

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.

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



PERFORMANCE OPTIMIZATION VTUNE & ADVISOR

Paulius Velesko

Application Engineer

paulius.velesko@intel.com

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)
 - Cache use needed to feed vector units
- Socket
 - Using all cores in a processor requires parallelization (MPI, OMP, ...)
 - Using coherent, shared socket caches
- Node
 - Minimize remote memory access (control memory affinity)
 - Minimize resource sharing (tune local memory access, disk IO and network traffic)

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>

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, \dots, r_n and have masses m_1, \dots, 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

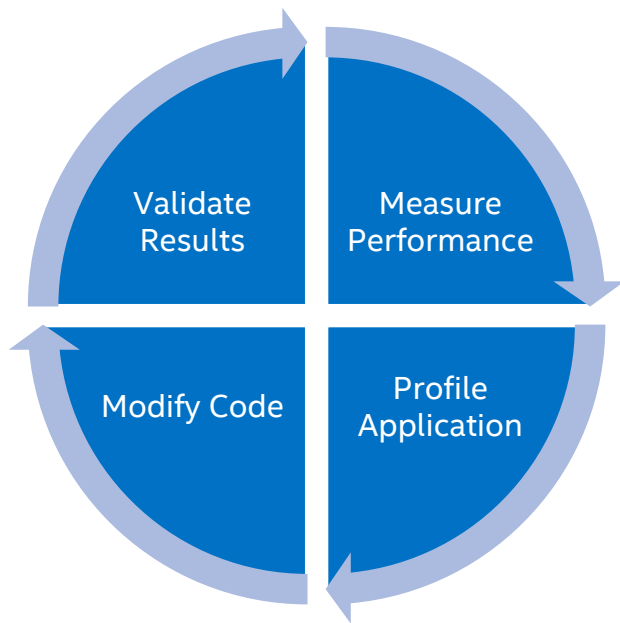
FREE* performance metrics

Compile with `-qopt-report=5`

- Which loops were vectorized
 - Vector Length
 - Estimated Gain
 - Alignment
 - Scatter/Gather
- Prefetching
- Issues preventing vectorization
- Inline reports
- Interprocedural optimizations
- Register Spills/Fills

```
LOOP BEGIN at ../src/timestep.F(4835,13)
remark #15389: vectorization support: reference nbd(i) has unaligned access [ ../src/timestep.F(4836,16) ]
remark #15381: vectorization support: unaligned access used inside loop body
remark #15335: loop was not vectorized: vectorization possible but seems inefficient. Use vector always directive or -vec-threshold0 to override
remark #15329: vectorization support: irregularly indexed store was emulated for the variable <coefd_(nbd_(i))>, part of index is read from memory
remark #15305: vectorization support: vector length 2
remark #15399: vectorization support: unroll factor set to 4
remark #15309: vectorization support: normalized vectorization overhead 0.139
remark #15450: unmasked unaligned unit stride loads: 1
remark #15463: unmasked indexed (or scatter) stores: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 4
remark #15477: vector cost: 4.500
remark #15478: estimated potential speedup: 0.880
remark #15488: --- end vector cost summary ---
remark #25439: unrolled with remainder by 2
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**

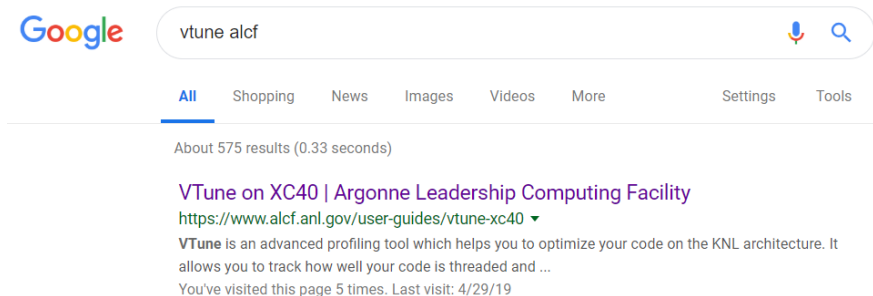
Where do I start?

</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



Version Optimizations

- Ver0
 - Initial implementation
- Ver1
 - -xMIC-AVX512
- Ver2
 - Use only floats
- Ver3
 - AoS -> SoA + SIMD Reduce

INTEL[®] ADVISOR

Vectorization and Static Analysis

<https://www.alcf.anl.gov/user-guides/advisor-xc40>

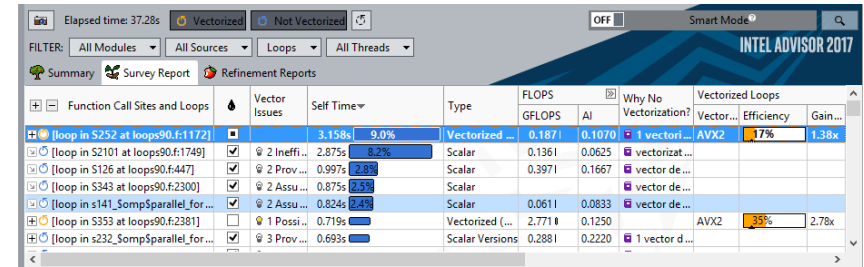
Intel® Advisor – Vectorization Optimization

Faster Vectorization Optimization:

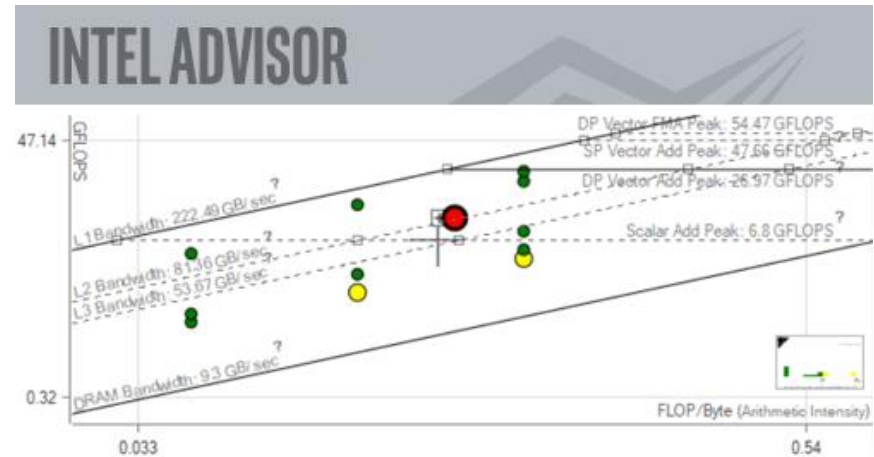
- Vectorize where it will pay off most
- Quickly ID what is blocking vectorization
- Tips for effective vectorization
- Safely force compiler vectorization
- Optimize memory stride

Roofline model analysis:

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



Function Call Sites and Loops	Vector Issues	Self Time	Type	FLOPS	AI	Why No Vectorization?	Vectorized Loops	
[loop in S252 at loops90.f:1172]	3.158s 9.0%	Vectorized ...	0.1871	0.1070	1 vectori...	AVX2	17%	1.38x
[loop in S2101 at loops90.f:1749]	2.875s 8.2%	2 Ineffi...	Scalar	0.1361	0.0625	vectorizat...		
[loop in S126 at loops90.f:447]	0.997s 2.8%	2 Prov...	Scalar	0.3971	0.1667	vector de...		
[loop in S343 at loops90.f:2300]	0.875s 2.5%	2 Assu...	Scalar			vector de...		
[loop in s141_Somp\$parallel_for...	0.824s 2.4%	2 Assu...	Scalar	0.0611	0.0833	vector de...		
[loop in S353 at loops90.f:2381]	0.719s	1 Possi...	Vectorized (...	2.771	0.1250	AVX2	35%	2.76x
[loop in s232_Somp\$parallel_for...	0.693s	3 Prov...	Scalar Versions	0.2881	0.2220	1 vector d...		



Add Parallelism with Less Effort, Less Risk and More Impact

<http://intel.ly/advisor-xe>

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

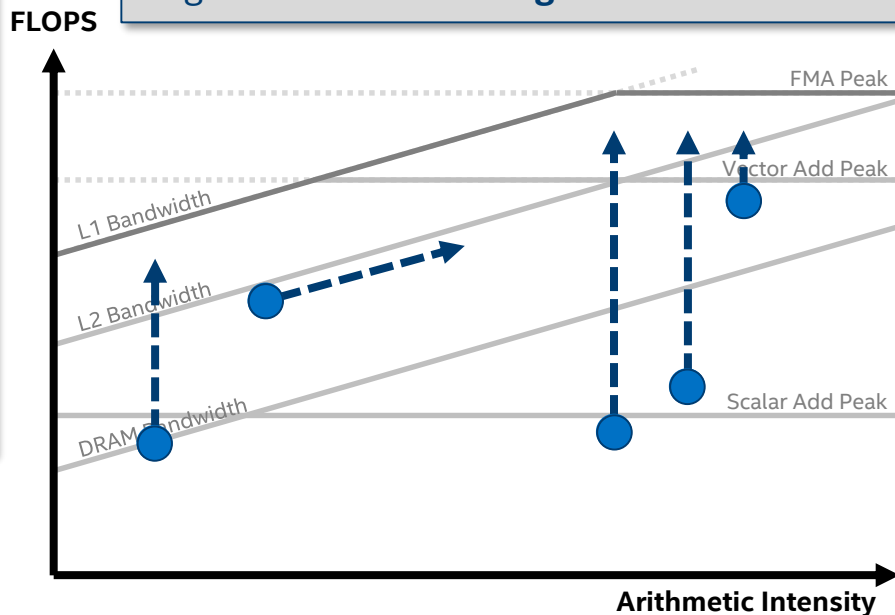
Check “Traits” in the Survey to see if FMAs are used. If not, try altering your code or compiler flags to **induce FMA usage**.

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.



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**

Use -h Option!

advixe-cl -h collect

Examples:

- 1) Survey the application to determine hotspots.
`advixe-cl --collect survey --project-dir ./advi --search-dir src:r=./src
-- ./bin/myApplication`
- 2) Collect memory access patterns data with specified loops for analysis.
`advixe-cl --collect map --mark-up-list=5,10,12 --project-dir ./advi
--search-dir src:r=./src -- ./bin/myApplication`
- 3) Collect survey data on 4 nodes of MPI cluster into the shared ./advi project directory.
`mpirun -n 4 advixe-cl --project-dir ./advi --collect survey
-- <PATH>/mpi-sample/1_mpi_sample_serial`
- 4) Collect dependencies data for all loops that are both innermost and hold above 2% of the total CPU time.
`advixe-cl --collect dependencies --project-dir ./advi --loops="loop-height=0,total-time>2"
-- ./bin/myApplication`

Generate Advisor Command Lines from the GUI

The screenshot shows the Intel Advisor application window with the 'Survey & Roofline' tab selected. A 'No Data' warning is displayed in the main area. A dialog box titled 'Copy Command Line to Clipboard' is open, showing a command line for MPI. The command line is: `mpirun -n 1 -gtool *advixe-cl -collect tripcounts -module-filter-mode=exclude -trip-counts -no-flop -no-stacks -no-callstack-tripcounts -no-flops-and-masks -no-callstack-flops -stack-stitching -no-profile-python -auto-finalize -project-dir C:\Users\paulius\test:0* "C:\Users\paulius\AppData\Local\Apps\PeXip Connect \pexip-connect.exe"`. The dialog also has a 'Generate command line for MPI' checkbox checked and a 'Copy' button.

Intel Advisor GUI showing the 'Copy Command Line to Clipboard' dialog box. The dialog displays the command line for MPI:

```
mpirun -n 1 -gtool *advixe-cl -collect tripcounts -module-filter-mode=exclude -trip-counts -no-flop -no-stacks -no-callstack-tripcounts -no-flops-and-masks -no-callstack-flops -stack-stitching -no-profile-python -auto-finalize -project-dir C:\Users\paulius\test:0* "C:\Users\paulius\AppData\Local\Apps\PeXip Connect \pexip-connect.exe"
```

The dialog also includes a 'Copy' button and a 'Close' button. A checkbox 'Generate command line for MPI' is checked.

Collect survey and tripcounts (roofline)

```
$ module load advisor
```

```
$ cd /projects/intel/pvelesko/nody-demo/ver0
```

```
$ make
```

```
$ cp /soft/perftools/intel/advisor/advixe.qsub ./
```

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

View Result

X-forwarding is not recommended.

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

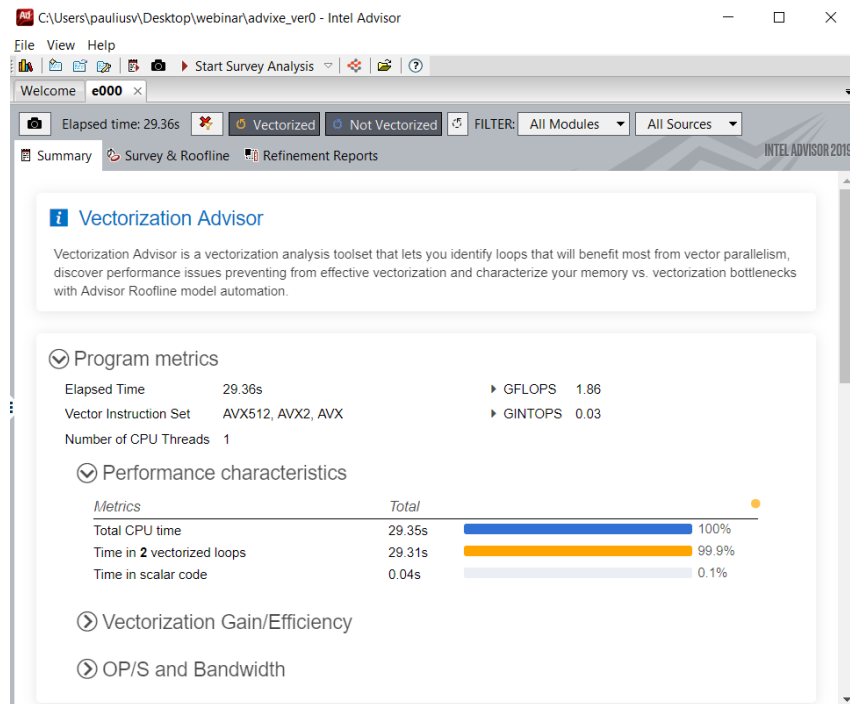
or

Generate a snapshot:

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

then scp to your local machine

Summary Report



Summary provides overall performance characteristics

Top time consuming loops are listed individually

Vectorization efficiency is based on used ISA

Survey Report (Source Tab)

The screenshot displays the Intel Advisor 2018 interface. At the top, a yellow banner indicates "Higher instruction set architecture (ISA) available" with the advice to "Consider recompiling your application using a higher ISA." Below this, a table lists performance issues. The primary issue is "1 Data type conversion" in a loop at GSsimulation.cpp:138, which is "Vectorized (Body)" with an efficiency of 91% and a gain of 1.82x. The table also shows other loops that are not vectorized, such as "inner loop was already v...".

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 GSsimulation::start at GSsimulation.cpp:138]	1 Data type con...	90.600s	90.600s	Vectorized (Body)		SSE2	91%	1.82x	2	0.993
[loop in GSsimulation::start at GSsimulation.cpp:136]		0.020s	90.620s	Scalar	inner loop was already v...					0.150
f_start		0.000s	90.620s	Function						
f_main		0.000s	90.620s	Function						
f GSsimulation::start		0.000s	90.620s	Function						

Below the table, the "Source" tab is active, showing the source code for the selected loop at GSsimulation.cpp:138. The code includes a nested loop structure. A tooltip for the inner loop at line 138 provides details: "Vectorized SSE; SSE2 loop processes Float32; Float64; Int64 data type(s) and includes Square Roots; Type Conversion; No loop transformations applied". A tooltip for the outer loop at line 136 states: "Scalar remainder loop [not executed]; No loop transformations applied".

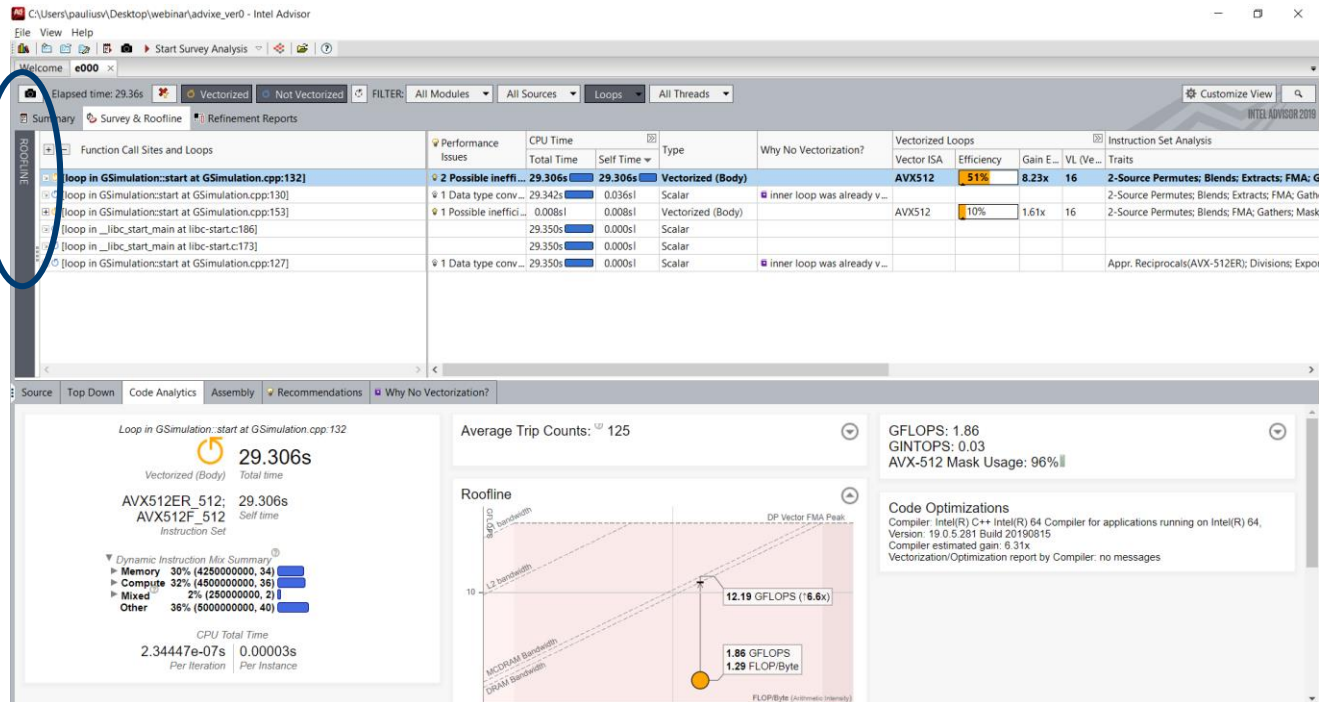
```
File: cache_76525f07bef212a1fcb8f6b3b3ab2632_GSimulation.cpp:138 GSimulation::start
Line Source Total Time % Loop/Function Time % Traits
135 for (int i = 0; i < n; i++) {
136     {
137         for (j = 0; j < n; j++)
138             for (k = 0; k < n; k++)
139                 {
140                     // ...
141                 }
142             }
143     }
```

Notice the following:

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

All of these elements may affect performance

Survey Report (Code Analytics Tab)

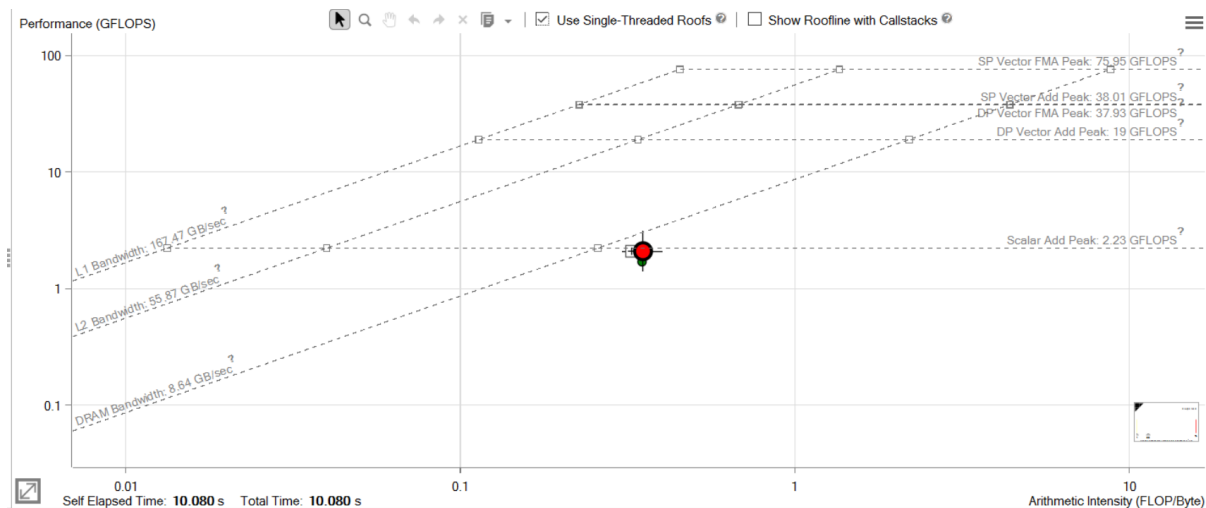


Analytics tab contains a wealth of information

- 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.

CARM (Cache-aware roofline model) 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)

Modify `advixe.qsub` to collect “survey” followed by “map”

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

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

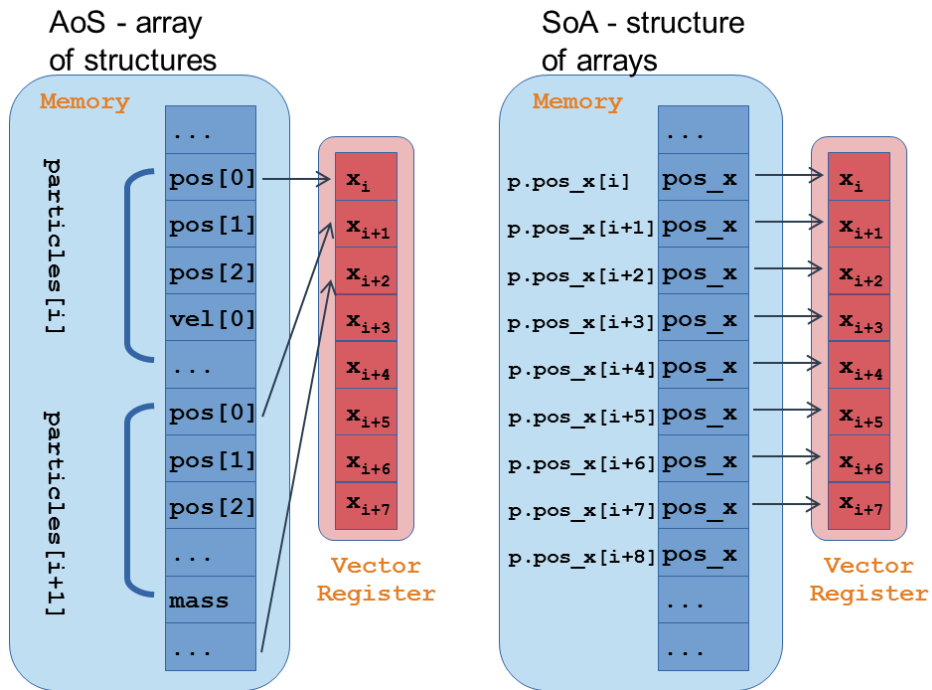
Re-structuring the code into a Structure Of Arrays (SOA) may lead to unit stride access and more effective vectorization

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
```

Constant stride are more complex

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

Non predictable access are usually bad

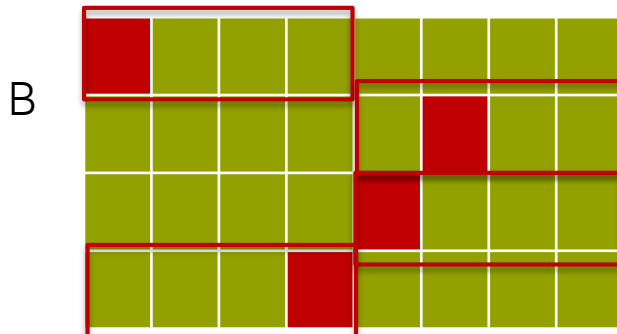
```
for (i=0; i<N; i++)  
    A[i] = B[C[i]]*d
```



For B, 1 cache line load computes 4 DP



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



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

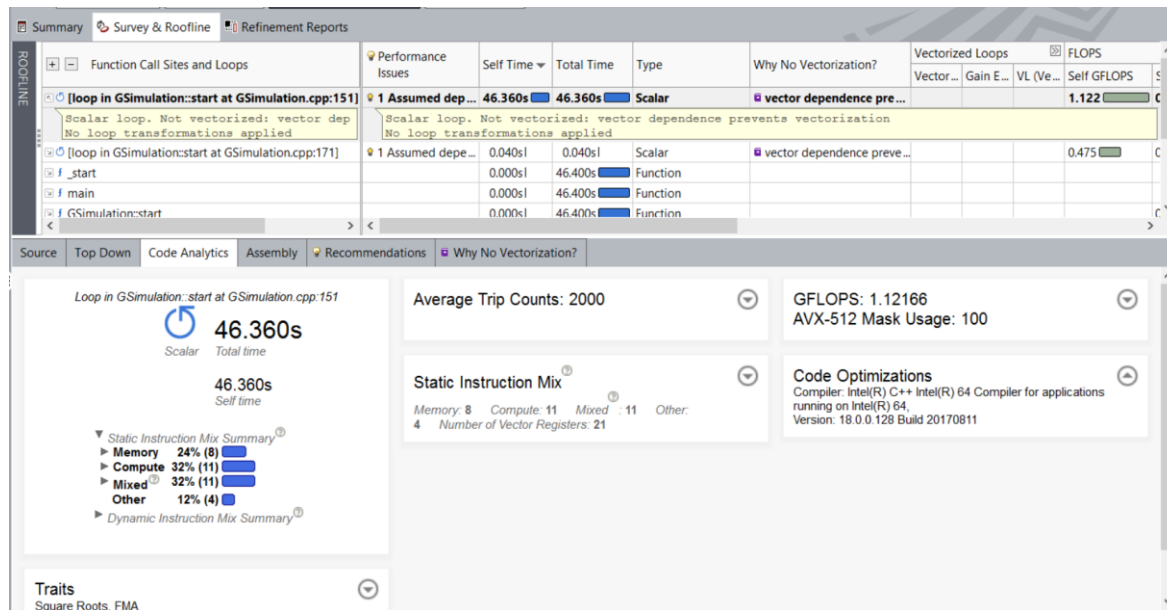
Performance After Data Structure Change

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)

Modify `advixe.qsub` to collect "survey" followed by "dependencies"

`qsub advixe.qsub ./ver3/nbody.x`

The screenshot displays the Intel Advisor interface for a dependency analysis. The top navigation bar includes 'Summary', 'Survey & Roofline', 'Refinement Reports', and 'Dependencies Source: GSimulation.cpp'. Below this, a table shows site location details for 'loop in start at GSimulation.cpp:157' on 'RAW-4', indicating 'No information available' for various metrics and '1 Proven (real) dependency present'. The main area is divided into 'Problems and Messages' and 'Read after write dependency: Code Locations'. The 'Problems and Messages' table lists several 'Read after write dependency' issues (P1-P6) between 'loop_site_1' and 'nbody.x'. The 'Code Locations' table shows the specific instructions causing these dependencies, such as calculating distanceInv and updating particle positions. A right-hand sidebar provides a 'Filter' and a summary of findings by severity, type, source, and module.

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];
}
```

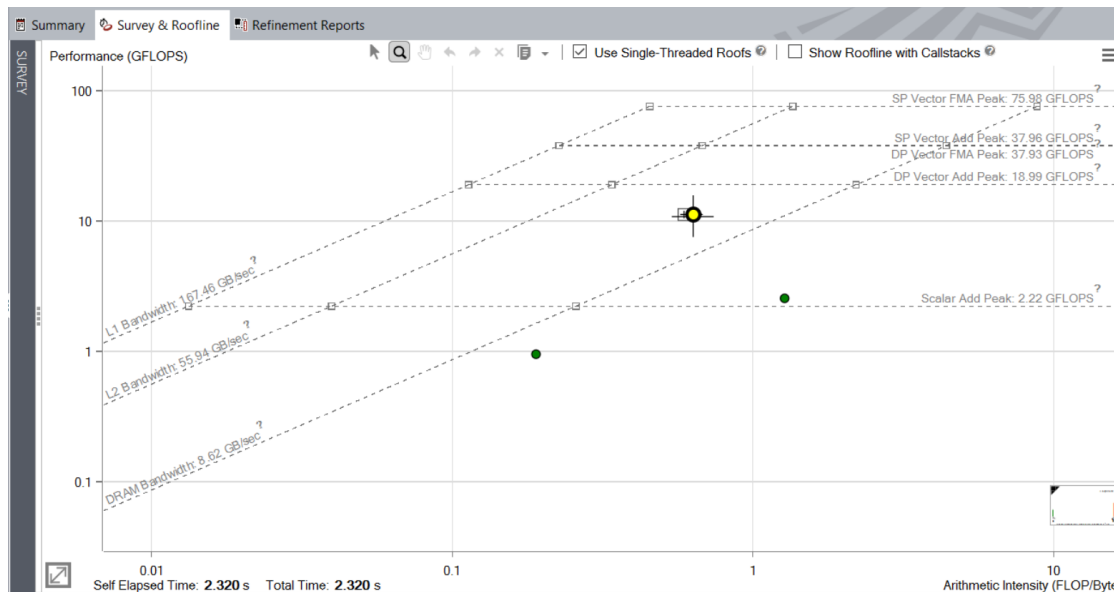
ISSUE: PROVEN (REAL) DEPENDENCY PRESENT

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's next?

Advisor Roofline – How much further can we go?

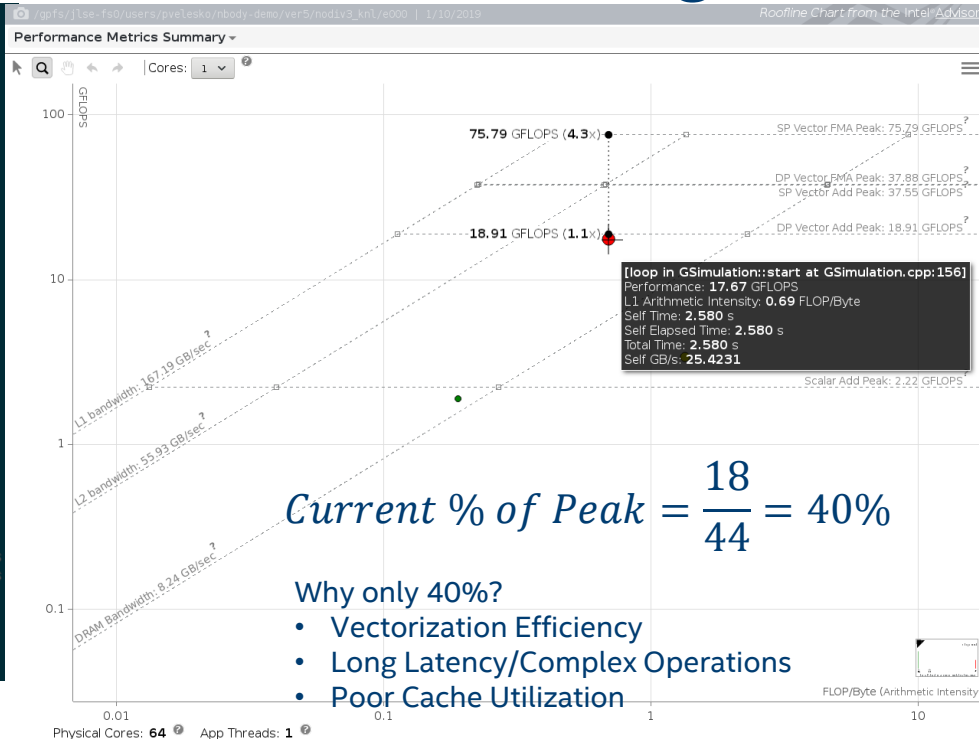
```

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;
}
    
```



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

Why only 40%?

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

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

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

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

Optimization Notice

Vectorization Efficiency?

Elapsed time: 5.19s Vectorized Not Vectorized FILTER: All Modules

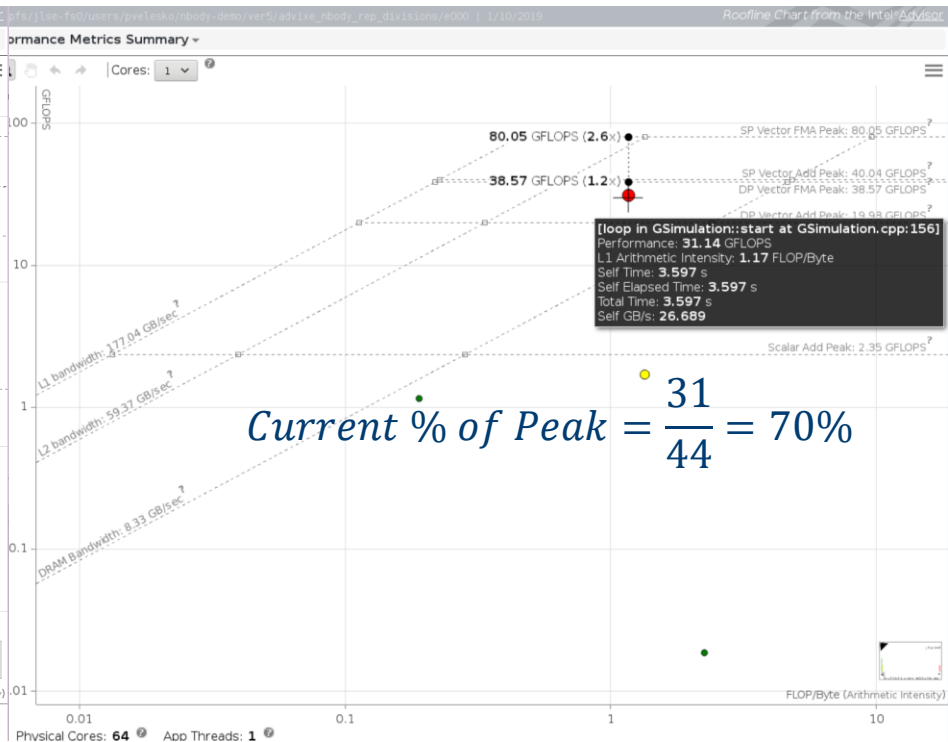
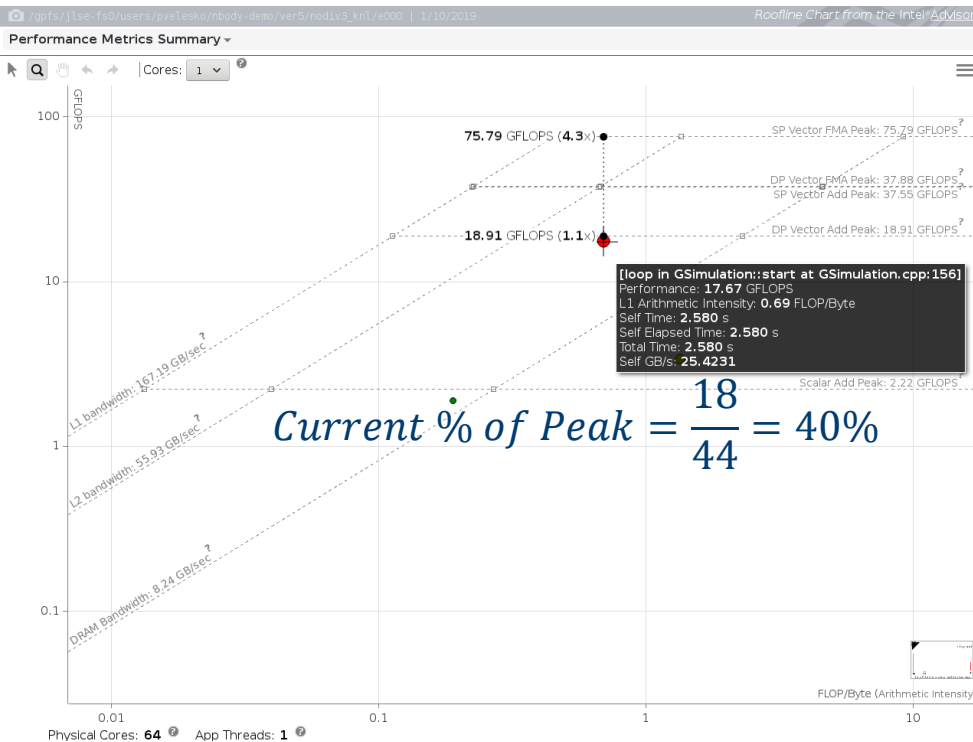
Summary Survey & Roofline Refinement Reports

ROOFLINE

Function Call Sites and Loops

Vectorized Loops				
Vec...	Efficiency	Gai...	VL (...)	
[loop in GSimulation::start at GSim	AVX.. ~97%	15. ..	16	
[loop in GSimulation::start at GSimulati				

Complex Operations?



Optimization Notice

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



Poor Cache Utilization?

INTEL[®] VTUNE[™] AMPLIFIER

Core-level hardware metrics

<https://www.alcf.anl.gov/user-guides/vtune-xc40>

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
```

```
module load vtune
```

```
vim Makefile # edit to add -dynamic
```

```
cp /soft/perfutils/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
```

Choose Analysis Type

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@jlselgin2

Command line:

```
/soft/compilers/intel/vtune_amplifier_2018.1.0.535340/bin64/ampixe-cl -collect hpc-performance -app-working-dir /usr/bin -- ls
```

Copy Close

Use -collect-with action

Hide knobs with default values

Start

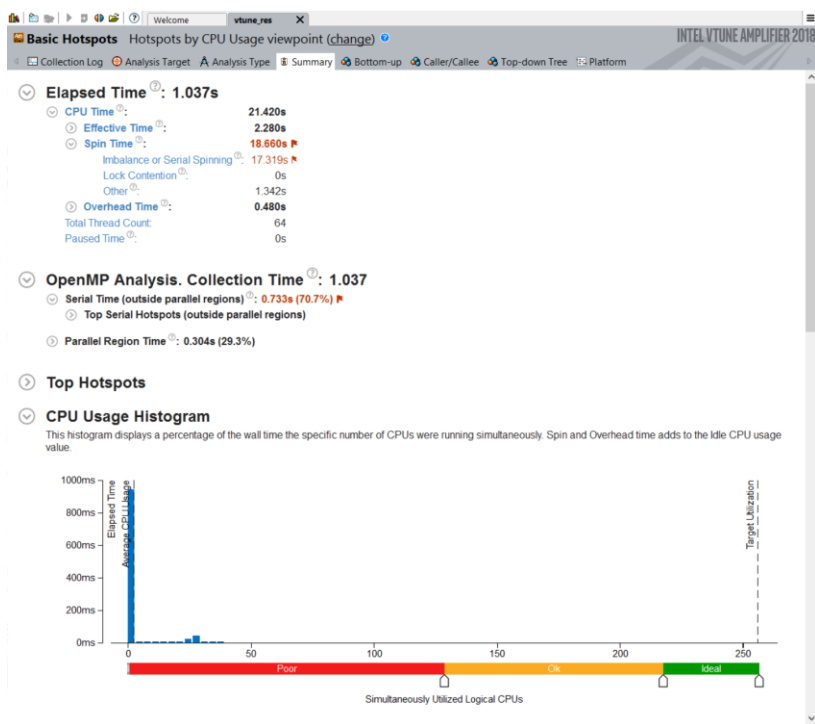
Start Paused

Choose Target

Command Line...

Hotspots analysis for nbody demo (ver7: threaded)

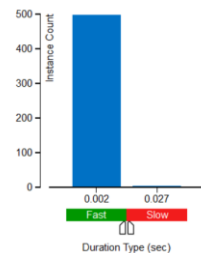
- `qsub ampxe.qsub ./your_exe ./inputs/inp`



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: startSompSparallel 64@unknown:146.182

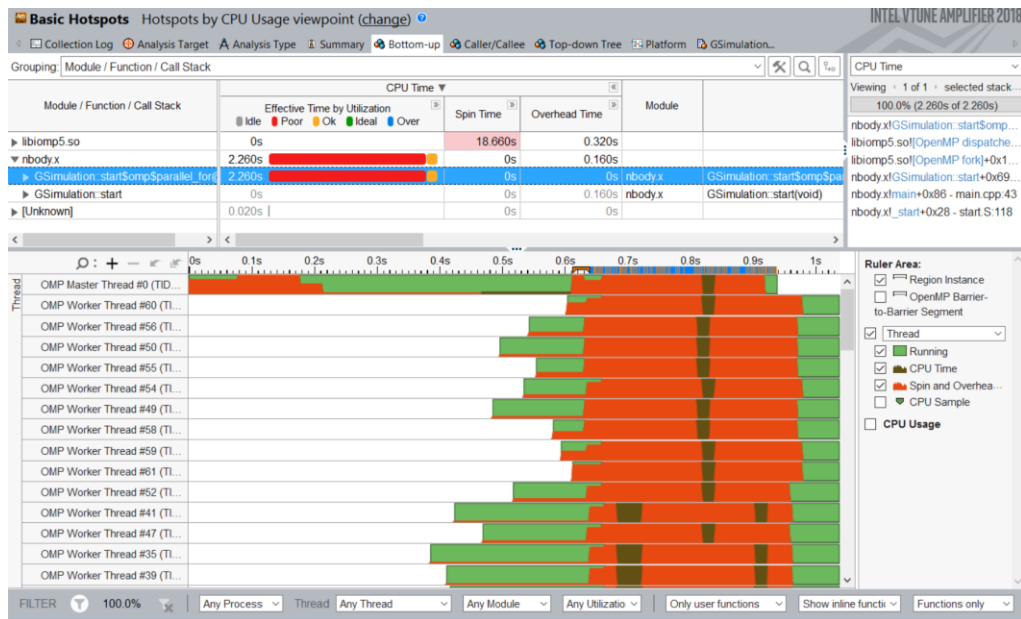


Lots of spin time indicate issues with load balance and synchronization

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 Hotspots view



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

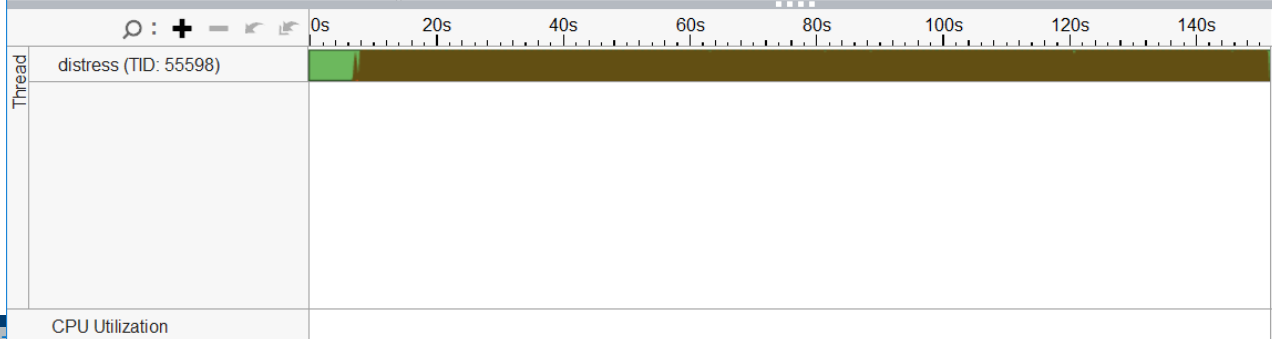
Double click on line to access source and assembly.

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

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

Grouping: Function / Call Stack

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_I9	1.688s	libimf.so	__libm_acos_I9		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
▶ csart	1.384s	libimf.so	csart		0x1d430



- Thread
 - Running
 - CPU Time
 - Spin and Overhead ...
 - CPU Sample
- CPU Utilization
 - CPU Time
 - Spin and Overhead ...



Hotspots

Hotspots by CPU Utilization

INTEL VTUNE AMPLIFIER 2019

Analysis Configuration

Collect Log

Summary

Bottom-up

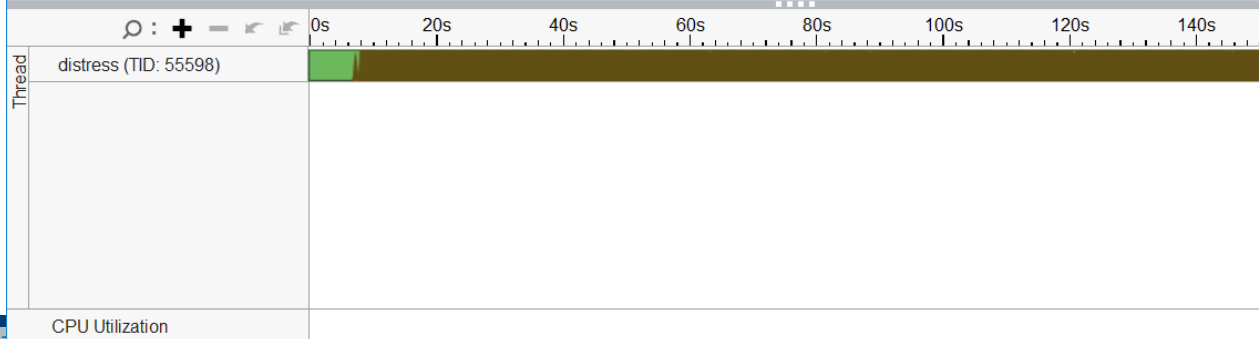
Caller/Callee

Top-down Tree

Platform

Grouping: Function / Call Stack

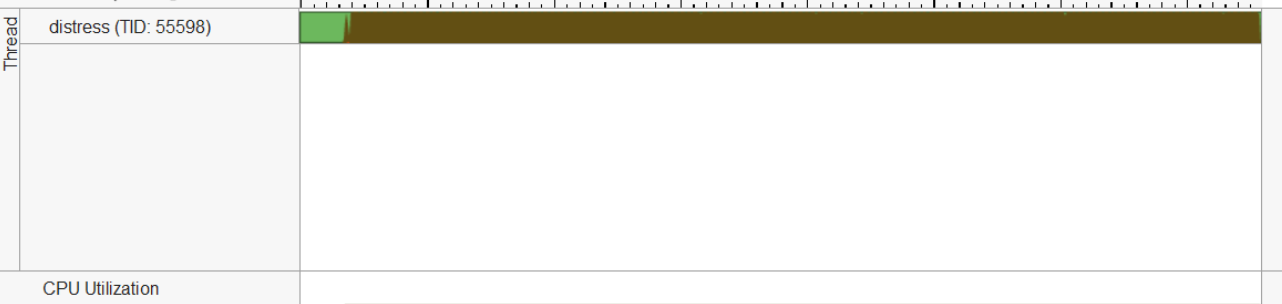
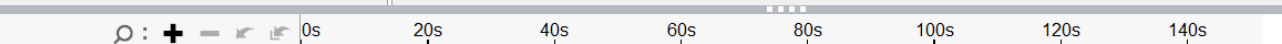
Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
▶ vdpowr_	13.1%	libmkl_intel_lp64.so	vdpowr_		0x695310
▶ aa	7.4%	distress	aa	aux.f90	0x41ec1c
▶ aa	6.8%	distress	aa	aux.f90	0x41ec9a
▶ invariants	6.4%	distress	invariants	aux.f90	0x41d550
▶ __libm_csqrt_ex	5.5%	libimf.so	__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
▶ csart	1.0%	libimf.so	csart		0x1d430



- Thread
- Running
 - CPU Time
 - Spin and Overhead ...
 - CPU Sample
- CPU Utilization
- CPU Time
 - Spin and Overhead ...

Grouping: Source Function / Function / Call Stack

Source Function / Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
▶ aa	14.2%	aa	aa	aux.f90	0
▶ vdpowr_	13.1%	vdpowr_	vdpowr_		0
▶ invariants	6.4%	invariants	invariants	aux.f90	0
▶ __libm_csqrt_ex	5.5%	__libm_csqrt_ex	__libm_csqrt_ex		0
▶ spinoru	5.5%	spinoru	spinoru	aux.f90	0
▶ ktjet	5.0%	ktjet	ktjet	analysis.f90	0
▶ __svml_log8_mask_b3	4.3%	__svml_log8_mask_b3	__svml_log8_mask_b3		0
▶ subqcd	3.2%	subqcd	subqcd	amplitudes.f90	0
▶ breit2lab	1.6%	breit2lab	breit2lab	PS.f90	0
▶ hamp_qlqlqb_1	1.4%	hamp_qlqlqb_1	hamp_qlqlqb_1	amplitudes.f90	0
▶ gettjet	1.3%	gettjet	gettjet	analysis.f90	0
▶ me0_qlqgg	1.3%	me0_qlqgg	me0_qlqgg	amplitudes.f90	0
▶ __libm_acos_l9	1.2%	__libm_acos_l9	__libm_acos_l9		0
▶ analyzejet	1.2%	analyzejet	analyzejet	analysis.f90	0
▶ hamp_alalaab 2	1.1%	hamp_alalaab 2	hamp_alalaab 2	amplitudes.f90	0



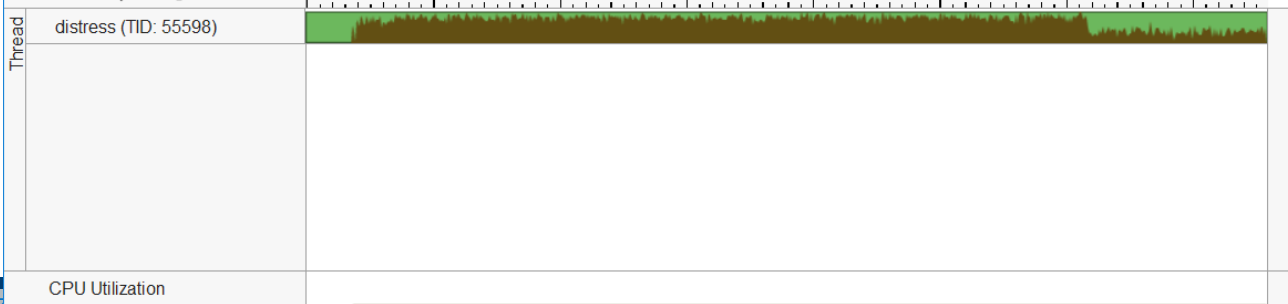
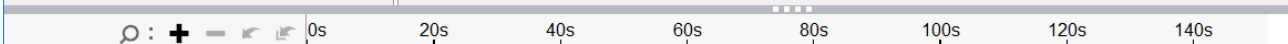
Thread

- Thread
- Running
- CPU Time
- Spin and Overhead ...
- CPU Sample
- CPU Utilization
 - CPU Time
 - Spin and Overhead ...



Grouping: Source Function / Function / Call Stack

Source Function / Function / Call Stack	CPU Time	Module	Function (Full)	Source File	Start Address
spinoru	27.5%	spinoru	spinoru	aux.f90	0
invariants	9.0%	invariants	invariants	aux.f90	0
getpdfs	8.3%	getpdfs	getpdfs	fitpdf.f90	0
ktjet	6.9%	ktjet	ktjet	analysis.f90	0
me0_qlqlgg	6.1%	me0_qlqlgg	me0_qlqlgg	amplitudes.f90	0
__svml_log8_mask_b3	5.9%	__svml_log8_mask_b3	__svml_log8_mask_b3		0
breit2lab	2.5%	breit2lab	breit2lab	PS.f90	0
dli2	2.4%	dli2	dli2	lis.f90	0
gettjet	1.8%	gettjet	gettjet	analysis.f90	0
analyzejet	1.6%	analyzejet	analyzejet	analysis.f90	0
me0_qlqlqcb_f3	1.6%	me0_qlqlqcb_f3	me0_qlqlqcb_f3	amplitudes.f90	0
ds_ql_s_nnlo_qcd_g	1.6%	ds_ql_s_nnlo_qcd_g	ds_ql_s_nnlo_qcd_g	sub.f90	0
me0_qlqlqcb_f4	1.3%	me0_qlqlqcb_f4	me0_qlqlqcb_f4	amplitudes.f90	0
ps4	1.3%	ps4	ps4	PS.f90	0
for costr	1.3%	for costr	for costr		0



Thread

- Running
- CPU Time
- Spin and Overhead ...
- CPU Sample

CPU Utilization

- CPU Time
- Spin and Overhead ...





Hotspots

Hotspots by CPU Utilization



INTEL VTUNE AMPLIFIER 2019

Analysis Configuration

Collection Log

Summary

Bottom-up

Caller/Callee

Top-down Tree

Platform

aux.f90 x

aux.f90 x

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 nnlobeamj]	19.3%		[Loop at line 141 in nnlobeamj]	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_f]	2.0%		[Loop at line 2055 in ds_ql_s_nnlo_qcd_f]	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_f]	1.8%		[Loop at line 1986 in ds_ql_s_nnlo_qcd_f]	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
[Loop at line 1812 in ds_ql_s_nnlo_qcd_g]	1.7%		[Loop at line 1812 in ds_ql_s_nnlo_qcd_g]	sub.f90	0

0s 20s 40s 60s 80s 100s 120s 140s

Thread

distress (TID: 55598)

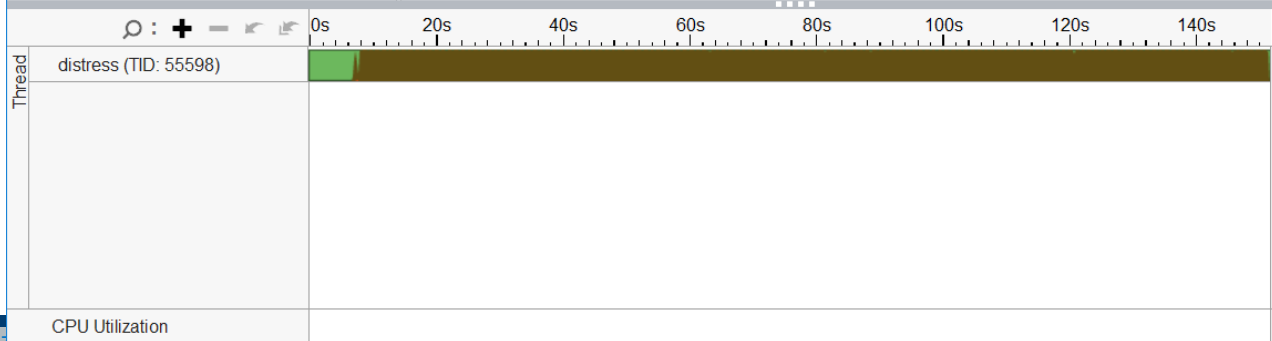
CPU Utilization

 Thread Running CPU Time Spin and Overhead ... CPU Sample CPU Utilization CPU Time Spin and Overhead ...



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 spinoru]	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 LHAPD:...	stl_algo.h	0x669c9
▶ [Loop at line 1169 in LHAPD]	0.1%	0.036s	libLHAPDF.so	[Loop at line 1169 in LHAPD:...	stl_tree.h	0x66960
▶ [Loop at line 139 in nnlobeamj]	19.1%	0s	distress	[Loop at line 139 in nnlobeamj]	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



- Thread
 - Running
 - CPU Time
 - Spin and Overhead ...
 - CPU Sample
- CPU Utilization
 - CPU Time
 - Spin and Overhead ...



Hotspots

Hotspots by CPU Utilization

INTEL VTUNE AMPLIFIER 2019

Analysis Configur

HPC Performance Characterization

Bottom-up

Caller/Callee

Top-down Tree

Platform

aux.f90 x

aux.f90 x

Grouping: Call Sta

Hotspots by CPU Utilization

Function	CPU Time: Self	Module	Function (Full)	Source File	Start Address
Total	100.0%	0s			
▼ [Outside any loop]	99.9%	0.020s			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 spinoru]	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 LHAPD:...	stl_algo.h 0x669c9
▶ [Loop at line 1169 in LHAPD]	0.1%	0.036s	libLHAPDF.so	[Loop at line 1169 in LHAPD:...	stl_tree.h 0x66960
▶ [Loop at line 139 in nnlobeamj]	19.1%	0s	distress	[Loop at line 139 in nnlobeamj]	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 cl...]	3.8%	4.494s	distress	[Loop at line 2750 in dxsec cl...]	xsec.f90 0x49d2b2

0s 20s 40s 60s 80s 100s 120s 140s

Thread

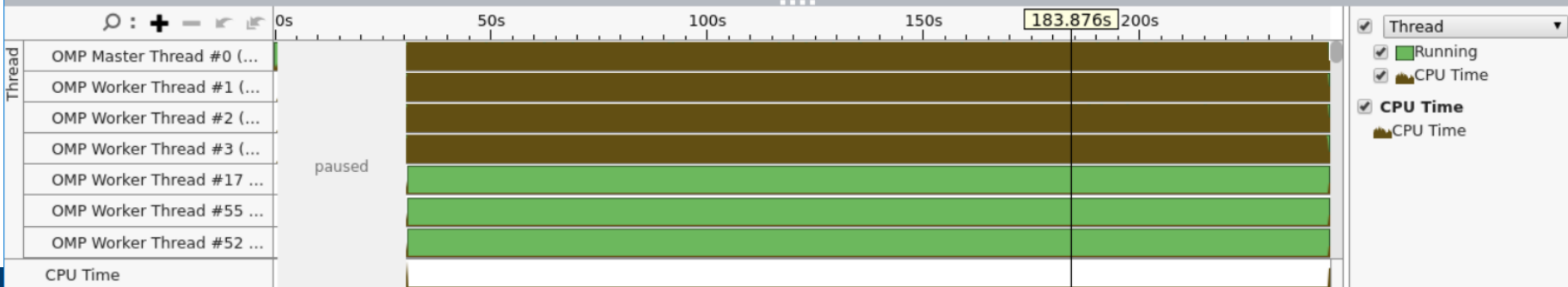
distress (TID: 55598)

CPU Utilization

- Thread
- Running
- CPU Time
- Spin and Overhead ...
- CPU Sample
- CPU Utilization
- CPU Time
- Spin and Overhead ...

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound					
					Memory Latency				Mer	
					L1 Hit Rate	L2 Hit Rate	L2 Hit Bound	L2 Miss Bound	UTLB Overhead	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_folc	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`

Poor Cache Utilization?

Using Vtune to

Am General Exploration Microarchitecture

Analysis Configuration Collection Log Summary

Grouping: Function / Call Stack

Function / Call Stack

GSimulation::start

apic_timer_interrupt

native_write_msr_safe

Grouping: Function / Call Stack

Function / Call Stack

GSimulation::start

lapic_next_deadline

```

ampLxe: Using report path 'j:\gpfis\j\se-fs0\users\pvellesko\nbody-demo\ver5\ampLxe_knl_nodiv_60k'
ampLxe: Executing actions 75 % Generating a report
Clockticks: 405,992,000.000
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 Speculators: 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
Dividers: 0.0% of Clockticks
MS Assists: 0.1% of Clockticks
FP Assists: 0.0% of Clockticks
Total Thread Count: 1
    
```

s

Front-End Bound	Back-End Bound	Retiring
0.1%	41.3%	58.6%
0.0%	46.7%	0.0%
0.0%	60.0%	0.0%

Memory Latency		
L2 Hit Bound	L2 Miss Bound	UTLB Overhead
0.9%	0.0%	0.0%
n n%	n n%	n n%

Optimization Notice

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



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;
}
```

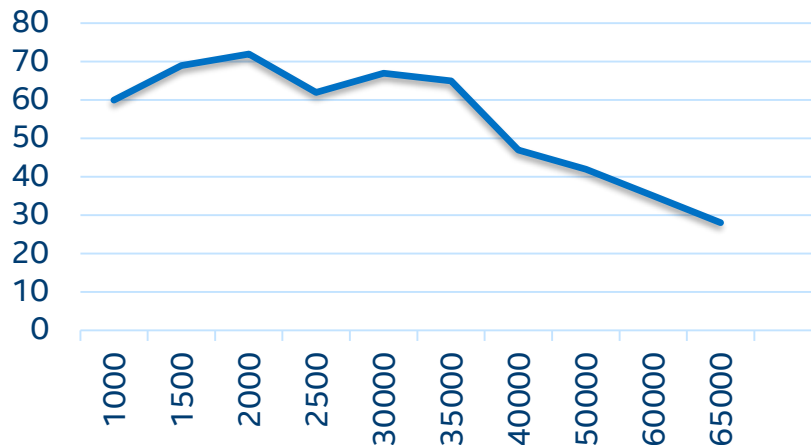
Maximum N before we lose caching?

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

$32\text{k} / (4 * 4) = 2\text{k}$ (L1)

$1\text{MB} / (7 * 4) = 35.7\text{k}$ (L2)

GFLOPs vs N



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:

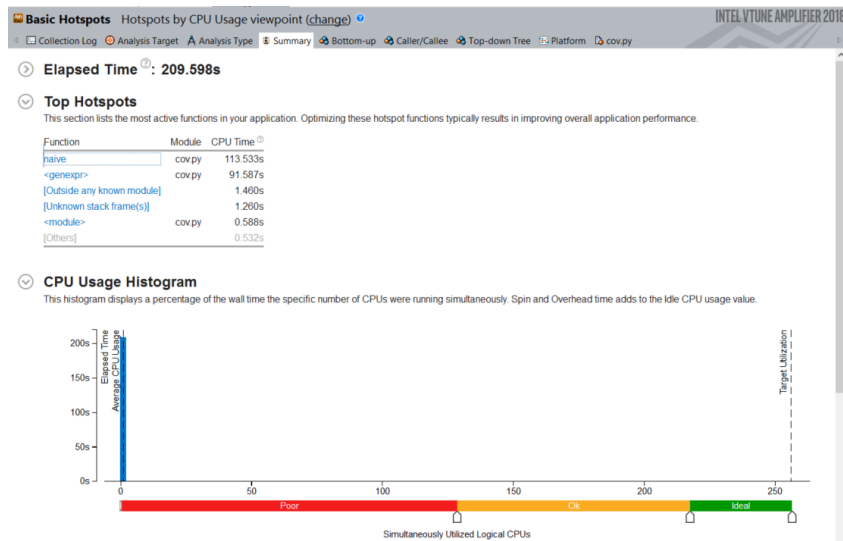
- The “application” should be the full path to the python interpreter used
- The python code should be passed as “arguments” to the “application”

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



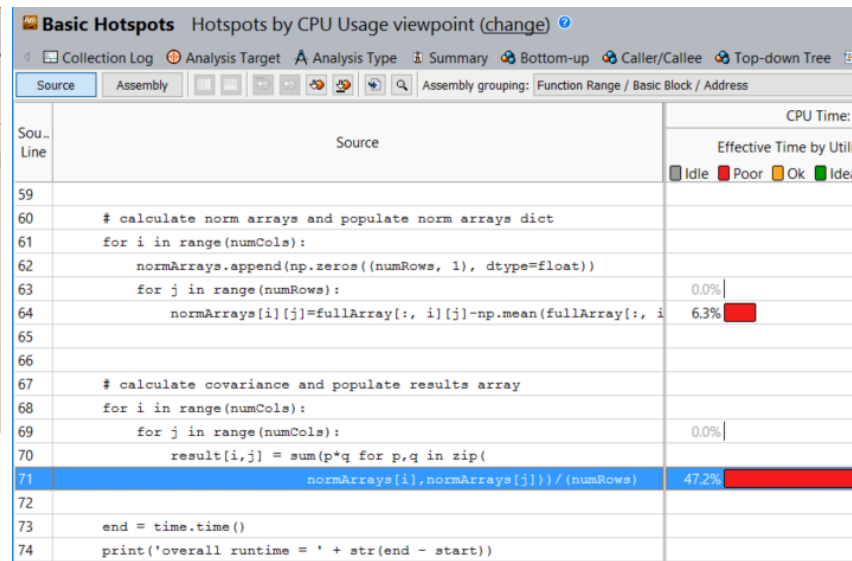
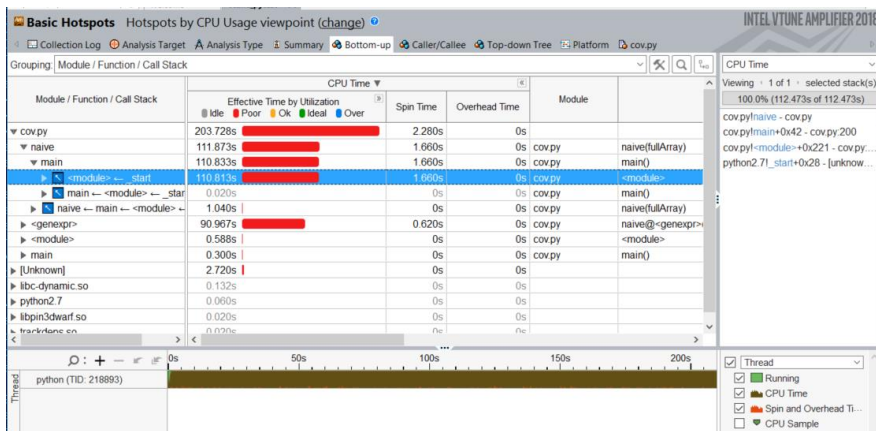
Naïve implementation of the calculation of a covariance matrix

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

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
```

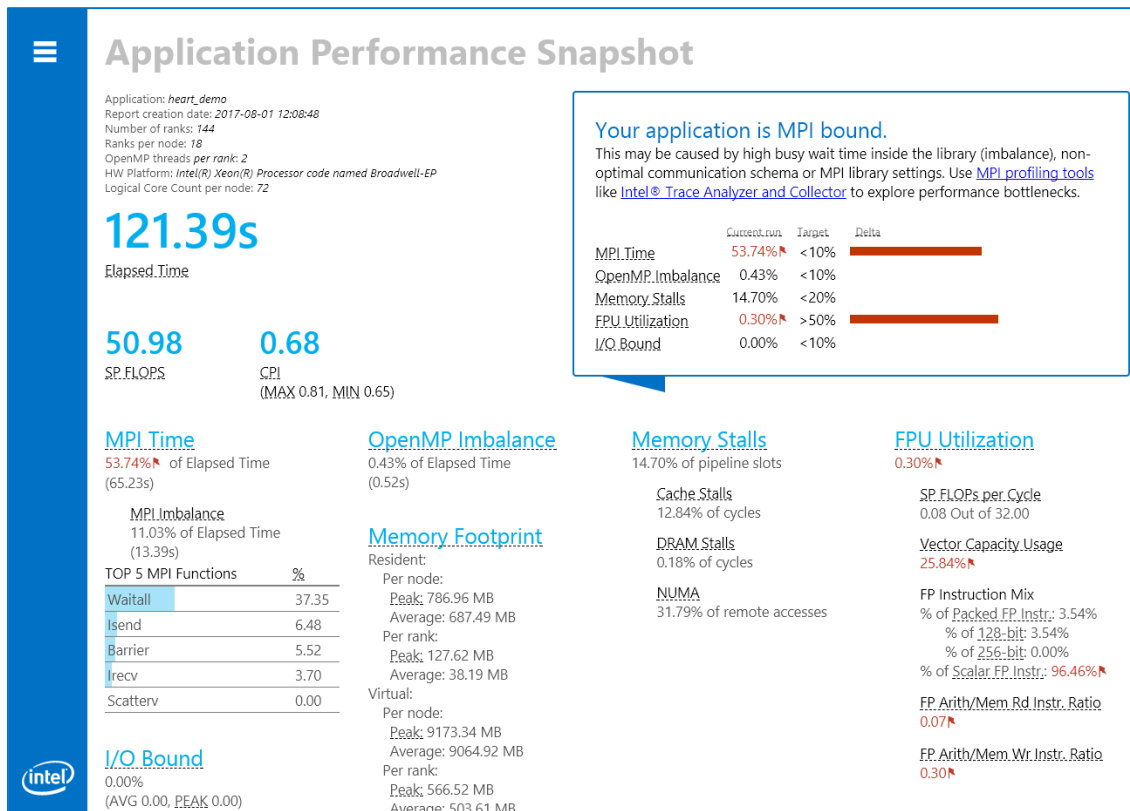
Launch your job in interactive or batch mode:

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

Produce text and html reports:

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

APS HTML Report



COMMON ISSUES

Fixes

No call stack information/unknown stack frame

- Check finalization log
 - Make sure Vtune finds your binary along with libraries that you call

Incompatible database scheme when trying to open result in GUI

- Make sure your local Vtune is the same version or newer

Vtune sampling driver.. using perf or errors mentioning PMU Resources

- Notify support@alcf.anl.gov or me pvelesko@anl.gov

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>

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

BACKUP

VTune Cheat Sheet

Compile with `-g -dynamic`

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

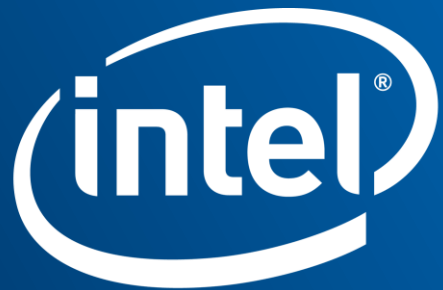
- `--result-dir=./vtune_output_dir`
- `--search-dir src:=../src --search-dir bin:=./`
- `-knob enable-stack-collection=true -knob collect-memory-bandwidth=false`
- `-knob analyze-openmp=true`
- `-finalization-mode=deferred` if finalization is taking too long on KNL
- `-data-limit=125` ← in mb
- `-trace-mpi`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

Advisor Cheat Sheet

Compile with `-g -dynamic`

```
advixe-cl -c roofline/dependencies/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`



Software