Legal Disclaimer & Optimization Notice

Performance optimization Vtune & Advisor

Paulius Velesko

paulius.velesko@intel.com

Application Engineer

Sample Code

git clone https://github.com/pvelesko/nbody-demo.git

Intel® Software Development Tools for Tuning

- **Compiler Optimization Reports** - Key to identify issues preventing automated optimization
- Intel® VTune™ Application Performance Snapshot - Overall performance
- ***Intel® Advisor*** *- Core and socket performance (vectorization and threading)*
- **Intel® *VTune*™** Amplifier - Node level performance (memory and more)
- Intel® Trace Analyzer and Collector - Cluster level performance (network)

Get the tools

Intel profiling tools are now FREE:

https://software.intel.com/en-us/vtune/choose-download

https://software.intel.com/en-us/advisor/choose-download

Agenda

- Optimize
  - Make it go fast
    - Vectorization
    - Memory
  - Make it scale
    - MPI
- Profiling AI/ML
- Get the example code:
  - git clone https://github.com/pvelesko/nbody-demo.git

Nbody demonstration

The naïve code that could

Nbody gravity simulation

forked from https://github.com/fbaru-dev/nbody-demo (Dr. Fabio Baruffa)

Let's consider a distribution of point masses located at r_1,…,r_n and have masses m_1,…,m_n.

We want to calculate the position of the particles after a certain time interval using the Newton law of gravity.

```
struct Particle
{
 public:
   Particle() { init();}
   void init()
   {
     pos[0] = 0.; pos[1] = 0.; pos[2] = 0.;
     vel[0] = 0.; vel[1] = 0.; vel[2] = 0.;
     acc[0] = 0.; acc[1] = 0.; acc[2] = 0.;
     mass   = 0.;
   }
   real_type pos[3];
   real_type vel[3];
   real_type acc[3];
   real_type mass;
};
```

```
for (i = 0; i < n; i++){         // update acceleration
   for (j = 0; j < n; j++){
     real_type distance, dx, dy, dz;
     real_type distanceSqr = 0.0;
     real_type distanceInv = 0.0;

     dx = particles[j].pos[0] - particles[i].pos[0];
     …

     distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;
     distanceInv = 1.0 / sqrt(distanceSqr);

     particles[i].acc[0] += dx * G * particles[j].mass *
                            distanceInv * distanceInv * distanceInv;
     particles[i].acc[1] += …
     particles[i].acc[2] += …
```

Intel® Compiler Reports

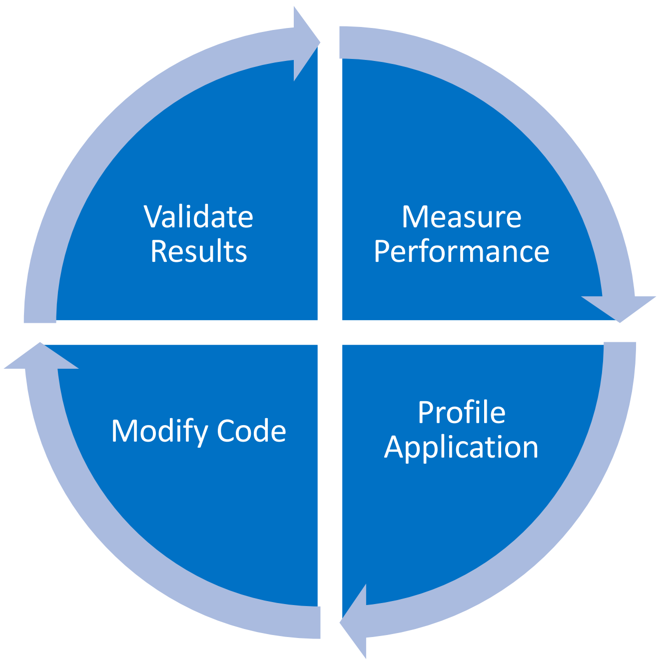
Generating the compiler report

```
cd ./nbody-demo/ver0
vim ./GSimulation.cpp # find the compute loop
vim ./Makefile; # add -qopt-report=5 flag
make
vim ./GSimulation.optrpt # search for the line number
```

Looking at the compiler report

```
LOOP BEGIN at GSimulation.cpp(127,20)
   remark #15542: loop was not vectorized: inner loop was already vectorized
   LOOP BEGIN at GSimulation.cpp(130,5)
      remark #15542: loop was not vectorized: inner loop was already vectorized
      LOOP BEGIN at GSimulation.cpp(132,7)
         remark #25085: Preprocess Loopnests: Moving Out Load and Store    [ GSimulation.cpp(145,4) ]
         remark #25085: Preprocess Loopnests: Moving Out Load and Store    [ GSimulation.cpp(146,4) ]
         remark #25085: Preprocess Loopnests: Moving Out Load and Store    [ GSimulation.cpp(147,4) ]
         remark #15415: vectorization support: non-unit strided load was generated for the variable <this->particles->pos[j][0]>, stride is 10   [ GSimulation.cpp(138,9) ]
         remark #15415: vectorization support: non-unit strided load was generated for the variable <this->particles->pos[j][1]>, stride is 10   [ GSimulation.cpp(139,9) ]
         remark #15415: vectorization support: non-unit strided load was generated for the variable <this->particles->pos[j][2]>, stride is 10   [ GSimulation.cpp(140,9) ]
         remark #15415: vectorization support: non-unit strided load was generated for the variable <this->particles->mass[j]>, stride is 10   [ GSimulation.cpp(145,36) ]
         remark #15415: vectorization support: non-unit strided load was generated for the variable <this->particles->mass[j]>, stride is 10   [ GSimulation.cpp(146,36) ]
         remark #15415: vectorization support: non-unit strided load was generated for the variable <this->particles->mass[j]>, stride is 10   [ GSimulation.cpp(147,36) ]
         remark #15305: vectorization support: vector length 16
         remark #15309: vectorization support: normalized vectorization overhead 0.356
         remark #15417: vectorization support: number of FP up converts: single precision to double precision 1   [ GSimulation.cpp(143,4) ]
         remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   [ GSimulation.cpp(143,4) ]
         remark #15417: vectorization support: number of FP up converts: single precision to double precision 6   [ GSimulation.cpp(145,4) ]
         remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   [ GSimulation.cpp(145,4) ]
         remark #15417: vectorization support: number of FP up converts: single precision to double precision 6   [ GSimulation.cpp(146,4) ]
         remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   [ GSimulation.cpp(146,4) ]
         remark #15417: vectorization support: number of FP up converts: single precision to double precision 6   [ GSimulation.cpp(147,4) ]
         remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   [ GSimulation.cpp(147,4) ]
         remark #15300: LOOP WAS VECTORIZED
         remark #15452: unmasked strided loads: 6
         remark #15475: --- begin vector cost summary ---
         remark #15476: scalar cost: 137
         remark #15477: vector cost: 20.000
         remark #15478: estimated potential speedup: 6.300
         remark #15487: type converts: 23
         remark #15488: --- end vector cost summary ---
      LOOP END
```

The Basic Tuning Cycle



Infinite cycle only broken by external constraints (time, papers, releases …)

Procedures for measuring performance and validating results are critical

Automation and **environment** control are key for **consistency**

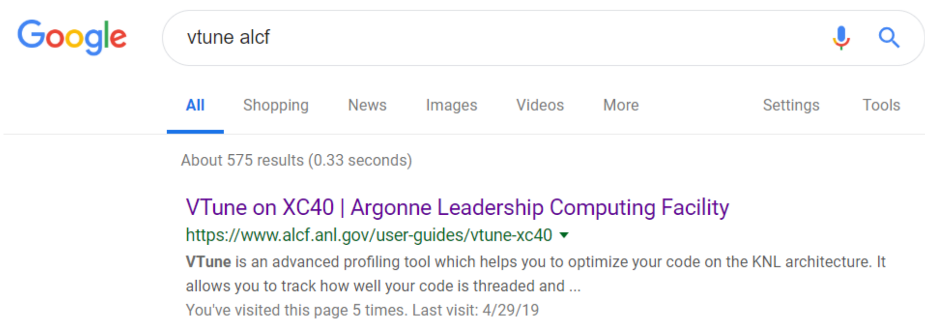
Where do I start?

/soft/perftools/intel/advisor/advixe.qsub

/soft/perftools/intel/vtune/amplxe.qsub

amplxe.qsub Script

- Copy and customize the script from /soft/perftools/intel/vtune/amplxe.qsub
- All-in-one script for profiling
  - Job size - ranks, threads, hyperthreads, affinity
  - **Attach to a single, multiple or all ranks**
  - Binary as arg#1, input as arg#2
    - qsub amplxe.qsub ./your_exe ./inputs/inp
  - Binary and source search directory locations
  - Timestamp + binary name + input name as result directory
  - Save cobalt job files to result directory



Intel® Advisor

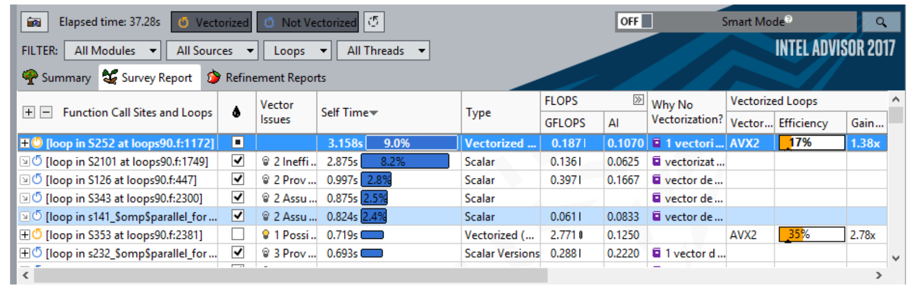
Intel® Advisor – Vectorization Optimization

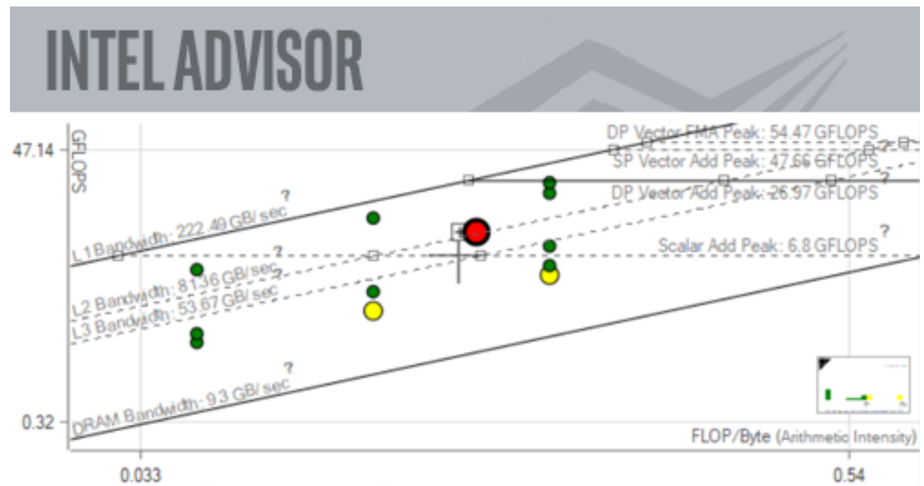
Faster Vectorization Optimization:

- Vectorize where it will pay off most
- Quickly ID what is blocking vectorization
- Tips for effective vectorization
- Safely force compiler vectorization
- Optimize memory stride

Roofline model analysis:

- Automatically generate roofline model
- Evaluate current performance
- Identify boundedness





Add Parallelism with Less Effort, Less Risk and More Impact

http://intel.ly/advisor-xe

Typical Vectorization Optimization Workflow

There is no need to recompile or relink the application, but the use of **-g** is recommended.

Note: if you're using Theta run out of /projects rather than /home

1. Collect survey (overhead ~5%) advixe-cl -c survey
   - Basic info (static analysis) - ISA, time spent, etc.
2. Collect Tripcounts and Flops (overhead 1-10x) advixe-cl -c tripcounts -flop
   - Investigate application place within roofline model
   - Determine vectorization efficiency and opportunities for improvement
3. Collect dependencies (overhead 5-1000x) advixe-cl -c dependencies
   - Differentiate between real and assumed issues blocking vectorization
4. Collect Memory Access Patterns advixe-cl -c map

Collect survey and tripcounts

```
cd /projects/intel/pvelesko/nody-demo/ver0
make
cp /soft/perftools/intel/advisor/advixe.qsub ./
qsub ./advixe.qsub ./nbody.x 2000 500
```

scp result back to your local machine

Text report can also be useful:

```
advixe-cl -R survey
```

View Result

X-forwarding is not recommended.

Tar the result along with sources (if you want to be able to view them)

or

Generate a snapshot:

```
$ advixe-cl --snapshot --pack --cache-sources --cache-binaries
```

then scp to your local machine

Analyze Result - advixe_ver0

Summary - ISA

CPU Time - Total vs Self

- Loops and Functions/Loops Only/Functions only

Top Down

- helpful when same function is called in multiple places

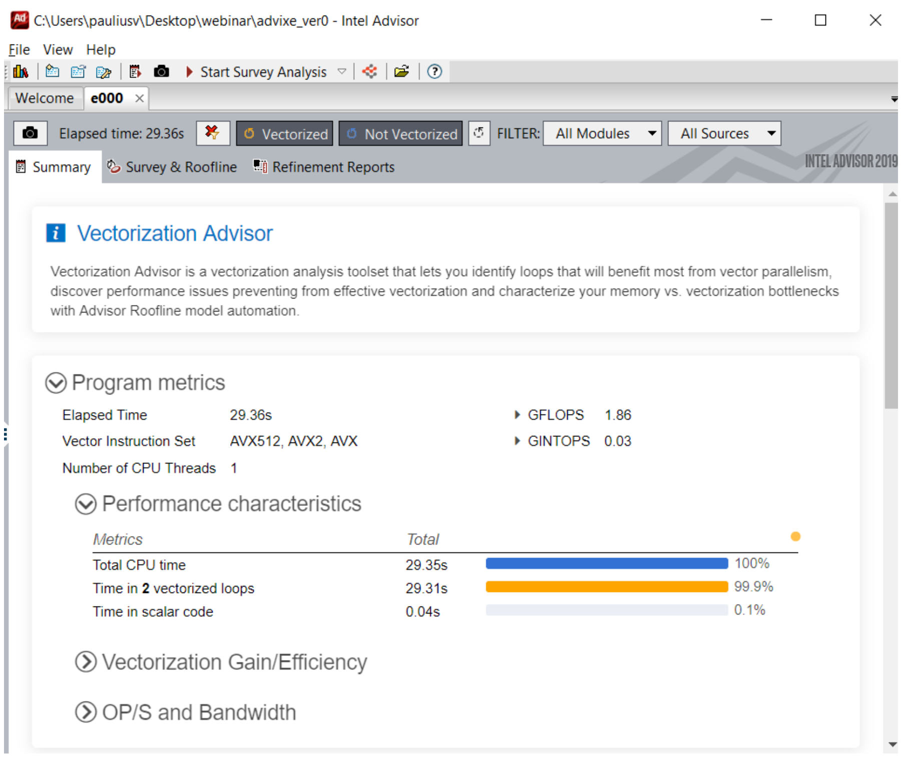
Compute Perf - FLOPs

Roofline

Click me! Advisor snapshot

ver0.advixeexpz

Summary Report



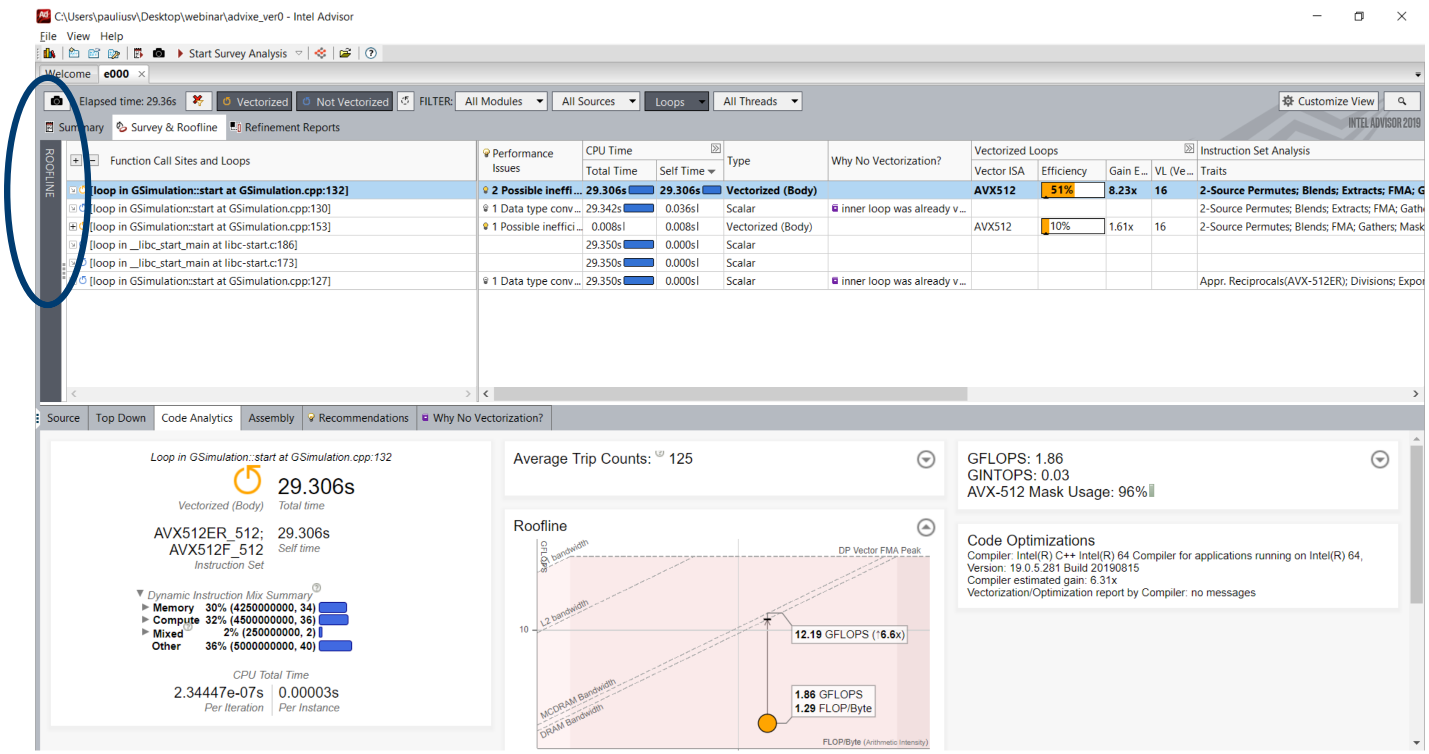
Summary provides overall performance characteristics

Top time consuming loops are listed individually

Vectorization efficiency is based on used ISA (in this case SSE2/SSE)

Note the warning regarding a higher ISA (in this case -xMIC-AVX512)

Survey Report (Code Analytics Tab)

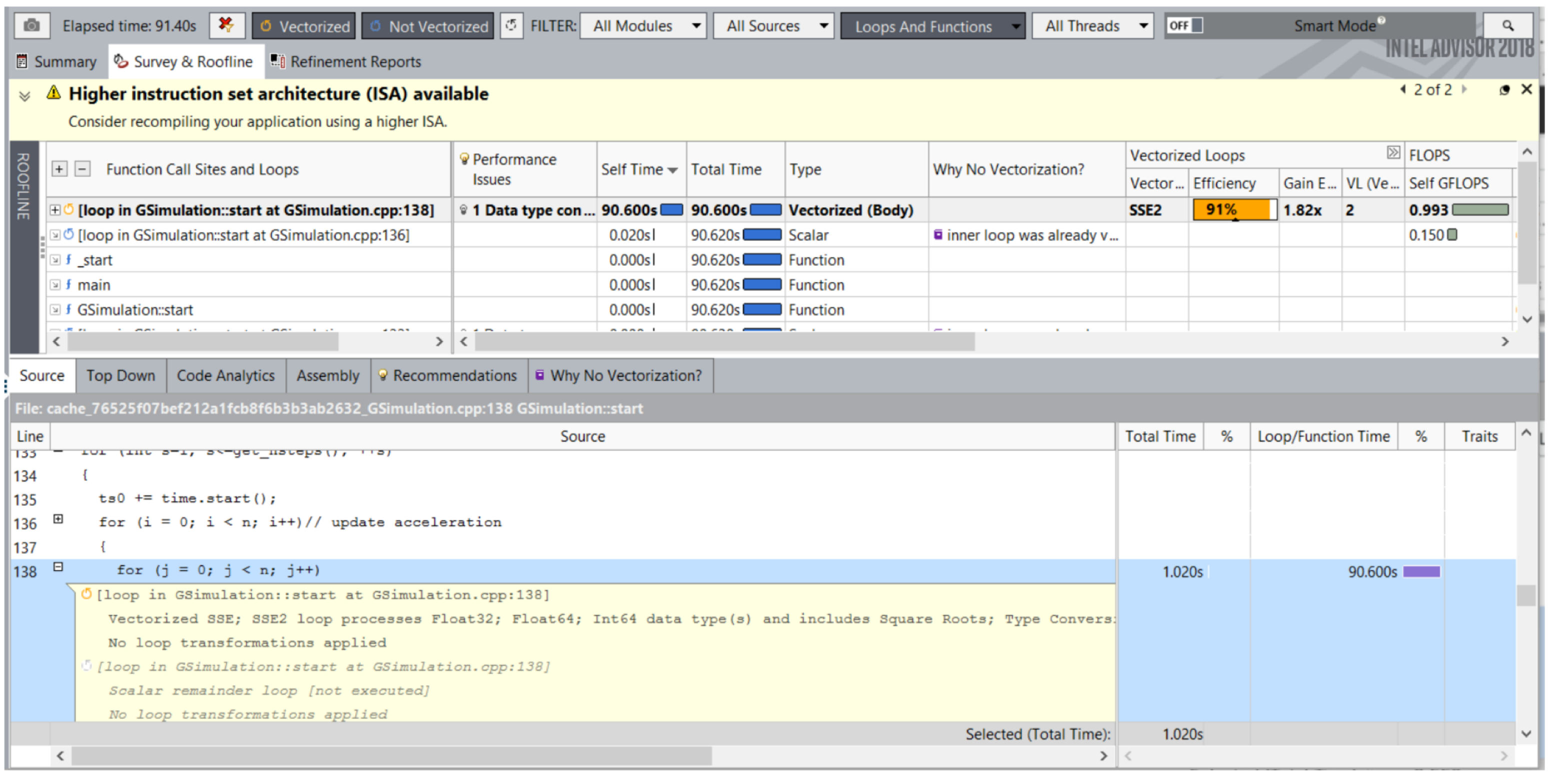


Analytics tab contains a wealth of information

- Instruction set
- Instruction mix
- Traits (sqrt, type conversions, unpacks)
- Vector efficiency
- Floating point statistics

And explanations on how they are measured or calculated - expand the box or hover over the question marks.

Survey Report (Source Tab)

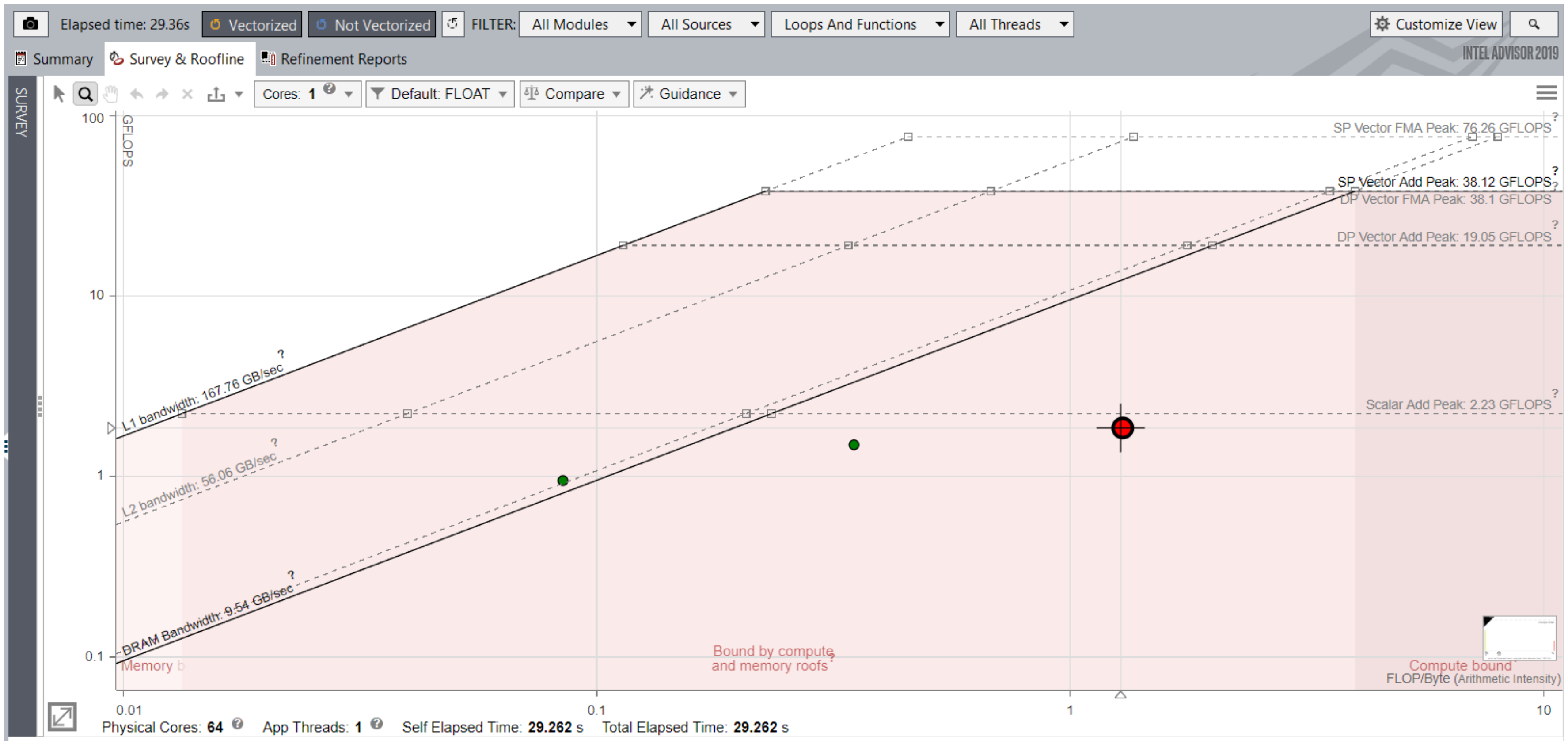


Notice the following:

- Higher ISA available
- Type conversion
- Use of square root

All of these elements may affect performance

Cache-Aware Roofline Model (CARM) Analysis



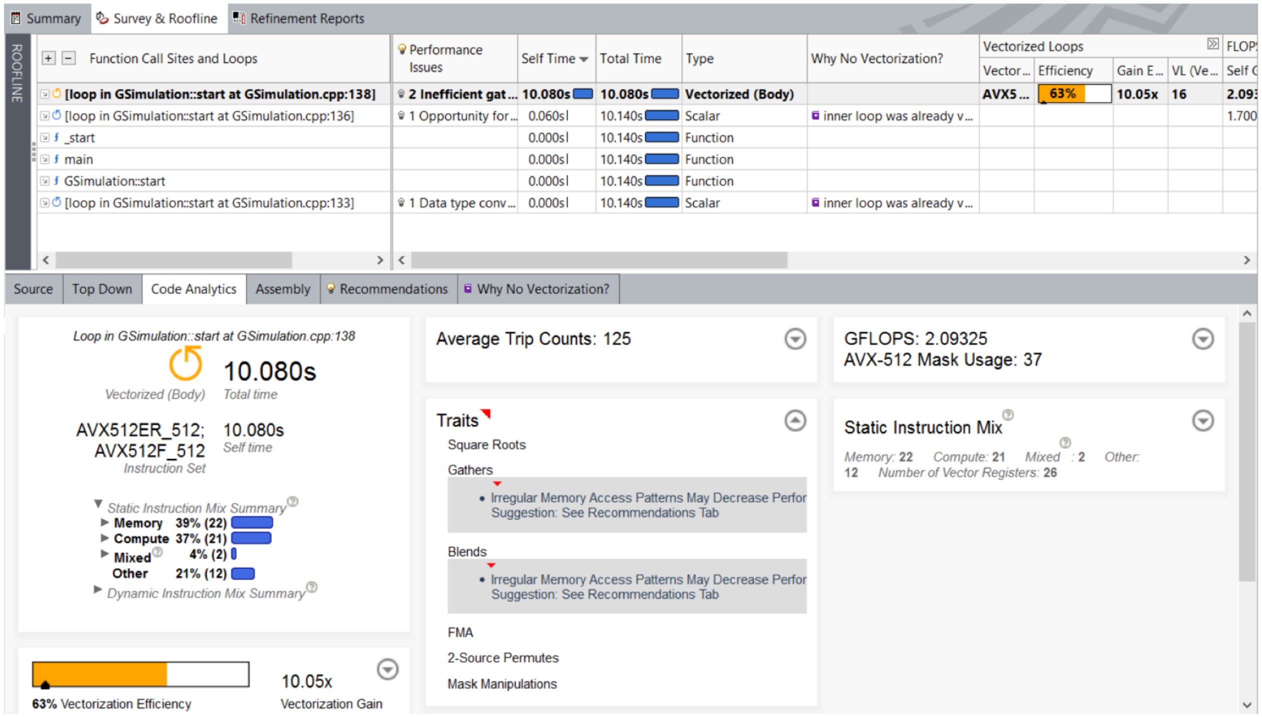
Follow recommendations and re-test

In this new version (ver2 in github sample) we introduce the following changes:

- Consistently use float types to avoid type conversions in GSimulation.cpp
- Recompile to target Intel® Xeon Phi 7230 with -xMIC-AVX512

Note changes in survey report:

- Reduced vectorization efficiency (harder with 512 bits)
- Type conversions gone
- Gathers/Blends point to memory issues and vector inefficiencies



Analyze Result - advixe_ver2

Roofline -

- Change in OI (due to FP converts)
- Jump in FLOPs

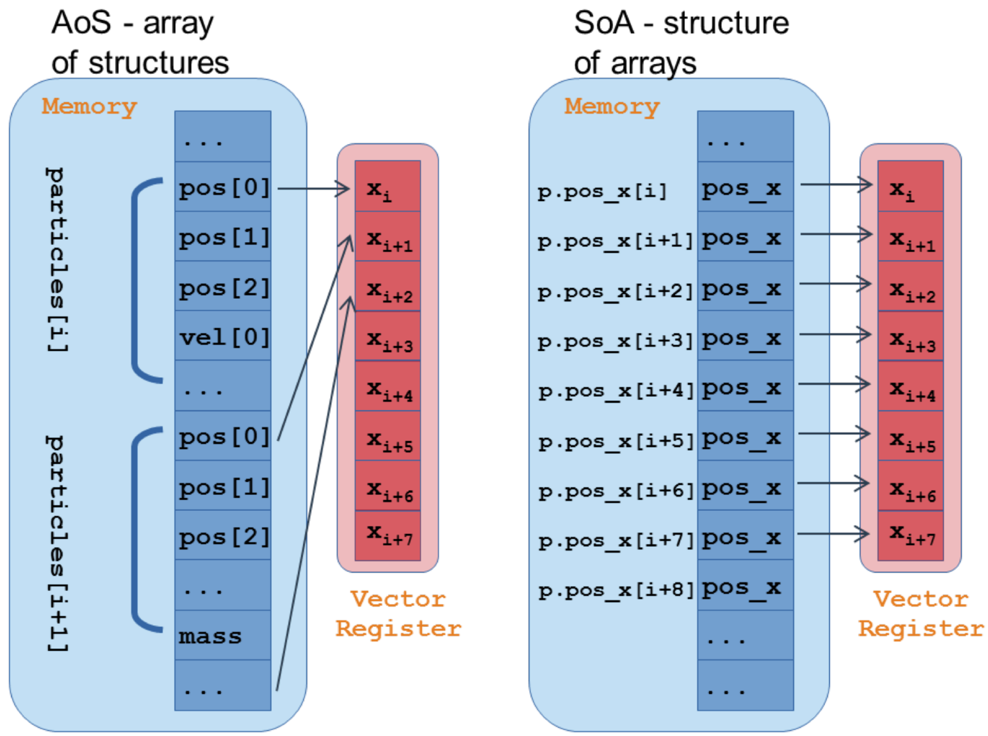
Memory Access

Vectorization: gather/scatter operation

The compiler might generate gather/scatter instructions for loops automatically vectorized where memory locations are not contiguous

```
struct Particle
{
 public:
   ...
   real_type pos[3];
   real_type vel[3];
   real_type acc[3];
   real_type mass;
};
```

```
struct ParticleSoA
{
 public:
   ...
   real_type *pos_x,*pos_y,*pos_z;
   real_type *vel_x,*vel_y,*vel_z;
   real_type *acc_x,*acc_y;*acc_z
   real_type *mass;
};
```



Memory access pattern analysis

How should I access data ?

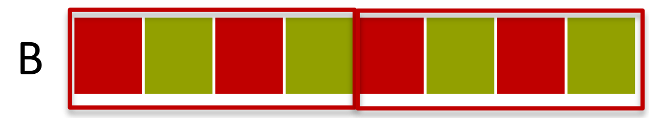
Unit stride access are faster

```
for (i=0; i<N; i++)
    A[i] = B[i]*d
```

For B, 1 cache line load computes 4 DP

Constant stride are more complex

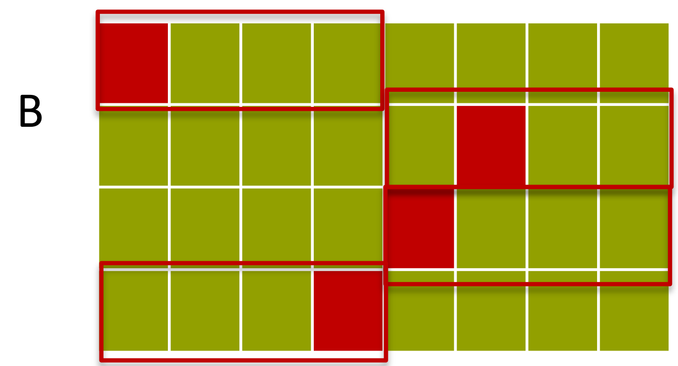
```
for (i=0; i<N; i+=2)
    A[i] = B[i]*d
```



For B, 2 cache line loads compute 4 DP with reconstructions

Non predictable access are usually bad

```
for (i=0; i<N; i++)
    A[i] = B[C[i]]*d
```



For B, 4 cache line loads compute 4 DP with reconstructions, prefetching might not work

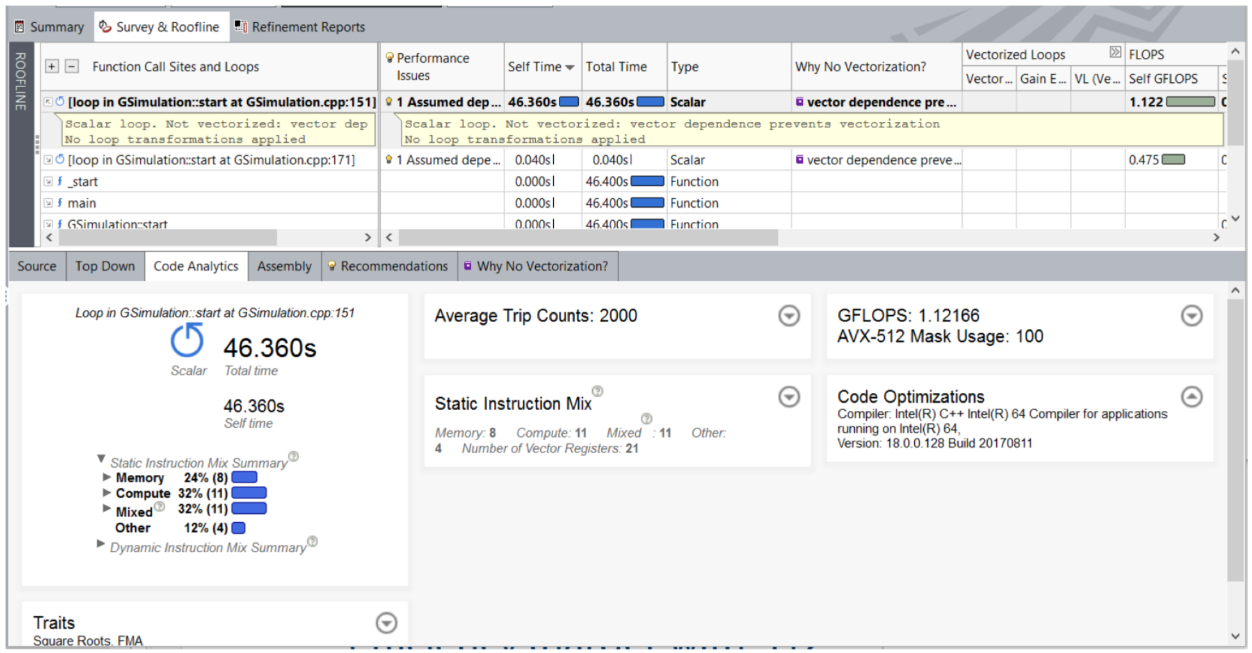
Follow recommendations and re-test

In this new version (ver3 in github sample) we introduce the following change:

- Change particle data structures from AOS to SOA

Note changes in report:

- Performance is lower
- Main loop is no longer vectorized
- Assumed vector dependence prevents automatic vectorization



Next step is clear: perform a **Dependencies** analysis

Suggested solutions

Memory Access Patterns Report | Dependencies Report | Recommendations

All Advisor-detectable issues: C++ | Fortran

Recommendation: Resolve dependency

The Dependencies analysis shows there is a real (proven) dependency in the loop. To fix: Do one of the following:

- If there is an anti-dependency, enable vectorization using the directive `#pragma omp simd safelen(length)` , where `length` is smaller than the distance between dependent iterations in anti-dependency. For example:

```
#pragma omp simd safelen(4)
for (i = 0; i < n - 4; i += 4)
{
    a[i + 4] = a[i] * c;
}
```

- If there is a reduction pattern dependency in the loop, enable vectorization using the directive `#pragma omp simd reduction(operator:list)` . For example:

```
#pragma omp simd reduction(+:sumx)
for (k = 0;k < size2; k++)
{
    sumx += x[k]*b[k];
}
```

ISSUE: PROVEN (REAL) DEPENDENCY PRESENT

The compiler assumed there is an anti-dependency (Write after read - WAR) or true dependency (Read after write - RAW) in the loop. Improve performance by investigating the assumption and handling accordingly.

Resolve dependency

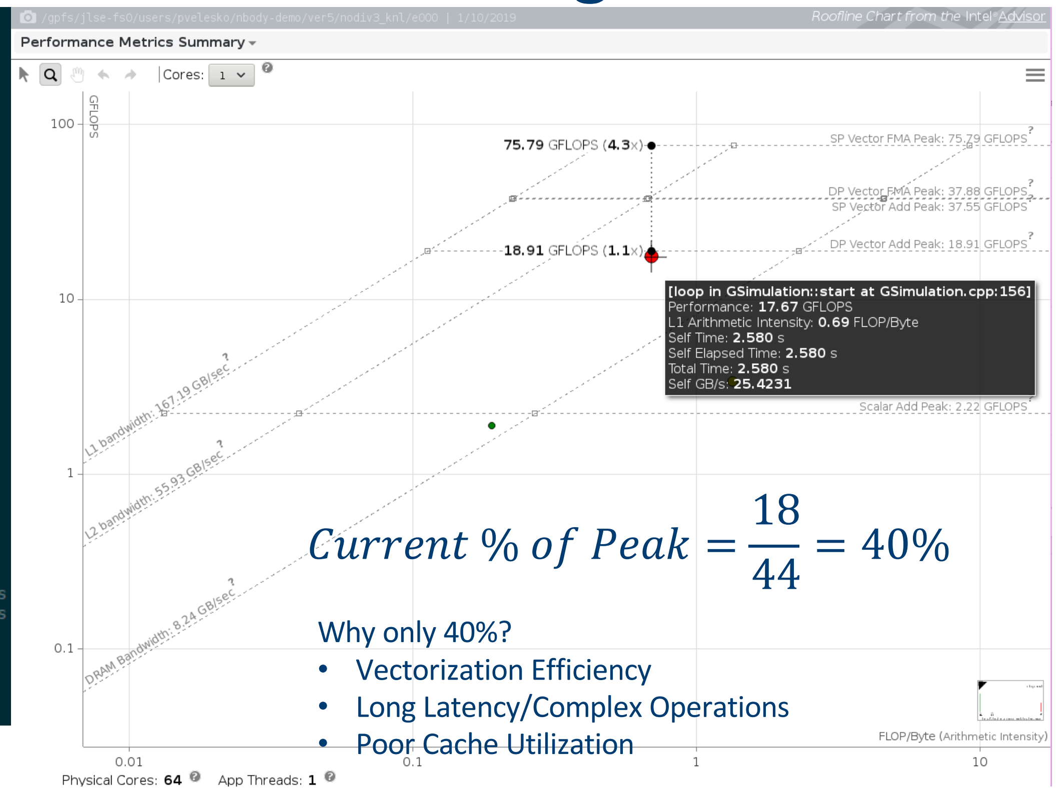
Analyze Result - advixe_ver4

Vectorization time back to normal

Reduced execution time

Advisor Roofline – How much further can we go?

```
    for (i = 0; i < n; i++)// update acceleration
    {
#ifdef ASALIGN
      __assume_aligned(particles->pos_x, alignment);
      __assume_aligned(particles->pos_y, alignment);
      __assume_aligned(particles->pos_z, alignment);
      __assume_aligned(particles->acc_x, alignment);
      __assume_aligned(particles->acc_y, alignment);
      __assume_aligned(particles->acc_z, alignment);
      __assume_aligned(particles->mass, alignment);
#endif
      real_type ax_i = particles->acc_x[i];
      real_type ay_i = particles->acc_y[i];
      real_type az_i = particles->acc_z[i];
#pragma omp simd simdlen(16) reduction(+:ax_i, ay_i, az_i)
      for (j = 0; j < n; j++)
      {
          real_type dx, dy, dz;
      real_type distanceSqr = 0.0f;
      real_type distanceInv = 0.0f;

      dx = particles->pos_x[j] - particles->pos_x[i];    //1flop
      dy = particles->pos_y[j] - particles->pos_y[i];    //1flop
      dz = particles->pos_z[j] - particles->pos_z[i];    //1flop

      distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;    //6flops
      distanceInv = 1.0f / sqrtf(distanceSqr);               //1div+1sqrt

      ax_i+= dx * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops
      ay_i += dy * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops
      az_i += dz * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops
      }
      particles->acc_x[i] = ax_i;
      particles->acc_y[i] = ay_i;
      particles->acc_z[i] = az_i;
    }
```



$$Current\ \%\ of\ Peak = \frac{18}{44} = 40\%$$

Why only 40%?

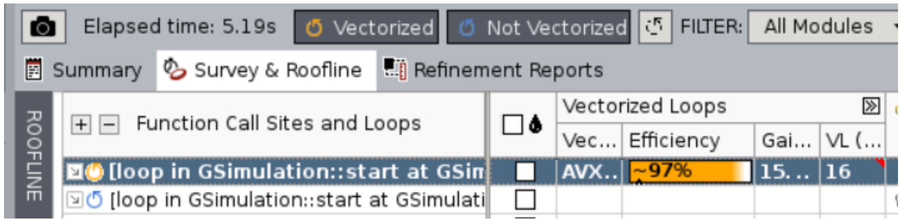
- Vectorization Efficiency
- Long Latency/Complex Operations
- Poor Cache Utilization

$$FMA\ Ratio = \frac{3}{29} = 10\%$$

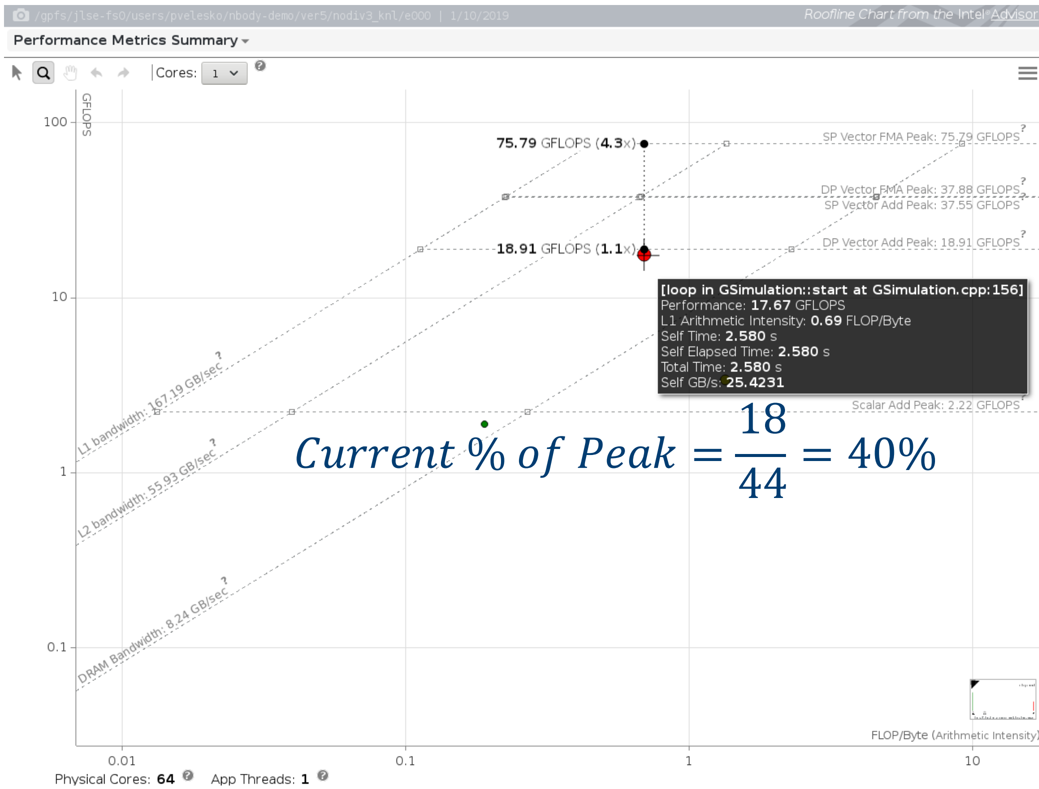
Peak = SP Vector ADD * (1+ FMA Ratio)

Peak = 40 * (1 + 0.1) = 44 GFLOPS

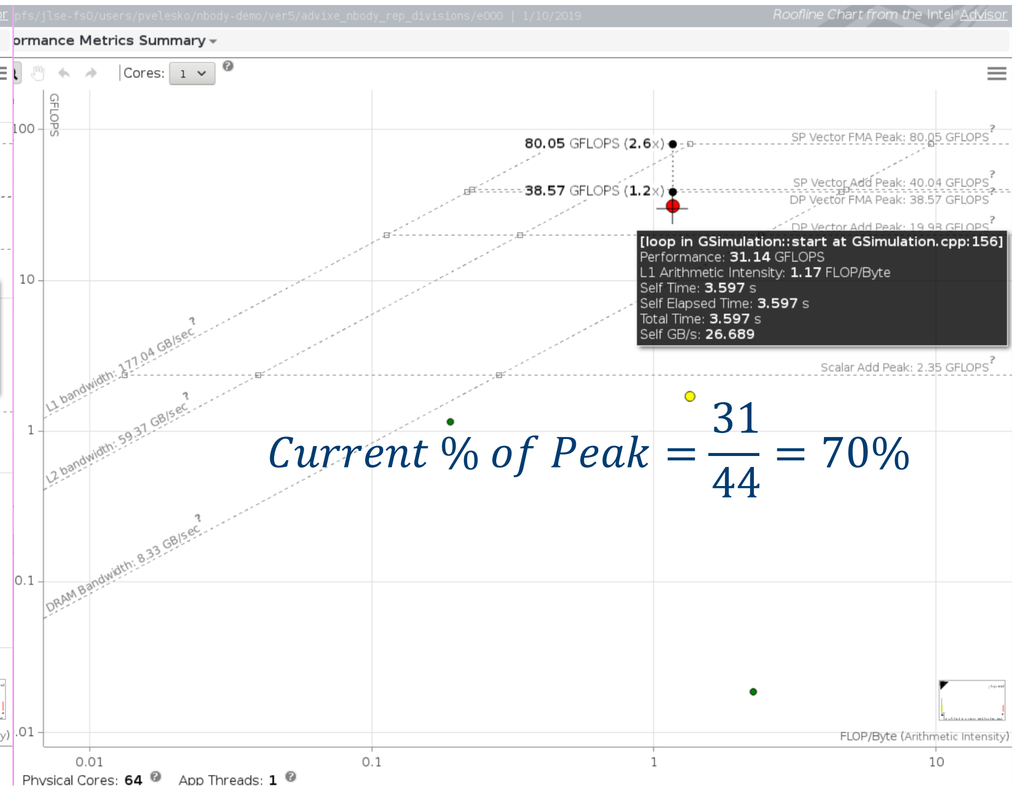
Vectorization Efficiency?



Complex Operations?



$$Current\ \%\ of\ Peak = \frac{18}{44} = 40\%$$



$$Current\ \%\ of\ Peak = \frac{31}{44} = 70\%$$

Memory Performance

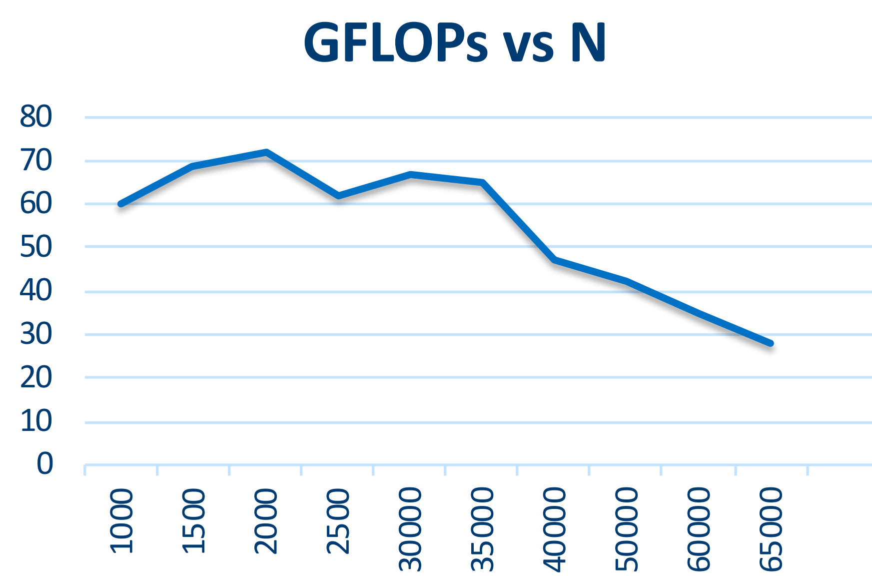
```
    for (i = 0; i < n; i++)// update acceleration
    {
#ifdef ASALIGN
      __assume_aligned(particles->pos_x, alignment);
      __assume_aligned(particles->pos_y, alignment);
      __assume_aligned(particles->pos_z, alignment);
      __assume_aligned(particles->acc_x, alignment);
      __assume_aligned(particles->acc_y, alignment);
      __assume_aligned(particles->acc_z, alignment);
      __assume_aligned(particles->mass, alignment);
#endif
      real_type ax_i = particles->acc_x[i];
      real_type ay_i = particles->acc_y[i];
      real_type az_i = particles->acc_z[i];
#pragma omp simd simdlen(16) reduction(+:ax_i, ay_i, az_i)
      for (j = 0; j < n; j++)
      {
          real_type dx, dy, dz;
      real_type distanceSqr = 0.0f;
      real_type distanceInv = 0.0f;

      dx = particles->pos_x[j] - particles->pos_x[i];    //1flop
      dy = particles->pos_y[j] - particles->pos_y[i];    //1flop
      dz = particles->pos_z[j] - particles->pos_z[i];    //1flop

      distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;    //6flops
      distanceInv = 1.0f / sqrtf(distanceSqr);           //1div+1sqrt

      ax_i+= dx * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops
      ay_i += dy * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops
      az_i += dz * G * particles->mass[j] * distanceInv * distanceInv * distanceInv; //6flops
      }
      particles->acc_x[i] = ax_i;
      particles->acc_y[i] = ay_i;
      particles->acc_z[i] = az_i;
    }
```

Maximum N before we lose caching?

KNL L1-32kB L2-1MB (1 tile/2cores)

32k/(4*4) = 2k (L1)

1MB/(7*4) = 35.7k(L2)



Intel® VTUNE™ Amplifier

Intel® VTune™ Amplifier

VTune Amplifier is a full system profiler

- Accurate
- Low overhead
- Comprehensive (microarchitecture, memory, IO, treading, …)
- Highly customizable interface
- Direct access to source code and assembly
- User-mode driverless sampling
- Event-based sampling

Analyzing code access to shared resources is critical to achieve good performance on multicore and manycore systems

Predefined Collections

Many available analysis types:

uarch-exploration	General microarchitecture exploration
hpc-performance	HPC Performance Characterization
memory-access	Memory Access
disk-io	Disk Input and Output
concurrency	Concurrency
gpu-hotspots	GPU Hotspots
gpu-profiling	GPU In-kernel Profiling
hotspots	Basic Hotspots
locksandwaits	Locks and Waits
memory-consumption	Memory Consumption
system-overview	System Overview
…	

Python Support

Collect uarch-exploration

cd /projects/intel/pvelesko/nody-demo/ver7

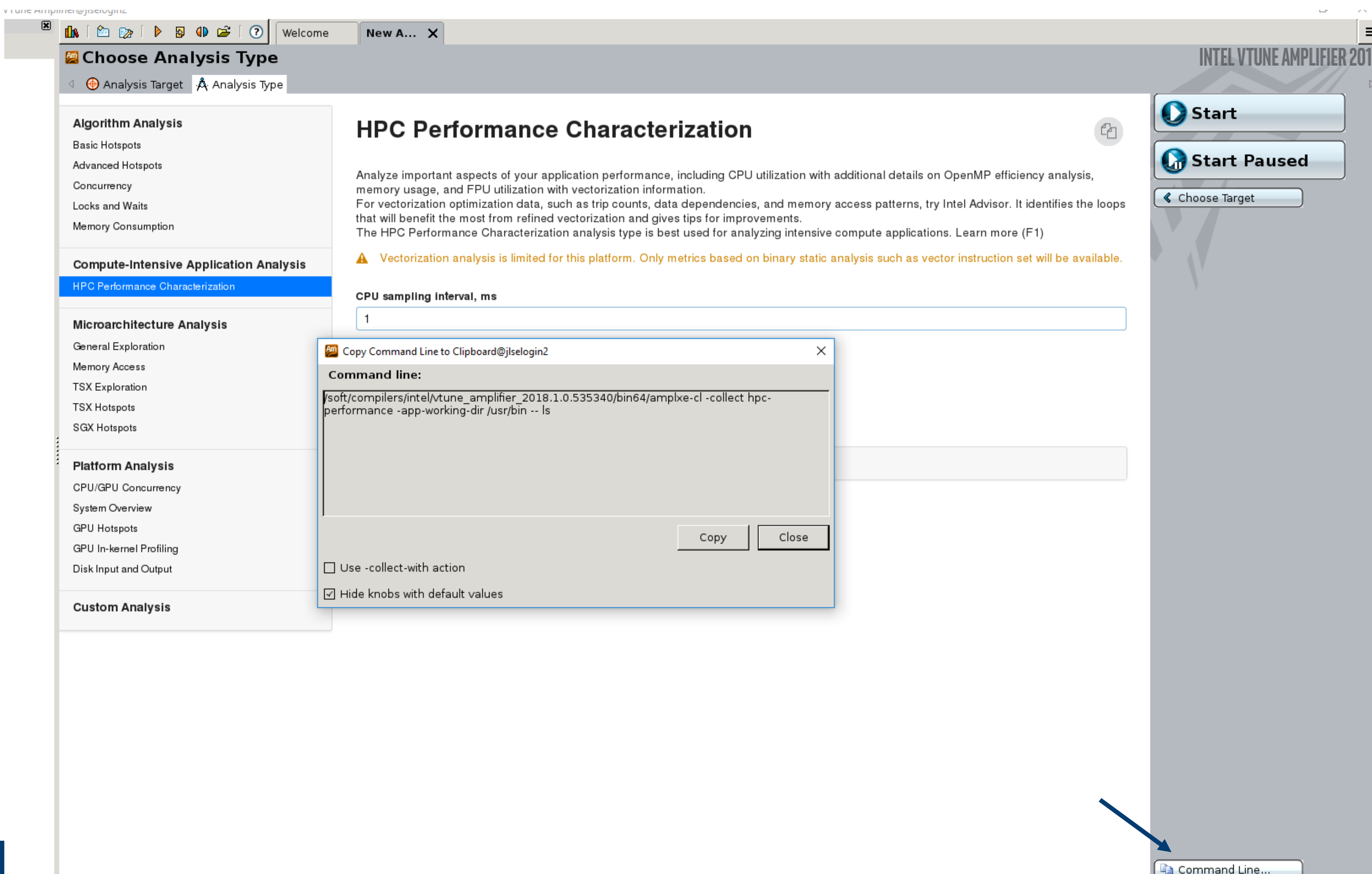
vim Makefile # edit to add -dynamic

cp /soft/perftools/intel/advisor/amplxe.qsub ./

vim amplxe.qsub # edit collection to “uarch-exploration”

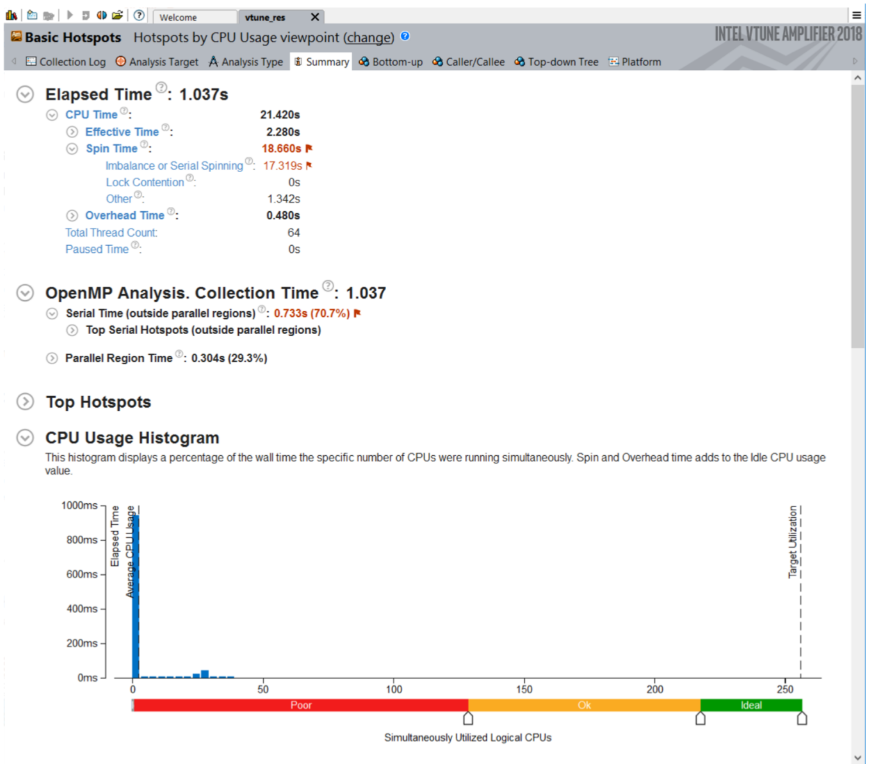
qsub ./advixe.qsub ./nbody.x 2000 500

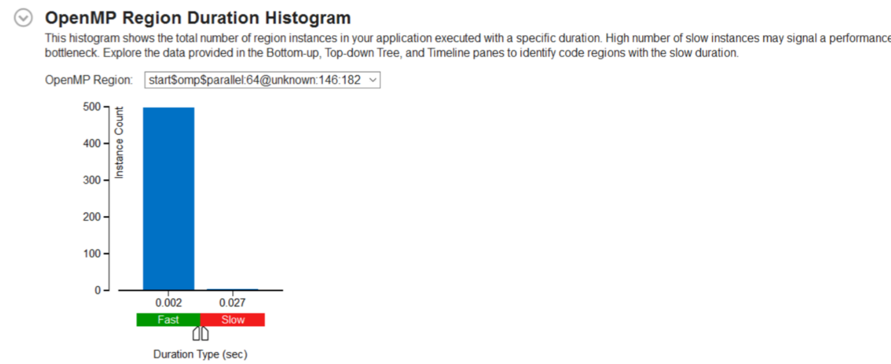
scp result back to your local machine


Welcome
New A...
Choose Analysis Type
INTEL VTUNE AMPLIFIER 2018
Analysis Target
Analysis Type
Algorithm Analysis
Basic Hotspots
Advanced Hotspots
Concurrency
Locks and Waits
Memory Consumption
Compute-Intensive Application Analysis
HPC Performance Characterization
Microarchitecture Analysis
General Exploration
Memory Access
TSX Exploration
TSX Hotspots
SGX Hotspots
Platform Analysis
CPU/GPU Concurrency
System Overview
GPU Hotspots
GPU In-kernel Profiling
Disk Input and Output
Custom Analysis
HPC Performance Characterization
Analyze important aspects of your application performance, including CPU utilization with additional details on OpenMP efficiency analysis, memory usage, and FPU utilization with vectorization information.
For vectorization optimization data, such as trip counts, data dependencies, and memory access patterns, try Intel Advisor. It identifies the loops that will benefit the most from refined vectorization and gives tips for improvements.
The HPC Performance Characterization analysis type is best used for analyzing intensive compute applications. Learn more (F1)
Vectorization analysis is limited for this platform. Only metrics based on binary static analysis such as vector instruction set will be available.
CPU sampling interval, ms
1
Copy Command Line to Clipboard@jlselogin2
Command line:
/soft/compilers/intel/vtune_amplifier_2018.1.0.535340/bin64/amplxe-cl -collect hpc-performance -app-working-dir /usr/bin -- ls
Copy
Close
Use -collect-with action
Hide knobs with default values
Start
Start Paused
Choose Target
Command Line...

Hotspots analysis for nbody demo (ver7: threaded)

- qsub amplxe.qsub ./your_exe ./inputs/inp



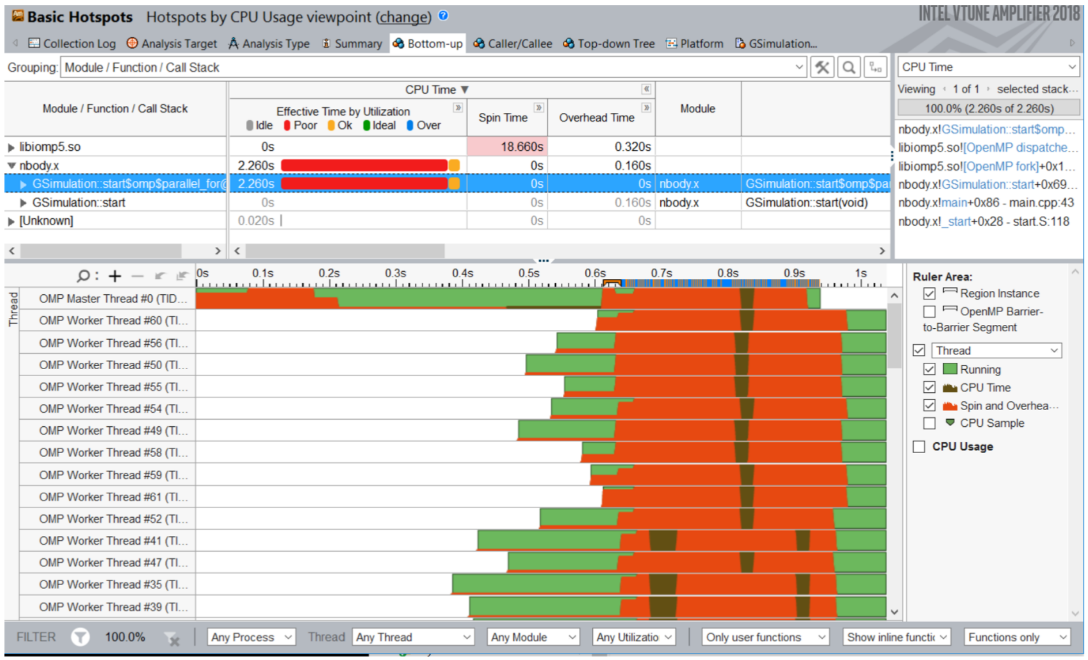


Lots of spin time indicate issues with load balance and synchronization

Given the short OpenMP region duration it is likely we do not have sufficient work per thread

Let's look a the timeline for each thread to understand things better…

Bottom-up Hotspots view

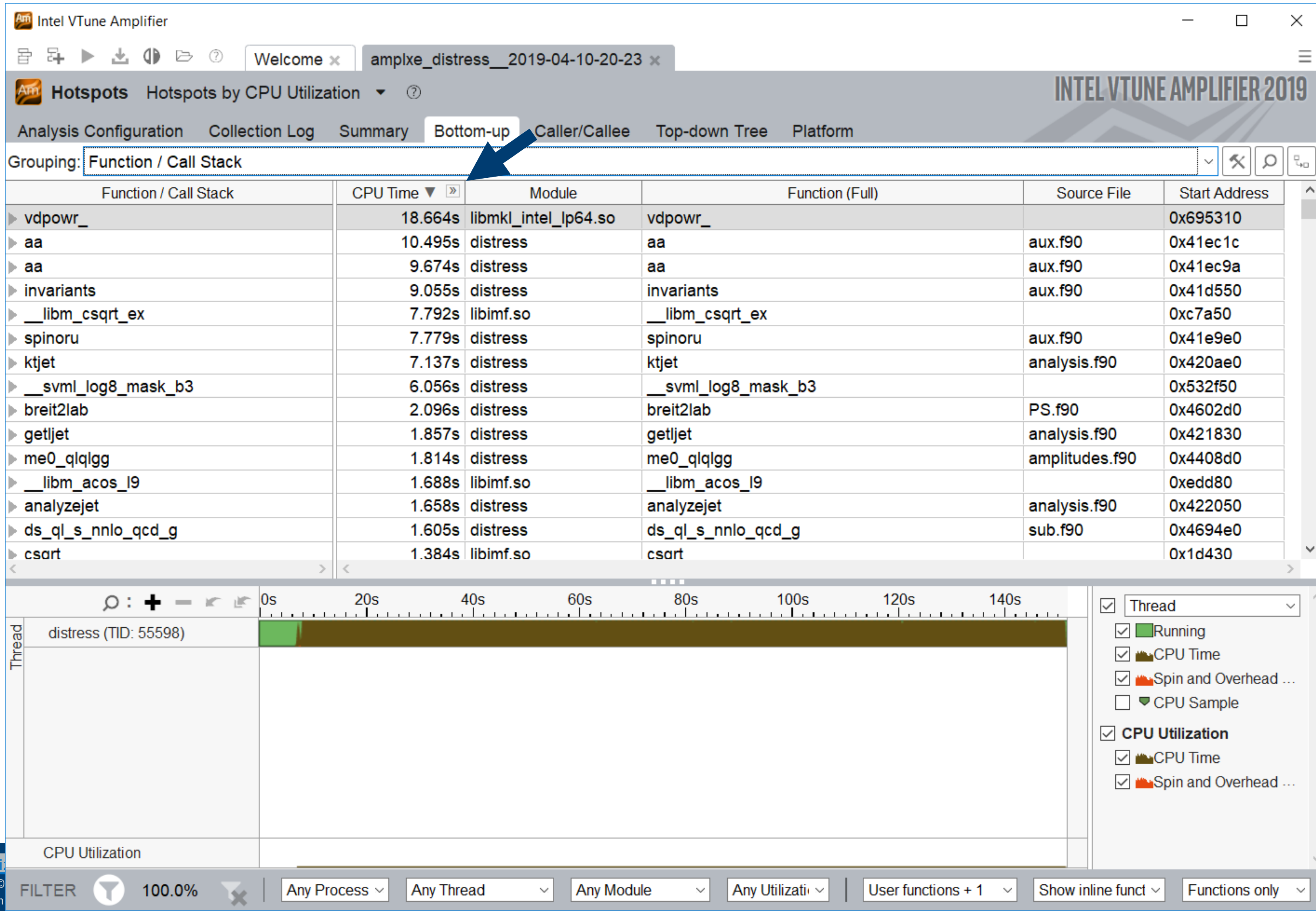


There is not enough work per thread in this particular example.

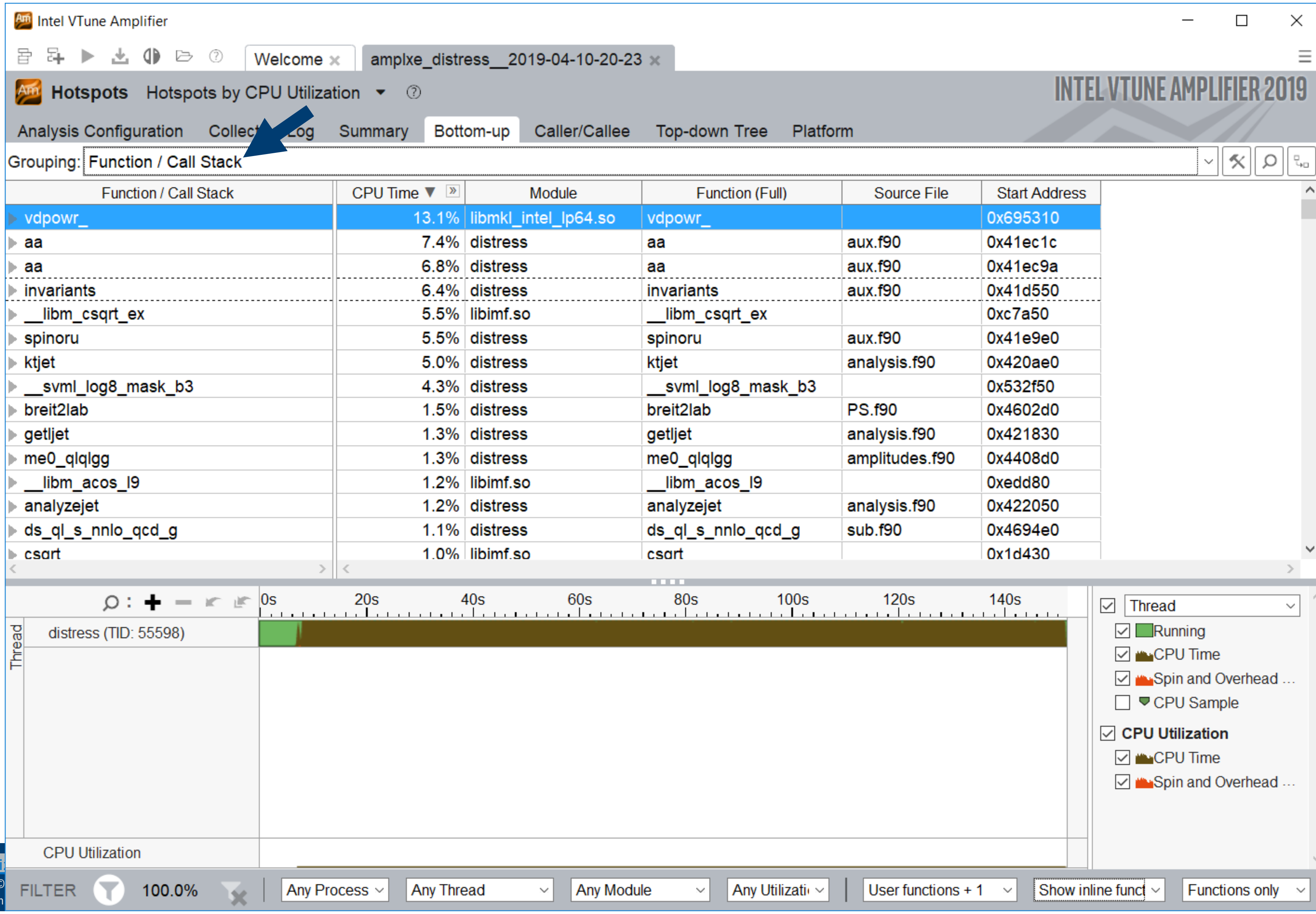
Double click on line to access source and assembly.

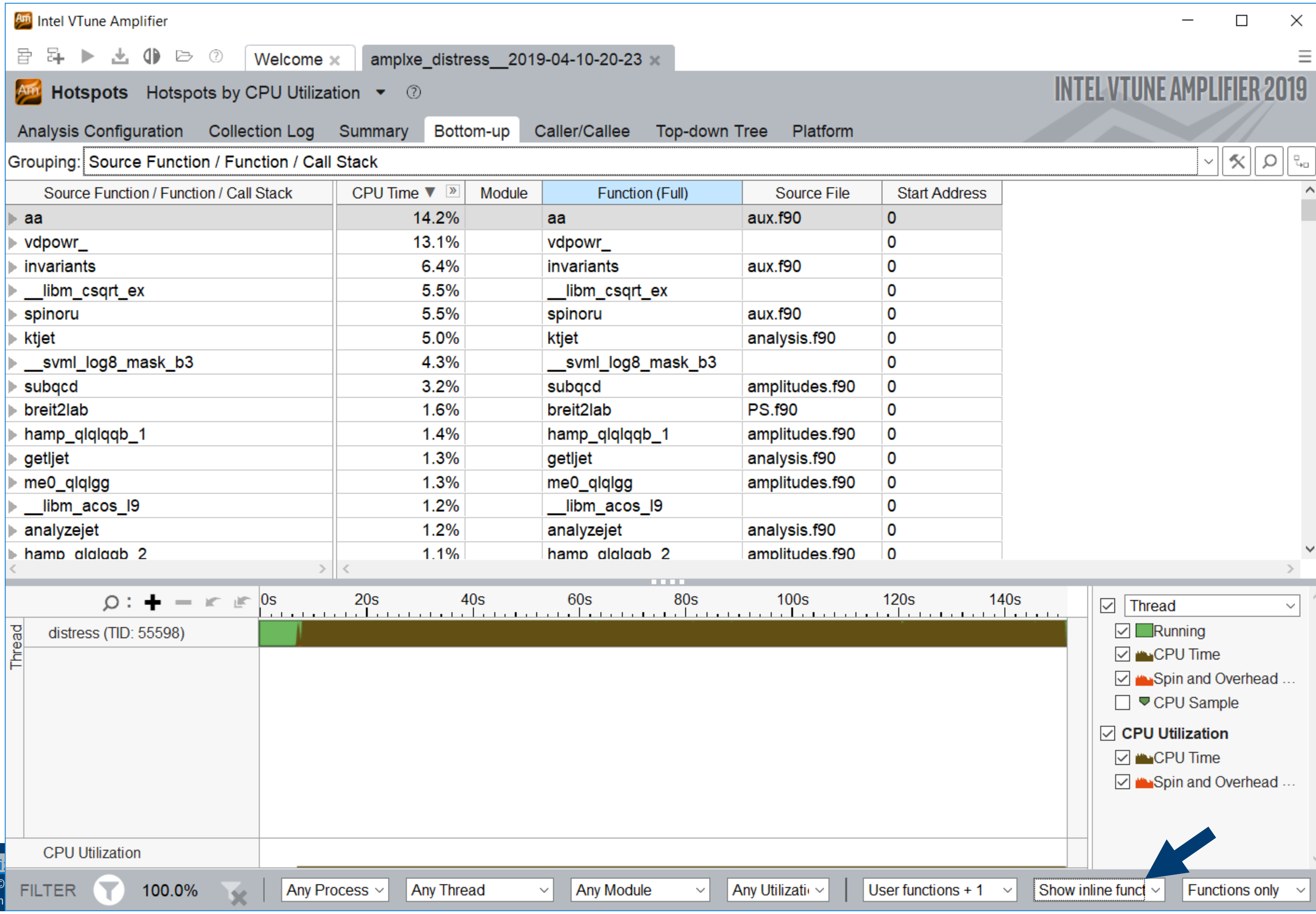
Notice the filtering options at the bottom, which allow customization of this view.

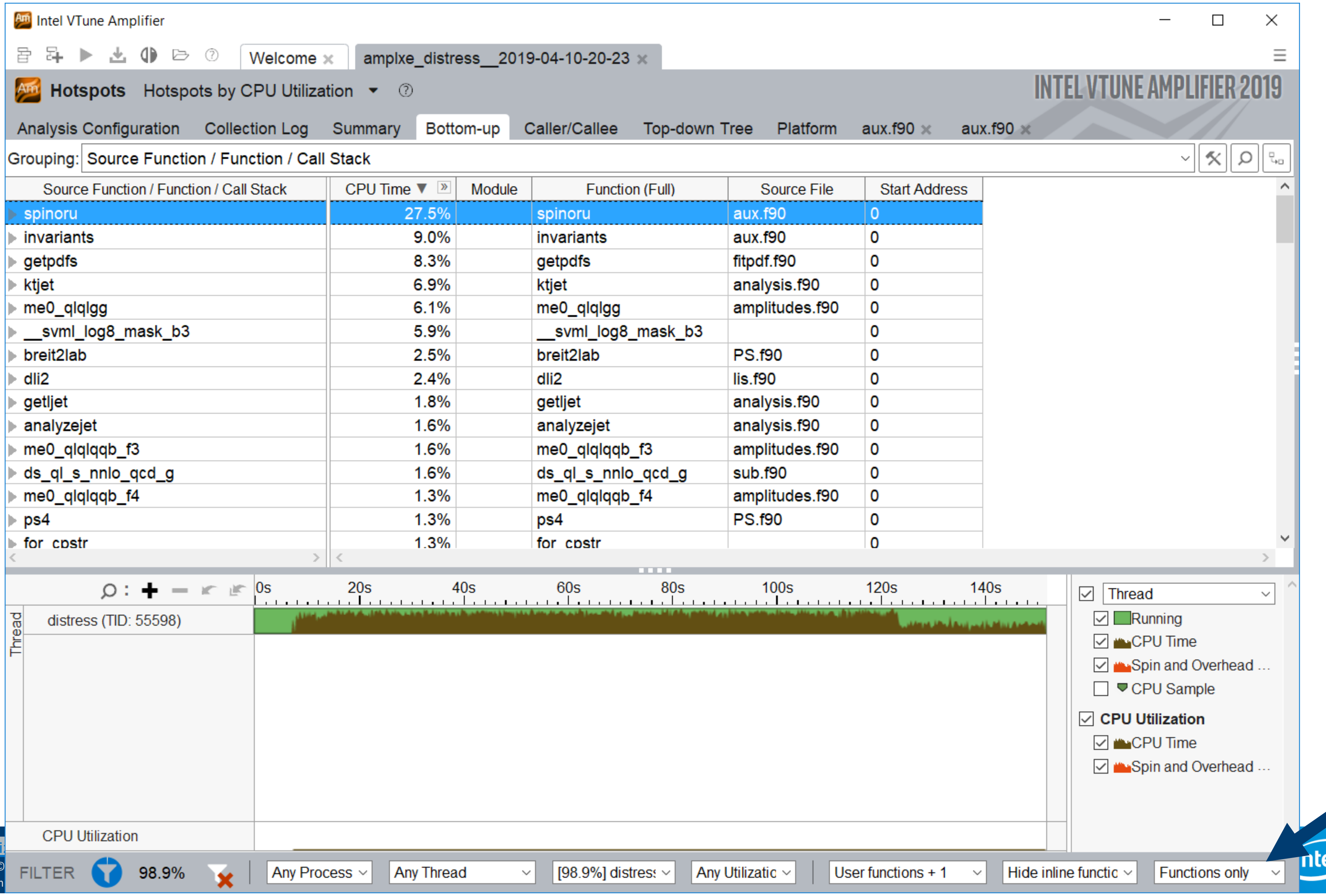
Next steps would include additional analysis to continue the optimization process.

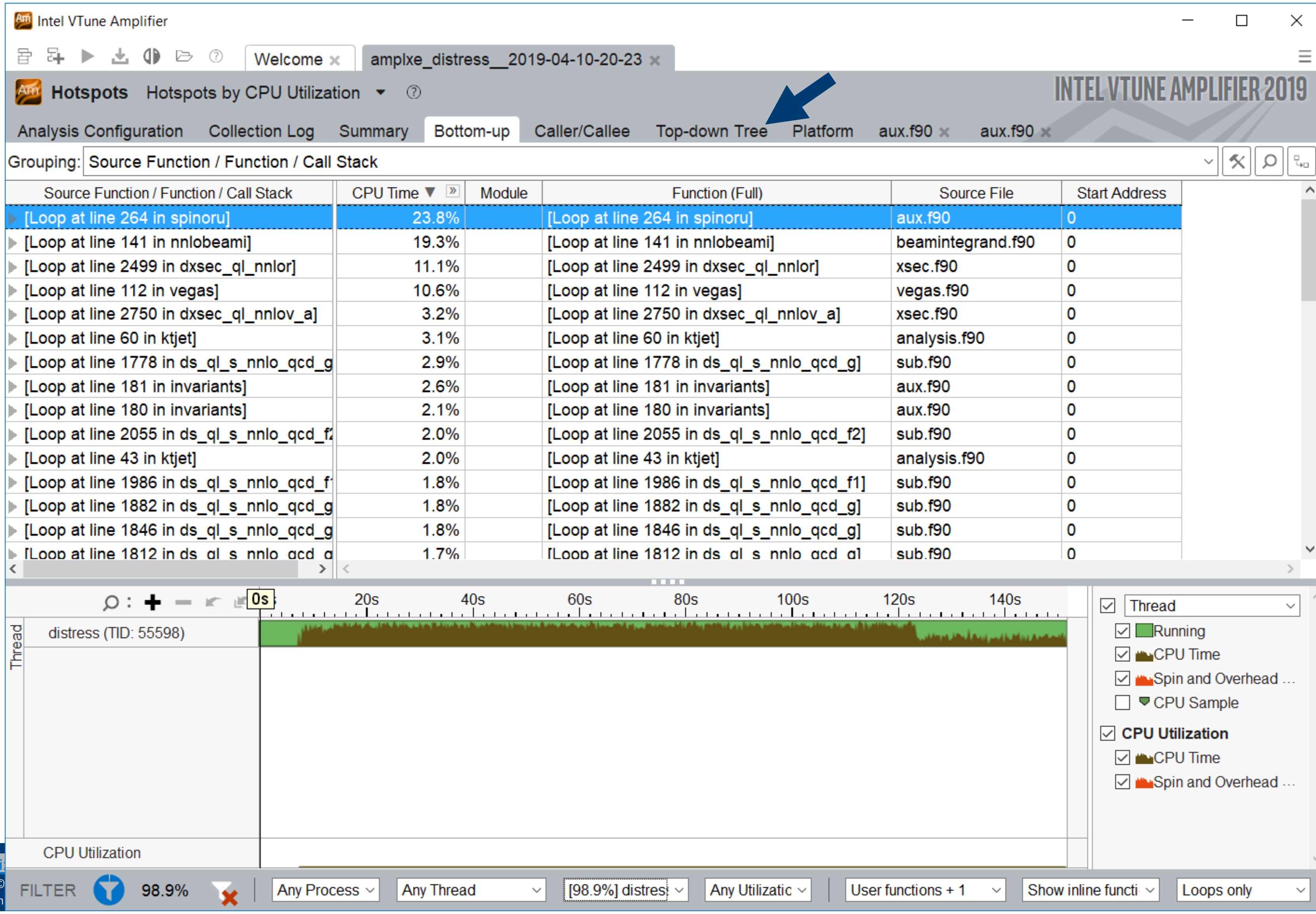


Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
vdpowr_	18.664s	libmkl_intel_lp64.so	vdpowr_		0x695310
aa	10.495s	distress	aa	aux.f90	0x41ec1c
aa	9.674s	distress	aa	aux.f90	0x41ec9a
invariants	9.055s	distress	invariants	aux.f90	0x41d550
__libm_csqrt_ex	7.792s	libimf.so	__libm_csqrt_ex		0xc7a50
spinoru	7.779s	distress	spinoru	aux.f90	0x41e9e0
ktjet	7.137s	distress	ktjet	analysis.f90	0x420ae0
__svml_log8_mask_b3	6.056s	distress	__svml_log8_mask_b3		0x532f50
breit2lab	2.096s	distress	breit2lab	PS.f90	0x4602d0
getljet	1.857s	distress	getljet	analysis.f90	0x421830
me0_qlqlgg	1.814s	distress	me0_qlqlgg	amplitudes.f90	0x4408d0
__libm_acos_l9	1.688s	libimf.so	__libm_acos_l9		0xedd80
analyzejet	1.658s	distress	analyzejet	analysis.f90	0x422050
ds_ql_s_nnlo_qcd_g	1.605s	distress	ds_ql_s_nnlo_qcd_g	sub.f90	0x4694e0
csqrt	1.384s	libimf.so	csqrt		0x1d430

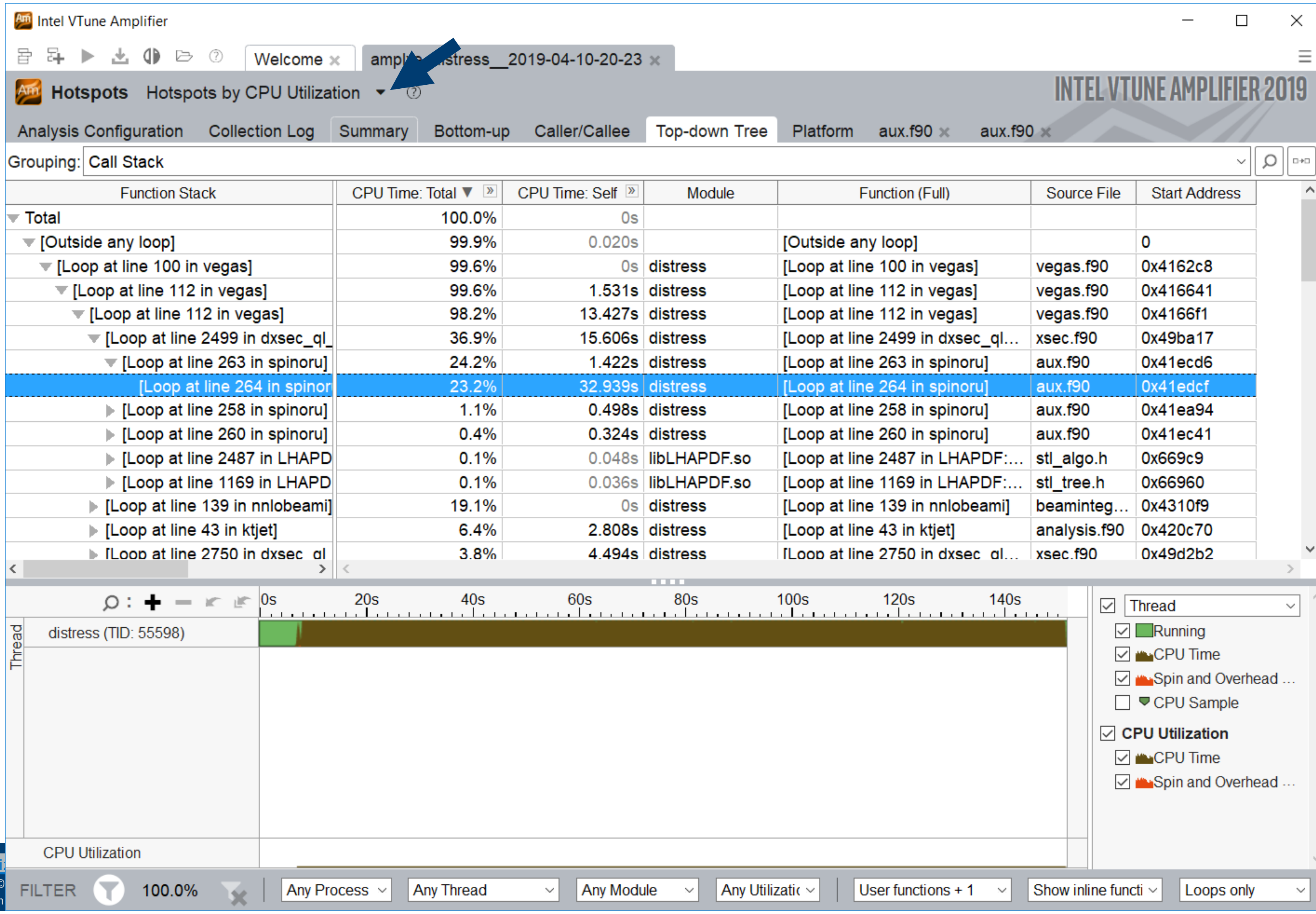

Intel VTune Amplifier
Welcome
amplxe_distress__2019-04-10-20-23
Hotspots
Hotspots by CPU Utilization
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Grouping: Function / Call Stack
Function / Call Stack
CPU Time
Module
Function (Full)
Source File
Start Address
vdpowr_ 13.1% libmkl_intel_lp64.so vdpowr_ 0x695310
aa 7.4% distress aa aux.f90 0x41ec1c
aa 6.8% distress aa aux.f90 0x41ec9a
invariants 6.4% distress invariants aux.f90 0x41d550
__libm_csqrt_ex 5.5% libimf.so __libm_csqrt_ex 0xc7a50
spinoru 5.5% distress spinoru aux.f90 0x41e9e0
ktjet 5.0% distress ktjet analysis.f90 0x420ae0
__svml_log8_mask_b3 4.3% distress __svml_log8_mask_b3 0x532f50
breit2lab 1.5% distress breit2lab PS.f90 0x4602d0
getljet 1.3% distress getljet analysis.f90 0x421830
me0_qlqlgg 1.3% distress me0_qlqlgg amplitudes.f90 0x4408d0
__libm_acos_l9 1.2% libimf.so __libm_acos_l9 0xedd80
analyzejet 1.2% distress analyzejet analysis.f90 0x422050
ds_ql_s_nnlo_qcd_g 1.1% distress ds_ql_s_nnlo_qcd_g sub.f90 0x4694e0
csqrt 1.0% libimf.so csqrt 0x1d430
0s
20s
40s
60s
80s
100s
120s
140s
Thread
distress (TID: 55598)
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 Utilizati
User functions + 1
Show inline funct
Functions only

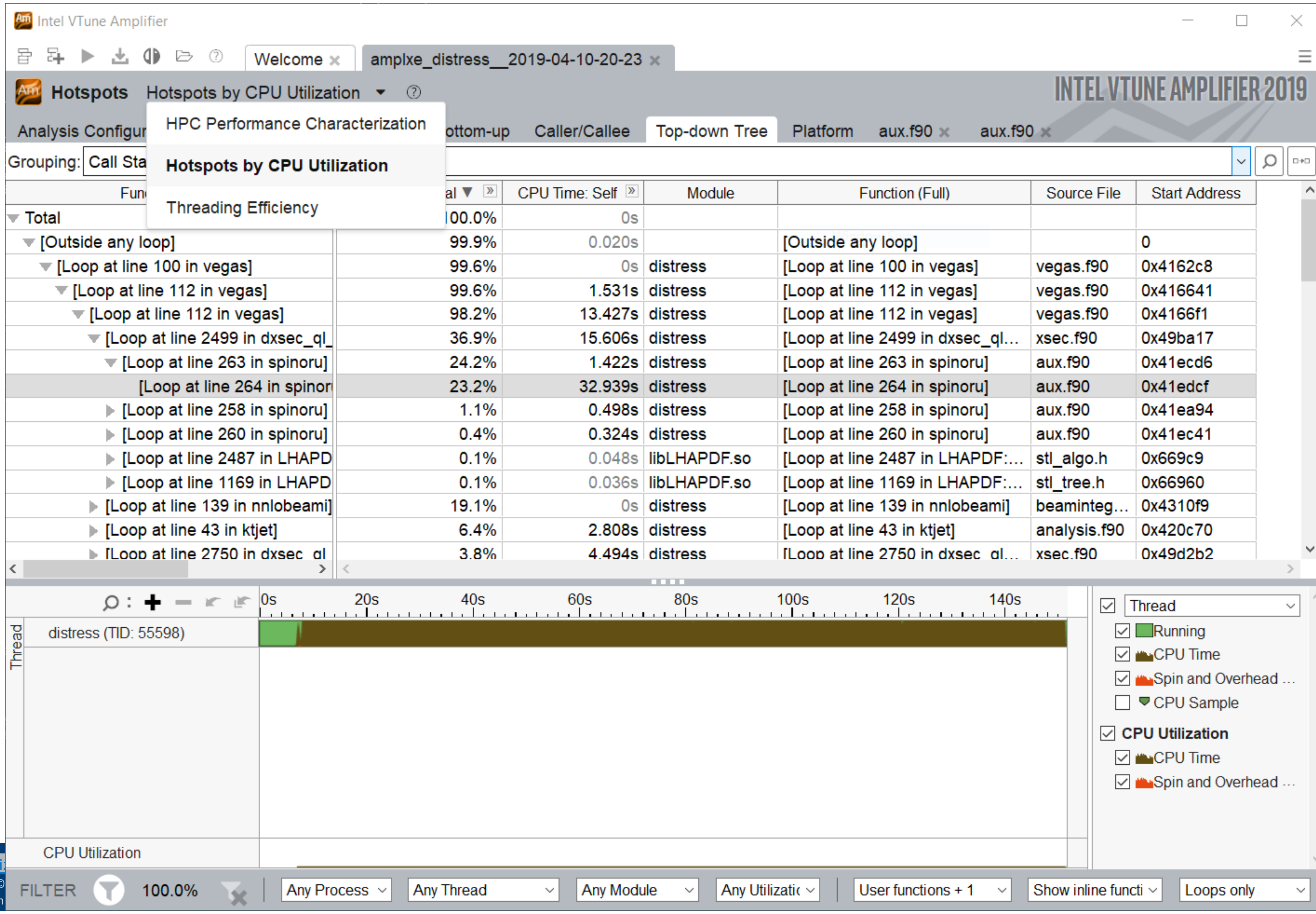

Intel VTune Amplifier
Welcome
amplxe_distress__2019-04-10-20-23
Hotspots
Hotspots by CPU Utilization
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
Grouping: Source Function / Function / Call Stack
Source Function / Function / Call Stack | CPU Time | Module | Function (Full) | Source File | Start Address
aa | 14.2% | aa | aux.f90 | 0
vdpowr_ | 13.1% | vdpowr_ | 0
invariants | 6.4% | invariants | aux.f90 | 0
__libm_csqrt_ex | 5.5% | __libm_csqrt_ex | 0
spinoru | 5.5% | spinoru | aux.f90 | 0
ktjet | 5.0% | ktjet | analysis.f90 | 0
__svml_log8_mask_b3 | 4.3% | __svml_log8_mask_b3 | 0
subqcd | 3.2% | subqcd | amplitudes.f90 | 0
breit2lab | 1.6% | breit2lab | PS.f90 | 0
hamp_qlqlqqb_1 | 1.4% | hamp_qlqlqqb_1 | amplitudes.f90 | 0
getljet | 1.3% | getljet | analysis.f90 | 0
me0_qlqlgg | 1.3% | me0_qlqlgg | amplitudes.f90 | 0
__libm_acos_l9 | 1.2% | __libm_acos_l9 | 0
analyzejet | 1.2% | analyzejet | analysis.f90 | 0
0s 20s 40s 60s 80s 100s 120s 140s
Thread
distress (TID: 55598)
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 Utilizati
User functions + 1
Show inline funct
Functions only

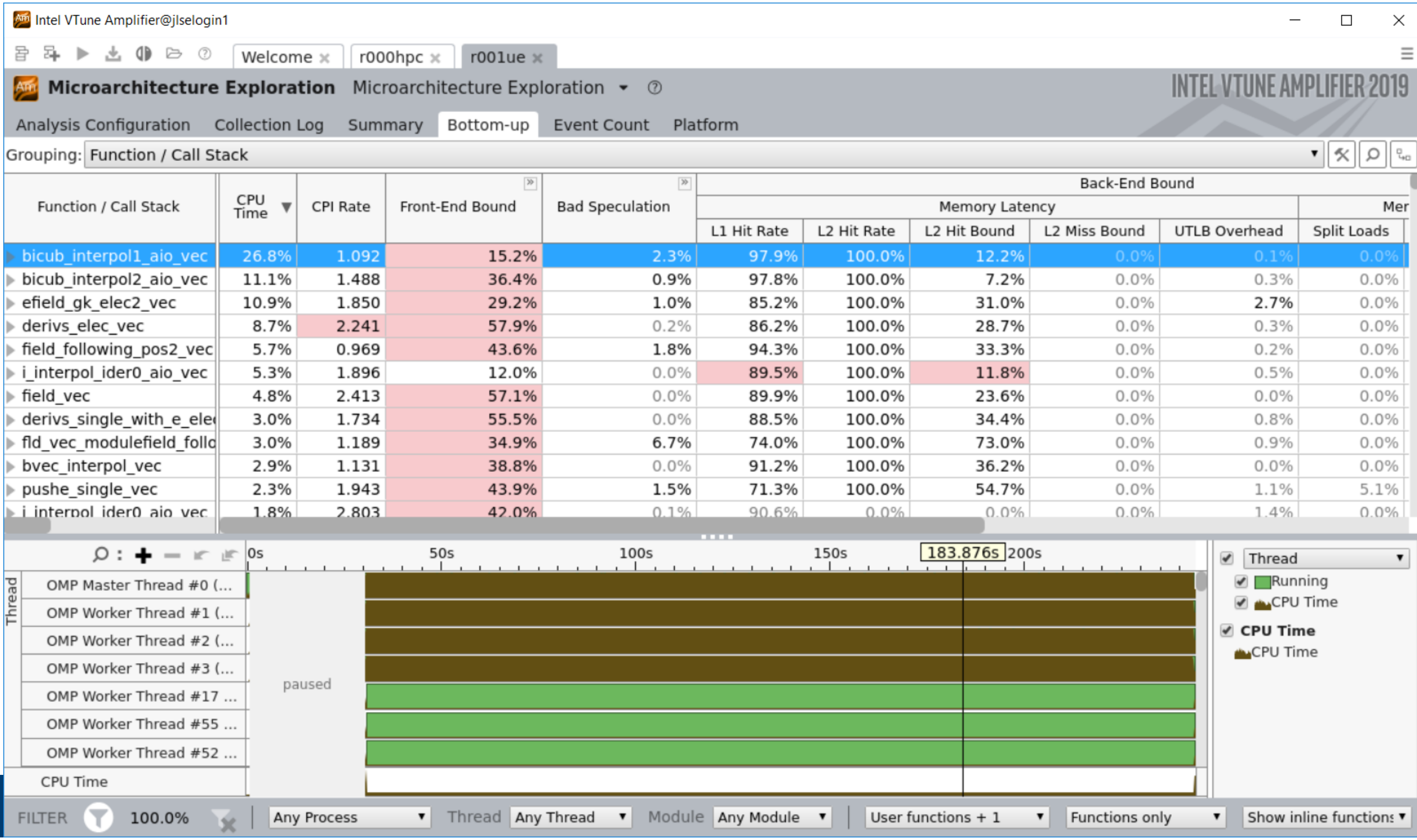

Intel VTune Amplifier
Welcome
amplxe_distress__2019-04-10-20-23
Hotspots
Hotspots by CPU Utilization
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
aux.f90
aux.f90
Grouping:
Source Function / Function / Call Stack
Source Function / Function / Call Stack | CPU Time | Module | Function (Full) | Source File | Start Address
spinoru | 27.5% | | spinoru | aux.f90 | 0
invariants | 9.0% | | invariants | aux.f90 | 0
getpdfs | 8.3% | | getpdfs | fitpdf.f90 | 0
ktjet | 6.9% | | ktjet | analysis.f90 | 0
me0_qlqlgg | 6.1% | | me0_qlqlgg | amplitudes.f90 | 0
__svml_log8_mask_b3 | 5.9% | | __svml_log8_mask_b3 | | 0
breit2lab | 2.5% | | breit2lab | PS.f90 | 0
dli2 | 2.4% | | dli2 | lis.f90 | 0
getljet | 1.8% | | getljet | analysis.f90 | 0
analyzejet | 1.6% | | analyzejet | analysis.f90 | 0
me0_qlqlqqb_f3 | 1.6% | | me0_qlqlqqb_f3 | amplitudes.f90 | 0
ds_ql_s_nnlo_qcd_g | 1.6% | | ds_ql_s_nnlo_qcd_g | sub.f90 | 0
me0_qlqlqqb_f4 | 1.3% | | me0_qlqlqqb_f4 | amplitudes.f90 | 0
ps4 | 1.3% | | ps4 | PS.f90 | 0
for_cpstr | 1.3% | | for_cpstr | | 0
0s
20s
40s
60s
80s
100s
120s
140s
Thread
distress (TID: 55598)
CPU Utilization
Thread
Running
CPU Time
Spin and Overhead ...
CPU Sample
CPU Utilization
CPU Time
Spin and Overhead ...
FILTER
98.9%
Any Process
Any Thread
[98.9%] distres
Any Utilizatic
User functions + 1
Hide inline functic
Functions only



Intel VTune Amplifier
Welcome
amplxe_distress__2019-04-10-20-23
Hotspots
Hotspots by CPU Utilization
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
aux.f90
aux.f90
Grouping: Source Function / Function / Call Stack
Source Function / Function / Call Stack | CPU Time | Module | Function (Full) | Source File | Start Address
[Loop at line 264 in spinoru] | 23.8% | [Loop at line 264 in spinoru] | aux.f90 | 0
[Loop at line 141 in nnlobeami] | 19.3% | [Loop at line 141 in nnlobeami] | beamintegrand.f90 | 0
[Loop at line 2499 in dxsec_ql_nnlor] | 11.1% | [Loop at line 2499 in dxsec_ql_nnlor] | xsec.f90 | 0
[Loop at line 112 in vegas] | 10.6% | [Loop at line 112 in vegas] | vegas.f90 | 0
[Loop at line 2750 in dxsec_ql_nnlov_a] | 3.2% | [Loop at line 2750 in dxsec_ql_nnlov_a] | xsec.f90 | 0
[Loop at line 60 in ktjet] | 3.1% | [Loop at line 60 in ktjet] | analysis.f90 | 0
[Loop at line 1778 in ds_ql_s_nnlo_qcd_g | 2.9% | [Loop at line 1778 in ds_ql_s_nnlo_qcd_g] | sub.f90 | 0
[Loop at line 181 in invariants] | 2.6% | [Loop at line 181 in invariants] | aux.f90 | 0
[Loop at line 180 in invariants] | 2.1% | [Loop at line 180 in invariants] | aux.f90 | 0
[Loop at line 2055 in ds_ql_s_nnlo_qcd_f2 | 2.0% | [Loop at line 2055 in ds_ql_s_nnlo_qcd_f2] | sub.f90 | 0
[Loop at line 43 in ktjet] | 2.0% | [Loop at line 43 in ktjet] | analysis.f90 | 0
[Loop at line 1986 in ds_ql_s_nnlo_qcd_f1 | 1.8% | [Loop at line 1986 in ds_ql_s_nnlo_qcd_f1] | sub.f90 | 0
[Loop at line 1882 in ds_ql_s_nnlo_qcd_g | 1.8% | [Loop at line 1882 in ds_ql_s_nnlo_qcd_g] | sub.f90 | 0
[Loop at line 1846 in ds_ql_s_nnlo_qcd_g | 1.8% | [Loop at line 1846 in ds_ql_s_nnlo_qcd_g] | sub.f90 | 0
0s 20s 40s 60s 80s 100s 120s 140s
Thread
distress (TID: 55598)
CPU Utilization
Thread
Running
CPU Time
Spin and Overhead ...
CPU Sample
CPU Utilization
CPU Time
Spin and Overhead ...
FILTER 98.9%
Any Process
Any Thread
[98.9%] distres
Any Utilizatic
User functions + 1
Show inline functi
Loops only


Intel VTune Amplifier
Welcome
amplxe-distress__2019-04-10-20-23
Hotspots Hotspots by CPU Utilization
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform aux.f90 aux.f90
Grouping: Call Stack
Function Stack | CPU Time: Total | CPU Time: Self | Module | Function (Full) | Source File | Start Address
Total 100.0% 0s
[Outside any loop] 99.9% 0.020s [Outside any loop] 0
[Loop at line 100 in vegas] 99.6% 0s distress [Loop at line 100 in vegas] vegas.f90 0x4162c8
[Loop at line 112 in vegas] 99.6% 1.531s distress [Loop at line 112 in vegas] vegas.f90 0x416641
[Loop at line 112 in vegas] 98.2% 13.427s distress [Loop at line 112 in vegas] vegas.f90 0x4166f1
[Loop at line 2499 in dxsec_ql_ 36.9% 15.606s distress [Loop at line 2499 in dxsec_ql… xsec.f90 0x49ba17
[Loop at line 263 in spinoru] 24.2% 1.422s distress [Loop at line 263 in spinoru] aux.f90 0x41ecd6
[Loop at line 264 in spinor 23.2% 32.939s distress [Loop at line 264 in spinoru] aux.f90 0x41edcf
[Loop at line 258 in spinoru] 1.1% 0.498s distress [Loop at line 258 in spinoru] aux.f90 0x41ea94
[Loop at line 260 in spinoru] 0.4% 0.324s distress [Loop at line 260 in spinoru] aux.f90 0x41ec41
[Loop at line 2487 in LHAPD 0.1% 0.048s libLHAPDF.so [Loop at line 2487 in LHAPDF:… stl_algo.h 0x669c9
[Loop at line 1169 in LHAPD 0.1% 0.036s libLHAPDF.so [Loop at line 1169 in LHAPDF:… stl_tree.h 0x66960
[Loop at line 139 in nnlobeami] 19.1% 0s distress [Loop at line 139 in nnlobeami] beaminteg… 0x4310f9
[Loop at line 43 in ktjet] 6.4% 2.808s distress [Loop at line 43 in ktjet] analysis.f90 0x420c70
[Loop at line 2750 in dxsec_ql 3.8% 4.494s distress [Loop at line 2750 in dxsec_ql… xsec.f90 0x49d2b2
0s 20s 40s 60s 80s 100s 120s 140s
Thread
distress (TID: 55598)
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 Utilizatio User functions + 1 Show inline functi Loops only


Intel VTune Amplifier
Welcome
amplxe_distress__2019-04-10-20-23
Hotspots
Hotspots by CPU Utilization
INTEL VTUNE AMPLIFIER 2019
HPC Performance Characterization
Hotspots by CPU Utilization
Threading Efficiency
Analysis Configur
ottom-up
Caller/Callee
Top-down Tree
Platform
aux.f90
aux.f90
Grouping: Call Sta
Fun
al
CPU Time: Self
Module
Function (Full)
Source File
Start Address
Total | 100.0% | 0s
[Outside any loop] | 99.9% | 0.020s | | [Outside any loop] | | 0
[Loop at line 100 in vegas] | 99.6% | 0s | distress | [Loop at line 100 in vegas] | vegas.f90 | 0x4162c8
[Loop at line 112 in vegas] | 99.6% | 1.531s | distress | [Loop at line 112 in vegas] | vegas.f90 | 0x416641
[Loop at line 112 in vegas] | 98.2% | 13.427s | distress | [Loop at line 112 in vegas] | vegas.f90 | 0x4166f1
[Loop at line 2499 in dxsec_ql_ | 36.9% | 15.606s | distress | [Loop at line 2499 in dxsec_ql… | xsec.f90 | 0x49ba17
[Loop at line 263 in spinoru] | 24.2% | 1.422s | distress | [Loop at line 263 in spinoru] | aux.f90 | 0x41ecd6
[Loop at line 264 in spinor | 23.2% | 32.939s | distress | [Loop at line 264 in spinoru] | aux.f90 | 0x41edcf
[Loop at line 258 in spinoru] | 1.1% | 0.498s | distress | [Loop at line 258 in spinoru] | aux.f90 | 0x41ea94
[Loop at line 260 in spinoru] | 0.4% | 0.324s | distress | [Loop at line 260 in spinoru] | aux.f90 | 0x41ec41
[Loop at line 2487 in LHAPD | 0.1% | 0.048s | libLHAPDF.so | [Loop at line 2487 in LHAPDF:… | stl_algo.h | 0x669c9
[Loop at line 1169 in LHAPD | 0.1% | 0.036s | libLHAPDF.so | [Loop at line 1169 in LHAPDF:… | stl_tree.h | 0x66960
[Loop at line 139 in nnlobeami] | 19.1% | 0s | distress | [Loop at line 139 in nnlobeami] | beaminteg… | 0x4310f9
[Loop at line 43 in ktjet] | 6.4% | 2.808s | distress | [Loop at line 43 in ktjet] | analysis.f90 | 0x420c70
[Loop at line 2750 in dxsec_ql | 3.8% | 4.494s | distress | [Loop at line 2750 in dxsec_ql… | xsec.f90 | 0x49d2b2
0s 20s 40s 60s 80s 100s 120s 140s
Thread
distress (TID: 55598)
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 Utilizatio
User functions + 1
Show inline functi
Loops only



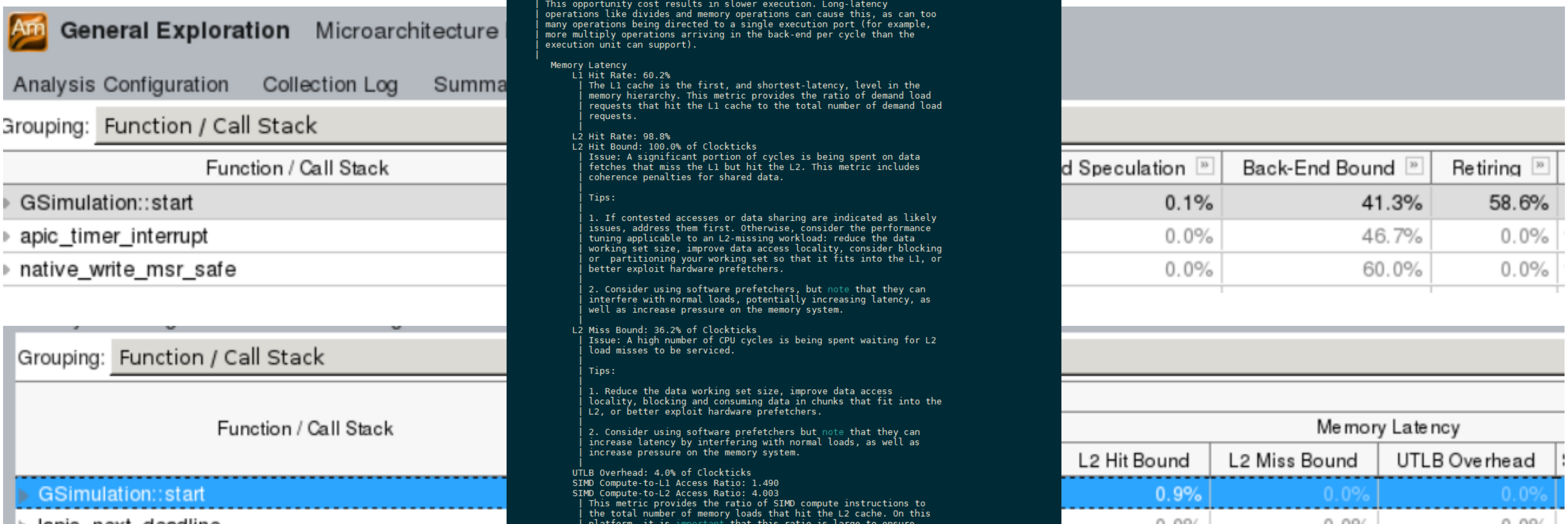
Function / Call Stack	CPU Time	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound: Memory Latency: L1 Hit Rate	L2 Hit Rate	L2 Hit Bound	L2 Miss Bound	UTLB Overhead	Mer: Split Loads
bicub_interpol1_aio_vec	26.8%	1.092	15.2%	2.3%	97.9%	100.0%	12.2%	0.0%	0.1%	0.0%
bicub_interpol2_aio_vec	11.1%	1.488	36.4%	0.9%	97.8%	100.0%	7.2%	0.0%	0.3%	0.0%
efield_gk_elec2_vec	10.9%	1.850	29.2%	1.0%	85.2%	100.0%	31.0%	0.0%	2.7%	0.0%
derivs_elec_vec	8.7%	2.241	57.9%	0.2%	86.2%	100.0%	28.7%	0.0%	0.3%	0.0%
field_following_pos2_vec	5.7%	0.969	43.6%	1.8%	94.3%	100.0%	33.3%	0.0%	0.2%	0.0%
i_interpol_ider0_aio_vec	5.3%	1.896	12.0%	0.0%	89.5%	100.0%	11.8%	0.0%	0.5%	0.0%
field_vec	4.8%	2.413	57.1%	0.0%	89.9%	100.0%	23.6%	0.0%	0.0%	0.0%
derivs_single_with_e_ele	3.0%	1.734	55.5%	0.0%	88.5%	100.0%	34.4%	0.0%	0.8%	0.0%
fld_vec_modulefield_follo	3.0%	1.189	34.9%	6.7%	74.0%	100.0%	73.0%	0.0%	0.9%	0.0%
bvec_interpol_vec	2.9%	1.131	38.8%	0.0%	91.2%	100.0%	36.2%	0.0%	0.0%	0.0%
pushe_single_vec	2.3%	1.943	43.9%	1.5%	71.3%	100.0%	54.7%	0.0%	1.1%	5.1%
i_interpol_ider0_aio_vec	1.8%	2.803	42.0%	0.1%	90.6%	0.0%	0.0%	0.0%	1.4%	0.0%

Viewing the result

- Text file reports:
  - amplxe-cl -help report How do I create a text report?
  - amplxe-cl -help report hotspots What can I change
  - amplxe-cl -R hotspots -r ./res_dir -column=? Which columns are available?
  - Ex: Report top 5% of loops, Total time and L2 Cache hit rates
    - amplxe-cl -R hotspots -loops-only

    -limit=5 -column="L2_CACHE_HIT, Time Self (%)"
- Vtune GUI
  - unset LD_PRELOAD; amplxe-gui

Using Vtune to ch




```
amplxe: Using result path `/gpfs/jlse-fs0/users/pvelesko/nbody-demo/ver5/amplxe_knl_nodiv_60k'
amplxe: Executing actions 75 % Generating a report                          Elapsed Time: 280.549s
    Clockticks: 405,093,000,000
    Instructions Retired: 342,199,000,000
    CPI Rate: 1.184
    MUX Reliability: 0.992
    Front-End Bound: 1.5% of Pipeline Slots
        ITLB Overhead: 0.0% of Clockticks
        BACLEARS: 0.1% of Clockticks
        MS Entry: 0.0% of Clockticks
        ICache Line Fetch: 1.0% of Clockticks
    Bad Speculation: 0.2% of Pipeline Slots
        Branch Mispredict: 0.2% of Clockticks
        SMC Machine Clear: 0.0% of Clockticks
        MO Machine Clear Overhead: 0.0% of Clockticks
    Back-End Bound: 56.2% of Pipeline Slots
     | A significant proportion of pipeline slots are remaining empty. When
     | operations take too long in the back-end, they introduce bubbles in the
     | pipeline that ultimately cause fewer pipeline slots containing useful
     | work to be retired per cycle than the machine is capable of supporting.
     | This opportunity cost results in slower execution. Long-latency
     | operations like divides and memory operations can cause this, as can too
     | many operations being directed to a single execution port (for example,
     | more multiply operations arriving in the back-end per cycle than the
     | execution unit can support).
     |
        Memory Latency
            L1 Hit Rate: 60.2%
             | The L1 cache is the first, and shortest-latency, level in the
             | memory hierarchy. This metric provides the ratio of demand load
             | requests that hit the L1 cache to the total number of demand load
             | requests.
             |
            L2 Hit Rate: 98.8%
            L2 Hit Bound: 100.0% of Clockticks
             | Issue: A significant portion of cycles is being spent on data
             | fetches that miss the L1 but hit the L2. This metric includes
             | coherence penalties for shared data.
             |
             | Tips:
             |
             | 1. If contested accesses or data sharing are indicated as likely
             | issues, address them first. Otherwise, consider the performance
             | tuning applicable to an L2-missing workload: reduce the data
             | working set size, improve data access locality, consider blocking
             | or  partitioning your working set so that it fits into the L1, or
             | better exploit hardware prefetchers.
             |
             | 2. Consider using software prefetchers, but note that they can
             | interfere with normal loads, potentially increasing latency, as
             | well as increase pressure on the memory system.
             |
            L2 Miss Bound: 36.2% of Clockticks
             | Issue: A high number of CPU cycles is being spent waiting for L2
             | load misses to be serviced.
             |
             | Tips:
             |
             | 1. Reduce the data working set size, improve data access
             | locality, blocking and consuming data in chunks that fit into the
             | L2, or better exploit hardware prefetchers.
             |
             | 2. Consider using software prefetchers but note that they can
             | increase latency by interfering with normal loads, as well as
             | increase pressure on the memory system.
             |
            UTLB Overhead: 4.0% of Clockticks
            SIMD Compute-to-L1 Access Ratio: 1.490
            SIMD Compute-to-L2 Access Ratio: 4.003
             | This metric provides the ratio of SIMD compute instructions to
             | the total number of memory loads that hit the L2 cache. On this
             | platform, it is important that this ratio is large to ensure
             | efficient usage of compute resources.
             |
            Contested Accesses (Intra-Tile): 0.0%
            Page Walk: 4.9% of Clockticks
        Memory Reissues
            Split Loads: 0.0%
            Split Stores: 0.0%
            Loads Blocked by Store Forwarding: 0.0%
    Retiring: 42.1% of Pipeline Slots
        VPU Utilization: 99.9% of Clockticks
        Divider: 0.0% of Clockticks
        MS Assists: 0.1% of Clockticks
        FP Assists: 0.0% of Clockticks
    Total Thread Count: 1
```

Microarchitecture Exploration - Caches

S	2k	2.5k	30k	35k	50k	60k
L1 Hit %	100%	63.9%	62.4%	48.5%	57.5%	60.2%
L2 Hit %	0%	100%	100%	100%	99.2%	98.8%
L2 Hit Bound %	0%	100%	100%	100%	100%	100%
L2 Miss Bound %	0%	0%	0%	0%	28.6%	36.2%

Profiling PYThon & ML applications

Python

Profiling Python is straightforward in VTune™ Amplifier, as long as one does the following:

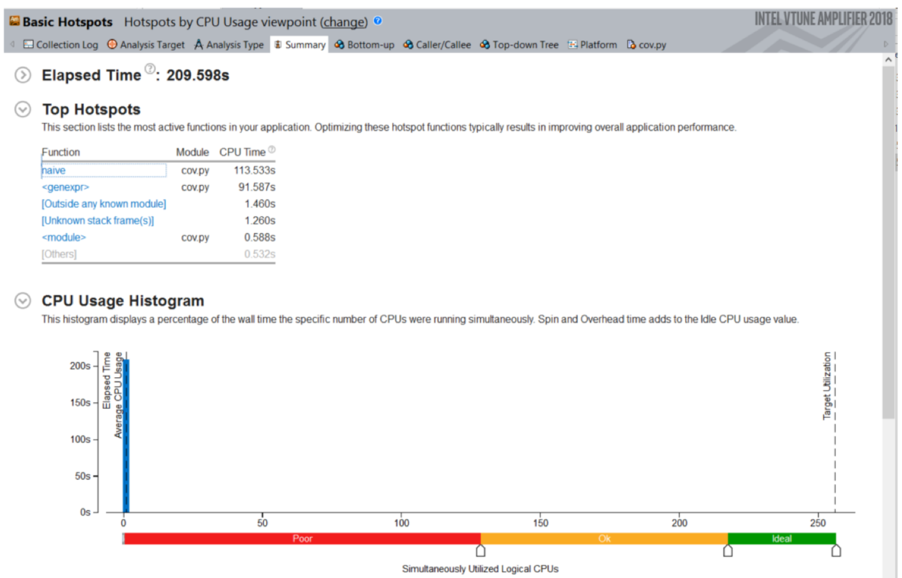
- The "application" should be the full path to the python interpreter used
- The python code should be passed as "arguments" to the "application"

In Theta this would look like this:

```
aprun -n 1 -N 1 amplxe-cl -c hotspots -r res_dir \
                -- /usr/bin/python3 mycode.py myarguments
```

Simple Python Example on Theta

```
aprun -n 1 -N 1 amplxe-cl -c hotspots -r vt_pytest \
                          -- /usr/bin/python ./cov.py naive 100 1000
```



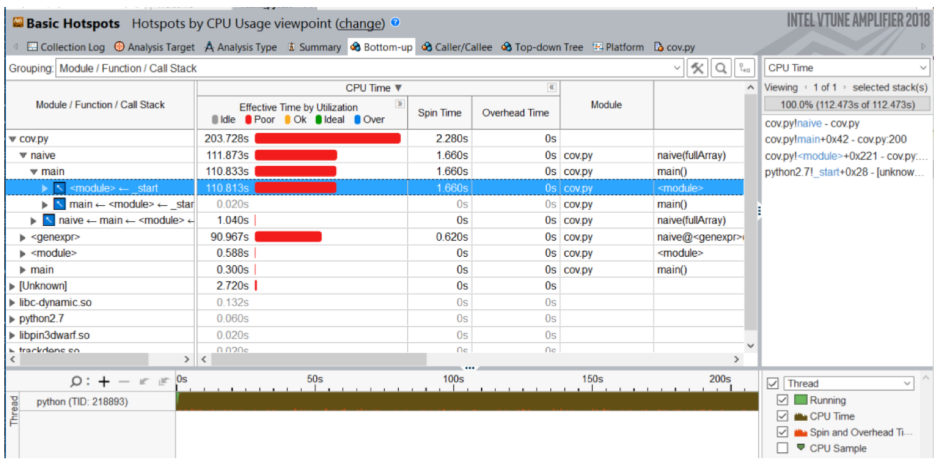
Naïve implementation of the calculation of a covariance matrix

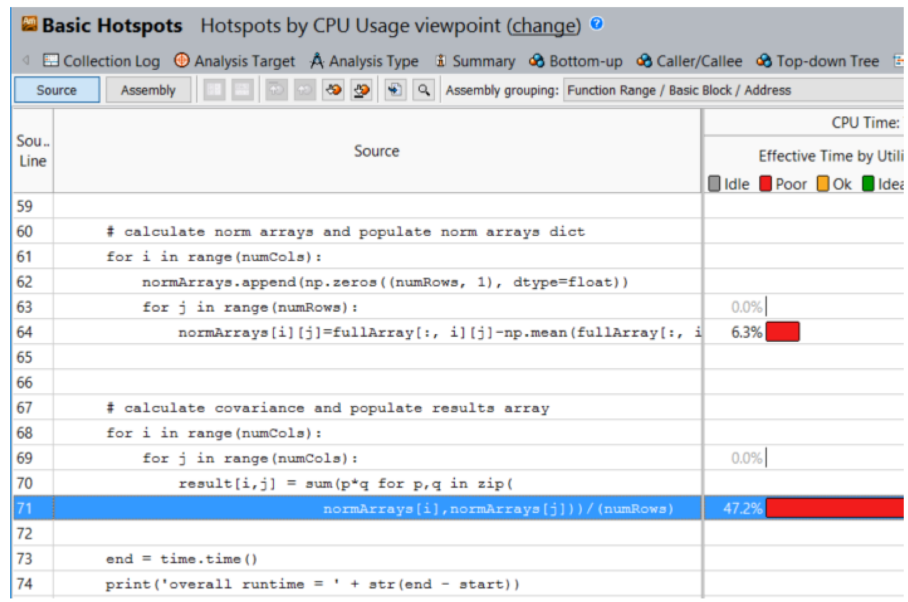
Summary shows:

- Single thread execution
- Top function is “naive”

Click on top function to go to Bottom-up view

Bottom-up View and Source Code





Inefficient array multiplication found quickly

We could use numpy to improve on this

Note that for mixed Python/C code a Top-Down view can often be helpful to drill down into the C kernels

Intel® VtunE™ Application Performance Snapshot

Performance overview at you fingertips

VTune™ Amplifier's Application Performance Snapshot

High-level overview of application performance

- Identify primary optimization areas
- Recommend next steps in analysis
- Extremely easy to use
- Informative, actionable data in clean HTML report
- Detailed reports available via command line
- Low overhead, high scalability

Usage on Theta

Launch all profiling jobs from **/projects** rather than **/home**

No module available, so setup the environment manually:

```
$ module  load vtune
$ export PMI_NO_FORK=1
```

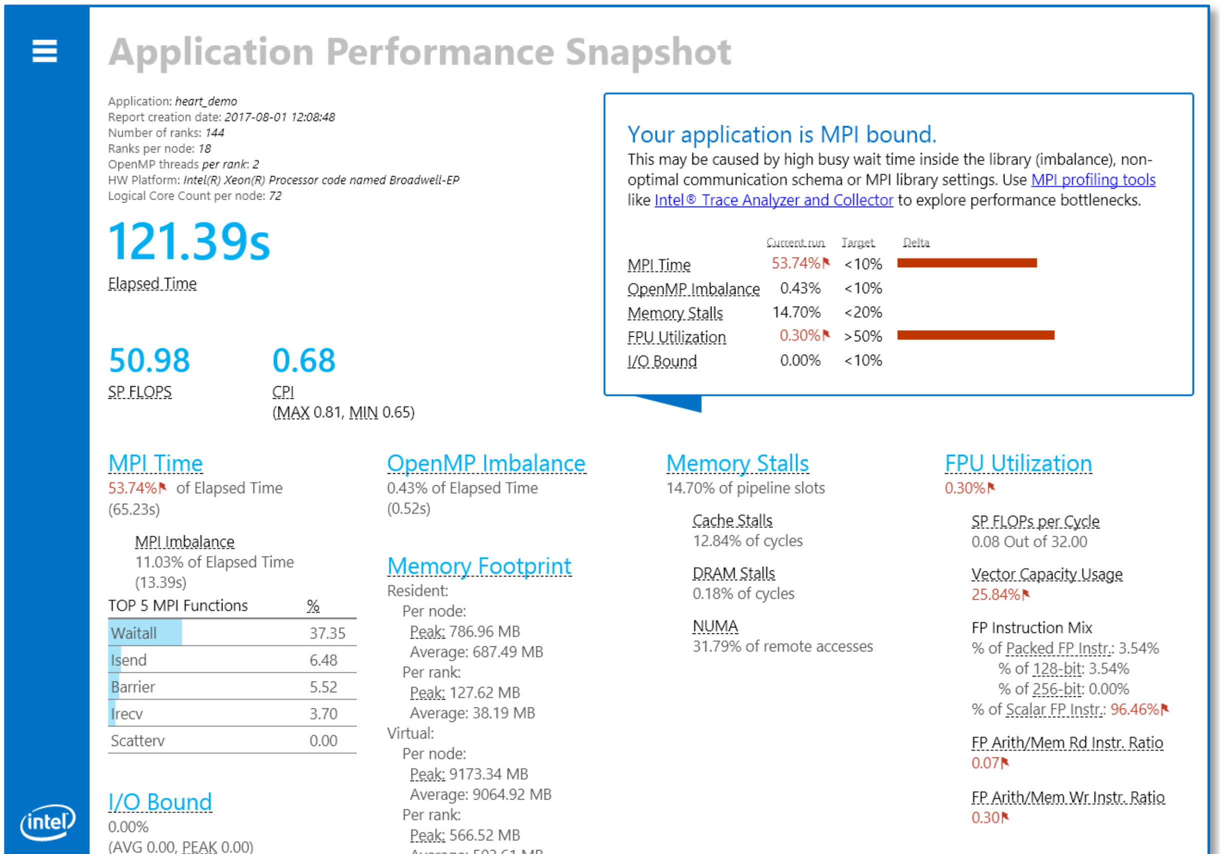
Launch your job in interactive or batch mode:

```
$ aprun -N <ppn> -n <totRanks> [affinity opts] aps ./exe
```

Produce text and html reports:

```
$ aprun -report ./aps_result_ ….
```

APS HTML Report



Application Performance Snapshot

Application: *heart_demo*
Report creation date: *2017-08-01 12:08:48*
Number of ranks: *144*
Ranks per node: *18*
OpenMP threads *per rank: 2*
HW Platform: *Intel(R) Xeon(R) Processor code named Broadwell-EP*
Logical Core Count per node: *72*

121.39s
Elapsed Time

50.98
SP FLOPS

0.68
CPI
(MAX 0.81, MIN 0.65)

Your application is MPI bound.
This may be caused by high busy wait time inside the library (imbalance), non-optimal communication schema or MPI library settings. Use MPI profiling tools like Intel® Trace Analyzer and Collector to explore performance bottlenecks.

	Current run	Target	Delta
MPI Time	53.74%	<10%	
OpenMP Imbalance	0.43%	<10%	
Memory Stalls	14.70%	<20%	
FPU Utilization	0.30%	>50%	
I/O Bound	0.00%	<10%	

MPI Time
53.74% of Elapsed Time
(65.23s)

MPI Imbalance
11.03% of Elapsed Time
(13.39s)

TOP 5 MPI Functions	%
Waitall	37.35
Isend	6.48
Barrier	5.52
Irecv	3.70
Scatterv	0.00

I/O Bound
0.00%
(AVG 0.00, PEAK 0.00)

OpenMP Imbalance
0.43% of Elapsed Time
(0.52s)

Memory Footprint
Resident:
Per node:
Peak: 786.96 MB
Average: 687.49 MB
Per rank:
Peak: 127.62 MB
Average: 38.19 MB
Virtual:
Per node:
Peak: 9173.34 MB
Average: 9064.92 MB
Per rank:
Peak: 566.52 MB
Average: 503.61 MB

Memory Stalls
14.70% of pipeline slots

Cache Stalls
12.84% of cycles

DRAM Stalls
0.18% of cycles

NUMA
31.79% of remote accesses

FPU Utilization
0.30%

SP FLOPs per Cycle
0.08 Out of 32.00

Vector Capacity Usage
25.84%

FP Instruction Mix
% of Packed FP Instr.: 3.54%
% of 128-bit: 3.54%
% of 256-bit: 0.00%
% of Scalar FP Instr.: 96.46%

FP Arith/Mem Rd Instr. Ratio
0.07

FP Arith/Mem Wr Instr. Ratio
0.30

Common issues

Fixes

No call stack information - check that finalization

Incompatible database scheme - make sure the same version of vtune

Vtune sampling driver.. using perf - use latest vtune/ driver needs a rebuild

Tips and tricks

Speeding up finalization

Advisor

add `--no-auto-finalize` to the aprun

followed by `advixe-cl R survey …` *without aprun* will cause to finalize on the momnode rather than KNL.

You can also finalize on thetalogin:

cd your_src_dir;

export SRCDIR=`pwd | xargs realpath`

advixe-cl -R survey --search-dir src:=${SRCDIR} ..

Vtune

add `--finalization-mode=none` to aprun

followed by `amplxe-cl -R hotspots …` *without aprun* will cause to finalize on momnode rather than KNL

You can also finalize on thetalogin:

cd your_src_dir;

export SRCDIR=`pwd | xargs realpath`

amplxe-cl -R hotspots --search-dir src:=${SRCDIR} ..

Managing overheads

Advisor Dependencies and MAP analyses can have huge overheads

If able, run on reduced problem size. Advisor just needs to figure out the execution flow.

Only analyze loops/functions of interest:

https://software.intel.com/en-us/advisor-user-guide-mark-up-loops

backup

When do I use Vtune vs Advisor?

Vtune

- What's my cache hit ratio?
- Which loop/function is consuming most time overall? (bottom-up)
- Am I stalling often? IPC?
- Am I keeping all the threads busy?
- Am I hitting remote NUMA?
- When do I maximize my BW?

Advisor

- Which vector ISA am I using?
- Flow of execution (callstacks)
- What is my vectorization efficiency?
- Can I safely force vectorization?
- Inlining? Data type conversions?
- Roofline

VTune Cheat Sheet

Compile with -g -dynamic

amplxe-cl -c hpc-performance -flags -- ./executable

- --result-dir=./vtune_output_dir
- --search-dir src:=../src --search-dir bin:=./
- -knob enable-stack-collection=true -knob collect-memory-bandwidth=false
- -knob analyze-openmp=true
- -finalization-mode=deferred if finalization is taking too long on KNL
- -data-limit=125 ← in mb
- -trace-mpi for MPI metrics on Theta
- amplxe-cl -help collect survey

https://software.intel.com/en-us/vtune-amplifier-help-amplxe-cl-command-syntax

Advisor Cheat Sheet

Compile with –g -dynamic

advixe-cl –c roofline/depencies/map –flags -- ./executable

- --project-dir=./advixe_output_dir
- --search-dir src:=../src --search-dir bin:=./
- -no-auto-finalize if finalization is taking too long on KNL
- --interval 1 (sample at 1ms interval, helps for profiling short runs)
- -data-limit=125 ← in mb
- advixe-cl -help

https://software.intel.com/en-us/advisor-help-lin-command-line-interface-reference

How much further can we go?

Introducing the Cache-Aware Roofline Model

Platform peak FLOPs

How many floating point operations per second

$$\text{Gflop/s} = min \begin{cases} \boldsymbol{Platform\ PEAK} \\ \boldsymbol{Platform\ BW * AI} \end{cases}$$

Theoretical value can be computed by specification

Example with 2 sockets Intel® Xeon® Processor E5-2697 v2

PEAK FLOP = 2 x 2.7 x 12 x 8 x 2 = **1036.8 Gflop/s**

- 2: Number of sockets
- 2.7: Core Frequency
- 12: Number of cores
- 8: Number of single precision element in a SIMD register
- 2: 1 port for addition, 1 for multiplication

More realistic value can be obtained by running **Linpack**

=~ 930 Gflop/s on a 2 sockets Intel® Xeon® Processor E5-2697 v2

Platform PEAK bandwidth

How many bytes can be transferred per second

$$\text{Gflop/s} = \boldsymbol{min}\begin{cases}\boldsymbol{Platform\ PEAK} \\ \boldsymbol{Platform\ BW} * \boldsymbol{AI}\end{cases}$$

Theoretical value can be computed by specification

Example with 2 sockets Intel® Xeon® Processor E5-2697 v2

PEAK BW = 2 x 1.866 x 8 x 4 = 119 GB/s

Number of sockets
Memory Frequency
Byte per channel
Number of mem channels

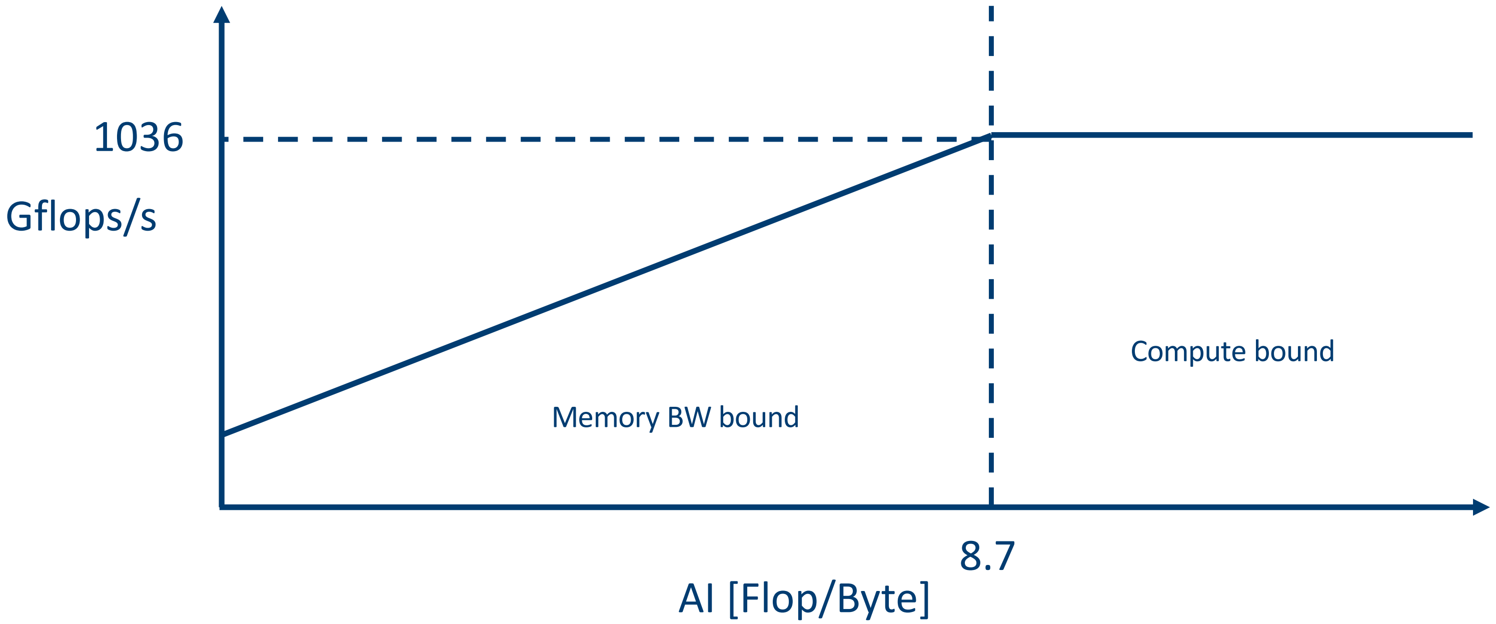
More realistic value can be obtained by running **Stream**

=~ 100 GB/s on a 2 sockets Intel® Xeon® Processor E5-2697 v2

Drawing the Roofline

Gflop/s$= min \begin{cases} \boldsymbol{Platform\ PEAK} \\ \boldsymbol{Platform\ BW} \ * \boldsymbol{AI} \end{cases}$

2 sockets Intel® Xeon® Processor E5-2697 v2
Peak Flop = 1036 Gflop/s
Peak BW = 119 GB/s



Cache-Aware Roofline

Next Steps

If Under the Vector Add Peak

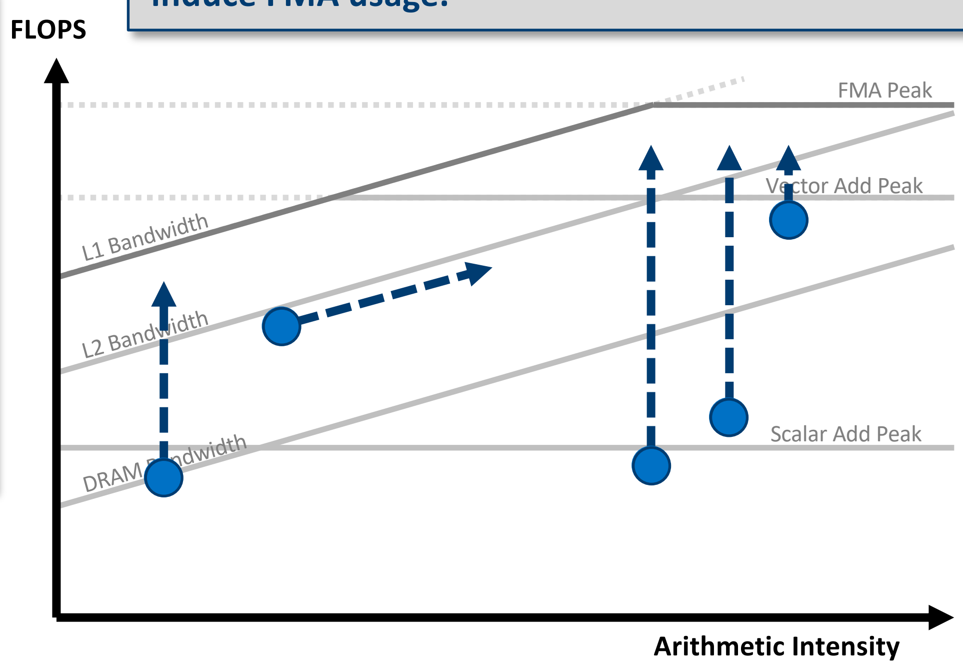
Check "Traits" in the Survey to see if FMAs are used. If not, try altering your code or compiler flags to **induce FMA usage.**

If just above the Scalar Add Peak

Check **vectorization efficiency** in the Survey. Follow the recommendations to improve it if it's low.

If under or near a memory roof…

- Try a MAP analysis. Make any appropriate **cache optimizations.**
- If cache optimization is impossible, try **reworking the algorithm to have a higher AI.**



If under the Scalar Add Peak…

Check the Survey Report to see if the loop vectorized. If not, try to **get it to vectorize** if possible. This may involve running Dependencies to see if it's safe to force it.

intel® Software