Compiling, optimising and debugging

Making the most of COSMA

ICC Theory Lunch

25th November 2019

Alastair Basden

Compiler overview

- 3 compiler families on COSMA:
  - Intel ICC (module load intel_comp)
    - Offers best performance
    - Several versions available
  - Gnu GCC (module load gnu_comp)
    - Several versions available
    - Newest generally gives best performance
  - AMD optimised compiler (module load aocc)
    - Avoid for now

Compiler choice

- icc generally produces faster code
- gcc is open source
- use of Makefiles will simplify compilation

Optimising code

- Key: Don’t optimise too early
- Ahmdals law
- Contiguous memory
- Memory allocation alignment
- Unroll loops
- Inline functions
- Reduce local variables (to fit in registers, rather than stack)
- Reduce function parameters
- Pass by reference not value
- Care with table lookups
- Reduce dynamic memory allocations (particularly in loops)
- Use optimised libraries
- Avoid repeated calculations and pointer dereferences within a loop

Compiler options

- -O3
  - Other flags, -march=native, -funroll-loops, -ffast-math, -Ofast
  - icc: -xHost, -fast
  - -O3 will allow code to run across COSMA. Other options may not.
- pragmas (hints to the compiler)
  - #pragma unroll(N)
  - Other pragmas are worth investigating

Vectorisation

- Operation on multiple floating point values simultaneously
  - Same operation applied to each
  - SIMD: Single instruction, multiple data
- Compilers can auto-vectorise if memory alignment is correct
  - Vectorisation reports can be obtained
  - e.g. compiler options (-fopt-info-vec-missed)
  - Intel vtune (see later)
  - #pragmas can be used to provide hints
- Vector intrinsics can also be used (i.e. similar to function calls)
- COSMA7 has 512-bit vector units (16x float or 8x doubles simultaneously)
- COSMA8 may have 256 or 512-bit units

Parallelisation

- Threading – pthreads
  - Most control
  - Within a node
- OpenMP
  - Easier to use, less control
  - Within a node
- MPI
  - Inter- and intra- node
  - Can be mixed with threading/openMP

Debugging

- The process of working out what is wrong
- Can be as simple as inserting “print” statements
  - though tools are available to help

Compute node access

- We used to allow users to ssh to compute nodes
  - Intel hyperthreading security bugs meant we had to stop that
- Selected users can still do so
  - If you wish to, please ask
- Can simplify the task of debugging and analysing running jobs
- Alternatively:
  - srun -p cosma7 -A dp004 -t 0:02:00 --x11 --pty /bin/bash
    - Then, once you get a prompt: module load cosma ; slurmx11-fix.sh
    - You can then run graphical tools from the node...

Debuggers on COSMA

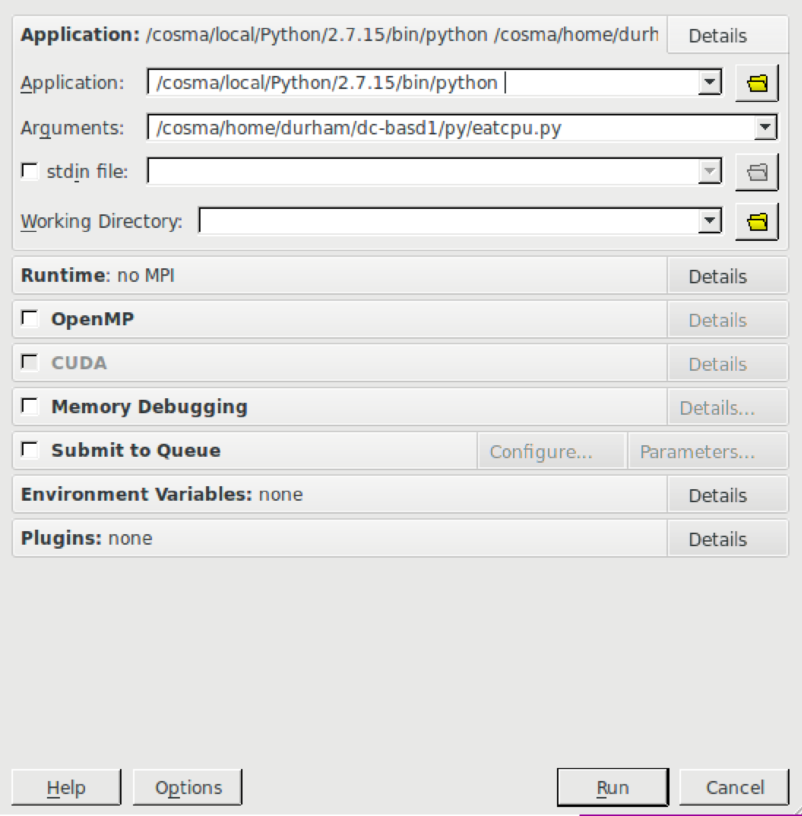
- gdb
- ddt

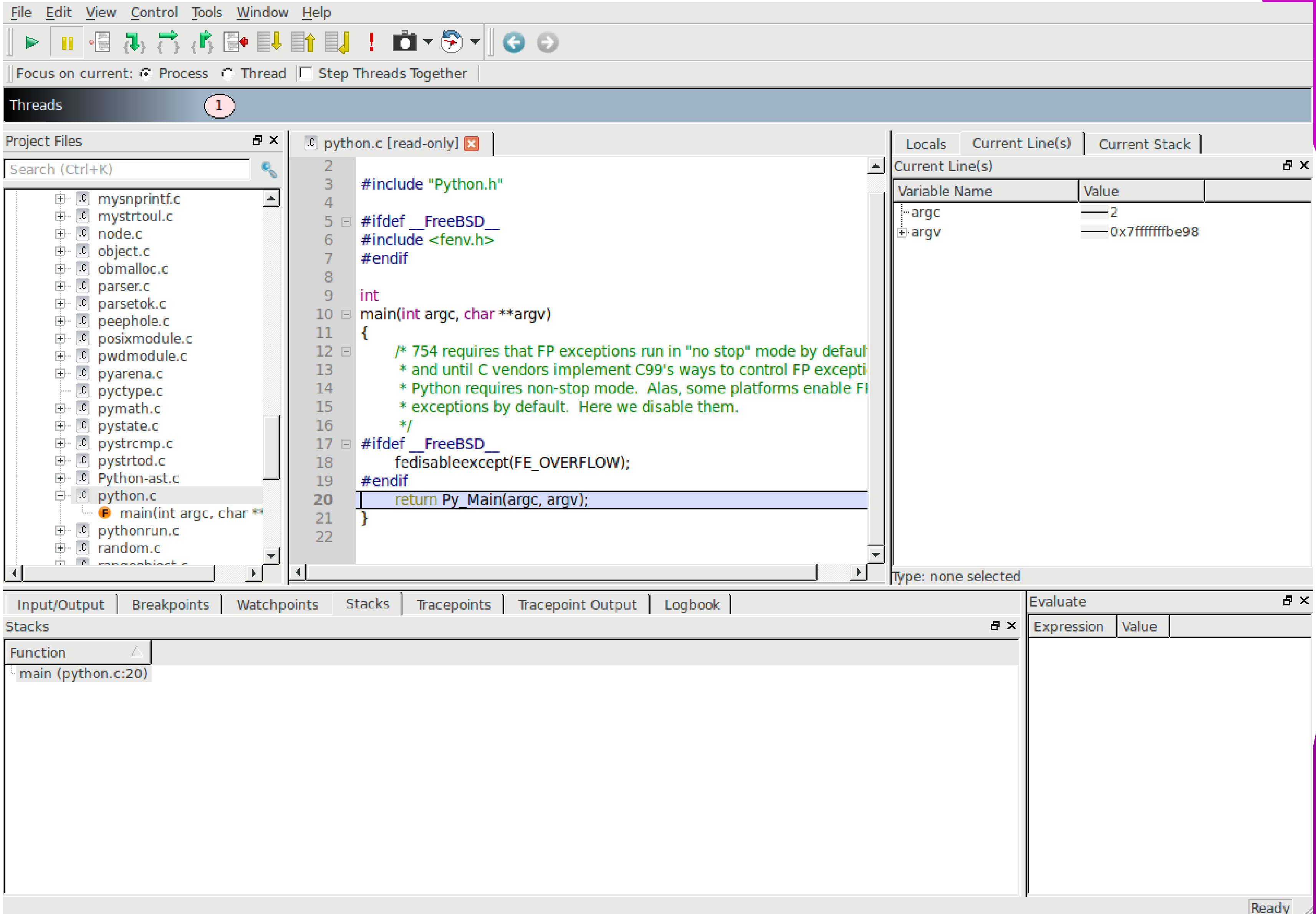
gdb

- Compile with -g (enables debugging symbols)
  - Remember to remove this for production codes, otherwise it will be slower than necessary
- gdb a.out
  - run <cmdline params…>
  - Can set breakpoints, investigate the stack, etc

ddt

- module load allinea/ddt
- ddt <<executable>>
- Works with MPI
  - Can connect to multiple MPI processes
- Understands COSMA queues
- Ask our resident expert - jch





File Edit View Control Tools Window Help
Focus on current: Process Thread Step Threads Together
Threads 1
Project Files
Search (Ctrl+K)
mysnprintf.c
mystrtoul.c
node.c
object.c
obmalloc.c
parser.c
parsetok.c
peephole.c
posixmodule.c
pwdmodule.c
pyarena.c
pyctype.c
pymath.c
pystate.c
pystrcmp.c
pystrtod.c
Python-ast.c
python.c
main(int argc, char **
pythonrun.c
random.c
python.c [read-only]
#include "Python.h"
#ifdef __FreeBSD__
#include <fenv.h>
#endif
int
main(int argc, char **argv)
{
/* 754 requires that FP exceptions run in "no stop" mode by defaul
* and until C vendors implement C99's ways to control FP excepti
* Python requires non-stop mode. Alas, some platforms enable FI
* exceptions by default. Here we disable them.
*/
#ifdef __FreeBSD__
fedisableexcept(FE_OVERFLOW);
#endif
return Py_Main(argc, argv);
}
Locals
Current Line(s)
Current Stack
Variable Name
Value
argc
2
argv
0x7fffffffbe98
Type: none selected
Input/Output
Breakpoints
Watchpoints
Stacks
Tracepoints
Tracepoint Output
Logbook
Stacks
Function
main (python.c:20)
Evaluate
Expression
Value
Ready

Other useful tools

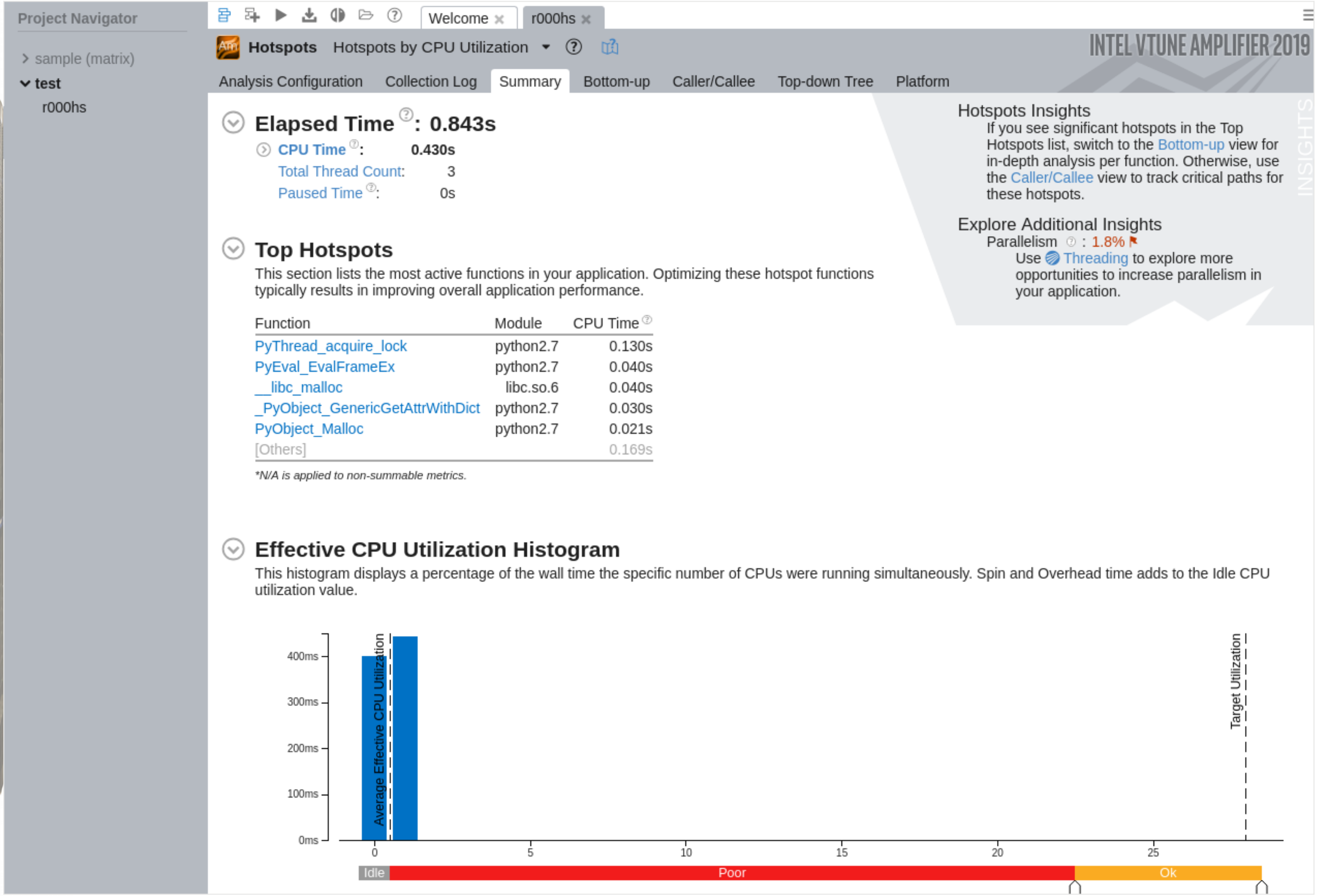
- valgrind
  - memory leaks, debugging and profiling
  - cache miss profiling
- electric fence (efence)
  - Detection of memory violations, e.g. reading/writing beyond the end of an array
  - Useful for double frees

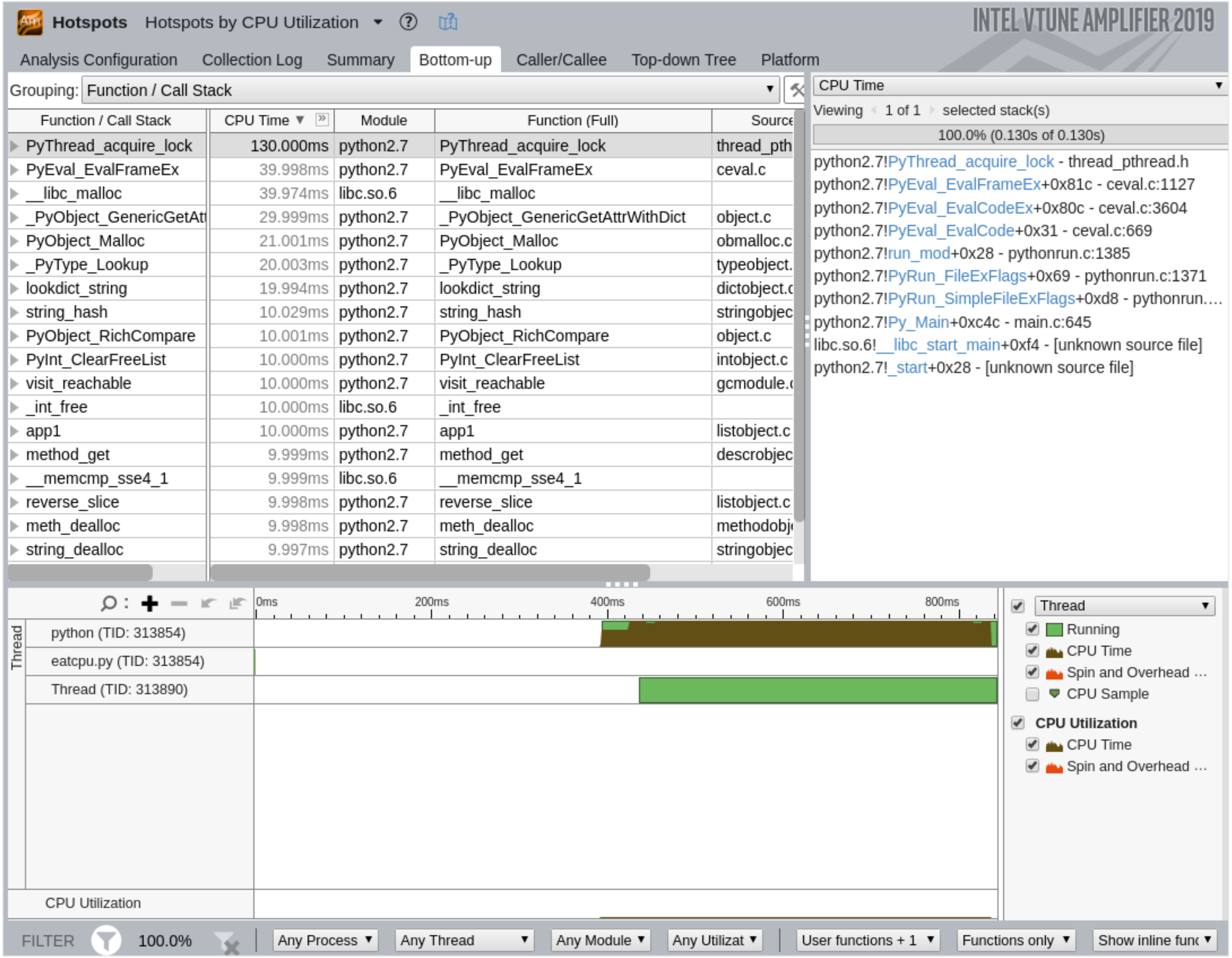
Measuring performance

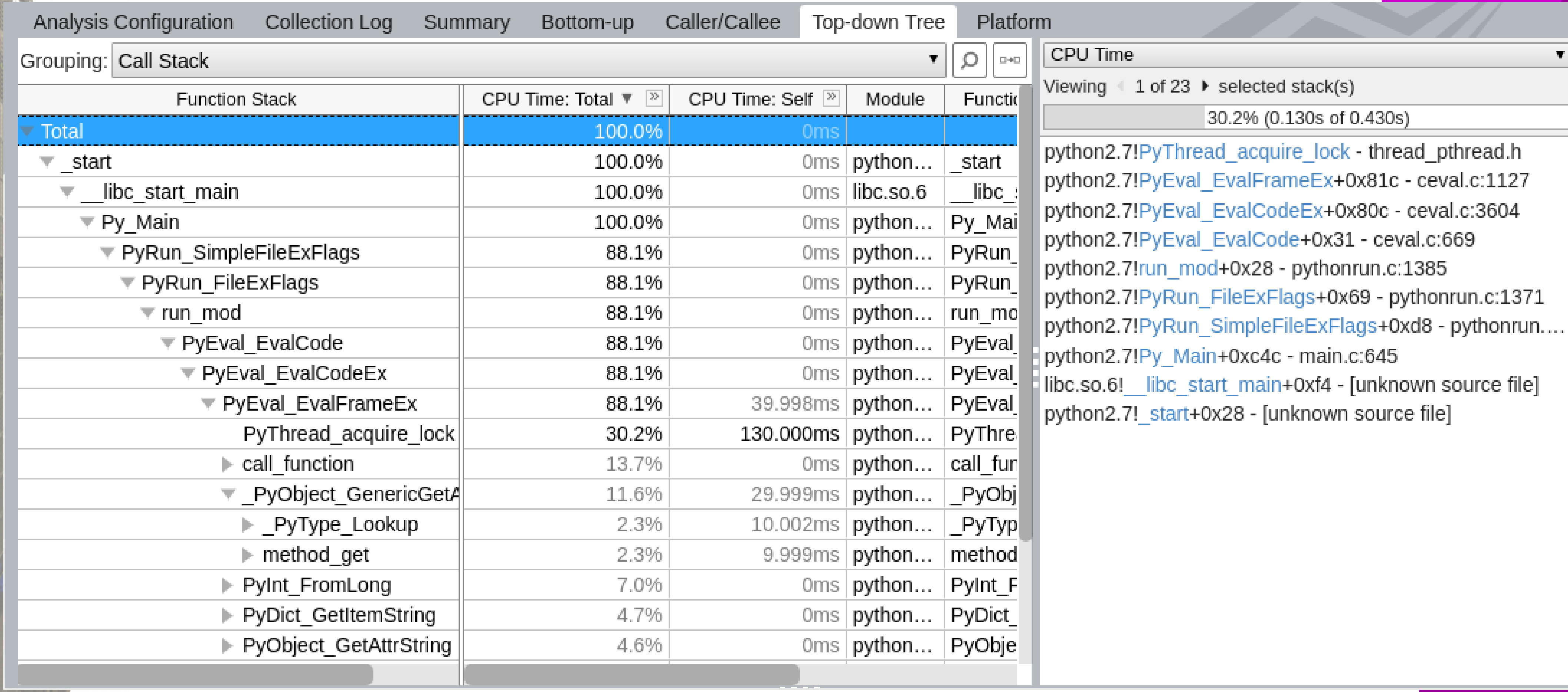
- Simpler tools include:
  - top/htop
  - perf
  - gprof
  - oprofile
- Others include:
  - Intel vtune/advisor
  - Allinea Map (no license)

Intel vtune amplifier

- Performance profiler
- module load vtune
  - amplxe-gui


Project Navigator
sample (matrix)
test
r000hs
Welcome
r000hs
INTEL VTUNE AMPLIFIER 2019
Hotspots
Hotspots by CPU Utilization
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Elapsed Time: 0.843s
CPU Time: 0.430s
Total Thread Count: 3
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
PyThread_acquire_lock | python2.7 | 0.130s
PyEval_EvalFrameEx | python2.7 | 0.040s
__libc_malloc | libc.so.6 | 0.040s
_PyObject_GenericGetAttrWithDict | python2.7 | 0.030s
PyObject_Malloc | python2.7 | 0.021s
[Others] | | 0.169s
*N/A is applied to non-summable metrics.
Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.
Explore Additional Insights
Parallelism: 1.8%
Use Threading to explore more opportunities to increase parallelism in your application.
INSIGHTS
Effective CPU Utilization 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 utilization value.
400ms
300ms
200ms
100ms
0ms
Average Effective CPU Utilization
Target Utilization
0
5
10
15
20
25
Idle
Poor
Ok


Hotspots
Hotspots by CPU Utilization
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Grouping: Function / Call Stack
Function / Call Stack
CPU Time
Module
Function (Full)
Source
PyThread_acquire_lock 130.000ms python2.7 PyThread_acquire_lock thread_pth
PyEval_EvalFrameEx 39.998ms python2.7 PyEval_EvalFrameEx ceval.c
__libc_malloc 39.974ms libc.so.6 __libc_malloc
_PyObject_GenericGetAt 29.999ms python2.7 _PyObject_GenericGetAttrWithDict object.c
PyObject_Malloc 21.001ms python2.7 PyObject_Malloc obmalloc.c
_PyType_Lookup 20.003ms python2.7 _PyType_Lookup typeobject.
lookdict_string 19.994ms python2.7 lookdict_string dictobject.c
string_hash 10.029ms python2.7 string_hash stringobjec
PyObject_RichCompare 10.001ms python2.7 PyObject_RichCompare object.c
PyInt_ClearFreeList 10.000ms python2.7 PyInt_ClearFreeList intobject.c
visit_reachable 10.000ms python2.7 visit_reachable gcmodule.c
_int_free 10.000ms libc.so.6 _int_free
app1 10.000ms python2.7 app1 listobject.c
method_get 9.999ms python2.7 method_get descrobjec
__memcmp_sse4_1 9.999ms libc.so.6 __memcmp_sse4_1
reverse_slice 9.998ms python2.7 reverse_slice listobject.c
meth_dealloc 9.998ms python2.7 meth_dealloc methodobj
string_dealloc 9.997ms python2.7 string_dealloc stringobjec
CPU Time
Viewing 1 of 1 selected stack(s)
100.0% (0.130s of 0.130s)
python2.7!PyThread_acquire_lock - thread_pthread.h
python2.7!PyEval_EvalFrameEx+0x81c - ceval.c:1127
python2.7!PyEval_EvalCodeEx+0x80c - ceval.c:3604
python2.7!PyEval_EvalCode+0x31 - ceval.c:669
python2.7!run_mod+0x28 - pythonrun.c:1385
python2.7!PyRun_FileExFlags+0x69 - pythonrun.c:1371
python2.7!PyRun_SimpleFileExFlags+0xd8 - pythonrun....
python2.7!Py_Main+0xc4c - main.c:645
libc.so.6!__libc_start_main+0xf4 - [unknown source file]
python2.7!_start+0x28 - [unknown source file]
0ms
200ms
400ms
600ms
800ms
Thread
python (TID: 313854)
eatcpu.py (TID: 313854)
Thread (TID: 313890)
CPU Utilization
Thread
Running
CPU Time
Spin and Overhead ...
CPU Sample
CPU Utilization
CPU Time
Spin and Overhead ...
FILTER
100.0%
Any Process
Any Thread
Any Module
Any Utilizat
User functions + 1
Functions only
Show inline func



Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Grouping: Call Stack
CPU Time
Viewing 1 of 23 selected stack(s)
30.2% (0.130s of 0.430s)
Function Stack
CPU Time: Total
CPU Time: Self
Module
Functic
Total 100.0% 0ms
_start 100.0% 0ms python… _start
__libc_start_main 100.0% 0ms libc.so.6 __libc_
Py_Main 100.0% 0ms python… Py_Mai
PyRun_SimpleFileExFlags 88.1% 0ms python… PyRun_
PyRun_FileExFlags 88.1% 0ms python… PyRun_
run_mod 88.1% 0ms python… run_mo
PyEval_EvalCode 88.1% 0ms python… PyEval_
PyEval_EvalCodeEx 88.1% 0ms python… PyEval_
PyEval_EvalFrameEx 88.1% 39.998ms python… PyEval_
PyThread_acquire_lock 30.2% 130.000ms python… PyThre
call_function 13.7% 0ms python… call_fun
_PyObject_GenericGetA 11.6% 29.999ms python… _PyObj
_PyType_Lookup 2.3% 10.002ms python… _PyTyp
method_get 2.3% 9.999ms python… method
PyInt_FromLong 7.0% 0ms python… PyInt_F
PyDict_GetItemString 4.7% 0ms python… PyDict_
PyObject_GetAttrString 4.6% 0ms python… PyObje
python2.7!PyThread_acquire_lock - thread_pthread.h
python2.7!PyEval_EvalFrameEx+0x81c - ceval.c:1127
python2.7!PyEval_EvalCodeEx+0x80c - ceval.c:3604
python2.7!PyEval_EvalCode+0x31 - ceval.c:669
python2.7!run_mod+0x28 - pythonrun.c:1385
python2.7!PyRun_FileExFlags+0x69 - pythonrun.c:1371
python2.7!PyRun_SimpleFileExFlags+0xd8 - pythonrun.…
python2.7!Py_Main+0xc4c - main.c:645
libc.so.6!__libc_start_main+0xf4 - [unknown source file]
python2.7!_start+0x28 - [unknown source file]

Summary

- Compile
- Debug
- Run
- Optimise
- Debug
- Run
- Debug
- Debug
- Debug
- Retire/academic position