Out-Performing NumPy is Hard: When and How to Try with Your Own C-Extensions

Christopher Ariza
CTO, Research Affiliates

About Me

CTO at Research Affiliates

Python programmer since 2000

PhD in music composition, professor of music technology

Python for algorithmic composition, computational musicology

Since 2012, builder of financial systems in Python

Creator of StaticFrame, an alternative DataFrame library

A passion for Python performance

Python Performance

- Python is relatively slow
  - All values are "boxed" in C `PyObjects`
  - Values not in contiguous memory
  - Must manage reference counts
- C-extensions using C-types are fast
- With NumPy, we get C-typed arrays in Python

Can NumPy Routines be Optimized?

1. Some NumPy routines are implemented in Python
2. Many NumPy routines do more than we need

Optimizing NumPy Routines written in Python

- Some NumPy routines are implemented in Python
  - `np.roll()`
  - `np.linspace()`
- All leverage lower-level C routines
- Little chance a C-implementation will be faster

Optimizing Excessively Flexible NumPy Routines

- NumPy routines are flexible
  - Handle non-array (i.e., list, tuple) inputs
  - Handle N-dimensional arrays
  - Handle full diversity of dtypes
  - Support diverse array memory layouts (non-contiguous memory)
- Flexibility has a performance cost
- More narrow routines might be more efficient

Finding the First `True`

A utility that was needed for StaticFrame

1D Boolean array: find the index of the first `True`

2D Boolean array: find the indices of the first `True` per axis

Search in both directions

Identify all `False`

File Edit View History Bookmarks Tools Help

python - Numpy first occ

https://stackoverflow.com/questions/16243955/numpy-first-occurrence-of-value-greater-than-existing-value

stackoverflow Products Search... Log in Sign up

Home

PUBLIC

Questions

Tags

Users

Companies

COLLECTIVES

Explore Collectives

TEAMS

Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.

Free

Numpy first occurrence of value greater than existing value

Ask Question

Asked 10 years, 1 month ago Modified 1 year, 8 months ago Viewed 249k times

216

I have a 1D array in numpy and I want to find the position of the index where a value exceeds the value in numpy array.

E.g.

```
aa = range(-10,10)
```

Find position in `aa` where, the value `5` gets exceeded.

python numpy

Share Improve this question Follow

edited Apr 25, 2017 at 17:03
Cœur
36.9k • 25 • 193 • 262

asked Apr 26, 2013 at 19:35
user308827
20.7k • 84 • 252 • 410

File Edit View History Bookmarks Tools Help

python - How to obtain o

https://stackoverflow.com/questions/54155844/how-to-obtain-only-the-first-true-value-from-each-row-of-a-numpy-array

stackoverflow Products Search... Log in Sign up

Home

PUBLIC

Questions

Tags

Users

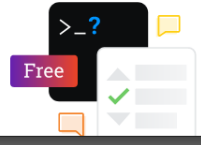
Companies

COLLECTIVES

Explore Collectives

TEAMS

Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.



How to obtain only the first True value from each row of a numpy array?

Ask Question

Asked 4 years, 4 months ago Modified 4 years, 4 months ago Viewed 5k times

8

I have a 4x3 boolean numpy array, and I'm trying to return a same-sized array which is all False, except for the location of the first True value on each row of the original. So if I have a starting array of

```
all_bools = np.array([[False, True, True],[True, True, True],[False, False, True],[Fal
all_bools
array([[False,  True,  True], # First true value = index 1
       [ True,  True,  True], # First true value = index 0
       [False, False,  True], # First true value = index 2
       [False, False, False]]) # No True Values
```

then I'd like to return

```
[[False, True, False],
 [True, False, False],
 [False, False, True],
 [False, False, False]]
```

so indices 1, 0 and 2 on the first three rows have been set to True and nothing else. Essentially any

File Edit View History Bookmarks Tools Help

first nonzero element (Tr

https://github.com/numpy/numpy/issues/2269 150%

numpy / numpy Public

Sponsor | Watch 588 | Fork 8.2k | Starred 23.7k

Code | Issues 2k | Pull requests 167 | Actions | Projects 10 | Wiki | Security | Insights

first nonzero element (Trac #1673) #2269

New issue

Open numpy-gitbot opened this issue on Oct 19, 2012 · 33 comments

numpy-gitbot commented on Oct 19, 2012

Original ticket http://projects.scipy.org/numpy/ticket/1673 on 2010-11-13 by trac user tom3118, assigned to unknown.

The "numpy for matlab users" suggests using
`nonzero(A)[0][0]`
to find the index of the first nonzero element of array A.

The problem with this is that A might be a million elements long and the first element might be zero.

This is an extremely common operation. An efficient, built-in method for this would be very useful. It also would easy people's transition from Matlab in which `find` is so common.

82

numpy-gitbot commented on Oct 22, 2012 Author

Assignees
No one assigned

Labels
01 - Enhancement | component: Other

Projects
None yet

Milestone
NumPy 2.0

Development
No branches or pull requests

Finding the First `True` with NumPy

- No NumPy function does just what we need
- Two options are close
  - `np.argmax()`
  - `np.nonzero()`

`np.argmax()`

Return the index of the maximum value in an array

If there are ties, the first index is returned

Specialized for Boolean arrays to short-circuit on first `True`

All `False` returns an ambigous `0`

Must call `np.any()` to discover all `False`

np.argmax()

```
>>> array = np.array([False, False, True, False, False])
>>> np.argmax(array) # finds first True
2
>>> np.argmax(np.array([False, False])) # if all False, reports 0
0
>>> np.argmax(np.array([True, False]))
0
```

`np.nonzero()`

Finds all non-zero positions

Returns a tuple of arrays per dimension

Cannot short-circuit

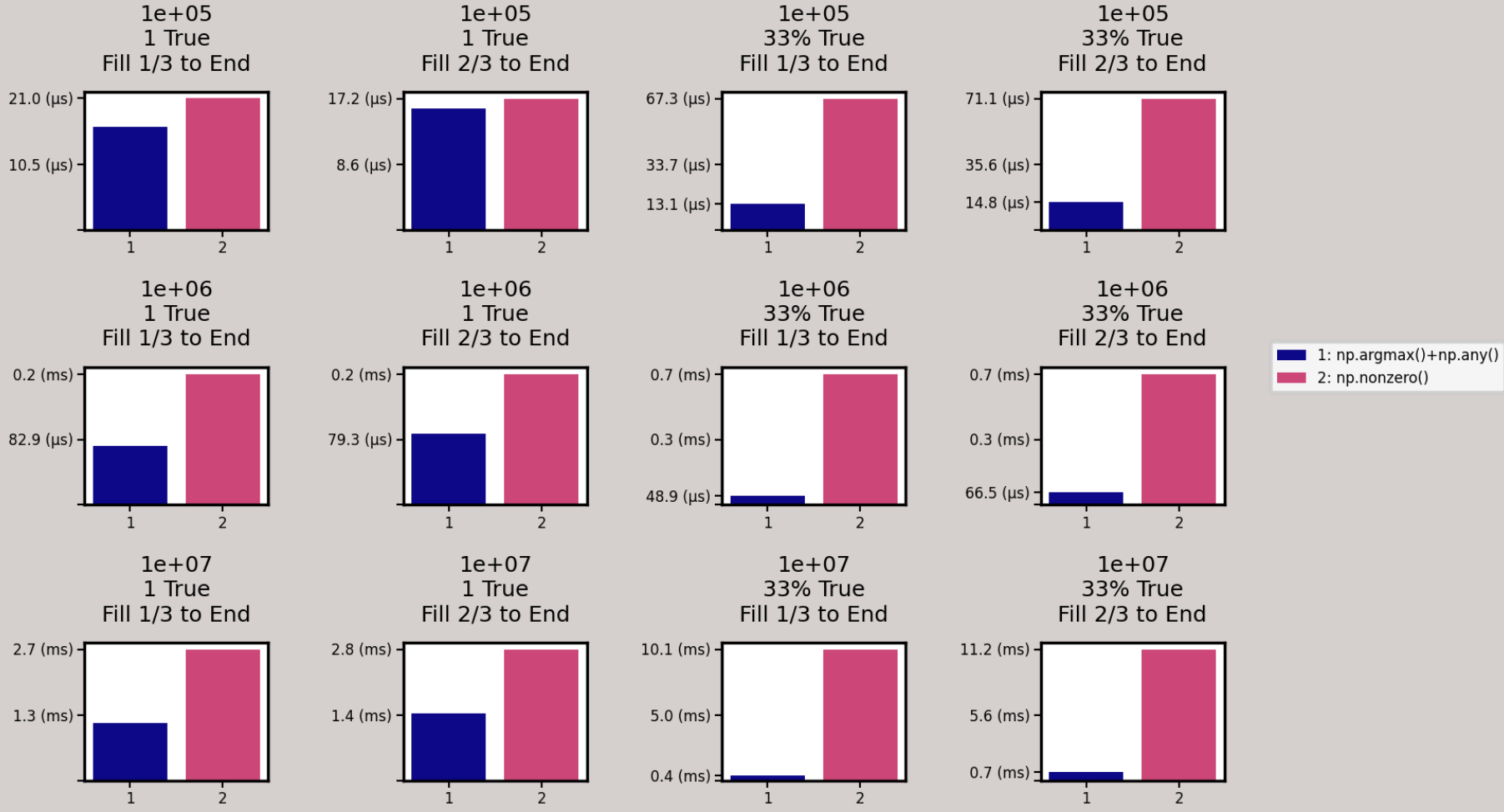
np.nonzero()

```
>>> array = np.array([False, False, True, False, False])
>>> np.nonzero(array)
(array([2]),)
>>> np.nonzero(array)[0][0]
2
>>> array = np.array([False, True, False, True, True])
>>> np.nonzero(array)
(array([1, 3, 4]),)
```

Performance of `np.argmax()` + `np.any()` & `np.nonzero()`

np.argmax() + np.any() & np.nonzero() Performance

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000

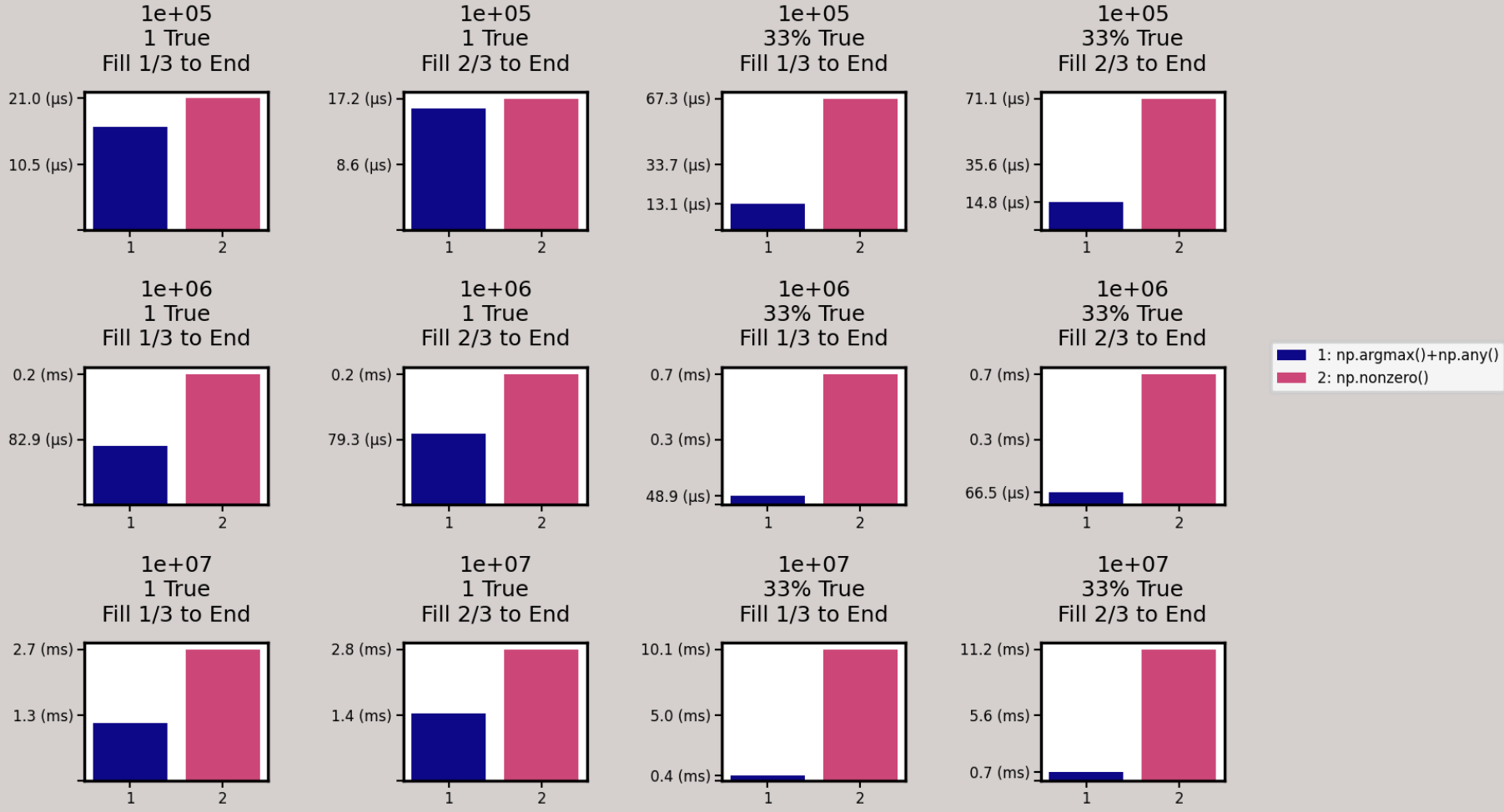


Performance Panels

- Three rows: size of 1D array (1e5, 1e6, 1e7)
- Four columns: different fill characteristics
  - One `True`
    - Set at 1/3rd to the end
    - Set at 2/3rd to the end
  - 33% of size is `True`
    - Filled from 1/3rd to the end
    - Filled from 2/3rd to the end

np.argmax() + np.any() & np.nonzero() Performance

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



Opportunities for Improvement

- `np.argmax()`
  - Must call `np.any()` to discover all-`False`
  - Worst case requires two iterations, but can short-circuit
  - Does not search in reverse
- `np.nonzero()`
  - Requires one full iteration (cannot short-circuit)
  - Collects more than we need
  - Must iterate over results to find first or last

Many Options for Performance

C implementation

Cython / Numba

Rust via PyO3

GPU

Good Candidates for C Implementation

Core routine can be done without `PyObjects`

Can operate directly on a C array

`first_true_1d()` as a C Extension

- A function with two arguments
  - NumPy array
  - A Boolean (`True` for forward, `False` for reverse)
- Evaluate each element, return the index of the first `True`
- If no `True`, return `-1`
- Code: https://github.com/flexatone/np-bench

Defining a C extension

A Minimal C Extension Module `np_bench`

```c
// ... include Python.h, numpy, etc.
static struct PyModuleDef npb_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "np_bench",
    .m_size = -1,
};
PyObject*
PyInit_np_bench(void)
{
    import_array();
    PyObject *m = PyModule_Create(&npb_module);
    if (!m || PyModule_AddStringConstant(m, "__version__", "0.1.0")
    ) {
        Py_XDECREF(m);
        return NULL;
    }
    return m;
}
```

A C Function as a Module-Level Python Function

```c
static PyObject*
first_true_1d(PyObject *Py_UNUSED(m), PyObject *args)
{
    PyArrayObject *array = NULL;
    int forward = 1;
    if (!PyArg_ParseTuple(args,
            "O!p:first_true_1d",
            &PyArray_Type, &array,
            &forward)) {
        return NULL;
    }
    // implmentation
    return PyLong_FromSsize_t(-1);
}
```

Adding a C Function to a Python Module

```c
static PyMethodDef npb_methods[] =  {
    {"first_true_1d", (PyCFunction)first_true_1d, METH_VARARGS, NULL},
    {NULL},
};
static struct PyModuleDef npb_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "np_bench",
    .m_size = -1,
    .m_methods = npb_methods,
};
```

Reading elements from an array in C

Four Ways to Read Elements

I. Reading `PyObjects` From Arrays (`PyArray_GETITEM`)

II. Casting Data Pointers to C-Types (`PyArray_GETPTR1`)

III. Using `NpyIter`

IV. Using C-Arrays and Pointer Arithmetic (`PyArray_DATA()`)

Working with `PyObjects`

I: Reading `PyObjects` From Arrays

Only process 1D arrays

Use `PyArray_GETPTR1()` to get pointer to element

Use `PyArray_GETITEM()` to build corresponding `PyObject`

Use Python C-API `PyObject_IsTrue()` to evaluate element

Must manage reference counts for `PyObjects`

I: Reading `PyObjects` From Arrays

```c
static PyObject*
first_true_1d_getitem(PyObject *Py_UNUSED(m), PyObject *args)
{
    // ... parse args
    if (PyArray_NDIM(array) != 1) {
        PyErr_SetString(PyExc_ValueError, "Array must be 1-dimensional");
        return NULL;
    }
    // ... implementation
}
```

I: Reading `PyObjects` From Arrays

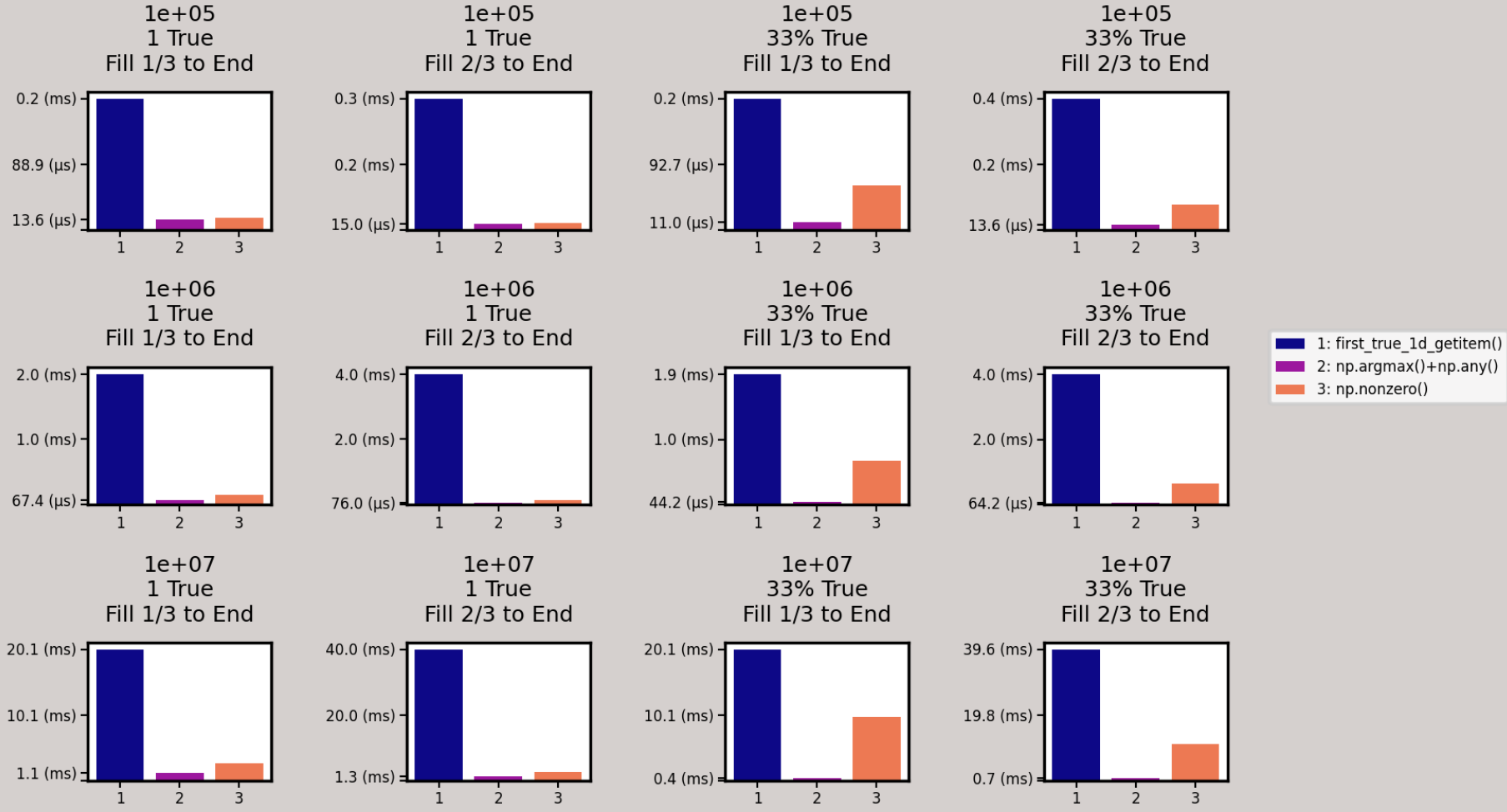
```
npy_intp size = PyArray_SIZE(array);
npy_intp i;
PyObject* element;
if (forward) {
    for (i = 0; i < size; i++) {
        element = PyArray_GETITEM(array, PyArray_GETPTR1(array, i));
        if(PyObject_IsTrue(element)) {
            Py_DECREF(element);
            break;
        }
        Py_DECREF(element);
    }
}
```

I: Reading `PyObjects` From Arrays

```
else { // reverse
    for (i = size - 1; i >= 0; i--) {
        element = PyArray_GETITEM(array, PyArray_GETPTR1(array, i));
        if(PyObject_IsTrue(element)) {
            Py_DECREF(element);
            break;
        }
        Py_DECREF(element);
    }
}
if (i < 0 || i >= size ) { // did not break
    i = -1;
}
return PyLong_FromSsize_t(i);
```

first_true_1d() Performance with PyArray_GETITEM()

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



Using C types instead of `PyObjects`

II: Casting Data Pointers to C-Types

Only process 1D, *Boolean* arrays

Use `PyArray_GETPTR1()` and cast to C type

No use of Python C-API, no reference counting

Can release the GIL over core loop

II: Casting Data Pointers to C-Types

```c
static PyObject*
first_true_1d_getptr(PyObject *Py_UNUSED(m), PyObject *args)
{
    // ... parse args
    if (PyArray_NDIM(array) != 1) {
        PyErr_SetString(PyExc_ValueError, "Array must be 1-dimensional");
        return NULL;
    }
    if (PyArray_TYPE(array) != NPY_BOOL) {
        PyErr_SetString(PyExc_ValueError, "Array must be of type bool");
        return NULL;
    }
    // ... implementation
}
```

II: Casting Data Pointers to C-Types

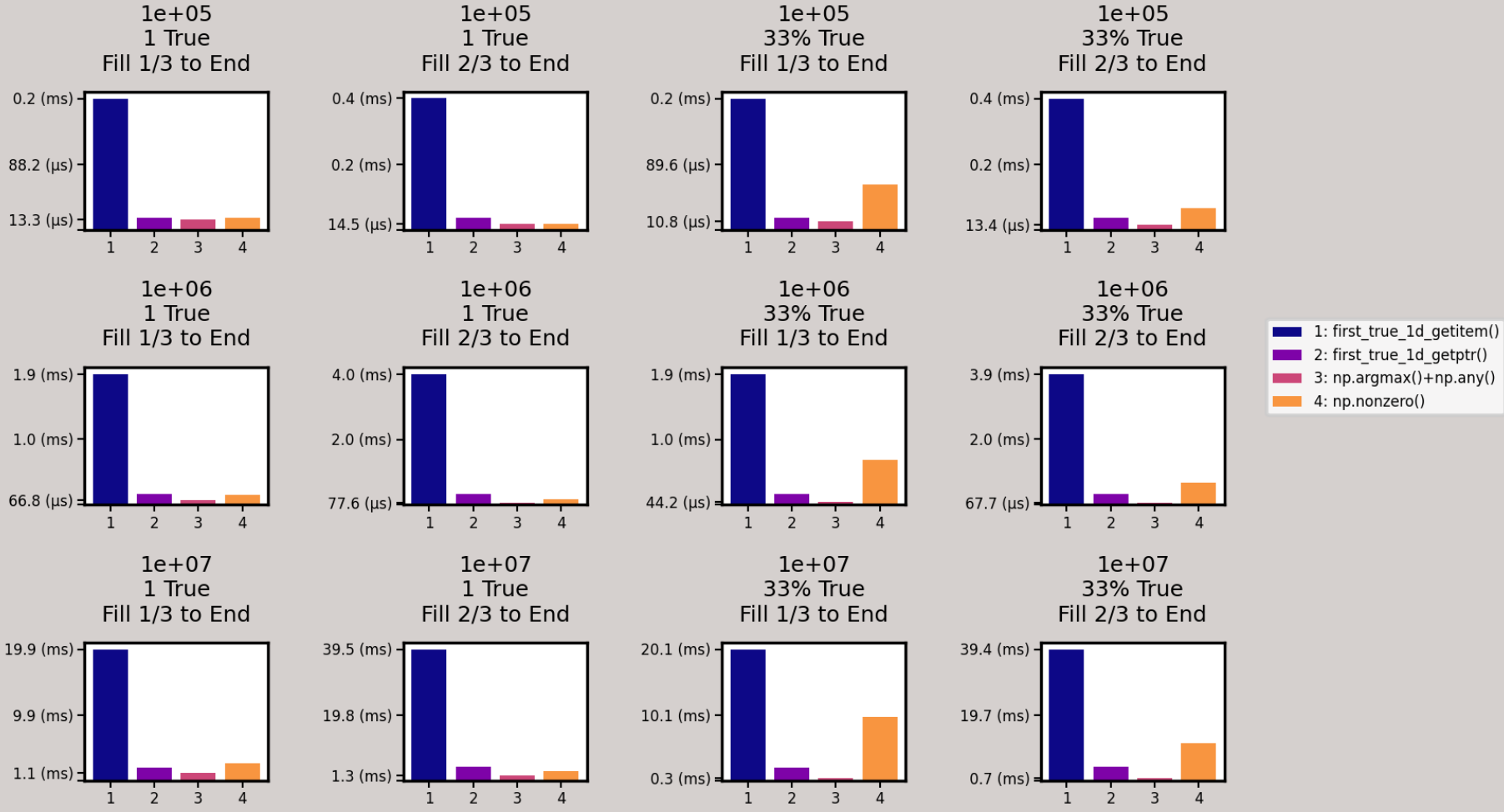
```
npy_intp size = PyArray_SIZE(array);
npy_intp i;

if (forward) {
    for (i = 0; i < size; i++) {
        if(*(npy_bool*)PyArray_GETPTR1(array, i)) {
            break;
        }
    }
}
```

II: Casting Data Pointers to C-Types

```
else { // reverse
    for (i = size - 1; i >= 0; i--) {
        if(*(npy_bool*)PyArray_GETPTR1(array, i)) {
            break;
        }
    }
}
if (i < 0 || i >= size ) { // did not break
    i = -1;
}
return PyLong_FromSsize_t(i);
```

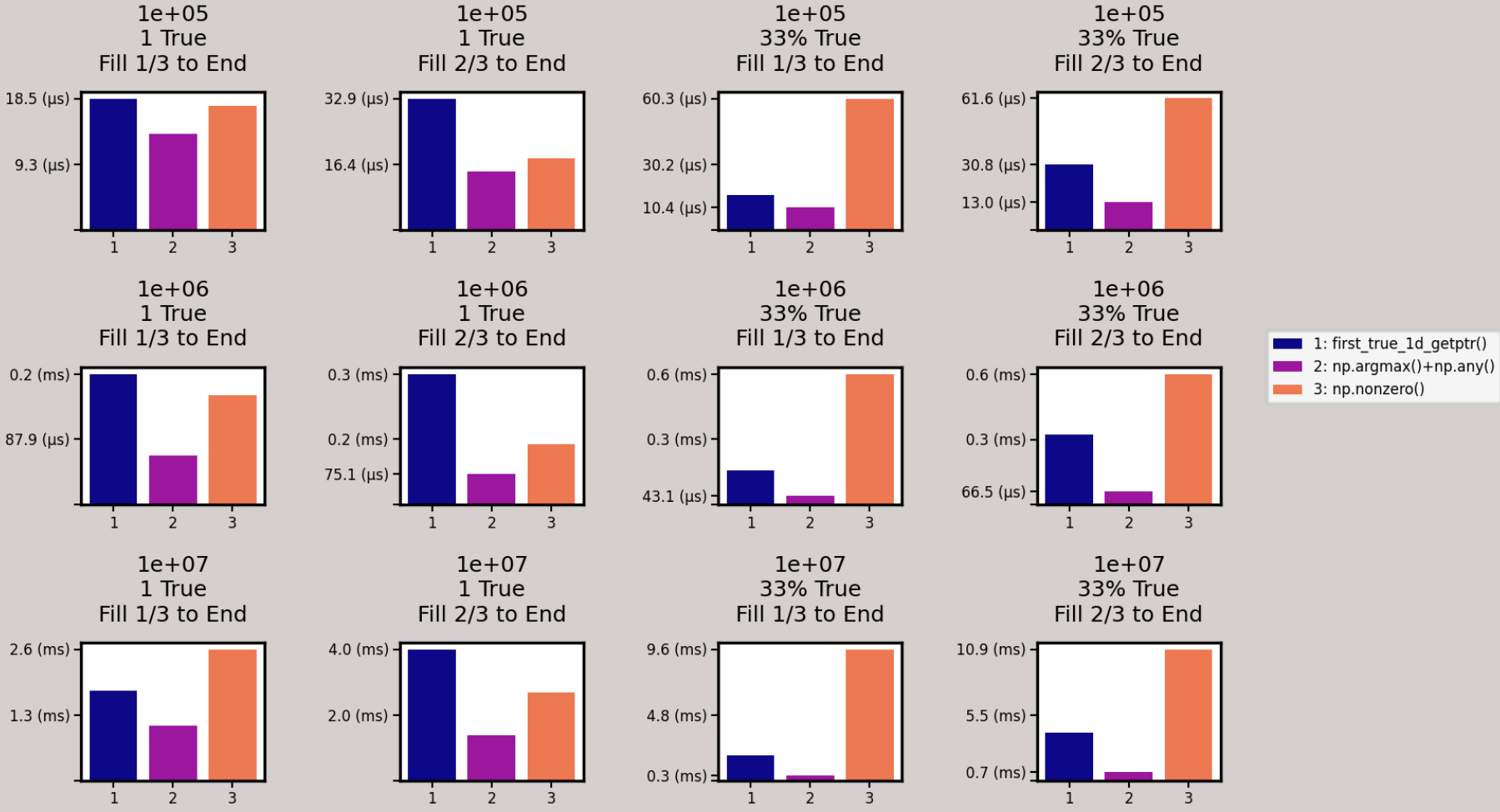
first_true_1d() Performance with PyArray_GETPTR1()

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



first_true_1d() Performance with PyArray_GETPTR1()

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



Other options within the NumPy C API

III: Using `NpyIter`

`NpyIter` provides common iteration interface in C

Supports all dimensionalities, dtypes, memory layouts

Performs stride-sized pointer arithmetic in inner loop

Requires more code

Does not support reverse iteration

III: Using `NpyIter`

```c
static PyObject*
first_true_1d_npyiter(PyObject *Py_UNUSED(m), PyObject *args)
{
    // ... parse args
    if (PyArray_NDIM(array) != 1) {
        PyErr_SetString(PyExc_ValueError, "Array must be 1-dimensional");
        return NULL;
    }
    if (PyArray_TYPE(array) != NPY_BOOL) {
        PyErr_SetString(PyExc_ValueError, "Array must be of type bool");
        return NULL;
    }
    // ... implementation
}
```

III: Using `NpyIter`

```
NpyIter *iter = NpyIter_New(
        array,                                      // array
        NPY_ITER_READONLY | NPY_ITER_EXTERNAL_LOOP, // iter flags
        NPY_KEEPORDER,                              // order
        NPY_NO_CASTING,                             // casting
        NULL                                        // dtype
        );
if (iter == NULL) {
    return NULL;
}
NpyIter_IterNextFunc *iter_next = NpyIter_GetIterNext(iter, NULL);
if (iter_next == NULL) {
    NpyIter_Deallocate(iter);
    return NULL;
}
```

III: Using `NpyIter`

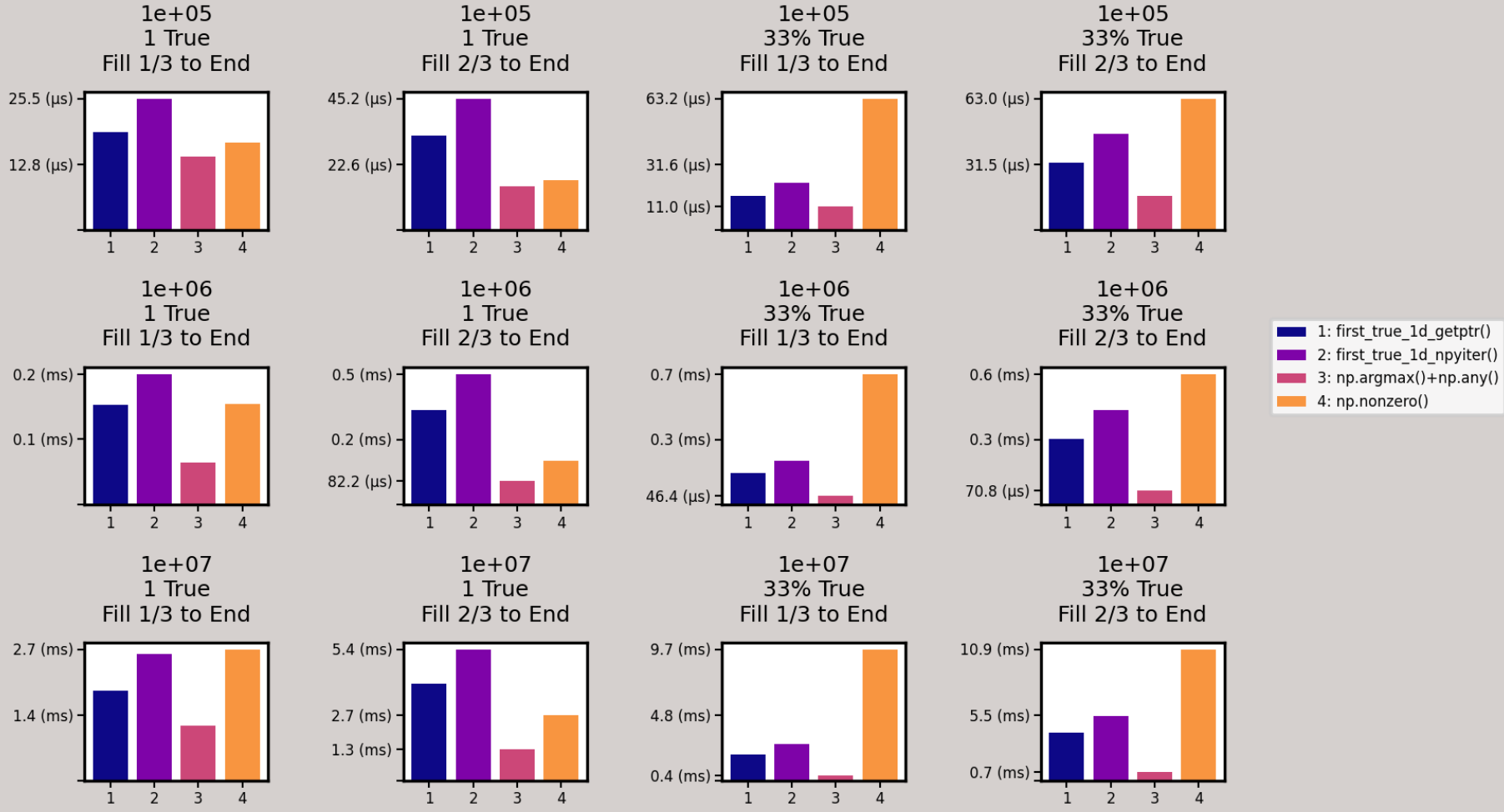
```
npy_bool **data_ptr_array = (npy_bool**)NpyIter_GetDataPtrArray(iter);
npy_bool *data_ptr;

npy_intp *stride_ptr = NpyIter_GetInnerStrideArray(iter);
npy_intp stride;

npy_intp *inner_size_ptr = NpyIter_GetInnerLoopSizePtr(iter);
npy_intp inner_size;

npy_intp i = 0;
```

III: Using `NpyIter`

```
    do {
        data_ptr = *data_ptr_array;
        stride = *stride_ptr;
        inner_size = *inner_size_ptr;
        while (inner_size--) {
            if (*data_ptr) {
                goto exit;
            }
            i++;
            data_ptr += stride;
        }
    } while(iter_next(iter));
    if (i == PyArray_SIZE(array)) {
        i = -1;
    }
exit:
    NpyIter_Deallocate(iter);
    return PyLong_FromSsize_t(i);
```

first_true_1d() Performance with NpyIter

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



Assuming array contiguity...

IV(a.): Using C-Arrays and Pointer Arithmetic

Only process 1D, Boolean, and *contiguous* arrays

Use `PyArray_DATA()` to get pointer to underlying C-array

Advance through array with pointer arithmetic

IV(a.): Using C-Arrays and Pointer Arithmetic

```c
static PyObject*
first_true_1d_ptr(PyObject *Py_UNUSED(m), PyObject *args)
{
    // ... parse args
    if (PyArray_NDIM(array) != 1) {
        PyErr_SetString(PyExc_ValueError, "Array must be 1-dimensional");
        return NULL;
    }
    if (PyArray_TYPE(array) != NPY_BOOL) {
        PyErr_SetString(PyExc_ValueError, "Array must be of type bool");
        return NULL;
    }
    if (!PyArray_IS_C_CONTIGUOUS(array)) {
        PyErr_SetString(PyExc_ValueError, "Array must be contiguous");
        return NULL;
    }
    // ... implementation
}
```

IV(a.): Using C-Arrays and Pointer Arithmetic

```c
npy_bool *array_buffer = (npy_bool*)PyArray_DATA(array);

npy_intp size = PyArray_SIZE(array);
Py_ssize_t i = -1;
npy_bool *p;
npy_bool *p_end;

if (forward) {
    p = array_buffer;
    p_end = p + size;
    while (p < p_end) {
        if (*p) {
            break;
        }
        p++;
    }
}
```

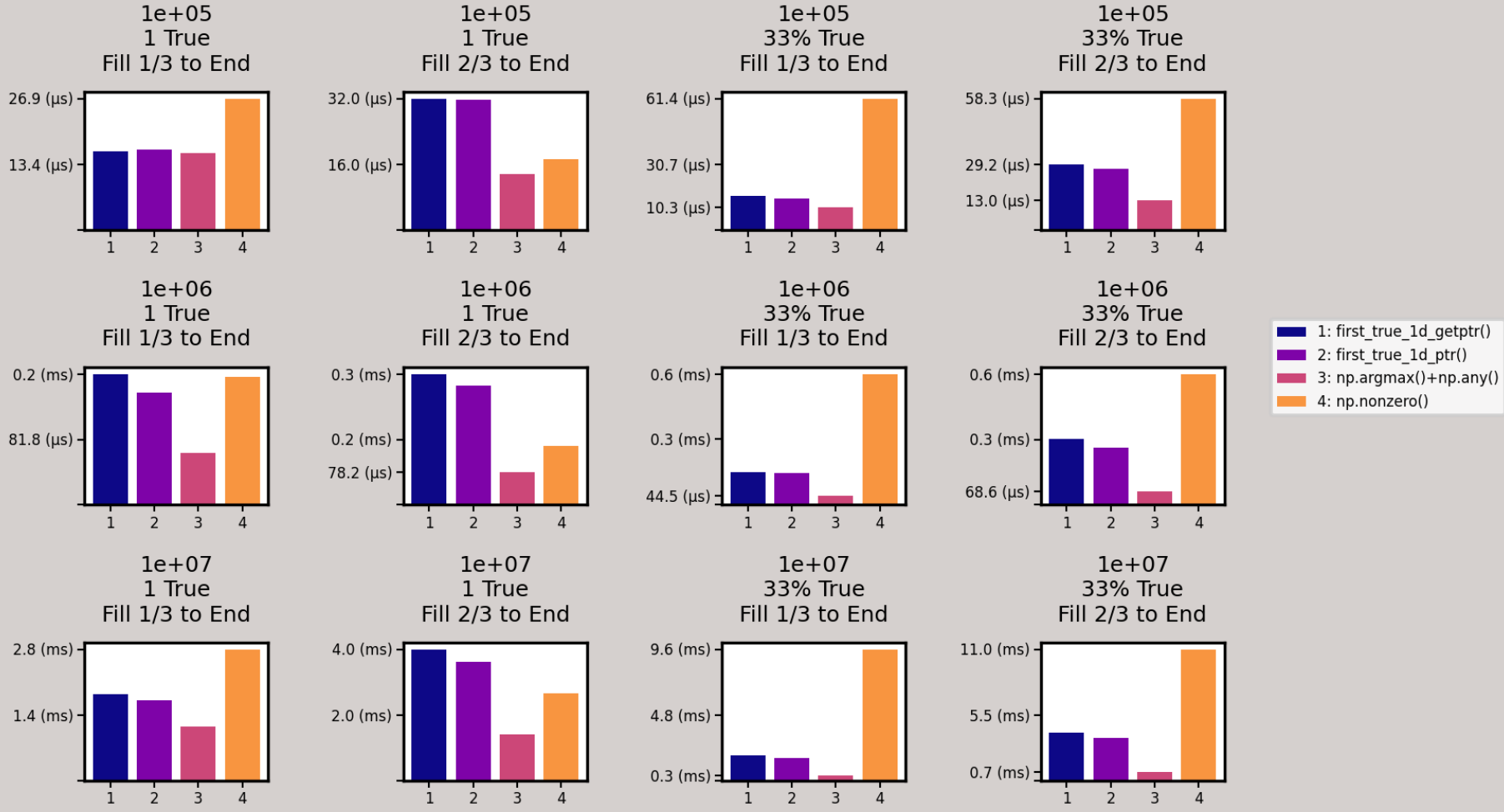
IV(a.): Using C-Arrays and Pointer Arithmetic

```
else { // reverse
    p = array_buffer + size - 1;
    p_end = array_buffer - 1;
    while (p > p_end) {
        if (*p) {
            break;
        }
        p--;
    }
}
if (p != p_end) { // if break encountered
    i = p - array_buffer;
}
return PyLong_FromSsize_t(i);
```

This must be the fastest approach...

first_true_1d() Performance with PyArray_DATA()

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



How is `np.argmax()` + `np.any()` still faster?

Performance Beyond Contiguous Iteration

Single instruction, multiple data (SIMD) instructions

SSE SIMD on x86-64 has 128 bit registers

16 1-byte Booleans can be processed in one instruction

AVX-512 permits processing 512 bit registers

True vectorization

NumPy SIMD `BOOL_argmax`

```
NPY_NO_EXPORT int NPY_CPU_DISPATCH_CURFX(BOOL_argmax)
(npy_bool *ip, npy_intp len, npy_intp *mindx, PyArrayObject *NPY_UNUSED(aip))
{
    npy_intp i = 0;
    const npyv_u8 zero = npyv_zero_u8();
    const int vstep = npyv_nlanes_u8;
    const int wstep = vstep * 4;
    for (npy_intp n = len & -wstep; i < n; i += wstep) {
        npyv_u8 a = npyv_load_u8(ip + i + vstep*0);
        npyv_u8 b = npyv_load_u8(ip + i + vstep*1);
        npyv_u8 c = npyv_load_u8(ip + i + vstep*2);
        npyv_u8 d = npyv_load_u8(ip + i + vstep*3);
        npyv_b8 m_a = npyv_cmpeq_u8(a, zero);
        npyv_b8 m_b = npyv_cmpeq_u8(b, zero);
        npyv_b8 m_c = npyv_cmpeq_u8(c, zero);
        npyv_b8 m_d = npyv_cmpeq_u8(d, zero);
        npyv_b8 m_ab = npyv_and_b8(m_a, m_b);
        npyv_b8 m_cd = npyv_and_b8(m_c, m_d);
        npy_uint64 m = npyv_tobits_b8(npyv_and_b8(m_ab, m_cd));
        if ((npy_int64)m != ((1LL << vstep) - 1)) { // a non-zero found
            break;
        }
    }
    // ... element-wise evaluation from current i to the end
}
```

NumPy SIMD `BOOL_argmax`

```
npyv_u8 a = npyv_load_u8(ip + i + vstep*0);
npyv_u8 b = npyv_load_u8(ip + i + vstep*1);
npyv_u8 c = npyv_load_u8(ip + i + vstep*2);
npyv_u8 d = npyv_load_u8(ip + i + vstep*3);
npyv_b8 m_a = npyv_cmpeq_u8(a, zero);
npyv_b8 m_b = npyv_cmpeq_u8(b, zero);
npyv_b8 m_c = npyv_cmpeq_u8(c, zero);
npyv_b8 m_d = npyv_cmpeq_u8(d, zero);
npyv_b8 m_ab = npyv_and_b8(m_a, m_b);
npyv_b8 m_cd = npyv_and_b8(m_c, m_d);
npy_uint64 m = npyv_tobits_b8(npyv_and_b8(m_ab, m_cd));
if ((npy_int64)m != ((1LL << vstep) - 1)) { // a non-zero found
    break;
}
```

Performance Beyond Contiguous Iteration

- SIMD is hard in C
- SIMD reduces loop iteration
- Loop unrolling
  - Reduce `for`-loop iterations
  - May optimizes CPU branch prediction

IV(b.): Using C-Arrays, Pointer Arithmetic, Loop Unrolling

Only process 1D, Boolean, contiguous arrays

Use `PyArray_DATA()` to get C-array

Advance through array with pointer arithmetic, unrolling units of 4

Use `lldiv` to get quotient and remainder

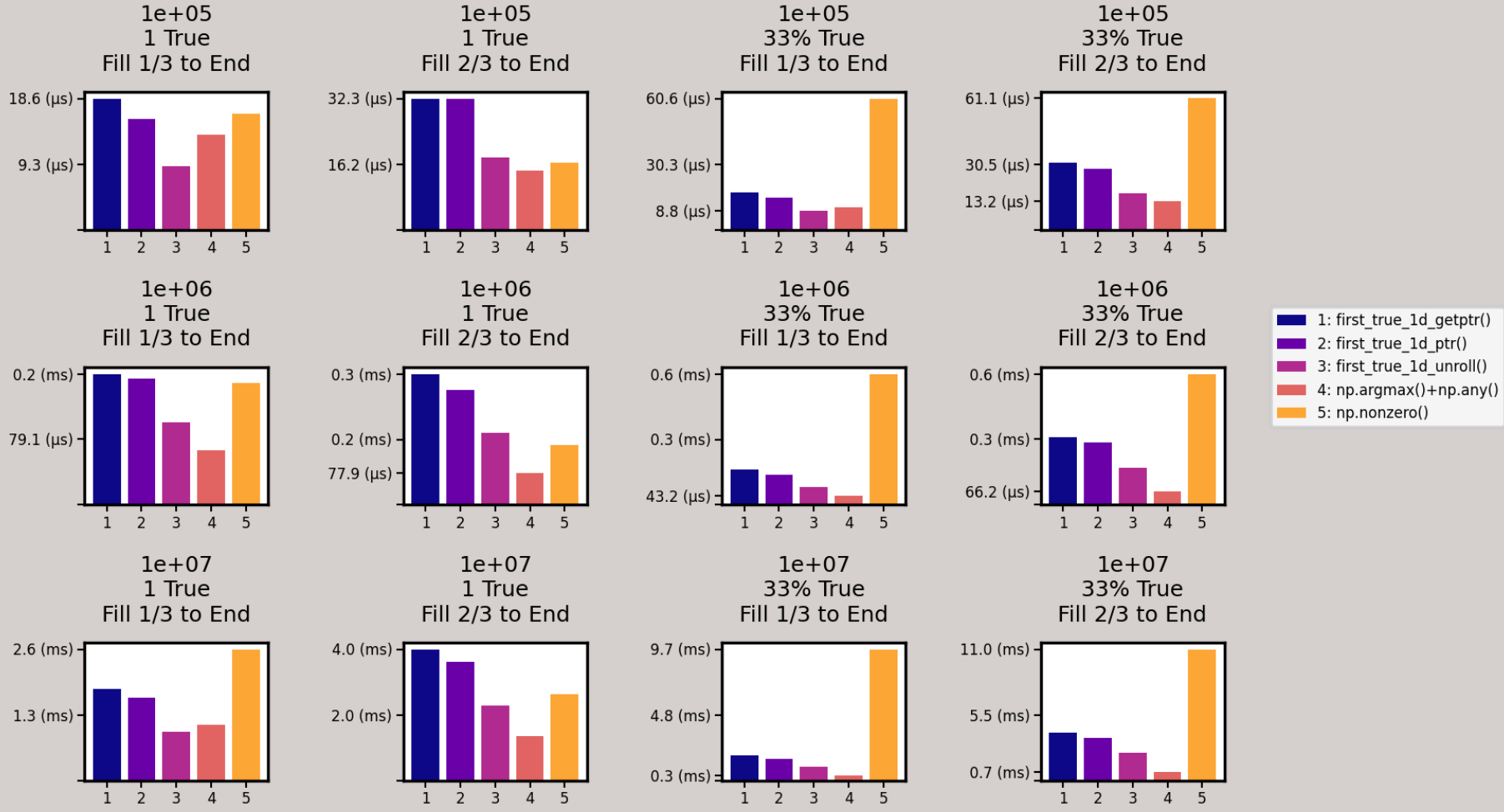
Use element-wise looping for remainder

IV(b.): Using C-Arrays, Pointer Arithmetic, Loop Unrolling

```
if (forward) {
    p = array_buffer;
    p_end = p + size;
    while (p < p_end - size_div.rem) {
        if (*p) {break;}
        p++;
        if (*p) {break;}
        p++;
        if (*p) {break;}
        p++;
        if (*p) {break;}
        p++;
    }
    while (p < p_end) {
        if (*p) {break;}
        p++;
    }
}
```

first_true_1d() Performance with PyArray_DATA() and Loop Unrolling

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



Performance Beyond Contiguous Iteration

- SIMD looks ahead for `True`
- Two look-ahead options
  - Use `memcmp()` to compare raw memory to a zero array buffer
  - Cast 8 bytes of memory to `npy_uint64` and compare to `0`
- Efficiently forward scans 8 1-byte Booleans
- Upon discovery of `True`, do element-wise iteration

IV(c.): Using C-Arrays, Forward Scan

Only process 1D, Boolean, contiguous arrays

Use `PyArray_DATA()` to get C-array

Forward scanning of 8 1-byte Booleans by casting to `npy_uint64`

Jump by `lookahead`, i.e., `sizeof(npy_uint64)`

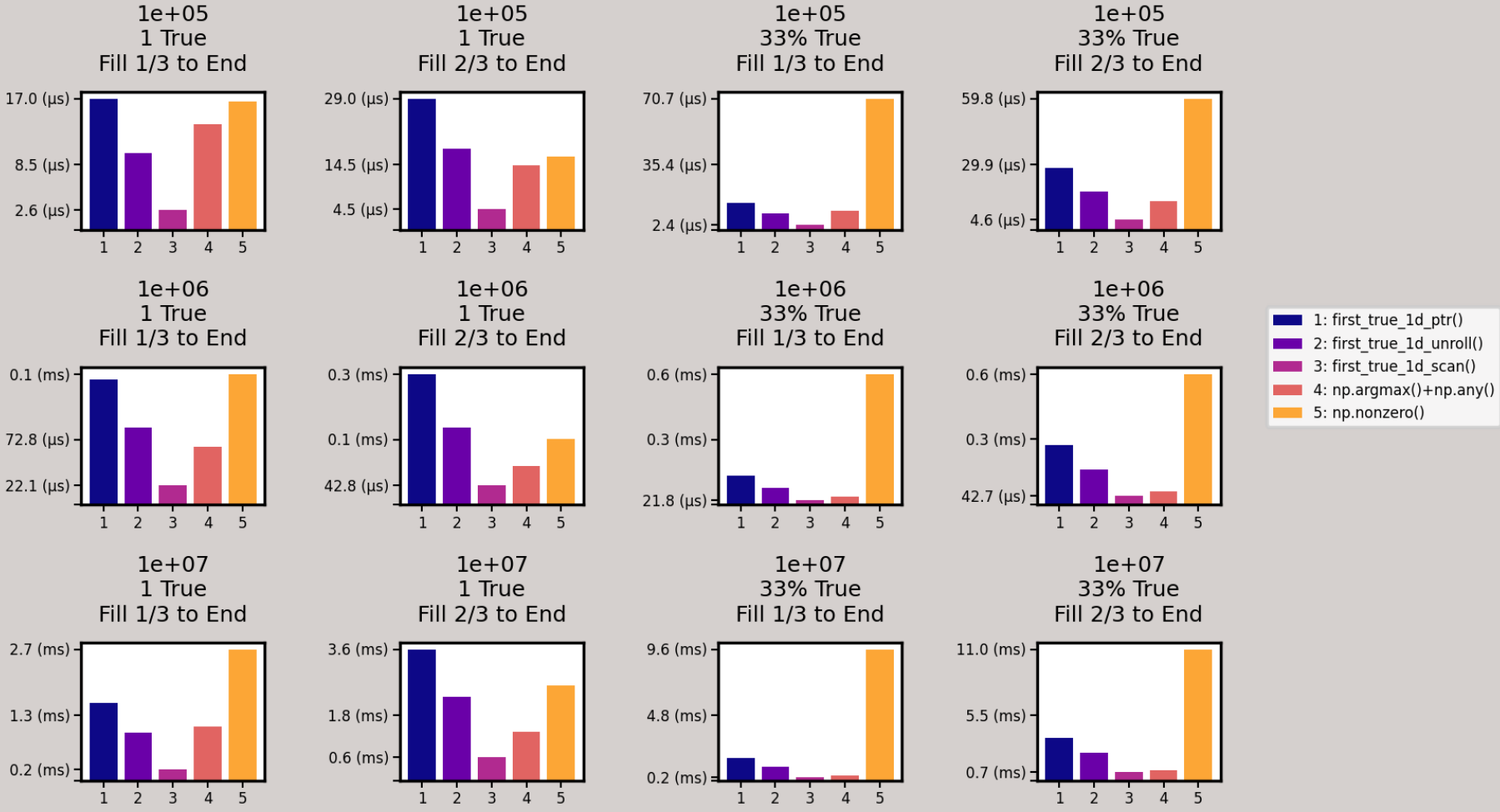
Less code than loop unrolling

IV(c.): Using C-Arrays, Forward Scan

```
if (forward) {
    p = array_buffer;
    p_end = p + size;
    while (p < p_end - size_div.rem) {
        if (*(npy_uint64*)p != 0) {
            break;
        }
        p += lookahead;
    }
    while (p < p_end) {
        if (*p) {
            break;
        }
        p++;
    }
}
```

first_true_1d() Performance with PyArray_DATA() and Forward Scan

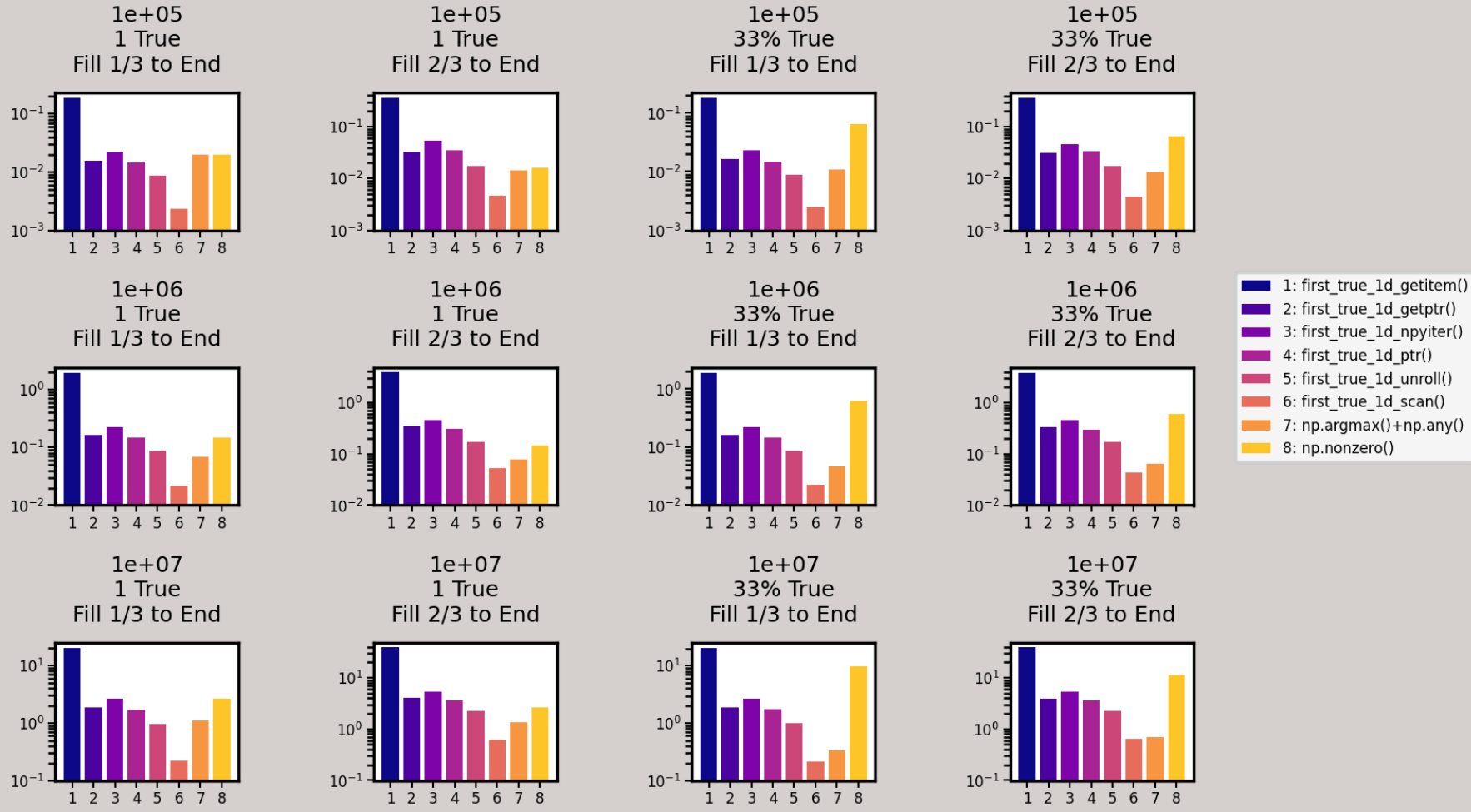
Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



The big (logarithmic) picture

first_true_1d() Performance (Log scale)

Plots of duration (lower is faster) / OS: Linux / NumPy: 1.23.5 / Iterations: 1000



Out-performing NumPy is hard!

Reflections

- Recognize when the work can be done with C-types
- Implement limited functions
  - Only support needed dimensionality
  - Only support needed dtypes
  - Require contiguity
- Constantly test performance

Thank You

Sli.dev slide toolkit

Code: https://github.com/flexatone/np-bench

`first_true_1d`, `first_true_2d` packaged: https://pypi.org/project/arraykit

StaticFrame: https://static-frame.dev