SIMD Autovectorization in Numba

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.

```
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.

```
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')
```

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:

```
@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
```

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

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

```
x64 = x32.astype(np.float64)
y64 = y32.astype(np.float64)
sqdiff(x64, y64)
```

```
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:

```
sqdiff.signatures
```

```
[(array(float32, 1d, C), array(float32, 1d, C)),
 (array(float64, 1d, C), array(float64, 1d, C))]
```

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:

```
%timeit sqdiff(x32, y32)
%timeit sqdiff(x64, y64)
```

```
1.95 µs ± 101 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
3.81 µs ± 73.8 ns per loop (mean ± std. dev. of 7 runs, 100000 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.)

```
print('float32:')
find_instr(sqdiff, keyword='subp', sig=0)
print('---\nfloat64:')
find_instr(sqdiff, keyword='subp', sig=1)
```

```
float32:
        vsubps  (%rax,%rsi,4), %ymm0, %ymm0
        vsubps  32(%rax,%rsi,4), %ymm1, %ymm1
        vsubps  64(%rax,%rsi,4), %ymm2, %ymm2
        vsubps  96(%rax,%rsi,4), %ymm3, %ymm3
        vsubps  128(%rax,%rsi,4), %ymm0, %ymm0
---
float64:
        vsubpd  (%rax,%rsi,8), %ymm0, %ymm0
        vsubpd  32(%rax,%rsi,8), %ymm1, %ymm1
        vsubpd  64(%rax,%rsi,8), %ymm2, %ymm2
        vsubpd  96(%rax,%rsi,8), %ymm3, %ymm3
        vsubpd  128(%rax,%rsi,8), %ymm0, %ymm0
```

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.

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:

```
@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
```

```
frac_diff1(x32, y32)
```

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

```
find_instr(frac_diff1, keyword='subp', sig=0)
```

```
No instructions found
```

`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` or `inf`, 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:

```
@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
```

```
frac_diff2(x32, y32)
```

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

```
find_instr(frac_diff2, keyword='subp', sig=0)
```

```
        vsubps  %ymm1, %ymm0, %ymm2
        vsubps  %ymm1, %ymm0, %ymm2
        vsubps  %ymm1, %ymm0, %ymm2
        vsubps  %ymm1, %ymm0, %ymm2
        vsubps  %ymm1, %ymm0, %ymm2
```

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

```
%timeit frac_diff2(x32, y32)
%timeit frac_diff2(x64, y64)
```

```
5.84 µs ± 45.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
5.83 µs ± 61.4 ns per loop (mean ± std. dev. of 7 runs, 100000 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:

```
frac_diff2.inspect_types(pretty=True)
```

```
/home/betcke/miniconda3/envs/dev/lib/python3.8/site-
packages/numba/core/annotations/pretty_annotate.py:7: FutureWarning: The pretty_annotate
functionality is experimental and might change API
  warn("The pretty_annotate functionality is experimental and might change API",
```

Function name: frac_diff2
in file: <ipython-input-12-115e70afe30f>
with signature: (array(float32, 1d, C), array(float32, 1d, C)) -> array(float32, 1d, C)

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

Function name: frac_diff2
in file: <ipython-input-12-115e70afe30f>
with signature: (array(float64, 1d, C), array(float64, 1d, C)) -> array(float64, 1d, C)

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

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

```
        $const30.2 = const(int, 2) :: Literal[int](2)
        $36binary_subscr.5 = getitem(value=x, index=i) :: float32
        $42binary_subscr.8 = getitem(value=y, index=i) :: float32
        $44binary_subtract.9 = $36binary_subscr.5 - $42binary_subscr.8 :: float32
        del $42binary_subscr.8
        del $36binary_subscr.5
        $46binary_multiply.10 = $const30.2 * $44binary_subtract.9 :: float64```


Notice that the constant `2` has been typed as an int 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:
```

```
@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
```

```
frac_diff3(x32, y32)
```

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

```
%timeit frac_diff3(x32, y32)
%timeit frac_diff3(x64, y64)
```

```
2.27 µs ± 17.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
5.84 µs ± 99.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

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

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:

```
@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
```

```
do_sum(x32)
find_instr(do_sum, keyword='mulp')  # look for vector multiplication
```

```
No instructions found
```

```
do_sum_fast(x32)
find_instr(do_sum_fast, keyword='mulp')
```

```
        vmulps  %xmm1, %xmm1, %xmm1
        vmulps  %xmm7, %xmm7, %xmm4
        vmulps  %xmm6, %xmm6, %xmm1
        vmulps  %xmm5, %xmm5, %xmm1
        vmulps  %xmm1, %xmm1, %xmm1
```

The fast version is 2x faster:

```
%timeit do_sum(x32)
%timeit do_sum_fast(x32)
```

```
9.96 µs ± 135 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
5.11 µs ± 68 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

By Timo Betcke