PERFORMANCE ANALYSIS OF PYTHON* APPLICATIONS USING INTEL® VTUNE™ AMPLIFIER

Legal Disclaimer & Optimization Notice

Tune Python + native code for better Performance

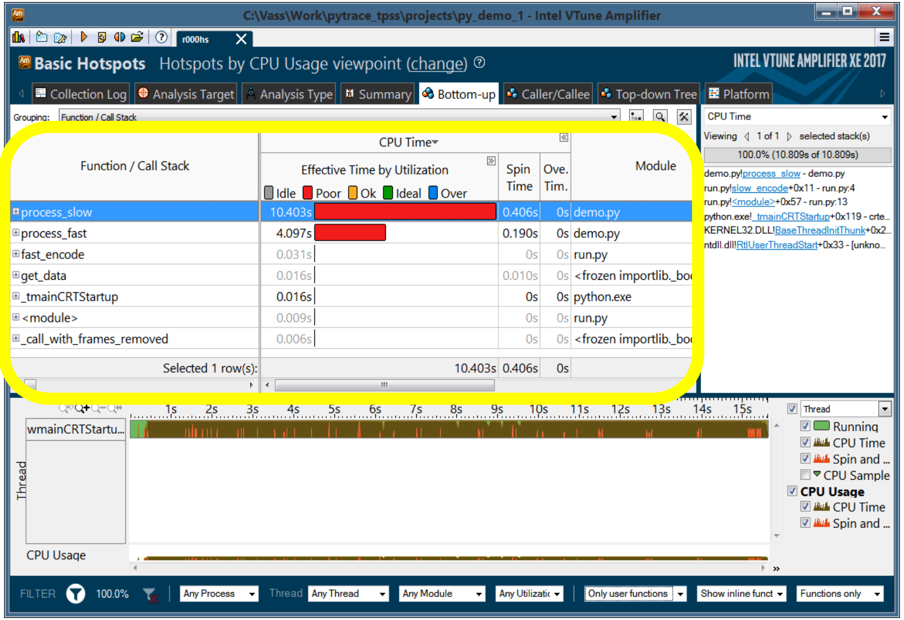
Intel® VTune™ Amplifier 2018, a performance analyzer in Intel® Parallel Studio XE suite

Challenge

- Full profile of Python + native applications
- Detect inefficient runtime execution

Solution

- Accurately identify performance hotspots at line-level
- Auto-detect mixed Python/C/C++ code and extensions
- Focus your tuning efforts for most impact on performance



Auto detection and performance analysis of Python and native functions

Available in Intel® VTune™ Amplifier 2017 Beta & Intel® Parallel Studio XE 2017

Download beta at https://software.intel.com/en-us/python-profiling

Performance analysis of the Covariance Matrix calculation

$$cov = \begin{bmatrix} \sum \frac{{x_1}^2}{N} & \sum \frac{x_1 x_2}{N} & \dots. & \sum \frac{x_1 x_n}{N} \\ \sum \frac{x_2 x_1}{N} & \sum \frac{{x_2}^2}{N} & \dots. & \sum \frac{x_2 x_n}{N} \\ \dots. & \dots. & \dots. & \dots. \\ \sum \frac{x_n x_1}{N} & \sum \frac{x_n x_2}{N} & \dots. & \sum \frac{{x_n}^2}{N} \end{bmatrix}$$

Activity #1: Run Vtune collection for Python: set environment

1. Setup parallel studio environment:

```
cd  ~/lab3
```

```
source /opt/intel/parallel_studio_xe_2019/psxevars.sh intel64
```

Activity #1: Run Vtune collection for Python: start amplifier gui

1. Check the script correctness: do not forget parameters!

```
python3 ./lab3.py somevec 100 1000
```

2. Observe the output (of covariance matrix calculation)

3. Start Intel Vtune Amplifier:

```
amplxe-gui
```

Intel® VTune™ Amplifier XE

Feature Highlights

Basic Hot Spot Analysis (Statistical Call Graph)

- Locates the time consuming regions of your application
- Provides associated call-stacks that let you know how you got to these time consuming regions
- Call-tree built using these call stacks

Advanced Hotspot and architecture analysis

- Based on Hardware Event-based Sampling (EBS)
- Pre-defined tuning experiments

Thread Profiling

- Visualize thread activity and lock transitions in the timeline
- Provides lock profiling capability
- Shows CPU/Core utilization and concurrency information

GPU Compute Performance Analysis

- Collect GPU data for tuning OpenCL applications. Correlate GPU and CPU activities

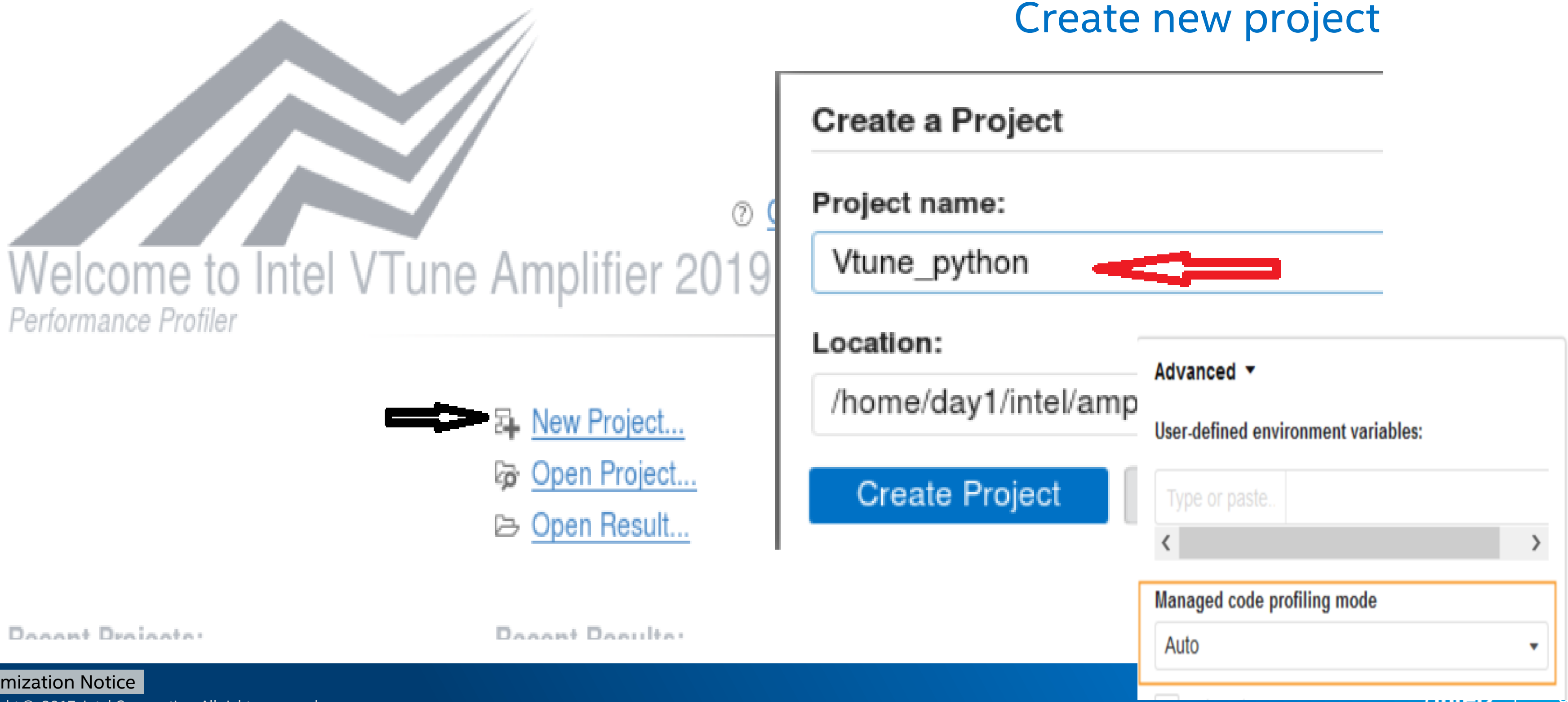
Intel® VTune™ Amplifier XE

Analysis Types (based on technology)

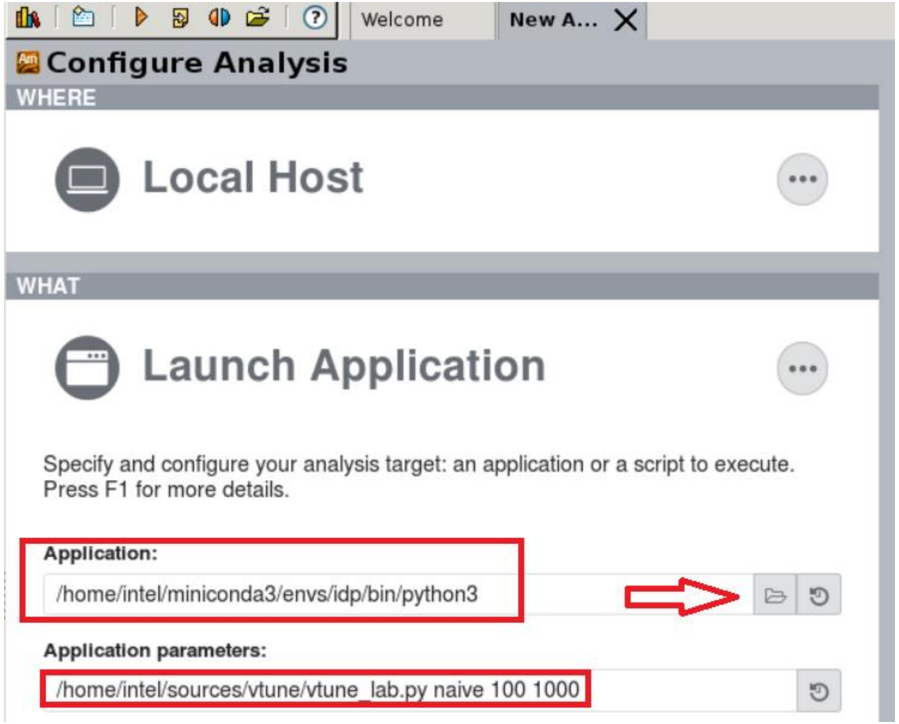
Software Collector Any x86 processor, any virtual, no driver	**Hardware Collector** Higher res., lower overhead, system wide
Basic Hotspots Which functions use the most time?	**Advanced Hotspots** Which functions use the most time? Where to inline? – Statistical call counts
Concurrency Tune parallelism. Colors show number of cores used.	**General Exploration** Where is the biggest opportunity? Cache misses? Branch mispredictions?
Locks and Waits Tune the #1 cause of slow threaded performance – waiting with idle cores.	**Advanced Analysis** Dig deep to tune bandwidth, cache misses, access contention, etc.

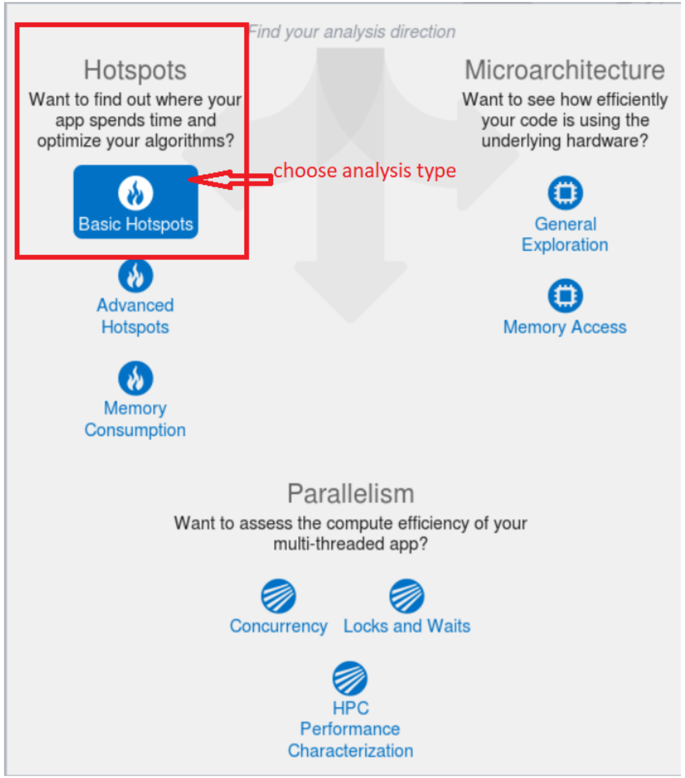
3.1. Configure Amplifier activity

Create new project



3.2 Configuring application to launch: do not forget to add app parameters!





3.3 Summary page: observer top hotspot list. Call function

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

INTEL VTUNE AMPLIFIER 2018

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Elapsed Time: 15.432s

CPU Time:	15.350s
Total Thread Count:	1
Paused Time:	0s

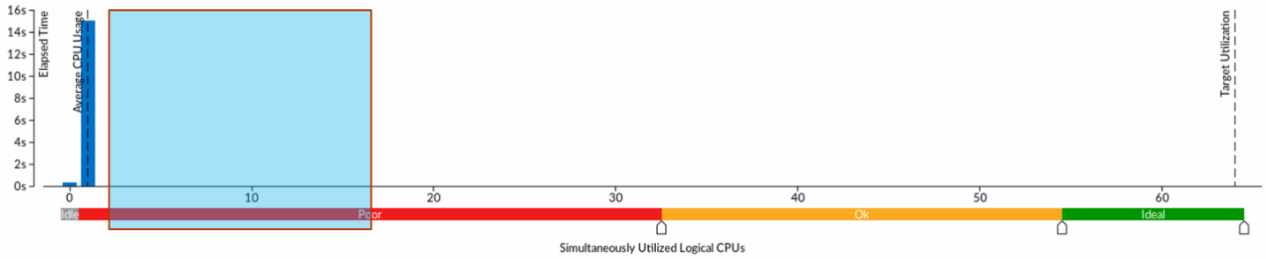
Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

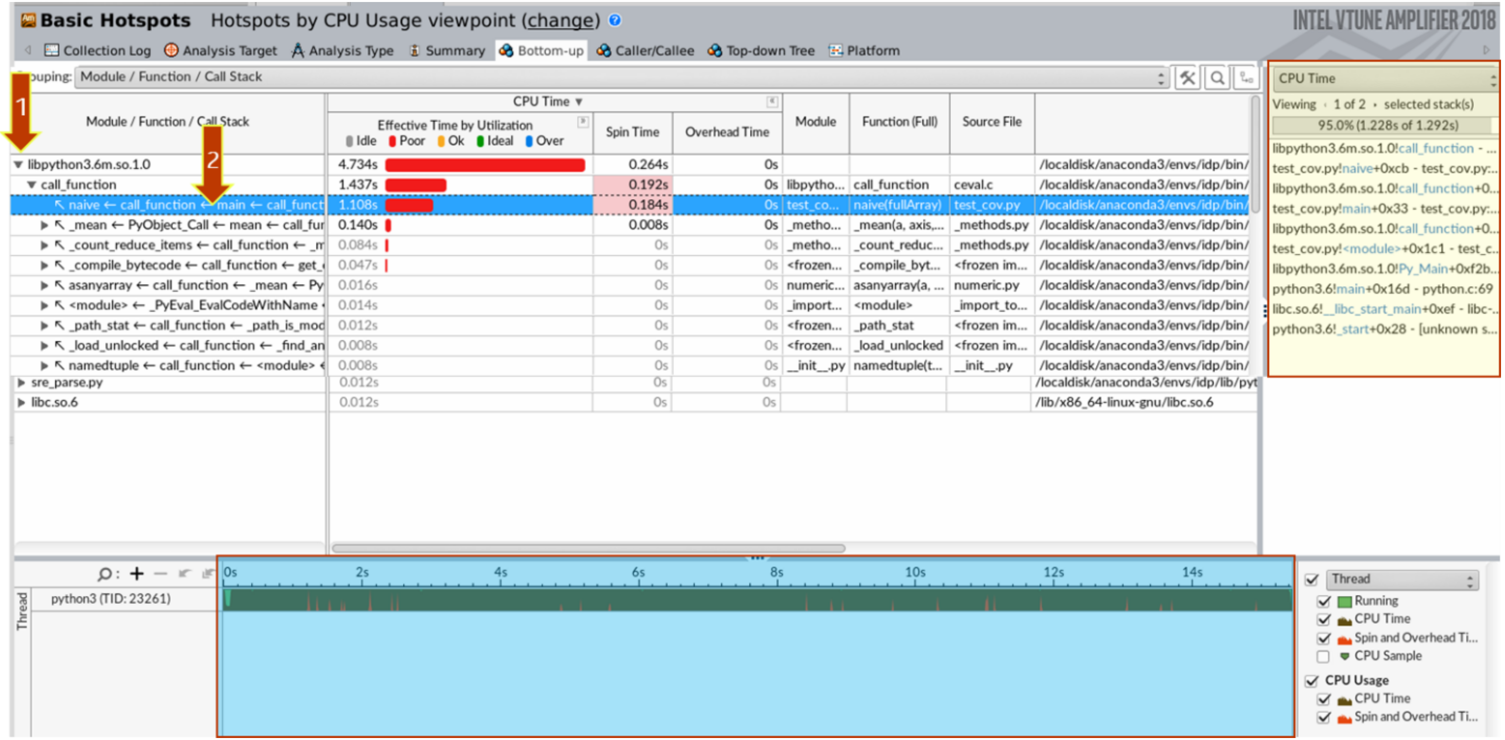
Function	Module	CPU Time
call_function	libpython3.6m.so.1.0	1.629s
PyObject_CallFunctionObjArgs	libpython3.6m.so.1.0	1.586s
PyUFunc_GenericFunction	umath.cpython-36m-x86_64-linux-gnu.so	0.898s
PyArray_NewFromDescr	multiarray.cpython-36m-x86_64-linux-gnu.so	0.784s
<genexpr>	test_cov.py	0.450s
[Others]		10.003s

CPU Usage Histogram

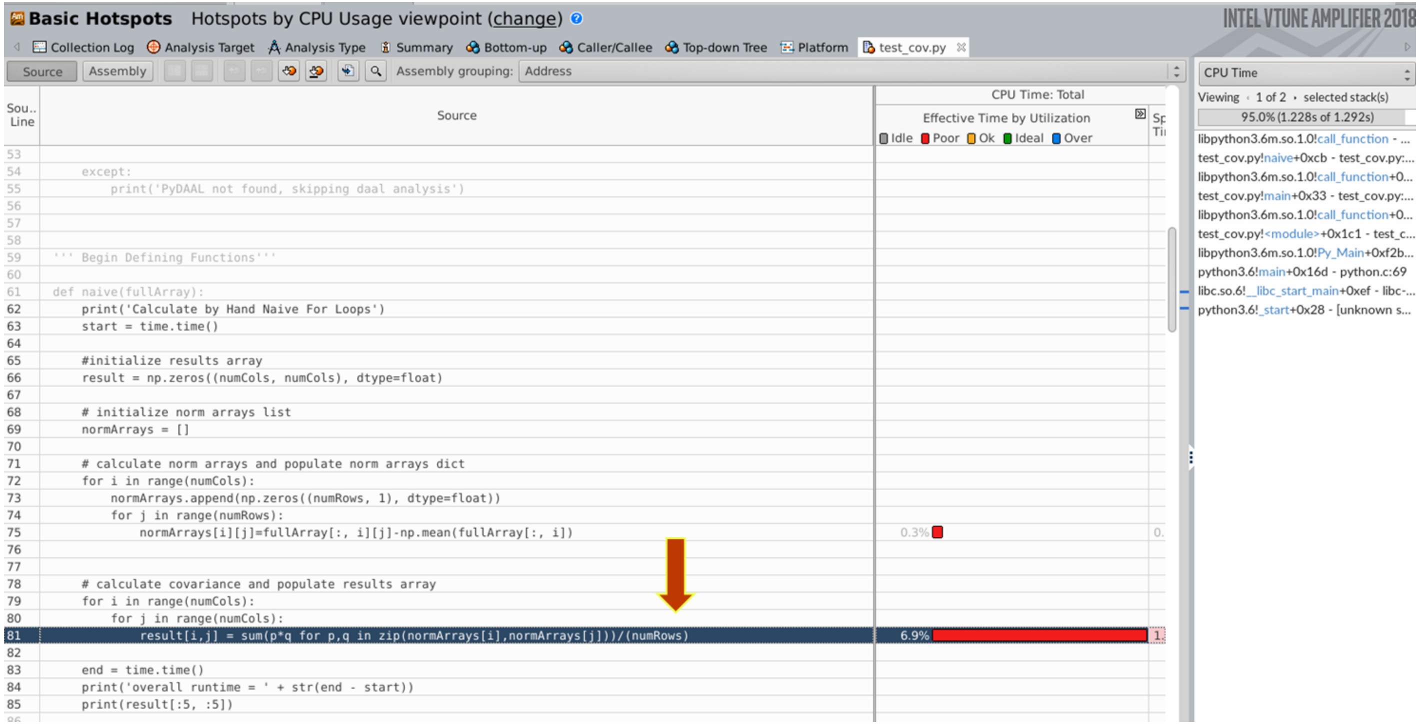
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



3.4 Basic hotspots bottom up view



3.5 Source code view



Activity #4: code modification. Numpy function substitute

```
    # calculate covariance and populate results array
    for i in range(numCols):
        for j in range(numCols):
            result[i,j] = sum(p*q for p,q in zip(normArrays[i],normArrays[j]))/(numRows)
```

6.9%

BEFORE

```
    # calculate covariance and populate results array
    for i in range(numCols):
        for j in range(numCols):
            result[i,j] = sum(p*q for p,q in zip(normArrays[i],normArrays[j]))/(numRows)
```

AFTER

```
    # calculate covariance and populat results array
    for i in range(numCols):
        for j in range(i+1, numCols):
            result[i,j] = sum(np.multiply(normArrays[i],normArrays[j]))/(numRows)
```

4.1 Re-run the performance.

- Run in the terminal

```
python3 lab3.py someVec 100 1000
```

Calculate by Hand Some Vectorization

overall runtime = 5.254168748855591

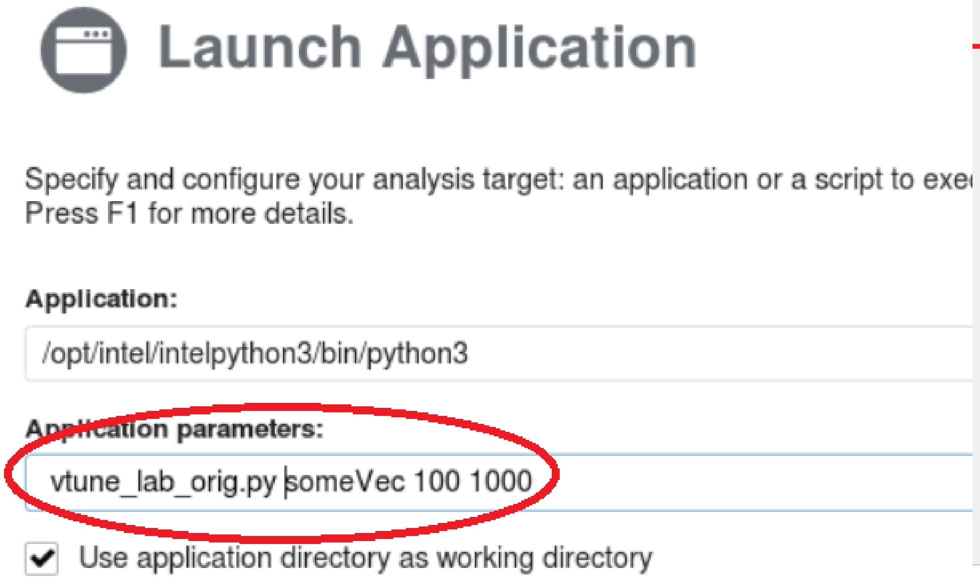
what could be improved further?

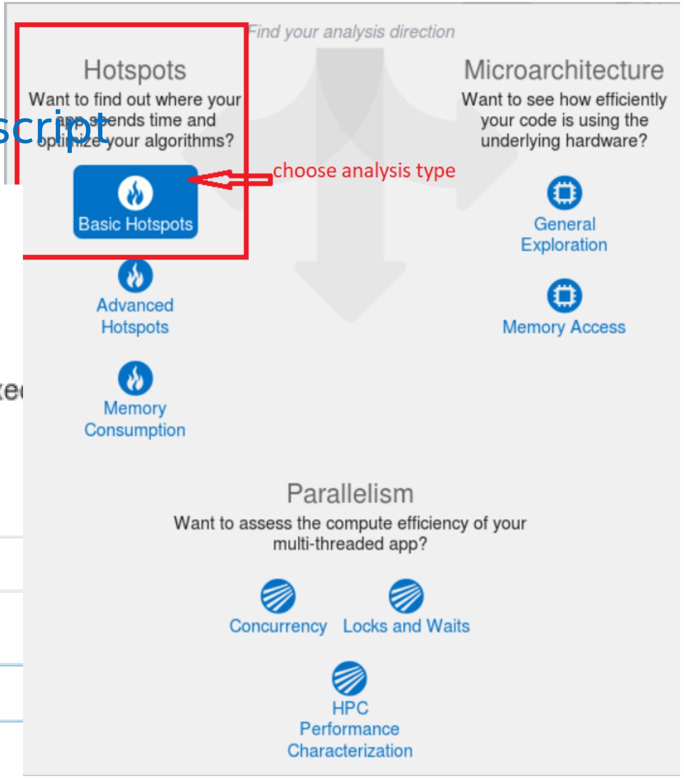
Activity #5

Further vectorization and performance improvement

5.1 Configuring application to launch:

Choose “someVec” module in the python script





5.2 Updated code: twice as fast.

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

INTEL VTUNE AMPLIFIER 201

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Elapsed Time: 7.135s

CPU Time:	7.120s
Total Thread Count:	1
Paused Time:	0s

Naive

Elapsed Time: 15.432s

CPU Time:	15.350s
Total Thread Count:	1
Paused Time:	0s

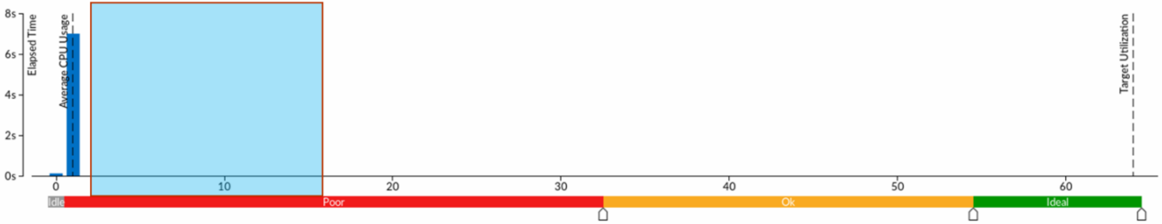
Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

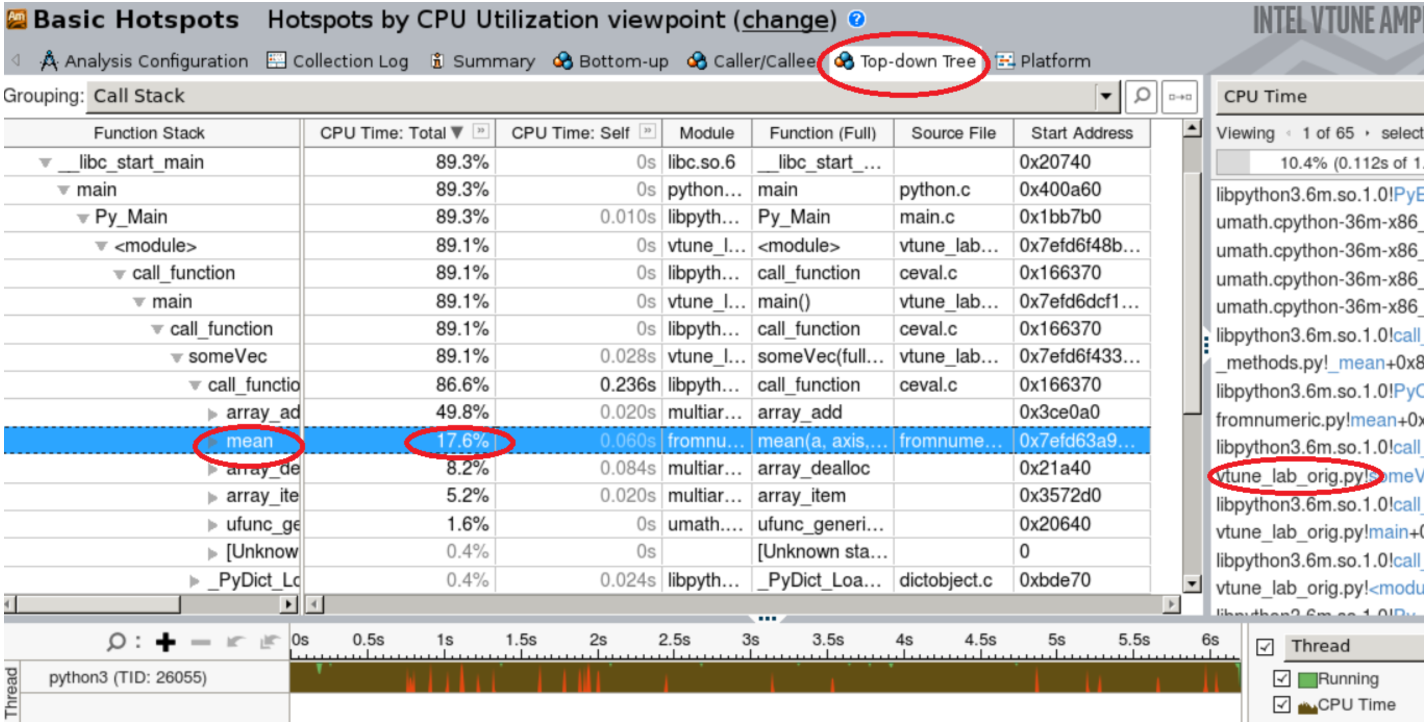
Function	Module	CPU Time
call_function	libpython3.6m.so.1.0	0.684s
PyObject_CallFunctionObjArgs	libpython3.6m.so.1.0	0.540s
PyUFunc_GenericFunction	umath.cpython-36m-x86_64-linux-gnu.so	0.368s
PyUFunc_DefaultLegacyInnerLoopSelector	umath.cpython-36m-x86_64-linux-gnu.so	0.296s
PyArray_NewFromDescr	multiarray.cpython-36m-x86_64-linux-gnu.so	0.276s
[Others]		4.956s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



5.3 Top down view to find the issue



5.4 Drill down to the source view

```
#"some vectorization" covariance function
def someVec(fullArray):
    print('Calculate by Hand Some Vectorization')
    start = time.time()

    #initialize results array
    result = np.zeros((numCols, numCols), dtype=float)


    # initialize norm arrays list
    normArrays = []

    # calculate norm arrays and populat norm arrays dict
    for i in range(numCols):
        normArrays.append(np.zeros((numRows, 1), dtype=float))
        for j in range(numRows):                                      0.1%
            normArrays[i][j]=fullArray[:, i][j]-np.mean(fullArray[:, i])   21.0%

    # calculate covariance and populat results array
    for i in range(numCols):
```

5.5 Code changes we need to do…

BEFORE

```
# calculate norm arrays and populat norm arrays dict
for i in range(numCols):
    normArrays.append(np.zeros((numRows, 1), dtype=float))
    for j in range(numRows):
        normArrays[i][j]=fullArray[:, i][j]-np.mean(fullArray[:, i])
```

AFTER

```
# calculate norm arrays and populat norm arrays dict
for i in range(numCols):
    normArrays.append(np.zeros((numRows, 1), dtype=float))
    normArrays[i]=np.subtract(fullArray[:, i], np.mean(fullArray[:, i]))
    for j in range(i+1):
        result[i,j] = sum(np.multiply(normArrays[i],normArrays[j]))/(numRows)
```

5.6 Re-run the test in command line

If the source code is modified within the same module please to this:

Run

```
python3 lab3.py someVec 100 1000
```

Calculate by Hand Some Vectorization

overall runtime = ?

Summary:

we identified the slowest line in our “Some Vectorization” covariance matrix function, and replaced it with a better organized for-loop and vector subtract method from Numpy*

Activity 6

Run

```
python3 lab3.py moreVec 100 1000
```

Calculate by Hand More Vectorization

overall runtime = 1.182539701461792

Let’s make sure it is the best possible performance!

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

INTEL VTUNE AMPLIFIER 2018

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Elapsed Time: 1.394s

CPU Time:	1.380s
Total Thread Count:	1
Paused Time:	0s

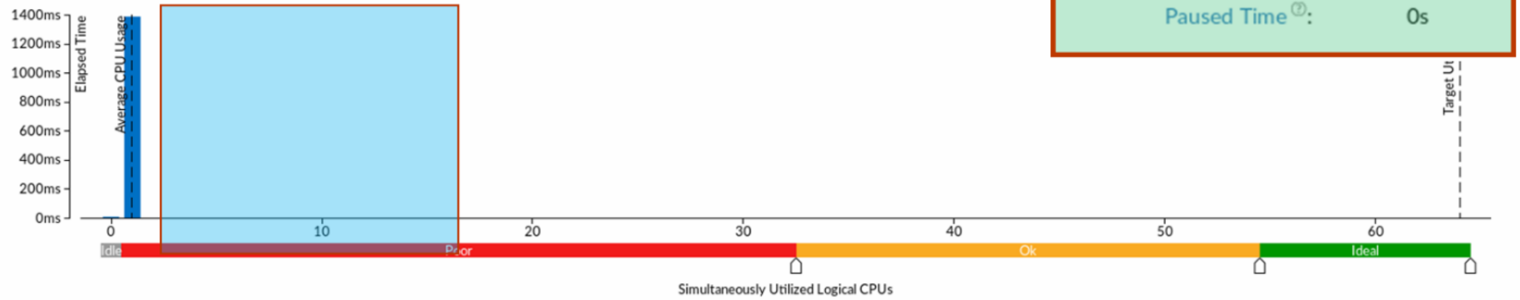
Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
call_function	libpython3.6m.so.1.0	0.182s
__GI_fetestexcept	libm.so.6	0.164s
PyCFunction_Call	libpython3.6m.so.1.0	0.137s
npy_get_floatstatus	umath.cpython-36m-x86_64-linux-gnu.so	0.100s
double_add	umath.cpython-36m-x86_64-linux-gnu.so	0.084s
[Others]		0.713s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage val



Naive

Elapsed Time: 15.432s

CPU Time:	15.350s
Total Thread Count:	1
Paused Time:	0s

Some Vectorization

Elapsed Time: 7.135s

CPU Time:	7.120s
Total Thread Count:	1
Paused Time:	0s

summary

VTUNE is the best tool ever and it now helps with Python code analysis, too!!!

backup

Resources

Get Intel's Distribution for Python:

- Visit https://software.intel.com/en-us/python-distribution for download, documentation and support
- Also available at Intel channel at Anaconda (you can use "conda install"!): https://anaconda.org/intel
- https://software.intel.com/en-us/vtune-amplifier-help-python-code-analysis
-

intel®