VecPy

***Vec**torizing **Py**thon for concurrent SIMD execution.*

David Farrow

I wanted to call it *PySPC...*

...but that name was already taken.

- Similar in spirit to ISPC
    - Data-parallel execution via multi-threading and SIMD
- Python-based
    - Parses Python, compiles shared library

Python is great, but it is *slow*

- Very popular programming language
  - Friendly syntax, good for rapid prototyping
  - Succinct but powerful: less is more
  - Built-in modules for parsing and inspecting code
- Byte-code based, sequential execution
  - Orders of magnitude slower than sequential C
  - Concurrent, not simultaneous, threading
  - No utilization of SIMD

VecPy: the best of both worlds

Write your function in Python…

(let VecPy do the heavy lifting)

...then execute an optimized* version of it.

*Written in C++, using all available execution contexts and vector extensions

Designed with four goals in mind

Simplicity Should make the programmer’s life easier.	***Flexibility*** Should support many architectures and data types.
Utility Should be useful for real-world applications.	***Efficiency*** Should completely utilize the hardware.

VecPy is simple to use

```
#Import VecPy
from vecpy.runtime import *
from vecpy.compiler_constants import *

#Define the kernel
def volume(radius, volume):
  volume = (4/3 * math.pi) * (radius ** 3)

#Generate some data
def data():
  array = get_array('f', 10)
  for i in range(len(array)): array[i] = (.1 + i/10)
  return array
radii, volumes = data(), data()

#Call VecPy to generate the native module
vectorize(volume, Options(Architecture.avx2, DataType.float))

#Import the newly-minted module and execute kernel
from vecpy_volume import volume
volume(radii, volumes)

#Print the results!
print('Radius:', ', '.join('%.3f'%(r) for r in radii))
print('Volume:', ', '.join('%.3f'%(v) for v in volumes))
```

VecPy has flexible targets

- Architectures
  - Generic
  - SSE4.2
  - AVX2
- Data Types
  - float
  - uint32
- Language Bindings
  - Python
  - Java
  - C++

VecPy implements a useful feature set

- Operators
  - `+ - * / // % ** == != > >= < <= & | ^ ~ << >> and or not`
- Functions
  - `abs, acos, acosh, asin, asinh, atan, atan2, atanh, ceil, copysign, cos, cosh, erf, erfc, exp, expm1, fabs, floor, fmod, gamma, hypot, lgamma, log, log10, log1p, log2, max, min, pow, pow, round, sin, sinh, sqrt, tan, tanh, trunc`
- Constants
  - `pi, e`
- Syntax
  - multi-assignments
  - if-elif-else branches
  - while loops

VecPy generates efficient code

```python
#Define the kernel
def mandelbrot(row, col, count, max:    'uniform',
               w_m1: 'uniform', h_m1:   'uniform',
               left: 'uniform', right:  'uniform',
               top:  'uniform', bottom: 'uniform'):
  """Tests if a point is in the Mandelbrot set."""
  x0 = left + col * (right - left) / w_m1
  y0 = bottom + (h_m1 - row) * (top - bottom) / h_m1
  x = y = count = 0
  xx, yy = (x * x), (y * y)
  #Standard escape time
  while (count < max and xx + yy < 16):
    x, y = (xx - yy + x0), (2 * x * y + y0)
    xx, yy = (x * x), (y * y)
    count += 1
  #Smooth shading
  if count < max:
    count += 1 - math.log2(math.log2(xx + yy) / 2)

#Call VecPy to generate the native module
from vecpy.runtime import *
from vecpy.compiler_constants import *
vectorize(mandelbrot, Options(Architecture.avx2, DataType.float))
```

VecPy generates efficient code

```
//>>> while (count < max and xx + yy < 16):
mask031 = _mm256_cmp_ps(count, max, _CMP_LT_OQ);
var032 = _mm256_add_ps(xx, yy);
mask034 = _mm256_cmp_ps(var032, lit033, _CMP_LT_OQ);
mask030 = _mm256_and_ps(mask031, mask034);
mask035 = _mm256_and_ps(mask030, MASK_TRUE);
while(_mm256_movemask_ps(mask035)) {
    //>>> x, y = (xx - yy + x0), (2 * x * y + y0)
    var037 = _mm256_sub_ps(xx, yy);
    var036 = _mm256_add_ps(var037, x0);
    var040 = _mm256_mul_ps(lit041, x);
    var039 = _mm256_mul_ps(var040, y);
    var038 = _mm256_add_ps(var039, y0);
    x = _mm256_or_ps(_mm256_and_ps(mask035, var036), _mm256_andnot_ps(mask035, x));
    y = _mm256_or_ps(_mm256_and_ps(mask035, var038), _mm256_andnot_ps(mask035, y));
    //>>> xx, yy = (x * x), (y * y)
    var042 = _mm256_mul_ps(x, x);
    var043 = _mm256_mul_ps(y, y);
    xx = _mm256_or_ps(_mm256_and_ps(mask035, var042), _mm256_andnot_ps(mask035, xx));
    yy = _mm256_or_ps(_mm256_and_ps(mask035, var043), _mm256_andnot_ps(mask035, yy));
    //>>> count += 1
    var044 = _mm256_add_ps(count, lit045);
    count = _mm256_or_ps(_mm256_and_ps(mask035, var044), _mm256_andnot_ps(mask035, count));
    //>>> while (count < max and xx + yy < 16):
    mask031 = _mm256_cmp_ps(count, max, _CMP_LT_OQ);
    var032 = _mm256_add_ps(xx, yy);
    mask034 = _mm256_cmp_ps(var032, lit033, _CMP_LT_OQ);
    mask030 = _mm256_and_ps(mask031, mask034);
    mask035 = _mm256_and_ps(mask030, MASK_TRUE);
}
```

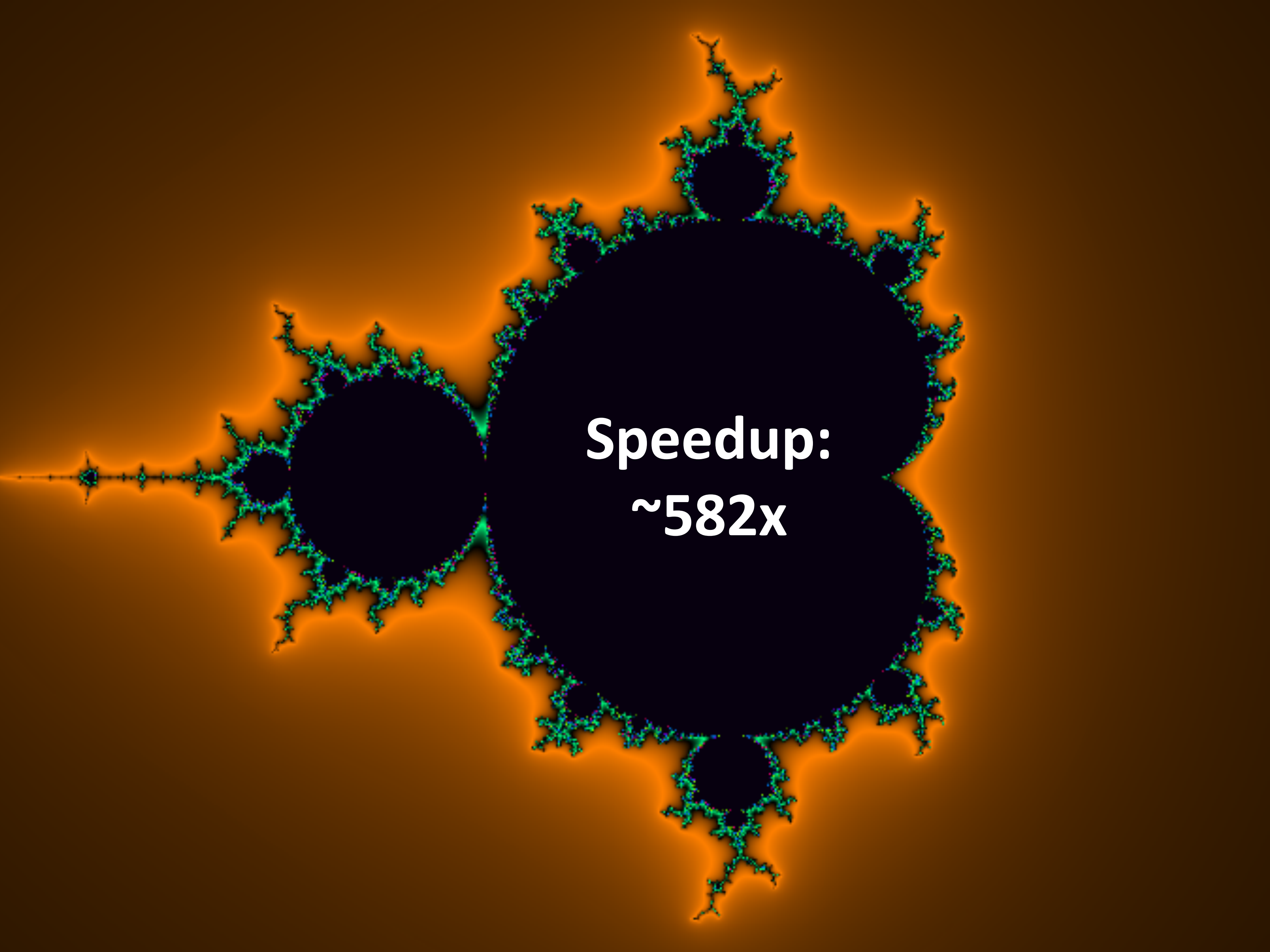
VecPy gives *sequential* speedup

- Mandelbrot performance comparison
    - 1920x1280 image
    - Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz (dual core)
    - Measured minimum render time over 10 frames

Pure-Python	*Pure-Java*	*VecPy-C++*
1.0x	**85.2x**	**97.4x**
41676ms	489ms	428ms

VecPy gives *parallel* speedup

	Generic	SSE4.2	AVX2
1 Thread	*1x* **1.0x** 428ms	*4x* **1.8x** 243ms	*8x* **2.5x** 171ms
2 Threads	*2x* **1.8x** 232ms	*8x* **3.2x** 136ms	*16x* **4.9x** 88ms
4 Threads (2 HT)	*4x* **2.4x** 178ms	*16x* **3.9x** 110ms	*32x* **6.0x** 71ms


Speedup:
~582x

Try it out! (Requires Python 3.x and g++)

- https://github.com/undefx/vecpy
- Clone, import, vectorize - no setup needed