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

Cyrus Omar and Jonathan Aldrich
Carnegie Mellon University, Pittsburgh, PA
{comar,aldrich}@cs.cmu.edu

## ABSTRACT

Researchers developing languages and abstractions for high-performance computing must consider a number of design criteria, including performance, verifiability, portability and ease-of-use. Despite the deficiencies of legacy tools and the availability of seemingly superior options, end-users have been reluctant to adopt new language-based abstractions. We argue that this can be largely attributed to a failure to consider three additional criteria: continuity, extensibility and interoperability. This paper introduces Ace, a language that aims to satisfy this more comprehensive set of design criteria. To do so, Ace introduces several novel compile-time mechanisms, makes principled design choices, and builds upon existing standards in HPC, particularly Python and OpenCL. OpenCL support, rather than being built into the language, is implemented atop an extensibility mechanism that also admits abstractions drawn from other seemingly disparate paradigms. The core innovation underlying this and other features of Ace is a novel reification of types as first-class objects at compile-time, representing a refinement to the concept of active libraries that we call *active types*. We validate our overall design by considering a case study of a simulation framework enabling the modular specification and efficient execution of ensembles of neural simulations across a cluster of GPUs.

## 1. 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, while about 20% of scientists also target either local clusters or supercomputers for more numerically-intensive computations [2]. To fully harness the power of these platforms, however, these so-called *professional end-user developers* [3] must increasingly write parallel programs.

Professional end-users today generally use dynamically-typed high-level languages like MATLAB, Python, R or Perl for tasks that are not performance-sensitive, such as small-scale data analysis and plotting [4]. For portions of their analyses where the performance overhead of dynamic type checking and automatic memory management is too high, they will typically call into code written in a statically-typed, low-level language, most commonly C or Fortran, that uses low-level parallel abstractions like pthreads and MPI [5, 6]. Unfortunately, these low-level languages and abstractions are notoriously difficult to use and automatic verification is intractable in general.

Researchers and domain experts often respond to these challenges by proposing novel language features that aim to strike an intermediate balance between **performance**, **verifiability**, **portability** and **ease-of-use**. Unfortunately, professional end-users rarely adopt new languages. Indeed, many end-users have become skeptical that novel approaches that originate in the research community can be practical. This viewpoint was perhaps most succinctly expressed by a participant 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 it may seem paradoxical, the ubiquity of this sentiment demands direct examination by researchers proposing novel abstractions and languages for eventual use by professional end-users in HPC.

We suggest three mutually-related design criteria that, unlike those in bold above, many languages and language-integrated abstractions have failed to adequately consider: **continuity**, **extensibility** and **interoperability**. These criteria encompass the intuitions that new abstractions will not be adopted in a vacuum, that programming systems must support change, and that interacting components of an application or workflow should be able to make use of different abstractions naturally and without the possibility of conflict arising at their interface boundaries.

In this paper, we introduce Ace, a programming language targeting professional end-users as well as researchers across high-performance computing and related domains. Ace has followed a principled design methodology guided by this more comprehensive set of design criteria in order to avoid many of the issues that have hindered previous language-based approaches in high-performance computing. These criteria, this design methodology, and the novel mechanisms and designs developed to satisfy these criteria constitute the generalizable contributions of this paper. We also hope that Ace itself will be useful to the HPC community[1], and we describe several use cases and an initial case study in order to preliminarily validate it's utility.

[1]Ace is openly available at http://acelang.org/.

We begin in Section 2 with simple examples that show how Ace can be used for low-level GPU programming and introduce the fundamental decisions made in its design. We discuss the motivations, based on the criteria above, behind key design decisions, including the use of static typing, a fixed syntax based on Python, a novel type propagation and inference scheme, and an explicit phase separation between compile-time and run-time logic. Each of these can be seen in Listings 1 and 2, which we will discuss further below.

The examples in Section 2 are based on an internalization of the OpenCL kernel programming language [7] as a library. This library is built atop a novel extensibility mechanism that we call *active typechecking and translation* (AT&T) and detail in Section 3. This mechanism relies on the core idea of representing types as first-class objects at compile-time. We refer to these objects as *active types* by analogy with *active libraries* [8] (see Section 6). Unlike prior approaches to extensibility where users globally modify the grammar or semantics of a language, and thus introduce conflicts between extensions, AT&T guarantees that extensions are composable by construction by pairing new rules with new types and limiting their scope to expressions of that type.

To demonstrate the flexibility of this mechanism beyond OpenCL, we continue in Section 3.3 by outlining examples of other, higher-level parallel abstractions that can be cleanly implemented using AT&T, including global address spaces, message passing and functional data parallelism. First-class support for each of these has required a new language in the past. AT&T allows compile-time logic to be safely included within libraries, and thus these first-class abstractions can be coexist naturally within a single program or workflow.

Ace can be used as a standalone language via the `acec` compiler and also as an interactive compilation environment. This mode of use, described in Section 4, uses just-in-time specialization to integrate the widely-adopted `numpy` library (including its internal type system) and OpenCL's host API (by a mechanism also available to other host APIs, such as CUDA's) with the Ace compiler itself to enable the direct invocation of Ace functions from within Python scripts with minimal overhead. Achieving this deep level of integration makes use of Ace's novel type representation as well.

To demonstrate the utility of Ace for scientific workloads, we describe in Section 5 a case study where Ace was used in this mode to develop a scientific simulation framework. This framework has been used to specify and efficiently execute thousands of realizations of a large stochastic neural circuit model on clusters of GPUs, achieving the same performance as raw OpenCL code while being more modular and concise than was feasible before.

We conclude in Sections 6 and 7 with related work and a discussion of the limitations of Ace at the time of writing, as well as a discussion of planned future work further validating Ace and building upon the concept of active typing.

# 2. LANGUAGE DESIGN AND USAGE

A variant of the standard "Hello, World!" example written in Ace is shown in Listing 1 and its compilation to statically-typed OpenCL kernel code is demonstrated in Listing 2. The OpenCL kernel language is a variant of C99 with some additions and restrictions to facilitate execution on GPUs and other accelerators, in addition to conventional CPUs. We will discuss it further in subsequent examples. We emphasize that this module, which we use throughout the paper,

**Listing 1** `[hello.py]` A basic Ace program demonstrating the two-phase structure of Ace programs and libraries.

```
1  from ace.OpenCL import OpenCL, printf
2
3  print "Hello, compile-time world!"
4
5  @OpenCL.fn
6  def main():
7      hello = "Hello, run-time world!"
8      printf(hello)
9  main = main.compile()
10
11 print "Goodbye, compile-time world!"
```

**Listing 2** Compiling `hello.py` using the `acec` compiler.

```
1  $ acec hello.py
2  Hello, compile-time world!
3  Goodbye, compile-time world!
4  $ cat hello.cl
5  __kernel void main() {
6      char* hello = "Hello, run-time world!";
7      printf(hello);
8  }
```

is simply a library like any other. The core of Ace gives no special treatment to it; it is distributed together with Ace for convenience. Aspects of its implementation will be described in Section 3.

This example demonstrates several key design decisions that characterize Ace: static typing of run-time behavior, a Python-based syntax, a phase distinction between compile-time and run-time logic, and programmatic compilation. We discuss each of these in the next three sections.

## 2.1 Static Typing and Syntax

Static type systems are powerful tools for programming language design and implementation. By tracking the type of a value statically, a typechecker can verify the absence of many kinds of errors over all inputs. This simplifies and increases the performance of the run-time system, as errors need not be detected dynamically using tag checks and other kinds of assertions. Many parallel programming abstractions are defined in terms of, or benefit from, a type system that enforces a communication protocol, ensures the consistency of data and simplifies the dynamics of the run-time system (see Section 3.3 for examples). Because **verifiability** and **performance** are key criteria and static typing is a core technique, Ace is fundamentally statically-typed.

It is legitimate to ask, however, why dynamically-typed languages are so widely-used in HPC. Although slow and difficult to reason about, these languages generally excel at satisfying the criteria of **ease-of-use**. More specifically, Cordy identified the principle of *conciseness* as elimination of redundancy and the availability of reasonable defaults [9]. Statically-typed languages, particularly those that HPC programmers are exposed to, are verbose, requiring explicit and often redundant type annotations on each function and variable declaration, separate header files, explicit template headers and instantiation and other sorts of annotations. The dynamically-typed languages used in HPC, on the other hand, avoid most of this overhead by relying on support from the run-time system. Ace was first conceived to explore the question: *does conciseness require run-time mechanisms, or*

**Listing 3** [listing3.py] A generic data-parallel higher-order map function written using the OpenCL user module.

```
1  from ace.OpenCL import OpenCL, get_global_id
2
3  @OpenCL.fn
4  def map(input, output, f):
5      gid = get_global_id(0)
6      output[gid] = f(input[gid])
```

**Listing 4** [listing4.py] The generic `map` function compiled to map the `add5` function over two types of input.

```
1  from listing3 import map
2  from ace.OpenCL import gptr, double, int
3
4  @OpenCL.fn
5  def add5(x):
6      return x + 5
7
8  D = gptr(double); I = gptr(int); A = add5.ace_type
9  map_add5_dbl = map.compile(D, D, A)
10 map_add5_int = map.compile(I, I, A)
```

*can one develop a statically-typed language with the same low-level memory and execution model of C but syntactic overhead comparable to a high-level scripting language?*

Rather than designing a new syntax, or modifying the syntax of C, we chose to utilize, *without modification*, the syntax of an existing language, Python. This choice was not arbitrary, but rather a key means by which Ace achieves both **ease-of-use** and **continuity**. Python's whitespace-delimited syntax is widely regarded as both concise and readable, and Python is amongst the most widely-adopted languages in computational science [10]. By directly adopting Python's syntax, Ace's syntax is immediately *familiar* and *acceptable* to a significant segment of the intended audience. Moreover, a key benefit of adopting it without modifications is that any tools that handle Python source code, including parsers, editors, style checkers and documentation generators, can be used on Ace code without modification.

## 2.2 Phase Separation

Ace's **continuity** with Python does not stop at its syntax. Perhaps the most immediately apparent departure from the standard "Hello, World!" example comes on lines 3 and 10 of Listing 1, which contain Python `print` statements that are executed at *compile-time*. Ace programs and libraries are Python scripts at the top-level, rather than a list of declarations as in most conventional languages. In other words, Python is the *compile-time metalanguage* of Ace.

A consequence of this choice is that Ace can leverage Python's well-developed package system and associated distribution infrastructure directly (e.g. Line 1). This serves to address another key **ease-of-use** issue often associated with C-based languages: the fragility of the preprocessor-based packaging system they historically have relied upon.

Ace functions defining run-time logic are introduced by a decorator. The `@OpenCL.fn` decorator on Line 5 indicates that the `main` function is a statically-typed Ace function targeting the OpenCL backend for code generation (Section 4). Without this decorator, the function would simply be a conventional Python function that could be called only at compile-time. Doing so would fail here: `printf` is not a Python function. Rather, it is a compile-time Python object representing an OpenCL primitive residing in the `OpenCL` module. It has an associated Ace type that controls what types of input it can receive and the type of its output given its input type (see Section 3).

**Listing 5** [listing4.cl] The OpenCL code generated by running `acec listing4.py`.

```
1  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2
3  double add5(double x) {
4      return x + 5;
5  }
6
7  __kernel void map_add5_dbl(
8    __global double* input,
9    __global double* output)
10 {
11     size_t gid;
12     gid = get_global_id(0);
13     output[gid] = add5(input[gid]);
14 }
15
16 int add5__1(int x) {
17     return x + 5;
18 }
19
20 __kernel void map_add5_int(
21   __global int* input,
22   __global int* output)
23 {
24     size_t = gid;
25     gid = get_global_id(0);
26     output[gid] = add5__1(input[gid]);
27 }
```

## 2.3 Programmatic Compilation

As defined on Lines 5-7, `main` is a *generic function*. This terminology in Ace is used to indicate a function for which types have not yet been assigned to arguments. Invoking the `compile` method on a generic function with a sequence of argument types invokes the active typechecking and translation mechanism we will describe in Section 3 to produce a *concrete function* – one with a single type for each argument, internal variable (such as `hello`) and the return value. In Listing 1, we name this concrete function `main`, overwriting the generic function of the same name that it is derived from. The compiler does nothing apparently interesting: there are no arguments, the single internal variable takes the OpenCL string type `char*` from its value and the return type is `void`.

Before moving on to more interesting examples, let us discuss the `acec` compiler shown operating at the shell in Listing 2. The `acec` compiler operates in two steps:

1. Executes the provided Python file (`hello.py`) which contains Ace functions and (in future examples) types and compile-time code generation logic.

2. Produces source code for concrete functions (produced using the `compile` method) in the top-level Python environment and any other concrete functions, type declarations and other program items required by or generated by these functions. This may produce one or more files (here, just `hello.cl`).

We will show in Section 4 that for backends with Python bindings, such as OpenCL, CUDA and C, generic functions can be executed directly, without this explicit compilation step to concrete functions, if desired.

## 2.4 Example 2: Higher-Order Map

The "Hello, World!" example demonstrates the structure of Ace programs, but it does not require working with types. Listing 3 shows an imperative, data-parallel map primitive written using the OpenCL library introduced above. To review, in OpenCL users can define functions, called *kernels*, that execute across thousands of threads. Each kernel has access to a unique index, called its *global id*, which can be to ensure that each thread operates on different parts of the input data (Line 5). The `map` kernel defined in Listing 3 applies a transfer function, `f`, to the element of the input array, `input`, corresponding to its global id. It writes the result of this call into the corresponding location in the provided output array, `output`.

As above, `map` is a *generic function* (specifically, it is an instance of the class `ace.GenericFn`). This means that its arguments have not been assigned types. The functionality given by the `map` definition is in fact applicable to many combinations of types for `input` and `output` and functions, `f`. In this sense, `map` is actually a *family* of functions defined for all types assignments for `input`, `output` and `f` such that the operations in the function's body are well-defined.

Running `acec listing3.py` would produce no output. To create a *concrete function* (that is, an instance of the class `ace.ConcreteFn`) that can be emitted by the compiler, types must be assigned to each of the arguments. Listing 4 shows how to use the `compile` method to specialize `map` in *two* different ways to apply the `add5` function, defined on Lines 4-6, to arrays that reside in global memory (OpenCL associates a memory space with pointer types). Line 9 results in a version specialized for arrays of `double`s and Line 10 results in a version for arrays of `int`s. The output of compilation is shown in Listing 5.

## 2.5 Types as Metalanguage Objects

The `compile` method assigns types to the arguments of a generic function. In Listing 4, the types we are using are given shorter names, for convenience, on Line 8 (that is, variables in the metalanguage can be used like `typedef` is used in a C-like language). The types `int` and `double` imported from the `OpenCL` module correspond to the OpenCL types of the same name. The types `gptr(int)` and `gptr(double)` correspond to `__global int*` and `__global double*`.

These types are *objects in the metalanguage*, Python. More specifically, types are instances of user-defined classes that inherit from the Ace-provided `ace.Type` class. For example, `gptr(double)` is an instance of `OpenCL.PtrType` instantiated with the target type, `double`, and memory space, `__global`, as constructor arguments. The types `double` and `int` are instances of `OpenCL.FloatType` and `OpenCL.IntegerType`, respectively. This notion of types as metalanguage objects is key to the Ace compilation model and also enables other mechanisms that we will discuss in subsequent sections.

## 2.6 Type Propagation

The type assigned to the third argument, `f`, on both Lines 4.9 and 4.10, is `add5.ace_type`. The `ace_type` attribute of a generic function is an instance of `ace.GenericFnType`, the type of Ace generic functions. Ace generic functions are compiled to concrete functions automatically at all internal call sites. That is, when the compiler encounters the call to `f` inside `map` when compiling `map_add5_double`, it compiles a version of `add5` specialized to the `double` type (seen on Line 5.3), and similarly when compiling `map_add5_int` (on Line 5.16, automatically given a unique name to avoid conflicts). This mechanism is called *type propagation*. We did not need to use `add5.compile(double)` before compiling `map_add5_dbl` because only functions that are never called in the process of compiling other functions in a module need type information explicitly provided, supporting **ease-of-use** by increasing conciseness.

**Listing 6** `[listing6.py]` A function demonstrating whole-function type inference when multiple values with differing types are assigned to a single identifier, `y`.

```
1  from ace.OpenCL import OpenCL, int, double, long
2
3  @OpenCL.fn
4  def threshold_scale(x, scale):
5      if x <= 0:
6          y = 0
7      else:
8          y = scale * x
9      return y
10
11 f = threshold_scale.compile(int, double)
12 g = threshold_scale.compile(int, long)
13 assert f.return_type == double
14 assert g.return_type == long
```

In effect, this scheme allows for a form of higher-order functional programming even when targeting languages, like OpenCL, that have no support for higher-order functions (OpenCL, unlike C99, does not support function pointers). This works because the `ace.GenericFnType` for one function, such as `add5`, is not equal to the `ace.GenericFnType` for a superficially similar function, such as `add6` (defined as one would expect). To put it in type theoretic terms, `ace.GenericFnType`s are singleton types, *uniquely inhabited* by a single generic function. A consequence of this is that they are not useful as first-class values (i.e. they cannot be written into a collection). This is often valuable, particularly in parallel programming where compile-time specialization is valuable to avoid **performance** and **ease-of-use** issues that occur when using function pointers.

Concrete functions, on the other hand, can be given a true function type (e.g. `add5` could be compiled to a concrete function with type `int` $\rightarrow$ `int`) if targeting a backend that supports them, such as C99, or by using an integer-indexed jump table in OpenCL (we have not implemented this mechanism using Ace as of the time of writing, but do not anticipate difficulties in doing so).

Type propagation via generic functions can be compared to template specialization in C++, where both the template headers (containing nested template parameters for function arguments) and specialization parameters at any call sites are inferred automatically from usage. This significantly simplifies a sophisticated feature of C++ and introduces it to OpenCL and C, which do not support templates. Other uses for C++ templates are subsumed by the metaprogramming features discussed in Section 5.

## 2.7 Whole-Function Type Inference

On Line 5 in the generic `map` function in Listing 3, the variable `gid` is initialized with the result of calling the OpenCL primitive `get_global_id`. The type for `gid` is never given explicitly. This is a simple case of Ace's more general *whole-function type inference*. In this case, `gid` will be inferred

to have type `size_t` because that is the return type of `get_global_id` (as defined in the OpenCL specification, which the `ace.OpenCL` module follows). The result can be observed on Lines 11 and 24 in Listing 5.

Inference is not restricted within single assignments, as in the `map` example, however. Multiple assignments to the same identifier with values of differing types, or multiple return statements, can be combined if the types in each case are compatible with one another (e.g. by a subtyping relation or an implicit coercion). In Listing 6, the `threshold_scale` function assigns different values to `y` in each branch of the conditional. In the first branch, the value `0` is an `int` literal. However, in the second branch of the loop, the type depends on the types of both arguments, `x` and `scale`. We show two choices for these types on Lines 11 and 12. Type inference correctly combines these two types according to OpenCL's C99-derived rules governing numeric types (defined by the user in the `OpenCL` module, as we will describe in Section 3). We can verify this programmatically on Lines 12 and 13. Note that this example would also work correctly if the assignments to `y` were replaced with `return` statements (in other words, the return value of a function is treated as an assignable for the purpose of type inference).

## 2.8 Annotation and Extension Inference

In addition to type annotations, OpenCL normally asks for additional annotations in a number of other situations. Users can annotate functions that meet certain requirements with the `__kernel` attribute, indicating that they are callable from the host. The `OpenCL` backend can check these and add this annotation automatically. Several types (notably, `double`) and specialized primitive functions require that an OpenCL extension be enabled via a `#pragma`. The OpenCL backend automatically detects many of these cases as well, adding the appropriate `#pragma` declaration. An example of this can be seen in Listing 5, where the use of the `double` type triggers the insertion of the `cl_khr_f64` extension.

# 3. EXTENSIBILITY

Thus far, we have been discussing the `OpenCL` module in our examples. This module faithfully implements all the types and operations of the OpenCL kernel language, a portable standard for writing low-level code on multi-core processors and accelerators [7]. The functions we wrote have used these primitives with the `OpenCL.OpenCL` backend, so the translation has been direct and no run-time overhead has been introduced. Thus, as described so far, Ace has affirmatively resolved the question it was originally conceived to address, discussed in Section 2.1.

## 3.1 Monolithic vs. Extensible Languages

OpenCL is not necessarily the best tool for every job in high-performance computing. Indeed, HPC is an area where designing a set of primitives that satisfy all users has been particularly challenging, and it appears unlikely that a broad consensus will emerge given the variety of architectures, applications, scales and user communities that it serves, and the number of seemingly promising abstractions that emerge continuously targeting various subsets of this problem space.

It is therefore a concern that most programming languages are *monolithic* – a collection of primitives are given first-class treatment by the language implementation, and users can only creatively combine them to implement algorithms and abstractions of their design. Although highly-expressive general-purpose mechanisms have been developed (such as object systems or algebraic datatypes), these may not suffice when researchers or domain experts wish to evolve aspects of the type system, exert control over the representation of data, introduce specialized run-time mechanisms, or if defining an abstraction in terms of existing mechanisms is unnatural or verbose (in summary, to push the boundaries of **verifiability**, **performance** and **ease-of-use**). In these situations, it would be desirable to have the ability to modularly extend existing systems (**continuity**) with new compile-time logic (**extensibility**) and be assured that such extensions will never interfere with one another when used in the same program (**interoperability**).

**Listing 7** `[ace.OpenCL]` A portion of the implementation of OpenCL pointer types implementing subscripting logic using the Ace extension mechanism, AT&T.

```
1  import ace
2
3  class PtrType(ace.Type):
4    def __init__(self, target_type, addr_space):
5      self.target_type = target_type
6      self.addr_space = addr_space
7
8    def resolve_Subscript(self, context, node):
9      slice_type = context.resolve(node.slice)
10     if isinstance(slice_type, IntegerType):
11       return self.target_type
12     else:
13       raise TypeError('<error message>', node)
14
15   def translate_Subscript(self, context, node):
16     value = context.translate(node.value)
17     slice = context.translate(node.slice)
18     return ace.copy_node(node,
19       value=value, slice=slice,
20       code=value.code + '[' + slice.code + ']')
21
22   # ...
23
24 def gptr(target_type):
25     return PtrType(target_type, "__global")
```

## 3.2 Active Typechecking and Translation

To achieve these criteria, Ace has been designed around a minimal, extensible core. Users introduce the compile-time logic associated with primitive types and operations from *within libraries*, as opposed to by some language-external mechanism such as a domain-specific language framework or extensible compiler (see Section 6).

The two phases of compilation that can be controlled by users are together referred to as *active typechecking and translation (AT&T)*. They are invoked the first time the `compile` method of a generic function is called, or when type propagation occurs, with a particular type assignment (a cached concrete function is returned subsequently).

### 3.2.1 Active Typechecking

When the compiler encounters an expression, it must first verify its validity by either assigning it a type or raising a meaningful type error. Rather than defining fixed logic for this, Ace defers control to the type of the *primary operand* according to a *dispatch protocol*. Because types are metalanguage instances of user-defined types derived from `ace.Type` (cf. Section 2.5), dispatch corresponds to calling a method

of this base class.

Let us again consider the data-parallel map example of Section 2. When `map` is compiled on Line 9 of Listing 4, the first two argument types are `gptr(double)`. This type is an instance of the user-defined `OpenCL.PtrType` which inherits from `ace.Type`. When the compiler encounters the expression `input[gid]` on Line 6, the dispatch protocol is to defer control to the type of `input` by invoking the method named `resolve_Subscript`.

The relevant portion of `OpenCL.PtrType` as well as `gptr` is shown in Listing 7. The `resolve_Subscript` method on Line 8 receives a context and the syntax tree of the node being considered. The context contains information about variables in scope and other contextual information, and also exposes a method, `resolve_type`, that this method uses to recursively resolve the types of other subexpressions, here the slice which will be `gid`, as needed. The context will have that `gid` is the machine-dependent integer type `size_t` as discussed in Section 2. On Line 10, it confirms that this type is an instance of an integer type and thus, it assigns the whole expression, `input[gid]`, the target type of the pointer, `double`. Had the function attempted to index `input` using a non-integer expression, the method would take the other branch of the conditional and raise a type error, with a custom error message, on Line 13. We note that error messages are an important component of **ease-of-use** [11]. Indeed, a widely-reported frustration with C++ is that it produces overly verbose and cryptic error messages, particularly when templates are used in clever ways.

#### 3.2.2 Dispatch Protocol

Below are examples of the rules that comprise the Ace dispatch protocol. Due to space constraints, we do not list the entire dispatch protocol, which contains a rule for each possible syntactic form in the language.

- Responsibility over **unary operations** like `-x` is given 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, via a different method call. Note that this operates similarly to the Python run-time operator overloading protocol; see Section 6.
- Responsibility over **attribute access** (`e.attr`), **subscript access** (`e[idx]`) and **calls** (`e(e1, ..., en)`) is handed to the type assigned to `e`.
- In support of the type inference mechanism described in Section 6, responsibility to resolve a type given multiple assignments can be taken by any of the types of the assignments, with priority given to later types.

#### 3.2.3 Active Translation

Once typechecking a method is complete, the compiler must subsequently translate each Ace source expression into an expression in the target language. This has been OpenCL in the examples thus far, but we describe a generalization of this in the next section.

It does so by again applying the dispatch protocol to call a method of the form `translate_`$X$, where $X$ is the syntactic form of the expression. 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, represented here as a string though it may also be represented in a structured manner. In our example, it is simply a direct translation to the corresponding OpenCL attribute access (Line 20), using the recursively-determined translations of the operands (Lines 16-17). More sophisticated abstractions may insert arbitrarily complex statements and expressions during this phase. The context also provides some support for non-local insertions, such as new top-level type declarations, imports and helper code (not shown).

#### 3.2.4 User-Defined Backends

Thus far, we have discussed using OpenCL itself as a backend for our implementation of the OpenCL primitives. This backend is called `OpenCL` in the `ace.OpenCL` module. Ace supports the definition of new backends in a manner similar to the introduction of new types, by extending the `ace.Backend` base class. Backends are specified for a function by using the decorator form `@backend.fn`, as can be seen in the previous examples. 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 statement forms like `while` (though `for` is handled by an iterator-like protocol, not shown).

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.) This allows us to use the OpenCL kernel language, which offers a simplified variant of C99, without relying on the full OpenCL stack, which may not be well-supported by vendors with proprietary solutions such as CUDA. Backends not based on the C family are also possible, but we leave such developments for future work. During the translation phase, types can access the backend via the context and emit different code for different supported backends (not shown above due to space considerations).

#### 3.2.5 Composability and Interoperability

Because a type can only exert control over typechecking and translation of operations where an expression of that type is the primary operand, extensions defined using AT&T can not interfere with one another by construction. This does not imply that there are no **interoperability** issues to consider, however. A type may need to know about other types (e.g. pointer types need to know about integer types) and if this is done without consideration of future extensions, it may be difficult to integrate data produced by one type system with another. These issues cannot easily be addressed by a language design, however.

### 3.3 Use Cases

The development of the full OpenCL language using only the extension mechanisms described above provides evidence of the power of this approach. However, to be truly useful, the mechanism must be able to express a wide array of higher-level primitive abstractions. We briefly describe a number of other abstractions that are possible using this mechanism. Many of these are currently available in existing languages either via libraries or as primitives of some specialized language. 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 [12].

### 3.3.1 Parallel Programming

#### *Partitioned Global Address Spaces.*

A number of recent languages designed for clusters use a partitioned global address space model, including UPC [13], Chapel [14] and others. These languages provide first-class support for accessing data transparently across a massively parallel cluster. Their type systems track information about *data locality* so that the compiler can emit more efficient code. The extension mechanism can also track this information in a manner analogous to how address space information is tracked in the OpenCL example above, and target an existing portable run-time such as GASNet [15].

#### *Object Models and Message Passing.*

Classes in object-oriented systems can be understood as inducing a type parameterized by a static type specification that emits code to perform dynamic dispatch by some protocol. In Ace, the object system would be a subclass of `ace.Type` (e.g. `JavaObject`), the specification would be a compile-time Python object (e.g. a dictionary mapping names to types, as is done with OpenCL `struct`s, augmented with information about subtyping and methods in more complex systems). The code for checking field and method accesses against the specification would be be in `resolve_Attribute` method and the dynamic dispatch mechanism would be emitted in `translate_Attribute`. That is, Ace supports arbitrary object models, rather than fixing a single one that may not fit all needs, as in almost all other object-oriented languages.

Many parallel programming abstractions can be understood as implementing object models with complex forms of dynamic dispatch. For example, in Charm++, dynamic dispatch involves message passing over a dynamically load-balanced network [16]. While Charm++ is a very sophisticated system, and we do not anticipate that implementing it as an Ace extension would be easy, it is entirely possible to do so using a system like Adaptive MPI that exposes its runtime system [17]. Erlang and other message-passing based systems can also be considered in this manner – processes can be thought of as objects, channels as methods and messages as calls.

#### *Functional Parallelism.*

Another promising approach for parallel programming is in automatic parallelization of purely functional programs. These languages express computations that do not explicitly manipulate memory. They rely instead on the composition of primitives like `map` and `reduce` that transform components of persistent data structures like lists, trees and maps. Data dependencies are thus directly encoded in programs, and automatic parallelization techniques have shown promise in using this information to automatically schedule these parallel programs on concurrent hardware and networks.

The Copperhead language, for example, is based on Python as well and allows users to express functional programs over lists using common functional primitives, compiling them to CUDA code [18]. We believe that Ace can express precisely the same programming model, implemented as an extension. We have prototyped support for a number of common persistent data structures, including algebraic datatypes, in Ace. Perhaps surprisingly, Python's imperative syntax does not preclude writing programs in a reasonable functional style (due largely to its support for tuples).

### 3.3.2 Other Use Cases in HPC

#### *Interoperability Layers.*

The design criteria of **continuity** and **interoperability** require consideration of the large body of codes written in a variety of existing languages, across a number of paradigms. Although it is often possible to call from one language to another using a foreign function interface (FFI), this is almost never natural or statically safe. In Ace, however, extensions could be written that internalize the foreign language's type system (note that dynamically-typed languages can be considered to have a single type, often called `dyn`) and emit code that hides the FFI from users, achieving **verifiability** and **ease-of-use**.

#### *Specialized Optimizations.*

In many cases, specialized code optimizations requires statically tracking invariants throughout a program. Often these optimizations can be encoded as a type system for this reason. For instance, a course project using Ace implemented substantial portions of the GPU-specific optimizations described in [19] as a library, using types to track affine transformations of the global ID in order to construct a summary of the memory access patterns of the kernel. This information can be used both for single-kernel optimization (as in [19]) and for future research on cross-kernel fusion and other optimizations.

#### *Domain-Specific Type Systems.*

Although not strictly related to HPC, a number of domain-specific type systems related to computational science can be implemented within Ace. For example, prior work has considered tracking units of measure (e.g. grams) statically to validate scientific code [20]. This cannot easily be implemented using existing abstraction mechanisms because this information should only be maintained statically to avoid excessive run-time overhead associated with tagging. The Ace extension mechanism allows this information to be tracked in the type system, but not included during translation.

#### *Instrumentation.*

Several sophisticated feedback-directed optimizations and adaptive run-time protocols require instrumenting code in various ways. The extension mechanism combined with support for patching classes dynamically using Python enables granular instrumentation that can consider both the syntactic 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.

## 4. ACE: A COMPILATION ENVIRONMENT

As discussed in the Introduction, a common workflow for professional end-users is to use a high-level scripting lan-

**Listing 8** [listing8.py] A full OpenCL program using the `Ace.OpenCL` Python bindings, including data transfer to and from a device and direct invocation of a generic function, `map`, as a kernel without explicit compilation.

```
1  import numpy as np
2  import ace.OpenCL.bindings as cl
3  from listing3 import map
4  from listing4 import add5
5
6  cl.ctx = cl.Context.for_device(0, 0)
7
8  input = np.ones((1024,))
9  d_in = cl.to_device(input)
10 d_out = cl.alloc(like=d_in)
11
12 map(d_in, d_out, add5,
13     global_size=d_in.shape, local_size=(128,))
14
15 out = cl.from_device(d_out)
16 assert (out == input + 5).all()
```

guage for orchestration, small-scale data analysis and visualization and call into a low-level language for performance-critical sections. Python is designed for this style of use [21] and is widely used by professional end-users in HPC as a high-level scripting language. It features mature support for calling into code written in low-level languages. Developers can call into native libraries using its foreign function interface (FFI), or by using a wrapper library like `pycuda` for code compiled with CUDA [22] or `weave` for C and C++.

Although C, C++ and CUDA's compilers are separate executables on the system, the OpenCL language was also designed for this workflow, in that it exposes the compiler and memory management directly as an API, called the *host API*. The `pyopencl` module exposes this API and supports basic interoperability with `numpy`, the low-level linear algebra package for Python. With both `pyopencl` and `pycuda`, users generate OpenCL or CUDA source code as strings, compile it programmatically, then execute it using the runtime APIs that each library provides [22]. This mode of use can be considered one where Python serves as an *interactive compilation environment*.

Ace supports a refinement to this workflow, as an alternative to the `acec` compiler described above. In other words, `acec` can generate source code for kernels but it does not specify how and from what language thet will be called. This mode of use allows Python to be the calling language.

At the time of writing, this is only supported with the OpenCL backend, but other backends can implement this feature as well by satisfying a simple interface specification. The OpenCL host API wraps and is a superset of the `pyopencl`, adding a simpler type-aware API. Both generic functions and concrete functions can then be called like regular Python functions, with additional keyword arguments specifying the global and local size (i.e. the number of threads and how they are grouped). Device buffers can carry type information, unlike in the basic OpenCL host API, and this type information can be propagated from Python to Ace functions directly, as if the call had been within a kernel, eliminating the `compile` step that we have been using thus far and thus all mention of types.

An example of this for the generic `map` function defined in Listing 3 is shown in Listing 8, with the call itself on Lines 12-13. The first two arguments to `map` are OpenCL buffers,

**Listing 9** [listing9.py] Metaprogramming with Ace, showing how to construct generic functions from both strings and abstract syntax trees, and how to manipulate syntax trees at compile-time.

```
1  from ace.OpenCL import OpenCL
2  import ace.astx as astx
3
4  plus = OpenCL.fn.from_str("""
5  def plus(a, b):
6      return a + b
7  """)
8
9  add5_ast = astx.specialize(plus.ast, b=5)
10 add5 = OpenCL.fn.from_ast(add5_ast)
```

generated using a simplified wrapper to the `pyopencl` APIs on Lines 9-10. This wrapper associates type information with each buffer, based on the type of the `numpy` array, and this is used to implicitly compile `map` as appropriate the first time it is called for any given combination of input types. Notice also that `add5` is passed in directly.

By way of comparison, the same program written using the OpenCL C API directly is an order of magnitude larger and significantly more difficult to read and understand. It does not support higher-order functions nor is there any way to write `map` in a type-generic way. A full implementation of the logic of `map` written using the `pyopencl` bindings and metaprogramming techniques as described in [22] is still twice as large and more difficult to comprehend than the code we have shown thus far. 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 a lack of space, the reader is referred to the language documentation.

# 5. CASE STUDY

An important criteria that practitioners use to evaluate a language or abstraction is whether significant case studies have been conducted with it, sometimes called **social proof** [6]. In this section, we briefly (due to space constraints) show an application of the mode of use discussed above to support a modular, high-performance scientific simulation library. This library was used to simulate thousands of parallel realizations of a stochastic spiking neurobiological circuit. The realizations were distributed over a cluster of GPUs using an existing Python library supporting remote procedure calls. This relied critically on Ace's representation of types to determine the memory usage of each realization automatically, and thus distribute realizations and allocate memory on each node appropriately. It also used metaprogramming features of Ace to generate kernels and specialize them to each node the simulation was running on.

## 5.1 Metaprogramming in Ace

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 captured using available abstractions [22]. OpenCL in particular relies on code generation as a fundamental mechanism, which is cited as justification for its lack of support for higher-order programming. Ace supports programmatic compilation, higher-order func-

tions and a flexible language extension mechanism, so many use cases for metaprogramming have been eliminated.

However, there are still valid uses for this feature and Ace fully supports it. On Lines 4-7 of Listing 9, an Ace function is constructed from a string containing its source using the `from_str` variant of the `OpenCL.fn` method. It can also be constructed directly from an abstract syntax tree (AST), as implemented by the Python standard `ast` package, using the `from_ast` variant of `fn`, demonstrated on Line 10. The AST here is generated programmatically by calling the `specialize` function, which produces a copy of the syntax tree of the `plus` function with the argument, `b`, eliminated and its uses replaced with a constant, `5`. This transformation as well as some others are distributed in `ace.astx` for convenience.

## 5.2 Background

A neural circuit can be modeled as a network of coupled differential equations, where each node corresponds to a single neuron or a small population. Each node has temporal dynamics given by one or more ordinary differential equations. 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.

The problem we discuss here involved studying a problem that required running up to 1,000 realizations of a network of between 4,000 and 10,000 stochastic 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 a 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 author 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 subsequent data analysis and visualization was conducted using Python, so marshalling the relevant data between processes also became an issue. Doing so during a simulation was not attempted.

## 5.3 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 Ace. 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[2]

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

**Listing 10** `[listing10.py]` An example of a nested simulation tree, showing that specifying a simulation is both simple and modular. The first argument to the constructor specifies each node's parent.

```
1  sim = Simulation(None, n_timesteps=10000)
2  neurons = ReducedLIF(sim, count=N, tau=20.0)
3  e_synapse = ExponentialSynapse(neurons, 'ge',
4      tau=5.0, reversal=60.0)
5  probe = StateVariableProbeCopyback(e_synapse)
```

**Listing 11** `[listing11.py]` An example of a hook that inserts code and also inserts new, nested hooks for downstream simulation nodes below that.

```
1  class ReduceLIF(ModelNode):
2    """Base class for spiking neuron models."""
3    def in_model_code(self, g): # g is the code generator
4      "idx_state = idx_model + count*" << g
5      "(realization_n - realization_start)" << g
6      self.insert_hook("read_incoming", g)
7      self.insert_hook("read_state", g)
8      self.insert_hook("calculate_inputs", g)
9      self.insert_hook("state_calculations", g)
10     self.insert_hook("spike_processing", g)
11   # ...
```

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 10 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 by adding a probe model as a child of the synapse model. Interleaved analyses are specified as nodes as well (not shown).

Implementations of these classes do not evaluate the simulation logic directly, but rather contain methods that generate Ace 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 11 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. Moreover, memory allocation is significantly simplified by the availability of typing information at compile-time. The amount of memory needed by a simulation can be calculated

statically and thus distributing the simulation over a cluster simply requires determining the available memory on each device in the cluster.

Once the kernel has been generated and memory has been allocated, the simulation can be executed directly using the bindings above. The results of this simulation are available to the Python code during and following the simulation and can be visualized and further analyzed using standard tools. Once the computations is complete, the garbage collector is able to handle deallocation of GPU memory automatically (a feature of the underlying `pyopencl` library [22].)

Using this Ace-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.

# 6. RELATED WORK

## 6.1 Active Libraries

Libraries that contain compile-time logic have been called *active libraries* in prior proposals [8]. A number of projects, such as Blitz++, have taken advantage of the C++ preprocessor and template-based metaprogramming system to implement domain-specific optimizations [23]. In Ace, we replace these brittle mini-languages with a general-purpose language and significantly expand the notion of active libraries by consideration of types as objects in the metalanguage. We thus call these types *active types*.

## 6.2 Structural Polymorphism

Generic functions represent a novel strategy for achieving *function polymorphism* – the ability to define functions that operate over more than a single type. In Ace, generic functions are implicitly polymorphic and can be called with arguments of *any type that supports the operations used by the function*. This approach is related to structural polymorphism, however [24]. Structural types make explicit the requirements on a function, unlike generic functions.

Structural typing can be compared to the more *ad hoc* approach taken by dynamically-typed languages, sometimes called "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 comparable to the C++ template system, as discussed previously.

## 6.3 Run-Time Indirection

*Operator overloading* [25] and *metaobject dispatch* [26] are run-time protocols that translate operator invocations into function calls. The function is typically selected according to the type or value of one or more operands. These protocols share the notion of *inversion of control* with type-level specification. However, type-level specification is a *compile-time* protocol focused on enabling specialized verification and implementation strategies, rather than simply enabling run-time indirection.

## 6.4 Term Rewriting Systems

Many languages and tools allow developers to rewrite expressions according to custom rules. These can broadly be classified as *term rewriting systems*. Macro systems, such as those characteristic of the LISP family of languages [27], are the most prominent example. Some compile-time metaprogramming systems also allow users to manipulate syntax trees (e.g. MetaML [28]), and external rewrite systems also exist for many languages. These differ in their direct exposure to syntax trees and their difficulties with propagating type information, since it is not directly encoded in the syntax. The AT&T mechanism is a type-based mechanism that avoids these issues.

## 6.5 Language Frameworks and Extensible Compilers

When the mechanisms available in an existing language prove insufficient, researchers and domain experts often design a new language. A number of tools have been developed to assist with this task, including compiler generators, language workbenches and domain-specific language frameworks (cf [29]). Extensible compilers can be considered a form of language framework as well due to portability issues that using compiler extensions can introduce. It is difficult or impossible for these language-external approaches to achieve interoperability, as discussed above.

## 6.6 Extensible Languages

Extensible languages like SugarJ [30] afford some of the extensibility benefits of Ace, but are not targeted toward HPC. They have largely focused on syntactic extensibility, while Ace relies on a fixed syntax and emphasizes semantic extensibility. They also generally allow users to extend languages globally, which leads to conflicts when multiple extensions are used. AT&T does not admit such conflicts.

# 7. CONCLUSION

Professional end-users demand much from new languages and abstractions. In this paper, we began by generating a concrete set of design and adoption criteria that we hope will be of interest and utility to the research community. Based on these constraints, we designed a new language, Ace, making several pragmatic design decisions and introducing several novel techniques, including type propagation via generic functions, extensible type inference, active typechecking and translation and type-aware Python-Ace-OpenCL bindings to uniquely satisfy many of the criteria we discussed, particularly the three criteria that are typically overlooked in other languages. We validated the extension mechanism with a mature implementation of the entirety of the OpenCL type system, as well as outlined a number of other use cases. 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.

Ace has some limitations at the moment. Debugging is only supported on the generated code, so if code generation introduces significant complexity, this can be an issue. The OpenCL library we have implemented is a reasonably straightforward internalization of OpenCL itself, however, so debugging has not been a problem thusfar. We believe that active types can be useful to control debugging, and plan to explore this in the future. We will also further explore the use cases and case studies that we have described to validate the design we propose here. We hope that Ace will be developed further by the community to strengthen the foundations upon which new abstractions are implemented and deployed into the HPC professional end-user community.

## 8. 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] K. O. W. Group *et al.*, "The opencl specification, version 1.1, 2010," *Document Revision*, vol. 44.

[8] 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

[9] J. Cordy, "Hints on the design of user interface language features: lessons from the design of turing," in *Languages for developing user interfaces*. AK Peters, Ltd., 1992, pp. 329–340.

[10] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[11] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 499–504.

[12] 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.

[13] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[14] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[15] D. Bonachea, "Gasnet specification, v1." *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.

[16] L. V. Kale and S. Krishnan, *CHARM++: a portable concurrent object oriented system based on C++*. ACM, 1993, vol. 28, no. 10.

[17] L. V. Kale and G. Zheng, "Charm++ and ampi: Adaptive runtime strategies via migratable objects," *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pp. 265–282, 2009.

[18] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. ACM, 2011, pp. 47–56.

[19] 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.

[20] 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

[21] M. F. Sanner *et al.*, "Python: a programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.

[22] 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.

[23] T. L. Veldhuizen, "Blitz++: The library that thinks it is a compiler," in *Advances in Software tools for scientific computing*. Springer, 2000, pp. 57–87.

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

[25] A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, "Revised report on the algorithmic language algol 68," *Acta Informatica*, vol. 5, pp. 1–236, 1975.

[26] G. Kiczales, J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*. Cambridge, MA: MIT Press, 1991.

[27] J. McCarthy, "History of lisp," in *History of programming languages I*. ACM, 1978, pp. 173–185.

[28] T. Sheard, "Using MetaML: A staged programming language," *Lecture Notes in Computer Science*, vol. 1608, pp. 207–??, 1999.

[29] M. Fowler and R. Parsons, *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[30] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, "Sugarj: Library-based syntactic language extensibility," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 391–406, 2011.