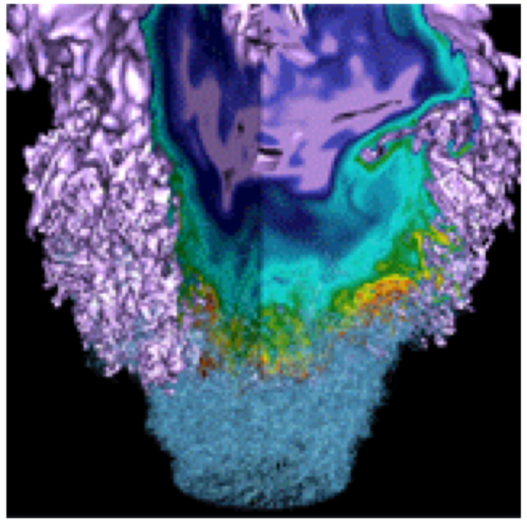
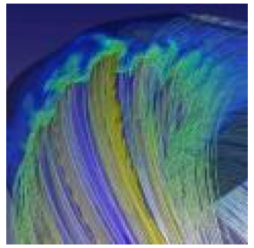
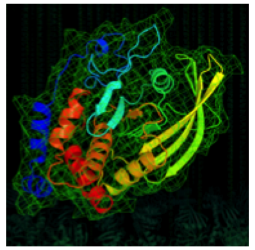
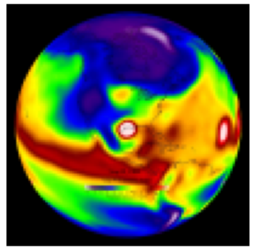
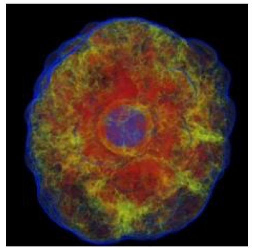
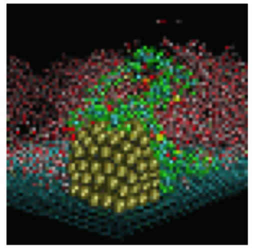
Hierarchical Roofline Analysis on CPUs



Charlene Yang
Lawrence Berkeley National Laboratory

ECP 2020, Houston

Outline

- Hierarchical Roofline on Intel CPUs
  - L1, L2, L3, HBM, DRAM

- Methodology for Roofline Data C llection
  - Machine characterization: pe
    - Empirical Roofline
  - Application characterizat
    - LIKWID, SDE, VTu

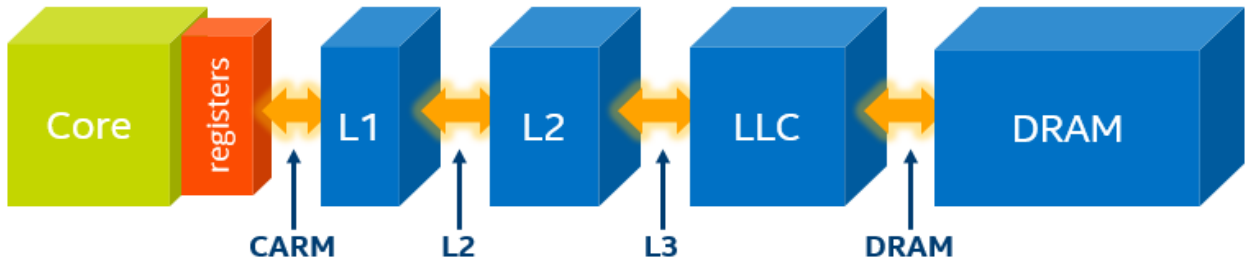
- A Stencil Example

This methodology can be extended to other CPUs, and other instruction types!

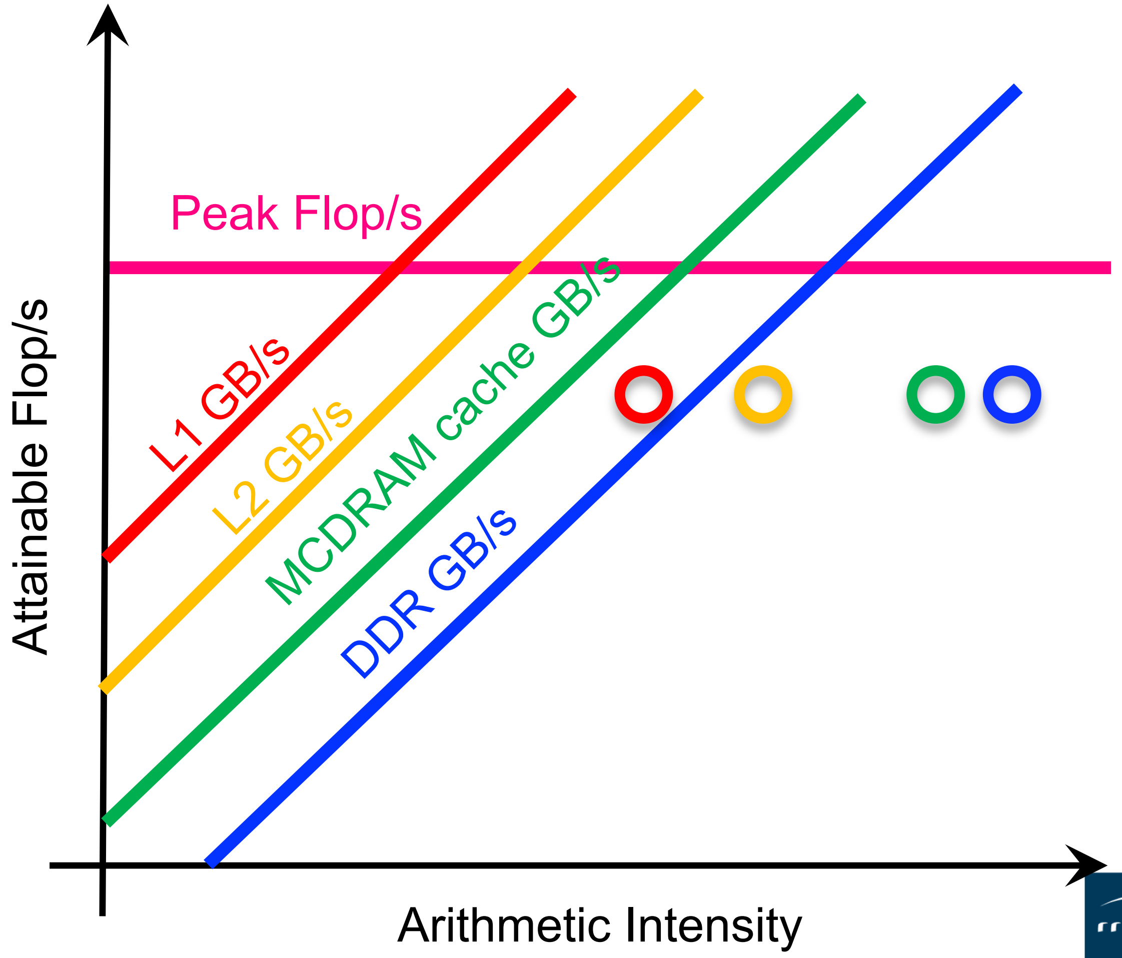
CPU Architecture: HSW

- **Goal: Hierarchical Roofline**
- **Machine Characterization**
  - compute/bandwidth peaks
- **Application Characterization**
  - Performance Throughput
    - FLOPs / runtime
  - Arithmetic Intensity
    - AI_DRAM = FLOPS / Bytes_DRAM
    - AI_MCDRAM = FLOPS / Bytes_MCDRAM
    - AI_L2 = FLOPS / Bytes_L2
    - AI_L1 = FLOPS / Bytes_L1

Data transfer between levels



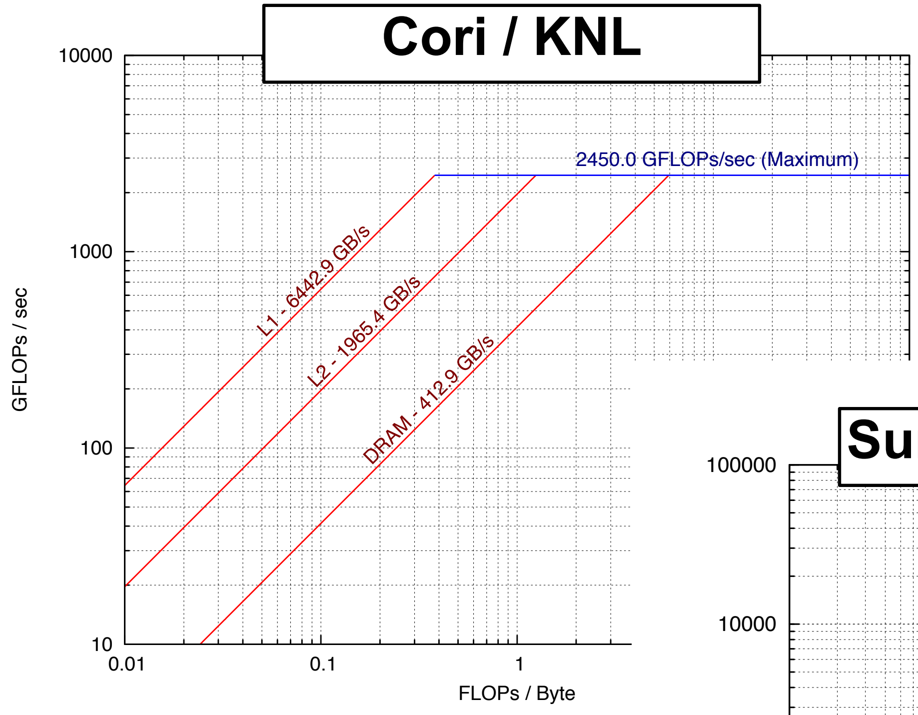
Courtesy of Zakhar Matveev

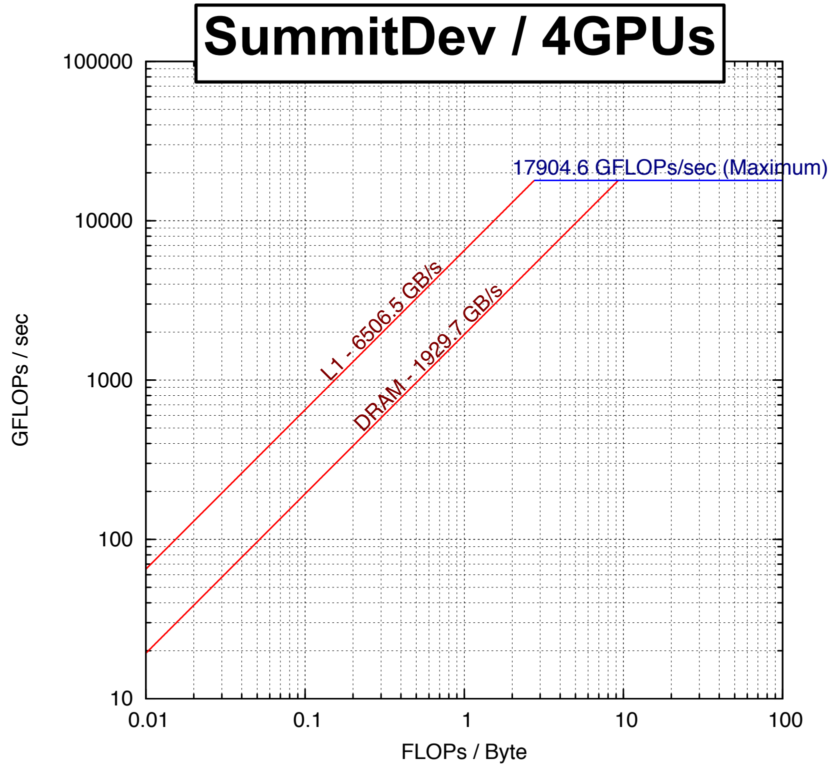


Machine Characterization

- **“Theoretical Performance” numbers can be highly optimistic…**
  - Pin BW vs. sustained bandwidth
  - TurboMode / Underclock for AVX
  - compiler failings on high-AI loops.

- **LBL developed the Empirical Roofline Toolkit (ERT)…**
  - Characterize CPU/GPU systems
  - Peak Flop rates
  - Bandwidths for each level of memory





https://bitbucket.org/berkeleylab/cs-roofline-toolkit/

https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

Application Characterization

- **How to get runtime, FLOPs, Bytes ….**
  - manual counting
  - performance counters
  - binary instrumentation

- **Tools we can use…**
  - LIKWID: vops, low overhead, no breakdown info
  - SDE + VTune: more accurate, high overhead, manual scripting required
  - Advisor: automated, high overhead, information rich
  - …

How Do We Count Flop's?

Manual Counting

- ▪ Go thru each loop nest and count the number of FP operations
- ✓ Works best for deterministic loop bounds
- ✓ or parameterize by the number of iterations (recorded at run time)
- ✗ Not scalable

Perf. Counters

- ▪ Read counter before/after
- ✓ More Accurate
- ✓ Low overhead (<%) == can run full MPI applications
- ✓ Can detect load imbalance
- ✗ Requires privileged access
- ✗ Requires manual instrumentation (+overhead) or full-app characterization
- ✗ Broken counters = garbage
- ✗ May not differentiate FMADD from FADD
- ✗ No insight into special pipelines

Binary Instrumentation

- ▪ Automated inspection of assembly at run time
- ✓ Most Accurate
- ✓ FMA-, VL-, and mask-aware
- ✓ Can count instructions by class/type
- ✓ Can detect load imbalance
- ✓ Can include effects from non-FP instructions
- ✓ Automated application to multiple loop nests
- ✗ >10x overhead (short runs / reduced concurrency)

How Do We Measure Data Movement?

Manual Counting

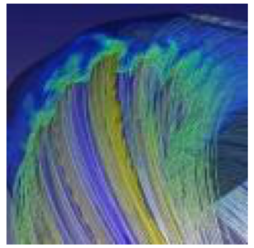
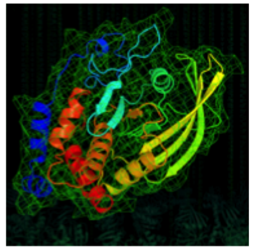
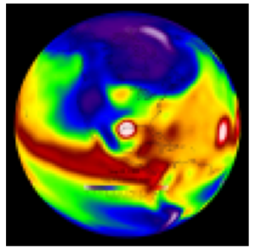
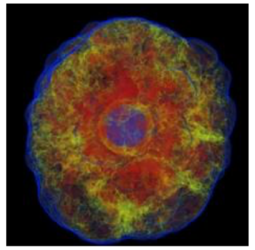
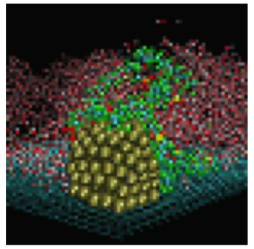
- Go thru each loop nest and estimate how many bytes will be moved
- Use a mental model of caches
- ✓ Works best for simple loops that stream from DRAM (stencils, FFTs, spare, …)
- ✗ N/A for complex caches
- ✗ Not scalable

Perf. Counters

- Read counter before/after
- ✓ Applies to full hierarchy (L2, DRAM,
- ✓ Much more Accurate
- ✓ Low overhead (<%) == can run full MPI applications
- ✓ Can detect load imbalance
- ✗ Requires privileged access
- ✗ Requires manual instrumentation (+overhead) or full-app characterization

Cache Simulation

- Build a full cache simulator driven by memory addresses
- ✓ Applies to full hierarchy and multicore
- ✓ Can detect load imbalance
- ✓ Automated application to multiple loop nests
- ✗ Ignores prefetchers
- ✗ >10x overhead (short runs / reduced concurrency)

Roofline with LIKWID

LIKWID

- LIKWID provides easy to use wrappers for measuring performance counters...
  - ✓ **Works on NERSC production systems**
  - ✓ Distills counters into user-friendly metrics (e.g. MCDRAM Bandwidth)
  - ✓ Minimal overhead (<1%)
  - ✓ Scalable in distributed memory (MPI-friendly)
  - ✓ Fast, high-level characterization
  - ✗ No timing breakdowns
  - ✗ Suffers from Garbage-in/Garbage Out
  - (i.e. hardware counter must be sufficient and correct)

https://github.com/RRZE-HPC/likwid

http://www.nersc.gov/users/software/performance-and-debugging-tools/likwid

LIKWID Utilities

likwid-topology	node topology
likwid-pin	process/thread affinity
likwid-memsweeper	cleanup memory & LLC
likwid-powermeter	power measurements
likwid-setFrequencies	CPU/uncore frequency manipulation
likwid-perfctr	hardware counter measurements
likwid-mpirun	hardware counter + MPI
likwid-bench	micro-benchmarking
likwid-agent	system monitoring
likwid-genTopoCfg	generate and store topology file

LIKWID Marker API

- By default, profiles whole program
- But Marker API allows regional profiling as well

```
#include <likwid.h>
……
LIKWID_MARKER_INIT;
#pragma omp parallel {
    LIKWID_MARKER_THREADINIT;
}
#pragma omp parallel {
    LIKWID_MARKER_START("foo");
    #pragma omp for
    for(i = 0; i < N; i++) {
            data[i] = omp_get_thread_num();
    }
    LIKWID_MARKER_STOP("foo");
}
LIKWID_MARKER_CLOSE;
```

focus on specific code regions

Example: likwid-perfctr –a

```
 Group name     Description
--------------------------------------------------------------------------------
HBM_OFFCORE     Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)
  TLB_INSTR     L1 Instruction TLB miss rate/ratio
   FLOPS_SP     Single Precision MFLOP/s
     BRANCH     Branch prediction miss rate/ratio
    L2CACHE     L2 cache miss rate/ratio
     ENERGY     Power and Energy consumption
FRONTEND_STALLS Frontend stalls
     ICACHE     Instruction cache miss rate/ratio
   TLB_DATA     L2 data TLB miss rate/ratio
        MEM     Memory bandwidth in MBytes/s
       DATA     Load to store ratio
         L2     L2 cache bandwidth in MBytes/s
   FLOPS_DP     Double Precision MFLOP/s
      CLOCK     Power and Energy consumption
  HBM_CACHE     Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)
        HBM     Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)
UOPS_STALLS     UOP retirement stalls
```

Example GPP: GFLOP/s

- GPP kernel on KNL: **171.960 GFLOPS/sec**
  - UOPS_RETIRED_PACKED_SIMD
  - UOPS_RETIRED_SCALAR_SIMD

- likwid-perfctr -C 0-63 -g **FLOPS_DP** ./gpp.knl.ex 512 2 32768 20
  - 8*UOPS_RETIRED_PACKED_SIMD+UOPS_RETIRED_SCALAR_SIMD

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	940.8064	14.7001	14.7001	14.7001
Runtime unhalted [s] STAT	402.9130	6.2371	9.8444	6.2955
Clock [MHz] STAT	96000.0155	1499.9955	1500.0007	1500.0002
CPI STAT	86.0772	1.3396	1.5850	1.3450
DP MFLOP/s (SSE assumed) STAT	44456.2105	688.9334	729.9324	694.6283
DP MFLOP/s (AVX assumed) STAT	86957.6422	1347.4354	1429.2337	1358.7132
DP MFLOP/s (AVX512 assumed) STAT	171960.5065	2664.4393	2827.8362	2686.8829
Packed MUOPS/s STAT	21250.7162	329.2510	349.6506	332.0424
Scalar MUOPS/s STAT	1954.7786	30.4313	30.6312	30.5434

Example GPP: MCDRAM + DDR GB/s

- kernel on KNL: **DDR 2.59GB/s + MCDRAM 63.71GB/s**
  - MC_CAS_READS/ MC_CAS_WRITES
  - EDC_RPQ_INSERTS/ EDC_WPQ_INSERTS
  - EDC_MISS_CLEAN/ EDC_MISS_DIRTY
- likwid-perfctr -C 0-63 -g **HBM_CACHE** ./gpp.knl.ex 512 2 32768 20

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	896.4352	14.0068	14.0068	14.0068
Runtime unhalted [s] STAT	390.2173	6.0393	9.6183	6.0971
Clock [MHz] STAT	95979.5220	1499.6763	1499.6807	1499.6800
CPI STAT	83.4239	1.2985	1.5496	1.3035
MCDRAM Memory read bandwidth [MBytes/s] STAT	63246.3054	0	63246.3054	988.2235
MCDRAM Memory read data volume [GBytes] STAT	885.8769	0	885.8769	13.8418
MCDRAM Memory writeback bandwidth [MBytes/s] STAT	468.4857	0	468.4857	7.3201
MCDRAM Memory writeback data volume [GBytes] STAT	6.5620	0	6.5620	0.1025
MCDRAM Memory bandwidth [MBytes/s] STAT	63714.7910	0	63714.7910	995.5436
MCDRAM Memory data volume [GBytes] STAT	892.4389	0	892.4389	13.9444
DDR Memory read bandwidth [MBytes/s] STAT	2569.3065	0	2569.3065	40.1454
DDR Memory read data volume [GBytes] STAT	35.9877	0	35.9877	0.5623
DDR Memory writeback bandwidth [MBytes/s] STAT	21.1772	0	21.1772	0.3309
DDR Memory writeback data volume [GBytes] STAT	0.2966	0	0.2966	0.0046
DDR Memory bandwidth [MBytes/s] STAT	2590.4837	0	2590.4837	40.4763
DDR Memory data volume [GBytes] STAT	36.2843	0	36.2843	0.5669

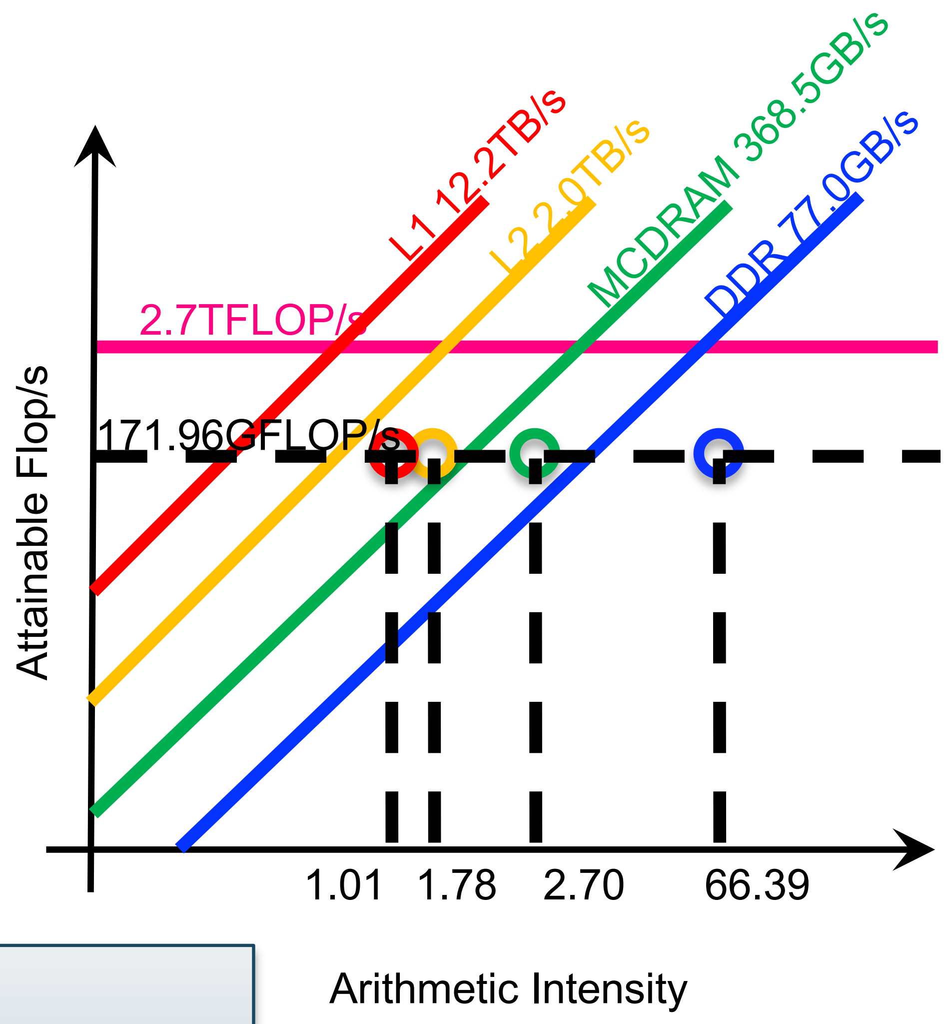
Example GPP: L2 GB/s

- kernel on KNL: L2 96.80GB/s
  - L2_REQUESTS_REFERENCE
  - OFFCORE_RESPONSE_0_OPTIONS
- likwid-perfctr -C 0-63 -g L2 ./gpp.knl.ex 512 2 32768 20

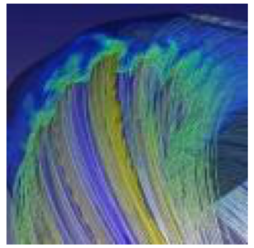
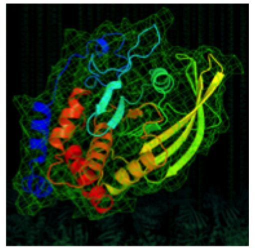
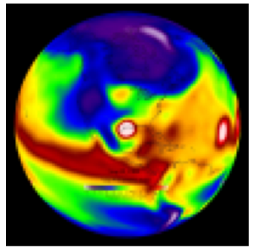
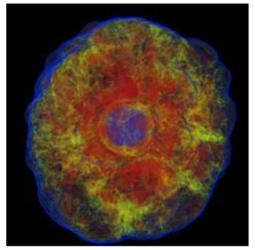
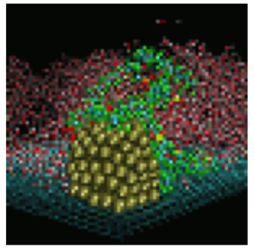
Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	895.5200	13.9925	13.9925	13.9925
Runtime unhalted [s] STAT	392.3078	6.0719	9.6599	6.1298
Clock [MHz] STAT	95999.4279	1499.9861	1499.9914	1499.9911
CPI STAT	83.8844	1.3055	1.5567	1.3107
L2 non-RFO bandwidth [MBytes/s] STAT	96803.9243	1498.7686	1904.3169	1512.5613
L2 non-RFO data volume [GByte] STAT	1354.5272	20.9715	26.6461	21.1645
L2 RFO bandwidth [MBytes/s] STAT	0	0	0	0
L2 RFO data volume [GByte] STAT	0	0	0	0
L2 bandwidth [MBytes/s] STAT	96803.9243	1498.7686	1904.3169	1512.5613
L2 data volume [GByte] STAT	1.354528e+06	20971.5004	26646.1299	21164.4950

Example GPP: L1 GB/s

- kernel on KNL: **L1 170.77GB/s**
  - MEM_UOPS_RETIRED_ALL_LOADS
  - MEM_UOPS_RETIRED_ALL_STORES
- likwid-perfctr -C 0-63 -g **DATA** ./gpp.knl.ex 512 2 32768 20
  - (MEM_UOPS_RETIRED_ALL_LOADS + MEM_UOPS_RETIRED_ALL_STORES)*64/runtime
  - -g DATA is for load-to-store ratio, but can be used to estimate L1 bandwidth (assume all loads are vector loads)



AI (DRAM):	66.39
AI (MCDRAM):	2.70
AI (L2):	1.78
AI (L1):	1.01
Performance:	171.960 GFLOPS/s

Roofline with SDE and VTune

Intel Software Development Emulator (SDE)

- **Dynamic instruction tracing**
  - ✓ Accounts for actual loop lengths and branches
  - ✓ Counts instruction types, lengths, etc…
  - ✓ Can mark individual regions
  - ✓ Support for MPI+OpenMP
  - ✓ Can be used to calculate FLOPs (VL-, FMA-, and precision-aware)
  - ✗ Post processing can be expensive.
  - ✗ No insights into cache behavior or DRAM data movement
  - ✗ X86 only

Parsing the Output

- When the job completes, you'll have a series of files prefixed with "sde_".
- Parse the output to summarize the results...
  `./parse-sde.sh sde_2p16t*`
- Use the "**Total FLOPs**" line as the numerator in all AI's and performance
- Use the "**Total Bytes**" line as the denominator in the L1 AI
- Can infer vectorization rates and precision

```
$ ./parse-sde.sh sde_2p16t*
Search stanza is "EMIT_GLOBAL_DYNAMIC_STATS"
elements_fp_single_1 = 0
elements_fp_single_2 = 0
elements_fp_single_4 = 0
elements_fp_single_8 = 0
elements_fp_single_16 = 0
elements_fp_double_1 = 2960
elements_fp_double_2 = 0
elements_fp_double_4 = 999999360
elements_fp_double_8 = 0
--->Total single-precision FLOPs = 0
--->Total double-precision FLOPs = 4000000400
--->Total FLOPs = 4000000400
mem-read-1 = 8618384
mem-read-2 = 1232
mem-read-4 = 137276433
mem-read-8 = 149329207
mem-read-16 = 1999998720
mem-read-32 = 0
mem-read-64 = 0
mem-write-1 = 264992
mem-write-2 = 560
mem-write-4 = 285974
mem-write-8 = 14508338
mem-write-16 = 0
mem-write-32 = 499999680
mem-write-64 = 0
--->Total Bytes read = 33752339756
--->Total Bytes written = 16117466472
--->Total Bytes = 49869806228
```

LIKWID vs. SDE

- Recall, LIKWID counts vector uops while SDE counts instructions
- Why does this matter?
  - VL-aware KNL has scalar but treats 128b, 256b, and 512b as 512b
  - precision-aware User has to know which precision they use
  - mask-aware KNL counters ignore masks
  - FMA-aware LIKWID assumes 1 flop per element
  - KNL counts vector integer, stores, NT stores, and gathers as vector uops (**and thus as potential flop/s**)

➢ **LIKWID's and SDE's counts of #FP ops and Gflop/s can be different (very different for linear algebra).**

LIKWID vs. SDE/VTune

- **SDE FLOPS:**
  - sde64 -knl -d -iform 1 -omix my_mix.out -global_region -- ./gpp.knl.ex 512 2 32768 20
  - ./parse-sde.sh my_mix.out
  - --->Total FLOPs = 2775769815463

LIKWID
2527.81 GFLOPS

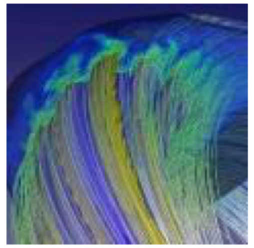
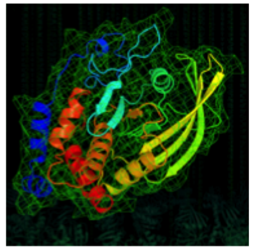
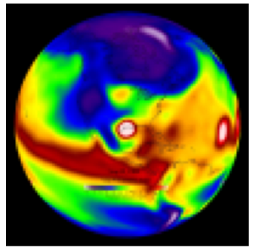
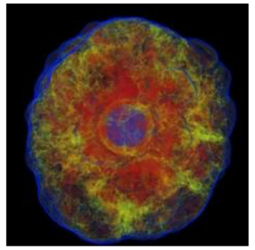
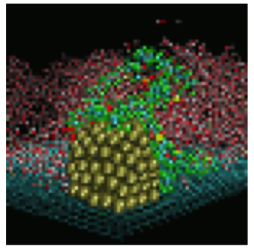
difference
~8.9%

- **VTune Bytes:**
  - amplxe-cl -collect memory-access -finalization-mode=deferred -r my_vtune/ -- ./gpp.knl.ex 512 2 32768 20
  - amplxe-cl -report summary -r my_vtune/ > my_vtune
  - ./parse-vtune.sh my_vtune.summary
  - DDR --->Total Bytes = 35983553088
  - HBM --->Total Bytes = 963486016448

LIKWID	difference
DDR: 36.28 GB	~0.8%
HBM: 892.44 GB	~7.4%

- http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

NERSC

Roofline with Advisor

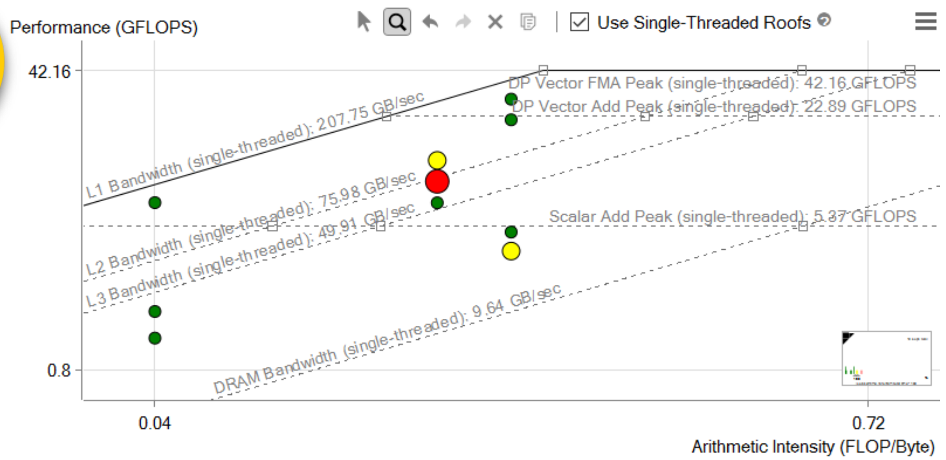
The Roofline Feature in Intel® Advisor

1

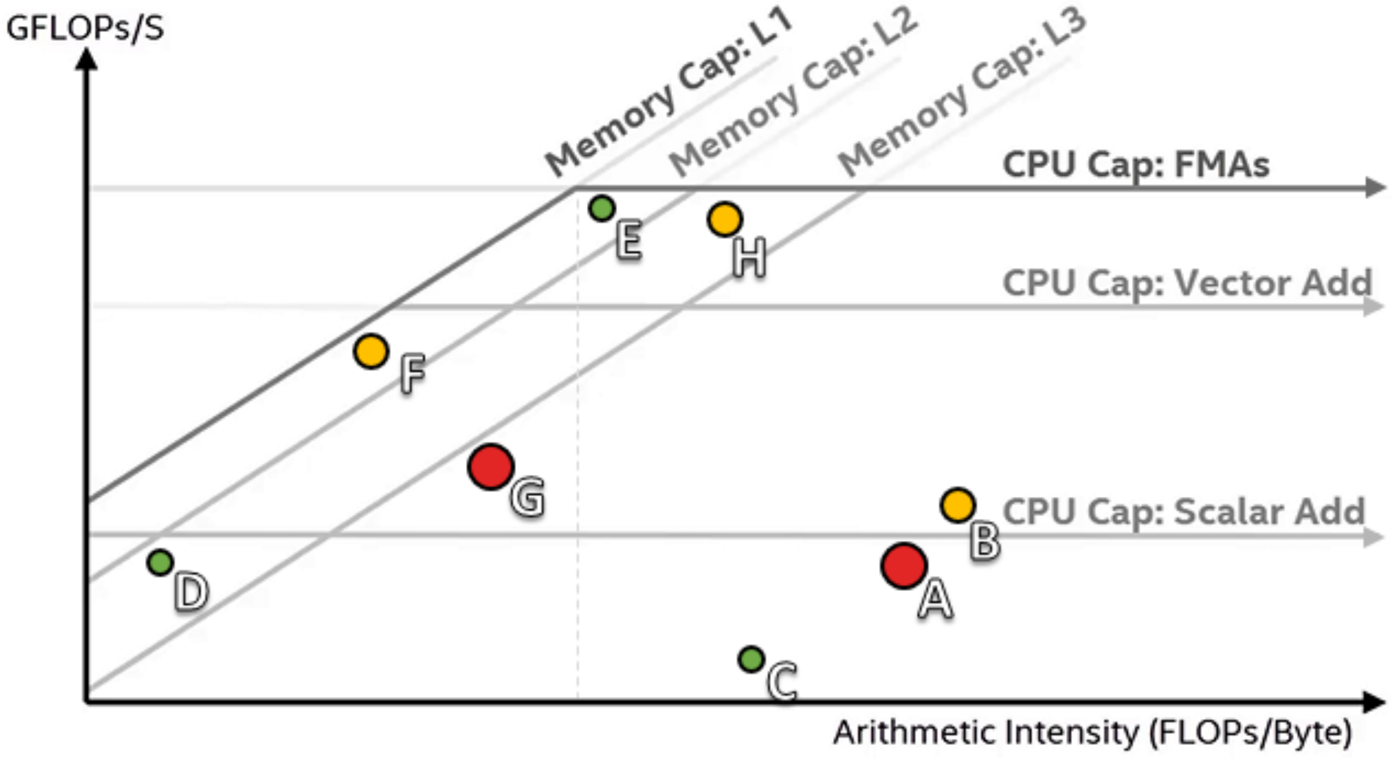
A single button or CLI command runs the Survey and FLOPS analyses to generate the Roofline chart.

2

3

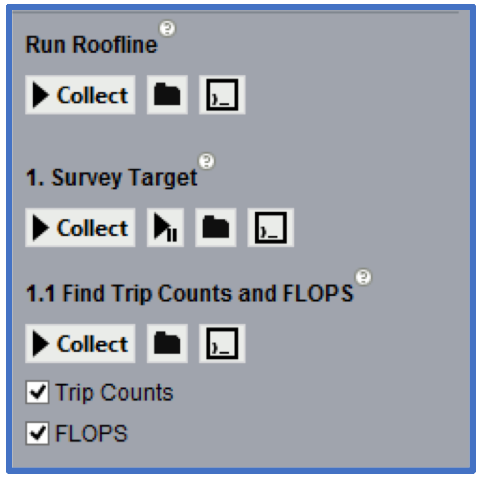


- Automate data collection, one dot per kerne
- Hierarchical Roofline for multiple caches
- Automatically benchmarks target system
- Fully integrated with other Advisor features



Courtesy of Zakhar Matveev

Intel Advisor: 2-pass Approach



Roofline : Axis X: **AI** = #**FLOP** / #**Bytes** Axis Y: **FLOP/S** = **#FLOP** (mask aware) / **#Seconds**	Overhead
Step 1: Survey (-collect survey) - Provide **#Seconds** - *Root access not needed* - User mode sampling, non-intrusive.	**1x**
Step 2: FLOPS *(-collect tripcounts –flops)* - Provide #**FLOP**, #**Bytes**, AVX-512 Mask - *Root access not needed* - Precise, instrumentation based, count number of instructions	**5-10x**

Intel Advisor: Command Lines for Roofline

```
$ source advixe-vars.sh
```

1st method. Not compatible with MPI applications :

```
$ advixe-cl -collect roofline --project-dir ./dir -- ./app
```

2nd method (old, more flexible):

```
$ advixe-cl -collect survey --project-dir ./dir -- ./app
$ advixe-cl -collect tripcounts -flop --project-dir ./dir -- ./app
```

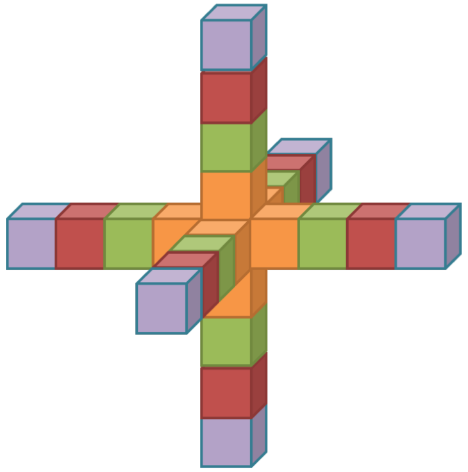
(optional) copy data to your UI desktop system

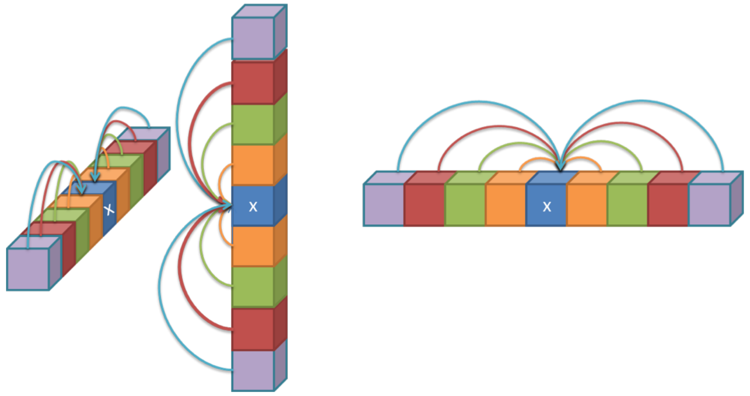
```
$ advixe-gui ./dir
```

IRM How-to:

https://software.intel.com/en-us/articles/integrated-roofline-model-with-intel-advisor

Intel Advisor: A Stencil Example Iso3DFD





```
For (int iz=0; iz<n3; iz++)
For (int iy=0; iy<n2; iy++)
For (int ix=0; ix<n1; ix++) {
    int offset = iz*dimn1n2 + iy*n1 + ix;
    float value = 0.0;
    value += ptr_prev[offset]*coeff[0];
    for(int ir=1; ir<= 8 ; ir++) {
        value += coeff[ir] * (ptr_prev[offset + ir] + ptr_prev[offset - ir]);
        value += coeff[ir] * (ptr_prev[offset + ir*n1] + ptr_prev[offset - ir*n1]);
        value += coeff[ir] * (ptr_prev[offset + ir*dimn1n2] + ptr_prev[offset - ir*dimn1n2]);
    }
    ptr_next[offset] = 2.0f* ptr_prev[offset] - ptr_next[offset] + value*ptr_vel[offset];
}
```

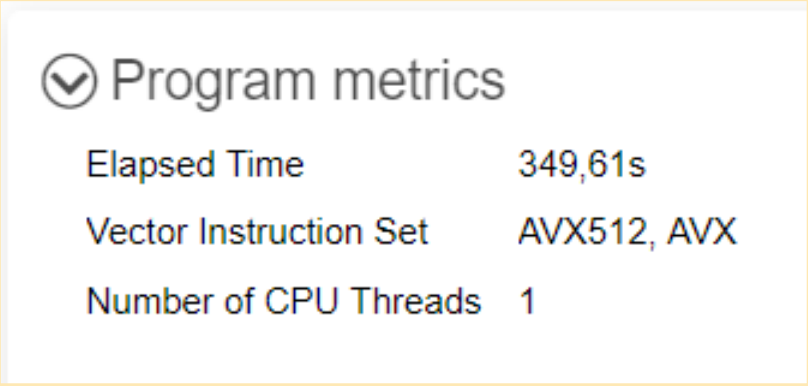
Intel Advisor: A Stencil Example Iso3DFD

Progressive levels of optimization

- Dev00: **unoptimized** implementation of iso3DFD
- Dev01: adding **OpenMP threading**
- Dev02: reverse loops improving **memory access** pattern
- Dev03: **vectorization**, improve compute throughput and L1 AI
- Dev04: implement **cache blocking**, improving DRAM AI

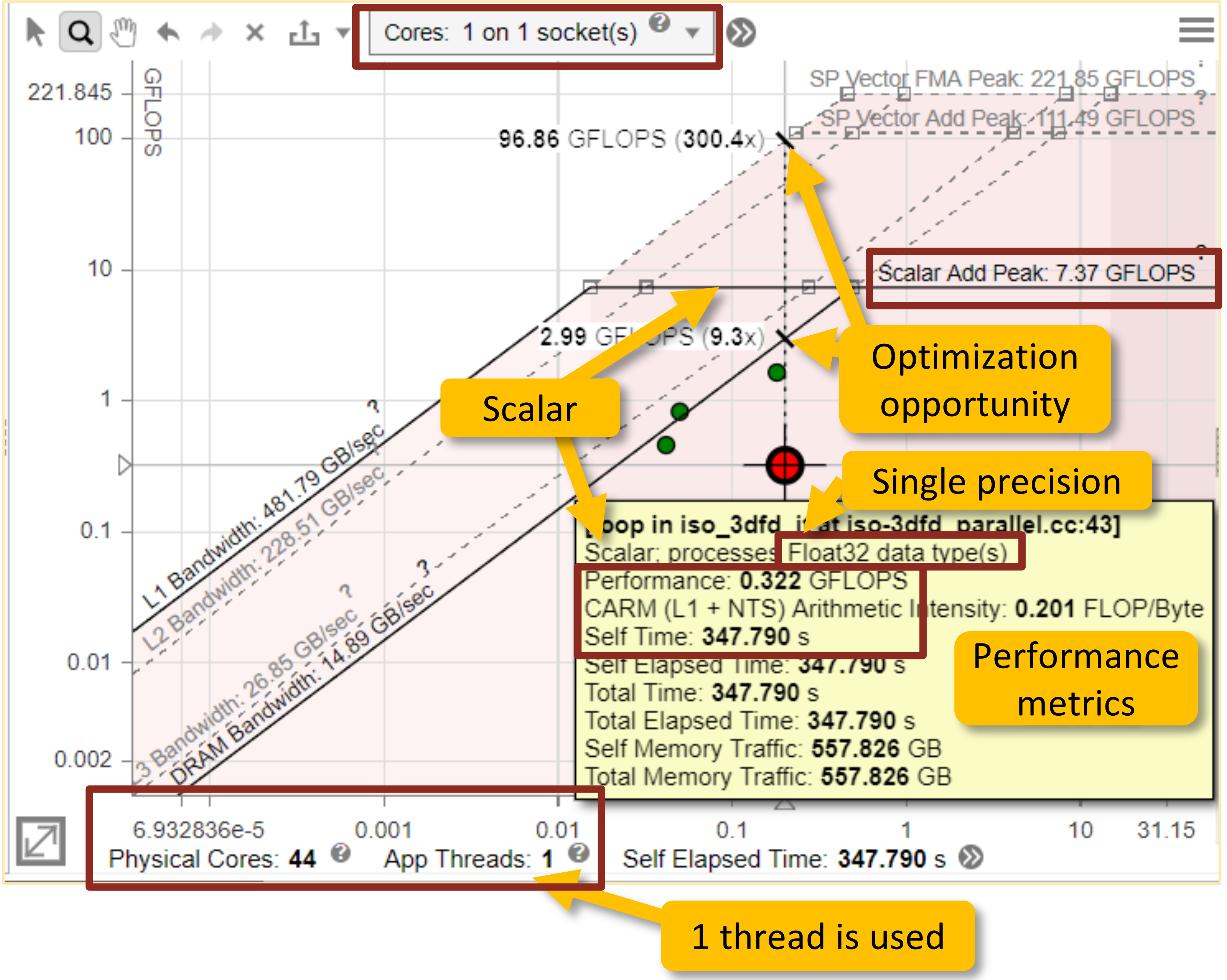
v00 – where am I?

- Main hotspot is loop at iso-3dfd_parallel.cc:43
- Performance is far from machine peak
- Problem:
  - Serial – 1 thread (Summary, Roofline)
  - Scalar

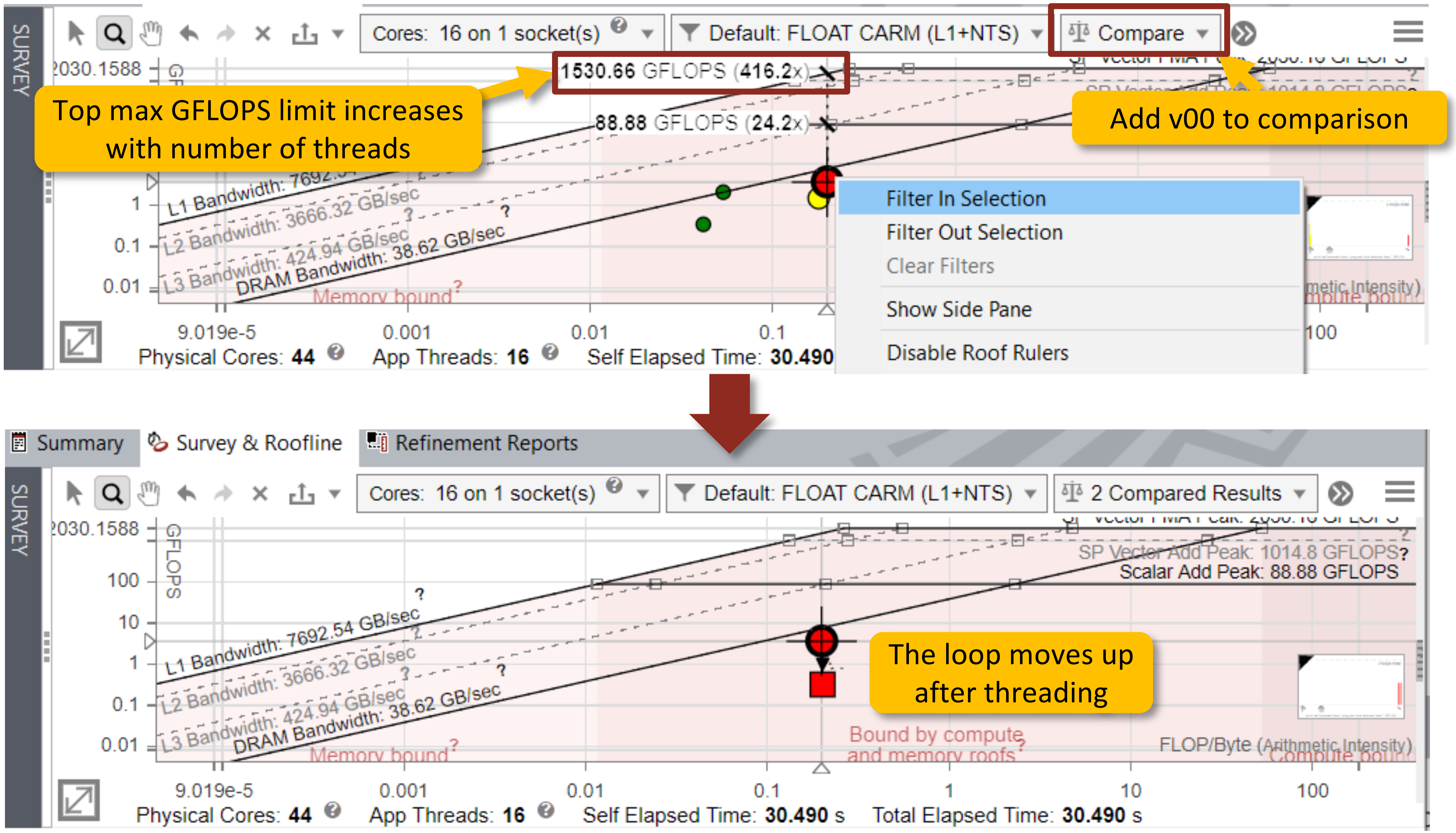


Program metrics

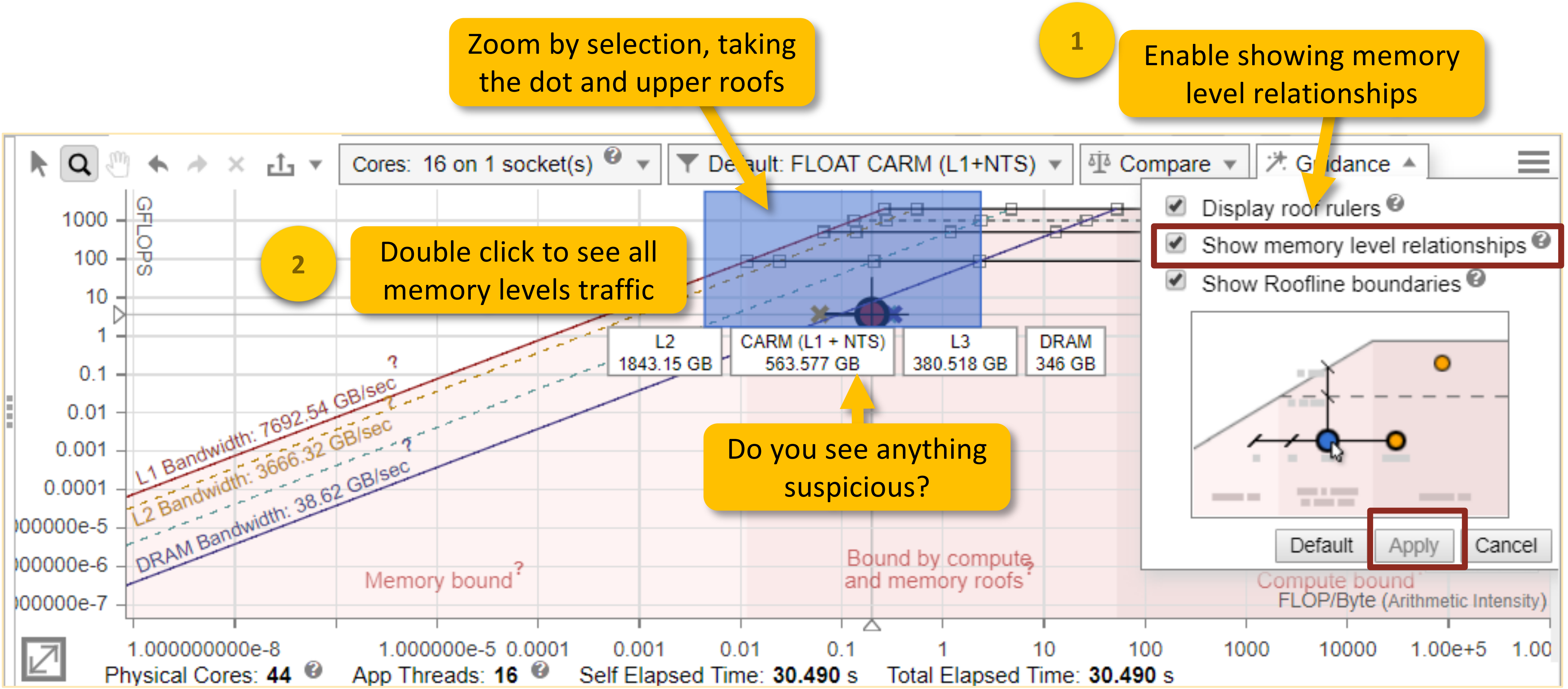
Elapsed Time	349,61s
Vector Instruction Set	AVX512, AVX
Number of CPU Threads	1



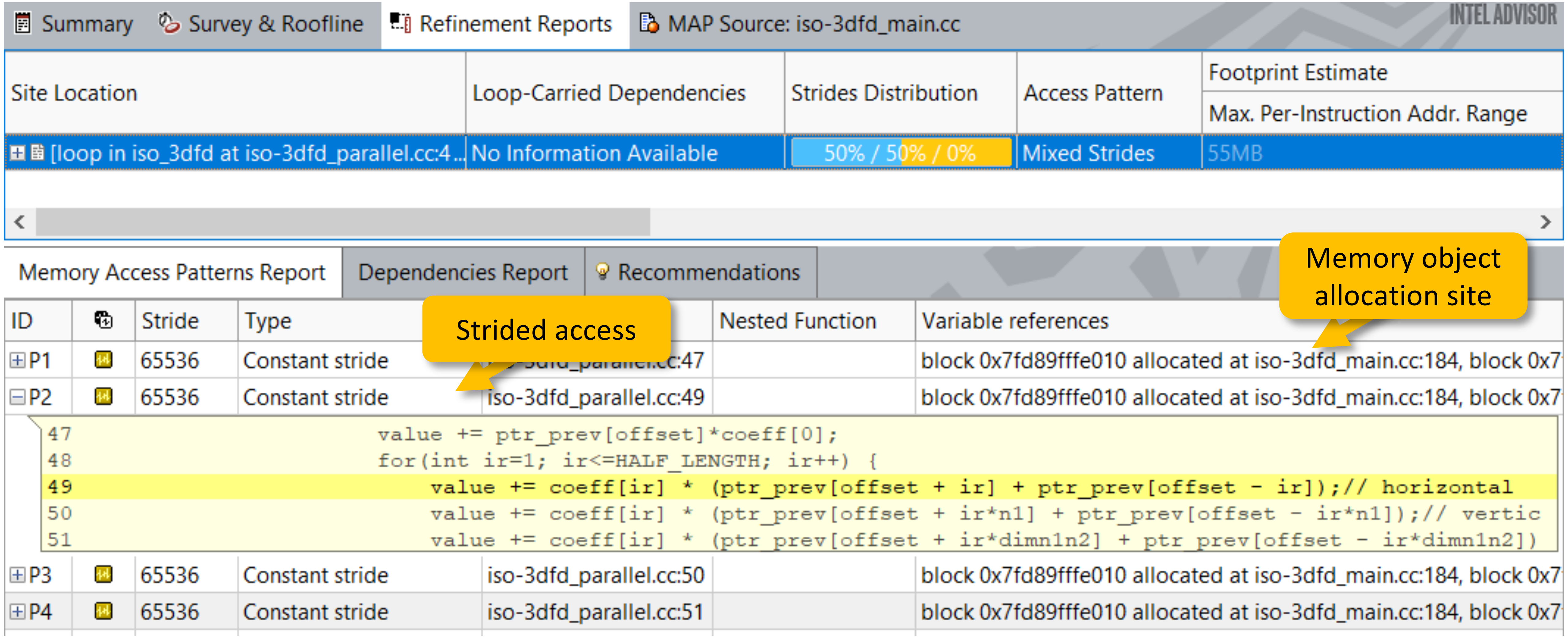
v01 – introduce OpenMP threading



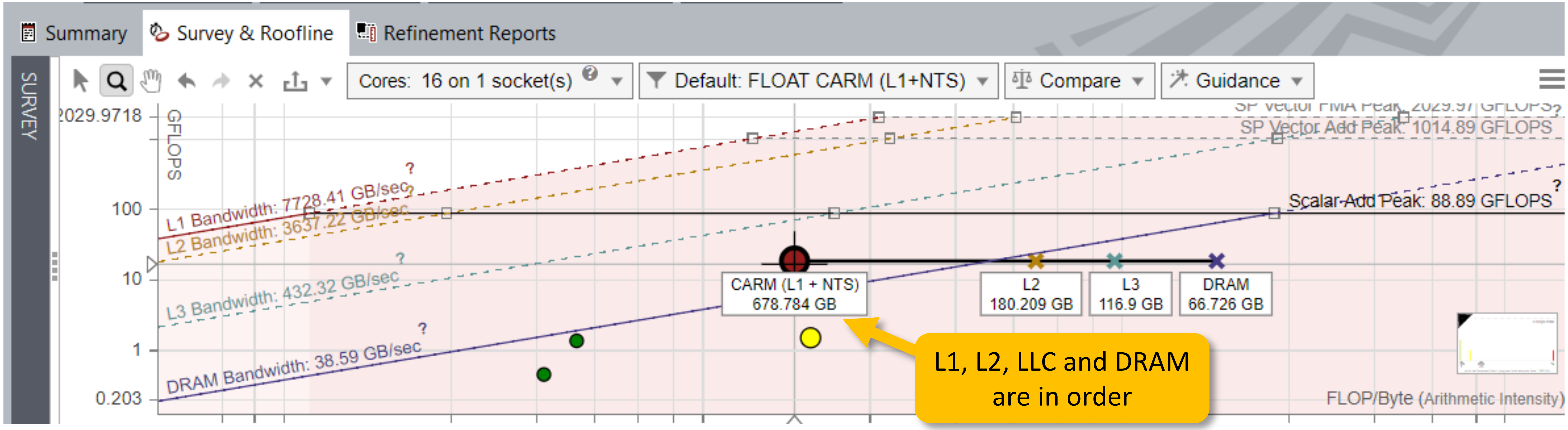
Enable Integrated Roofline Model



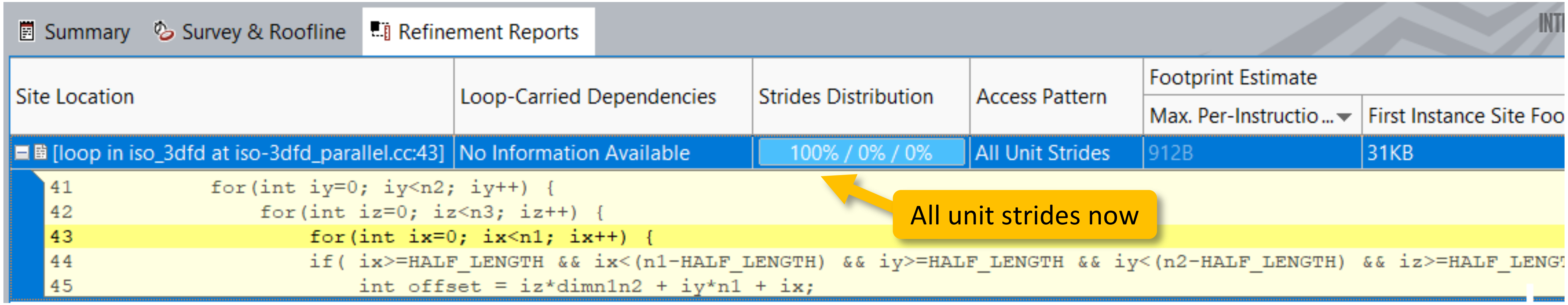
v01 – Memory Access Patterns



v02 – reverse loops

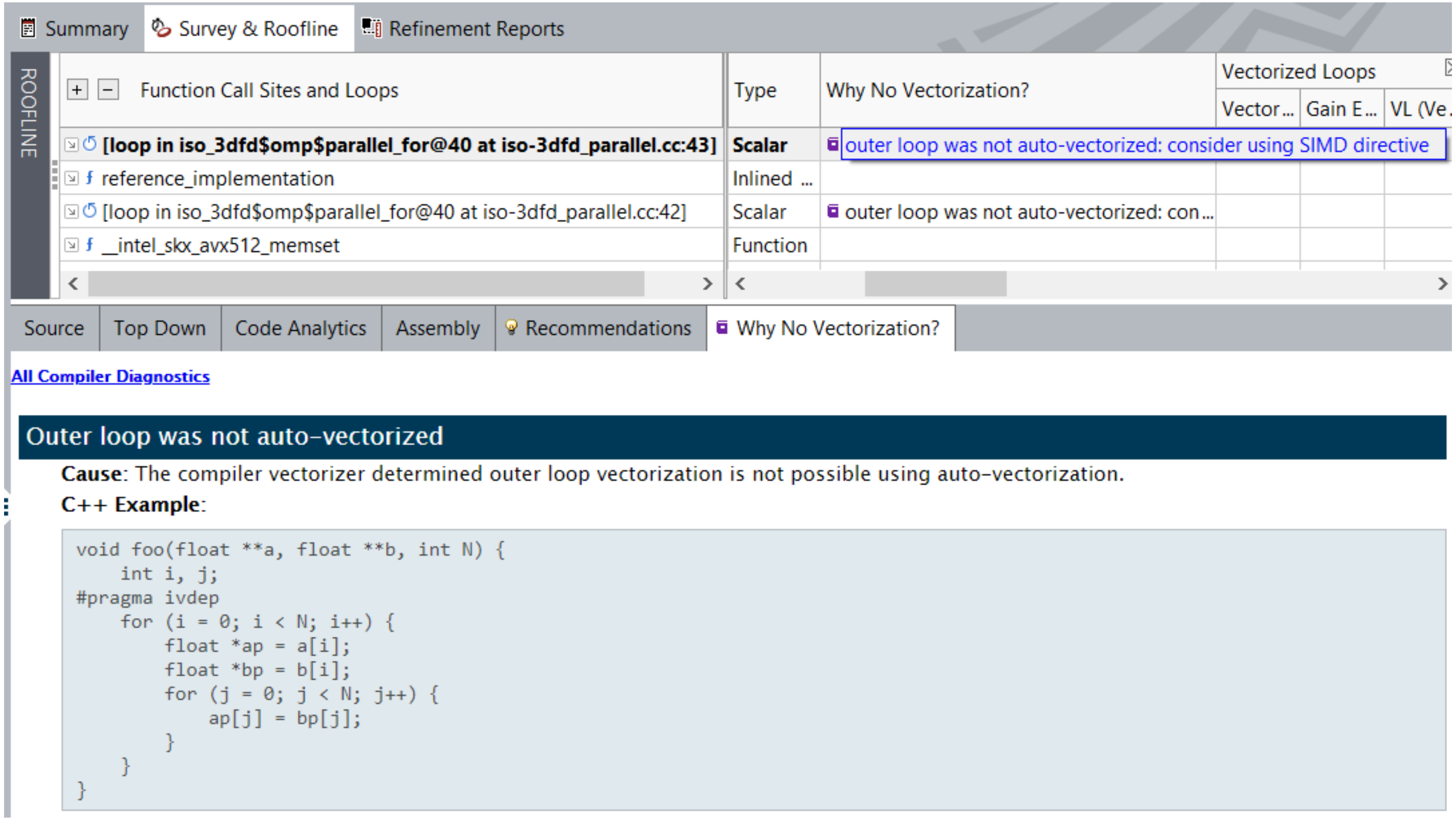


L1, L2, LLC and DRAM are in order

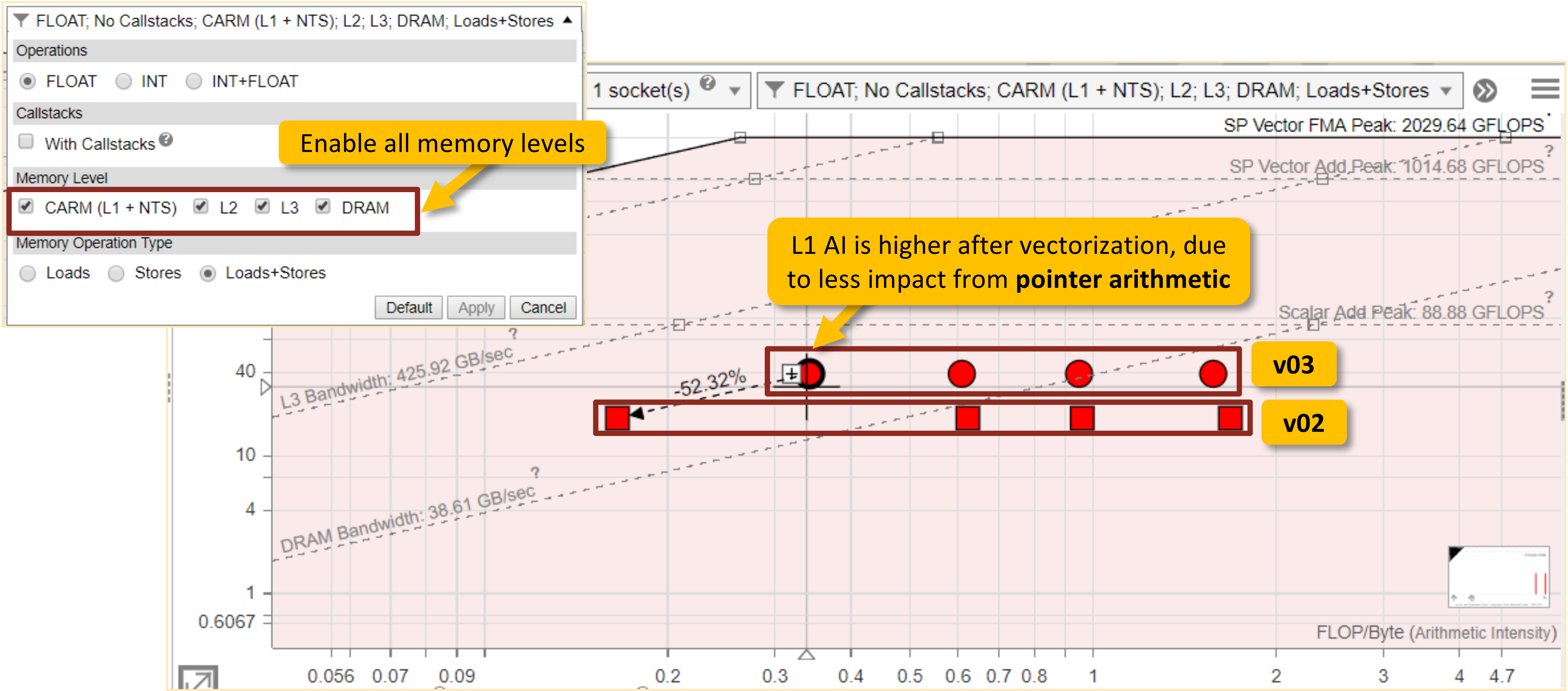


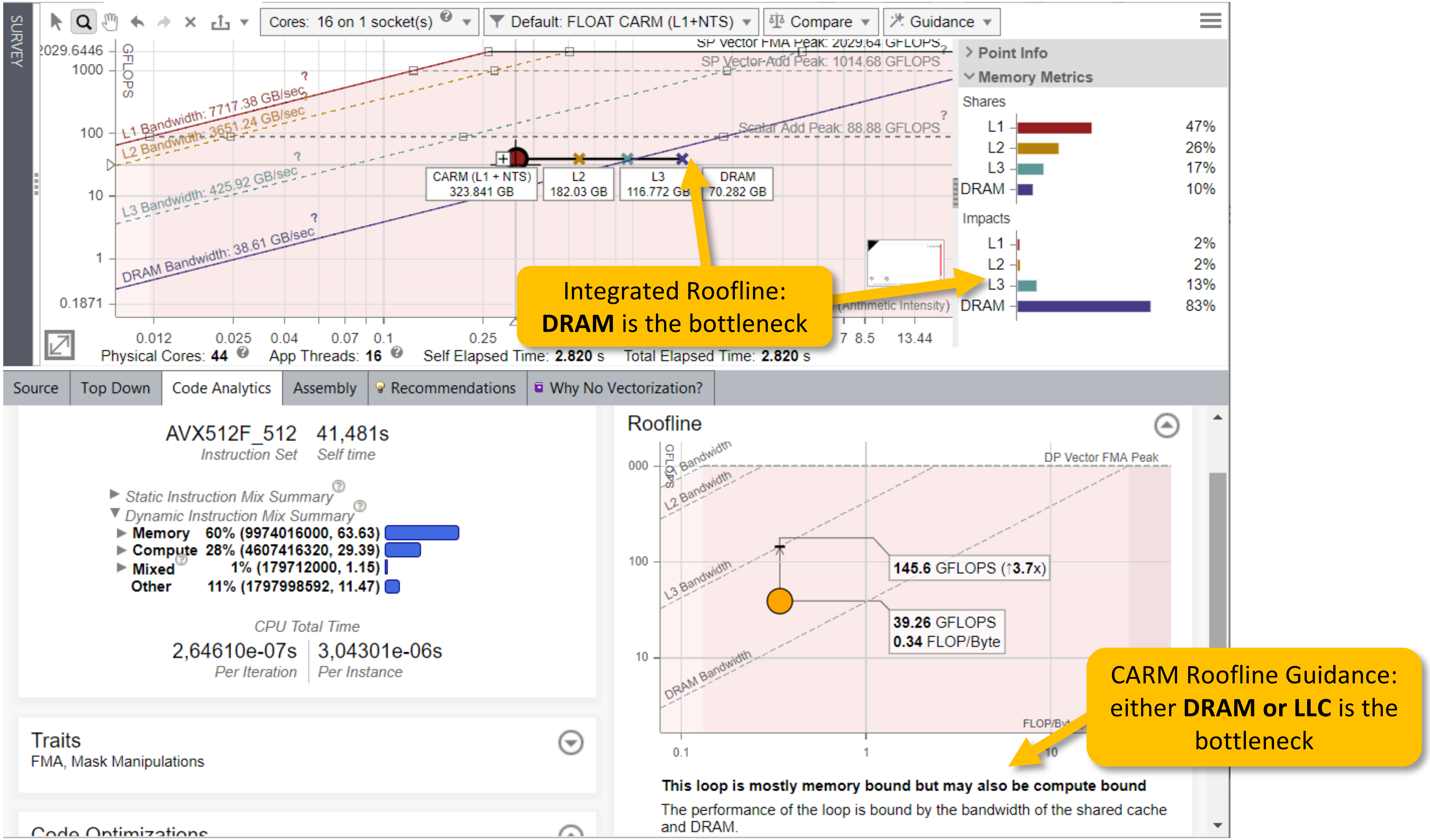
All unit strides now

v02 – find reason for no vectorization



Compare all memory levels with v02





SURVEY
Cores: 16 on 1 socket(s)
Default: FLOAT CARM (L1+NTS)
Compare
Guidance
GFLOPS
2029.6446
1000
100
10
1
0.1871
SP Vector FMA Peak: 2029.64 GFLOPS
SP Vector Add Peak: 1014.68 GFLOPS
Scalar Add Peak: 88.88 GFLOPS
L1 Bandwidth: 7717.38 GB/sec
L2 Bandwidth: 3651.24 GB/sec
L3 Bandwidth: 425.92 GB/sec
DRAM Bandwidth: 38.61 GB/sec
CARM (L1 + NTS) 323.841 GB
L2 182.03 GB
L3 116.772 GB
DRAM 70.282 GB
(Arithmetic Intensity)
0.012
0.025
0.04
0.07
0.1
0.25
7
8.5
13.44
Physical Cores: 44
App Threads: 16
Self Elapsed Time: 2.820 s
Total Elapsed Time: 2.820 s
Point Info
Memory Metrics
Shares
L1 47%
L2 26%
L3 17%
DRAM 10%
Impacts
L1 2%
L2 2%
L3 13%
DRAM 83%
Integrated Roofline:
DRAM is the bottleneck
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
AVX512F_512
Instruction Set
41,481s
Self time
Static Instruction Mix Summary
Dynamic Instruction Mix Summary
Memory 60% (9974016000, 63.63)
Compute 28% (4607416320, 29.39)
Mixed 1% (179712000, 1.15)
Other 11% (1797998592, 11.47)
CPU Total Time
2,64610e-07s
Per Iteration
3,04301e-06s
Per Instance
Traits
FMA, Mask Manipulations
Code Optimizations
Roofline
DP Vector FMA Peak
L1 Bandwidth
L2 Bandwidth
L3 Bandwidth
DRAM Bandwidth
000
100
10
0.1
1
10
FLOP/Byte
145.6 GFLOPS (↑3.7x)
39.26 GFLOPS
0.34 FLOP/Byte
This loop is mostly memory bound but may also be compute bound
The performance of the loop is bound by the bandwidth of the shared cache and DRAM.
CARM Roofline Guidance:
either DRAM or LLC is the
bottleneck

v04 – implement cache blocking