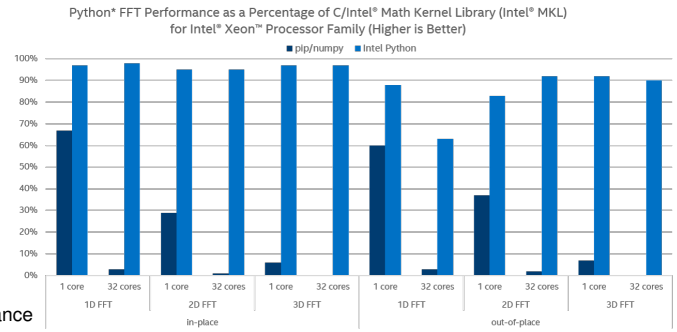
Accelerating Scientific Python with Intel Optimizations

Oleksandr Pavlyk‡*, Denis Nagorny‡†, Andres Guzman-Ballen‡†, Anton Malakhov‡†, Hai Liu‡†, Ehsan Totoni‡†, Todd A. Anderson‡†, Sergey Maidanov‡†

Abstract—It is well-known that the performance difference between Python and basic C code can be up 200x, but for numerically intensive code another speed-up factor of 240x or even greater is possible. The performance comes from software's ability to take advantage of CPU's multiple cores, single instruction multiple data (SIMD) instructions, and high performance caches. The article describes optimizations, included in Intel® Distribution for Python*, aimed to automatically boost performance of numerically intensive code. This paper is intended for Python programmers who want to get the most out of their hardware but do not have time or expertise to re-code their applications using techniques such as native extensions or Cython.



Index Terms—numpy,scipy,scikit-learn,numba,simd,parallel,optimization,performance

Introduction

Scientific software is usually algorithmically rich and compute intensive. The expressiveness of Python language as well as abundance of quality packages offering implementations of advanced algorithms allow scientists and engineers alike to code their software in Python. The ability of this software to solve realistic problems in a reasonable time is often hampered by inefficient use of hardware resources. Intel Distribution for Python [IDP] attempts to enable scientific Python community with optimized computational packages, such as NumPy*, SciPy*, Scikit-learn*, Numba* and PyDAAL across a range of Intel® processors, from Intel® Core™ CPUs to Intel® Xeon® and Intel® Xeon Phi™ processors. This paper offers a detailed report about optimization that went into the Intel® Distribution for Python*, which might be interesting for developers of SciPy tools.

Fast Fourier Transforms

Intel® Distribution for Python* offers a thin layered interface for the Intel® Math Kernel Library (Intel® MKL) that allows efficient access to native FFT optimizations from a range of NumPy and SciPy functions. The optimizations are provided for real and complex data types in both single and double precision. Update 2 improves performance of both one-dimensional and multi-dimensional transforms, for in-place and out-of-place modes of operation. As a result, Python performance may improve up to 60x over Update 1 and is now close to performance of native C/Intel MKL.

Thanks to Intel® MKL's flexibility in its supports for arbitrarily strided input and output arrays[1] both one-dimensional and multi-dimensional complex Fast Fourier Transforms along distinct axes can be performed directly, without the need to copy the input into a contiguous array first (the cost of copying, whose complexity is $\mathcal{O}(n)$, is not negligible compared to the cost of computing the transform, whose complexity is $\mathcal{O}(n \log n)$, and copying, being memory bound, does not scale well with the number of available cores). Furthermore, input strides can be arbitrary, including negative or zero, as long strides remain an integer multiple of array's item size, otherwise a copy will be made.

The wrapper supports both in-place and out-of-place modes, enabling it to efficiently power both `numpy.fft` and `scipy.fftpack` submodules. In-place operations are only performed where possible.

Direct support for multivariate transforms along distinct array axis. Even when multivariate transform ends up being computed as iterations of one-dimensional transforms, all subsequent iterations are performed in place for efficiency.

The update also provides dedicated support for complex FFTs on real inputs, such as `np.fft.fft(real_array)`, by leveraging corresponding functionality in MKL[2].

Dedicated support for specialized real FFTs, which only store independent complex harmonics. Both `numpy.fft.rfft` and `scipy.fftpack.rfft` storage modes are natively supported via Intel® MKL.

* Corresponding author: Oleksandr.Pavlyk@intel.com
‡ Intel Corporation
† These authors contributed equally.

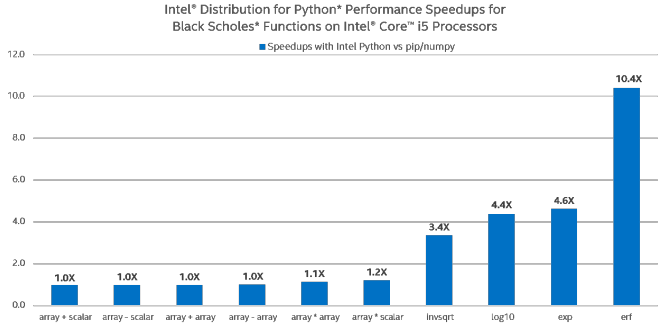
1. https://software.intel.com/en-us/mkl-developer-reference-c-dfti-input-strides-dfti-output-strides#10859C1F-7C96-4034-8E66-B671CE789AD6
2. https://software.intel.com/en-us/mkl-developer-reference-c-dfti-complex-storage-dfti-real-storage-dfti-conjugate-even-storage#CONJUGATE_EVEN_STORAGE

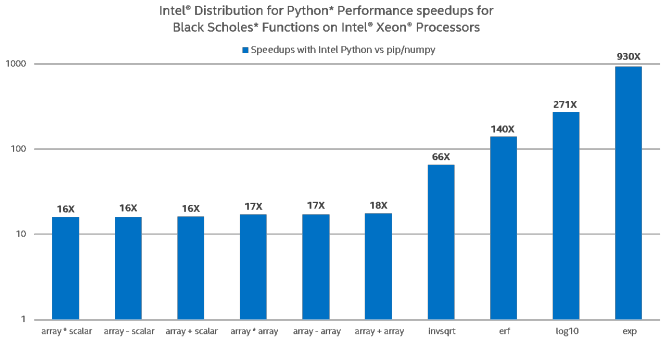
command	`fft(arg)`	`fft(arg, axis=0)`	`fft2(arg)`	`fftn(arg)`
`arg.shape`	$(3 \cdot 10^6,)$	*(1860, 1420)*	*(275, 274, 273)*	*(275, 274, 273)*
`arg.strides`	$(10 \cdot 16,)$	C-contiguous	F-contiguous	$(16, 274 \cdot 275 \cdot 16, 275 \cdot 16)$
repetitions	16	16	8	8
IDP 2017.0.3	0.162±0.01	0.113±0.01	8.87±0.08	0.86±0.01
IDP 2017.0.1	0.187±0.06	1.046±0.03	10.3±0.1	12.38±0.03
pip numpy	2.333±0.01	1.769±0.02	29.94±0.03	34.455±0.007

TABLE 1: *Table of total times of repeated executions of FFT computations using `np.fft` functions for arrays of complex doubles in different Python distributions on Intel (R) Xeon (R) E5-2698 v3 @ 2.30GHz with 64GB of RAM.*

command		`fft(arg)`	`fft(arg)`	`fft2(arg)`	`fft2(arg)`	`fftn(arg)`	`fftn(arg)`
`overwrite_x`		False	True	False	True	False	True
`arg.shape`		$(3 \cdot 10^6,)$	$(3 \cdot 10^6,)$	*(1860, 1420)*	*(1860, 1420)*	*(273, 274, 275)*	*(273, 274, 275)*
IDP 2017.0.3	cd	1.40±0.02	0.885±0.005	0.090±0.001	0.067±0.001	0.868±0.007	0.761±0.001
	cs	0.734±0.004	0.450±0.002	0.056±0.001	0.041±0.0002	0.326±0.003	0.285±0.002
IDP 2017.0.1	cd	1.77±0.02	1.760±0.012	2.208±0.004	2.219±0.002	22.77±0.38	22.7±0.5
	cs	5.79±0.14	5.75±0.02	1.996±0.1	2.258±0.001	27.12±0.05	26.8±0.25
pip numpy	cd	26.06±0.01	23.51±0.01	4.786±0.002	3.800±0.003	67.69±0.12	81.46±0.01
	cs	28.4±0.1	11.9±0.05	5.010±0.003	3.77±0.02	69.49±0.02	80.54±0.07

TABLE 2: *Table of times of repeated execution of `scipy.fftpack` functions with `overwrite_x=True` (in-place) and `overwrite_x=False` (out-of-place) on a C-contiguous arrays of complex double and complex singles.*

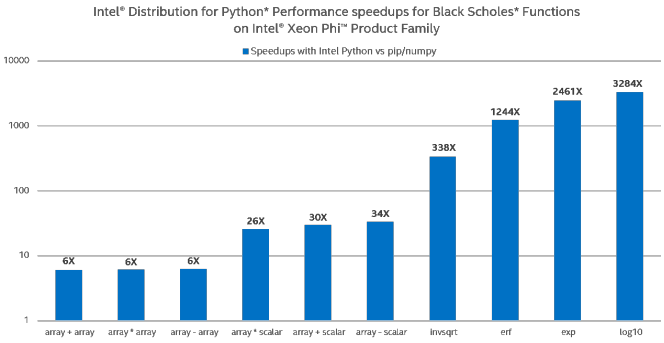




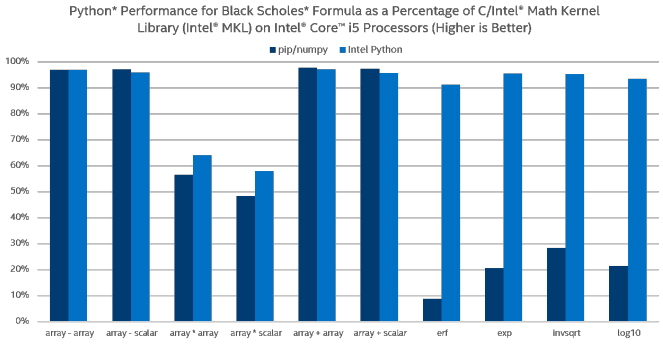
Arithmetic and transcendental expressions

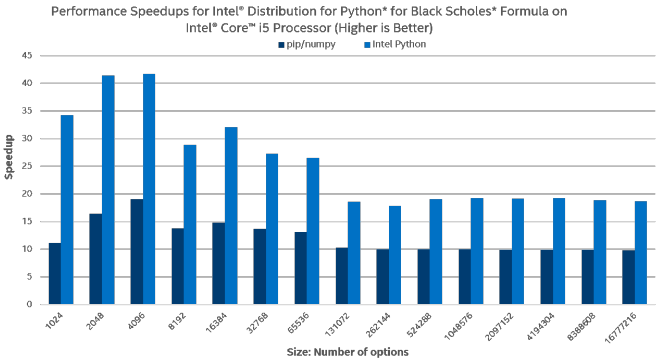
One of the great benefits of the Intel® Distribution for Python* is the performance boost gained from leveraging SIMD and multithreading in (select) NumPy's UMath arithmetic and transcendental operations across the range of Intel® CPUs, from Intel® Core™ to Intel® Xeon™ & Intel® Xeon Phi™. With stock Python as our baseline, we demonstrate the scalability of Intel® Distribution for Python* by using functions that are intensively used in financial math applications and machine learning:

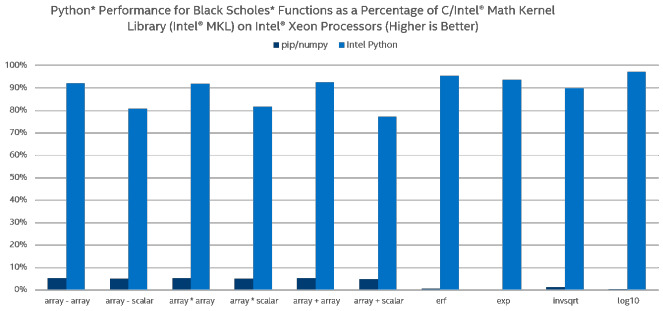
One can see that stock Python (pip-installed NumPy from PyPI) on Intel® Core™ i5 performs basic operations such as addition, subtraction, and multiplication just as well as Intel® Python, but not on Intel® Xeon™ and Intel® Xeon Phi™, where Intel® Distribution for Python* provides over 10x speedup. This can be explained by the fact that basic arithmetic operations in stock NumPy are hard-coded AVX intrinsics (and thus already leverage SIMD, but do not scale to other instruction set architectures (ISA), e.g. AVX-512). These operations in stock Python also do not leverage multiple cores (i.e. no multi-threading of loops under the hood of NumPy exist with such operations). Intel Python's implementation allows for this scalability by utilizing

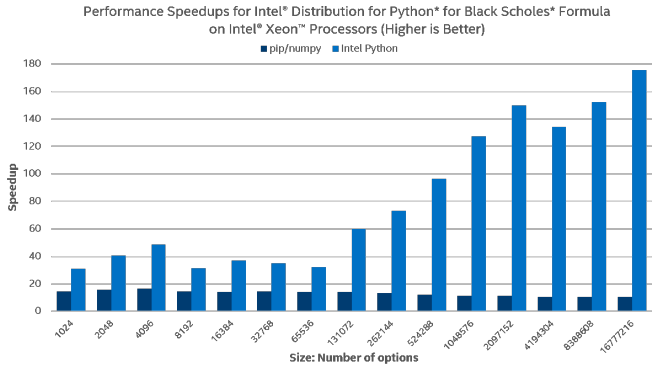


both respective Intel® MKL VML CPU-dispatched and multi-threaded primitives under the hood, and Intel® SVML intrinsics - a compiler-provided short vector math library that vectorizes math functions for both IA-32 and Intel® 64-bit architectures on supported operating systems. Depending on the problem size, NumPy will choose one of the two approaches. On small array sizes, Intel® SVML outperforms VML due to high library call overhead, but for larger problem sizes, VML's ability to both vectorize math









functions and multi-thread loops offsets the overhead.

Specifically, on Intel® Core™ i5 processor the Intel® Distribution for Python delivers greater performance in numerical evaluation of transcendental functions (log, exp, erf, etc.) due to utilization of both SIMD and multi-threading. We do not see any visible benefit of multi-threading basic operations (as shown on the graph) unless NumPy arrays are very large (not shown on the graph). On Intel® Xeon™ processor, the 10x-1000x boost is explained by leveraging both (a) AVX2 instructions to evaluate transcendentals and (b) multiple cores (32 in our setup). Even greater scalability of Intel® Xeon Phi™ relative to Intel® Xeon™ is explained by larger number of cores (64 in our setup) and wider vector registers.

The following charts provide another view of Intel® Distribution for Python performance versus stock Python on arithmetic and transcendental vector operations in NumPy by measuring how close UMath performance is to the respective native MKL call:

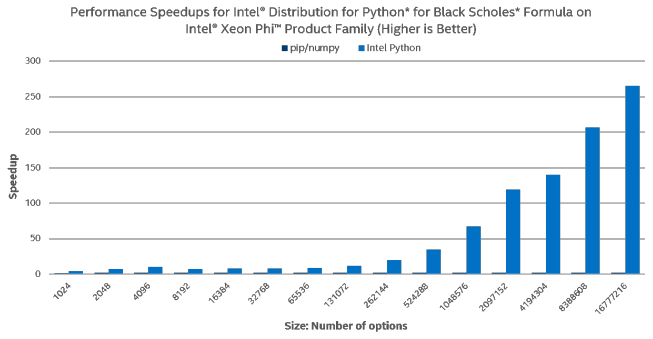
Again on Intel® Core™ i5 the stock Python performs well on basic operations (due to hard-coded AVX intrinsics and because multi-threading from Intel® Distribution for Python does not add much on basic operations) but does not scale on transcendentals (loops with transcendentals are not vectorized in stock Python). Intel® Distribution for Python delivers performance close to native speeds (90% of MKL) on relatively big problem sizes.

To demonstrate the benefits of vectorization and multi-threading in a real-world application, we chose to use the Black Scholes model, used to estimate the price of financial derivatives, specifically European vanilla stock options. A Python implementation of the Black Scholes formula gives an idea of how NumPy UMath optimizations can be noticed at the application level:

One can see that on Intel® Core™ i5 the Black Scholes Formula scales nicely with Intel Python on small problem sizes but does not perform well on bigger problem sizes, which is explained by small cache sizes. Stock Python does marginally scale due to leveraging AVX instructions on basic arithmetic operations, but it is a whole different story on Intel® Xeon™ and Intel® Xeon Phi™. Using Intel® Distribution for Python to execute the same Python code on server processors, much greater scalability on much greater problem sizes is observed. Intel® Xeon Phi™ scales better due to bigger number of cores and as expected, while the stock Python does not scale on server processors due to the lack of AVX2/AVX-512 support for transcendentals and no utilization of multiple cores.

Memory management optimizations

Update 2 introduces extensive optimizations in NumPy memory management operations. As a dynamic language, Python manages memory for the user. Memory operations, such as allocation, deallocation, copy, and move, affect performance of essentially all Python programs. Specifically, Update 2 ensures NumPy allocates arrays that are properly aligned in memory (their address is a multiple of a specific factor, usually 64) on Linux, so that NumPy and SciPy compute functions can benefit from respective aligned versions of SIMD memory access instructions. This is especially relevant for Intel® Xeon Phi™ processors. The most significant improvements in memory optimizations in Update 2



comes from replacing original memory copy and move operations with optimized implementations from Intel® MKL. The result: improved performance because these Intel® MKL routines are optimized for both a range of Intel® CPUs and multiple CPU cores.

Faster Machine Learning with Scikit-learn

Scikit-learn is well-known library that provides a lot of algorithms for many areas of machine learning. Having limited developer resources this project prefers universal solutions and proven algorithms. For performance improvement scikit-learn uses Cython and underlying BLAS/LAPACK libraries through SciPy and Numpy. OpenBLAS and MKL uses threaded based parallelism to utilize multicores of modern CPUs. Unfortunately BLAS/LAPACK's functions are too low level primitives and their usage is often not very efficient comparing to possible high-level parallelism. For high-level parallelism scikit-learn uses multiprocessing approach that is not very efficient from technical point of view. On the other hand Intel provides Intel® Data Analytics Acceleration Library (Intel® DAAL) that helps speed up big data analysis by providing highly optimized algorithmic building blocks for all stages of data analytics (preprocessing, transformation, analysis, modeling, validation, and decision making) in batch, online, and distributed processing modes of computation. It is originally written in C++ and provides Java and Python bindings. DAAL is heavily optimized for all Intel® Architectures including Intel® Xeon Phi™, but it is not at all clear how to use DAAL binding from Python. DAAL bindings for python are generated automatically and reflects original C++ API very closely. This makes its usage quite complicated because of its use of non pythonic idioms and scarce documentation.

In order to combine the power of well optimized native code with the familiar to machine learning community API the Intel Distribution for Python includes fruits of efforts of scikit-learn optimization. Thus beginning with version 2017.0.2 the Intel Distribution for Python includes scikit-learn with daal4sklearn submodule. Specifically, daal4sklearn optimizes Principal Component Analysis (PCA), Linear and Ridge Regressions, Correlation and Cosine Distances, and K-Means in scikit-learn using Intel® DAAL. Speedups may range from 1.5x to 160x.

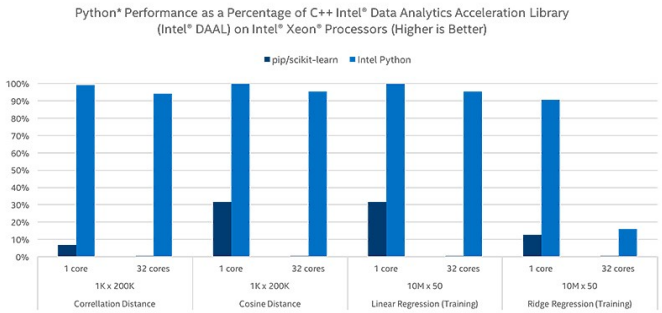
There is no direct matching between scikit-learn's and Intel® DAAL's APIs. Moreover, they aren't fully compatible for all inputs, therefore in those cases where daal4sklearn detects incompatibility it falls back to original sklearn's implementation.

Scikit-learn uses multiprocessing approach to parallelize computations. The unfortunate consequence of this choice may be a large memory footprint as each cloned process has access to its own copy of all input data. This precludes scikit-learn from effectivly utilizing many-cores architectures as Intel® Xeon Phi™ for big workloads. On the other hand DAAL internally uses multi-threading approach sharing the same data across all cores. This allows to DAAL to use less memory and to process bigger workloads which especially important for ML algorithms.

Daal4sklearn is enabled by default and provides a simple API to toggle these optimizations:

```
from sklearn.daal4sklearn import dispatcher
dispatcher.disable()
dispatcher.enable()
```

Several benchmarks [sklearn_benches] were prepared to demonstrate performance that can be achieved with Intel® DAAL. A



fragment from the benchmark used to measure performance of K-means is given below.

```
problem_sizes = [
        (10000,  2),  (10000,  25), (10000,  50),
        (50000,  2),  (50000,  25), (50000,  50),
        (100000, 2),  (100000, 25), (100000, 50)]
X={}
for rows, cols in problem_sizes:
    X[(rows, cols)] = rand(rows, cols)

kmeans = KMeans(n_clusters=10, n_jobs=args.proc)

@st_time
def train(X):
    kmeans.fit(X)

for rows, cols in problem_sizes:
    print (rows, cols, end=' ')
    X_local = X[(rows, cols)]
    train(X_local)
    print('')
```

Using all 32 cores of Intel® Xeon® processor E5-2698 v3 IDP's K-Means can be more than 50 times faster than the python included with Ubuntu 14.04. P below means the number of CPU cores used.

We compared the similar runs for other algorithms and normalized results by results obtained with DAAL in C++ without python to estimate overhead from python wrapping.

You can find some benchmarks [sklearn_benches]

Numba vectorization

Wikipedia defines SIMD as:

> Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use.

To utilize power of CPU's SIMD instructions compilers need to implement special optimization passes, so-called code vectorization. Modern optimizing compilers implement automatic vectorization - a special case of automatic parallelization, where a computer program is converted from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation, which processes a single operation on multiple pairs of operands at once.

rows	cols	IDP,s P=1	IDP,s P=32	System,s P=1	System,s P=32	Vs System,P=1	Vs System,P=32
10000	2	0.01	0.01	0.38	0.27	28.55	36.52
10000	25	0.05	0.01	1.46	0.57	27.59	48.22
10000	50	0.09	0.02	2.21	0.87	23.83	40.76
50000	2	0.08	0.01	1.62	0.57	20.57	47.43
50000	25	0.67	0.07	14.43	2.79	21.47	38.69
50000	50	1.05	0.10	24.04	4.00	22.89	38.52
100000	2	0.15	0.02	3.33	0.87	22.30	56.72
100000	25	1.34	0.11	33.27	5.53	24.75	49.07
100000	50	2.21	0.17	63.30	8.36	28.65	47.95

TABLE 3

According Numba's project page Numba is an Open Source NumPy-aware optimizing compiler for Python. It uses the remarkable LLVM compiler infrastructure to compile Python syntax to machine code. And it is quite expected that Numba tries to use all these features to improve performance especially for scientific applications.

LLVM implemented auto-vectorization for simple cases several years ago but there remain sigificant problems with vectorization of elementary transcendental math functions. To enable proper vectorization support a special vectorized implementation of math functions such as `sin`, `cos`, `exp` is needed.

The Intel® C++ Compiler provides short vector math library (SVML) intrinsics implementing vectorized mathematical functions. These intrinsics are available for IA-32 and Intel® 64 architectures running on supported operating systems.

The SVML intrinsics are vector variants of corresponding scalar math operations using `__m128`, `__m128d`, `__m256`, `__m256d`, and `__m256i` data types. They take packed vector arguments, simultaneously perform the operation on each element of the packed vector argument, and return a packed vector result. Due to low overhead of the packing for aligned contiguously laid out data, vector operations may offer speed-ups over scalar operations which are proportional to the width of the vector register.

For example, the argument to the `_mm_sin_ps` intrinsic is a packed 128-bit vector of four 32-bit precision floating point numbers. The intrinsic simultaneously computes values of the sine function for each of these four numbers and returns the four results in a packed 128-bit vector, all within about the time of scalar evaluation of only one argument.

Using SVML intrinsics is faster than repeatedly calling the scalar math functions. However, the intrinsics may differ from the corresponding scalar functions in accuracy of their results.

Besides intrinsics available with Intel® compiler there is opportunity to call vectorized implementations directly from svml library by their names.

Beginning with version 4.0 LLVM features (experimental) model of autovectorization using SVML library, so a full stack of technologies is now available to exploit in-core parallelization of python code. To enable the autovectorization feature in Numba, included in the Intel® Distribution for Python*, user needs to set `NUMBA_INTEL_SVML` environmental variable to a non-zero value, prompting Numba to load SVML library and to pass an appropriate option to LLVM.

Let's see how it works with a small example:

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

def foo(x,y):
    for i in range(x.size):
        y[i] = math.sin(x[i])
foo_compiled = njit(foo)
```

Inspite of the fact that numba generates call for usual `sin` function, as seen in the following excerpt from the generated LLVM code:

```
label 16:
    $16.2 = iternext(value=$phi16.1)     ['$16.2',
                                          '$phi16.1']
    $16.3 = pair_first(value=$16.2)      ['$16.2',
                                          '$16.3']
    $16.4 = pair_second(value=$16.2)     ['$16.2',
                                          '$16.4']
    del $16.2                            []
    $phi19.1 = $16.3                     ['$16.3',
                                          '$phi19.1']
    del $16.3                            []
    branch $16.4, 19, 48                 ['$16.4']
label 19:
    del $16.4                            []
    i = $phi19.1                         ['$phi19.1',
                                          'i']
    del $phi19.1                         []
    $19.2 = global(math: <module 'math'\
     from '/path_stripped/lib-dynload/\
     math.cpython-35m-x86_64-...,.so'>) ['$ 19.2']
    $19.3 = getattr(attr=sin,
                    value=$19.2)         ['$19.2',
                                          '$19.3']
    del $19.2                            []
    $19.6 = getitem(index=i, value=x)    ['$19.6',
                                          'i', 'x']
    $19.7 = call $19.3($19.6)            ['$19.3',
                                          '$19.6',
                                          '$19.7']
    del $19.6                            []
    del $19.3                            []
    y[i] = $19.7                         ['$19.7',
                                          'i', 'y']
    del i                                []
    del $19.7                            []
    jump 16                              []
```

We can see direct use of the SVML-provided vector implementation of sine function:

```
leaq    96(%rdx), %r14
leaq    96(%rsi), %r15
movabsq $__svml_sin4_ha, %rbp
movq    %rbx, %r13
.p2align        4, 0x90
```

```
.LBB0_13:
        vmovups -96(%r14), %ymm0
        vmovups -64(%r14), %ymm1
        vmovups %ymm1, 32(%rsp)
        vmovups -32(%r14), %ymm1
        vmovups %ymm1, 64(%rsp)
        vmovups (%r14), %ymm1
        vmovups %ymm1, 128(%rsp)
        callq   *%rbp
        vmovups %ymm0, 96(%rsp)
        vmovups 32(%rsp), %ymm0
        callq   *%rbp
        vmovups %ymm0, 32(%rsp)
        vmovups 64(%rsp), %ymm0
        callq   *%rbp
        vmovups %ymm0, 64(%rsp)
        vmovupd 128(%rsp), %ymm0
        callq   *%rbp
        vmovups 96(%rsp), %ymm1
        vmovups %ymm1, -96(%r15)
        vmovups 32(%rsp), %ymm1
        vmovups %ymm1, -64(%r15)
        vmovups 64(%rsp), %ymm1
        vmovups %ymm1, -32(%r15)
        vmovupd %ymm0, (%r15)
        subq    $-128, %r14
        subq    $-128, %r15
        addq    $-16, %r13
        jne     .LBB0_13
```

Thanks to enabled support of high accuracy SVML functions in LLVM this jitted code sees more than 4x increase in performance.

svml enabled:

```
%timeit foo_compiled(x,y)
1000 loops, best of 3: 403 us per loop
```

svml disabled:

```
%timeit foo_compiled(x,y)
1000 loops, best of 3: 1.72 ms per loop
```

Auto-parallelization for Numba

In this section, we introduce a new feature in Numba that automatically parallelizes NumPy programs. Achieving high performance with Python on modern multi-core CPUs is challenging since Python implementations are generally interpreted and prohibit parallelism. To speed up sequential execution, Python functions can be compiled to native code using Numba, implemented with the LLVM just-in-time (JIT) compiler. All a programmer has to do to use Numba is to annotate their functions with Numba's `@jit` decorator. However, the Numba JIT will not parallelize NumPy functions, even though the majority of them are known to have parallel semantics, and thus cannot make use of multiple cores. Furthermore, even if individual NumPy functions were parallelized, a program containing many such functions would likely have lackluster performance due to poor cache behavior. Numba's existing solution is to allow users to write scalar kernels in OpenCL style, which can be executed in parallel. However, this approach requires significant programming effort to rewrite existing array code into explicit parallelizable scalar kernels and therefore hurts productivity and may be beyond the capabilities of some programmers. To achieve both high performance and high programmer productivity, we have implemented an automatic parallelization feature as part of the Numba JIT compiler. With auto-parallelization turned on, Numba attempts to identify operations with parallel semantics and to fuse adjacent ones together to form kernels that are automatically run in parallel, all fully automated without manual effort from the user.

Our implementation supports the following parallel operations:

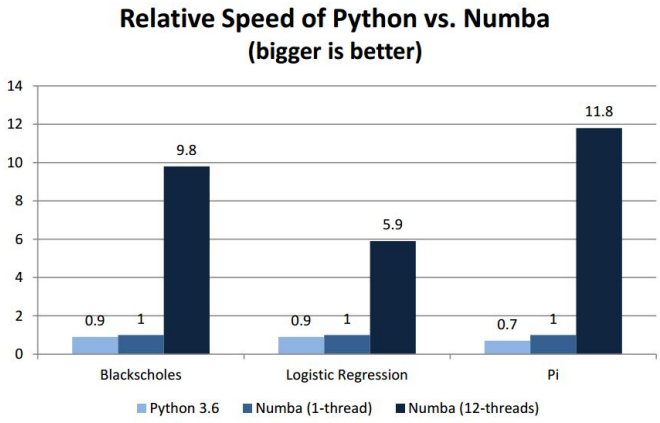
1) Common arithmetic functions between NumPy arrays, and between arrays and scalars, as well as NumPy ufuncs. They are often called *element-wise* or *point-wise* array operations:
   - unary operators: `+ - ~`
   - binary operators: `+ - * / /? % | >> ^ << & ** //`
   - comparison operators: `== != < <= > >=`
   - NumPy ufuncs that are supported in Numba's nopython mode.
   - User defined *DUFunc* through `@vectorize`.
2) NumPy reduction functions `sum` and `prod`, although they have to be written as `numpy.sum(a)` instead of `a.sum()`.
3) NumPy `dot` function between a matrix and a vector, or two vectors. In all other cases, Numba's default implementation is used.
4) Multi-dimensional arrays are also supported for the above operations when operands have matching dimension and size. The full semantics of NumPy broadcast between arrays with mixed dimensionality or size is not supported, nor is the reduction across a selected dimension.
5) NumPy `array` created from list comprehension is turned into direct array allocation and initialization without intermediate list.
6) Explicit parallelization via `prange` that turns a for-loop into a parallel loop.

As an example, consider the following Logistic Regression function:

```
@jit(parallel=True)
def logistic_regression(Y, X, w, iters):
    for i in range(iters):
        w += np.dot(
               Y / (1.0 + np.exp(Y * np.dot(X, w))),
               X )
    return w
```

We will not discuss details of the algorithm, but instead focus on how this program behaves with auto-parallelization:

1) Input `Y` is a vector of size `N`, `X` is an `N x D` matrix, and `w` is a vector of size `D`.
2) The function body is an iterative loop that updates variable `w`. The loop body consists of a sequence of vector and matrix operations.
3) The inner `dot` operation produces a vector of size `N`, followed by a sequence of arithmetic operations either between a scalar and vector of size `N`, or two vectors both of size `N`.
4) The outer `dot` produces a vector of size `D`, followed by an inplace array addition on variable `w`.
5) With auto-parallelization, all operations that produce array of size `N` are fused together to become a single parallel kernel. This includes the inner `dot` operation and all point-wise array operations following it.
6) The outer `dot` operation produces a result array of different dimension, and is not fused with the above kernel.



Here, the only thing required to take advantage of parallel hardware is to set the `parallel=True` option for `@jit`, with no modifications to the `logistic_regression` function itself. If we were to give an equivalent parallel implementation using Numba's `@guvectorize` decorator, it would require a pervasive change that rewrites the code to extract kernel computation that can be parallelized, which is both tedious and challenging.

We measure the performance of automatic parallelization over three workloads, comparing auto-parallelization with Numba's sequential JIT and Python 3.6, normalized to the sequential (1-thread) speed of Numba.

Auto-parallelization proves to be an effective optimization for these benchmarks, achieving speedups from 5.9x to 11.8x over sequential Numba on 12-core Intel® Xeon® X5680 @3.33GHz with 64GB RAM. The benchmarks are available as part of Numba's source distribution [numba].

Our future plan is to support array range selection, enable auto-parallelization of more NumPy functions, as well as to add new features such as iterative stencils. We also plan to implement more optimizations that help make parallel programs run fast, improving both performance and productivity for Python programmers in the scientific domain.

Summary

The Intel® Distribution for Python is powered by Anaconda* and conda build infrastructures that give all Python users the benefit of interoperability within these two environments and access to the optimized packages through a simple `conda install` command. Intel® Distribution for Python* delivers significant performance optimizations for many core algorithms and Python packages, while maintaining the ease of downloading and installation.

REFERENCES

[fft_bench] http://github.com/intelpython/fft_benchmark
[sklearn_benches] https://github.com/dvnagorny/sklearn_benchs
[numba] https://github.com/numba/numba
[IDP] Intel |R| Distribution for Python*