Software

INTEL® VTUNE™ AMPLIFIER WORKSHOP

Agenda

Getting Setup

Intel® VTune™ Amplifier Introduction

- matrix sample

Intel® Optane™ DC Persistent Memory Profiling

- PMDK Sample

Platform Profiler

- Server Sample
- Workloads (not included)

Setup

1. License:

https://registrationcenter.intel.com/en/forms/?productid=3218

2. Install VTune Amplifier (USB or Download Installer)

3. Copy “Results” folder from USB to laptop

4. Pass USB to neighbor

INTEL® VTUNE™ AMPLIFIER INTRO

Faster, Scalable Code, Faster

Intel® VTune™ Amplifier Performance Profiler

Accurate Data - Low Overhead

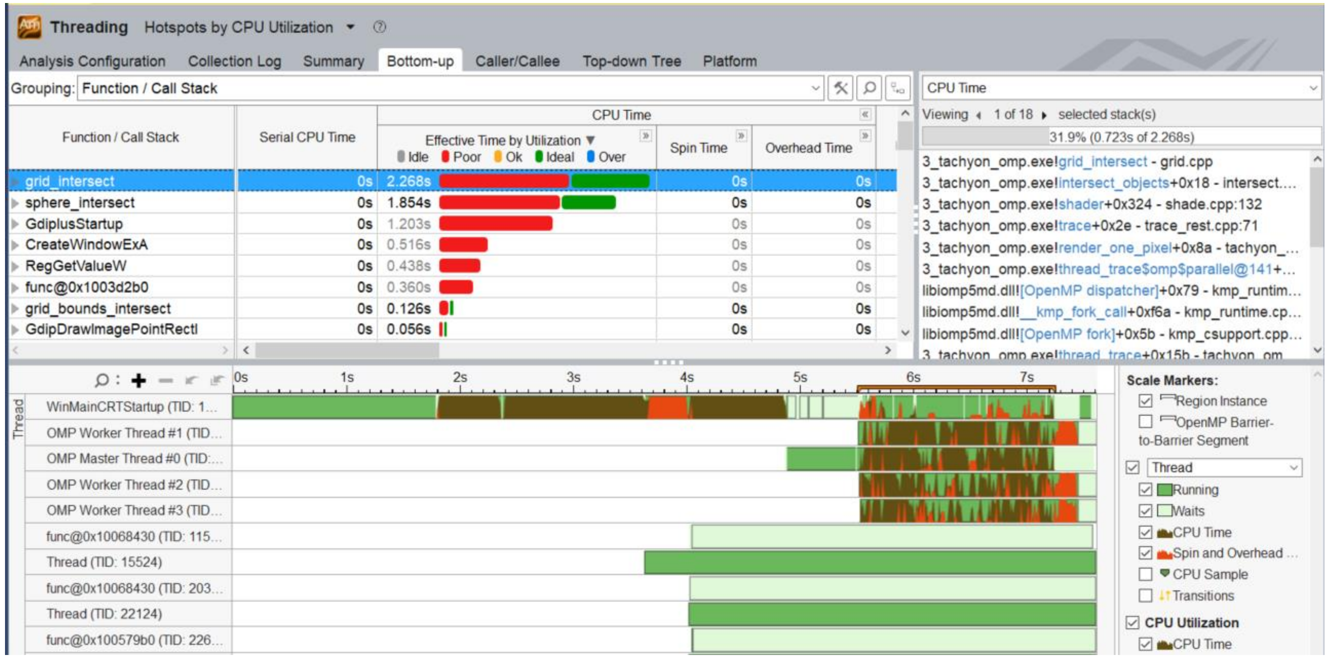
- CPU, GPU, FPU, threading, bandwidth…

Meaningful Analysis

- Threading, OpenMP region efficiency
- Memory access, storage device

Easy

- Data displayed on the source code
- Easy set-up, no special compiles



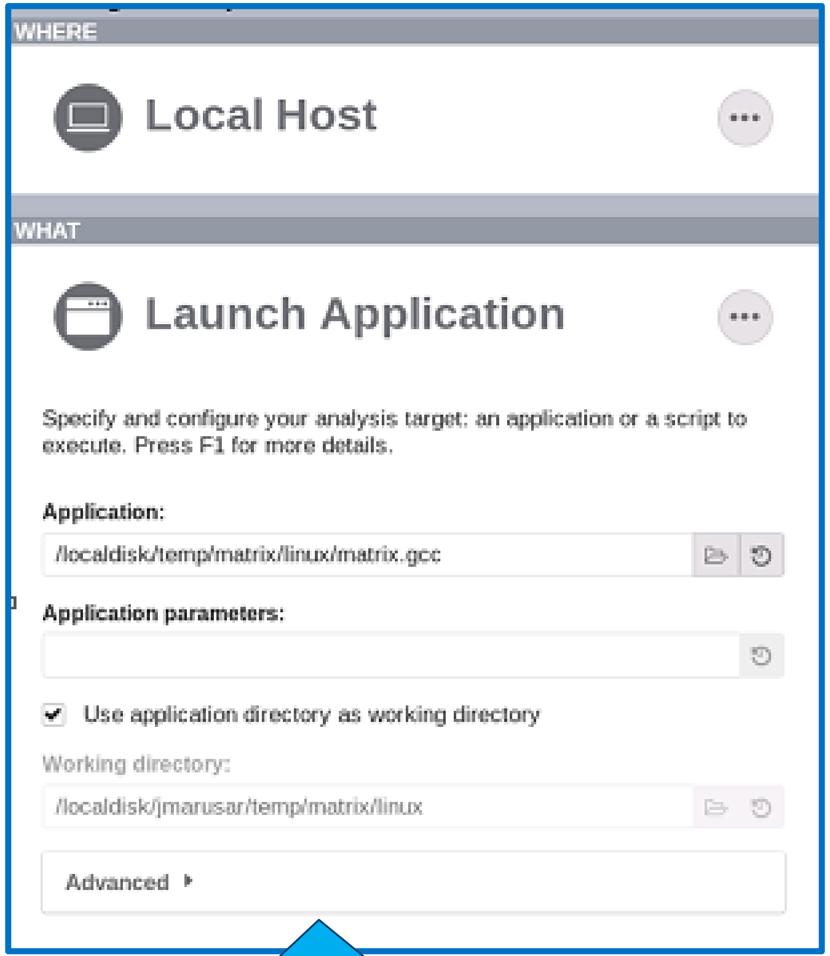
"Last week, Intel® VTune™ Amplifier helped us find almost 3X performance improvement. This week it helped us improve the performance another 3X."

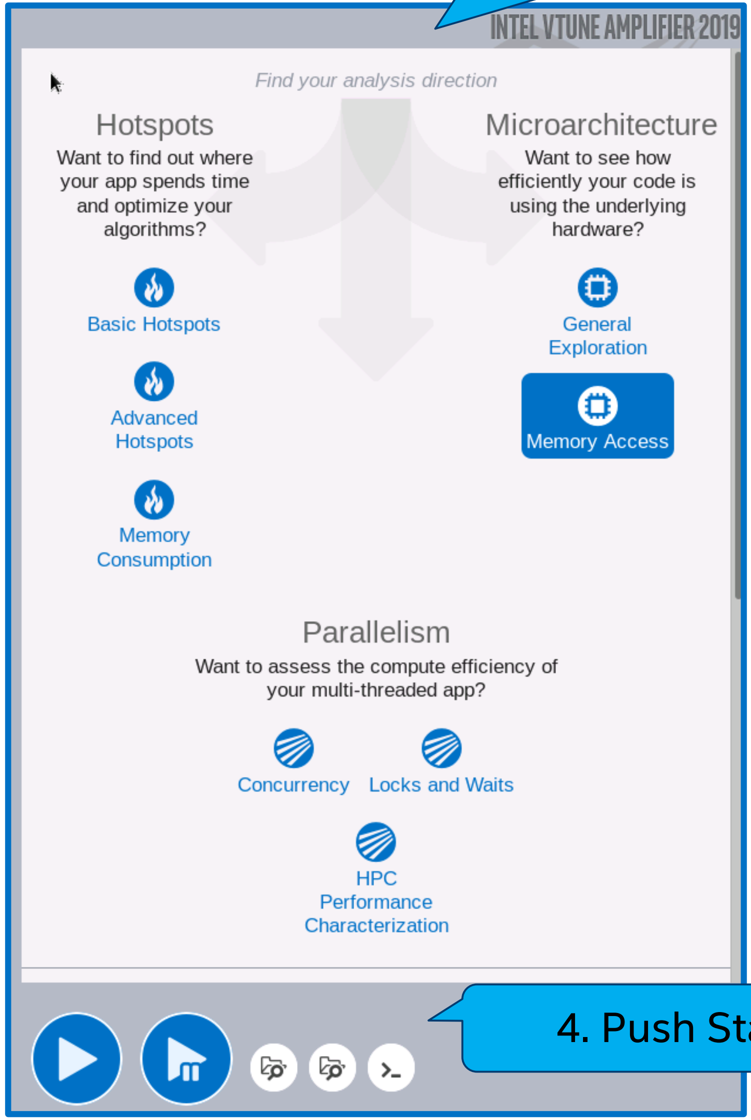
Claire Cates
Principal Developer
SAS Institute Inc.

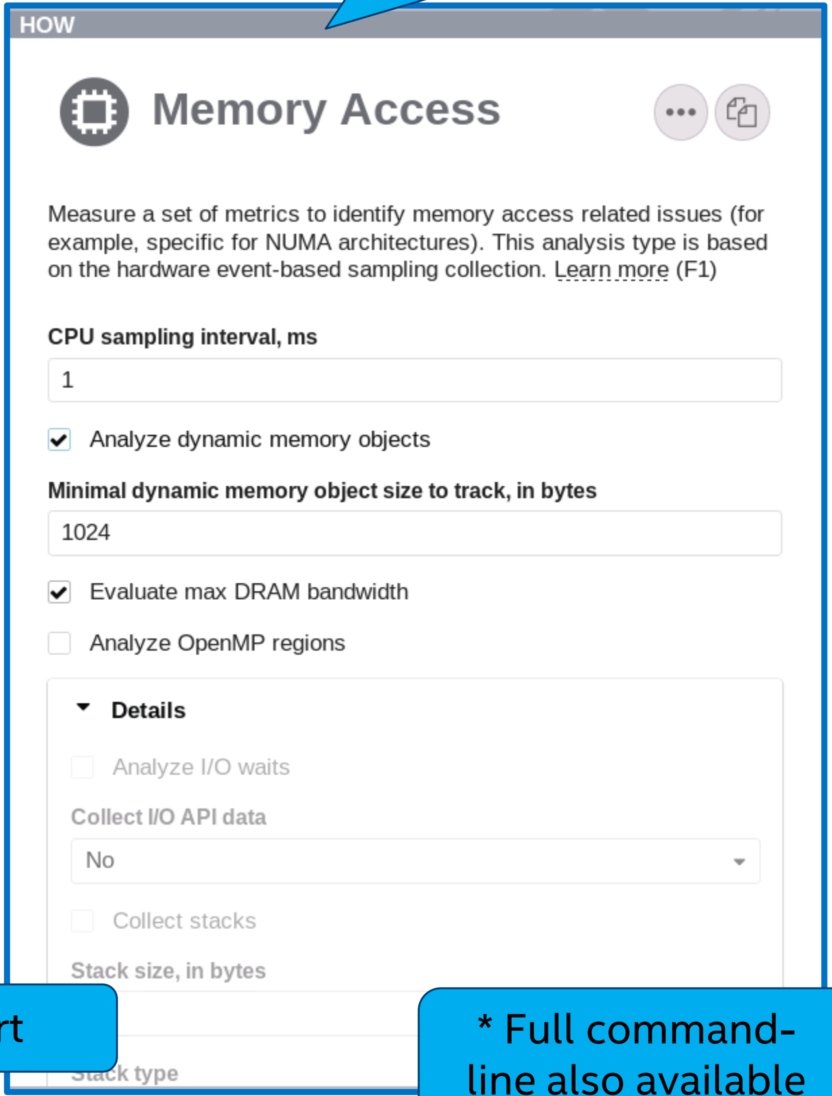
Setting up a profile is easy

2. Choose Analysis Type

3. Collection options







1. What/where to profile

4. Push Start

* Full command-line also available

Two Great Ways to Collect Data

Intel® VTune™ Amplifier

Software Collector	Hardware Collector
Uses OS interrupts	Uses the on chip Performance Monitoring Unit (PMU)
Collects from a single process tree	Collect system wide or from a single process tree.
~10ms default resolution	~1ms default resolution (finer granularity - finds small functions)
Either an Intel® or a compatible processor	Requires a genuine Intel® processor for collection
Call stacks show calling sequence	Optionally collect call stacks
Works in virtual environments	Works in a VM only when supported by the VM (e.g., vSphere*, KVM)
No driver required	Requires a driver - Easy to install on Windows - Linux requires root (or use default perf driver)

No special recompiles - C, C++, C#, Fortran, Java, Assembly

Example: Hotspots Analysis

Summary View

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up

Elapsed Time: 5.554s

CPU Time:	10.504s
Instructions Retired:	21,698,000,000
CPI Rate:	1.257
CPU Frequency Ratio:	1.041
Total Thread Count:	9
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
grid_intersect	3_tachyon_omp.exe	5.539s
sphere_intersect	3_tachyon_omp.exe	3.247s
func@0x1002e59d	libiomp5md.dll	0.148s
shader	3_tachyon_omp.exe	0.117s
KeDelayExecutionThread	ntoskrnl.exe	0.091s
[Others]	N/A*	1.361s

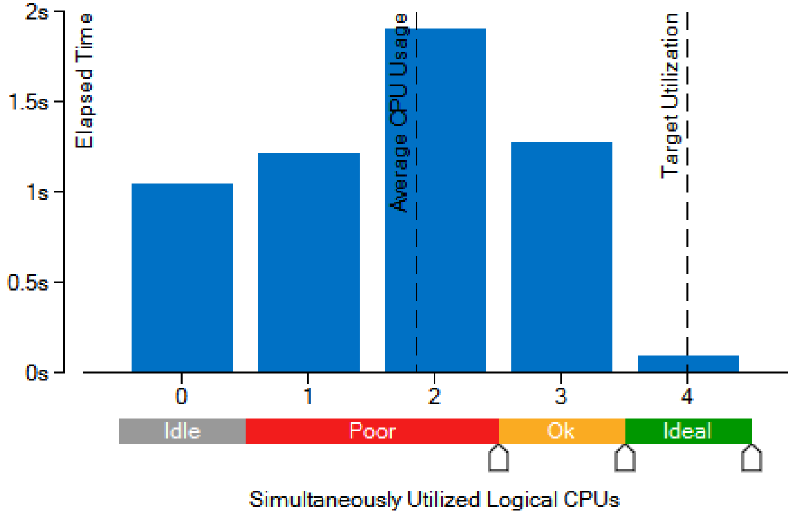
*N/A is applied to non-summable metrics.

Average Bandwidth

Package	Total, GB/sec	Read, GB/sec	Write, GB/sec
package_0	5.715	3.504	2.212

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

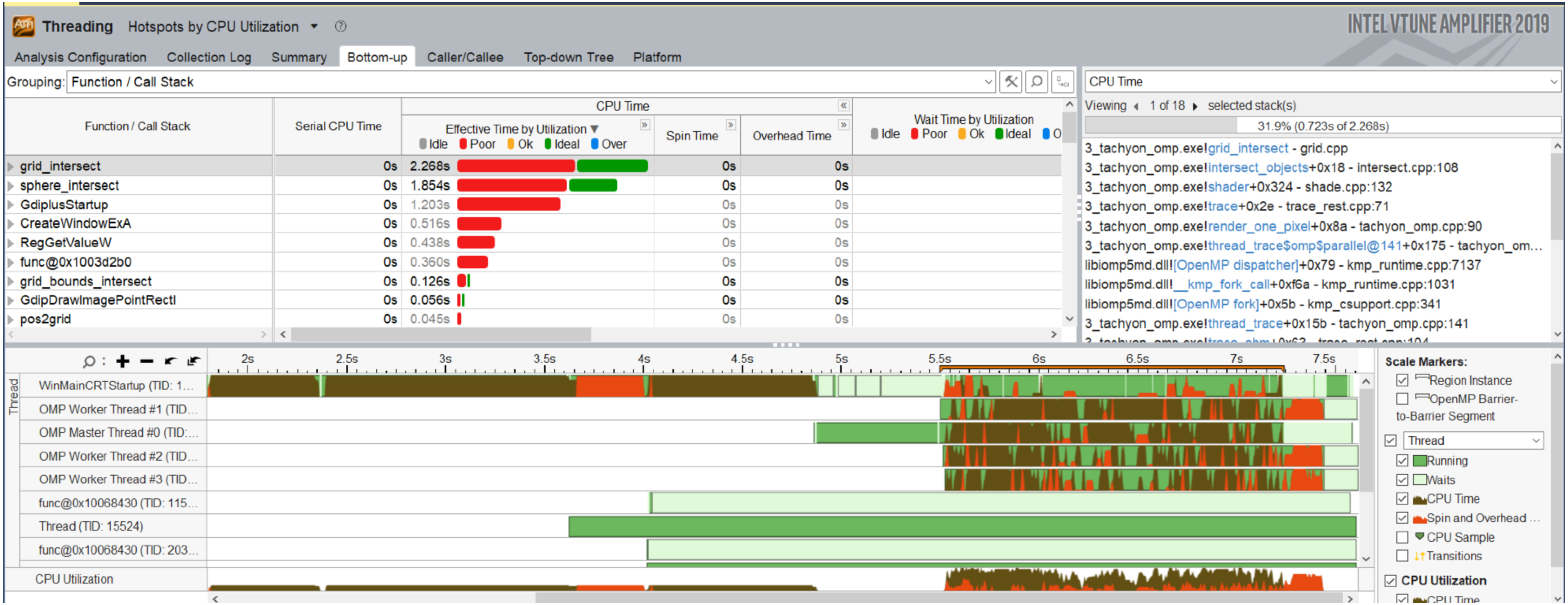


Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Example: Threading Analysis

Bottom-up View



Identifying and Diagnosing Inefficiency

Microarchitecture Analysis

```
> amplxe-cl -collect uarch-exploration -- ./myapp.out
```

- Microarchitecture Exploration (previously General Exploration) is a hardware events analysis. It is preconfigured to sample the appropriate events on your architecture and calculates the proper metrics from them.
- Potential tuning opportunities are highlighted in pink.
- To check the efficiency of a hotspot, look at the Retiring metric. If it's less than the expected number for your application type, it's probably inefficient.
  - Hotspots with high retiring values may still have room for improvement.

App Type	Expected
Client/ Desktop	20-50%
Server/ Database/ Distributed	10-30%
HPC	30-70%

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Event Count | Platform

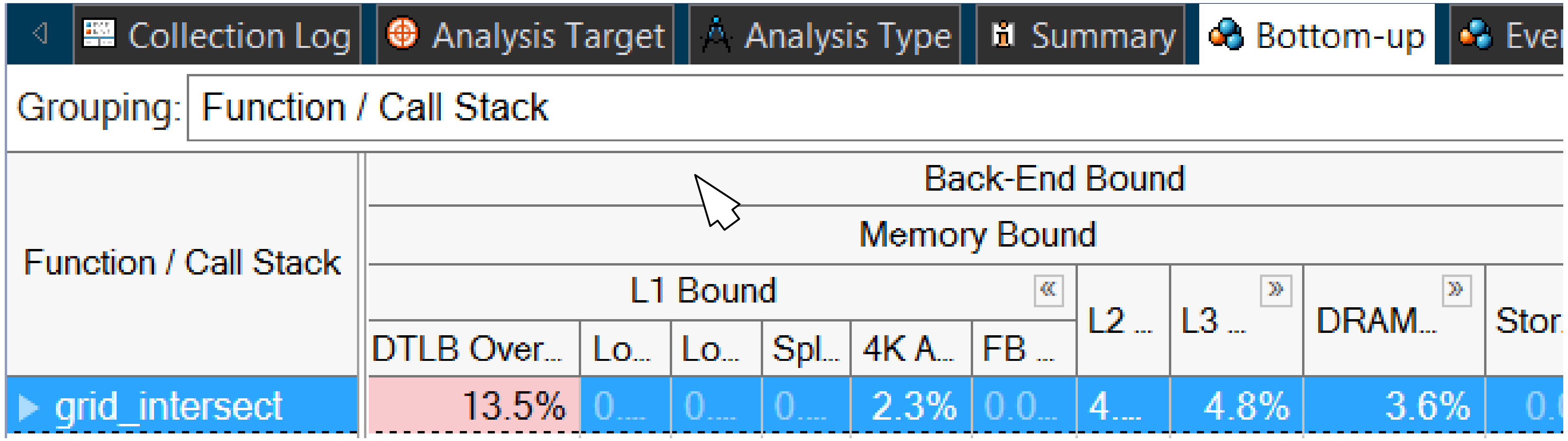
Grouping: Function / Call Stack

Function / Call Stack	Instructions Retired	CPI Rate	Front-End Bound	Bad Speculation	Back-End Bound	Retiring
initialize_2D_buffer	85,219,200,000	0.266	0.5%	0.0%	0.0%	100.0%
grid_intersect	10,963,200,000	0.706	4.6%	15.1%	46.4%	33.9%
sphere_intersect	10,946,400,000	0.601	2.1%	1.6%	47.5%	48.8%
grid_bounds_intersect	480,000,000	1.105	13.0%	2.2%	52.3%	32.5%
tri_intersect	216,000,000	0.789	0.0%	20.2%	39.3%	40.5%

Categorizing and Correcting Inefficiencies

Microarchitecture Exploration Analysis

- Intel® VTune™ Amplifier has hierarchical expanding metrics categorized by the four slot types.
- You can expand your way down, following the hotspot, to identify the root cause of the inefficiency.
  - Sub-metrics highlight pink on their own merits, just like top level metrics.
- Hovering over a metric produces a helpful, detailed tooltip (not shown).
  - There are tooltips on Summary tabs too: hover over any ⓘ icon.



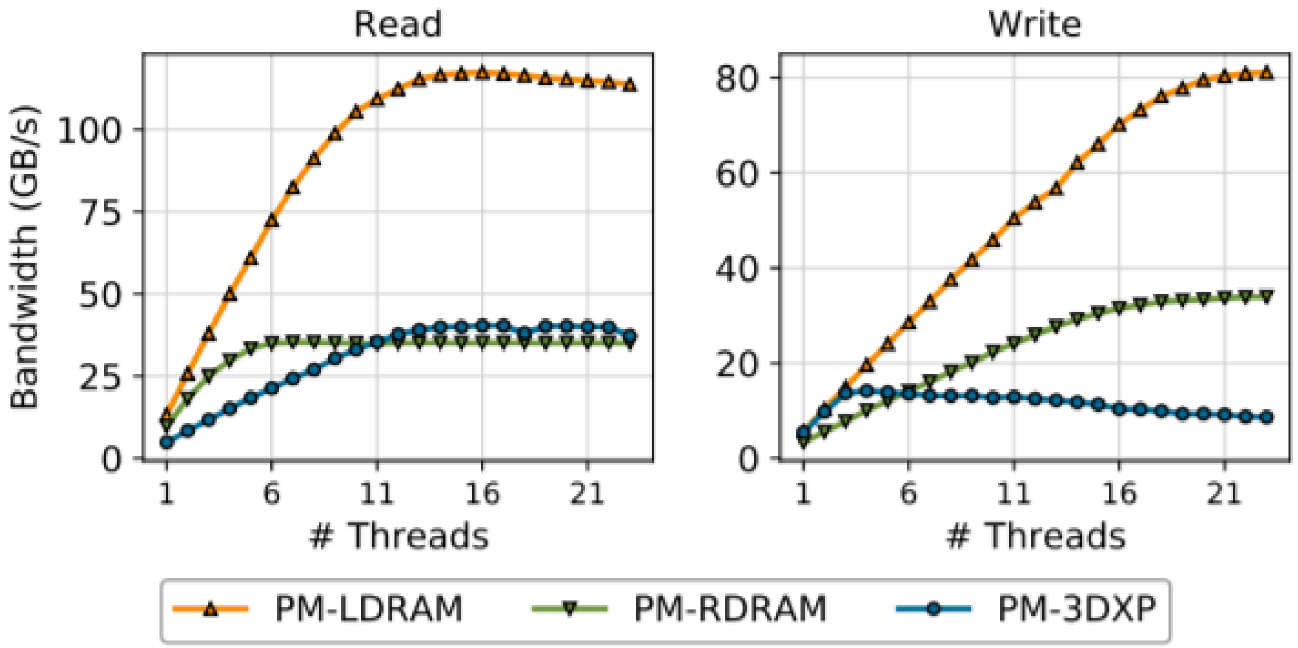
Matrix Sample

matrix\vc14\VTune Amplifier Results\matrix\matrix.amplxeproj

PERSISTENT MEMORY BANDWIDTH USE CASE STUDY

Goal

We now know performance characteristics for Intel® Optane™ DC persistent memory DIMMs



Would like to write memory benchmark achieving max persistent memory bandwidth as close to the limit as possible

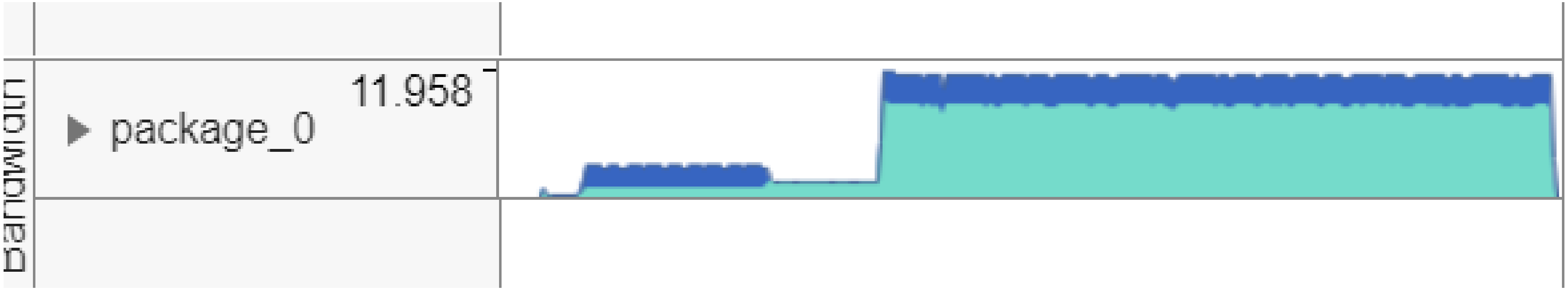
Algorithm

Use triad kernel similar to the one used is well-known stream benchmark

Original code:

```
for (j = 0; j < REPEATS; j++)
{
    #pragma omp parallel for
    for (i = 0; i < size; i++)
    {
        D_RW(c)[i] = multiplier * D_RO(a)[i] + D_RO(b)[i];
    }
}
```

Initial VTune Amplifier Results (ma_orig)



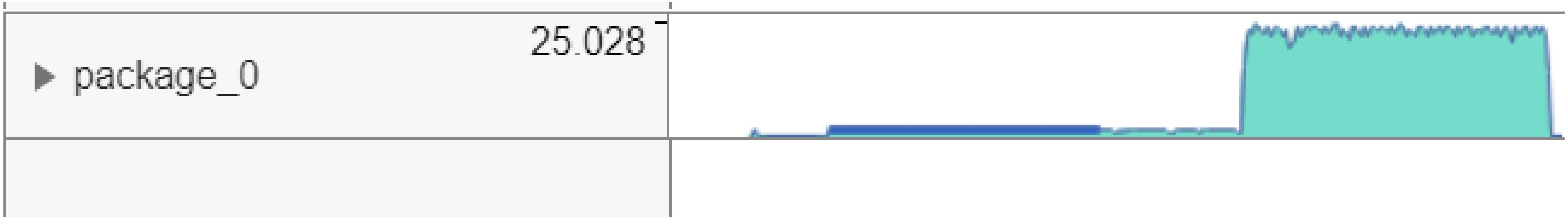
- Bandwidth peaks at about 12 GB/s. Much lower than expected
- Recall that write bandwidth is much lower for persistent memory. Could be write-limited performance.
- Let's try to avoid writing to persistent memory

Read-only Persistent Memory

Allocated array ‘c’ in DRAM instead of persistent memory (i.e. use regular malloc instead of PMDK API for it)

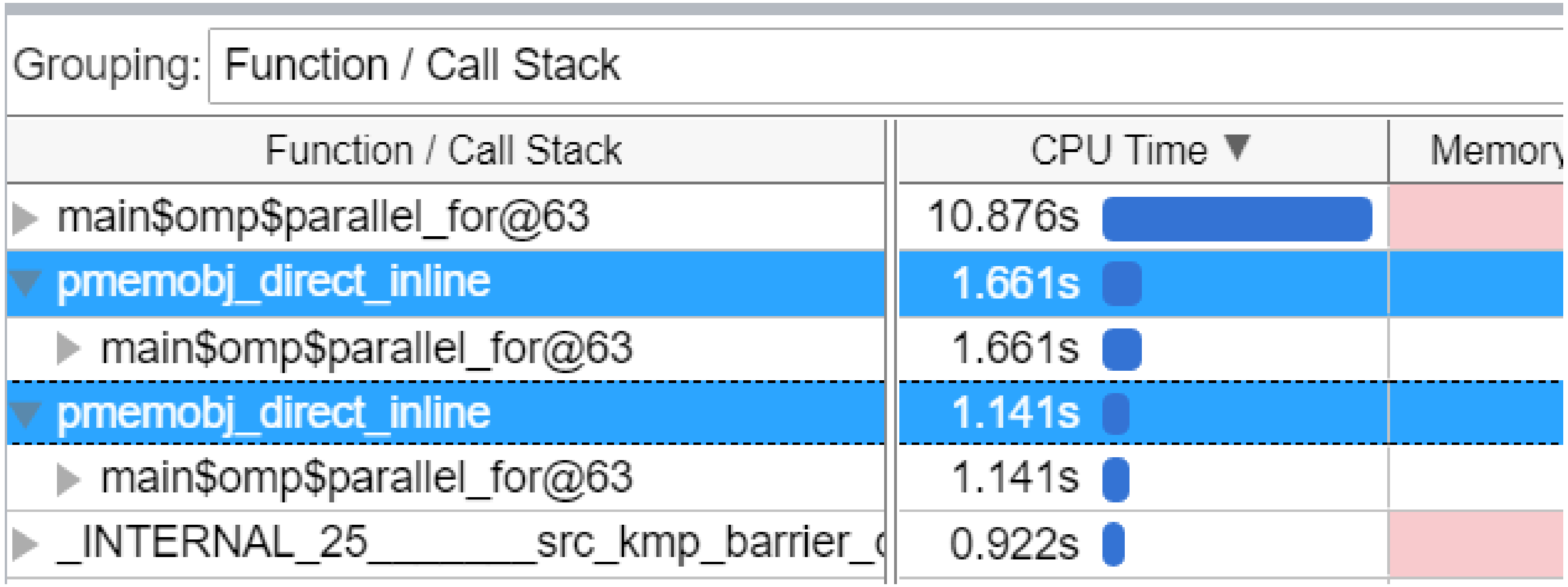
```
for (j = 0; j < REPEATS; j++)
{
    #pragma omp parallel for
    for (i = 0; i < size; i++)
    {
        c[i] = multiplier * D_RO(a)[i] + D_RO(b)[i];
    }
}
```

VTune Amplifier result for read-only persistent memory (ma_read_only)



- A significant improvement – bandwidth now peaks at 25 GB/s
- Now let's examine the code more precisely in VTune Amplifier

VTune Amplifier result for read-only persistent memory



Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Memory
mainompparallel_for@63	10.876s	
pmemobj_direct_inline	1.661s	
mainompparallel_for@63	1.661s	
pmemobj_direct_inline	1.141s	
mainompparallel_for@63	1.141s	
_INTERNAL_25________src_kmp_barrier_c	0.922s	

We see some ‘pmemobj_direct_inline’ functions called from our main loop taking some time. What are these?

VTune Amplifier result for read-only persistent memory

Source	
`for (j = 0; j < REPEATS; j++)`	
`{`	
`#pragma omp parallel for`	
`for (i = 0; i < size; i++)`	0.67
`{`	0.00
`c[i] = multiplier * D_RO(a)[i] + D_RO(b)[`	10.20
`}`	
`}`	

Address	Source Line	Assembly	
0x40180c		**Block 23:**	
0x40180c	66	xor r9d, r9d	
0x40180f		**Block 24:**	
0x40180f	66	mov edi, dword ptr [r15+r13*4]	0
0x401813	66	imul edi, dword ptr [rsp+0x30]	2
0x401818	66	add edi, dword ptr [r9+r13*4]	0
0x40181c	66	mov dword ptr [rbx+r13*4], edi	6
0x401820	64	inc r13	0
0x401823	65	cmp r13, qword ptr [rsp+0x8]	0
0x401828	65	jle 0x40172d <Block 6>	

- The functions are from the D_RO macro
- This prevents compilers from vectorizing the code as can be seen in the assembly

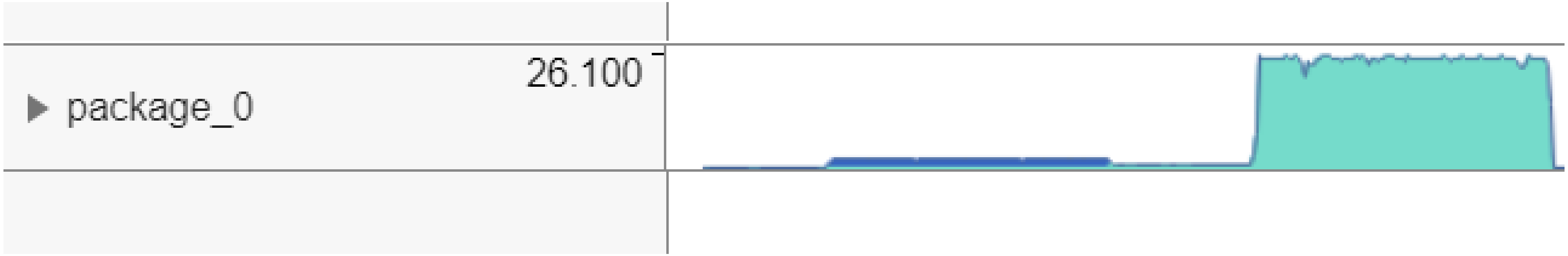
Move D_RO out of the loop

```
const int* _a = D_RO(a);
const int* _b = D_RO(b);

for (j = 0; j < REPEATS; j++)
{
    #pragma omp parallel for
    for (i = 0; i < size; i++)
    {
        c[i] = multiplier * _a[i] + _b[i];
    }
}
```

- Did a simple modification by moving D_RO out of the main loop
- Let's run VTune and see what this changed

VTune Amplifier result w/o D_RO (ma_vect_default)



- Peak bandwidth grew a little bit to 26 GB/s
- Let's examine the code

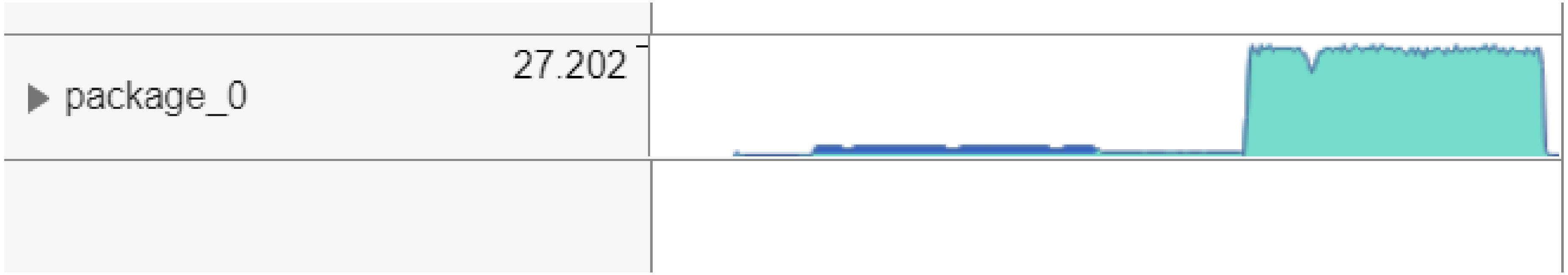
VTune Amplifier result w/o D_RO (ma_vect_default)

`for (j = 0; j < REPEATS; j++)`	
`{`	
`    #pragma omp parallel for`	0.171
`    for (i = 0; i < size; i++)`	
`    {`	
`        c[i] = multiplier * _a[i] + _b[i];`	13.391
`    }`	
`}`	

0x4018c3	68	psrlq xmm1, 0x20	
0x4018c8		**Block 17:**	
0x4018c8	68	movdqu xmm3, xmmword ptr [r10-	0.036s
0x4018ce	68	movdqa xmm4, xmm2	5.148s
0x4018d2	68	pmuludq xmm4, xmm3	0.107s
0x4018d6	68	psrlq xmm3, 0x20	0.077s
0x4018db	68	pmuludq xmm3, xmm1	0.030s
0x4018df	68	pand xmm4, xmm0	0.075s
0x4018e3	68	psllq xmm3, 0x20	0.090s

- As expected compiler was able to vectorize the code
- But it uses SSE
- Let's rebuild with -xCORE-AVX2 and see if using wider vectors will help

VTune Amplifier result for AVX2 vectorization (ma_vect_avx256)



Bandwidth now peaks at more than 27 GB/s

VTune Amplifier result for AVX2 vectorization (ma_vect_avx256)

`for (j = 0; j < REPEATS; j++)`	
`{`	
`#pragma omp parallel for`	0.325s
`for (i = 0; i < size; i++)`	
`{`	
`c[i] = multiplier * _a[i] + _b[i];`	12.911s
`}`	
`}`	

0x401853	65	jb 0x40183f <Block 13>	
0x401855		**Block 14:**	
0x401855	65	vpbroadcastd ymm0, dword ptr [rsp]	
0x40185b		**Block 15:**	
0x40185b	68	vpmulld ymm1, ymm0, ymmword ptr [r10+r8*4]	0.03
0x401861	68	vpaddd ymm2, ymm1, ymmword ptr [rdi+r8*4]	6.84
0x401867	68	vmovdqu ymmword ptr [r9+r8*4], ymm2	6.02
0x40186d	65	add r8, 0x8	0.32
0x401871	65	cmp r8, r15	
0x401874	65	jb 0x40185b <Block 15>	

We can confirm that the loop is now uses AVX instructions (256-bit YMM registers)

PLATFORM PROFILER DEMO

DEMO

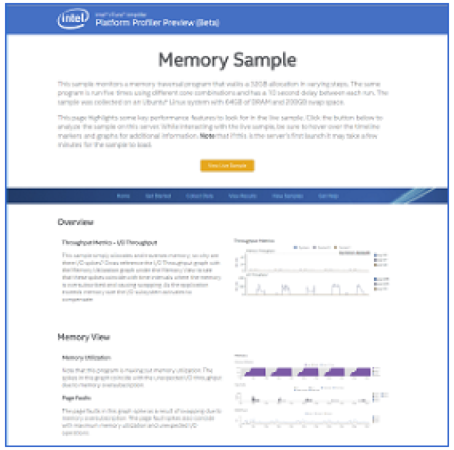
See Platform Profiler in Action

Matrix Multiplication

This sample was collected using an application derived from the Intel® Math Kernel Library Matrix Multiplication C Sample. The sample is not heavily CPU-bound and demonstrates efficient memory usage and throughput. However, the application is only running on a few cores and could benefit from parallelization. The sample was collected on an Ubuntu* Linux system with 64GB of DRAM and 36 cores.

File Copy

This sample was collected while continuously copying 30+ randomly sized files for five minutes on socket 1 followed by five minutes on socket 0. It shows a marked increase in efficiency when using socket 0 and demonstrates that applications that use a lot of file operations can benefit from using the socket closest to storage. The sample was collected on an Ubuntu* Linux system with two sockets and one NVMe* disk.



Memory Sample

This sample monitors a memory traversal program that walks a 32GB allocation in varying steps. The same program is run five times using different core combinations and has a 10 second delay between each run. The sample was collected on an Ubuntu* Linux system with 64GB of DRAM and 200GB swap space.