simd

February 5, 2026

1 SIMD Autovectorization in Numba

WARNING: *Due to CPU limitations on Binder, not all the benefits of SIMD instructions will be visible. You will see better SIMD performance if you download this notebook and run on Intel Haswell / AMD Zen or later.*

Most modern CPUs have support for instructions that apply the same operation to multiple data elements simultaneously. These are called "Single Instruction, Multiple Data" (SIMD) operations, and the LLVM backend used by Numba can generate them in some cases to execute loops more quickly. (This process is called "autovectorization.")

For example, Intel processors have support for SIMD instruction sets like:

- SSE (128-bit inputs)
- AVX (256-bit inputs)
- AVX-512 (512-bit inputs, Skylake-X and later or Xeon Phi)

These wide instructions typically operate on as many values as will fit into an input register. For AVX instructions, this means that either 8 float32 values or 4 float64 values can be processed as a single input. As a result, the NumPy dtype that you use can potentially impact performance to a greater degree than when SIMD is not in use.

[1]:
```
import numpy as np
from numba import jit
```

It can be somewhat tricky to determine when LLVM has successfully autovectorized a loop. The Numba team is working on exporting diagnostic information to show where the autovectorizer has generated SIMD code. For now, we can use a fairly crude approach of searching the assembly language generated by LLVM for SIMD instructions.

It is also interesting to note what kind of SIMD is used on your system. On x86_64, the name of the registers used indicates which level of SIMD is in use:

- SSE: `xmmX`
- AVX/AVX2: `ymmX`
- AVX-512: `zmmX`

where X is an integer.

Note: The method we use below to find SIMD instructions will only work on Intel/AMD CPUs. Other platforms have entirely different assembly language syntax for SIMD instructions.

```
[2]: def find_instr(func, keyword, sig=0, limit=5):
         count = 0
         for l in func.inspect_asm(func.signatures[sig]).split('\n'):
             if keyword in l:
                 count += 1
                 print(l)
                 if count >= limit:
                     break
         if count == 0:
             print('No instructions found')
```

1.1 Basic SIMD

Let's start with a simple function that returns the square difference between two arrays, as you might write for a least-squares optimization:

```
[3]: @jit(nopython=True)
     def sqdiff(x, y):
         out = np.empty_like(x)
         for i in range(x.shape[0]):
             out[i] = (x[i] - y[i])**2
         return out
```

```
[4]: x32 = np.linspace(1, 2, 10000, dtype=np.float32)
     y32 = np.linspace(2, 3, 10000, dtype=np.float32)
     sqdiff(x32, y32)
```

```
[4]: array([1.        , 0.99999976, 1.        , …, 1.        , 1.0000002 ,
            1.        ], dtype=float32)
```

```
[5]: x64 = x32.astype(np.float64)
     y64 = y32.astype(np.float64)
     sqdiff(x64, y64)
```

```
[5]: array([1.        , 0.99999976, 1.        , …, 1.        , 1.00000024,
            1.        ])
```

Numba has created two different implementations of the function, one for `float32` 1-D arrays, and one for `float64` 1-D arrays:

```
[6]: sqdiff.signatures
```

```
[6]: [(Array(float32, 1, 'C', False, aligned=True),
       Array(float32, 1, 'C', False, aligned=True)),
      (Array(float64, 1, 'C', False, aligned=True),
       Array(float64, 1, 'C', False, aligned=True))]
```

This allows Numba (and LLVM) to specialize the use of the SIMD instructions for each situation. In particular, using lower precision floating point allows twice as many values to fit into a SIMD register. We will see that for the same number of elements, the `float32` calculation goes twice as fast:

[7]:
```
%timeit sqdiff(x32, y32)
%timeit sqdiff(x64, y64)
```

```
909 ns ± 23.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
2.14  s ± 47 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

We can check for SIMD instructions in both cases. (Due to the order of compilation above, signature 0 is the `float32` implementation and signature 1 is the `float64` implementation.)

[8]:
```
print('float32:')
find_instr(sqdiff, keyword='fsub', sig=0) # Notación para arquitectura M4
print('---\nfloat64:')
find_instr(sqdiff, keyword='fsub', sig=1) # Notación para arquitectura M4
```

```
float32:
        fsub    s0, s0, s1
        fsub.4s v0, v0, v4
        fsub.4s v1, v1, v5
        fsub.4s v2, v2, v6
        fsub.4s v3, v3, v7
---
float64:
        fsub    d0, d0, d1
        fsub.2d v0, v0, v4
        fsub.2d v1, v1, v5
        fsub.2d v2, v2, v6
        fsub.2d v3, v3, v7
```

In x86_64 assembly, SSE uses `subps` for "subtraction packed single precision" (AVX uses `vsubps`), representing vector float32 operations. The `subpd` instruction (AVX = `vsubpd`) stands for "subtraction packed double precision", representing float64 operations.

1.2 SIMD and Division

In general, the autovectorizer cannot deal with branches inside loops, although this is an area where LLVM is likely to improve in the future. Your best bet for SIMD acceleration is to only have pure math operations in the loop.

As a result, you would naturally assume a function like this would be OK:

[9]:
```
@jit(nopython=True)
def frac_diff1(x, y):
    out = np.empty_like(x)
    for i in range(x.shape[0]):
        out[i] = 2 * (x[i] - y[i]) / (x[i] + y[i])
    return out
```

[10]:
```
frac_diff1(x32, y32)
```

[10]:
```
array([-0.6666667 , -0.66662216, -0.66657776, …, -0.400032  ,
       -0.40001604, -0.4       ], dtype=float32)
```

[11]:
```
find_instr(frac_diff1, keyword='fsub', sig=0)
```

```
        fsub    s0, s0, s1
```

`No instructions found`?!

The problem is that division by zero can behave in two different ways:

- In Python, division by zero raises an exception.
- In NumPy, division by zero results in a `NaN`, like in C.

By default, Numba `@jit` follows the Python convention, and `@vectorize`/`@guvectorize` follow the NumPy convention. When following the Python convention, a simple division operation `r = x / y` expands out into something like:

```
if y == 0:
    raise ZeroDivisionError()
else:
    r = x / y
```

This branching code causes the autovectorizer to give up, and no SIMD to be generated for our example above.

Fortunately, Numba allows you to override the “error model” of the function if you don’t want a `ZeroDivisionError` to be raised:

[12]:
```
@jit(nopython=True, error_model='numpy')
def frac_diff2(x, y):
    out = np.empty_like(x)
    for i in range(x.shape[0]):
        out[i] = 2 * (x[i] - y[i]) / (x[i] + y[i])
    return out
```

[13]:
```
frac_diff2(x32, y32)
```

[13]:
```
array([-0.6666667 , -0.66662216, -0.66657776, …, -0.400032  ,
       -0.40001604, -0.4       ], dtype=float32)
```

[14]:
```
find_instr(frac_diff2, keyword='fsub', sig=0)
```

```
        fsub    s2, s0, s1
        fsub.4s v2, v0, v1
```

We have SIMD instructions again, but when we check the speed:

[15]:
```
%timeit frac_diff2(x32, y32)
%timeit frac_diff2(x64, y64)
```

```
3.01  s ± 143 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
2.22  s ± 55.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

This is faster than the no-SIMD case, but there doesn't seem to be a speed benefit with `float32` inputs. What's going on?

The remaining issue is very subtle. We can see it if we look at a type-annotated version of the function:

[16]:
```
frac_diff2.inspect_types() # pretty=True falla por culpa de la version
```

```
frac_diff2 (Array(float32, 1, 'C', False, aligned=True), Array(float32, 1, 'C',
False, aligned=True))
--------------------------------------------------------------------------------
# File:
/var/folders/65/r55xnbt14k5fgbrtf11p9g4m0000gp/T/ipykernel_6315/2090011437.py
# --- LINE 1 ---
# label 0
#   x = arg(0, name=x)  :: array(float32, 1d, C)
#   y = arg(1, name=y)  :: array(float32, 1d, C)

@jit(nopython=True, error_model='numpy')

# --- LINE 2 ---

def frac_diff2(x, y):

    # --- LINE 3 ---
    #   $4load_global.0 = global(np: <module 'numpy' from
'/Users/ivandominguez/anaconda3/lib/python3.11/site-
packages/numpy/__init__.py'>)  :: Module(<module 'numpy' from
'/Users/ivandominguez/anaconda3/lib/python3.11/site-
packages/numpy/__init__.py'>)
    #   $16load_method.2 = getattr(value=$4load_global.0, attr=empty_like)  ::
Function(<built-in function empty_like>)
    #   del $4load_global.0
    #   out = call $16load_method.2(x, func=$16load_method.2, args=[Var(x,
2090011437.py:1)], kws=(), vararg=None, varkwarg=None, target=None)  ::
(Array(float32, 1, 'C', False, aligned=True), omitted(default=None)) ->
array(float32, 1d, C)
    #   del $16load_method.2

    out = np.empty_like(x)

    # --- LINE 4 ---
    #   $56load_global.5 = global(range: <class 'range'>)  :: Function(<class
'range'>)
    #   $70load_attr.8 = getattr(value=x, attr=shape)  :: UniTuple(int64 x 1)
    #   $const80.9.1 = const(int, 0)  :: Literal[int](0)
    #   $82binary_subscr.10 = static_getitem(value=$70load_attr.8, index=0,
index_var=$const80.9.1, fn=<built-in function getitem>)  :: int64
    #   del $const80.9.1
    #   del $70load_attr.8
    #   $96call.11 = call $56load_global.5($82binary_subscr.10,
func=$56load_global.5, args=[Var($82binary_subscr.10, 2090011437.py:4)], kws=(),
vararg=None, varkwarg=None, target=None)  :: (int64,) -> range_state_int64
    #   del $82binary_subscr.10
    #   del $56load_global.5
    #   $106get_iter.12 = getiter(value=$96call.11)  :: range_iter_int64
    #   del $96call.11
    #   $phi108.0 = $106get_iter.12  :: range_iter_int64
    #   del $106get_iter.12
    #   jump 108
    # label 108
    #   $108for_iter.1 = iternext(value=$phi108.0)  :: pair<int64, bool>
    #   $108for_iter.2 = pair_first(value=$108for_iter.1)  :: int64
    #   $108for_iter.3 = pair_second(value=$108for_iter.1)  :: bool
    #   del $108for_iter.1
    #   $phi110.1 = $108for_iter.2  :: int64
    #   del $108for_iter.2
    #   branch $108for_iter.3, 110, 196
    # label 110
    #   del $108for_iter.3
    #   i = $phi110.1  :: int64
    #   del $phi110.1

    for i in range(x.shape[0]):

        # --- LINE 5 ---
        #   $const112.2.2 = const(int, 2)  :: Literal[int](2)
        #   $118binary_subscr.5 = getitem(value=x, index=i, fn=<built-in
function getitem>)  :: float32
        #   $132binary_subscr.8 = getitem(value=y, index=i, fn=<built-in
function getitem>)  :: float32
        #   $binop_sub142.9 = $118binary_subscr.5 - $132binary_subscr.8  ::
float32
        #   del $132binary_subscr.8
        #   del $118binary_subscr.5
        #   $binop_mul146.10 = $const112.2.2 * $binop_sub142.9  :: float64
        #   del $const112.2.2
        #   del $binop_sub142.9
        #   $154binary_subscr.13 = getitem(value=x, index=i, fn=<built-in
function getitem>)  :: float32
        #   $168binary_subscr.16 = getitem(value=y, index=i, fn=<built-in
function getitem>)  :: float32
        #   $binop_add178.17 = $154binary_subscr.13 + $168binary_subscr.16  ::
float32
        #   del $168binary_subscr.16
        #   del $154binary_subscr.13
        #   $binop_truediv182.18 = $binop_mul146.10 / $binop_add178.17  ::
float64
        #   del $binop_mul146.10
        #   del $binop_add178.17
        #   out[i] = $binop_truediv182.18  :: (Array(float32, 1, 'C', False,
aligned=True), int64, float64) -> none
        #   del i
        #   del $binop_truediv182.18
        #   jump 108

        out[i] = 2 * (x[i] - y[i]) / (x[i] + y[i])

    # --- LINE 6 ---
    # label 196
    #   del y
    #   del x
    #   del $phi110.1
    #   del $phi108.0
    #   del $108for_iter.3
    #   $198return_value.1 = cast(value=out)  :: array(float32, 1d, C)
    #   del out
    #   return $198return_value.1

    return out


================================================================================
frac_diff2 (Array(float64, 1, 'C', False, aligned=True), Array(float64, 1, 'C',
False, aligned=True))
--------------------------------------------------------------------------------
# File:
/var/folders/65/r55xnbt14k5fgbrtf11p9g4m0000gp/T/ipykernel_6315/2090011437.py
# --- LINE 1 ---
# label 0
#   x = arg(0, name=x)  :: array(float64, 1d, C)
#   y = arg(1, name=y)  :: array(float64, 1d, C)

@jit(nopython=True, error_model='numpy')

# --- LINE 2 ---

def frac_diff2(x, y):
    # --- LINE 3 ---
    #   $4load_global.0 = global(np: <module 'numpy' from
'/Users/ivandominguez/anaconda3/lib/python3.11/site-
packages/numpy/__init__.py'>)  :: Module(<module 'numpy' from
'/Users/ivandominguez/anaconda3/lib/python3.11/site-
packages/numpy/__init__.py'>)
    #   $16load_method.2 = getattr(value=$4load_global.0, attr=empty_like)  ::
Function(<built-in function empty_like>)
    #   del $4load_global.0
    #   out = call $16load_method.2(x, func=$16load_method.2, args=[Var(x,
2090011437.py:1)], kws=(), vararg=None, varkwarg=None, target=None)  ::
(Array(float64, 1, 'C', False, aligned=True), omitted(default=None)) ->
array(float64, 1d, C)
    #   del $16load_method.2

    out = np.empty_like(x)

    # --- LINE 4 ---
    #   $56load_global.5 = global(range: <class 'range'>)  :: Function(<class
'range'>)
    #   $70load_attr.8 = getattr(value=x, attr=shape)  :: UniTuple(int64 x 1)
    #   $const80.9.1 = const(int, 0)  :: Literal[int](0)
    #   $82binary_subscr.10 = static_getitem(value=$70load_attr.8, index=0,
index_var=$const80.9.1, fn=<built-in function getitem>)  :: int64
    #   del $const80.9.1
    #   del $70load_attr.8
    #   $96call.11 = call $56load_global.5($82binary_subscr.10,
func=$56load_global.5, args=[Var($82binary_subscr.10, 2090011437.py:4)], kws=(),
vararg=None, varkwarg=None, target=None)  :: (int64,) -> range_state_int64
    #   del $82binary_subscr.10
    #   del $56load_global.5
    #   $106get_iter.12 = getiter(value=$96call.11)  :: range_iter_int64
    #   del $96call.11
    #   $phi108.0 = $106get_iter.12  :: range_iter_int64
    #   del $106get_iter.12
    #   jump 108
    # label 108
    #   $108for_iter.1 = iternext(value=$phi108.0)  :: pair<int64, bool>
    #   $108for_iter.2 = pair_first(value=$108for_iter.1)  :: int64
    #   $108for_iter.3 = pair_second(value=$108for_iter.1)  :: bool
    #   del $108for_iter.1
    #   $phi110.1 = $108for_iter.2  :: int64
    #   del $108for_iter.2
    #   branch $108for_iter.3, 110, 196
    # label 110
    #   del $108for_iter.3
    #   i = $phi110.1  :: int64
    #   del $phi110.1
    for i in range(x.shape[0]):

        # --- LINE 5 ---
        #   $const112.2.2 = const(int, 2)  :: Literal[int](2)
        #   $118binary_subscr.5 = getitem(value=x, index=i, fn=<built-in
function getitem>)  :: float64
        #   $132binary_subscr.8 = getitem(value=y, index=i, fn=<built-in
function getitem>)  :: float64
        #   $binop_sub142.9 = $118binary_subscr.5 - $132binary_subscr.8  ::
float64
        #   del $132binary_subscr.8
        #   del $118binary_subscr.5
        #   $binop_mul146.10 = $const112.2.2 * $binop_sub142.9  :: float64
        #   del $const112.2.2
        #   del $binop_sub142.9
        #   $154binary_subscr.13 = getitem(value=x, index=i, fn=<built-in
function getitem>)  :: float64
        #   $168binary_subscr.16 = getitem(value=y, index=i, fn=<built-in
function getitem>)  :: float64
        #   $binop_add178.17 = $154binary_subscr.13 + $168binary_subscr.16  ::
float64
        #   del $168binary_subscr.16
        #   del $154binary_subscr.13
        #   $binop_truediv182.18 = $binop_mul146.10 / $binop_add178.17  ::
float64
        #   del $binop_mul146.10
        #   del $binop_add178.17
        #   out[i] = $binop_truediv182.18  :: (Array(float64, 1, 'C', False,
aligned=True), int64, float64) -> none
        #   del i
        #   del $binop_truediv182.18
        #   jump 108

        out[i] = 2 * (x[i] - y[i]) / (x[i] + y[i])

    # --- LINE 6 ---
    # label 196
    #   del y
    #   del x
    #   del $phi110.1
    #   del $phi108.0
    #   del $108for_iter.3
    #   $198return_value.1 = cast(value=out)  :: array(float64, 1d, C)
    #   del out
    #   return $198return_value.1

    return out
```

```
================================================================================
```

If you expand out line 5 in the float32 version of the function, you will see the following bit of Numba IR:

```
$const28.2 = const(float, 2.0) :: float64
$28.5 = getitem(value=x, index=i) :: float32
$28.8 = getitem(value=y, index=i) :: float32
$28.9 = $28.5 - $28.8 :: float32
del $28.8
del $28.5
$28.10 = $const28.2 * $28.9 :: float64
```

Notice that the constant 2 has been typed as a float64 value. Later, this causes the multiplication `2 * (x[i] - y[i]` to promote up to float64, and then the rest of the calculation becomes float64. This is a situation where Numba is being overly conservative (and should be fixed at some point), but we can tweak this behavior by casting the constant to the type we want:

[17]:
```python
@jit(nopython=True, error_model='numpy')
def frac_diff3(x, y):
    out = np.empty_like(x)
    dt = x.dtype # Cast the constant using the dtype of the input
    for i in range(x.shape[0]):
        # Could also use np.float32(2) to always use same type, regardless of␣
 ↪input
        out[i] = dt.type(2) * (x[i] - y[i]) / (x[i] + y[i])
    return out
```

[18]:
```python
frac_diff3(x32, y32)
```

[18]:
```
array([-0.6666667 , -0.66662216, -0.66657776, …, -0.400032   ,
       -0.40001604, -0.4       ], dtype=float32)
```

[19]:
```python
%timeit frac_diff3(x32, y32)
%timeit frac_diff3(x64, y64)
```

```
1.23  s ± 34.1 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
2.27  s ± 60.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Now our float32 version is nice and speedy (and 6x faster than what we started with, if we only care about float32).

1.3 SIMD and Reductions

The autovectorizer can also optimize reduction loops, but only with permission. Normally, compilers are very careful not to reorder floating point instructions because floating point arithmetic is approximate, so mathematically allowed transformations do not always give the same result. For example, it is not generally true for floating point numbers that:

```
(a + (b + c)) == ((a + b) + c)
```

For many situations, the round-off error that causes the difference between the left and the right is not important, so changing the order of additions is acceptable for a performance increase.

To allow reordering of operations, we need to tell Numba to enable `fastmath` optimizations:

[20]:
```
@jit(nopython=True)
def do_sum(A):
    acc = 0.
    # without fastmath, this loop must accumulate in strict order
    for x in A:
        acc += x**2
    return acc

@jit(nopython=True, fastmath=True)
def do_sum_fast(A):
    acc = 0.
    # with fastmath, the reduction can be vectorized as floating point
    # reassociation is permitted.
    for x in A:
        acc += x**2
    return acc
```

[21]:
```
do_sum(x32)
find_instr(do_sum, keyword='fmul')  # Notación para el M4
```

```
        fmul    s1, s1, s1
        fmul    s2, s2, s2
        fmul    s3, s3, s3
        fmul    s4, s4, s4
        fmul    s1, s1, s1
```

[22]:
```
do_sum_fast(x32)
find_instr(do_sum_fast, keyword='fmul') # Notación para el M4
```

```
        fmul    s4, s4, s4
        fmul    s5, s5, s5
        fmul    s6, s6, s6
        fmul    s7, s7, s7
        fmul    s1, s1, s1
```

The fast version is 4x faster:

[23]:
```
%timeit do_sum(x32)
%timeit do_sum_fast(x32)
```

```
5.56  s ± 260 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
1.87  s ± 8.83 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

1.4 SIMD and Special Functions

If you follow the above guidelines, SIMD autovectorization will work for all basic math operations (+,-,*,\), but generally will not work for function calls in the loop, unless LLVM can inline the function and there is only basic math in the function body.

However, we build Numba (if you get conda packages from Anaconda or wheels from PyPI) using a patched version of LLVM that supports vectorization of special math functions when Intel SVML ("Short Vector Math Library") is present. This library comes with the Intel compiler, and is also freely redistributable. We've installed it in the current conda environment using `conda install -c numba icc_rt`, as we can verify here:

[24]:
```
# ! conda install -c numba icc_rt no funciona en esta arquitectura
# su equivalente parece ser el siguiente

! conda install -c conda-forge numba
```

```
Channels:
 - conda-forge
 - defaults
Platform: osx-arm64
Collecting package metadata (repodata.json): done
Solving environment: done

# All requested packages already installed.
```

[25]:
```
# en este caso el equivalente para mac sería:

! conda list "numba|llvm|libblas"
```

```
# packages in environment at /Users/ivandominguez/anaconda3:
#
# Name                    Version           Build             Channel
libllvm14                 14.0.6            h19fdd8a_4
libllvm20                 20.1.8            h1701f07_0
llvm-openmp               20.1.8            he822017_0
llvmlite                  0.45.1            py311hc8eb11b_0
numba                     0.62.1            py311h471b49b_0
```

Thanks to this library, we can still get SIMD vectorization in a function like this:

[26]:
```
SQRT_2PI = np.sqrt(2 * np.pi)

@jit(nopython=True, error_model='numpy', fastmath=True)
def kde(x, means, widths):
    '''Compute value of gaussian kernel density estimate.

    x - location of evaluation
    means - array of kernel means
    widths - array of kernel widths
    '''
    n = means.shape[0]
    acc = 0.
    for i in range(n):
        acc += np.exp( -0.5 * ((x - means[i]) / widths[i])**2 ) / widths[i]
    return acc / SQRT_2PI / n
```

```
[27]: # The distribution we are approximating is flat between -1 and 1, so we expect␣
       ↪a KDE value of ~0.5 everywhere
      means = np.random.uniform(-1, 1, size=10000)
      # These widths are not selected in any reasonable way.  Consult your local␣
       ↪statistician before approximating a PDF.
      widths = np.random.uniform(0.1, 0.3, size=10000)

      kde(0.4, means, widths)
```

```
[27]: 0.48401212391619747
```

We can see that SIMD instructions were generated:

```
[28]: find_instr(kde, 'fsub')
```

```
        fsub.2d v0, v4, v0
        fsub.2d v1, v4, v1
        fsub.2d v2, v4, v2
        fsub.2d v3, v4, v3
        fsub    d0, d1, d0
```

We can also see that calls to the special Intel SVML functions for `exp` were generated:

```
[29]: find_instr(kde, keyword='v') # equivalente para mac
```

```
        .build_version macos, 16, 0
        .globl  __ZN8__main__3kdeB3v19B64c8tJTIeFIjxB2IKSgI4CrvQCk0Z4yRYcWsCAgXr
KFt1X1eogKfVaTWDAFkEM0AQAEd5ArrayIdLi1E1C7mutable7alignedE5ArrayIdLi1E1C7mutable
7alignedE
__ZN8__main__3kdeB3v19B64c8tJTIeFIjxB2IKSgI4CrvQCk0Z4yRYcWsCAgXrKFt1X1eogKfVaTWD
AFkEM0AQAEd5ArrayIdLi1E1C7mutable7alignedE5ArrayIdLi1E1C7mutable7alignedE:
        mov     x19, x7
        mov     x20, x0
```

```
[30]: %timeit kde(0.4, means, widths)
```

```
22.3  s ± 1.22  s per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

If we recompile the function (which is possible since the `.py_func` attribute holds the original Python function) with the extra flags to allow division and reductions to work, this stops all autovectorization of the loop:

[31]:
```
slow_kde = jit(nopython=True)(kde.py_func)

slow_kde(0.4, means, widths)
```

[31]:
```
0.48401212391619675
```

Note that we get a slightly different answer, both due to the different order of operations, and the small differences in SVML `exp` compared to the default `exp`. We also see that there is no SIMD or calls to SVML:

[32]:
```
find_instr(slow_kde, keyword='fsub') # equivalente para mac
print('---')
find_instr(slow_kde, keyword='v') # equivalente para mac
```

```
        fsub    d0, d8, d0
---
        .build_version macos, 16, 0
        .globl  __ZN8__main__3kdeB3v21B38c8tJTIeFIjxB2IKSgI4CrvQClQZ6FczSBAA_3dE
d5ArrayIdLi1E1C7mutable7alignedE5ArrayIdLi1E1C7mutable7alignedE
__ZN8__main__3kdeB3v21B38c8tJTIeFIjxB2IKSgI4CrvQClQZ6FczSBAA_3dEd5ArrayIdLi1E1C7
mutable7alignedE5ArrayIdLi1E1C7mutable7alignedE:
        mov     x20, x7
        mov     x21, x1
```

And the function is much slower than the original:

[33]:
```
%timeit kde(0.4, means, widths)
%timeit slow_kde(0.4, means, widths)
```

```
22.2  s ± 788 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
22.7  s ± 42.2 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

And only the SIMD vectorized version is faster than doing this in pure NumPy:

[34]:
```
def numpy_kde(x, means, widths):
    acc = (np.exp( -0.5 * ((x - means) / widths)**2 ) / widths).mean()
    # .mean() already divides by n
    return acc / SQRT_2PI
```

[35]:
```
numpy_kde(0.4, means, widths)
```

[35]:
```
0.48401212391619775
```

[36]:
```
%timeit numpy_kde(0.4, means, widths)
```

```
34  s ± 526 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Why is NumPy as fast as it is? In this case, it is because the Anaconda build of NumPy uses MKL to accelerate (with SIMD and threads) many of the individial ufuncs, so it is only when Numba can combine all the operations together that the speed boost emerges.

Incidentally, although we wrote out the iteration for `kde` as a for loop to highlight what was going on, you still get the benefit of SIMD in Numba when compiling array expressions. We could have compiled `numpy_kde` directly:

[37]:
```
numba_numpy_kde = jit(nopython=True, error_model='numpy',␣
 ↪fastmath=True)(numpy_kde)

numba_numpy_kde(0.4, means, widths)
```

[37]:
```
0.48401212391619736
```

[38]:
```
find_instr(numba_numpy_kde, keyword='fsub')
print('---')
find_instr(numba_numpy_kde, keyword='v')
```

```
        fsub    d0, d1, d0
        fsub    d0, d2, d0
        fsub    d0, d1, d0
        fsub    d0, d2, d0
        fsub    d0, d2, d0
---
        .build_version macos, 16, 0
        .globl  __ZN8__main__9numpy_kdeB3v22B64c8tJTIeFIjxB2IKSgI4CrvQCk0Z4yRYcW
sCAgXrKFt1X1eogKfVaTWDAFkEM0AQAEd5ArrayIdLi1E1C7mutable7alignedE5ArrayIdLi1E1C7m
utable7alignedE
__ZN8__main__9numpy_kdeB3v22B64c8tJTIeFIjxB2IKSgI4CrvQCk0Z4yRYcWsCAgXrKFt1X1eogK
fVaTWDAFkEM0AQAEd5ArrayIdLi1E1C7mutable7alignedE5ArrayIdLi1E1C7mutable7alignedE:
        mov     x24, x7
        mov     x20, x6
```

And it is nearly as fast as our manual looping version, and 2x faster than NumPy alone:

[39]:
```
%timeit kde(0.4, means, widths)
%timeit numba_numpy_kde(0.4, means, widths)
%timeit numpy_kde(0.4, means, widths)
```

```
22.1  s ± 490 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
22.4  s ± 46.1 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
33.4  s ± 221 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```