

Profiling Tools

Introduction

This tutorial will guide you how to run intel profiling tools in AMD processors, we explore also the type of code we can profile.

Advisor

Project setup

First of all, we prepare an executable to run the tests. You can use any code to run these examples. Here we use for the nqueens example provided by advisor installation. We copy it from advisor installation directory:

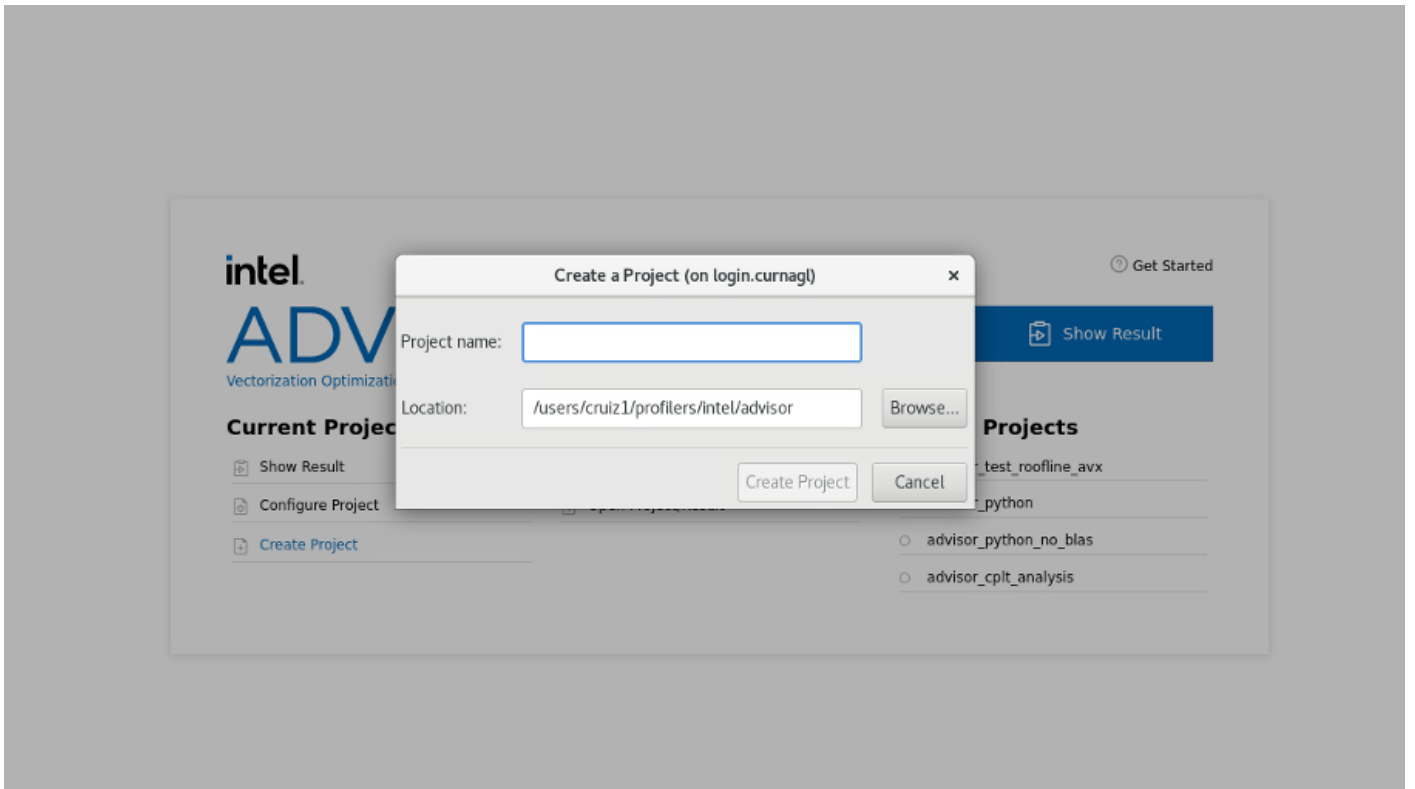
```
cp /dcsrsoft/spack/external/intel/2021.2/advisor/2021.2.0/samples/en/C++/nqueens_Advisor.tgz .
```

Then, extract the contents and compile the serial version:

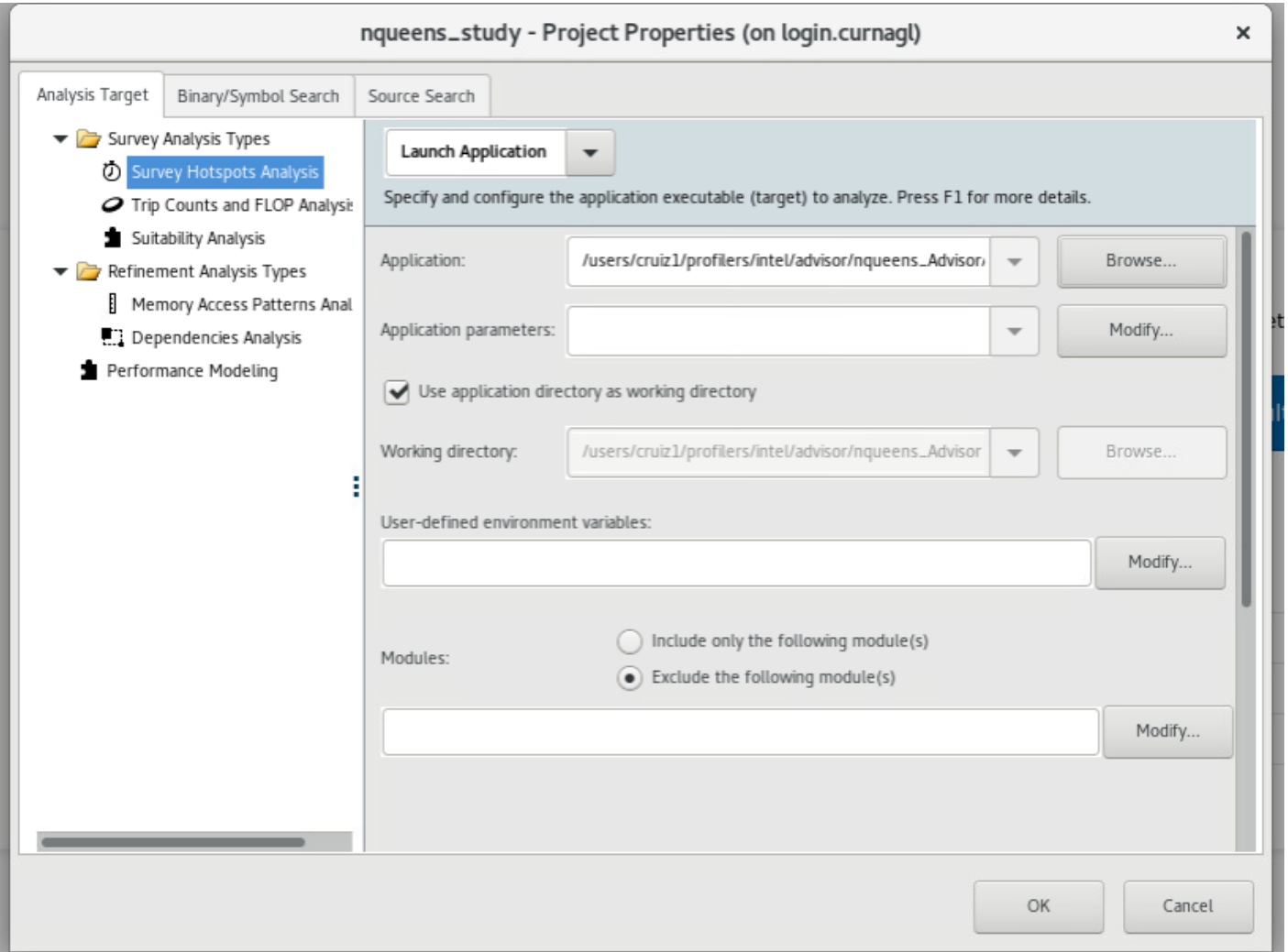
```
make 1_nqueens_serial
```

Creating a project

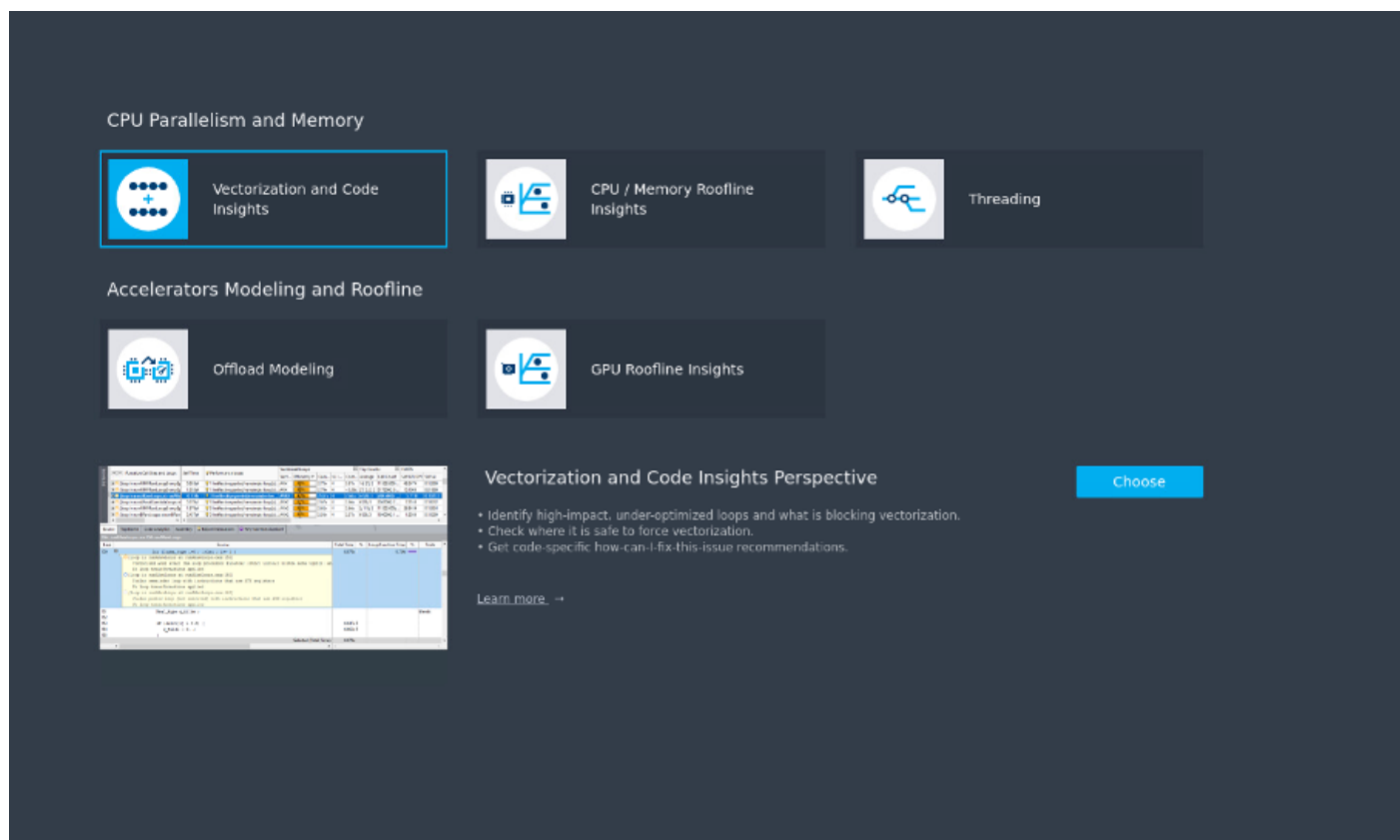
We create a project using advisor gui:



We configure the path of our nqueens executable (or the executable you want to profile), and we click on OK.



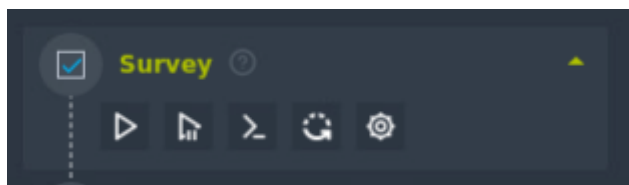
Several analysis are proposed :



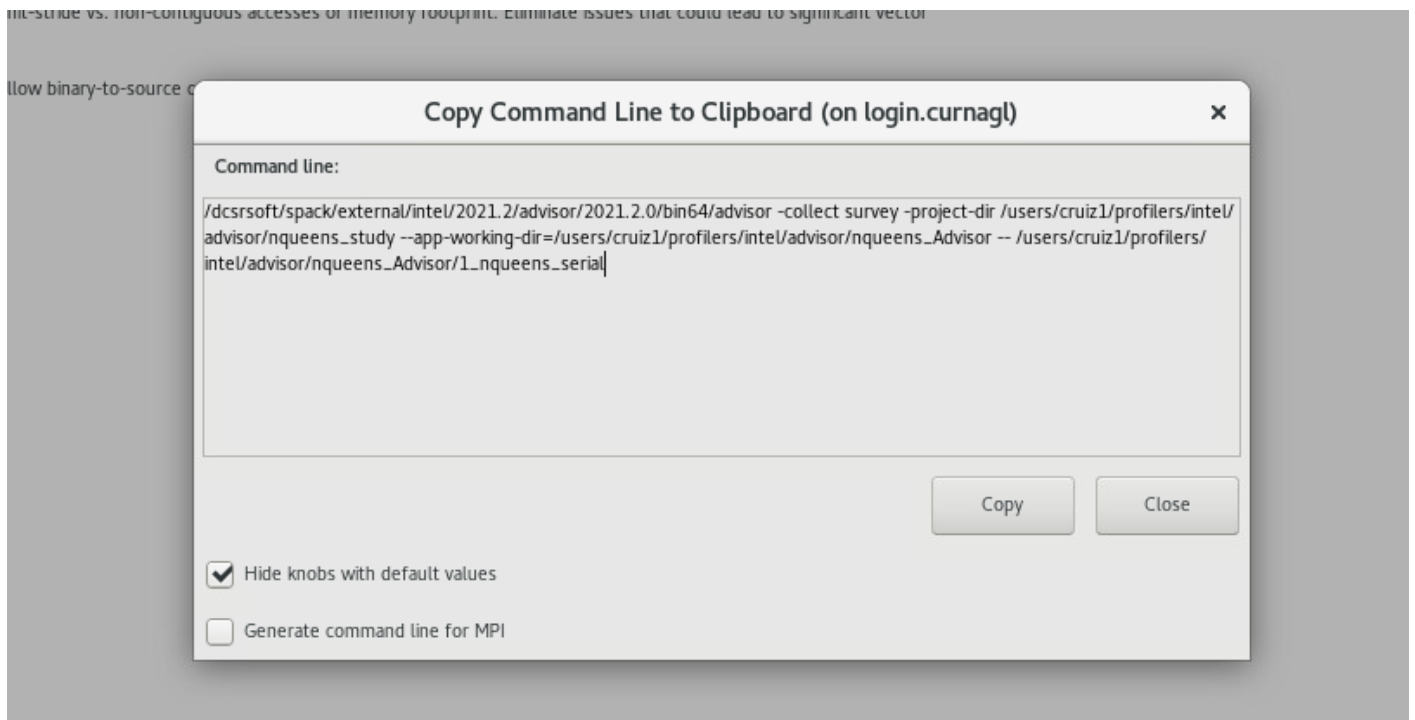
We start with **Vectorization and Code insights** which will give us information about the parallelization opportunities in the code. It identifies loops that will benefit most from vector parallelism, discover performance issues, etc. The summary window will give us more details.

Using SLURM

To use Advisor in the cluster, it is better to use the command line. The GUI can provide the commands we should run. Let's run the survey, to see the command to run, click on the following button



This will show the exact command to use:



We can copy that line in our slurm job:

```
#!/bin/sh

#SBATCH --job-name test-
prof

#SBATCH --error advisor-
%j.error

#SBATCH --output advisor-
%j.out

#SBATCH -N
1

#SBATCH --cpus-per-task
1
```

```
#SBATCH --partition
```

```
cpu
```

```
#SBATCH --time 1:00:00
```

```
dcsrsoft/spack/external/intel/2021.2/advisor/2021.2.0/bin64/advisor -collect survey -project-dir /users/cruiz1/profilers/intel/advisor/nqueens_study --app-working-dir=/users/cruiz1/profilers/intel/advisor/nqueens_Advisor -- /use\rs/cruiz1/profilers/intel/advisor/nqueens_Advisor/1_nqueens_serial
```

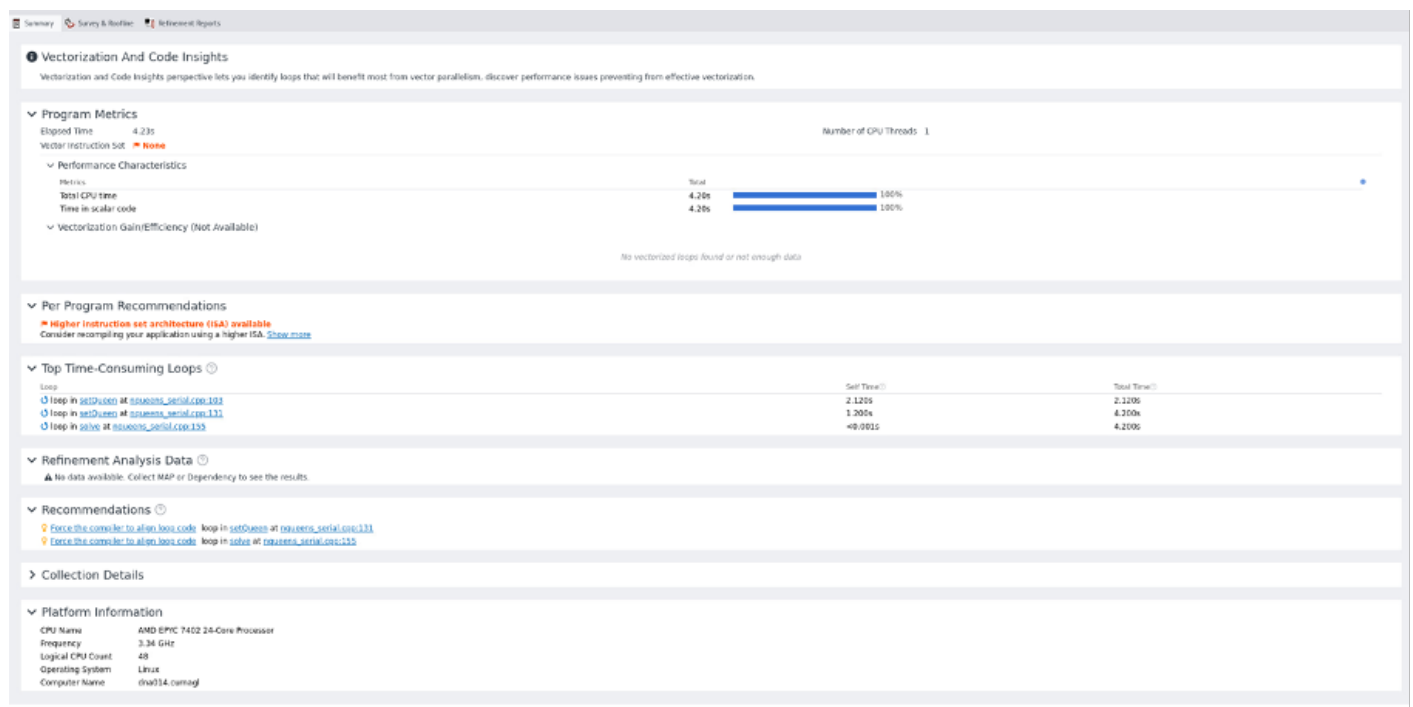
we launch the job:

```
sbatch slurm_advisor.sh
```

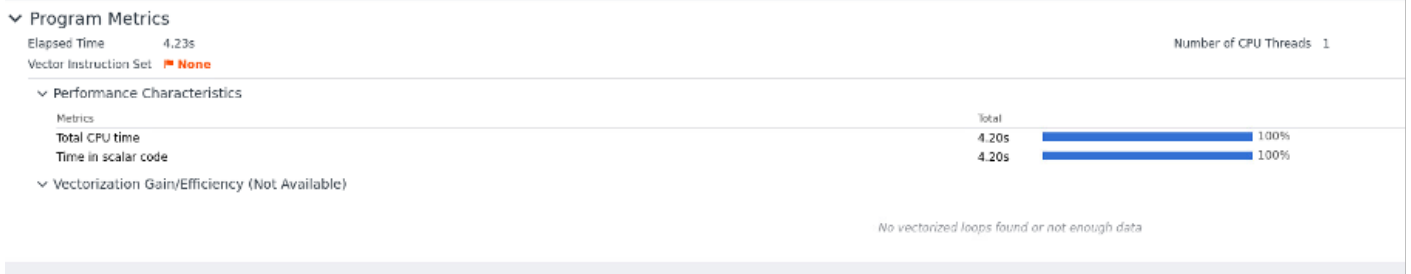
check for errors in Slurm output files.

Checking results

If we close and reopen the project, we see that we have some results:



We have recommendations for using other instruction sets because no vector instruction set was detected.



Per Program Recommendations

Higher instruction set architecture (ISA) available

Consider recompiling your application using a higher ISA. [Show more](#)

Top Time-Consuming Loops ?

Loop

loop in [setQueen](#) at [nqueens_serial.cpp:103](#)

loop in [setQueen](#) at [nqueens_serial.cpp:131](#)

loop in [solve](#) at [nqueens_serial.cpp:155](#)

It detects correctly the AMD CPU

Platform Information

CPU Name AMD EPYC 7402 24-Core Processor
Frequency 3.34 GHz
Logical CPU Count 48
Operating System Linux
Computer Name dna014.curnagl

In the survey window we can observe the time consuming parts of the code. Each line on the table represent either a function call or a loop. Several useful information is presented by line such as: vector instructions used, length of the vector instruction and type of data.

Function Call, Sites and Loops	Performance Issues	CPU Time	Type	Why? No Vectorization?	Vectorized Loops	Instruction Set Analysis	Assembly	Location
		Total Time	Self Time		Vector ISA	Gain Estimate	VL (Vector Length)	
loop in setQueen at nqueens_serial.cpp:103		2.120s	2.120s	Scalar	Compiler decides sufficient...			nqueens_serial.cpp:103
loop in setQueen at nqueens_serial.cpp:131	1 Missaligned loop...	4.200s	0.000s	Scalar				nqueens_serial.cpp:131
setQueen		4.200s	0.000s	Function				nqueens_serial.cpp:96
main		4.200s	0.000s	Function				nqueens_serial.cpp:176
loop in solve at nqueens_serial.cpp:155	1 Missaligned loop...	4.200s	0.000s	Scalar				nqueens_serial.cpp:155
solve		4.200s	0.000s	Function				nqueens_serial.cpp:143

On the window above, we should see recommendation about the vector instructions to use. This is missing probably due to the fact that we are using an AMD processors. Compilation of code using

Intel compiler did not help.

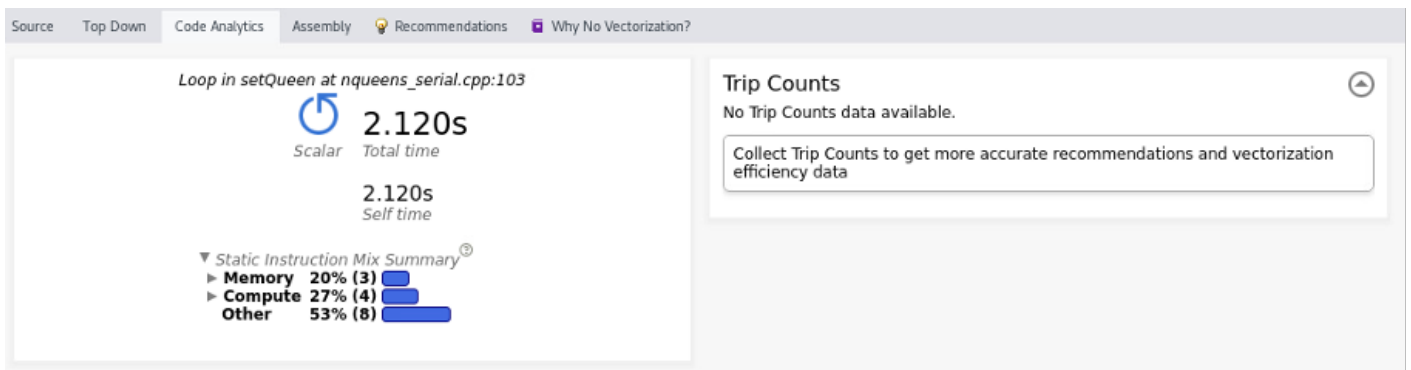
The lower half of the screen shows the following tabs:

- source code (available if compiled with -g)
- top down shows the call tree
- code analysis shows the most time consuming loop as well as a profile of the application in terms of resources (CPU, memory)

On the top down tab, we can see where the call is taking place:

Location		
Source Location	Module	
	python3.8	
	libc.so.6	
run_test.py:1	run_test.py	
ceval.c:4950	libpython3.8.so.1.0	
traceback_utils.py:59	traceback_utils.py	
mnist.py:70	mnist.py	
base.py:626	base.py	
dense.py:111	dense.py	
arrayprint.py:1579	arrayprint.py	
flatten.py:60	flatten.py	
import.c:1722	libpython3.8.so.1.0	
typeobject.c:6603	libpython3.8.so.1.0	
abstract.c:1018	libpython3.8.so.1.0	
	libc.so.6	
	libpthread.so.0	
	libpthread.so.0	
	libtensorflow_framework.so.2	
	_pywrap_tensorflow_internal.so	
	_pywrap_tensorflow_internal.so	
	_pywrap_tensorflow_internal.so	

Below a screenshot of the code analysis window.

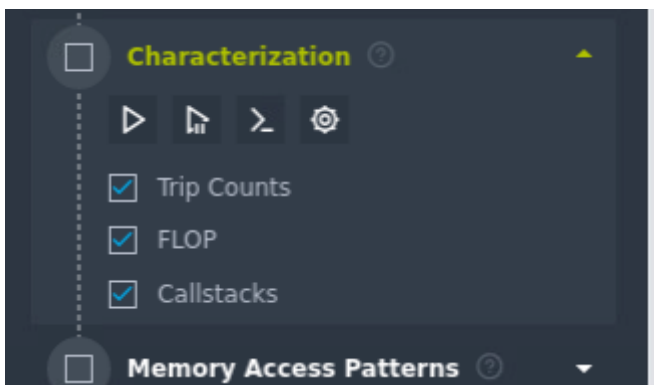


Collecting trip counts

We choose characterization analysis. To improve the analysis we should choose a loop, this can be done on the survey window:

[loop in setQueen at nqueens_serial.cpp:131]	<input checked="" type="checkbox"/>	1 Misaligned loop...	4.200s	100.0%	1.200s	Scalar
f setQueen	<input type="checkbox"/>		4.200s	100.0%	0.880s	Function
f _start	<input type="checkbox"/>		4.200s	100.0%	0.000s	Function
f main	<input type="checkbox"/>		4.200s	100.0%	0.000s	Function
[loop in solve at nqueens_serial.cpp:155]	<input type="checkbox"/>	1 Misaligned loop...	4.200s	100.0%	0.000s	Scalar
f solve	<input type="checkbox"/>		4.200s	100.0%	0.000s	Function





And then launch the characterization, again we ask for the cmd line :









The generated command will contain the additional options:

```
tripcounts -flop -stacks -mark-up-list-2
```

We can see the different trip counts for each loop:

Compute Performance  				Memory 		Trip Counts 	
Self GFLOPS	Total GFLOPS	Self AI	Total AI			Average	Call Count
0	0	0	0	6.472		4	337571330
0	0	0	0	1.512		14	10030692
0	0	0	0	3.029			377901398
	0		0				
	0		0				
	0		0				
	0	0	0	< 0.001			1

We can now repeat the process for memory access analysis. After running the analysis, we have new information:

Memory Access Patterns Report   Dependencies Report  Recommendations										
ID		Stride	Type	Source	Nested Function	Variable references	Max. Per-Instruction Addr. Range	Modules	Site Name	Access Type
P1			Parallel site information	nqueens_serial.cpp:105				1_nqueens_serial	loop_site_3	
P3		1	Unit stride	nqueens_serial.cpp:105		block 0x1538ec0 allocated at nqueens_serial.cpp:150	28B	1_nqueens_serial	loop_site_3	Read

If we compile the code with more performant instruction set, this is detected in the summary window:

Program Metrics	
Elapsed Time	13.68s
Vector Instruction Set	AVX2, AVX
Performance Characteristics	
Metrics	
Total CPU time	
Time in scalar code	
Vectorization Gain/Efficiency (Not Available)	

and the call stack window:

Performance Issues	CPU Time		Type	Why No Vectorization?	Vectorized Loops			Instru
	Total Time	Self Time			Vector ISA	Gain Estimate	VL (Ve ...	
2 Vector regist...	19.680s	19.680s	Vectorized (Body)		AVX		4	Divis
1 Unoptimized fl...	16.590s	16.590s	Scalar					Divisi
2 Vector register...	14.150s	14.150s	Vectorized (Body)		AVX2		4; 8	Extra
1 Vector register...	10.310s	10.310s	Vectorized (Body)		AVX2		4; 8	Extra
	2.230s	2.230s	Vectorized (Body)		AVX		2	
	2.229s	2.229s	Scalar					FMA
	1.840s	1.840s	Scalar					FMA
	1.660s	1.660s	Vectorized (Body)		AVX2		2	FMA
	1.591s	1.591s	Vectorized (Body)		AVX2		2	FMA
	1.440s	1.440s	Scalar					FMA
	1.350s	1.350s	Vectorized (Body)		AVX2		4	FMA
1 Possible ineffici...	1.170s	1.170s	Vectorized (Body)		AVX2		4; 32	Extra
	1.080s	1.080s	Scalar					FMA
	1.080s	1.080s	Vectorized (Body)		AVX2		2	FMA
	1.000s	1.000s	Vectorized (Body)		AVX		4	
	0.950s	0.950s	Vectorized (Body)		AVX2		4	FMA
	0.950s	0.950s	Vectorized (Body)		AVX2		4	Extra
	0.890s	0.890s	Vectorized (Body)		AVX2		4	Extra
	0.760s	0.760s	Vectorized (Body)		AVX2		2	FMA

This screenshot was obtained profiling HPL Benchmark.

MPI profiling

The command proposed by the GUI is not the appropriate, we should use the following command:

```
srun advisor --collect survey --trace-mpi -project-dir
/users/cruiz1/profilers/intel/advisor/analysis_mpi_trace-2 --app-working-
dir=/users/cruiz1/profilers/intel/advisor/mpi_sample --
/users/cruiz1/profilers/intel/advisor/mpi_sample/mpi_sample_serial
```

The default behavior generates a profile database per rank which is not ideal to understand the interactions between MPI ranks. We can use the option `--mpi-trace` but unfortunately it does not seem to give more additional information as it only works if we use the same host.

One possible approach is to only profile one processes using SLURM multiprogram option:

```
srun --multi-prog task.conf
```

the task.conf would look like:

```
0 /dcsrsoft/spack/external/intel/2021.2/advisor/2021.2.0/bin64/advisor -collect survey -
project-dir $PROJECT_DIR -- $PATH_BINARY/xhpl

1-3 ./xhpl
```

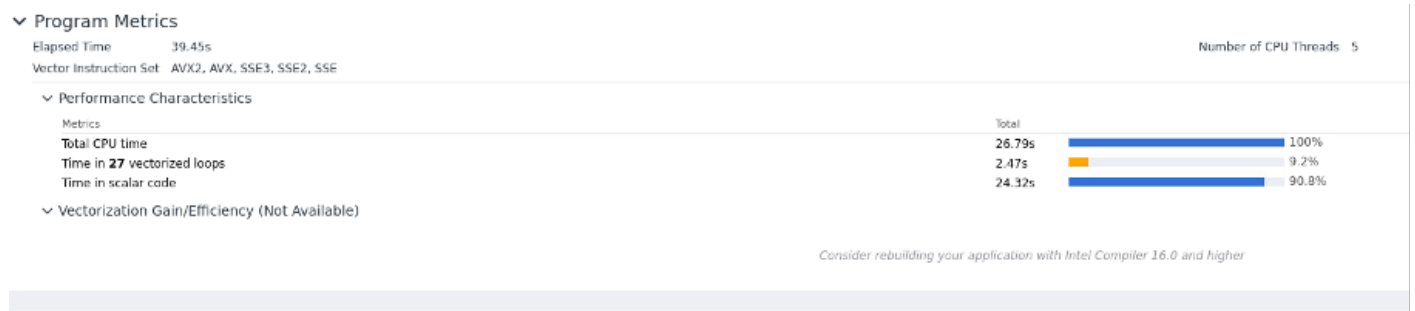
In this example, we profile the rank 0.

Python application

It is possible to profile python applications by adding '--profile-python' option. For example to profile a tensorflow code:

```
advisor -collect survey --profile-python -project-dir
/users/cruiz1/profilers/intel/advisor/tensor_flow_study -- python
/users/cruiz1/python/run_test.py
```

We have the following summary:



We can see that the code uses vector instruction (In this example the CPU version of Tensorflow was used).

The complete call tree shows:

⊕ _start	81.2%	21.750s	0.000s [Function
⊖ clone	18.8%	5.040s	0.000s [Function
⊖ start_thread	18.8%	5.040s	0.000s [Function
⊖ [loop in start_thread]	18.8%	5.040s	0.000s [Scalar
⊖ tensorflow::(anonymous namespace)::PThread::ThreadFn	18.8%	5.040s	0.000s [Function
⊖ std::_Function_handler<void (void), tensorflow::thread::EigenEnvironment::CreateThre	17.6%	4.710s	0.000s [Function
⊖ Eigen::ThreadPoolTempl<tensorflow::thread::EigenEnvironment>::WorkerLoop	17.4%	4.660s	0.000s [Function
⊖ [loop in Eigen::ThreadPoolTempl<tensorflow::thread::EigenEnvironment>::V	17.4%	4.660s	0.000s [Scalar
⊖ [loop in Eigen::ThreadPoolTempl<tensorflow::thread::EigenEnvironment	17.4%	4.660s	0.000s [Scalar
⊖ [loop in Eigen::ThreadPoolTempl<tensorflow::thread::EigenEnvironn	17.4%	4.660s	0.000s [Scalar
⊖ [loop in Eigen::ThreadPoolTempl<tensorflow::thread::EigenEnvir	17.4%	4.660s	0.000s [Scalar
⊖ [loop in Eigen::ThreadPoolTempl<tensorflow::thread::EigenE	17.4%	4.650s	0.000s [Scalar
⊖ [loop in Eigen::ThreadPoolTempl<tensorflow::thread::Eig	17.4%	4.650s	0.000s [Scalar
⊖ tensorflow::(anonymous namespace)::ExecutorState<te	15.5%	4.140s	0.020s [Function
⊖ [loop in tensorflow::(anonymous namespace)::Ex	15.3%	4.100s	0.000s [Scalar
⊖ [loop in tensorflow::(anonymous namespace)	15.2%	4.080s	0.030s [Scalar
⊖ tensorflow::ThreadPoolDevice::Compute	14.2%	3.810s	0.040s [Function
⊖ tensorflow::BaseBatchMatMulOp<Eigen:	4.6%	1.220s	0.010s [Function
⊖ tensorflow::LaunchBatchMatMul<Eig	4.4%	1.190s	0.000s [Function
⊖ Eigen::TensorEvaluator<Eigen::Te	4.4%	1.190s	0.000s [Function
⊖ Eigen::TensorContractionEval	4.3%	1.150s	0.000s [Function
⊖ [loop in Eigen::Tensor	4.3%	1.150s	0.000s [Scalar
⊖ [loop in Eigen::Ten	4.3%	1.150s	0.000s [Scalar
⊖ [loop in Eigen::	4.3%	1.150s	0.000s [Scalar
⊖ Eigen::internal	2.9%	0.790s	0.000s [Function
⊖ dnnLsgen	2.9%	0.790s	0.000s [Function
⊖ dnnLsgen	2.9%	0.790s	0.000s [Function

We can explore the main script and how CPU time is distributed:

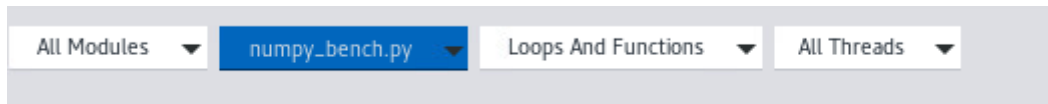
```

0      # Matrix multiplication
1      N = 20
2      t = time()
3      for i in range(N):
4          np.dot(A, B)
5      delta = time() - t
6      print('Dotted two %dx%d matrices in %0.2f s.' % (size, size, delta / N))
7      del A, B
-

```

0.010s {
14.030s ■

Unfortunately, this is not automatic, we should filter it using the source file filter, example:



Intel VTune

Limitations

Limited number of analysis

Unfortunately, for most of the analysis:

- hpc-performance
- memory-acesss
- performance snapshot
- uarch-exploration
- io

we obtained the following error message:

```

vtune: Error: This analysis type is not applicable to the system because VTune Profiler cannot
recognize the processor.

If this is a new Intel processor, please check for an updated version of VTune Profiler. If
this is an unreleased Intel processor

```

for io analysis we have the following error:

vtune: Error: Cannot enable event-based sampling collection: Architectural Performance Monitoring version is 0. Make sure the vPMU feature is enabled in your hypervisor.

Maximum number of threads

The tool detect a maximum number of 16 threads

Launching analysis in SLURM

We can still do some analysis like 'hotspots analysis'.

```
#!/bin/sh
#SBATCH --job-name test-vtune
#SBATCH --error vtune-%j.error
#SBATCH --output vtune-%j.out
#SBATCH -N 1
#SBATCH --cpus-per-task 8
#SBATCH --partition cpu
#SBATCH --time 1:00:00

export OMP_NUM_THREADS=8
source /dcsoft/spack/external/intel/2021.2/vtune/2021.2.0/amplxe-vars.sh
vtune -collect hotspots ./matrix
```

Hotspot analysis

The summary window looks like:

⌵

Elapsed Time[?]: 135.860s

⌵ CPU Time[?]: 134.620s

Total Thread Count: 17

Paused Time[?]: 0s

⌵

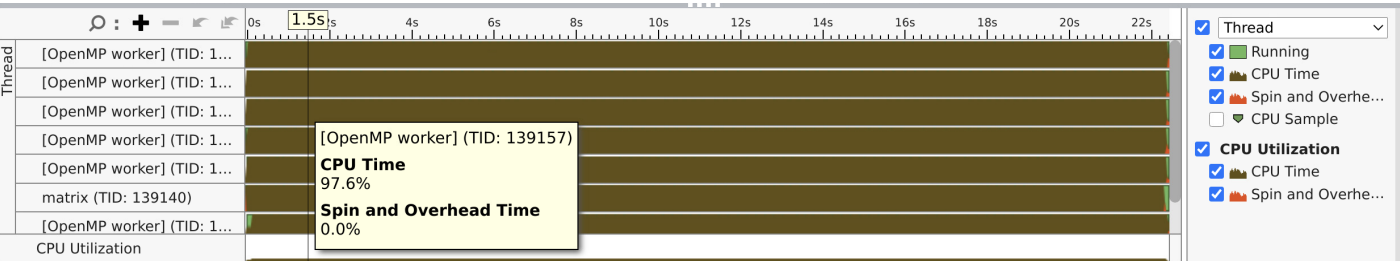
Top Hotspots

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 [?]
multiply1	matrix	134.600s
ThreadFunction	matrix	0.010s
printf	libc.so.6	0.010s

**N/A is applied to non-summable metrics.*

On the bottom section we can see a profile per thread, where we can see how well balanced is the application:



Memory consumption analysis

⌵

Elapsed Time[?]: 13.582s

Allocation Size:134.3 MB

Deallocation Size:134.3 MB

Allocations:26

Total Thread Count:16

Paused Time[?]:0s

⌵

Top Memory-Consuming Functions

This section lists the most memory-consuming functions in your application.

Function	Memory Consumption	Allocation/Deallocation Delta	Allocations	Module
main	134.2 MB	8.2 KB	5	matrix
__alloc_dir	32.8 KB	0.0 B	1	libc.so.6
multiply1	5.0 KB	5.0 KB	2	matrix
GetModelParams	208.0 B	208.0 B	1	matrix
[OpenMP fork]	136.0 B	136.0 B	1	libgomp.so.1
[Others]			16	N/A*

*N/A is applied to non-summable metrics.

Threading analysis:

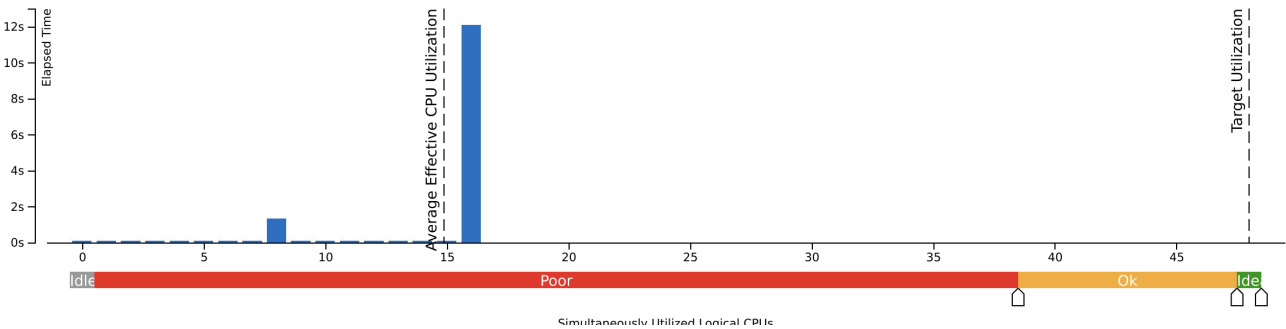
This graph shows the distribution of active threads for a given computation. We observe for this example 8 and 16 that run simultaneously.

⌵

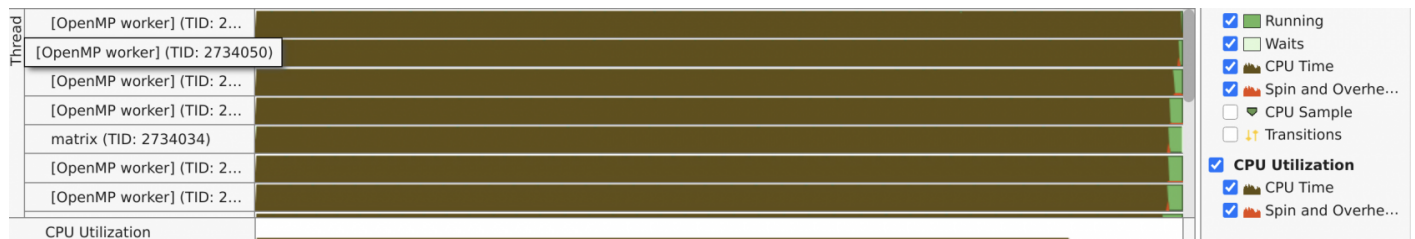
Effective CPU Utilization[?]: 31.0% (14.878 out of 48 logical CPUs)

⌵ 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.

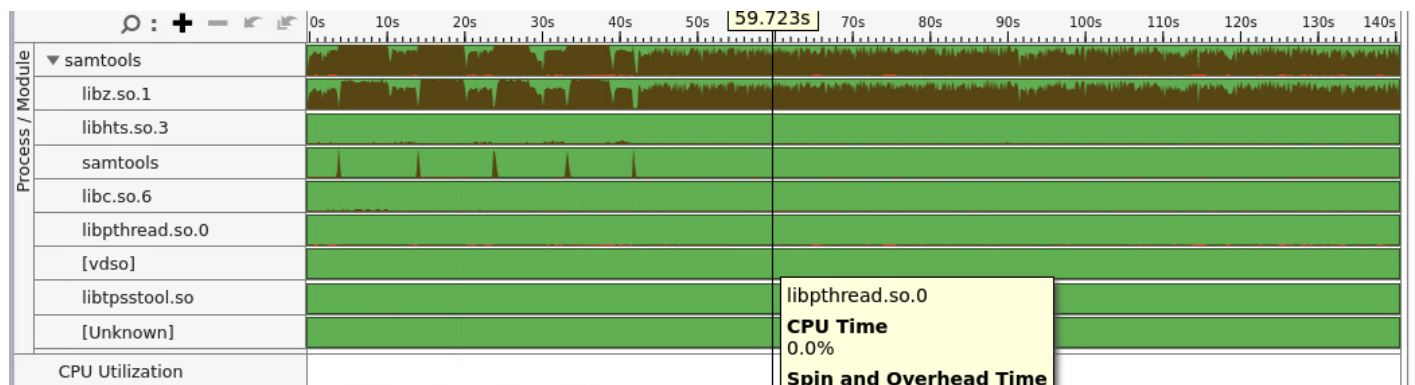


It shows more details:



Application using external libraries:

we can see how the CPU time was used by external libraries. This is accomplished by doing choosing process/module view.



Révision #23

Créé 7 avril 2022 11:55:15 par Cristian Ruiz

Mis à jour 22 mars 2023 12:26:43 par Cristian Ruiz