



# PROFILING YOUR APPLICATION WITH INTEL<sup>®</sup> VTUNE<sup>™</sup> AMPLIFIER AND INTEL<sup>®</sup> ADVISOR

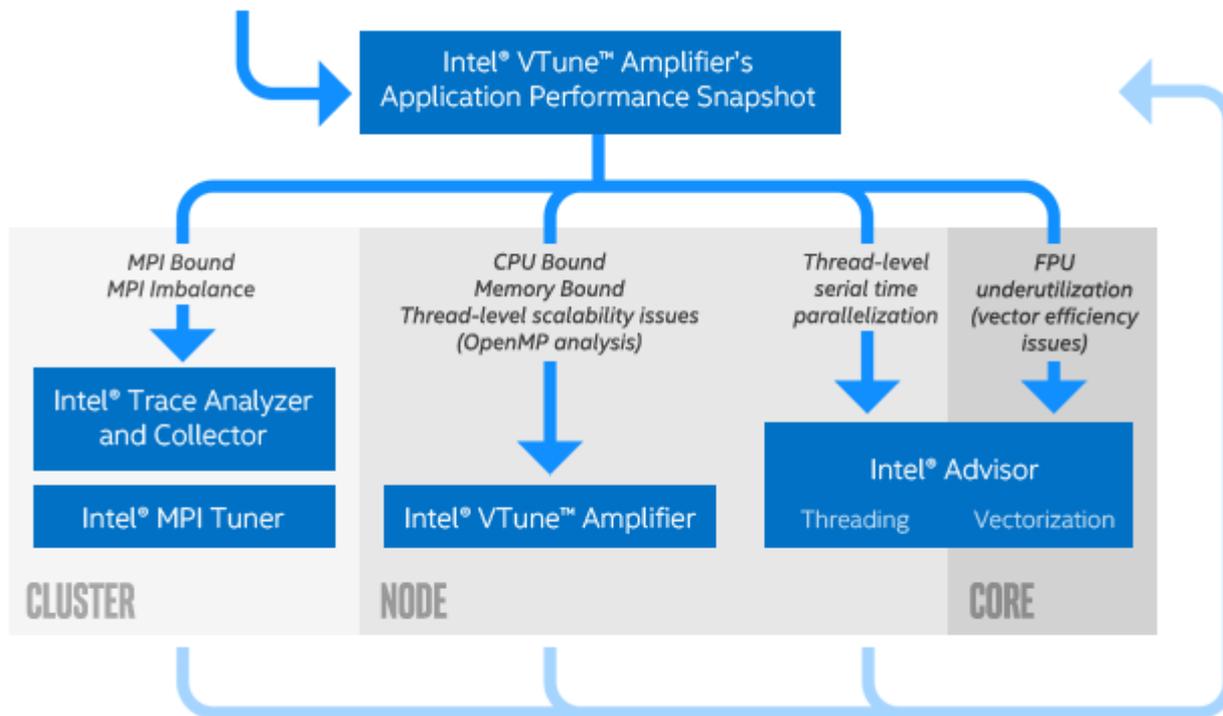
Carlos Rosales-Fernandez

# Tuning at Multiple Hardware Levels

Exploiting all features of modern processors requires good use of the available resources

- Core
  - Vectorization is critical with 512bit FMA vector units (32 DP ops/cycle)
  - Targeting the current ISA is fundamental to fully exploit vectorization
- Socket
  - Using all cores in a processor requires parallelization (MPI, OMP, ... )
  - Up to 64 Physical cores and 256 logical processors per socket on Theta!
- Node
  - Minimize remote memory access (control memory affinity)
  - Minimize resource sharing (tune local memory access, disk IO and network traffic)

# Tuning Workflow



# 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:

```
$ source /opt/intel/vtune_amplifier/apsvars.sh
```

```
$ export PMI_NO_FORK=1
```

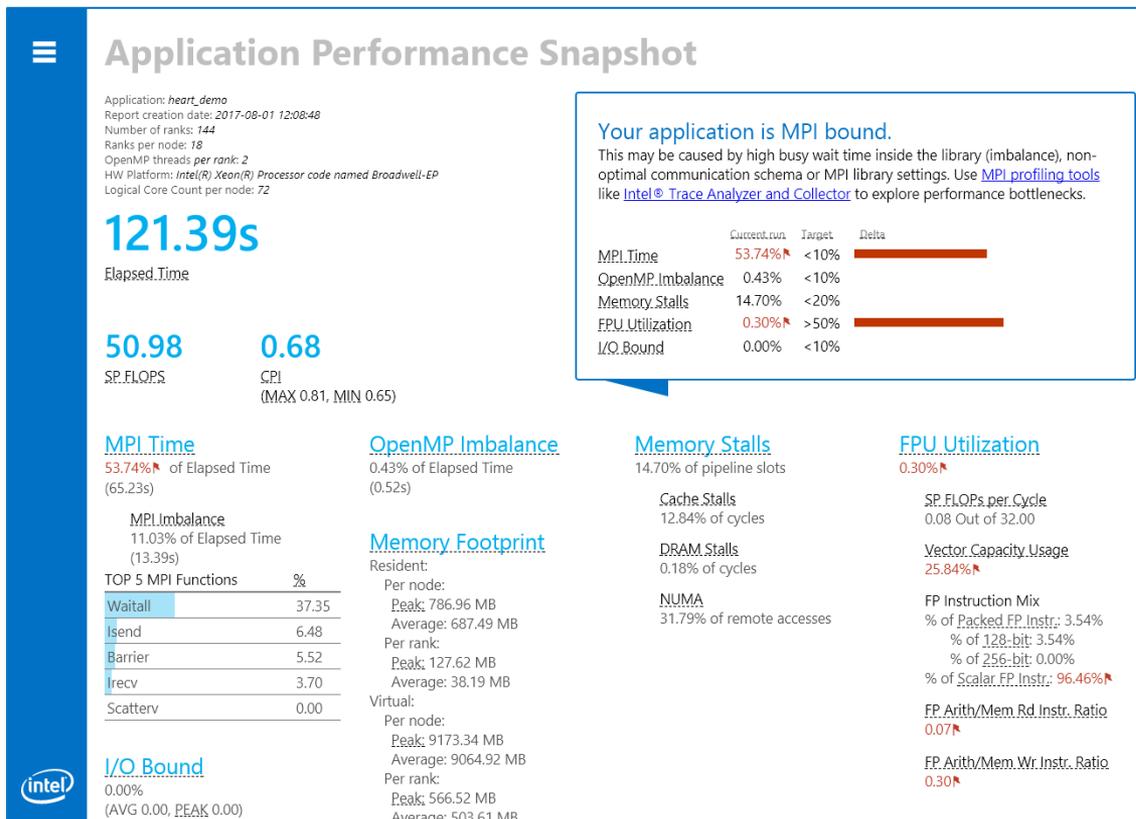
Launch your job in interactive or batch mode:

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

Produce text and html reports:

```
$ aps -report=./aps_result_ ...
```

# APS HTML Report



## Optimization Notice

Copyright © 2018, Intel Corporation. All rights reserved.  
\*Other names and brands may be claimed as the property of others.



# INTEL<sup>®</sup> ADVISOR

Vectorization and Threading

# Intel® Advisor

Modern HPC processors explore different level of parallelism:

- between the cores: multi-threading (Theta: 64 cores, 256 threads)
- within a core: vectorization (Theta: 8 DP elements, 16 SP elements)

Adapting applications to take advantage of such high parallelism is quite demanding and requires code modernization

The Intel® Advisor is a software tool for vectorization and thread prototyping

The tool guides the software developer to resolve issues during the vectorization process



# Typical Vectorization Optimization Workflow

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

1. Collect survey and tripcounts data
  - Investigate application place within roofline model
  - Determine vectorization efficiency and opportunities for improvement
2. Collect memory access pattern data
  - Determine data structure optimization needs
3. Collect dependencies
  - Differentiate between real and assumed issues blocking vectorization

# Using Intel® Advisor on Theta

Two options to setup collections: GUI (**advixe-gui**) or command line (**advixe-cl**).

I will focus on the command line since it is better suited for batch execution, but the GUI provides the same capabilities in a user-friendly interface.

I recommend taking a snapshot of the results and analyzing in a local machine (Linux, Windows, Mac) to avoid issues with lag.

Some things of note:

- Use **/projects** rather than **/home** for profiling jobs
- Set your environment:

```
$ source /opt/intel/advisor/advixe-vars.sh
```

```
$ export LD_LIBRARY_PATH=/opt/intel/advisor/lib64:$LD_LIBRARY_PATH
```

```
$ export PMI_NO_FORK=1
```

# Sample Script

```
#!/bin/bash
#COBALT -t 30
#COBALT -n 1
#COBALT -q debug-cache-quad
#COBALT -A <project>
```

→ Basic scheduler info (the usual)

```
export LD_LIBRARY_PATH=/opt/intel/advisor/lib64:$LD_LIBRARY_PATH
source /opt/intel/advisor/advixe-vars.sh
export PMI_NO_FORK=1
```

→ Environment setup

Two separate collections

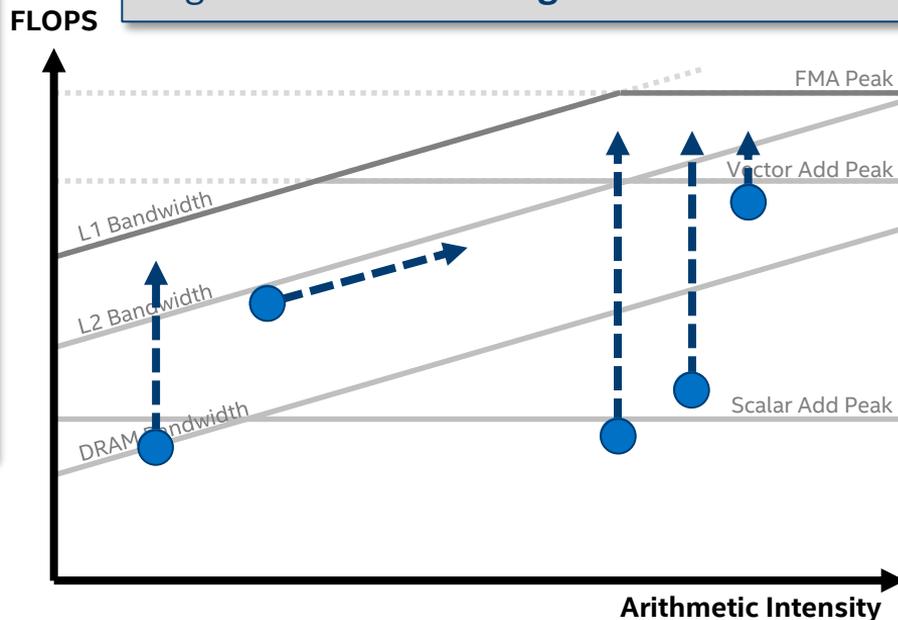
```
aprun -n 1 -N 1 advixe-cl -c survey --project-dir ./adv_res --search-dir src:=./ --search-dir bin:=./ -- ./exe
aprun -n 1 -N 1 advixe-cl -c tripcounts -flops-and-masks --project-dir ./adv_res \
    --search-dir src:=./ --search-dir bin:=./ -- ./exe
```

# Cache-Aware Roofline

## Next Steps

### 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 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 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.

# NBODY DEMONSTRATION

The naïve code that could

# Nbody gravity simulation

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

Let's consider a distribution of point masses  $m_1, \dots, m_n$  located at  $r_1, \dots, r_n$ .

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] += ...
```

# Collect Roofline Data

Starting with version 2 of the code we collect both survey and tripcounts data:

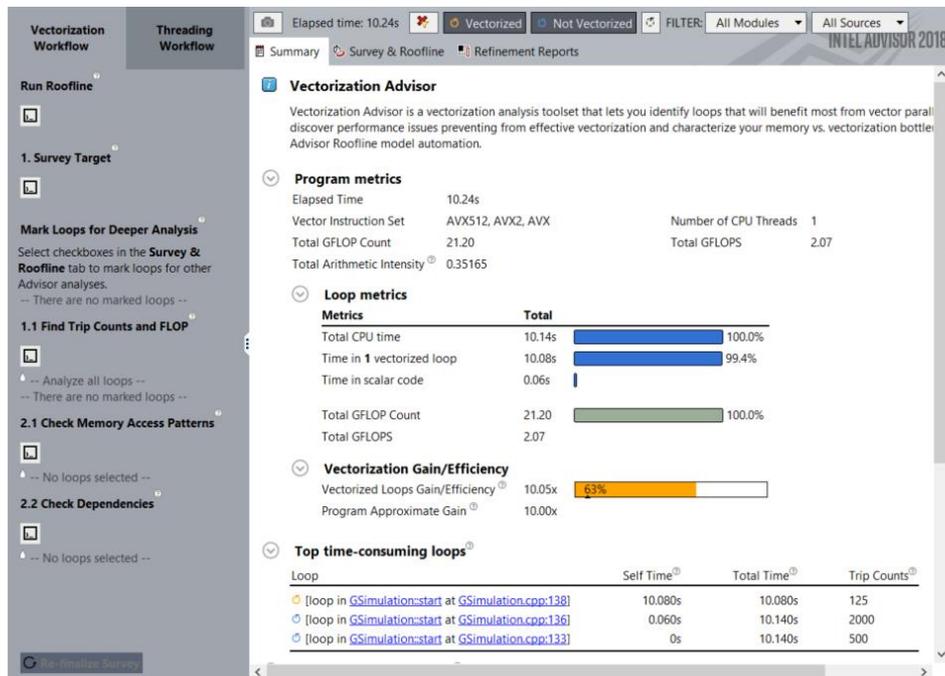
```
export LD_LIBRARY_PATH=/opt/intel/advisor/lib64:$LD_LIBRARY_PATH
source /opt/intel/advisor/advixe-vars.sh
export PMI_NO_FORK=1
aprun -n 1 -N 1 advixe-cl --collect survey --project-dir ./adv_res --search-dir src:=./ \
    --search-dir bin:=./ -- ./nbody.x
aprun -n 1 -N 1 advixe-cl --collect tripcounts -flops-and-masks --project-dir ./adv_res \
    --search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

And generate a portable snapshot to analyze anywhere:

```
advixe-cl --snapshot --project-dir ./adv_res --pack --cache-sources \
    --cache-binaries --search-dir src:=./ --search-dir bin:=./ -- nbody_naive
```

If finalization is too slow on compute add **-no-auto-finalize** to collection line.

# Summary Report



GUI left panel provides access to further tests

Summary provides overall performance characteristics

- Lists instruction set(s) used
- Top time consuming loops are listed individually
- Loops are annotated as vectorized and non-vectorized
- Vectorization efficiency is based on used ISA, in this case Intel® Advanced Vector Extensions 512 (AVX512)

# Survey Report (Source)

Elapsed time: 10.24s | Vectorized | Not Vectorized | FILTER: All Modules | All Sources | Loops And Functions | All Threads | OFF | Smart Mode

Summary | Survey & Roofline | Refinement Reports

Function Call Sites and Loops	Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops		FLOPS		
						Vector...	Efficiency	Gain E...	VL (Ve...	Self GFLOPS
[loop in GSimulation::start at GSimulation.cpp:138]	2 Inefficient gather/sc...	10.080s	10.080s	Vectorized (Body)		AVX5...	63%	10.05x	16	2.093
[loop in GSimulation::start at GSimulation.cpp:136]	1 Opportunity for outer l...	0.060s	10.140s	Scalar	inner loop was already v...					1.700
f_start		0.000s	10.140s	Function						
f_main		0.000s	10.140s	Function						
f GSimulation::start		0.000s	10.140s	Function						
[loop in GSimulation::start at GSimulation.cpp:133]	1 Data type conversions ...	0.000s	10.140s	Scalar	inner loop was already v...					

Source | Top Down | Code Analytics | Assembly | Recommendations | Why No Vectorization?

File: cache\_ee70d097069d33cb5b0f6e401a4f9d97\_GSimulation.cpp:138 GSimulation::start

Line	Source	Total Time	%	Loop/function Time	%	Traits
132	const double t0 = time.start();					
133	for (int s=1; s<=get_nsteps(); ++s)					
134	{					
135	t0 += time.start();					
136	for (i = 0; i < n; i++)// update acceleration					
137	{					
138	for (j = 0; j < n; j++)	0.100s		10.080s		
	[loop in GSimulation::start at GSimulation.cpp:138]					
	Vectorized AVX512ER_512; AVX512P_512 loop processes Float32; Int32; UInt32 data type(s) and includes 2-Source Perm					
	No loop transformations applied					
139	{					
140	real_type dx, dy, dz;					
141	real_type distanceSq = 0.0f;					
142	real_type distanceInv = 0.0f;					
143	}					

Selected (Total Time): 0.100s

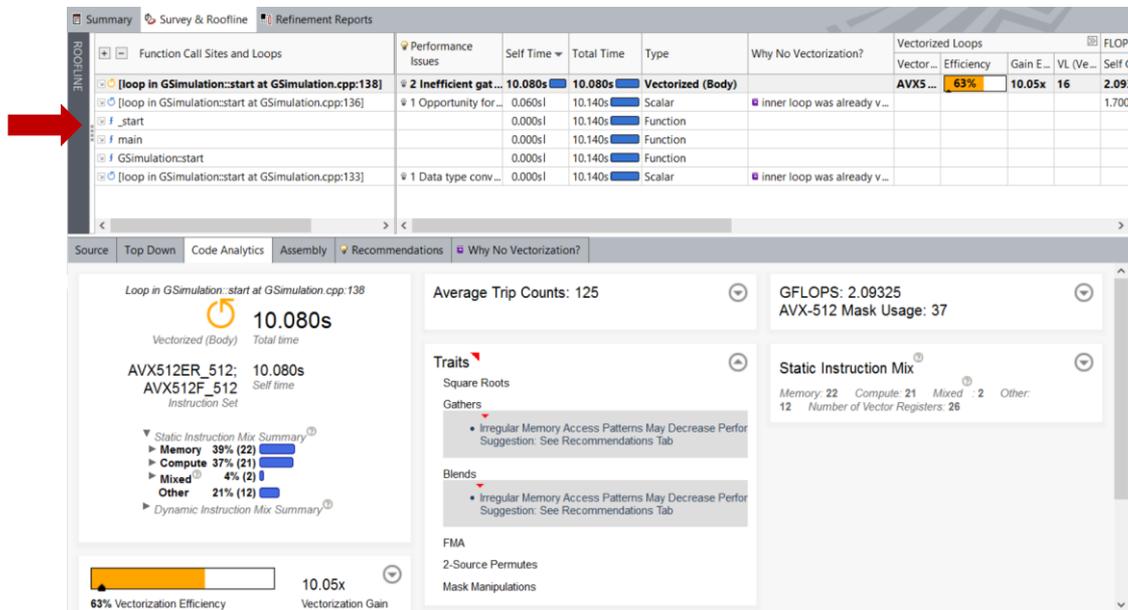
## Inline information regarding loop characteristics

- ISA used
- Types processed
- Compiler transformations applied
- Vector length used
- ...

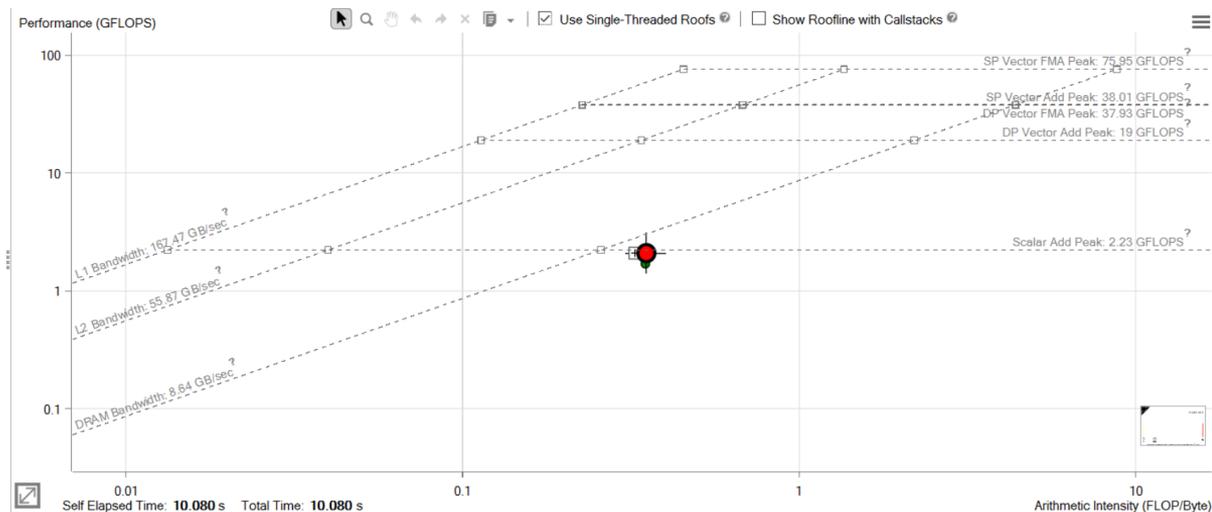
# Survey Report (Code Analytics)

## Detailed loop information

- Instruction mix
- ISA used, including subgroups
- Loop traits
  - FMA
  - Square root
  - Gathers / Blends point to memory issues and vector inefficiencies



# CARM Analysis



Using single threaded roof

Code vectorized, but performance on par with scalar add peak?

- Irregular memory access patterns force gather operations.
- Overhead of setting up vector operations reduces efficiency.

Next step is clear: perform a **Memory Access Pattern** analysis

# Memory Access Pattern Analysis (Refinement)

```
aprun -n 1 -N 1 advixe-cl --collect map --project-dir ./adv_res \  
--search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

Summary							Refinement Reports			MAP Source: GSimulation.cpp		
Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site Name	Recommendations						
[loop in start at GSimulation.cpp:1... No information available		33% / 33% / 33%	Mixed strides	5KB	loop_site_1	2 Inefficient gather/scatter instructions present						
Memory Access Patterns Report							Dependencies Report		Recommendations			
ID	Stride	Type	Source	Nested Function	Variable references	Max. Site Footprint	Modules	Site Name	Access Type			
P1	10,40	Constant stride	GSimulation.cpp:144		block 0x60a0b0 allocated at GSimulation.cpp:109	4KB	nbody.x	loop_site_1	Read			
<pre>142 real_type distanceInv = 0.0f; 143 144 dx = particles[j].pos[0] - particles[i].pos[0]; //iflop 145 dy = particles[j].pos[1] - particles[i].pos[1]; //iflop 146 dz = particles[j].pos[2] - particles[i].pos[2]; //iflop</pre>												
P2		Gather stride	GSimulation.cpp:144		block 0x60a0b0 allocated at GSimulation.cpp:109	5KB	nbody.x	loop_site_1	Read			
<pre>142 real_type distanceInv = 0.0f; 143 144 dx = particles[j].pos[0] - particles[i].pos[0]; //iflop 145 dy = particles[j].pos[1] - particles[i].pos[1]; //iflop 146 dz = particles[j].pos[2] - particles[i].pos[2]; //iflop</pre>												
P3		Parallel site information	GSimulation.cpp:144				nbody.x	loop_site_1				
<pre>142 real_type distanceInv = 0.0f; 143 144 dx = particles[j].pos[0] - particles[i].pos[0]; //iflop 145 dy = particles[j].pos[1] - particles[i].pos[1]; //iflop 146 dz = particles[j].pos[2] - particles[i].pos[2]; //iflop</pre>												
P5	0	Uniform stride	GSimulation.cpp:149			4B	nbody.x	loop_site_1	Read			
<pre>147 148 distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared; //6flops 149 distanceInv = 1.0f / sqrtf(distanceSqr); //1div+1sqrt 150 151 particles[i].acc[0] += dx * G * particles[j].mass * distanceInv * distanceInv * distanceInv; //6flops</pre>												

Storage of particles is in an Array Of Structures (AOS) style

This leads to regular, but non-unit strides in memory access

- 33% unit
- 33% uniform, non-unit
- 33% non-uniform

Re-structuring the code into a Structure Of Arrays (SOA) may lead to unit stride access and more effective vectorization



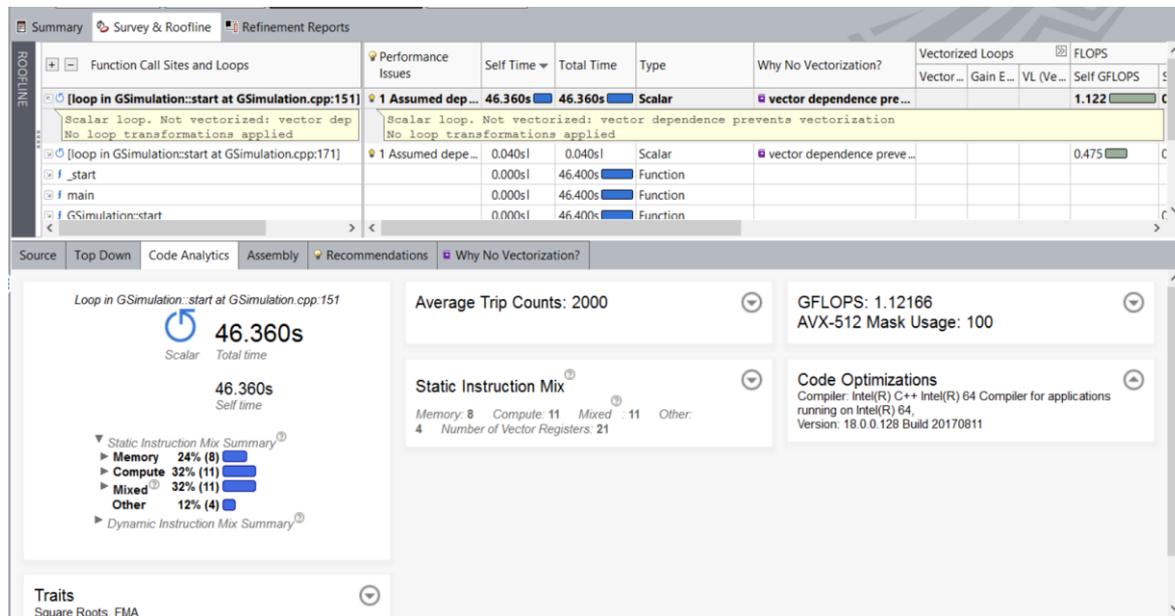
# Performance After Data Structure Change

In this new version ( version 3 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

# Dependencies Analysis (Refinement)

```
aprun -n 1 -N 1 advixe-cl --collect dependencies --project-dir ./adv_res \  
--search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

The screenshot displays the Intel Advisor interface for dependency analysis. The top section shows a summary of the analysis for a loop in `GSimulation.cpp` at site `loop_site_1`, indicating a proven (real) dependency present. Below this, the 'Problems and Messages' table lists several issues, including a 'Read after write dependency' (P4) and 'Parallel site information' (P1, P3, P5, P6). The detailed view for P4 shows the code locations for the dependency, including the start of the loop and the calculation of `distanceInv` and `particles->acc_x[i]`.

ID	Type	Site Name	Sources	Modules	State
P1	Parallel site information	loop_site_1	GSimulation.cpp	nbody.x	✓ Not a problem
P3	Read after write dependency	loop_site_1	GSimulation.cpp	nbody.x	New
P4	Read after write dependency	loop_site_1	GSimulation.cpp; main.cpp	nbody.x	New
P5	Read after write dependency	loop_site_1	GSimulation.cpp	nbody.x	New
P6	Read after write dependency	loop_site_1	GSimulation.cpp	nbody.x	New

ID	Instruction Address	Description	Source	Function	Variable references	Module	State
X3	0x401c85	Parallel site	GSimulation.cpp:157	start		nbody.x	New
X6	0x401cb8, 0x401d17	Read	GSimulation.cpp:164	start	register XMM1	nbody.x	New
X7	0x401d1e	Write	GSimulation.cpp:164	start		nbody.x	New

Dependencies analysis has high overhead:

- Run on reduced workload

Advisor Findings:

- RAW dependency
- Multiple reduction-type dependencies

# Recommendations

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];
}
```

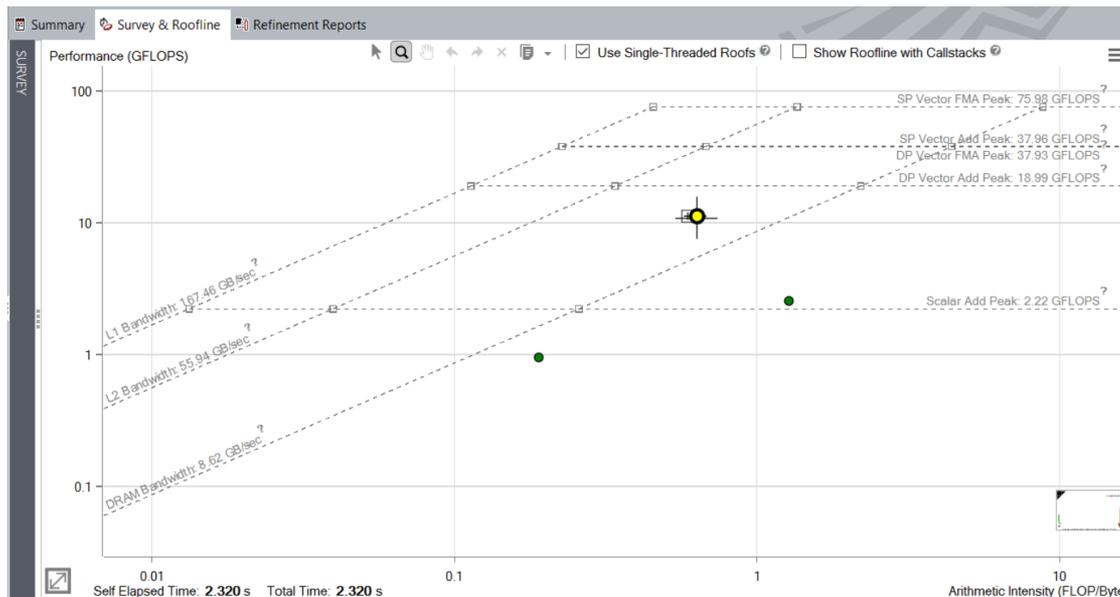
### ISSUE: PROVEN (REAL) DEPENDENCY PRESENT

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](#)

# Performance After Resolved Dependencies



New memory access pattern plus vectorization produces much improved performance!

# What next?

Performance per core may be improved, but it is capped at ~5x current value.

Let's explore threading with a suitability analysis.

- Recompile including annotation definitions
- Add headers to file
- Annotate suggested loops
- Run suitability collection

Elapsed time: 2.43s

Vectorized Not Vectorized FILTER: All Modules All Sources

Summary Survey & Roofline Refinement Reports Annotation Report Suitability Report

Threading Advisor is a threading design and prototyping tool that lets you analyze, design, tune, and check threading design without disrupting your normal development.

**No source files found to scan for annotations.**  
No appropriate source files were found in your project.

**Program metrics**

Elapsed Time	2.43s	Number of CPU Threads	1
Vector Instruction Set	AVX512, AVX2, AVX	Total GFLOPS	10.77
Total GFLOP Count	26.12		
Total Arithmetic Intensity <sup>®</sup>	0.63431		

**Loop metrics**

**Vectorization Gain/Efficiency**

**Top time-consuming loops<sup>®</sup>**

Consider adding parallel site and task annotations around these time-consuming loops found during Survey analysis.

Loop	Self Time <sup>®</sup>	Total Time <sup>®</sup>	Trip Counts <sup>®</sup>
[loop in GSimulation:start at GSimulation.cpp:143]	0s	2.380s	500
[loop in GSimulation:start at GSimulation.cpp:146]	0.040s	2.360s	2000
[loop in GSimulation:start at GSimulation.cpp:154]	2.320s	2.320s	125
[loop in GSimulation:start at GSimulation.cpp:177]	0.020s	0.020s	2000

**Collection details**

# Annotating the code

Add annotations as shown on the left sample

Complex sites may be analyzed in more detail using task sections if needed

- ANNOTATE\_SITE\_BEGIN / ANNOTATE\_SITE\_END
- ANNOTATE\_TASK\_BEGIN / ANNOTATE\_TASK\_END

Recompile including annotation definitions:

```
-I/opt/intel/advisor/include
```

Collect suitability data

```
aprun -n 1 -N 1 advixe-cl --collect suitability --project-dir ./adv_res \  
--search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

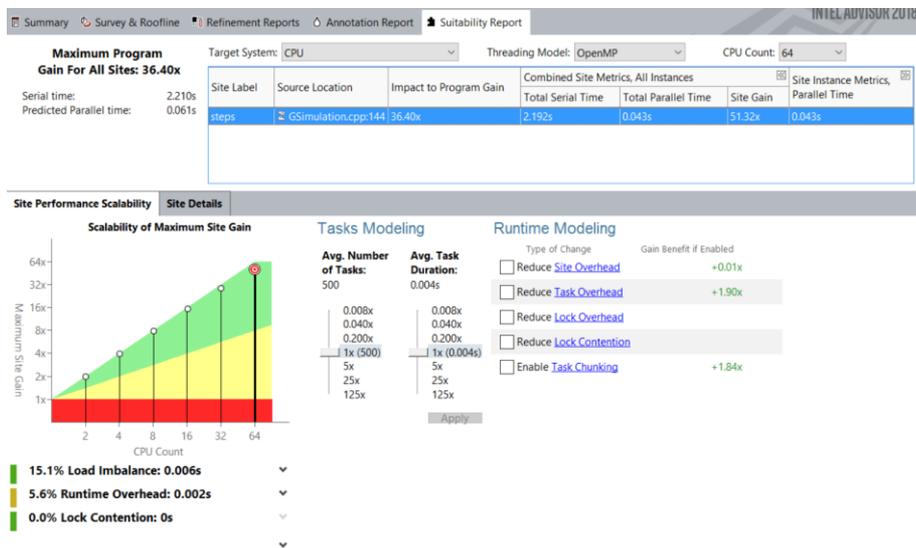
```
#include "advisor-annotate.h"  
...  
ANNOTATE_SITE_BEGIN(steps)  
for (int s=1; s<=get_nsteps(); ++s)  
{  
    ...  
    ANNOTATE_TASK_BEGIN(particles)  
    for (i = 0; i < n; i++)  
    {  
        ...  
    }  
    ANNOTATE_TASK_END(particles)  
}  
ANNOTATE_SITE_END(steps)
```

# Suitability report

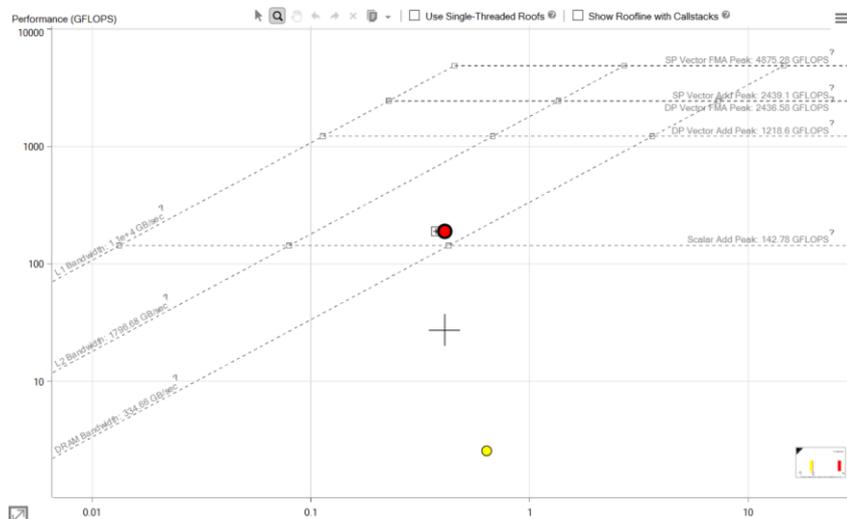
Good speedup expected, but far from ideal ( ~56% efficiency ).

Modeling shows that increasing the task length would improve efficiency.

Next step: add omp parallel region to code and re-rest



# Roofline for Threaded Version



Now using regular roofline, instead of single-threaded

Still room for improvement, but at this point we need additional detail regarding shared resource utilization

```
for (int s=1; s<=get_nsteps(); ++s)
{
    ts0 += time.start();

    #pragma omp parallel for
    for (i = 0; i < n; i++)    // update acceleration
    {
        ...
        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 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];
```

**INTEL<sup>®</sup> VTUNE<sup>™</sup> 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

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

VTune Amplifier takes over where Intel® Advisor left

# Predefined Collections

Many available analysis types:

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

Python Support

# The HPC Performance Characterization Analysis

## Threading: CPU Utilization

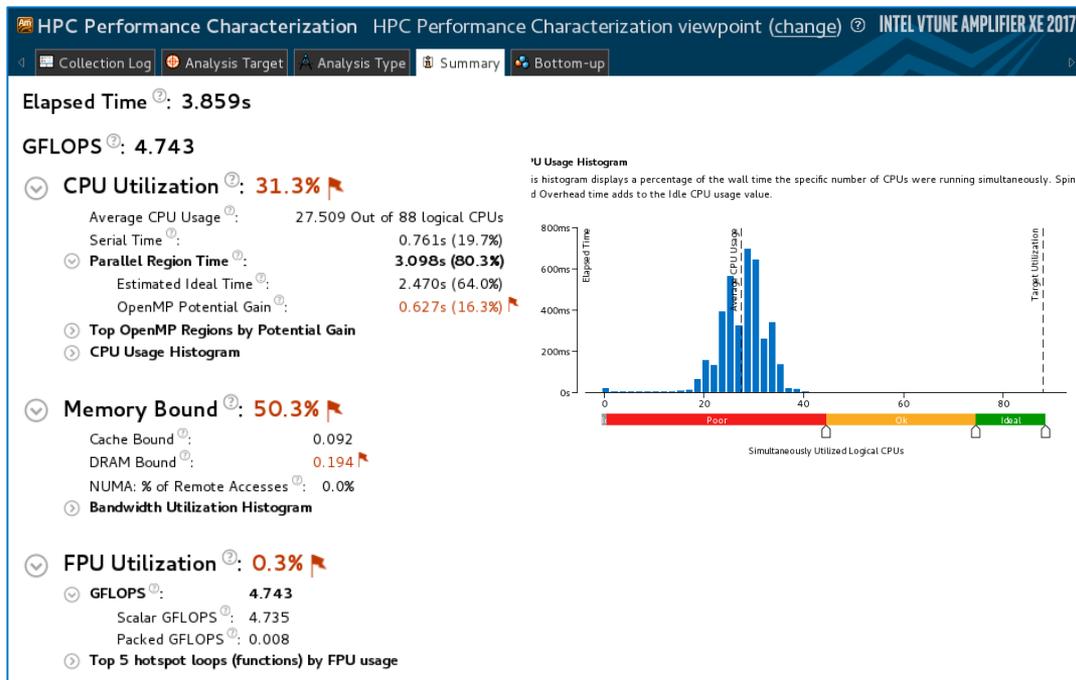
- Serial vs. Parallel time
- Top OpenMP regions by potential gain
- Tip: Use hotspot OpenMP region analysis for more detail

## Memory Access Efficiency

- Stalls by memory hierarchy
- Bandwidth utilization
- Tip: Use Memory Access analysis

## Vectorization: FPU Utilization

- FLOPS<sup>†</sup> estimates from sampling
- Tip: Use Intel Advisor for precise metrics and vectorization optimization



<sup>†</sup> For 3rd, 5th, 6th Generation Intel® Core™ processors and second generation Intel® Xeon Phi™ processor code named Knights Landing.

# Memory Access Analysis

## Tune data structures for performance

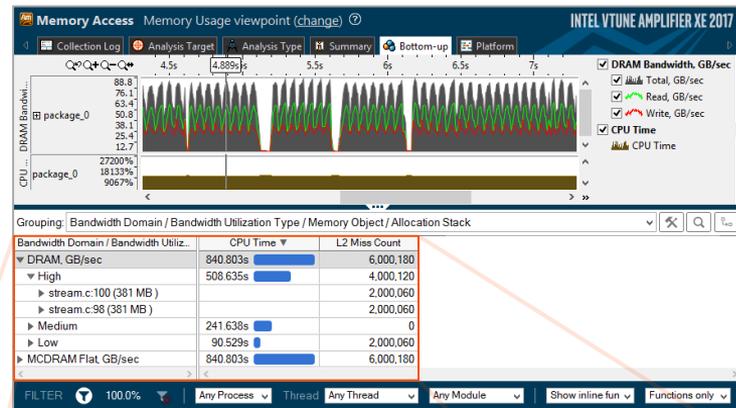
- Attribute cache misses to data structures (not just the code causing the miss)
- Support for custom memory allocators

## Optimize NUMA latency & scalability

- True & false sharing optimization
- Auto detect max system bandwidth
- Easier tuning of inter-socket bandwidth

## Easier install, Latest processors

- No special drivers required on Linux\*
- Intel® Xeon Phi™ processor MCDRAM (high bandwidth memory) analysis



Bandwidth Domain / Bandwidth Utiliz...	CPU Time	L2 Miss Count
▼ DRAM, GB/sec	840.803s	6,000,180
▼ High	508.635s	4,000,120
▶ stream.c:100 (381 MB)		2,000,060
▶ stream.c:98 (381 MB)		2,000,060
▶ Medium	241.638s	0
▶ Low	90.529s	2,000,060
▶ MCDRAM Flat, GB/sec	840.803s	6,000,180

# Using Intel® VTune™ Amplifier on Theta

Two options to setup collections: GUI (`amplxe-gui`) or command line (`amplxe-cl`).

I will focus on the command line since it is better suited for batch execution, but the GUI provides the same capabilities in a user-friendly interface.

Some things of note:

- Use `/projects` rather than `/home` for profiling jobs
- Set your environment:

```
$ source /opt/intel/vtune_amplifier/amplxe-vars.sh
```

```
$ export LD_LIBRARY_PATH=/opt/intel/vtune_amplifier/lib64:$LD_LIBRARY_PATH
```

```
$ export PMI_NO_FORK=1
```

# Sample Script

```
#!/bin/bash
#COBALT -t 30
#COBALT -n 1
#COBALT -q debug-cache-quad
#COBALT -A <project>
```

→ Basic scheduler info (the usual)

```
export LD_LIBRARY_PATH=/opt/intel/vtune_amplifier/lib64:$LD_LIBRARY_PATH
source /opt/intel/vtune_amplifier/amplxe-vars.sh
export PMI_NO_FORK=1
export OMP_NUM_THREADS=64; export OMP_PROC_BIND=spread; export OMP_PLACES=cores
```

→ Environment setup

```
aprun -n 1 -N 1 -cc depth -d 256 -j 4 amplxe-cl -c hotspots -knob analyze-openmp=true \
-r ./vtune_res -- ./exe
```

Invoke VTune™ Amplifier

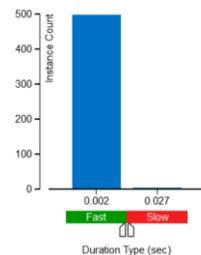
# Hotspots analysis for nbody demo (ver7: threaded)



## OpenMP Region Duration Histogram

This histogram shows the total number of region instances in your application executed with a specific duration. High number of slow instances may signal a performance bottleneck. Explore the data provided in the Bottom-up, Top-down Tree, and Timeline panes to identify code regions with the slow duration.

OpenMP Region: startSomp\$parallel\_64@unknown:146.182

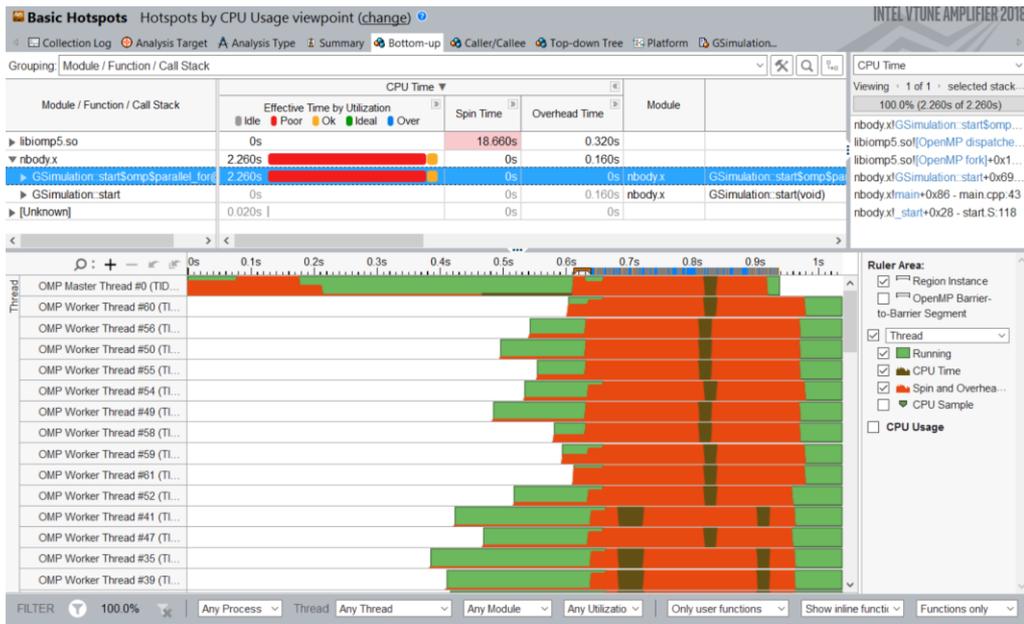


Lots of spin time indicate issues with load balance and synchronization

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

Let's look at the timeline for each thread to understand things better...

# Bottom-up 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.

# Python

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

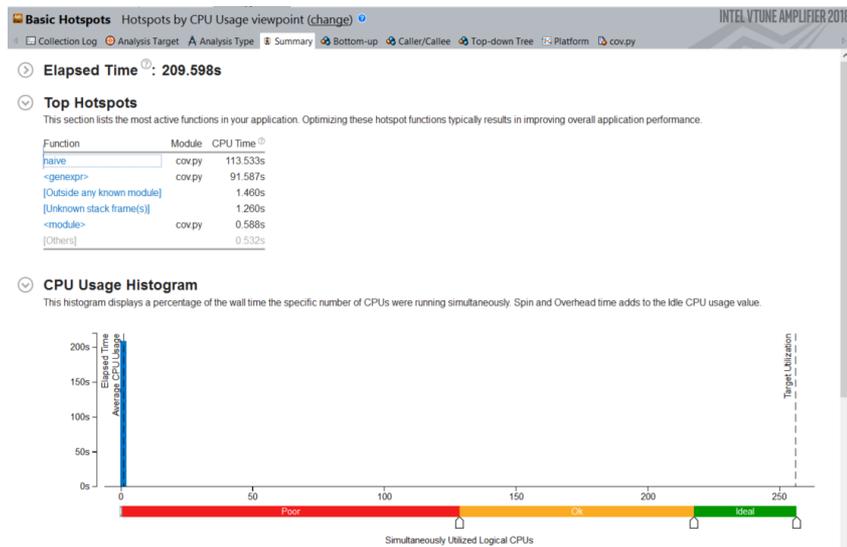
- The “application” should be the full path to the python interpreter used
- The python code should be passed as “arguments” to the “application”

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 ampxe-cl -c hotspots -r vt_pytest \  
-- /usr/bin/python ./cov.py naive 100 1000
```



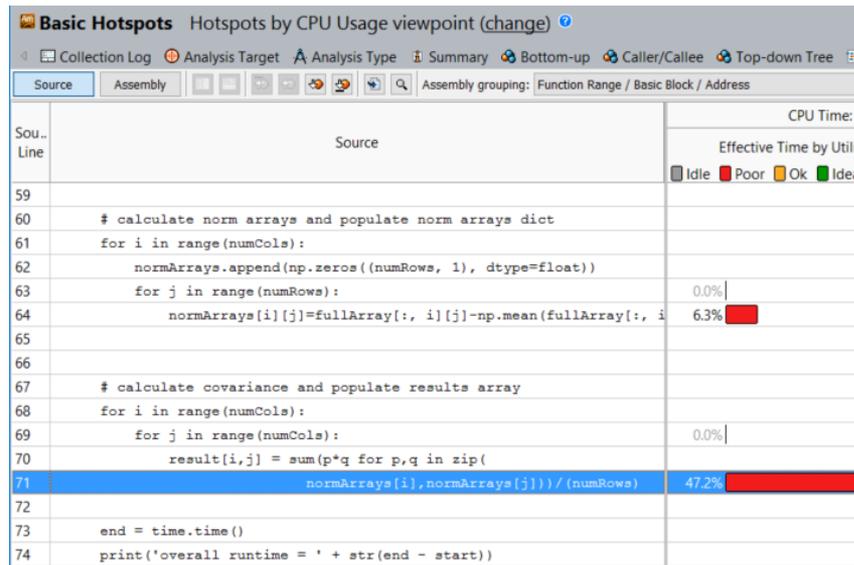
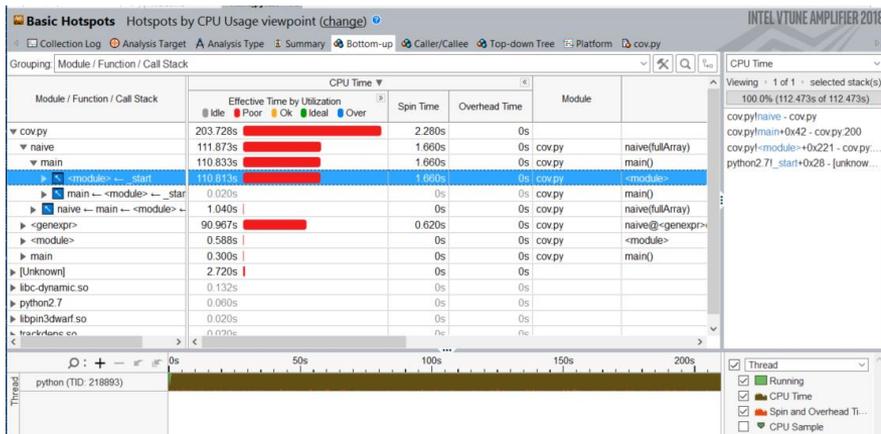
Naïve implementation of the calculation of a covariance matrix

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

# Useful Options on Theta

If finalization is slow you can use `-finalization-mode=deferred` and simply finalize on a login node or a different machine

If the collection stops because too much data has been collected you can override that with the `-data-limit=0` option (unlimited) or to a number (in MB)

Use the `-trace-mpi` option to allow VTune Amplifier to assign execution to the correct task when not using the Intel® MPI Library.

Reduce results size by limiting your collection to a single node using an `mpmd` style execution:

```
aprun -n X1 -N Y amplxe-cl -c hpc-performance -r resdir -- ./exe : \  
-n X2 -N Y ./exe
```

# EMON Collection

General Exploration analysis may be performed using EMON

- Reduced size of collected data
- Overall program data, no link to actual source (only summary)
- Useful for initial analysis of production and large scale runs
- Currently available as experimental feature

```
export AMPLXE_EXPERIMENTAL=emon
```

```
aprun [...] amplxe-cl -c general-exploration -knob summary-mode=true [...]
```

# Resources

## Product Pages

- <https://software.intel.com/sites/products/snapshots/application-snapshot>
- <https://software.intel.com/en-us/advisor>
- <https://software.intel.com/en-us/intel-vtune-amplifier-xe>

## Detailed Articles

- <https://software.intel.com/en-us/articles/intel-advisor-on-cray-systems>
- <https://software.intel.com/en-us/articles/using-intel-advisor-and-vtune-amplifier-with-mpi>
- <https://software.intel.com/en-us/articles/profiling-python-with-intel-vtune-amplifier-a-covariance-demonstration>

# Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

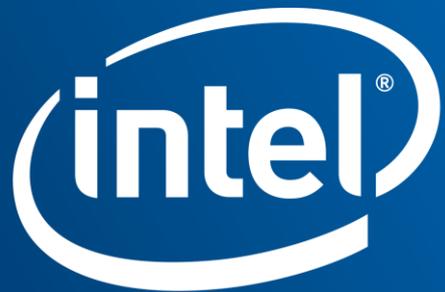
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit [www.intel.com/benchmarks](http://www.intel.com/benchmarks).

Copyright © 2018, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

## Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804



Software