Numba MVP 2023

Numba is currently a large, single code base, we’re trying to break this into reusable components as a “compiler toolkit” with future Numba just being an instance/configuration of this toolkit.
Two parts to today’s talk:

1. PIXIE
2. RVSDG and NIL
What impacts the consumption format of software?

Primary decision: AOT or JIT?

If AOT, which ISA? If JIT, handle dependency chain.

Little choice about how to consume!
What does the compilation-execution landscape of packaged software look like (in Python)?

Better runtime performance

AOT   AOT + ISA dispatch   JIT   JIT + IPO

Better execution latency
What does the compilation-execution landscape of packaged software look like (in Python)?

Better runtime performance

AOT
AOT + ISA dispatch
JIT
JIT + IPO

Better execution latency

Cython/Custom C-extensions

e.g. NumPy (NEP-038)

@numba.jit users

?! (Julia)
Numba toolkit component: PIXIE

Portable Instructions eXchanged In Executable (PIXIE) project.

PIXIE:

○ Is part of the Numba project’s R&D for 2023.
○ Is a cornerstone project in the Numba compiler-toolkit
○ Acts as a “target” for new compiler technology as its output format is designed to put the choice about where and when to compile into the hands of the user.

The PIXIE project provides a set of tools to create PIXIE libraries

● It is a pure Python package.
● It depends on llvmlite.
● Main entry point is the PIXIE-compiler API
  ○ It takes LLVM IR/bitcode as input (along with PIXIE specific export instructions if desired)
  ○ It produces a PIXIE library.
What is a PIXIE library?

- A library created by the PIXIE tools!
- They are platform native libraries (e.g. ELF) compiled using the LLVM toolchain, linked using native tooling.
- They are often Python C-extensions (but don't have to be)
  - `>>> import my_PIXIE_module # just works!`
- Can contain additional information for use in JIT compilation.
  - The LLVM bitcode used as input to the library.
  - Numba Intermediate Language (NIL) version of the functions in the library.
  - A Python “overlay” (more about this on next slide)
- Can contain “feature set” versioned symbols along with symbols that will dispatch to them. The PIXIE-compiler helps users create these.

- Can contain a python `.specialize()` function to permit trivial recompilation to the host architecture. The symbol dispatchers and Python “overlay” are both aware of the specialised library.
PIXIE Python overlay

```python
>>> import my_PIXIE_module
>>> my_PIXIE_module.__PIXIE__ # nested dictionaries of useful things.

{'bitcode': <the bitcode as bytes!>,
 'c_header': <A C-header file to #include to use this library elsewhere>,
 'linkage': <external library on which this library depends, e.g. ['m', 'blas']>,
 'NIL': <Numba intermediate language repr of this library>,
 'symbols': <expanded later>,}
```

Symbols key contents (e.g. foo compiled for sse3 baseline with avx variant):

```python
{'foo': {'void(int *)': {'address': <the runtime address>,
 'baseline_feature': 'sse3',
 'cfunc': <ctype.CFUNCTYPE callable>,
 'ctypes_cfunctype': <ctypes.CFUNCTYPE instance>,
 'feature_variants': {'+avx': {address: <runtime address>,
                               cfunc:<ctype.CFUNCTYPE callable>,
                               symbol: '_Z3fooB4+avxPi'}},
 'source_file': 'path/to/foo_source.c',
 'symbol': '_Z3fooB5+sse3Pi'}
```
2023 MVPs… cross module optimisation

User code that needs to combine functions from separate packages

E.g. SciPy optimiser

E.g. objective function from domain specific package

```
def foobar(*args):
    return foo(bar, *args)
```

```
def foo(fn, *args):
```

```
def bar(*args):
```
2023 MVPs… cross module optimisation.

User code that needs to combine functions from separate packages

Call library symbol

E.g. SciPy optimiser

AOT - library calls

pixie lib 3

def foobar(*args):
    return foo(bar, *args)

pixie lib 1

def foo(fn, *args)

pixie lib 2

def bar(*args)

E.g. objective function from domain specific package
2023 MVPs... cross module optimisation.

User code that needs to combine functions from separate packages

E.g. SciPy optimiser

E.g. objective function from domain specific package

Inline bitcode

JIT - inline bitcode

pixie lib 1

def foo(fn, *args)

pixie lib 2

def bar(*args)

pixie lib 3

def foobar(*args):
    return foo(bar, *args)
2023 MVPs... blended compilation

- C/C++ integration route
- New AOT compiler
- New output target
- New optimisation capability
Numba++ vision

RVSDG
Why Regionalized Value-State Dependence Graph?

- CPython bytecode
  - unstructured CFG e.g GOTOS
  - Changes quickly due to faster-cpython
- CFG restructuring (recovery) is needed
  - For canonicalization
  - For control-flow aware transformation and analysis

**Algorithms for restructuring of CFG**
RVSDG Simplifies

- 3 node types
  - Linear
  - Switches
  - Tail-Loop
- Acyclic demand-dependence graph
- Regions allow multi-level IRs (dialects)
- Explicit encoding of states
  - See imperative programs via a pure-functional lens

Figure above and details from this RVSDG paper:
Representation without Taxation

- Avoid recomputing for SSA, loop analysis
- Simplifies transformation

Also see paper:

Figure above and details from this RVSDG paper:
The Plan

- Reusable bytecode frontend
- CPython bytecode -> Structured representation -> Numba IL
- Structured representation
  - Not a full fledged RVSDG
  - Only have canonicalized CFG. So a Structured CFG form.
  - Useful for other static analysis and program transformation tools
- Numba IL (NIL) will include other RVSDG properties to simplify the compiler
  - Properties like
    - explicit encoding of states and effects
    - multi-level
PIXIE: additional slides follow…
### Package authors determine how the user can use the package.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Package author</th>
<th>Distribution</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOT</td>
<td>Compilation time</td>
<td>Shared library with no dependency.</td>
<td>Run time</td>
</tr>
</tbody>
</table>

The packager pays the compilation time to generate a library targeting a single baseline ISA (and for each platform!).

The user experiences very little latency before their program starts running compiled code, but the run time is only as good as the baseline ISA.

| JIT   | The packager pays no compilation time. | “Source” with dependency on JIT compiler presence. | Run time |

The user experiences a high latency before their program starts running compiled code as the source has to be JIT compiled. However, the run time is likely faster than in the AOT case as the compiled code is specialised to the user’s hardware.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Packager</th>
<th>Distribution</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIXIE (UST - AOT)</td>
<td><strong>Compilation time</strong></td>
<td>The packager pays the compilation time to generate a library targeting potentially multiple ISAs.</td>
<td>User 1. Wants code to run ASAP.</td>
</tr>
</tbody>
</table>

**Run time**

Execution latency (loading shared library and PIXIE dispatch)

The user gets a better runtime as PIXIE dispatches to variants that closely match the ISA of the machine on which the code is running.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Packager</th>
<th>Distribution</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIXIE (UST - JIT-AOT)</td>
<td><strong>Compilation time</strong> The packager pays the compilation time to generate a library targeting potentially multiple ISAs.</td>
<td>PIXIE library. No hard dependency.</td>
<td>User 2. Wants the code to run with maximum performance.</td>
</tr>
</tbody>
</table>

- **Compilation time**

Calling `.specialize()` on a PIXIE library generates a new library specialised to the current machine that the original PIXIE library is aware of and will dispatch to.

- **Run time**

Execution latency… first run large, subsequent tiny.

The user gets better runtime as PIXIE specializes exactly to the current machine (or some target machine). The specialized libraries are redistributable, this is like portable AOT compilation.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Packager</th>
<th>Distribution</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIXIE (UST - JIT-LT/O)</td>
<td><strong>Compilation time</strong></td>
<td>PIXIE library. No hard dependency.</td>
<td>User 3. Wants the functions in the PIXIE library to take part in the optimisation of their JIT program (whole program optimisation/LTO).</td>
</tr>
</tbody>
</table>

The packager pays the compilation time to generate a library targeting potentially multiple ISAs.

The PIXIE library contains the LLVM bitcode for the functions present in the library. The PIXIE Python overlay makes it trivial to access this bitcode. The users JIT code can then use this as part of whole program optimisation.

The user gets potentially better runtime performance than even that available from using PIXIE `specialize()` compiled versions of the functions because the JIT can “see” across function boundaries.
Numba++ vision

Backup slide: Multi-level dialects
Multi-level Dialect

- Popularized by MLIR
- Divide a compiler into a composition of dialects and their transformations
- Domain specific optimization at the level of the dialects

- Use xDSL (https://xdsl.dev/)
  - pure-Python
  - Avoid C++/tablegen

Project partners

School of Informatics (University of Edinburgh)

The School of Informatics leads this project, with Dr Tobias Grosser (the project PI) and Dr Michel Steuwer who are compiler, MLIR and LLVM experts. Being active collaborators to the MLIR and LLVM communities, they bring expertise around the development of appropriate MLIR dialects, the mechanism to connect Python based DSLs with MLIR, and integration with LLVM. Furthermore, their wide connections throughout the compiler communities means that the project is able to engage closely with that community and act as a bridge between people working in HPC and compilers.

Department of Computing (Imperial College London)

Professor Paul Kelly and Dr Gerard Gorman provide DSL expertise to the project. Leading the development of numerous DSLs, including the popular and widely successful Devito, their knowledge and experience is invaluable to ensure that the ecosystem being developed is wide ranging across DSLs. Furthermore, Devito is an explicit target of this project, using the Python-based MLIR integration toolbox that we will develop to integrate this DSL with our ecosystem. Additionally Paul and Gerard bring significant experience of working with seismic community, cloud computing, and interactive HPC technologies such as Jupyter notebooks, all of which are important considerations in this project to ensure that the DSL eco-system we will develop can interoperate with.

EPCC (University of Edinburgh)

EPCC is the UK’s leading supercomputing centre, with Dr Nick Brown and Dr Amyra Krause providing HPC expertise at both the technology and application level. This includes expertise with a number of the project’s target applications that we will be using as vehicles to evaluate the developed eco-system. During the project we will be using many of EPCC’s hosted HPC systems
Every library as a dialect

- API is **Syntax** *(structure)*
  - `def foo(A) -> B`
  - `def bar(B) -> C`

- Compiler = turns **Syntax** into **Semantic**

- Dialect is a mini-language
  - Comes with its own dialect-level transformation

```
Syntax -> grammar
Semantic -> meaning
Grammatically correct != meaningful
“Colorless green ideas sleep furiously” (Noam Chomsky)
numpy.prod(numpy.arange(10))
```

```
Library implementation provides semantic per library operation. Compiler provides semantic base on the entire program's use of the library API.
```
Composition of dialects

- provide tools to empower library writers to convert APIs into dialects.
- stop being the bottleneck in implementing library support
Multiple pathways to semantic

Compilation is a **structure-preserving** transformation that converts syntax to semantic.

There are **unobserved structures**! API spec is sparse.
High-level semantic simplifies optimization

```python
import numpy as np
from numba import njit

@njit
def base(nelem: int) -> int:
    arr = np.arange(nelem)
    return np.add(arr, arr).sum()

@njit
def opt_avoid_add(nelem: int) -> int:
    arr = np.empty(nelem, dtype=np.intp)
    for i in range(arr.size):
        arr[i] = i * 2
    return arr.sum()

@njit
def opt_avoid_sum(nelem: int) -> int:
    c = 0
    for i in range(nelem):
        c += i * 2
    return c

@njit
def opt_avoid_loop(nelem: int) -> int:
    return nelem * (nelem - 1)
```
With help from the community

- Divide the work of compiler engineering
  - Avoid giant monolithic compiler
- Allow domain experts to provide optimization at the level of each dialect
  - Don’t have to wait for the compiler engineer to learn <insert complicated topic> to write optimization passes
More than the sum of its parts

- The community is not just helping Numba to build a compiler
- Dialect serve as a spec for alternative implementations
  - NumPy is becoming the Python Array DSL already; cupy, pytorch, jax
- Bring compiler technology to the level of libraries enables tricky features
  - operation fusion
  - cross library optimization
  - auto conversion to distributed code
  - autodiff