Roofline on CPU-based Systems

Samuel Williams
Computational Research Division
Lawrence Berkeley National Lab
SWWilliams@lbl.gov

With slides from Charlene Yang,
Doug Doerfler, and Zakhar Matveev

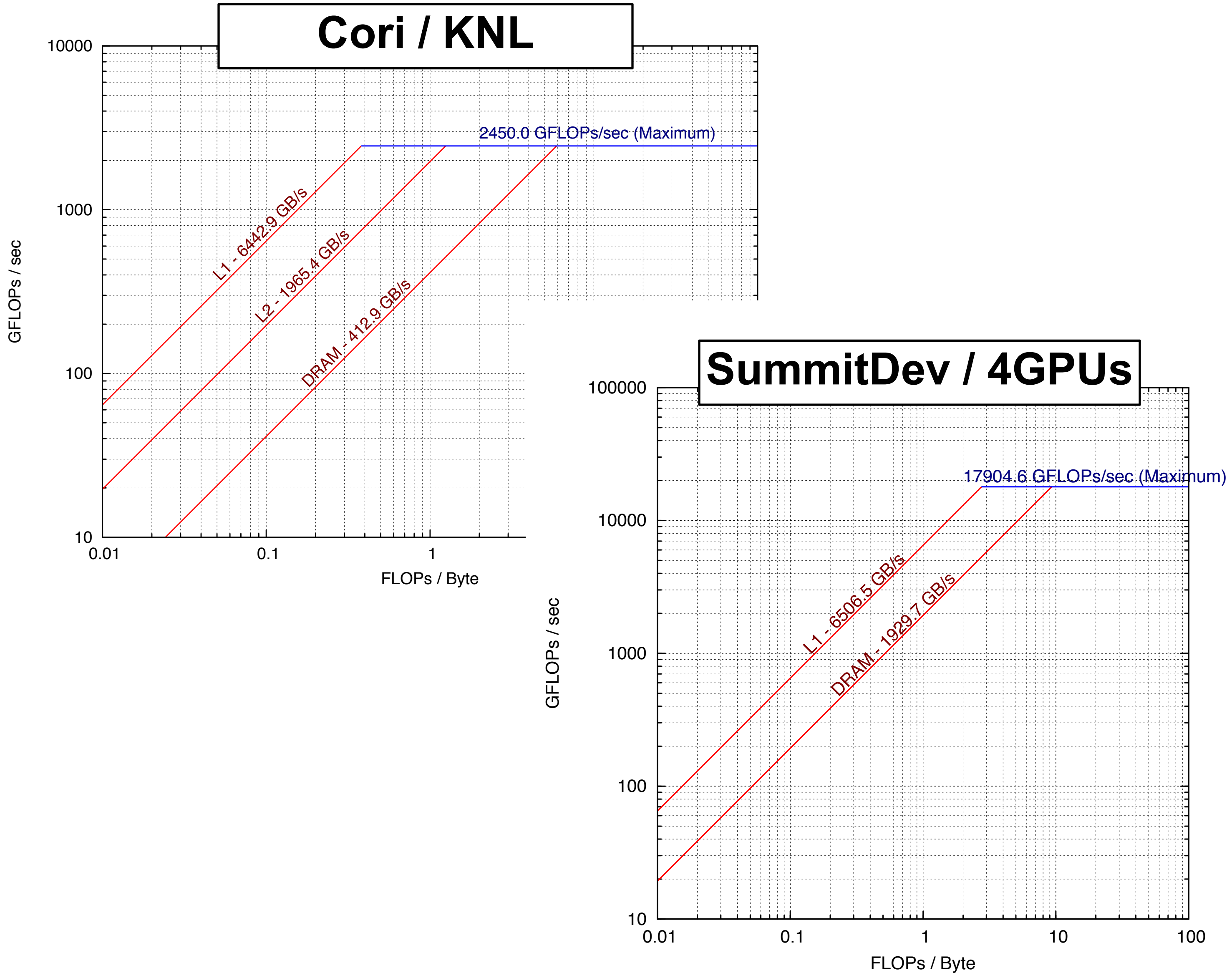
Acknowledgements

- This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.
- This material is based upon work supported by the DOE RAPIDS SciDAC Institute.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

Machine Characterization

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
  - **MPI+OpenMP/CUDA == multiple GPUs**



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

https://github.com/cyanguwa/nersc-roofline/

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

ERT Configuration

Kernel.c

```
loop over ntrials
    distribute dataset on threads and each
computes ERT_FLOPS
```

Kernel.h

```
ERT_FLOPS=1: a = b + c
ERT_FLOPS=2: a = a x b + c
```

config.txt

```
ERT_FLOPS        1,2,4,8,16,32,64
ERT_MPI_PROCS    2,4,8,16,32,64
ERT_OPENMP_THREADS 1-256
ERT_MEMORY_MAX   1073741824
ERT_WORKING_SET_MIN 1
ERT_TRIALS_MIN   1
...
```

Driver.c (uses some Macros from config.txt)

```
initialize MPI, OpenMP
loop over dataset sizes <= ERT_MEMORY_MAX
    loop over trial sizes >= ERT_TRIALS_MIN
        start timer
        call kernel
        end timer
```

Job script

```
./ert config.txt
```

ert (Python)

```
create directories
loop over ERT_FLOPS, MPI_PROCS/OMP_THREADS
    call driver, kernel
```

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

https://github.com/cyanguwa/nersc-roofline/

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

ERT Caveats

- Nominally, ERT runs a series of benchmarks
  - Read-modify-write Polynomial of degree-K on a vector of size N
  - Trivially auto-vectorized
  - Demands a unroll-and-jam or large OOO window to hit peak.
  - 1:1 Read:Write ratio
  - Varies both K and N
- From these it extrapolates cache capacities and bandwidths
  - By convention it labels the largest/slowest 'DRAM' and the smallest/fastest 'L1'
  - If N<LLC size, then it will identify the LLC 'DRAM' (e.g. on KNL, N>16GB)
  - On architectures that don't cache writes in the L1 (or are WT), ERT will label L2 as 'L1'
  - On architectures that have a 2:1 read:write cache bandwidth, ERT will underestimate aggregate cache bandwidth (it uses a 1:1 benchmark)

Application Characterization

Measuring AI

- To characterize execution with Roofline we need…
  - **Time**
  - **Flops** (=> flop’s / time)
  - **Data movement** between each level of memory (=> Flop’s / GB’s)
- We can look at the full application…
  - Coarse grained, 30-min average
  - Misses many details and bottlenecks
- or we can look at individual loop nests…
  - Requires auto-instrumentation on a loop by loop basis
  - Moreover, we should probably differentiate data movement or flops on a core-by-core basis.

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)

Performance Counter Issues

Performance Counter Limitations

- Capture aspects architects (not programmers) think are important
- May lack important detail
- Not standardized (vendor-specific)
- Not required to be functional or correct (not part of the ISA)

Performance Counters and SIMD

- SIMD instruction sets are ever evolving.
- Today, they can incorporate…
  - Different Vector Lengths (VL)… 128b, 256b, 512b, …
  - Different precisions… double, single, half, … 8x64b, 16x32b, or 32x16b
  - Use of FMA (1 or 2 flops per element)
  - Use of masks (predicates) to disable execution on certain lanes.
- Thus, a performance counter might be:
  - VL-aware (#operations scales with VL)
  - Precision-aware (#operations increases with reduced precision)
  - FMA-aware (FMAs are 2 flops per element vs. 1)
  - Mask-aware (#operations only incudes unmasked operations)

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)

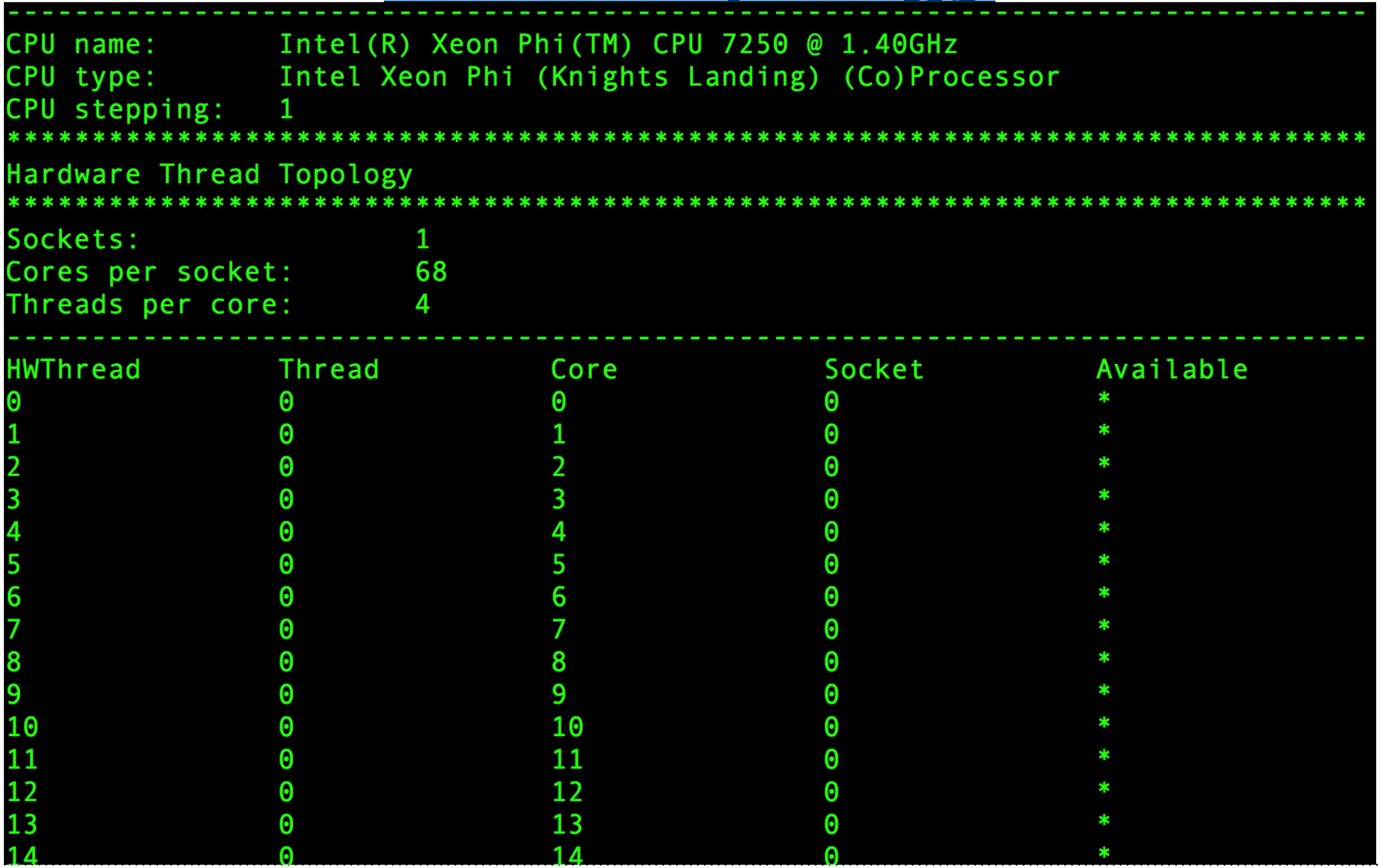
https://github.com/RRZE-HPC/likwid

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

LIKWID Tools

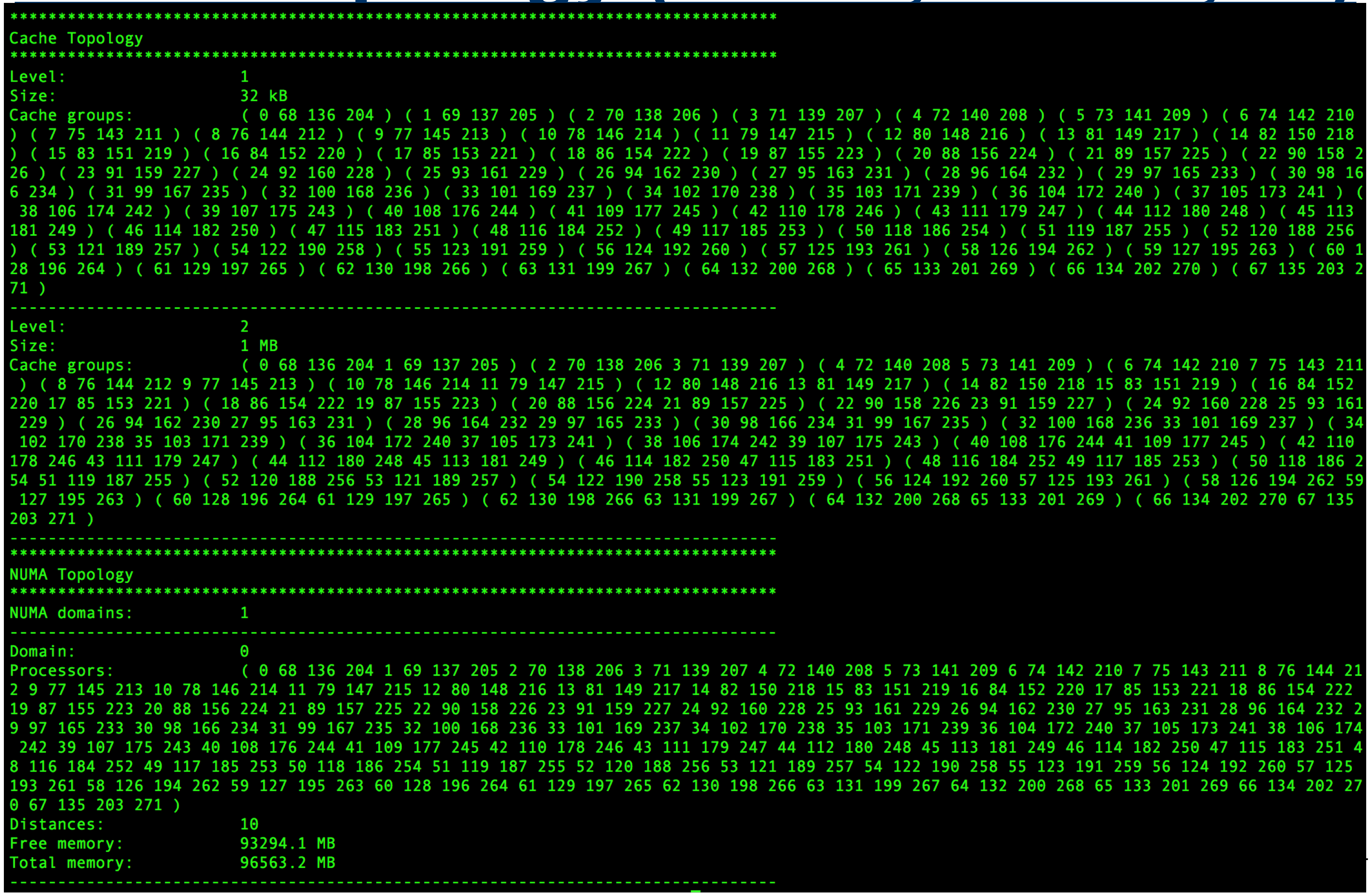
likwid-topology	node topology
likwid-pin	process/thread affinity
likwid-memsweeper	cleanup memory & LLC
likwid-powermeter	power measurements
likwid-setFrequencies	CPU/uncore frequency manipulation
likwid-perfctr	hardware counter measurements
likwid-mpirun	hardware counter + MPI
likwid-bench	micro-benchmarking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

likwid-topology



```
--------------------------------------------------------------------------------
CPU name:       Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
CPU type:       Intel Xeon Phi (Knights Landing) (Co)Processor
CPU stepping:   1
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       68
Threads per core:       4
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
4               0               4               0               *
5               0               5               0               *
6               0               6               0               *
7               0               7               0               *
8               0               8               0               *
9               0               9               0               *
10              0               10              0               *
11              0               11              0               *
12              0               12              0               *
13              0               13              0               *
14              0               14              0               *
```

likwid-topology (cache, NUMA, …)



```
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 68 136 204 ) ( 1 69 137 205 ) ( 2 70 138 206 ) ( 3 71 139 207 ) ( 4 72 140 208 ) ( 5 73 141 209 ) ( 6 74 142 210
) ( 7 75 143 211 ) ( 8 76 144 212 ) ( 9 77 145 213 ) ( 10 78 146 214 ) ( 11 79 147 215 ) ( 12 80 148 216 ) ( 13 81 149 217 ) ( 14 82 150 218
) ( 15 83 151 219 ) ( 16 84 152 220 ) ( 17 85 153 221 ) ( 18 86 154 222 ) ( 19 87 155 223 ) ( 20 88 156 224 ) ( 21 89 157 225 ) ( 22 90 158 2
26 ) ( 23 91 159 227 ) ( 24 92 160 228 ) ( 25 93 161 229 ) ( 26 94 162 230 ) ( 27 95 163 231 ) ( 28 96 164 232 ) ( 29 97 165 233 ) ( 30 98 16
6 234 ) ( 31 99 167 235 ) ( 32 100 168 236 ) ( 33 101 169 237 ) ( 34 102 170 238 ) ( 35 103 171 239 ) ( 36 104 172 240 ) ( 37 105 173 241 ) (
 38 106 174 242 ) ( 39 107 175 243 ) ( 40 108 176 244 ) ( 41 109 177 245 ) ( 42 110 178 246 ) ( 43 111 179 247 ) ( 44 112 180 248 ) ( 45 113
181 249 ) ( 46 114 182 250 ) ( 47 115 183 251 ) ( 48 116 184 252 ) ( 49 117 185 253 ) ( 50 118 186 254 ) ( 51 119 187 255 ) ( 52 120 188 256
) ( 53 121 189 257 ) ( 54 122 190 258 ) ( 55 123 191 259 ) ( 56 124 192 260 ) ( 57 125 193 261 ) ( 58 126 194 262 ) ( 59 127 195 263 ) ( 60 1
28 196 264 ) ( 61 129 197 265 ) ( 62 130 198 266 ) ( 63 131 199 267 ) ( 64 132 200 268 ) ( 65 133 201 269 ) ( 66 134 202 270 ) ( 67 135 203 2
71 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   1 MB
Cache groups:           ( 0 68 136 204 1 69 137 205 ) ( 2 70 138 206 3 71 139 207 ) ( 4 72 140 208 5 73 141 209 ) ( 6 74 142 210 7 75 143 211
 ) ( 8 76 144 212 9 77 145 213 ) ( 10 78 146 214 11 79 147 215 ) ( 12 80 148 216 13 81 149 217 ) ( 14 82 150 218 15 83 151 219 ) ( 16 84 152
220 17 85 153 221 ) ( 18 86 154 222 19 87 155 223 ) ( 20 88 156 224 21 89 157 225 ) ( 22 90 158 226 23 91 159 227 ) ( 24 92 160 228 25 93 161
 229 ) ( 26 94 162 230 27 95 163 231 ) ( 28 96 164 232 29 97 165 233 ) ( 30 98 166 234 31 99 167 235 ) ( 32 100 168 236 33 101 169 237 ) ( 34
 102 170 238 35 103 171 239 ) ( 36 104 172 240 37 105 173 241 ) ( 38 106 174 242 39 107 175 243 ) ( 40 108 176 244 41 109 177 245 ) ( 42 110
178 246 43 111 179 247 ) ( 44 112 180 248 45 113 181 249 ) ( 46 114 182 250 47 115 183 251 ) ( 48 116 184 252 49 117 185 253 ) ( 50 118 186 2
54 51 119 187 255 ) ( 52 120 188 256 53 121 189 257 ) ( 54 122 190 258 55 123 191 259 ) ( 56 124 192 260 57 125 193 261 ) ( 58 126 194 262 59
 127 195 263 ) ( 60 128 196 264 61 129 197 265 ) ( 62 130 198 266 63 131 199 267 ) ( 64 132 200 268 65 133 201 269 ) ( 66 134 202 270 67 135
203 271 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           1
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 68 136 204 1 69 137 205 2 70 138 206 3 71 139 207 4 72 140 208 5 73 141 209 6 74 142 210 7 75 143 211 8 76 144 21
2 9 77 145 213 10 78 146 214 11 79 147 215 12 80 148 216 13 81 149 217 14 82 150 218 15 83 151 219 16 84 152 220 17 85 153 221 18 86 154 222
19 87 155 223 20 88 156 224 21 89 157 225 22 90 158 226 23 91 159 227 24 92 160 228 25 93 161 229 26 94 162 230 27 95 163 231 28 96 164 232 2
9 97 165 233 30 98 166 234 31 99 167 235 32 100 168 236 33 101 169 237 34 102 170 238 35 103 171 239 36 104 172 240 37 105 173 241 38 106 174
 242 39 107 175 243 40 108 176 244 41 109 177 245 42 110 178 246 43 111 179 247 44 112 180 248 45 113 181 249 46 114 182 250 47 115 183 251 4
8 116 184 252 49 117 185 253 50 118 186 254 51 119 187 255 52 120 188 256 53 121 189 257 54 122 190 258 55 123 191 259 56 124 192 260 57 125
193 261 58 126 194 262 59 127 195 263 60 128 196 264 61 129 197 265 62 130 198 266 63 131 199 267 64 132 200 268 65 133 201 269 66 134 202 27
0 67 135 203 271 )
Distances:              10
Free memory:            93294.1 MB
Total memory:           96563.2 MB
--------------------------------------------------------------------------------
```

likwid-pin

- **`likwid-pin -c N:0,8,16,24 ./xthi.x`**
- **`likwid-pin -c S0:0,8@S1:0,8 ./xthi.x`**

```
Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
Hello from rank 0, thread 1, on nid00028. (core affinity = 8)
Hello from rank 0, thread 2, on nid00028. (core affinity = 16)
Hello from rank 0, thread 3, on nid00028. (core affinity = 24)
```

HSW

- **`likwid-pin -c E:N:128:2:4 ./xthi.x`**

```
Hello from rank 0, thread 0, on nid02308. (core affinity = 0)
Hello from rank 0, thread 1, on nid02308. (core affinity = 68)
Hello from rank 0, thread 2, on nid02308. (core affinity = 1)
Hello from rank 0, thread 3, on nid02308. (core affinity = 69)
* snip *
Hello from rank 0, thread 126, on nid02308. (core affinity = 63)
Hello from rank 0, thread 127, on nid02308. (core affinity = 131)
```

KNL

- **likwid-perfctr takes the same specification as its processor list**

Profiling with LIKWID

- likwid-perfctr (threaded) + likwid-mpirun (MPI/hybrid)

- no GUI
- low overhead -> SDE, VTune, etc
- no code instrumentation required -> CrayPat-tracing
- no root access required -> VTune
- no extra modules required to be installed -> VTune

- use Linux 'msr' module to access MSR (Model Specific Register) files

- Cori:

```
module load vtune
sbatch/salloc --perf=likwid
module load likwid
```

Profiling with LIKWID (2)

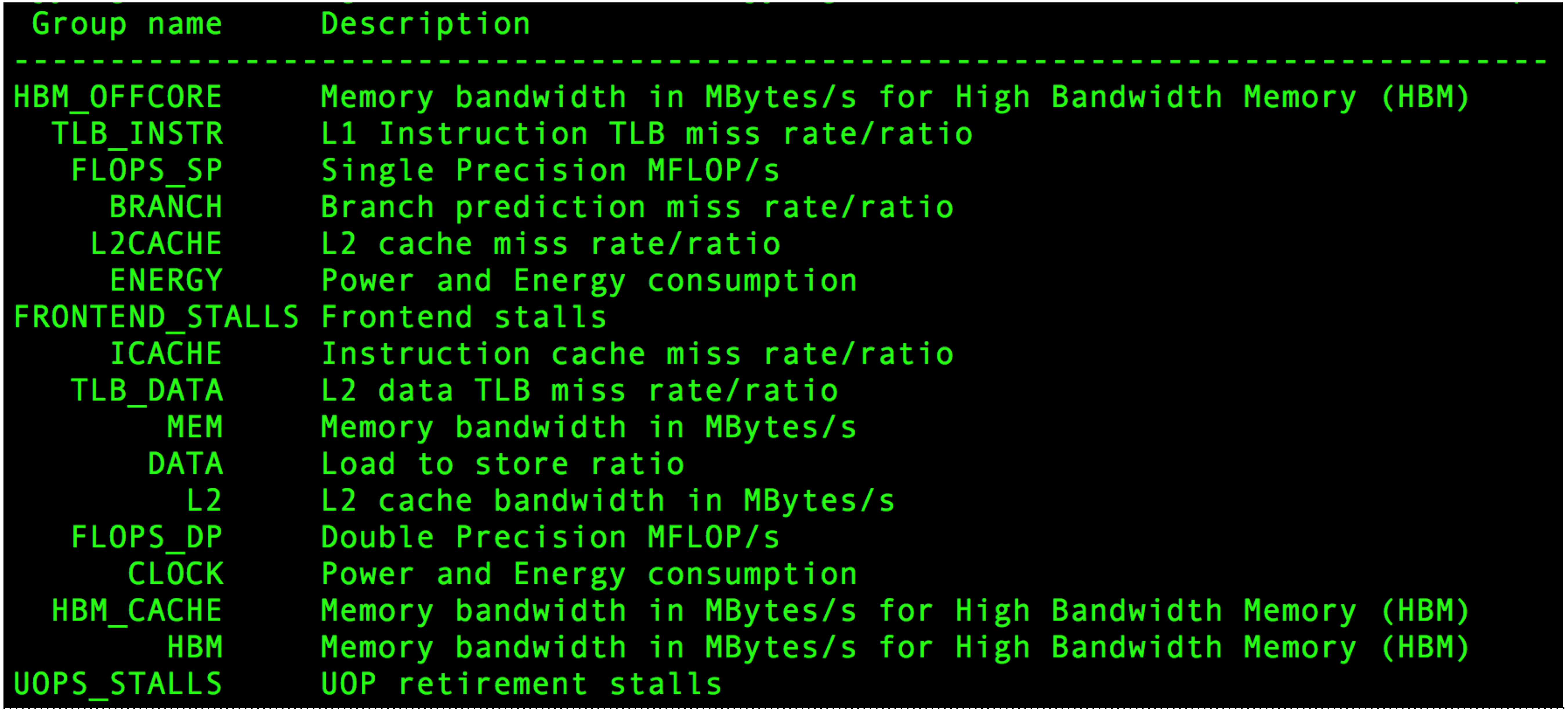
- Alternately, one can construct a script and monitor only process 0

```
srun -n8 -c32              ./a.out args
srun -n8 -c32 ./perfctr.sh ./a.out args
```

where perfctr.sh is

```
#!/bin/bash
let SLURM_MPI_RANK=$SLURM_PROCID
if [ $SLURM_MPI_RANK = 0 ];then
# only process 0 runs likwid and it monitors only logical CPUs 0-31
likwid-perfctr -C 0-31 -g CACHES $@
else
$@
fi
```

Likwid-perfctr –a (KNL)



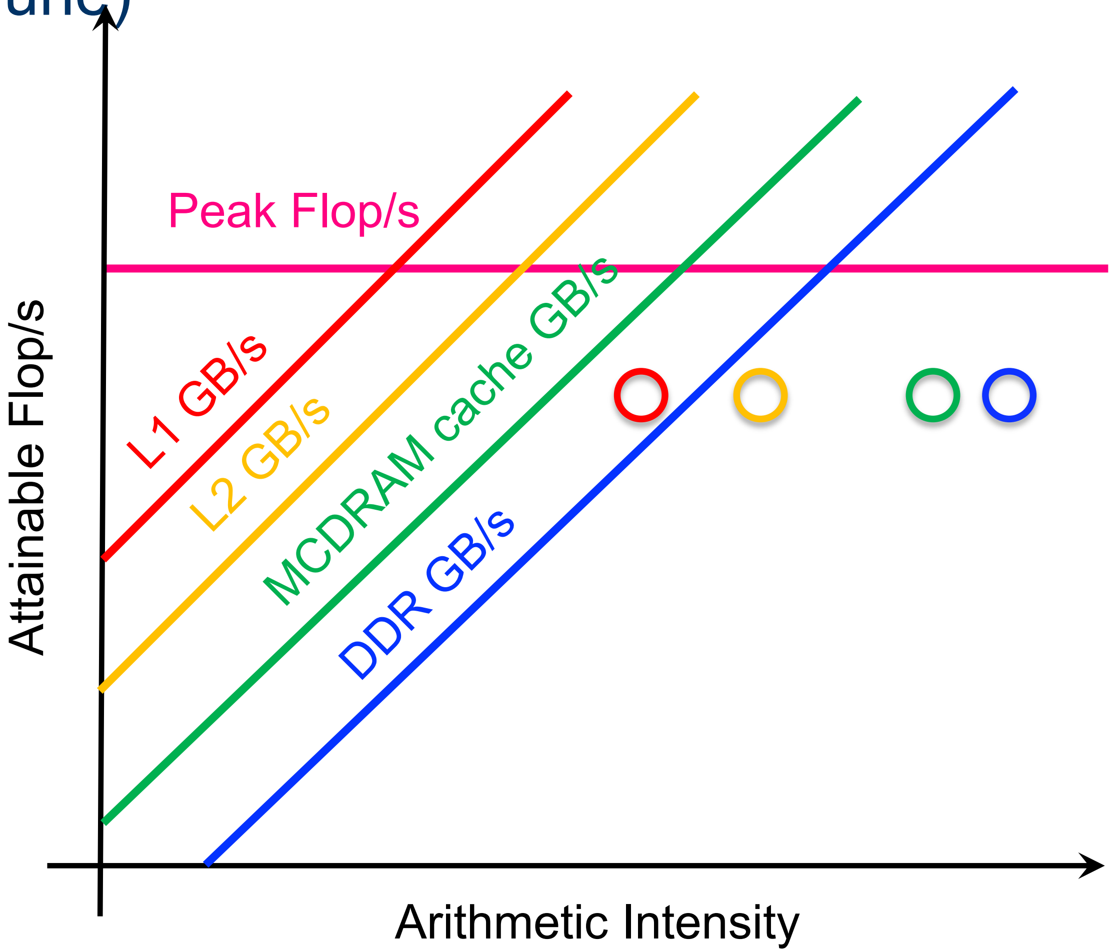
```
 Group name     Description
--------------------------------------------------------------------------------
HBM_OFFCORE     Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)
  TLB_INSTR     L1 Instruction TLB miss rate/ratio
   FLOPS_SP     Single Precision MFLOP/s
     BRANCH     Branch 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
```

Using LIKWID for Roofline

- GPP kernel from BerkeleyGW
- Arithmetic Intensity = FLOPS / Bytes (= SDE / VTune)
  = FLOPS/sec / Bytes/sec
  = **FLOPS_DP / Bandwidth**

- AI (DRAM) = FLOPS_DP / Bandwidth (DRAM)
- AI (MCDRAM) = FLOPS_DP / Bandwidth (MCDRAM)
- AI (L2) = FLOPS_DP / Bandwidth (L2)
- AI (L1) = FLOPS_DP / Bandwidth (L1)

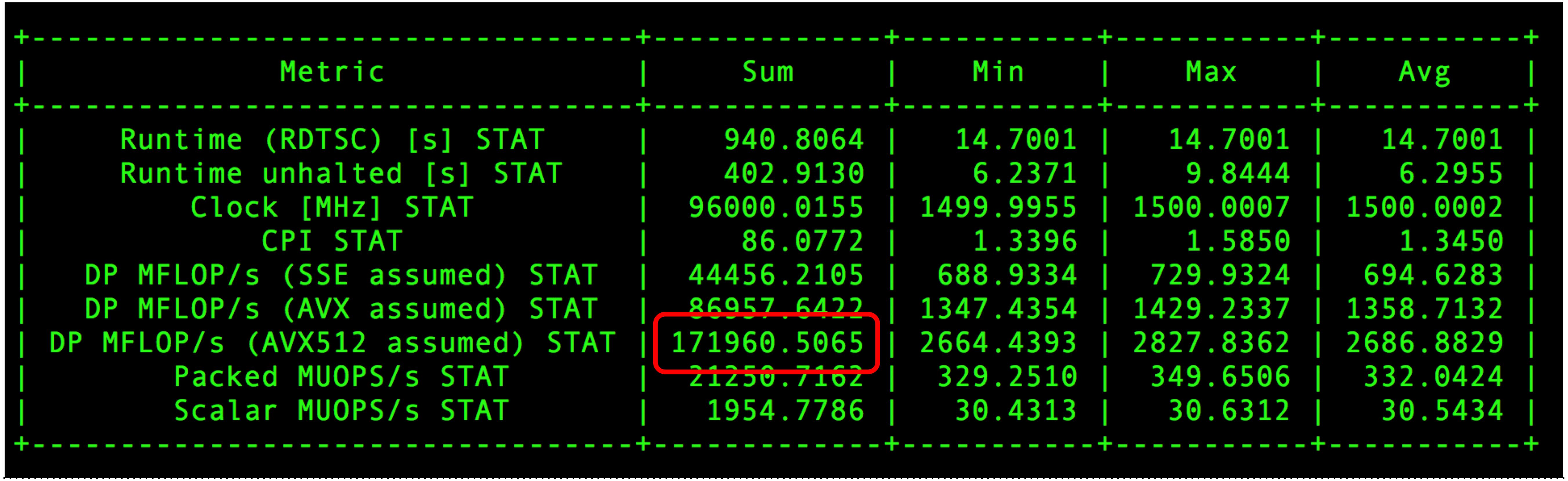
- **Performance = FLOPS_DP**



GFlop/s

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

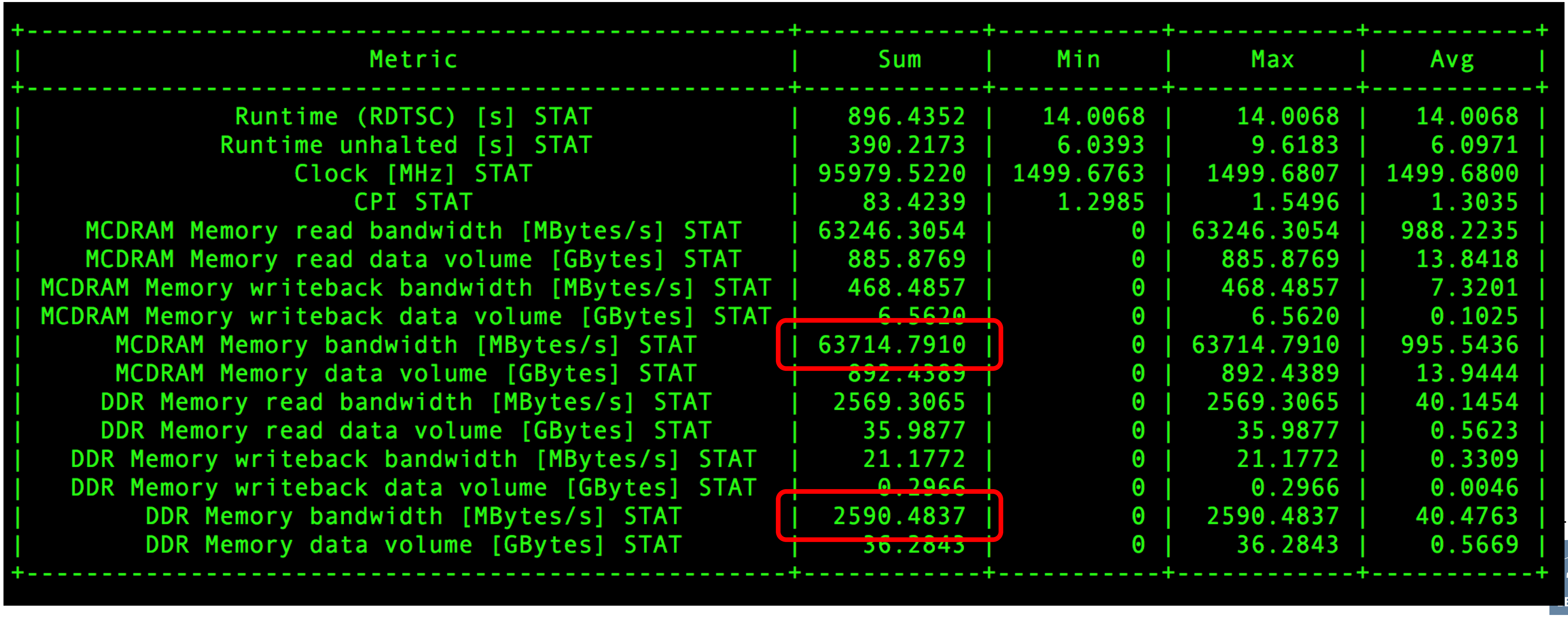
- likwid-perfctr -C 0-63 -g **FLOPS_DP** ./gpp.knl.ex 512 2 32768 20
  - 8*UOPS_RETIRED_PACKED_SIMD+UOPS_RETIRED_SCALAR_SIMD



```
+-----------------------------------+-------------+------------+------------+------------+
|              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 |
+-----------------------------------+-------------+------------+------------+------------+
```

MCDRAM and DDR GB/s

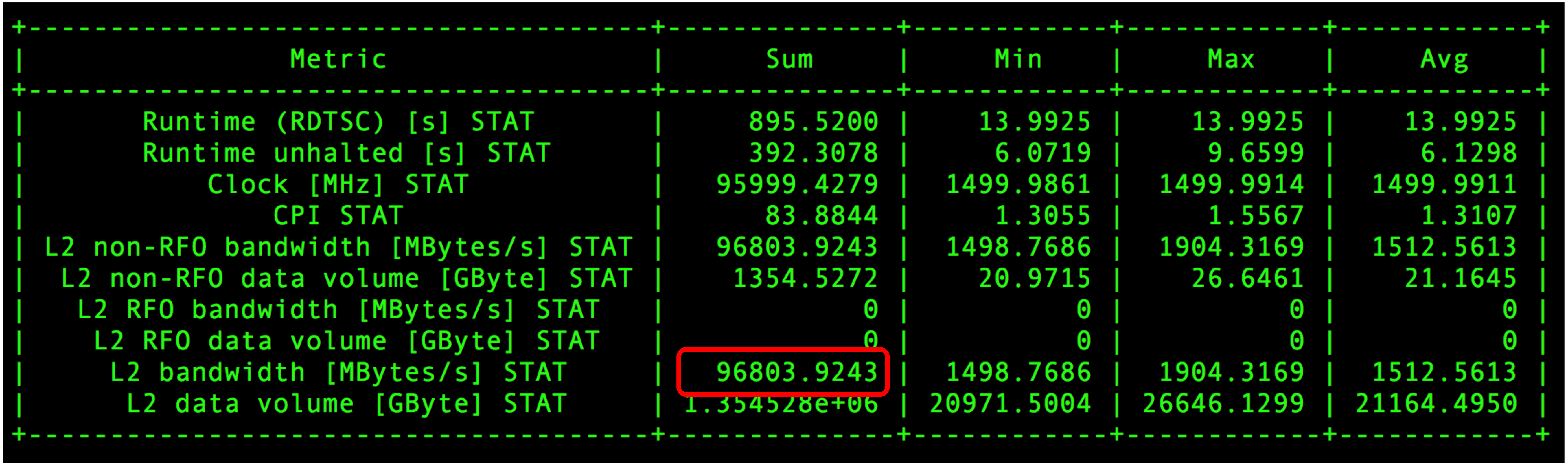
- kernel on KNL: **DDR 2.59GB/s + MCDRAM 63.71GB/s**
  - MC_CAS_READS/ MC_CAS_WRITES
  - EDC_RPQ_INSERTS/ EDC_WPQ_INSERTS
  - EDC_MISS_CLEAN/ EDC_MISS_DIRTY
- likwid-perfctr -C 0-63 -g **HBM_CACHE** ./gpp.knl.ex 512 2 32768 20



```
+----------------------------------------------------+------------+-----------+------------+-----------+
|                       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 |
+----------------------------------------------------+------------+-----------+------------+-----------+
```

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 |
+---------------------------------------+--------------+-------------+-------------+-------------+
```

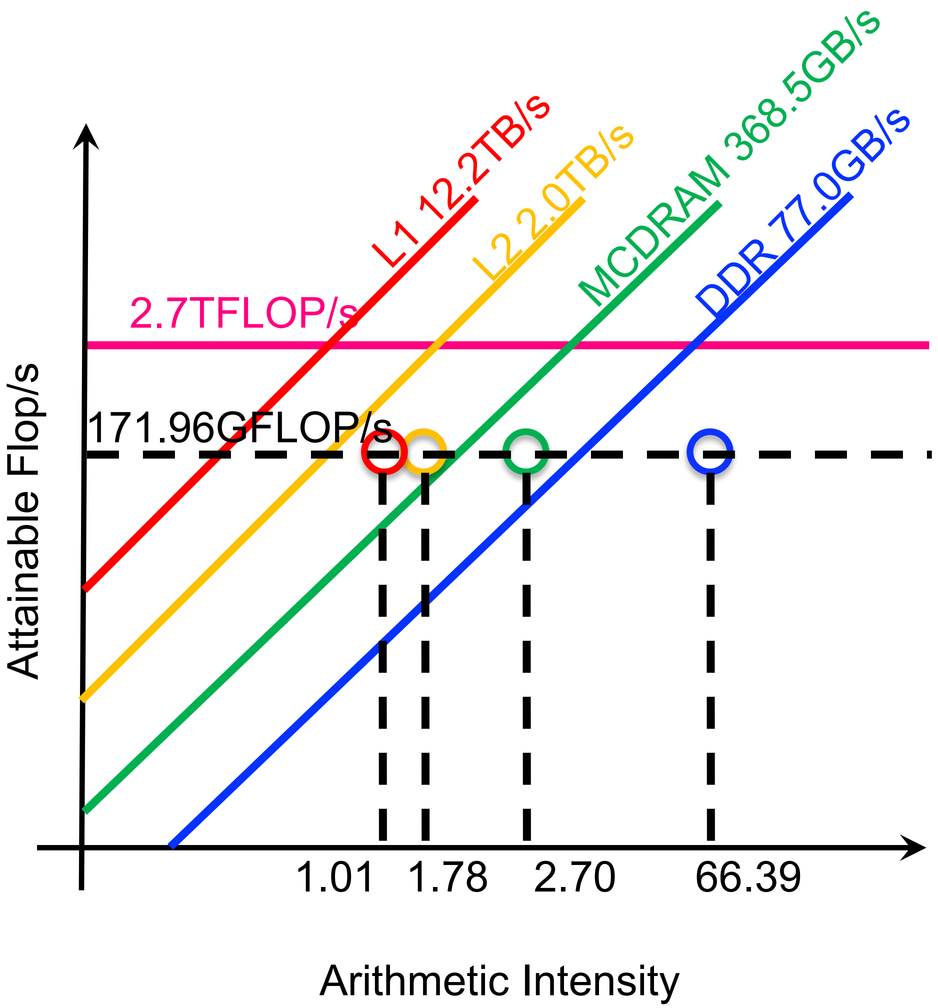
L1 GB/s

- 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)

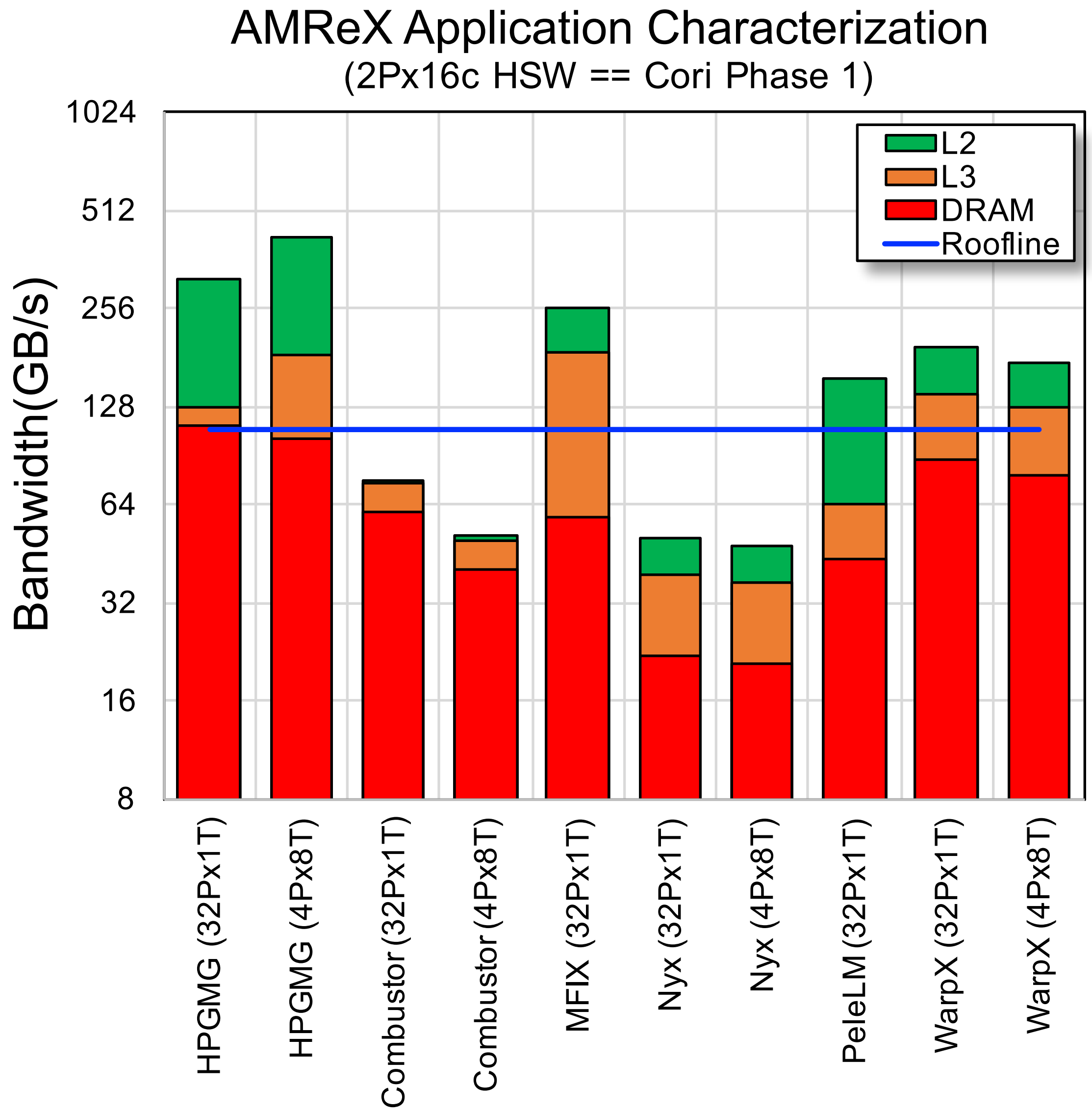
Resultant Roofline

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



LIKWID on AMReX apps

- Used LIKWID to characterize AMReX applications
- Measured cache and DRAM bytes.
  - Averaged over 30min executions and 32 processes
  - Only 2 applications (not counting HPGMG proxy) used >50% of memory bandwidth on average
  - Used this data to estimate average AI for each level of the memory hierarchy
  - Used this data to infer requisite cache tapering



Likwid-mpirun

- `srun -n 2 -c 32 --cpu-bind=cores likwid-perfctr -C 0,8 -g MEM -o test_%h_%p_%r.txt ./xthi.x`

%h	hostname
%p	process ID
%r	MPI rank

- `likwid-mpirun -pin S0:0,8_S1:0,8 -g MEM ./xthi.x`

HSW

```
Hello from rank 0, thread 0, on nid00191. (core affinity = 0)
Hello from rank 0, thread 1, on nid00191. (core affinity = 8)
Hello from rank 1, thread 0, on nid00191. (core affinity = 16)
Hello from rank 1, thread 1, on nid00191. (core affinity = 24)
```

- Uncore counters are measured on a per-socket basis

Marking Specific Regions

```
#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

- `cc -qopenmp -DLIKWID_PERFMON -I$LIKWID_INCLUDE -L$LIKWID_LIB -llikwid -dynamic test.c -o test.x`
- `likwid-perfctr -C 0-3 -g MEM -m ./test.x`

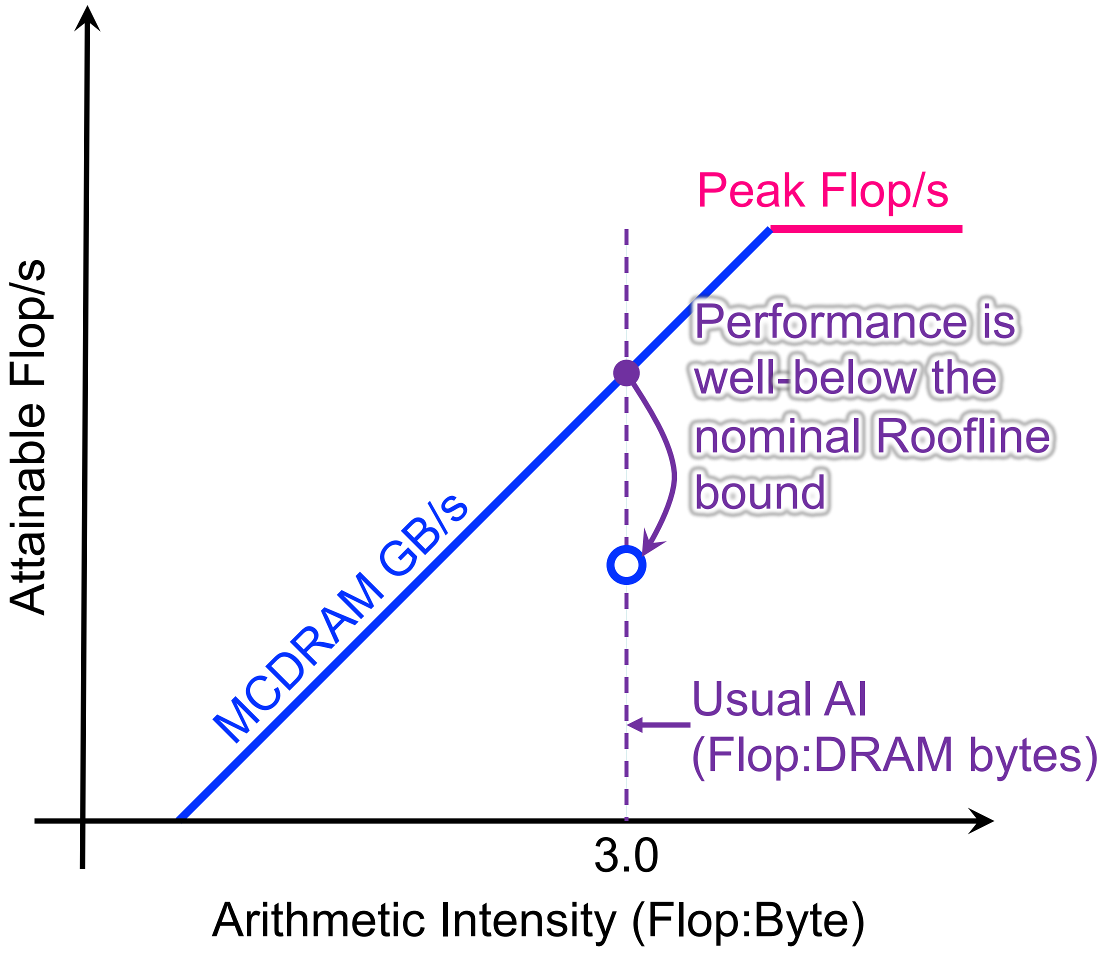
FLOP Roofline vs. VUOP Roofline

- Nominally, Roofline is based on Flop/s, GB/s, and Flop/Byte
- Such metrics make sense from the user perspective.

- On SIMD machines, one might consider vuop/s instead of flop/s
  - ✓ vuop/s (scalar + vector) can easily be mapped to vector unit utilization
  - ✓ 100% vector unit utilization can bottleneck performance
  - ✓ Performance counters give vuop/s and not flop/s
  - ✗ 100% vector unit utilization does not imply 100% of peak (FMA, scalar vs. vector)

FLOP Roofline

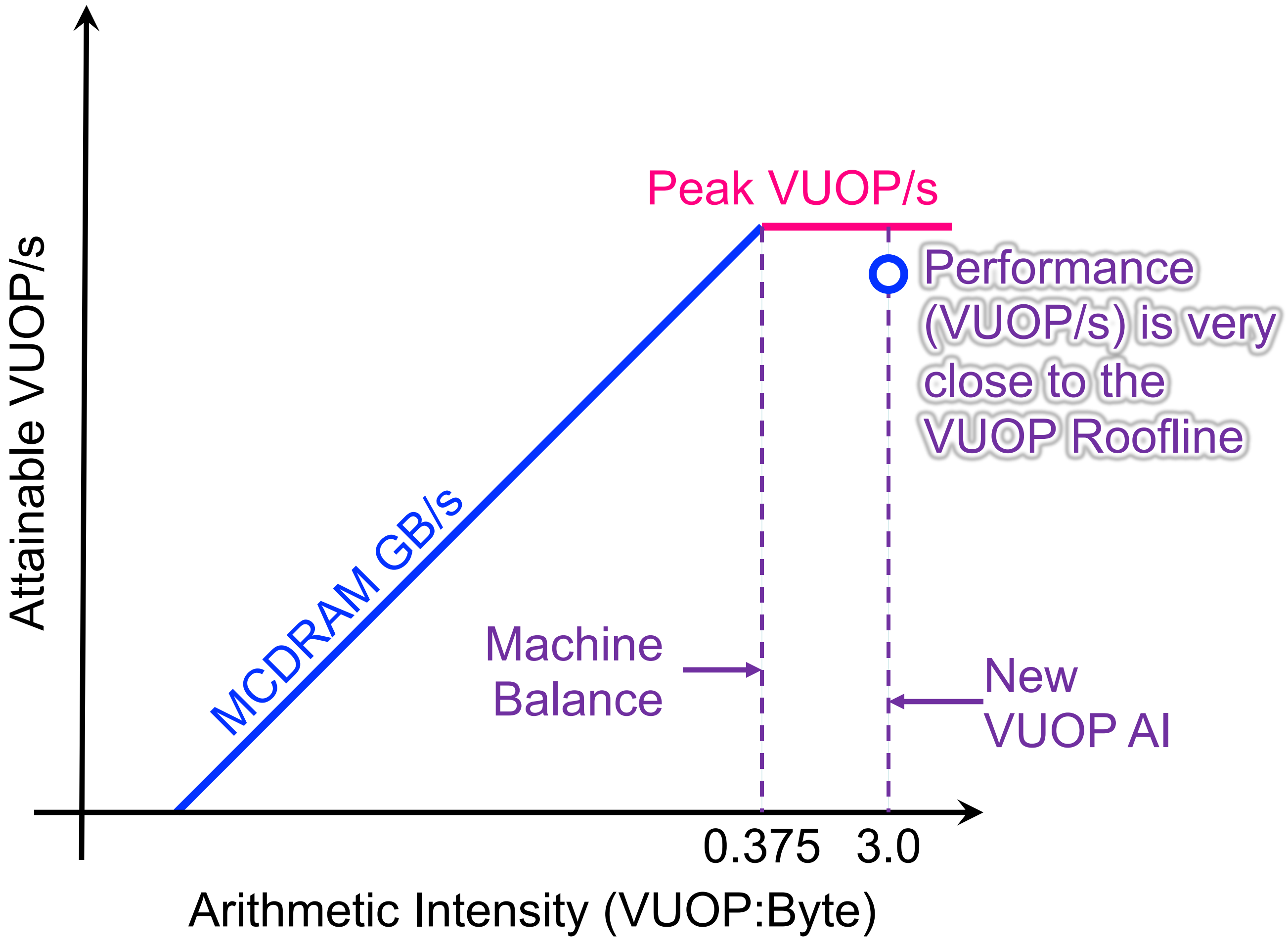
- With performance counters alone, its hard to deduce why performance is well-below the FLOP Roofline.
  - VL?
  - Precision?
  - FMA?
  - Masks?
  - Non-FP vector instructions
- Moreover, one might conclude a code is memory bound when in reality is compute-bound



VUOP Roofline

- In a VUOP KNL Roofline
  - machine peak (VUOP/s) is 16x lower
  - machine balance is 16x lower (0.375)
- Consider an example where all flops are scalar adds (VADDSD)
  - 1 FLOP / VUOP
  - AI = 3 FLOPs/Byte = **3 VUOPS/Byte**
- Although FLOP/s was far from its Roofline, VUOP/s is 16x closer to its peak

➢ **Need source code analysis to understand VL, FMA, … issues**



FLOP Roofline

- Use of FMA doesn't change Arithmetic Intensity (FMA == FMUL+FADD == 2 FLOPs)
- Use of SIMD doesn't change Arithmetic Intensity
- Presence of vector integer operations doesn't change Arithmetic Intensity
- Moving from 64b to 32b data types **doubles** AI
- High fraction of Roofline implies high performance

VUOP Roofline

- Use of FMA cuts Arithmetic Intensity in **half** (half the number of VUOPS)
- Use of SIMD reduces Arithmetic Intensity by a factor of Vector Length (**e.g. cuts number of VUOPS by 8x**)
- Presence of vector integer operations **increases** Arithmetic Intensity
- Moving from 64b to 32b data types doesn't change Arithmetic Intensity
- High fraction of Roofline implies high vector unit utilization (**but not necessarily high performance**)

Roofline with SDE

Why isn’t LIKWID good enough?

- LIKWID counts vector uops
- KNL vuop counters aren’t…
    - VL-aware
    - precision-aware
    - mask-aware
    - FMA-aware
- Counters don’t differentiate instruction types (FP, int, shuffle, …)
- **Flop counters were broken on Haswell.**
- Thus, LIKWID might be a good starting point, but its not perfect.

➢ **Need tools that actually count flops correctly and ones that can be used to understand nuances of instruction mixes.**

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

https://software.intel.com/en-us/articles/intel-software-development-emulator

Compiling with SDE at NERSC

- Makefile…

```
MPICC = cc
CFLAGS = -g -O3 -dynamic -qopenmp -restrict -qopt-streaming-stores always  \
         -DSTREAM_ARRAY_SIZE=400000000 -DNTIMES=50 \
         -I$(VTUNE_AMPLIFIER_XE_2018_DIR)/include
LDFLAGS = -L$(VTUNE_AMPLIFIER_XE_2018_DIR)/lib64 -littnotify

stream_mpi.exe: stream_mpi.c Makefile
      $(MPICC) $(CFLAGS) stream_mpi.c -o stream_mpi.exe $(LDFLAGS)

clean:
      rm -f stream_mpi.exe
```

- module load sde
  make

https://bitbucket.org/dwdoerf/stream-ai-example.git

Running with SDE at NERSC

```
srun -n 4 -c 6 sde -ivb -d -iform 1 -omix
  my_mix.out -i -global_region -start_ssc_mark
  111:repeat -stop_ssc_mark 222:repeat -- foo.exe
```

- -ivb is used to target Edison's Ivy Bridge ISA (for Cori use -hsw for Haswell or -knl for KNL processors)
- -d specifies to only collect dynamic profile information
- -iform 1 turns on compute ISA iform mix
- -omix specifies the output file (and turns on -mix)
- -i specifies that each process will have a unique file name based on process ID (needed for MPI)
- -global_region will include any threads spawned by a process (needed for OpenMP)

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

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
```

Marking Regions of Interest for SDE

```
// Code must be built with appropriate paths for VTune include file (ittnotify.h) and
   library (-littnotify)
   #include <ittnotify.h>

   __SSC_MARK(0x111); // start SDE tracing, note it uses 2 underscores
   __itt_resume();    // start VTune, again use 2 underscores

   for (k=0; k<NTIMES; k++) {
   #pragma omp parallel for
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
   a[j] = b[j]+scalar*c[j];
   }

   __itt_pause();     // stop VTune
   __SSC_MARK(0x222); // stop SDE tracing
```

!

Essential when analyzing Individual kernels

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

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	error
2527.81 GFLOPS	~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.summary`
  - `./parse-vtune.sh my_vtune.summary`
  - DDR --->Total Bytes = 35983553088
  - HBM --->Total Bytes = 963486016448

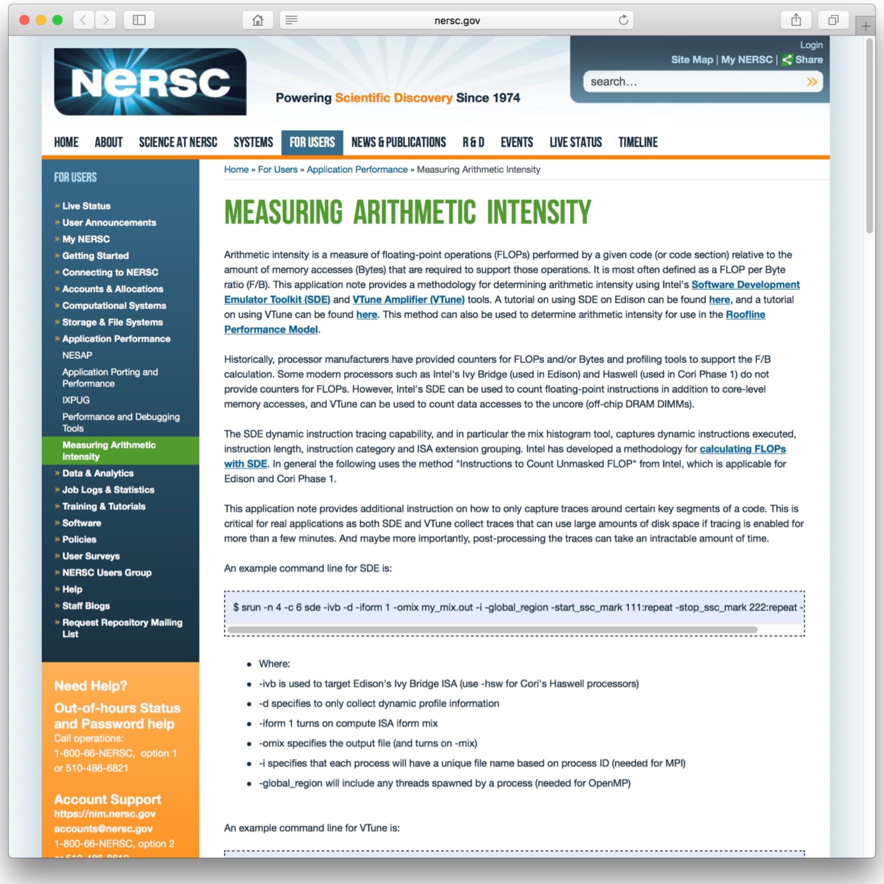
LIKWID	error
DDR: 36.28 GB	~0.8%
HBM: 892.44 GB	~7.4%

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

Roofline with LIKWID + SDE

Initially Cobbled Together Tools…

- Use tools known/observed to work on NERSC's Cori (KNL, HSW)…
  - Used **Intel SDE** (Pin binary instrumentation + emulation) to create software Flop counters
  - Used **Intel VTune** performance tool (NERSC/Cray approved) to access uncore counters
- ➢ Accurate measurement of Flop's (HSW) and DRAM data movement (HSW and KNL)
- ➢ Used by NESAP (NERSC KNL application readiness project) to characterize apps on Cori…



MEASURING ARITHMETIC INTENSITY

Arithmetic intensity is a measure of floating-point operations (FLOPs) performed by a given code (or code section) relative to the amount of memory accesses (Bytes) that are required to support those operations. It is most often defined as a FLOP per Byte ratio (F/B). This application note provides a methodology for determining arithmetic intensity using Intel's **Software Development Emulator Toolkit (SDE)** and **VTune Amplifier (VTune)** tools. A tutorial on using SDE on Edison can be found **here**, and a tutorial on using VTune can be found **here**. This method can also be used to determine arithmetic intensity for use in the **Roofline Performance Model**.

Historically, processor manufacturers have provided counters for FLOPs and/or Bytes and profiling tools to support the F/B calculation. Some modern processors such as Intel's Ivy Bridge (used in Edison) and Haswell (used in Cori Phase 1) do not provide counters for FLOPs. However, Intel's SDE can be used to count floating-point instructions in addition to core-level memory accesses, and VTune can be used to count data accesses to the uncore (off-chip DRAM DIMMs).

The SDE dynamic instruction tracing capability, and in particular the mix histogram tool, captures dynamic instructions executed, instruction length, instruction category and ISA extension grouping. Intel has developed a methodology for **calculating FLOPs with SDE**. In general the following uses the method "Instructions to Count Unmasked FLOP" from Intel, which is applicable for Edison and Cori Phase 1.

This application note provides additional instruction on how to only capture traces around certain key segments of a code. This is critical for real applications as both SDE and VTune collect traces that can use large amounts of disk space if tracing is enabled for more than a few minutes. And maybe more importantly, post-processing the traces can take an intractable amount of time.

An example command line for SDE is:

```
$ srun -n 4 -c 6 sde -ivb -d -iform 1 -omix my_mix.out -i -global_region -start_ssc_mark 111:repeat -stop_ssc_mark 222:repeat -
```

- Where:
- -ivb is used to target Edison's Ivy Bridge ISA (use -hsw for Cori's Haswell processors)
- -d specifies to only collect dynamic profile information
- -iform 1 turns on compute ISA iform mix
- -omix specifies the output file (and turns on -mix)
- -i specifies that each process will have a unique file name based on process ID (needed for MPI)
- -global_region will include any threads spawned by a process (needed for OpenMP)

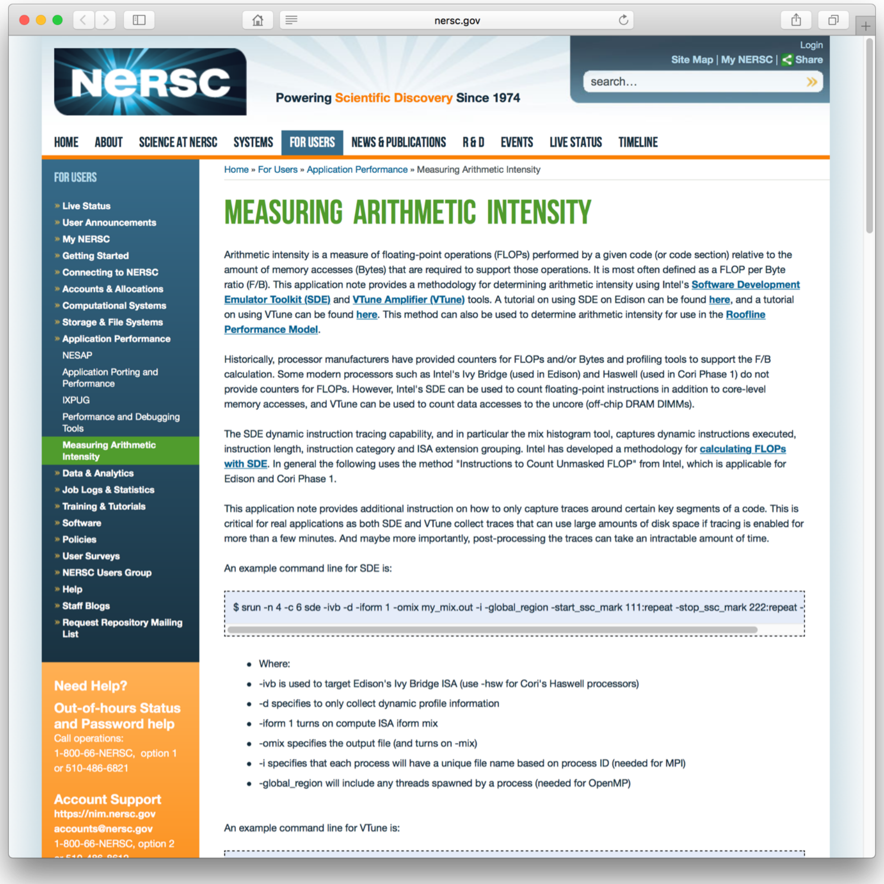
An example command line for VTune is:

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

NERSC is LBL's production computing division
CRD is LBL's Computational Research Division
NESAP is NERSC's KNL application readiness project
LBL is part of SUPER (DOE SciDAC3 Computer Science Institute)

More Recently…

- Use tools known/observed to work on NERSC's Cori (KNL, HSW)…
  - Used **Intel SDE** (Pin binary instrumentation + emulation) to create software Flop counters
  - Used **LIKWID** performance counter tool (NERSC/Cray approved) to access uncore counters
- ➢ Accurate measurement of Flop's (HSW) and DRAM data movement (HSW and KNL)
- ➢ Used by NESAP (NERSC KNL application readiness project) to characterize apps on Cori…



Home » For Users » Application Performance » Measuring Arithmetic Intensity

MEASURING ARITHMETIC INTENSITY

Arithmetic intensity is a measure of floating-point operations (FLOPs) performed by a given code (or code section) relative to the amount of memory accesses (Bytes) that are required to support those operations. It is most often defined as a FLOP per Byte ratio (F/B). This application note provides a methodology for determining arithmetic intensity using Intel's Software Development Emulator Toolkit (SDE) and VTune Amplifier (VTune) tools. A tutorial on using SDE on Edison can be found here, and a tutorial on using VTune can be found here. This method can also be used to determine arithmetic intensity for use in the Roofline Performance Model.

Historically, processor manufacturers have provided counters for FLOPs and/or Bytes and profiling tools to support the F/B calculation. Some modern processors such as Intel's Ivy Bridge (used in Edison) and Haswell (used in Cori Phase 1) do not provide counters for FLOPs. However, Intel's SDE can be used to count floating-point instructions in addition to core-level memory accesses, and VTune can be used to count data accesses to the uncore (off-chip DRAM DIMMs).

The SDE dynamic instruction tracing capability, and in particular the mix histogram tool, captures dynamic instructions executed, instruction length, instruction category and ISA extension grouping. Intel has developed a methodology for calculating FLOPs with SDE. In general the following uses the method "Instructions to Count Unmasked FLOP" from Intel, which is applicable for Edison and Cori Phase 1.

This application note provides additional instruction on how to only capture traces around certain key segments of a code. This is critical for real applications as both SDE and VTune collect traces that can use large amounts of disk space if tracing is enabled for more than a few minutes. And maybe more importantly, post-processing the traces can take an intractable amount of time.

An example command line for SDE is:

```
$ srun -n 4 -c 6 sde -ivb -d -iform 1 -omix my_mix.out -i -global_region -start_ssc_mark 111:repeat -stop_ssc_mark 222:repeat -
```

- Where:
- -ivb is used to target Edison's Ivy Bridge ISA (use -hsw for Cori's Haswell processors)
- -d specifies to only collect dynamic profile information
- -iform 1 turns on compute ISA iform mix
- -omix specifies the output file (and turns on -mix)
- -i specifies that each process will have a unique file name based on process ID (needed for MPI)
- -global_region will include any threads spawned by a process (needed for OpenMP)

An example command line for VTune is:

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

NERSC is LBL's production computing division
CRD is LBL's Computational Research Division
NESAP is NERSC's KNL application readiness project
LBL is part of SUPER (DOE SciDAC3 Computer Science Institute)

BERKELEY LAB

Hierarchical Roofline vs. Cache-Aware Roofline

…understanding different Roofline formulations in Intel Advisor

There are two Major Roofline Formulations:

- Hierarchical Roofline (original Roofline w/ DRAM, L3, L2, …)…
  - **Williams, et al, “Roofline: An Insightful Visual Performance Model for Multicore Architectures”, CACM, 2009**
  - **Chapter 4 of “Auto-tuning Performance on Multicore Computers”, 2008**
  - Defines multiple bandwidth ceilings and multiple AI’s per kernel
  - Performance bound is the minimum of flops and the memory intercepts (superposition of original, single-metric Rooflines)
- Cache-Aware Roofline
  - **Ilic et al, "Cache-aware Roofline model: Upgrading the loft", IEEE Computer Architecture Letters, 2014**
  - Defines multiple bandwidth ceilings, but uses a single AI (flop:L1 bytes)
  - As one looses cache locality (capacity, conflict, …) performance falls from one BW ceiling to a lower one at constant AI
- Why Does this matter?
  - Some tools use the Hierarchical Roofline, some use cache-aware **== Users need to understand the differences**
  - Cache-Aware Roofline model was integrated into production Intel Advisor
  - Evaluation version of Hierarchical Roofline[1] (cache simulator) has also been integrated into Intel Advisor

---

[1] Experimental Feature, the look and feel and exact behavior is subject for change

Hierarchical Roofline

- Captures cache effects
- AI is Flop:Bytes after being ***filtered by lower cache levels***
- Multiple Arithmetic Intensities

  (one per level of memory)
- AI ***dependent*** on problem size

  (capacity misses reduce AI)
- Memory/Cache/Locality effects are ***observed as decreased AI***
- Requires ***performance counters or cache simulator*** to correctly measure AI

Cache-Aware Roofline

- Captures cache effects
- AI is Flop:Bytes ***as presented to the L1 cache (plus non-temporal stores)***
- Single Arithmetic Intensity
- AI ***independent*** of problem size
- Memory/Cache/Locality effects are ***observed as decreased performance***
- Requires static analysis or ***binary instrumentation*** to measure AI

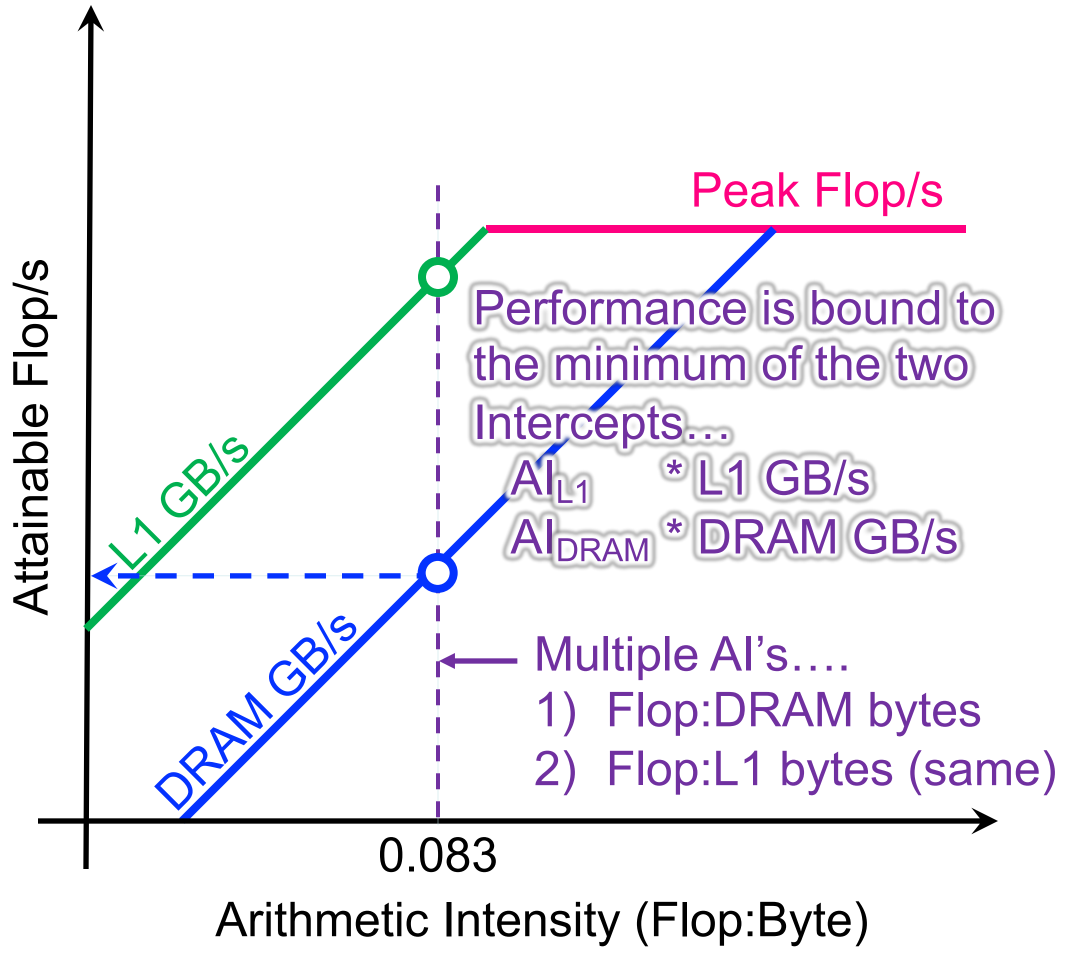
Example: STREAM

- L1 AI…
  - 2 flops
  - 2 x 8B load (old)
  - 1 x 8B store (new)
  - = 0.08 flops per byte
- No cache reuse…
  - Iteration i doesn’t touch any data associated with iteration i+delta for any delta.
- … leads to a DRAM AI equal to the L1 AI

```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

Example: STREAM

Hierarchical Roofline



Performance is bound to the minimum of the two Intercepts…

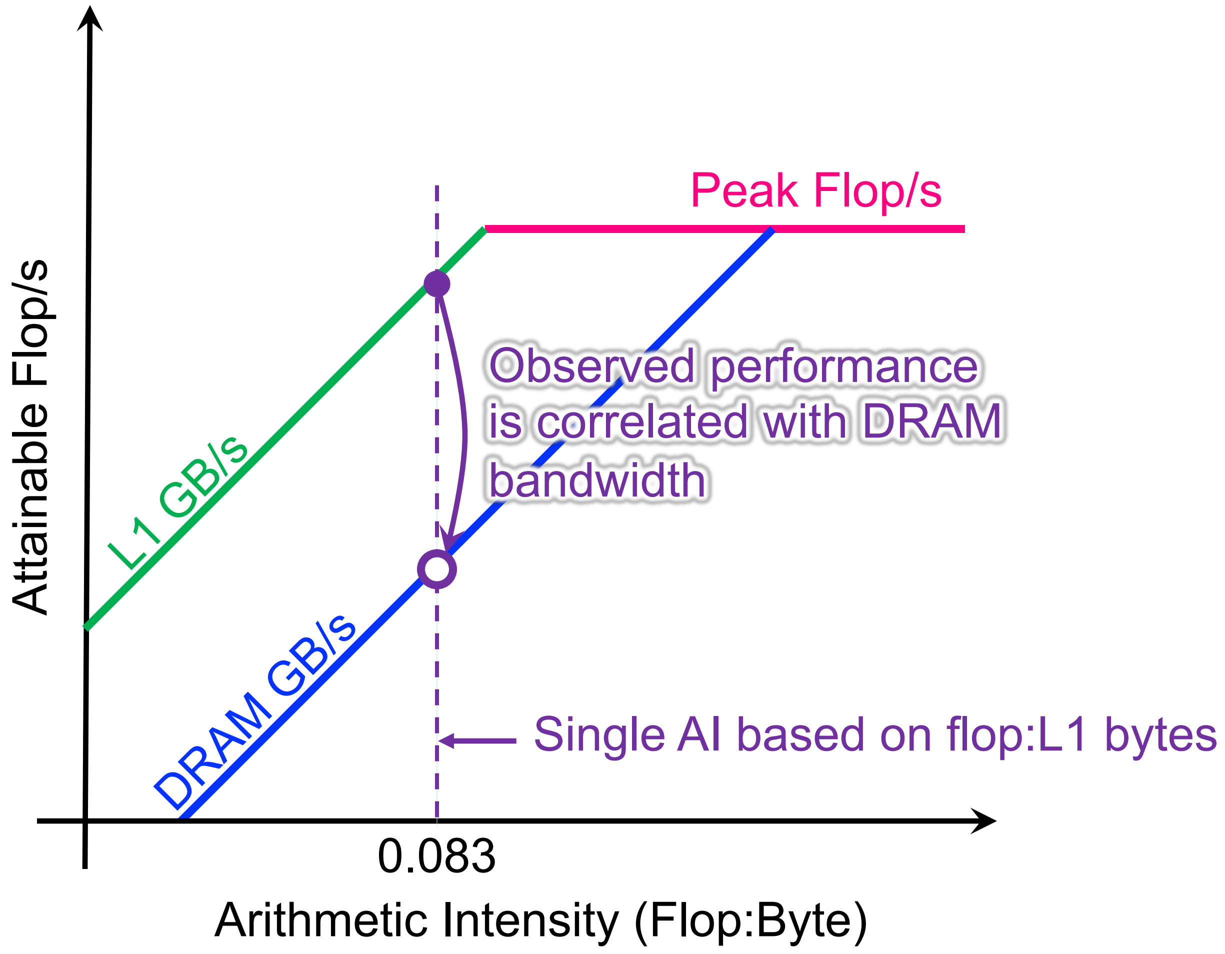
AI_{L1} * L1 GB/s

AI_{DRAM} * DRAM GB/s

Multiple AI's….

1) Flop:DRAM bytes
2) Flop:L1 bytes (same)

Cache-Aware Roofline



Observed performance is correlated with DRAM bandwidth

Single AI based on flop:L1 bytes

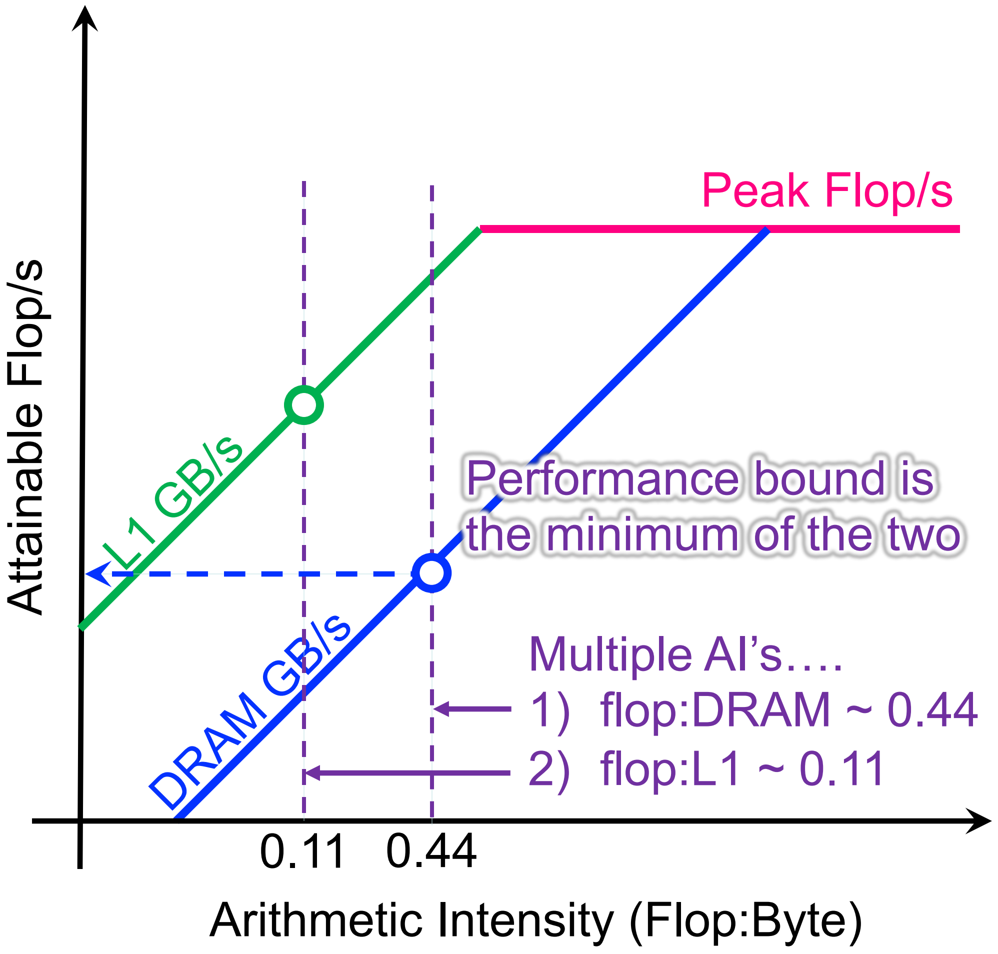
Example: 7-point Stencil (Small Problem)

- L1 AI…
  - 7 flops
  - 7 x 8B load (old)
  - 1 x 8B store (new)
  - = 0.11 flops per byte
  - some compilers may do register shuffles to reduce the number of loads.
- Moderate cache reuse…
  - `old[k][j][i+1]` is reused on next iteration of i.
  - `old[k][j+1][i]` is reused on next iteration of j.
  - `old[k+1][j][i]` is reused on next iterations of k.
- … leads to DRAM AI larger than the L1 AI

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  new[k][j][i] = -6.0*old[k  ][j  ][i  ]
                    + old[k  ][j  ][i-1]
                    + old[k  ][j  ][i+1]
                    + old[k  ][j-1][i  ]
                    + old[k  ][j+1][i  ]
                    + old[k-1][j  ][i  ]
                    + old[k+1][j  ][i  ];
}}}
```

Example: 7-point Stencil (Small Problem)

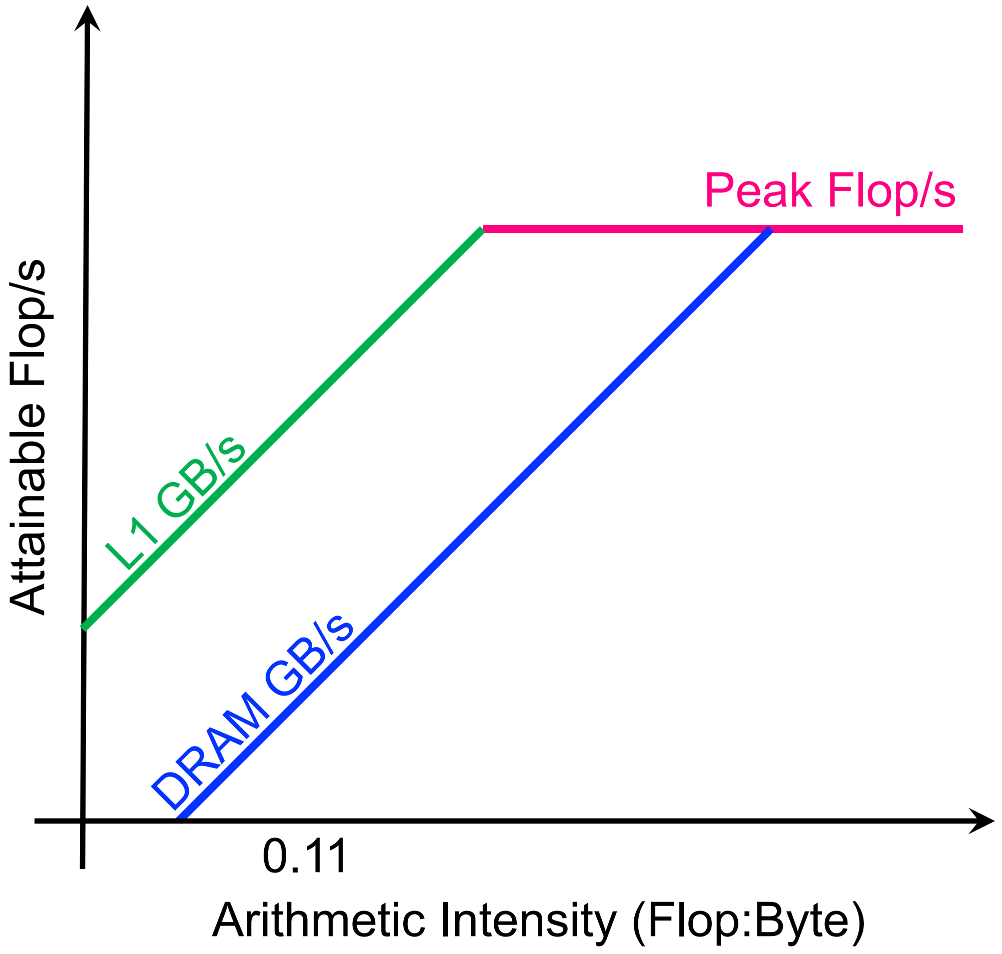
Hierarchical Roofline



Performance bound is the minimum of the two

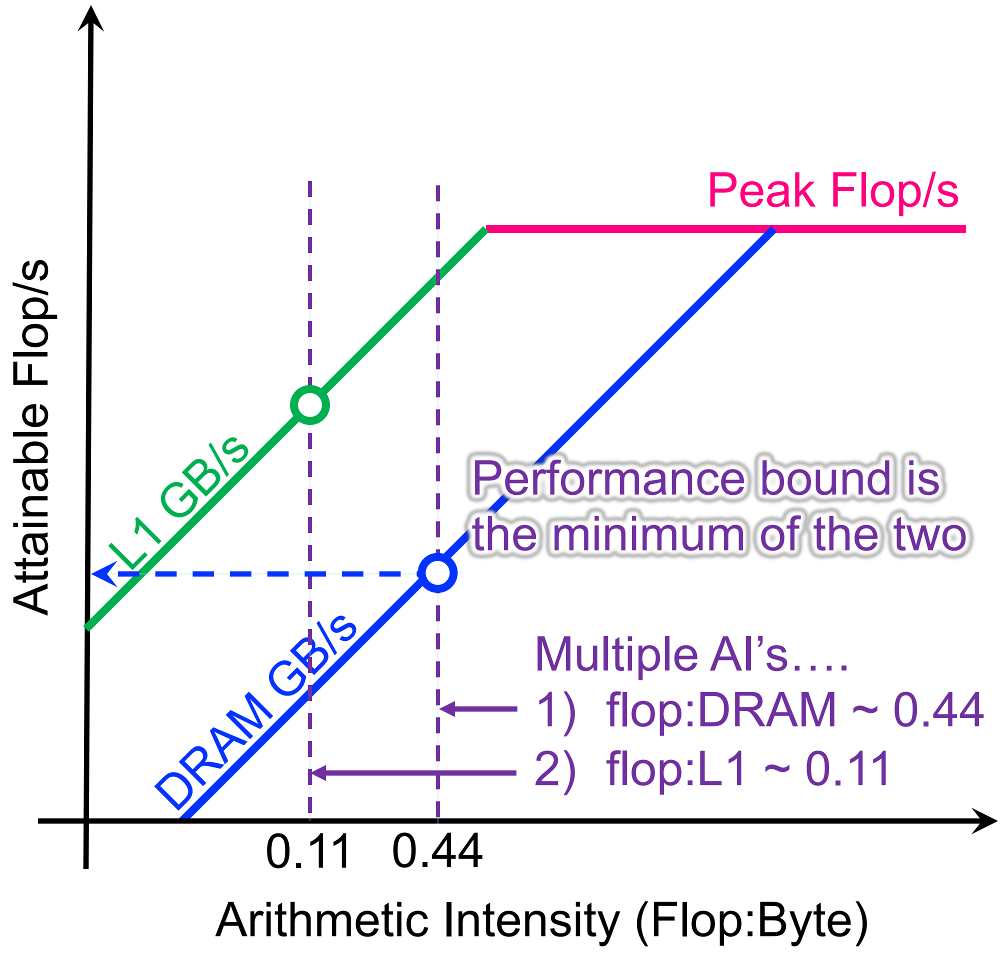
Multiple AI's….
1) flop:DRAM ~ 0.44
2) flop:L1 ~ 0.11

Cache-Aware Roofline



Example: 7-point Stencil (Small Problem)

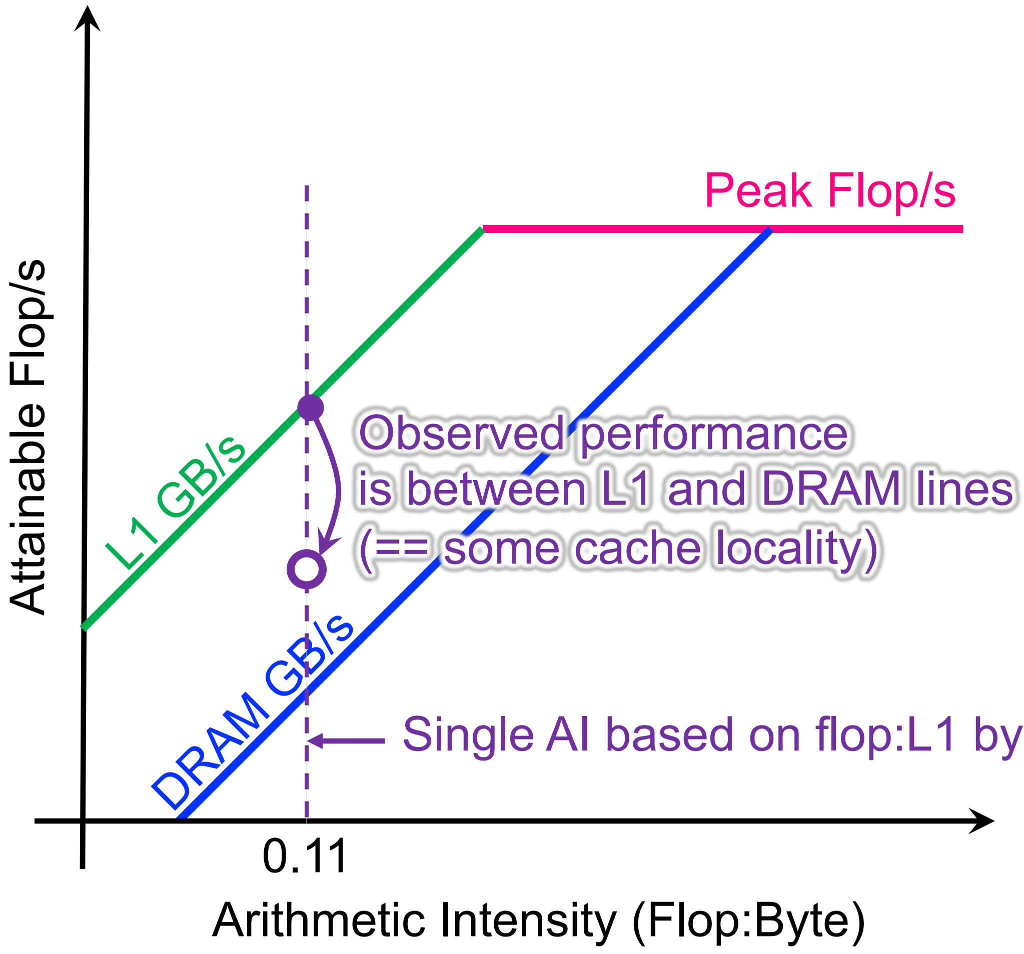
Hierarchical Roofline



Performance bound is the minimum of the two

Multiple AI's….
1) flop:DRAM ~ 0.44
2) flop:L1 ~ 0.11

Cache-Aware Roofline

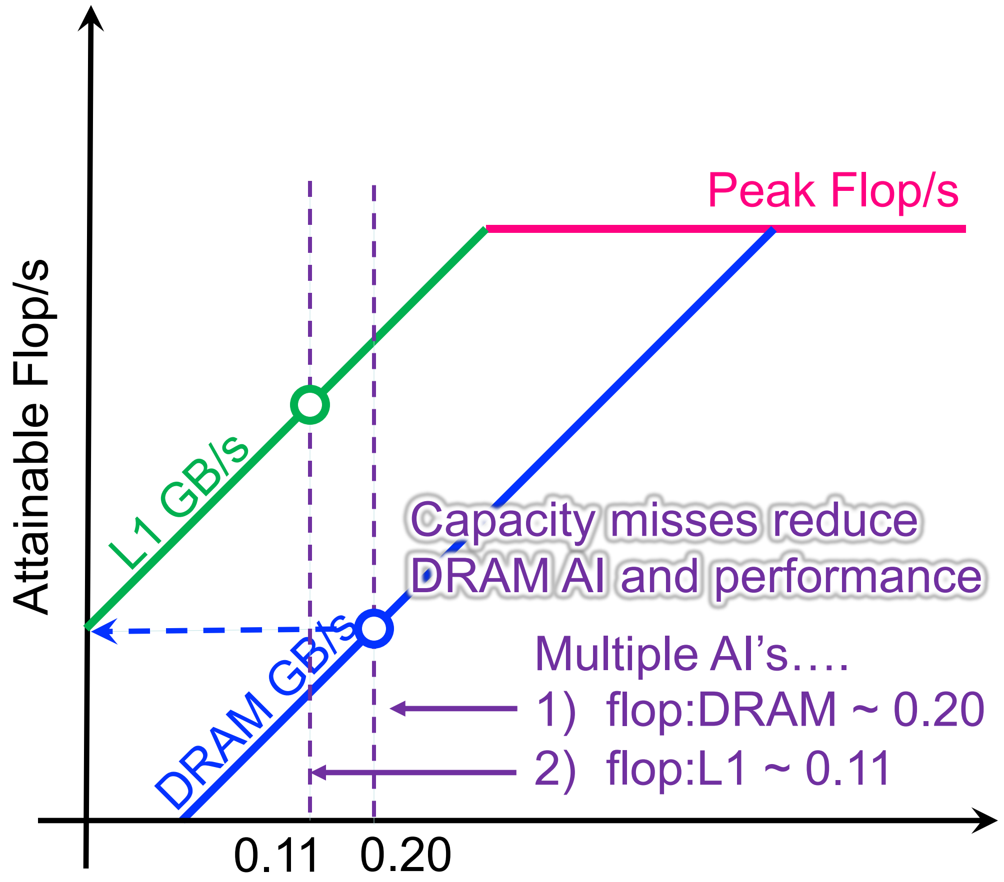


Observed performance is between L1 and DRAM lines (== some cache locality)

Single AI based on flop:L1 bytes

Example: 7-point Stencil (Large Problem)

Hierarchical Roofline

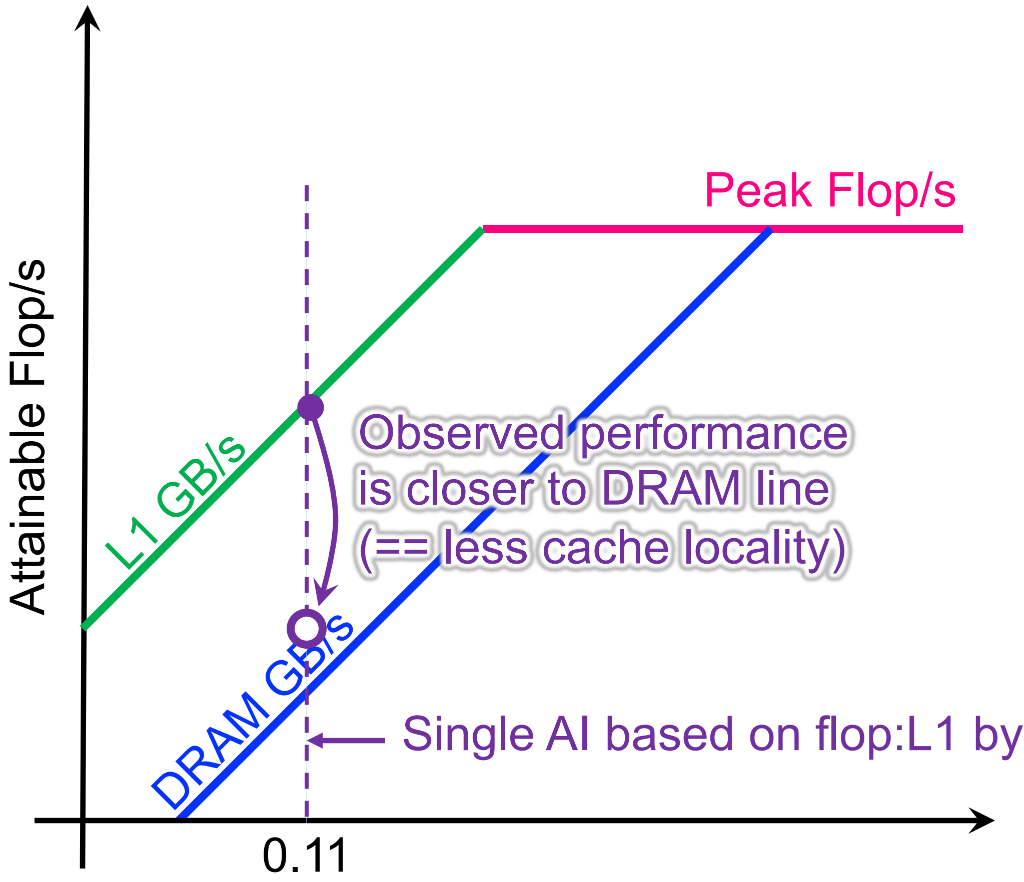


Capacity misses reduce DRAM AI and performance

Multiple AI's….
1) flop:DRAM ~ 0.20
2) flop:L1 ~ 0.11

Arithmetic Intensity (Flop:Byte)

Cache-Aware Roofline



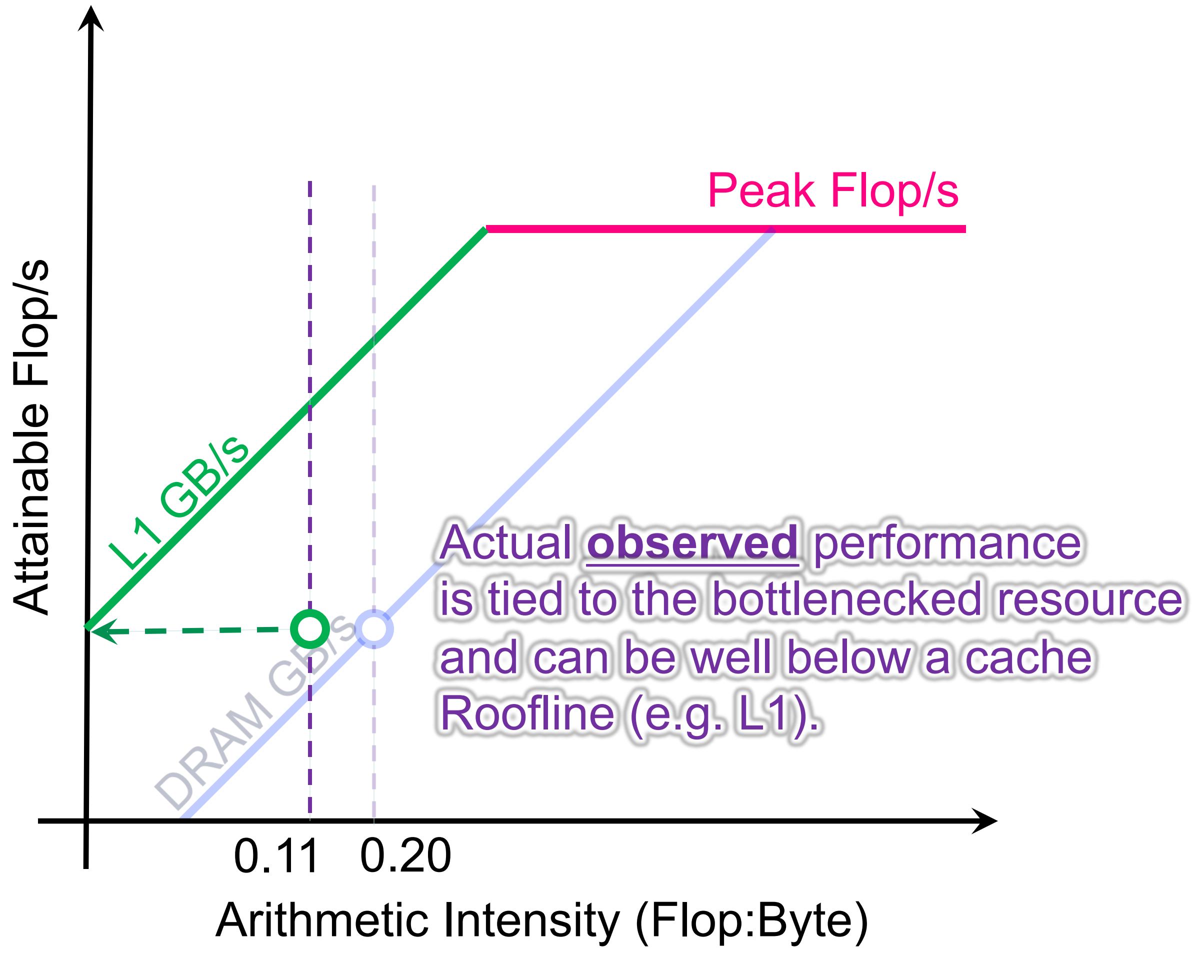
Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes

Arithmetic Intensity (Flop:Byte)

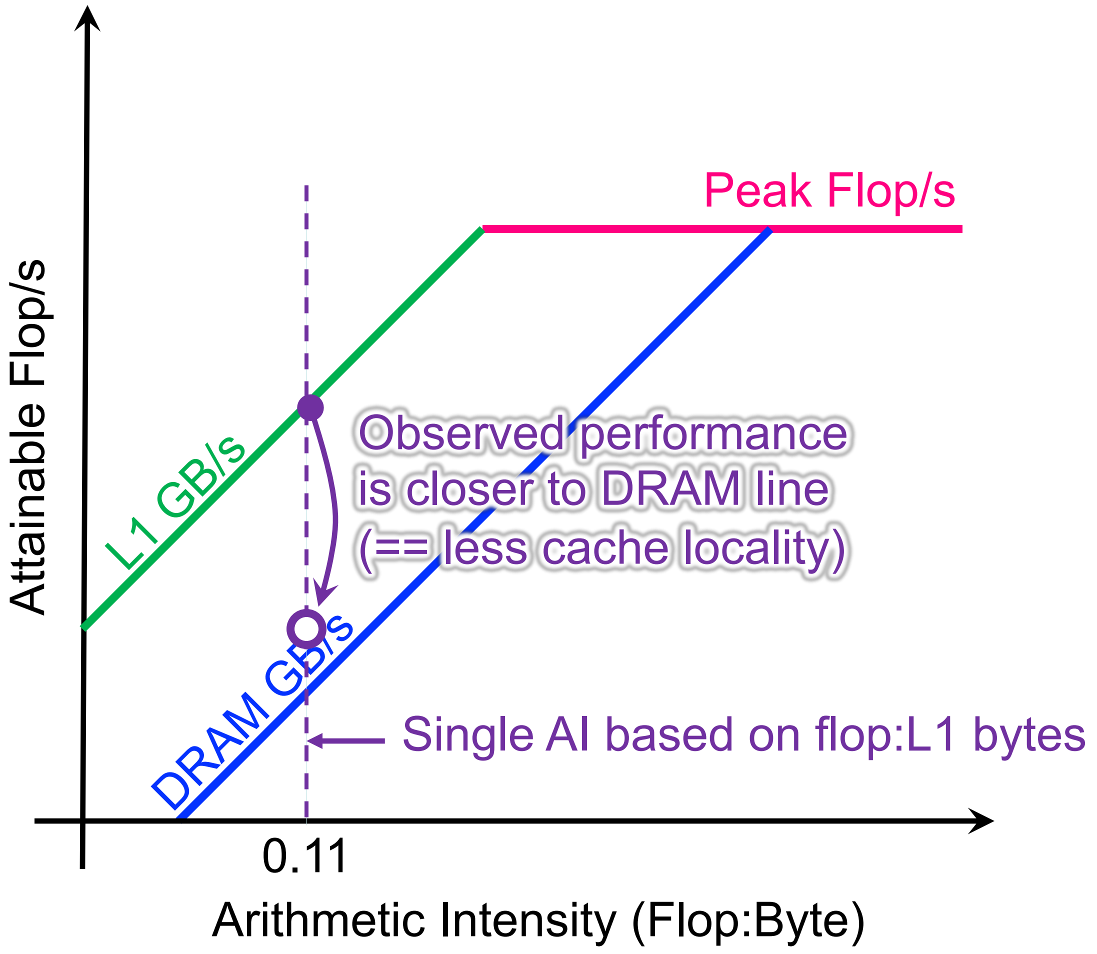
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Actual **observed** performance is tied to the bottlenecked resource and can be well below a cache Roofline (e.g. L1).

Cache-Aware Roofline

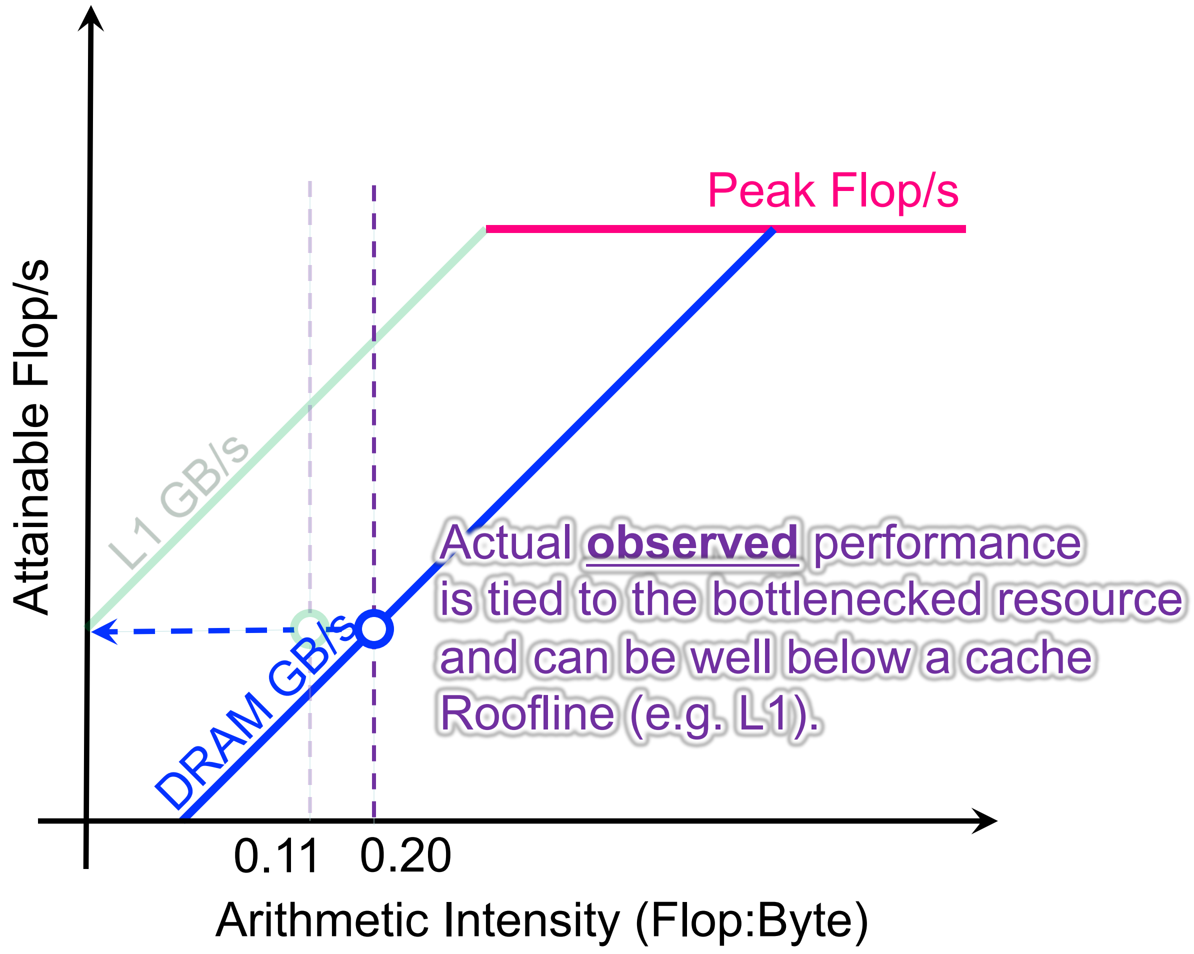


Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes

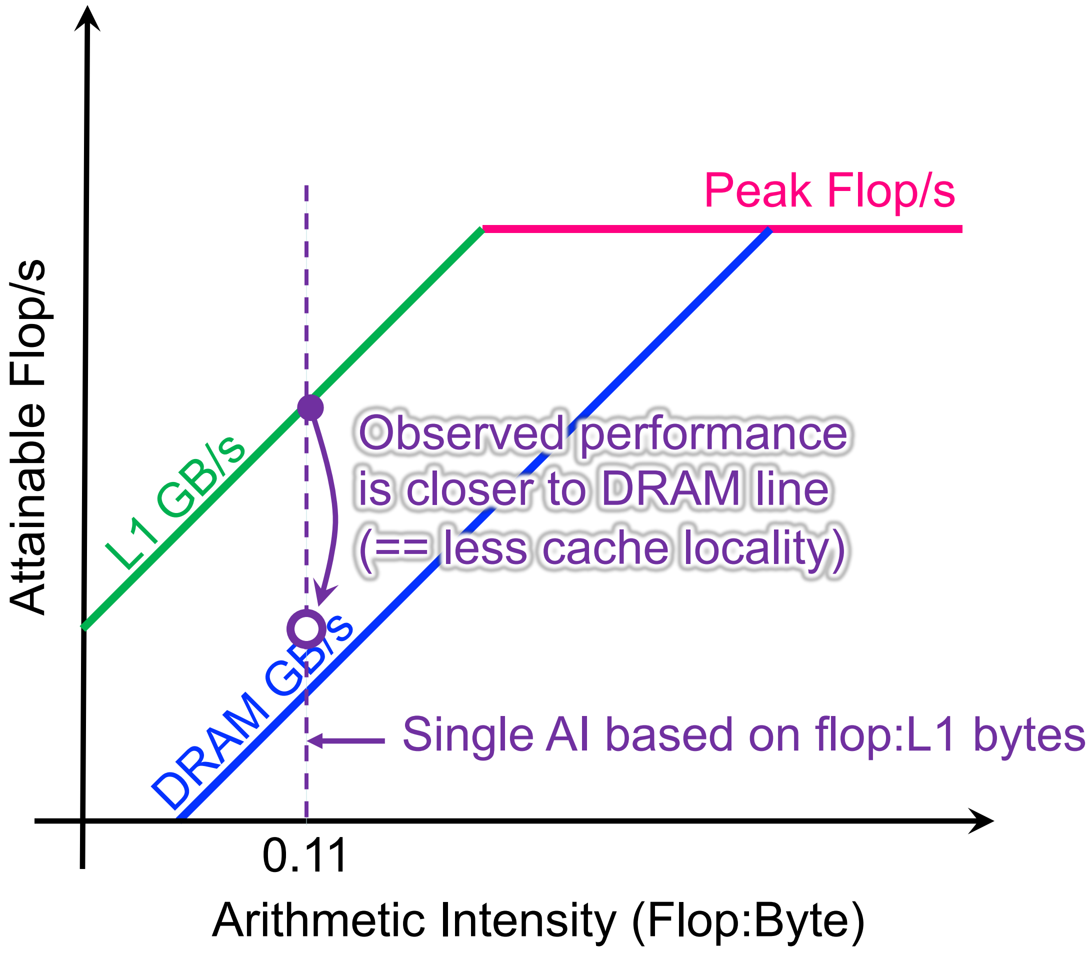
Example: 7-point Stencil (Observed Perf.)

Hierarchical Roofline



Actual **observed** performance is tied to the bottlenecked resource and can be well below a cache Roofline (e.g. L1).

Cache-Aware Roofline



Observed performance is closer to DRAM line (== less cache locality)

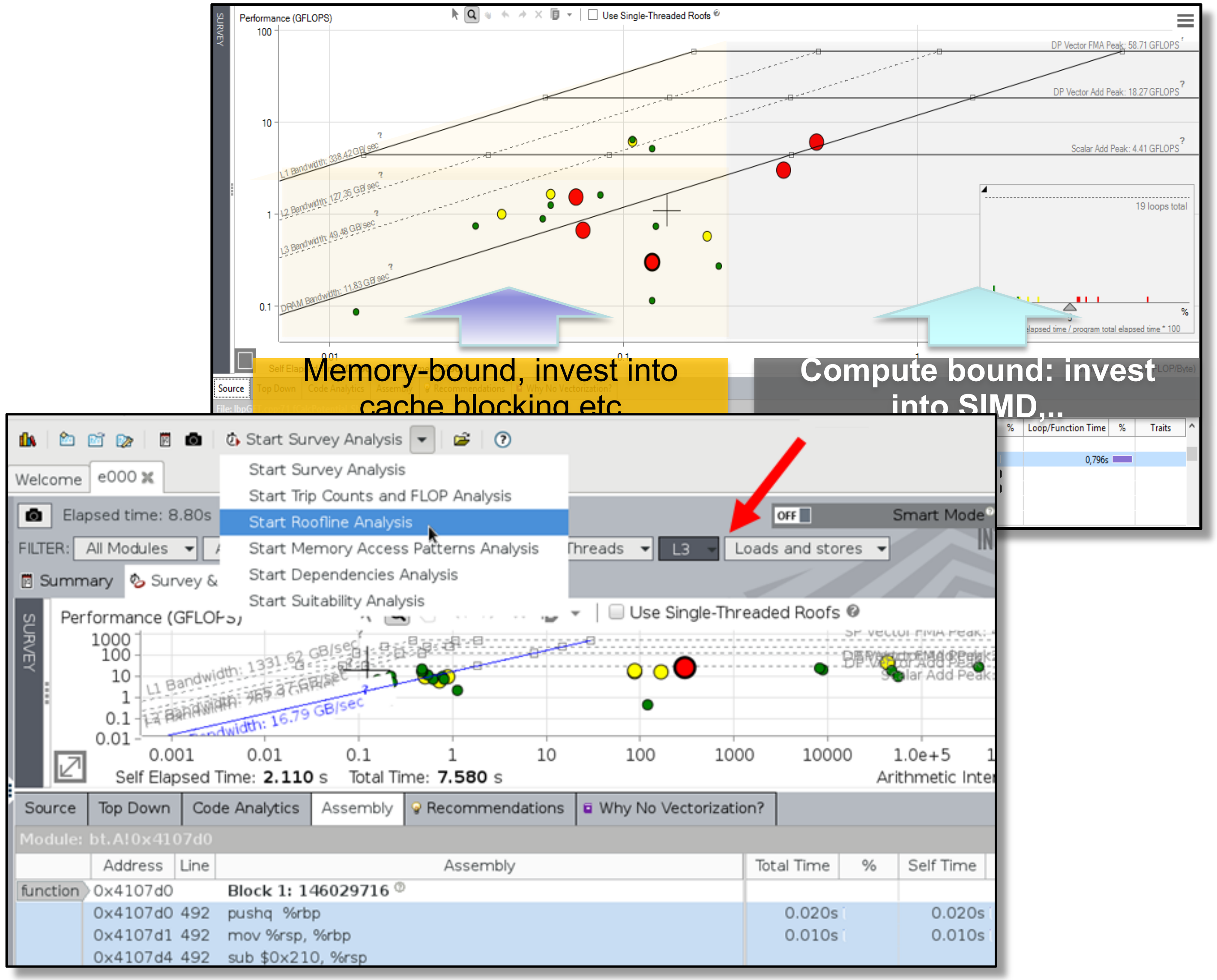
Single AI based on flop:L1 bytes

Roofline with Intel® Advisor

slides from Zakhar Matveev (intel)

Intel Advisor

- Includes Roofline Automation…
  - ✓ Automatically instruments applications (one dot per loop nest/function)
  - ✓ Computes FLOPS and AI for each function (**CARM**)
  - ✓ AVX-512 support that incorporates masks
  - ✓ **Integrated Cache Simulator[1] (hierarchical roofline / multiple AI's)**
  - ✓ Automatically benchmarks target system (calculates ceilings)
  - ✓ Full integration with existing Advisor capabilities



Memory-bound, invest into cache blocking etc

Compute bound: invest into SIMD,..

Self Elapsed Time: 2.110 s Total Time: 7.580 s

Module: bt.A!0x4107d0

	Address	Line	Assembly	Total Time	%	Self Time
function	0x4107d0		Block 1: 146029716			
	0x4107d0	492	pushq %rbp	0.020s		0.020s
	0x4107d1	492	mov %rsp, %rbp	0.010s		0.010s
	0x4107d4	492	sub $0x210, %rsp			

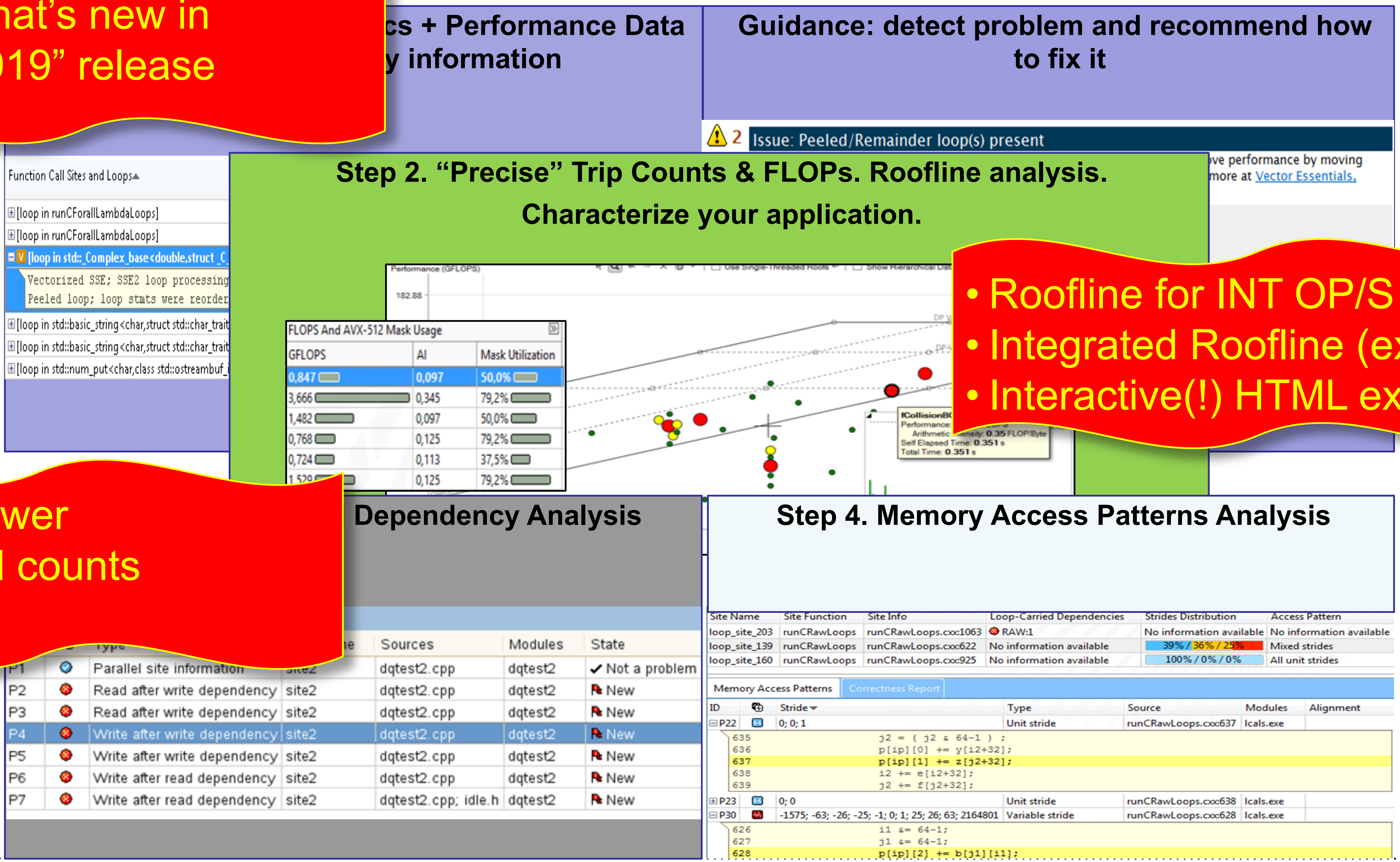
http://www.nersc.gov/users/training/events/roofline-training-1182017-1192017

---

[1] Experimental Feature, the look and feel and exact behavior is subject for change

Intel® Advisor: Components

What's new in
"2019" release



cs + Performance Data
y information

Guidance: detect problem and recommend how to fix it

⚠ 2 Issue: Peeled/Remainder loop(s) present

ve performance by moving
more at Vector Essentials.

Function Call Sites and Loops▲

- ⊞[loop in runCForallLambdaLoops]
- ⊞[loop in runCForallLambdaLoops]
- ⊟ V [loop in std::_Complex_base<double,struct _C
  - Vectorized SSE; SSE2 loop processing
  - Peeled loop; loop stmts were reorder
- ⊞[loop in std::basic_string<char,struct std::char_trait
- ⊞[loop in std::basic_string<char,struct std::char_trait
- ⊞[loop in std::num_put<char,class std::ostreambuf_

Step 2. "Precise" Trip Counts & FLOPs. Roofline analysis.

Characterize your application.

FLOPS And AVX-512 Mask Usage

GFLOPS	AI	Mask Utilization
0,847	0,097	50,0%
3,666	0,345	79,2%
1,482	0,097	50,0%
0,768	0,125	79,2%
0,724	0,113	37,5%
1,529	0,125	79,2%

- Roofline for INT OP/S
- Integrated Roofline (exp)
- Interactive(!) HTML export

Dependency Analysis

- MAC OS viewer
- Function call counts
- Python API..

		Type		Sources	Modules	State
P1	●	Parallel site information	site2	dqtest2.cpp	dqtest2	✓ Not a problem
P2	⊗	Read after write dependency	site2	dqtest2.cpp	dqtest2	New
P3	⊗	Read after write dependency	site2	dqtest2.cpp	dqtest2	New
P4	⊗	Write after write dependency	site2	dqtest2.cpp	dqtest2	New
P5	⊗	Write after write dependency	site2	dqtest2.cpp	dqtest2	New
P6	⊗	Write after read dependency	site2	dqtest2.cpp	dqtest2	New
P7	⊗	Write after read dependency	site2	dqtest2.cpp; idle.h	dqtest2	New

Step 4. Memory Access Patterns Analysis

Site Name	Site Function	Site Info	Loop-Carried Dependencies	Strides Distribution	Access Pattern
loop_site_203	runCRawLoops	runCRawLoops.cxx:1063	⊗ RAW:1	No information available	No information available
loop_site_139	runCRawLoops	runCRawLoops.cxx:622	No information available	39% / 36% / 25%	Mixed strides
loop_site_160	runCRawLoops	runCRawLoops.cxx:925	No information available	100% / 0% / 0%	All unit strides

Memory Access Patterns | Correctness Report

ID		Stride▾	Type	Source	Modules	Alignment
⊟P22		0; 0; 1	Unit stride	runCRawLoops.cxx:637	lcals.exe	

```
j2 = ( j2 & 64-1 ) ;
p[ip][0] += y[i2+32];
p[ip][1] += z[j2+32];
i2 += e[i2+32];
j2 += f[j2+32];
```

ID		Stride▾	Type	Source	Modules	Alignment
⊞P23		0; 0	Unit stride	runCRawLoops.cxx:638	lcals.exe	
⊟P30		-1575; -63; -26; -25; -1; 0; 1; 25; 26; 63; 2164801	Variable stride	runCRawLoops.cxx:628	lcals.exe	

```
i1 &= 64-1;
j1 &= 64-1;
p[ip][2] += b[j1][i1];
```

Intel® Advisor: 2-pass Approach

Roofline: X-Axis (AI): #FLOPs / #Bytes Y-Axis (FLOP/s): #FLOP(mask-aware)/time	**Overhead**
Step 1: Survey (`-collect survey`) • Records run times • User-mode sampling; non-intrusive • ***No need for root access***	**1x**
Step 2: FLOPs (`-collect tripcounts -flops`) • Record #FLOPs, #Bytes, AVX512 masks • Precise, instrumentation-based count of the number of instructions • ***No need for root access***	**3-5x** **(8-37x)[1]**

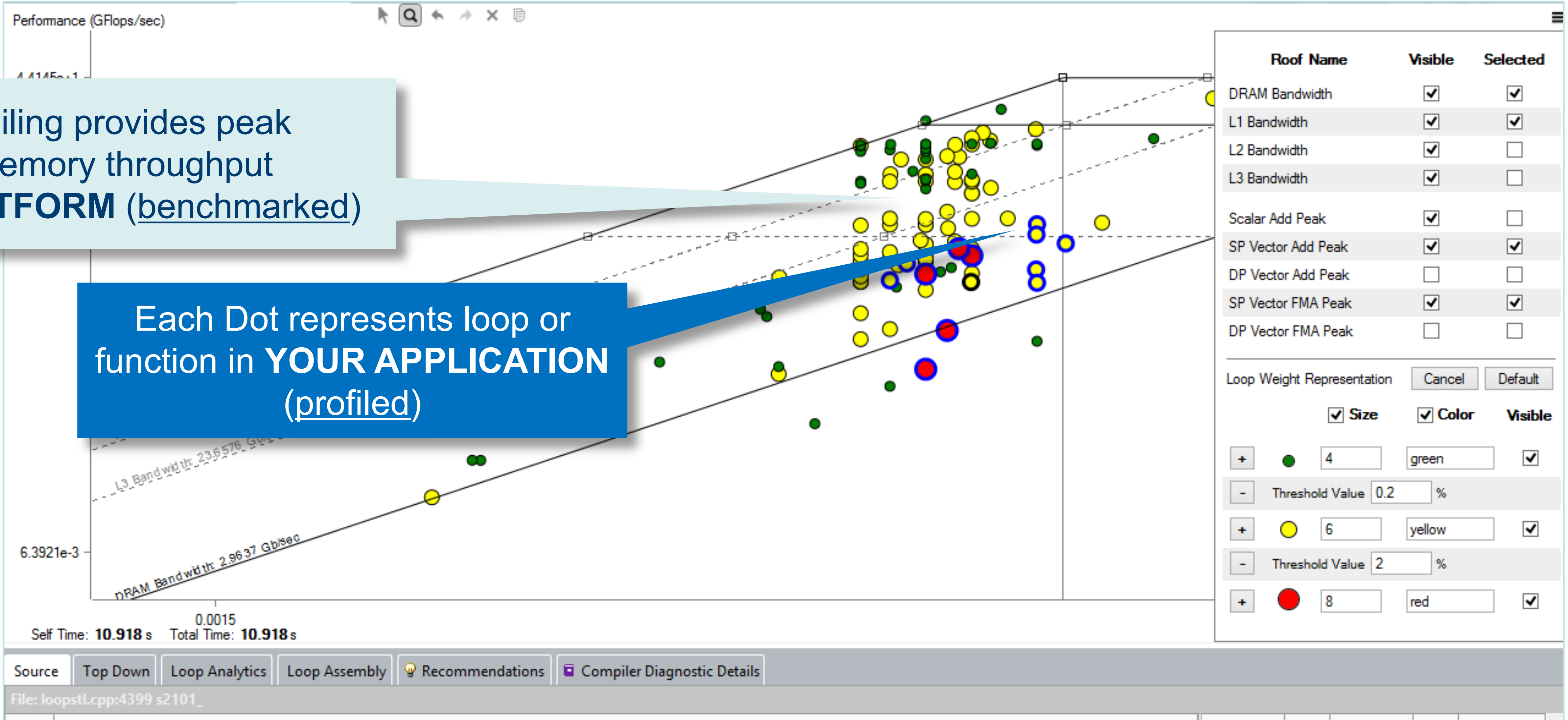
---

[1] With Integrated Roofline (Cache Simulator) enabled.

Intel® Advisor: Roofline Automation

Each Ceiling provides peak
CPU/Memory throughput
of your **PLATFORM** (benchmarked)

Each Dot represents loop or
function in **YOUR APPLICATION**
(profiled)



Self Time: **10.918** s Total Time: **10.918** s

Roof Name	Visible	Selected
DRAM Bandwidth	☑	☑
L1 Bandwidth	☑	☑
L2 Bandwidth	☑	☐
L3 Bandwidth	☑	☐
Scalar Add Peak	☑	☐
SP Vector Add Peak	☑	☑
DP Vector Add Peak	☐	☐
SP Vector FMA Peak	☑	☑
DP Vector FMA Peak	☐	☐

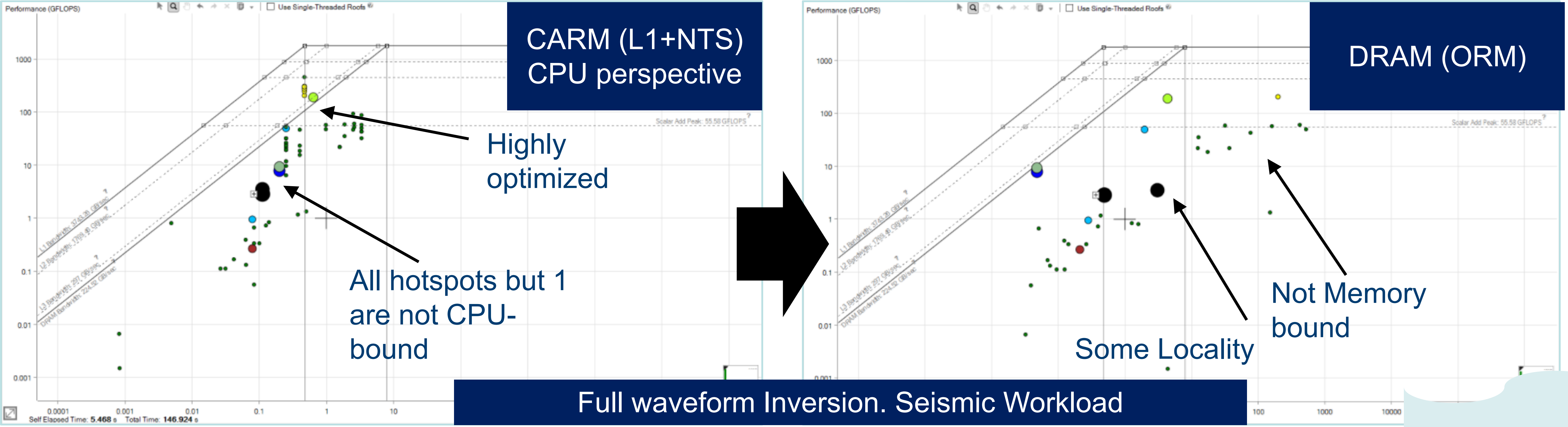
Loop Weight Representation

		☑ Size	☑ Color	Visible
+	●	4	green	☑
-	Threshold Value	0.2	%	
+	○	6	yellow	☑
-	Threshold Value	2	%	
+	●	8	red	☑

File: loopstl.cpp:4399 s2101_

Automatic and integrated – first class citizen in Intel® Advisor

NEW: Integrated Roofline



CARM (L1+NTS)
CPU perspective

Highly optimized

All hotspots but 1 are not CPU-bound

DRAM (ORM)

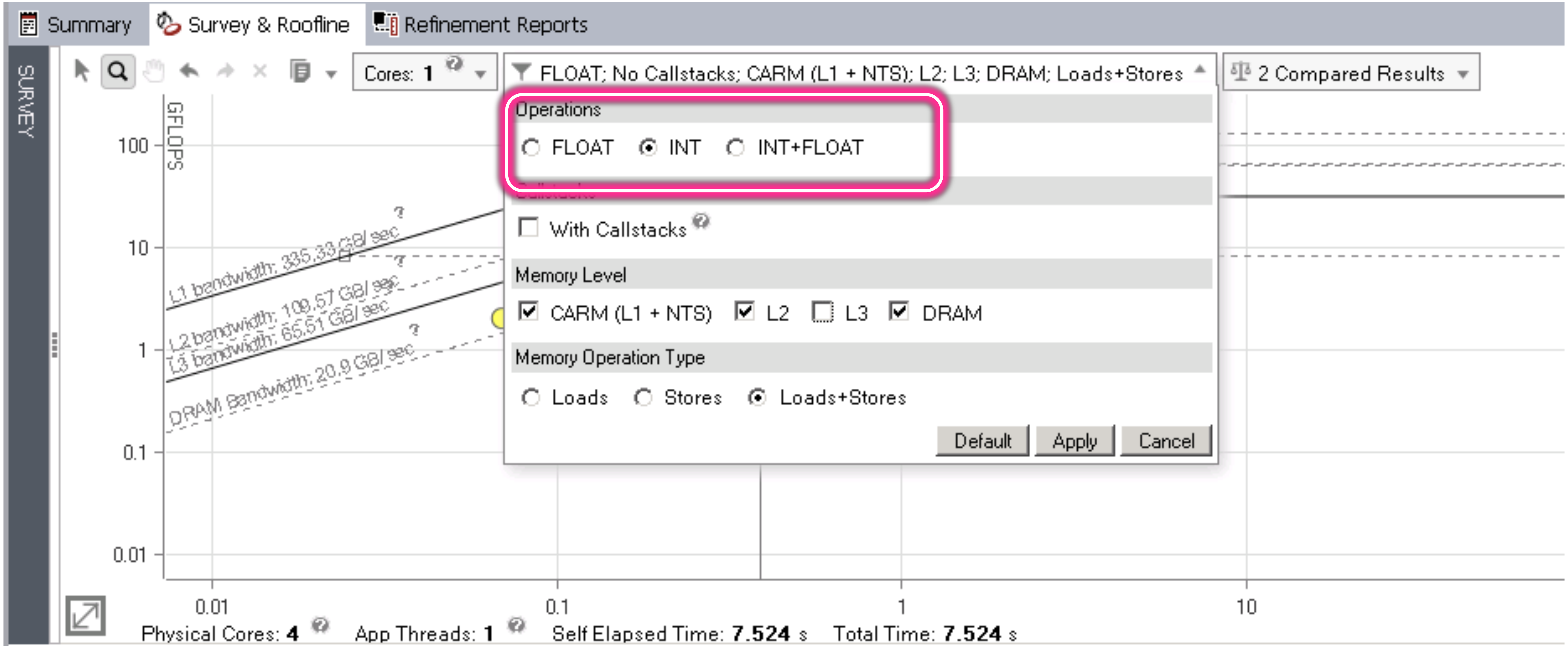
Not Memory bound

Some Locality

Full waveform Inversion. Seismic Workload

Data: Courtesy Philippe Thierry

NEW: Integer, Float, Int+Float Rooflines



Cores: 1

FLOAT; No Callstacks; CARM (L1 + NTS); L2; L3; DRAM; Loads+Stores

2 Compared Results

Operations

○ FLOAT ● INT ○ INT+FLOAT

☐ With Callstacks

Memory Level

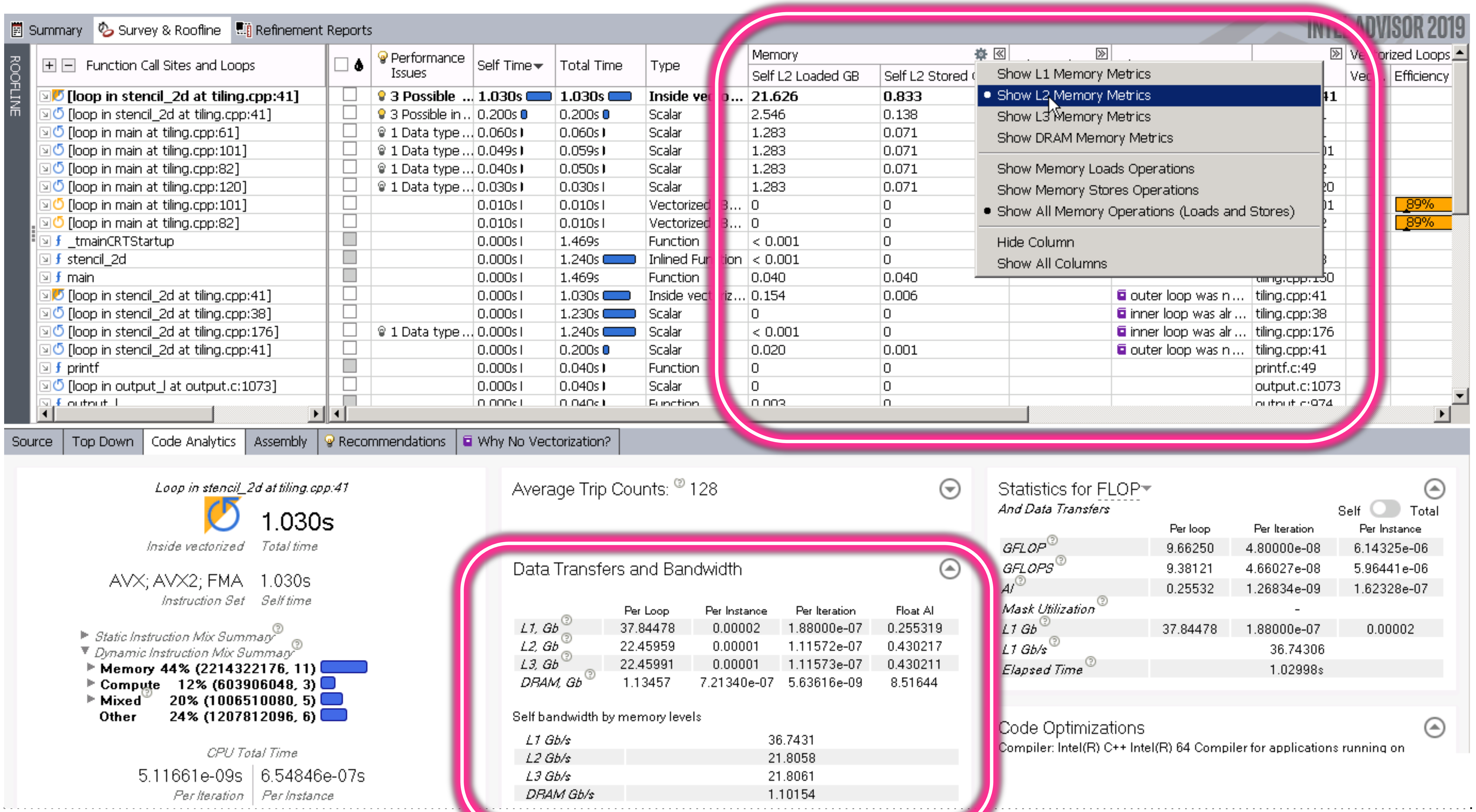
☑ CARM (L1 + NTS) ☑ L2 ☐ L3 ☑ DRAM

Memory Operation Type

○ Loads ○ Stores ● Loads+Stores

Physical Cores: **4** App Threads: **1** Self Elapsed Time: **7.524** s Total Time: **7.524** s

NEW: Memory Traffic in Survey Grid



Function Call Sites and Loops	Performance Issues	Self Time	Total Time	Type	Self L2 Loaded GB	Self L2 Stored			Efficiency
[loop in stencil_2d at tiling.cpp:41]	**3 Possible ...**	**1.030s**	**1.030s**	**Inside ve o...**	**21.626**	**0.833**		**41**	
[loop in stencil_2d at tiling.cpp:41]	3 Possible in..	0.200s	0.200s	Scalar	2.546	0.138			
[loop in main at tiling.cpp:61]	1 Data type ...	0.060s	0.060s	Scalar	1.283	0.071			
[loop in main at tiling.cpp:101]	1 Data type ...	0.049s	0.059s	Scalar	1.283	0.071		01	
[loop in main at tiling.cpp:82]	1 Data type ...	0.040s	0.050s	Scalar	1.283	0.071		2	
[loop in main at tiling.cpp:120]	1 Data type ...	0.030s	0.030s	Scalar	1.283	0.071		20	
[loop in main at tiling.cpp:101]		0.010s	0.010s	Vectorized B...	0	0		01	89%
[loop in main at tiling.cpp:82]		0.010s	0.010s	Vectorized B...	0	0		2	89%
_tmainCRTStartup		0.000s	1.469s	Function	< 0.001	0			
stencil_2d		0.000s	1.240s	Inlined Fun tion	< 0.001	0		3	
main		0.000s	1.469s	Function	0.040	0.040		cpp:150	
[loop in stencil_2d at tiling.cpp:41]		0.000s	1.030s	Inside vect iz...	0.154	0.006	outer loop was n ...	tiling.cpp:41	
[loop in stencil_2d at tiling.cpp:38]		0.000s	1.230s	Scalar	0	0	inner loop was alr ...	tiling.cpp:38	
[loop in stencil_2d at tiling.cpp:176]	1 Data type ...	0.000s	1.240s	Scalar	< 0.001	0	inner loop was alr ...	tiling.cpp:176	
[loop in stencil_2d at tiling.cpp:41]		0.000s	0.200s	Scalar	0.020	0.001	outer loop was n ...	tiling.cpp:41	
printf		0.000s	0.040s	Function	0	0		printf.c:49	
[loop in output_l at output.c:1073]		0.000s	0.040s	Scalar	0	0		output.c:1073	
output_l		0.000s	0.040s	Function	0.003	0		output.c:974	

Loop in stencil_2d at tiling.cpp:41

1.030s
Inside vectorized *Total time*

AVX; AVX2; FMA 1.030s
Instruction Set *Self time*

- *Static Instruction Mix Summary*
- *Dynamic Instruction Mix Summary*
  - **Memory 44% (2214322176, 11)**
  - **Compute 12% (603906048, 3)**
  - **Mixed 20% (1006510080, 5)**
  - **Other 24% (1207812096, 6)**

CPU Total Time
5.11661e-09s *Per Iteration* | 6.54846e-07s *Per Instance*

Average Trip Counts: 128

Data Transfers and Bandwidth

	Per Loop	Per Instance	Per Iteration	Float AI
L1, Gb	37.84478	0.00002	1.88000e-07	0.255319
L2, Gb	22.45959	0.00001	1.11572e-07	0.430217
L3, Gb	22.45991	0.00001	1.11573e-07	0.430211
DRAM, Gb	1.13457	7.21340e-07	5.63616e-09	8.51644

Self bandwidth by memory levels

L1 Gb/s	36.7431
L2 Gb/s	21.8058
L3 Gb/s	21.8061
DRAM Gb/s	1.10154

Statistics for FLOP
And Data Transfers

Self Total

	Per loop	Per Iteration	Per Instance
GFLOP	9.66250	4.80000e-08	6.14325e-06
GFLOPS	9.38121	4.66027e-08	5.96441e-06
AI	0.25532	1.26834e-09	1.62328e-07
Mask Utilization		-	
L1 Gb	37.84478	1.88000e-07	0.00002
L1 Gb/s		36.74306	
Elapsed Time		1.02998s	

Code Optimizations
Compiler: Intel(R) C++ Intel(R) 64 Compiler for applications running on

Integrated Roofline Model

Old Approach…

```
source advixe-vars.sh
advixe-cl -collect survey --project-dir ./your_project -- <your-executable-with-parameters>
advixe-cl -collect tripcounts  -enable-cache-simulation -flop --project-dir ./your_project -- <your-executable-with-parameters>
```

New Approach (but not compatible with MPI)…

```
source advixe-vars.sh
advixe-cl -collect roofline  -enable-cache-simulation --project-dir ./your_project -- <your-executable-with-parameters>
```

(optional) copy data to your UI desktop system

```
advixe-gui ./your_project
```

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

Advisor on NERSC's Cori

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

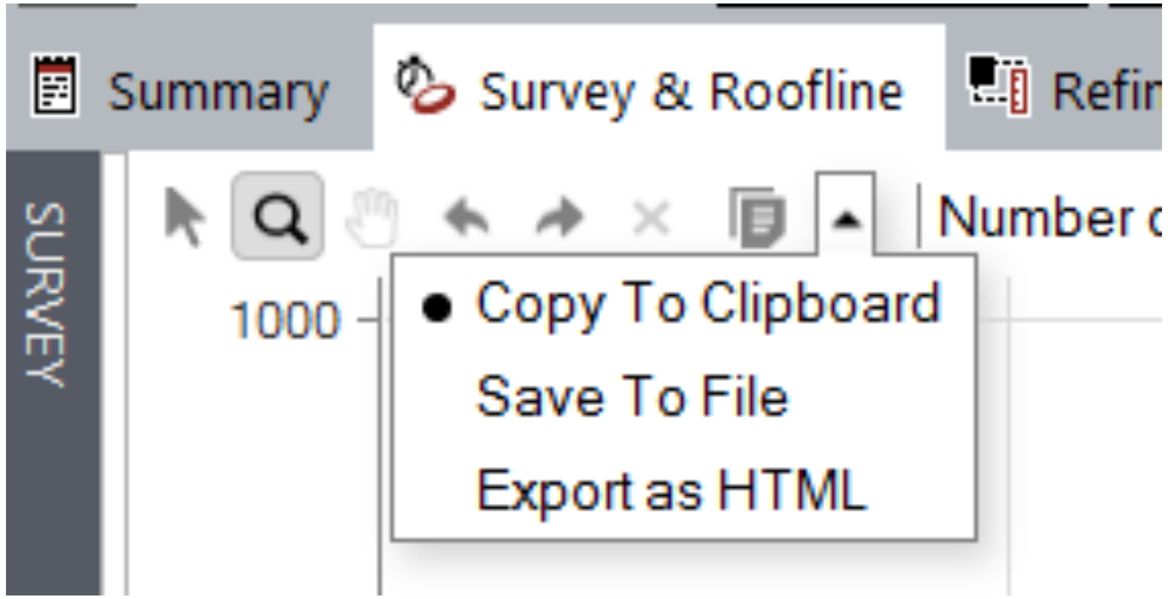
```
module load advisor/2018.integrated_roofline
cc -g -dynamic -openmp -O2 -o mycode.exe mycode.c
```

- Best to run advisor only on rank 0... srun calls a script like…

```
#!/bin/bash
if [[ $SLURM_PROCID == 0 ]];then
advixe-cl -collect=survey --project-dir knl-result -data-limit=0 -- ./a.out
else
sleep 30
./a.out
fi
```

Exporting Roofline Figures

- Advisor can directly export a HTML Roofline figure …



- Alternately, you can output directly from the command line (no GUI needed)…

```
advixe-cl -report roofline --project-dir ./your_project > roofline.html
```

Questions?

Summary

Summary

- In this talk, we discussed several approaches to constructing Rooflines on CPUs…
    - Machine Characterization
    - Using LIKWID to access performance counters
    - Using SDE to get more accurate FLOP counts
    - Using Advisor to provide a single tool that integrates cache simulation and accurate FLOP counts.

Backup