Speeding up numerical calculations in Python[*]

A.A. Fedotov, V.N. Litvinov, A.F. Melik-Adamyan

Intel Corporation

This article describes the tools, techniques and optimizations that Intel brings to the Python[*] developers community. Intel develops high performance libraries and profilers as well as extends support for multi-core and SIMD parallelism across Python toolchain so that developers can achieve near native performance in Python, avoiding the need to rewrite in C/C++. This article will demonstrate tools and libraries applied to manage popular real world problem.

Keywords: python, numerical calculations, performance.

1. Introduction

In this article we will describe the tools, techniques and optimizations that Intel brings to the Python* developers community. We are developing high performance libraries and profilers as well as extending support for multi-core and SIMD parallelism across Python toolchain so that developers can achieve near native performance in Python, avoiding the need to rewrite in C/C++. This will be illustrated on a popular real world problem.

2. Problem statement

2.1 Popularity of Python language

Studies by several companies show that Python is among the most demanded languages. Codeval.com studies from year of 2016 demonstrate that Python is the number one programming language in hiring demand followed by Java[*] and C++ (see Fig. 1) [1]. In addition, Python language offers great productivity to developers due to its expressiveness and ease of coding (see Fig. 2). [2, 3]

Python use continues to grow in many domains that require interactive prototyping. Quants develop trading algorithms, data scientists build analytics models, and researchers prototype numerical simulations.

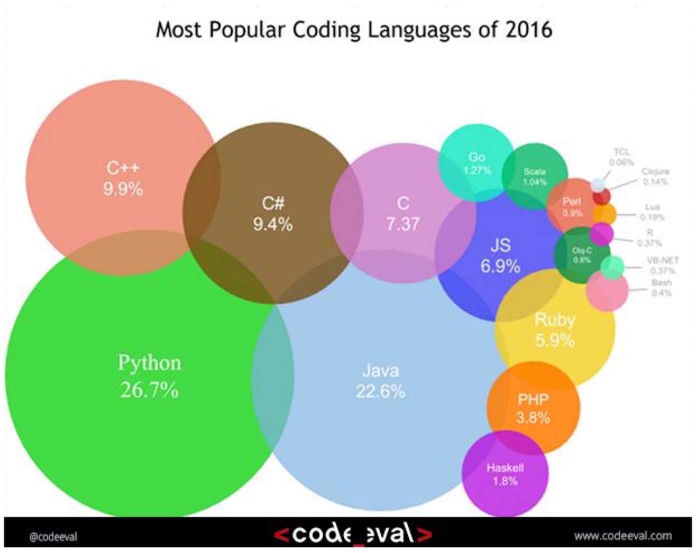


Fig. 1. Most popular coding languages as of February 2, 2016 by www.codeeval.com

[*] Other names and brands may be claimed as the property of others.

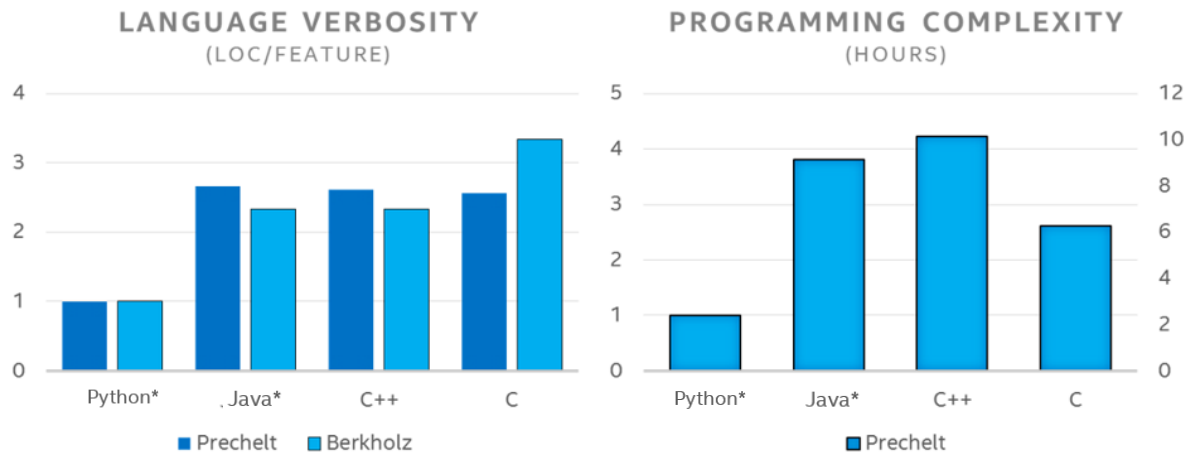


Fig. 2. Programming languages productivity. [2, 3]

2.2. Python performance problems

All too often, scaling the prototype code to production means a developer recoding the algorithm in a language such as C++ or Java. This is primarily done due to Python's performance issues and scalability problems (one cannot write effective parallel Python code because of infamous GIL – Global Interpreter Lock). Rewriting takes time, reduces flexibility, and can lead to errors.

Lots of performance problems are caused by Python interpreter's implementation being unfriendly to modern hardware – as interpreter needs to run on all sorts of hardware platforms it cannot effectively use new CPU features like Intel® Streaming SIMD Extensions (Intel® SSE) or Intel® Advanced Vector Extensions (Intel® AVX).

In addition, Python implementation was not made with cache-locality in mind: consecutive data is not stored in a sequential way; data access requires a lot of pointer dereferencing, etc. All this significantly impedes Python interpreter achieving good performance on modern hardware platforms.

3. Known Solutions for Python problems

Performance problems mentioned in previous section are usually addressed by using C extensions like NumPy, SciPy, Scikit-learn*. Such libraries help fully utilize hardware at the same time allowing Python developer to express his/her thoughts in a clear, concise way.

However, standard implementations of such libraries are too generic and do not utilize hardware capabilities to the maximum, and typically do not do vectorizing and usually are single-threaded while modern hardware is usually multi-core or even multi-socket machine [4]. Considering this, it is developer's responsibility to ensure that his/her application utilizes all CPU resources while it is better to be done by the library since algorithm parallelization is not a simple task and requires a lot of expertise in parallel programming, which typical Python developer does not have and does not even need to have.

Also note that it's not a no-brainer task to replace all the performance-critical code with libraries because the developer needs to know which part of his/her code actually is performance-critical. To understand that it might not be enough to know the codebase, or the codebase might be so big that it is impossible for a person to analyze it through analytically. Thus one needs tools to help one analyzing the code, and this article is going to talk about profiling, which usually highlights slow parts of the code.

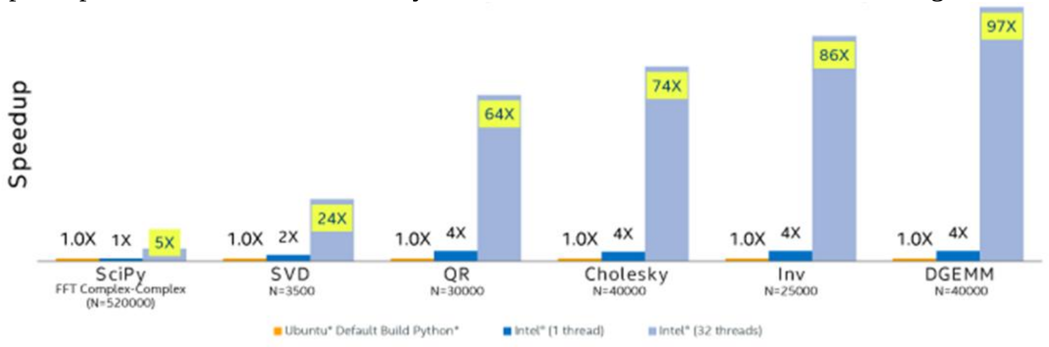
There are a number of profilers available: cProfile module which is included in standard library, profiler included in Python Tools for Microsoft* Visual Studio*, line_profiler package available on PyPI and so on. The problem with most Python profilers is that they usually have quite a big overhead (slowing down application being profiled) and most of them don't provide line-level source information.

4. What Intel brings to Python community

Intel has a bunch of well-known libraries used for building high-performance application like Intel® Math Kernel Library (Intel® MKL), Intel® Threading Building Blocks (Intel® TBB), implementation of OpenMP* standard, implementation of MPI standard, Intel® Data Analytics Acceleration Library (Intel® DAAL), etc., and profiling tools like Intel® VTune™ Amplifier, but until recently those were mostly accessible and usable for C/C++/Fortran developers.

Now Intel brings the power of these instruments to the Python developers community by making freely available Intel® Distribution for Python which utilizes Intel MKL and OpenMP to parallelize numerical calculations in extensions like NumPy. In addition, Intel introduces Intel TBB bindings for Python, which could be used to make parallelization of applications easier by dealing with most of the problems, related to that (load balancing, oversubscription, etc.) [5]. Intel DAAL bindings for Python is made available to foster big data analysis done in Python, and Intel VTune now supports profiling mixed Python/C/C++/Fortran applications allowing developers who care about performance to study their applications and see if there are any addressable bottlenecks.

Speedups of Intel Distribution for Python on selected functions are shown on Fig.3.



Configuration Info:
- Fedora* built Python*: Python 2.7.10 (default, Sep 8 2015), NumPy 1.9.2, SciPy 0.14.1, multiprocessing 0.70a1 built with gcc 5.1.1;

Hardware: 96 CPUs (HT ON), 4 sockets (12 cores/socket), 1 NUMA node, Intel(R) Xeon(R) E5-4657L v2@2.40GHz, RAM 64GB

Operating System: Fedora release 23 (Twenty Three)

Fig. 3. Python* Performance Boost on Selected Numerical Functions with Intel® Distribution for Python (2017 beta) vs. Ubuntu Python*.

Also unlike many Python profilers that use high overhead instrumentation which can distort the result, Intel VTune Amplifier uses very low overhead sampling technique that gives accurate enough results and shows line-level information at the same time. It also shows the developer all the code in the application, be in Python, C, C++ or even Fortran, thus not requiring one to use several profilers (each for a language) when optimizing an application written in several languages.

5. Applying libraries and tools to real world example

Consider such a popular problem as construction of a recommendation system. Let us assume that input data for this problem is items purchased and users' ratings for them. Constructed recommendation system should be able to solve two independent tasks: 1) build up a recommendation rule; 2) perform an actual recommendation for a particular user. For the latter we assume that the system is working as a web-service, which could have many user requests in a given time interval to build a recommendation for them.

5.1 Building recommendation rule

There are several types of algorithms for creation of a model for recommender system. In this article we focus on item-based recommendations where the model is represented by items similarity matrix. Each element of the matrix s_{ij} is the measure of similarity between the items i and j. In our implementation we chose cosine distance as a measure of such similarity.

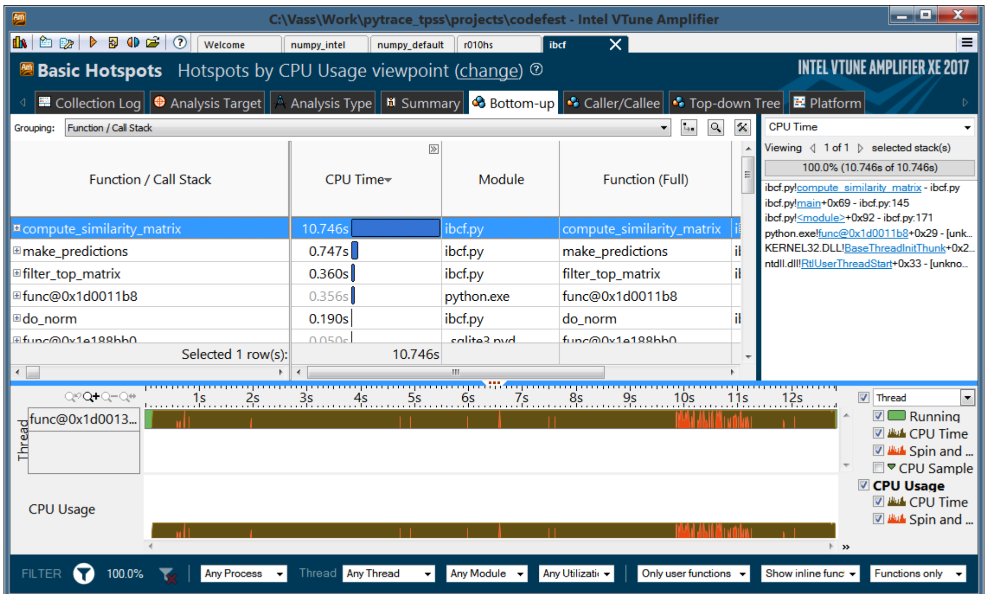
The whole algorithm of model creation consists of the following steps:

1) Load users' ratings from database into the matrix. Rows of the matrix represent different items while columns represent different users.
2) Compute items similarity as a cosine distance:
   a) Normalize the matrix with users' ratings through dividing each row by its Euclidean length.
   b) Compute dot product between the matrix rows.
3) For each item *i* choose *k* most similar items with the largest measure of similarity and set the similarities for all other items to zero.

Here is the code of the step 2b implemented in pure python without additional packages:

```python
def compute_similarity_matrix(matrix):
    items_num, users_num = len(matrix), len(matrix[0])
    cosine_sim_matrix = [ items_num * [0] for i in range( items_num ) ]
    for i in range( items_num ):
        for j in range( items_num ):
            sum = 0
            for k in range( users_num ):
                sum += matrix[i][k] * matrix[j][k]
            cosine_sim_matrix[i][j] = sum
    for i in range( items_num ):
        cosine_sim_matrix[i][i] = 0
    return cosine_sim_matrix
```

For performance measurements of this implementation we used data from http://grouplens.org/, that contains 1,000,000 ratings made by 6,040 users for 3,260 movies. This pure Python implementation took 338 minutes to load data and build similarity matrix. To understand where the main bottleneck come from we used Python profiling feature available in Intel VTune Amplifier XE. To significantly decrease the time needed for profiling we reduced the number of ratings from 1,000,000 to 20,000. The results of the profiling could be seen on Fig. 4 and Fig. 5.


C:\Vass\Work\pytrace_tpss\projects\codefest - Intel VTune Amplifier

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Grouping: Function / Call Stack

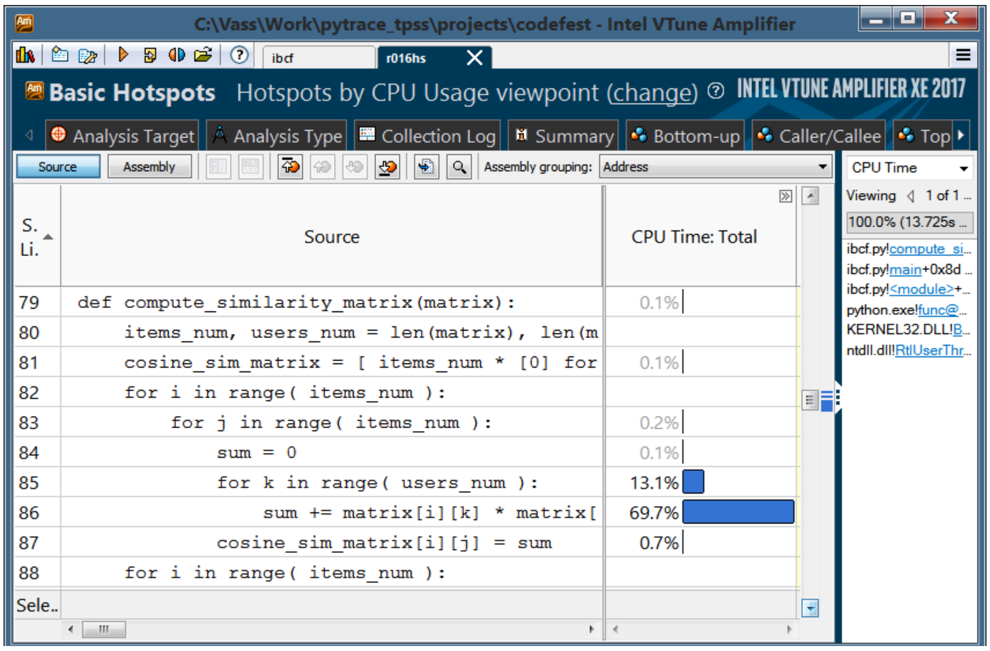
Function / Call Stack	CPU Time	Module	Function (Full)
compute_similarity_matrix	10.746s	ibcf.py	compute_similarity_matrix
make_predictions	0.747s	ibcf.py	make_predictions
filter_top_matrix	0.360s	ibcf.py	filter_top_matrix
func@0x1d0011b8	0.356s	python.exe	func@0x1d0011b8
do_norm	0.190s	ibcf.py	do_norm
func@0x1e188bb0	0.050s	sqlite3.pyd	func@0x1e188bb0
Selected 1 row(s):	10.746s		

CPU Time
Viewing 1 of 1 selected stack(s)
100.0% (10.746s of 10.746s)

```
ibcf.py!compute_similarity_matrix - ibcf.py
ibcf.py!main+0x69 - ibcf.py:145
ibcf.py!<module>+0x92 - ibcf.py:171
python.exe!func@0x1d0011b8+0x29 - [unk...
KERNEL32.DLL!BaseThreadInitThunk+0x2...
ntdll.dll!RtlUserThreadStart+0x33 - [unkno...
```

FILTER 100.0% Any Process Thread Any Thread Any Module Any Utilizati Only user functions Show inline func Functions only

Fig. 4 Bottom-up view of Intel® VTune™ Amplifier XE shows the most compute-intensive part - the computation of items similarity.



S. Li.	Source	CPU Time: Total
79	`def compute_similarity_matrix(matrix):`	0.1%
80	`    items_num, users_num = len(matrix), len(m`	
81	`    cosine_sim_matrix = [ items_num * [0] for`	0.1%
82	`    for i in range( items_num ):`	
83	`        for j in range( items_num ):`	0.2%
84	`            sum = 0`	0.1%
85	`            for k in range( users_num ):`	13.1%
86	`                sum += matrix[i][k] * matrix[`	69.7%
87	`            cosine_sim_matrix[i][j] = sum`	0.7%
88	`    for i in range( items_num ):`	
Sele..		

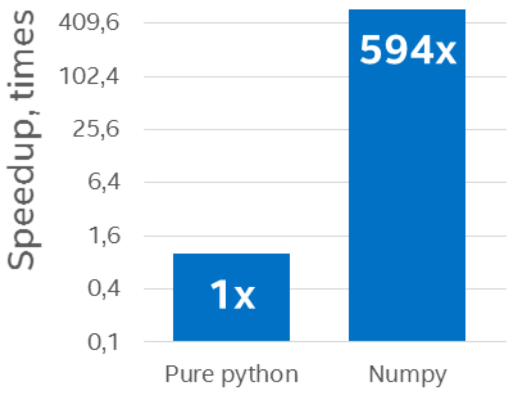
Fig. 5 Intel® VTune™ Amplifier XE shows that the program spent the biggest part of execution time at line 86.

From the profiles we concluded that algorithm spends most of the time in dot product. In Python such operations are not as efficient as in compiled languages like C/C++ or Fortran. This happens because of the reasons listed in the chapter 2.2.

Fortunately, Python has a number of extensions provided as additional packages that help to mitigate if not all but most of the mentioned issues. De facto standard package for linear algebra computations is called Numpy. It introduces additional data type called ndarray, array of an arbitrary dimension represented as a contiguous block in memory. Linear algebra operations are implemented in Numpy via NETLIB, a popular Fortran library.

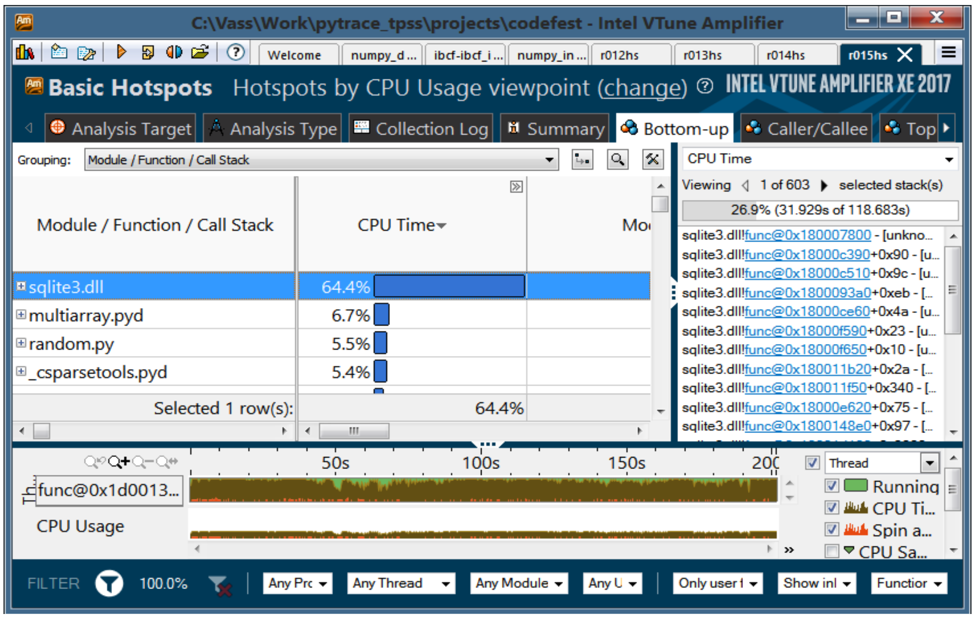
The Fig. 6 shows the performance improvement after changing our initial similarity matrix computations to the following code:

```
import numpy as np
matrix = np.array(shape=(len(items), len(users)))
...
cosine_sim_matrix = matrix.dot(matrix.T)
```



Configuration Info: - Fedora* built Python*: Python 2.7.10 (default, Sep 8 2015), NumPy 1.9.2, SciPy 0.14.1, multiprocessing 0.70a1 built with gcc 5.1.1; Hardware: 96 CPUs (HT ON), 4 sockets (12 cores/socket), 1 NUMA node, Intel(R) Xeon(R) E5-4657L v2@2.40GHz, RAM 64GB, Operating System: Fedora release 23 (Twenty Three)

Fig.6 Performance boost of Numpy computations over the pure Python* implementation



Module / Function / Call Stack	CPU Time
sqlite3.dll	64.4%
multiarray.pyd	6.7%
random.py	5.5%
_csparsetools.pyd	5.4%
Selected 1 row(s):	64.4%

```
sqlite3.dll!func@0x180007800 - [unkno...
sqlite3.dll!func@0x18000c390+0x90 - [u...
sqlite3.dll!func@0x18000c510+0x9c - [u...
sqlite3.dll!func@0x1800093a0+0xeb - [...
sqlite3.dll!func@0x18000ce60+0x4a - [u...
sqlite3.dll!func@0x18000f590+0x23 - [u...
sqlite3.dll!func@0x18000f650+0x10 - [u...
sqlite3.dll!func@0x180011b20+0x2a - [...
sqlite3.dll!func@0x180011f50+0x340 - [...
sqlite3.dll!func@0x18000e620+0x75 - [...
sqlite3.dll!func@0x1800148e0+0x97 - [...
```

Fig. 7 Bottom-up view of the Intel® VTune™ Amplifier XE shows that the application spends 64.4% of the time in data loading from the database

The profile on Fig. 7 shows that new bottleneck is in the data loading step. This makes further optimization of the computational part unreasonable.

5.2. Generation of user recommendations

The second part of the collaborative filtering task is to predict user preferences for not rated items from the ratings that he or she already provided.

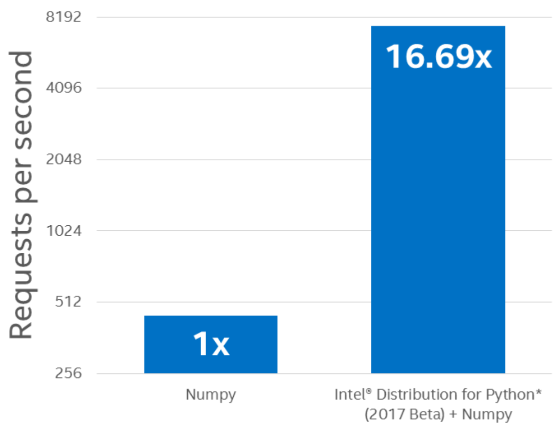
To compute the user preference of the item *i* we multiply *i*-th row of item-item similarity matrix by the vector of user ratings. By doing this for each row of the similarity matrix, except those rows for which the user has already provided the ratings, we predict user's preferences for all items. The resulting preferences vector is sorted to choose top *k* elements.

This code shows how to implement this part of the algorithm using Numpy:

```
x = topk_matrix.dot(user_vector)
quorum = 10
data_ids = np.argsort(x.data[0])[-quorum:][::-1]
```

For the application that works as a web-service task to compute similarity matrix can be performed infrequently (e.g. once a day) as compared to the number of users' requests to get the recommendation (e.g. hundreds of thousands of requests per second, which could arrive at server in parallel). Therefore, the task to generate the recommendation for a particular user could be actually seen as a subtask of a larger task – fulfil several requests.

The Fig. 8 demonstrates the performance advantage of Python interpreter and Numpy package taken from Intel Distribution for Python over Python interpreter and Numpy package taken from Fedora 23 for the task of users' recommendation generation.

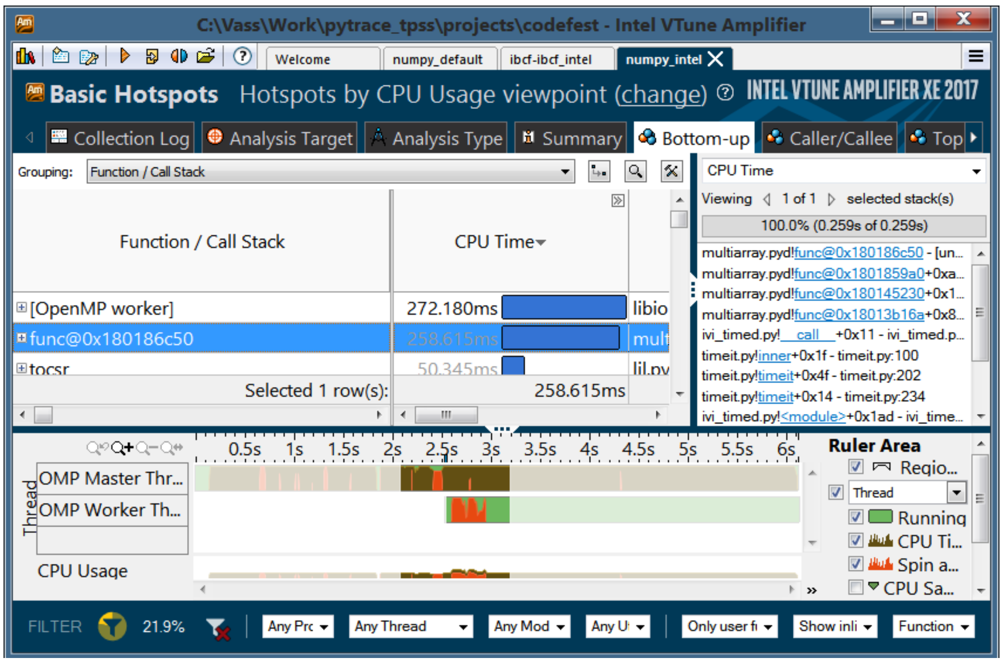


Configuration Info: - Versions: Intel® Distribution for Python 2.7.11 2017, Beta (Mar 04, 2016), Intel® MKL version 11.3.2 for Intel Distribution for Python 2017, Beta , Fedora* built Python*: Python 2.7.10 (default, Sep 8 2015), NumPy 1.9.2, SciPy 0.14.1, multiprocessing 0.70a1 built with gcc 5.1.1; Hardware: 96 CPUs (HT ON), 4 sockets (12 cores/socket), 1 NUMA node, Intel(R) Xeon(R) E5-4657L v2@2.40GHz, RAM 64GB, Operating System: Fedora release 23

Fig. 8 Performance comparison of generating of user recommendations between Python* interpreter and Numpy package from Fedora* 23 and Intel® Distribution For Python* with its own Numpy.

The reasons of such an improvement are:

- Python sources compiled by Intel® C/C++ Compiler.
- Calls to Intel Math Kernel Library instead of calls to standard NETLIB for matrix to vector multiplication.
- Usage of OpenMP library for multithreaded computations (see Fig. 9).



Function / Call Stack	CPU Time	
[OpenMP worker]	272.180ms	libio
func@0x180186c50	258.615ms	mult
tocsr	50.345ms	lil.py
Selected 1 row(s):	258.615ms	

```
100.0% (0.259s of 0.259s)
multiarray.pyd!func@0x180186c50 - [un...
multiarray.pyd!func@0x1801859a0+0xa...
multiarray.pyd!func@0x180145230+0x1...
multiarray.pyd!func@0x18013b16a+0x8...
ivi_timed.py!__call__+0x11 - ivi_timed.p...
timeit.py!inner+0x1f - timeit.py:100
timeit.py!timeit+0x4f - timeit.py:202
timeit.py!timeit+0x14 - timeit.py:234
ivi_timed.py!<module>+0x1ad - ivi_time...
```

Fig. 9 Shows usage of OpenMP* technology for parallelization of users' preferences computations.

It could be presumed that usage of Intel Distribution for Python is unnecessary because one could build Numpy package with MKL-enabled calls oneself. However, it is not an easy task [9, 10, 11, 12].

Another possibility to parallelize computations of user preferences is to use Python's native tools for that. Let us consider ThreadPool class (part of standard library), which could be used in the following way:

```
def process_in_parallel(n, body):
    from multiprocessing.pool import ThreadPool
    global tp_pool, numthreads
    if 'tp_pool' not in globals():
        print "Creating ThreadPool(%s)" % numthreads
```

```
          tp_pool = ThreadPool(int(numthreads))
      tp_pool.map(body, xrange(n))
```

Fig. 10 Usage of ThreadPool class

Where `body` is the unit of parallelizable work. In our case it is a generation of recommendation for a number of users, requests from which have arrived t server about the same time.

"Numpy + ThreadPool" bar from the Fig.11 shows that enabling parallelization using code snippet from Fig. 10 gives relatively good performance boost as compared to use of Intel® Distribution for Python, though requires source code modification to achieve this.

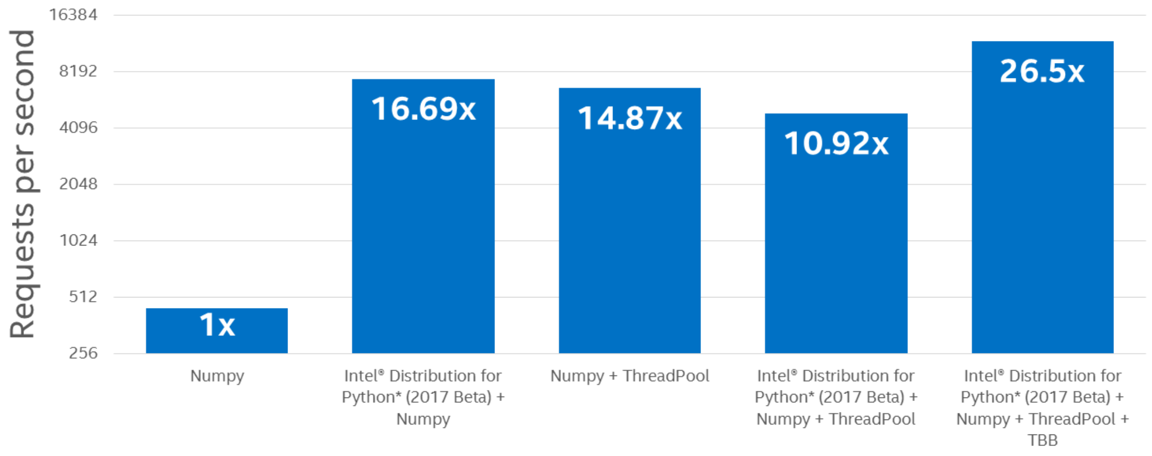
If the same technique is applied to Intel Distribution for Python, which is already parallelized, then performance suddenly degrades as it is shown on "Intel Distribution for Python + Numpy + ThreadPool" bar. This happens because the work is parallelized on two independent levels: one level is ThreadPool and another one is Intel MKL, which is used in each ThreadPool's `body`. Both levels act as if they control all the cores of the machine. In the worst case the number of threads created could be a square of the number of cores and the application would spend most of its time on context switching instead of doing useful work. This is known as oversubscription problem.

To overcome the oversubscription issue we use Intel Threading Building Blocks library because it is known to be good at nested parallelism. Python wrapper of this library replaces standard ThreadPool package, as well as other thread pool implementations popular libraries Dask and Joblib, effectively not requiring modification of the source code of an application. Use this command line switch to enable Python wrapper for Intel TBB library:

```
$ python -m TBB <your_script>.py
```

This will also enable Intel TBB threading layer in Intel MKL.

As a result we got even better speedup for the task (see "Intel Distribution for Python + Numpy + ThreadPool + TBB" bar on Fig.11)



Configuration Info: - Versions: Intel® Distribution for Python* 2.7.11 2017, Beta (Mar 04, 2016), MKL version 11.3.2 for Intel® Distribution for Python* 2017, Beta, Fedora* built Python*: Python 2.7.10 (default, Sep 8 2015), NumPy 1.9.2, SciPy 0.14.1, multiprocessing 0.70a1 built with gcc 5.1.1; Hardware: 96 CPUs (HT ON), 4 sockets (12 cores/socket), 1 NUMA node, Intel(R) Xeon(R) E5-4657L v2@2.40GHz, RAM 64GB, Operating System: Fedora release 23 (Twenty Three)

Fig.11 Comparison of various combinations of usages of different Numpy packages with ThreadPool and Intel® Distribution for Python*

6. Future improvements

Intel Data Analytics Acceleration Library could be used to scale a solution for the mentioned problem onto computer clusters and distributed systems. Its Python bindings present another algorithm for constructing recommendation systems called alternating least squares (ALS), which is known to be more adjustable and efficient than the algorithm based on items similarities.

7. Conclusion

For a long period scripting languages like Python were considered languages for prototyping due to their poor performance. However, by using dedicated Python packages that call highly optimized functions, exploit multithreading and fully utilize modern CPUs performance potential developers could significantly speedup their algorithms achieving near native executing speed, avoiding the need to rewrite them in C/C++. Using profiling tools that can quickly pinpoint the bottlenecks in existing code also helps developers to improve performance of their applications in the most effective way by investing most developing time in improving performance of parts of code that really matter instead of trying to fix every single line.

8. Legal disclaimer and optimization notice

References

1. Studies on popularity of coding languages by www.codeeval.com. February, 2016. URL: http://blog.codeeval.com/codeevalblog/2016/2/2/most-popular-coding-languages-of-2016
2. L.Prechelt, An empirical comparison of seven programming languages // IEEE Computer, 2000, Vol. 33, Issue 10, pp. 23-29
3. D.Berkholz, Programming languages ranked by expressiveness. March, 2013. RedMonk. URL: http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/
4. J.Cownie, HPC Trends and What They Mean for Me, Imperial College, Oct. 2015.

5. A.Malakhov, Unleash parallel performance of Python programs. April, 2016. URL: https://software.intel.com/en-us/blogs/2016/04/04/unleash-parallel-performance-of-python-programs
6. Intel® Distribution for Python web site. URL: https://software.intel.com/en-us/python-distribution
7. Mixed Python/C/C++ Performance Analysis Feature – Beta. URL: https://software.intel.com/en-us/python-profiling
8. R.James, Intel® Data Analytics Acceleration Library. August, 2015. URL: https://software.intel.com/en-us/blogs/daal
9. Intel® Developer Zone Forum Thread "Numpy+MKL install fails: Could not locate executable icc". URL: https://software.intel.com/en-us/forums/intel-math-kernel-library/topic/536412
10. Intel® Developer Zone Forum Thread "Problem building 64 bit numpy using MKL and vc11 (Windows)". URL: https://software.intel.com/en-us/forums/intel-math-kernel-library/topic/532319
11. Intel® Developer Zone Forum Thread "MKL FATAL ERROR: undefined symbol: i_free". URL: https://software.intel.com/en-us/forums/intel-math-kernel-library/topic/300857
12. Intel® Developer Zone Forum Thread "Error running NumPy example: MKL FATAL ERROR: Cannot load libmkl_avx.so or libmkl_def.so.". URL: https://software.intel.com/en-us/forums/intel-math-kernel-library/topic/392223