically increase the likelihood that an error is found at compile-time and even eliminate entire classes of errors [**?**]. A large portion of academic research on programming language design has focused on designing such type systems. Unfortunately, as with many new parallel abstractions, these generally require that a language be modified with new primitives and most work has either been with prototype languages or functional languages like Ocaml, Haskell or Coq, which, due to some of the factors mentioned here, have seen limited (though growing) adoption in scientific computing so far.

In the absence of a formal proof of correctness, users must test programs with specific inputs, relying both on language-provided run-time checks and user-provided run-time assertions to catch errors. Most languages have unit testing frameworks designed to minimize the burden of developing unit tests, though these are used infrequently

in science [**?**][**?**]. Several languages (e.g. Eiffel) also have support for specifying pre- and post-conditions for functions, so that the assertions that must hold about arguments are visible in the function signature, but this feature has not been well-supported by widely-used languages to date.

C. Adoption Criteria

While most of the design criteria described in the previous sections have been the topic of significant (and ongoing) research, there are a number of other, more practical issues that can significantly affect adoption of a new language or tool. It should be noted that certain design decisions may make it easier to satisfy these adoption to criteria as well, an important consideration given that incentive structures in academia often do not reward efforts to improve a language along these dimensions [**?**]. We give examples of such design decisions when we describe cl.oquence in Section 3.

1) Performance: Performance is, of course, a critical concern in scientific and (particularly [**?**]) high-performance computing. Low-level languages typically elect to give the user direct access to hardware primitives. High-level languages must rely on a sophisticated compiler to translate high-level abstractions into performant code. Although the latter approach would be ideal, program optimization remains an active research area with many open problems. Indeed, many relevant algorithms have been shown to be NP-Complete, so compilers must rely on heuristics. Humans remain better at inventing and applying heuristics, given enough motivation and experience, than computers in many cases. Indeed, even when the compiler can sometimes produce better optimizations, users generally insist on being able to form a mental model of what their code is doing at the machine-level so that they can *reason* about the effects of code changes on overall performance [**?**]. As such, users generally demand that low-level abstractions remain available alongside high-level abstractions. It may also be useful to support a model where programmers can formalize, package and directly apply optimization heuristics programmatically, rather than relying on a "black-box" compiler.

2) Portability: A number of surveys have revealed that portability is one of the most important issues considered by scientists and engineers [**?**][**?**]. Several processor architectures and operating systems are in widespread use, with more appearing on a fairly regular basis. Low-level languages like C have generally achieved a reasonable level of portability, although aspects of the language that are underspecified can be sources of errors. The OpenCL language was designed with portability as a major design criteria, and compilers for OpenCL are now available for a diverse collection of processor and accelerator architectures. High-level languages are generally highly portable.

It should be noted, however, that simple portability is not entirely sufficient – users want to be able to write programs that can be ported to new architectures and operating systems and achieve high performance without significant retuning. This remains an area of active research. Recent efforts to build abstractions that generate efficient code for devices with multi-level memories, as implemented in the Sequoia language [**?**], have progressed toward this goal.

3) Useful Error Messages: When the compiler or runtime system of a language generates an error, users must determine the source of the problem using the information provided in an error message. Studies have shown that good error messages can dramatically reduce the time it takes to debug programs [**?**]. Indeed, a widely-reported frustration with C++ is that it produces overly verbose and cryptic error messages, particularly when using templates.

4) Tool and Infrastructure Support: Modern software development now relies on a number of tools to assist with common tasks, such as debuggers, profilers, syntax-aware editors, documentation generators, style checkers and interactive interpreters. Similarly, most established languages benefit from a centralized packaged repository (e.g. PyPI) and a package manager that can automate the installation of new packages. Such support is not trivial to develop from scratch, but few early adopters are likely willing to do without these tools for non-trivial projects [**?**].

5) Backwards Compatibility: End-user communities have produced a number of highly tuned and tested packages that are widely used in their fields. Porting these libraries to a new language requires significant effort and few communities will be willing to do so, particularly before a language is very well established. There is generally a greater willingness to develop wrappers that invoke functions from these packages, however. A powerful foreign function interface, ideally able to handle native code as well as code in existing widely-used high-level languages, enables these efforts to proceed smoothly and can provide a language with a large package library relatively early in its development.

Many of the reasonably successful approaches to date are built incrementally on existing languages as libraries or as simple extensions to existing languages (e.g. OpenMP, UPC, Co-array Fortran, Cilk++, MPI, CUDA, OpenCL). Notably, this allows existing programs to continue to function correctly, while enabling the gradual integration of the new constructs in parts of the code that would derive the greatest benefit. In contrast, approaches that have required total rewrites have had more difficulty recruiting early adopters (e.g. X10 [**?**].)

6) Learning Material: Existing languages benefit from a large collection of books, presentations and tutorials that allow new users to get started quickly. The most mature languages benefit further from the availability of courses and professional training seminars. New languages often suffer due to the lack of such polished learning material, or in some cases, due to the lack of *any* significant learning material targeted at end users (rather than other language designers.)

7) Open Availability: New languages and abstractions are often viewed with suspicion if they do not come with source code that can be modified under a free software license. The most unencumbered licenses (other than public domain releases) are BSD-style licenses. So called *copyleft* licenses like the GPL are also popular, requiring primarily that modifications to the compiler be distributed under the same license.

8) Social Proof: Finally, users look to the language's user community for evidence that the language is a serious effort that will not be abandoned and that there are other developers building libraries and sharing information over established communication channels. The availability of non-trivial demonstrations have been cited in surveys as critical to adoption as well [**?**]. These criteria, as with many of the others described above, can be difficult to satisfy, particularly for "clean-slate" language designs, which may help to explain why language adoption is often slow. However, even projects with a narrower scope, such as new parallel or domain-specific abstractions, must tackle these issues.

III. The cl.oquence Language

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.

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 II-B1 and the adoption criteria described in Section II-C4 without significant development effort.

```
import clq
from clq.backends.opencl import
    OpenCL, get_global_id

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

Figure 1. Data-parallel higher-order map in cl.oquence. The `get_global_id` function is an OpenCL primitive function.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

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

Figure 2. An OpenCL kernel function that maps the sin function over a vector of double-precision floating-point numbers in the global memory address space.

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 II-C2.

In OpenCL itself, however, there is no way to write a function like `map`. As described in Section II-B4, 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 II-B1, 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* [**?**].

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 2.) The OpenCL extension we have developed automatically infers many of these annotations as well, consistent with the principle of conciseness described in Section II-B1.

2) Structural Polymorphism: In Section II-B4, 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 [**?**].

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 II-B1.

3) Higher-Order Functions: Section II-B4 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 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 [**?**]. 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

```
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 3. Programmatically specializing the `map` function from Figure 1 for use by the standalone compiler to produce code equivalent to Figure 2.

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 II-B5.

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 3 shows how the specific OpenCL function given in Figure 2 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,

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

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 4 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 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 II-B3. 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.

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

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 5 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 II-C3.)

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 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 II-B2 and II-B3, 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

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

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 II-B2, 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 [**?**]. 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 II-B6, 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. [**?**]).

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 [**?**] 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 [**?**]) 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/.

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

IV. Case Study: Neurobiological Circuit Simulation

An important criteria that practitioners use to evaluate a language or abstraction, as discussed in Section II-C8, 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 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 6 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 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 7 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 III-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 [**?**].)

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

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

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.

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 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.