Measuring hardware performance counters on ARCHER2 using LIKWID

Arno Proeme & Evgenij Belikov

EPCC, The University of Edinburgh

07/08/2025

www.archer2.ac.uk

Reusing this material

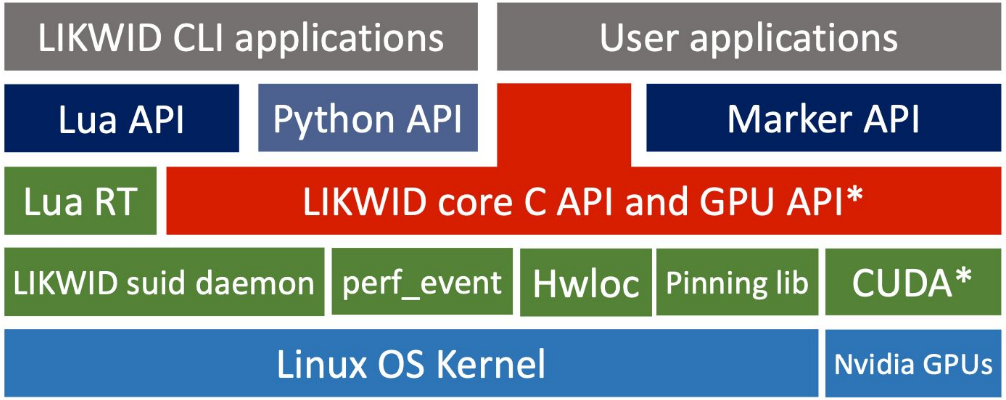
Partners

LIKWID overview

Command line performance tools:

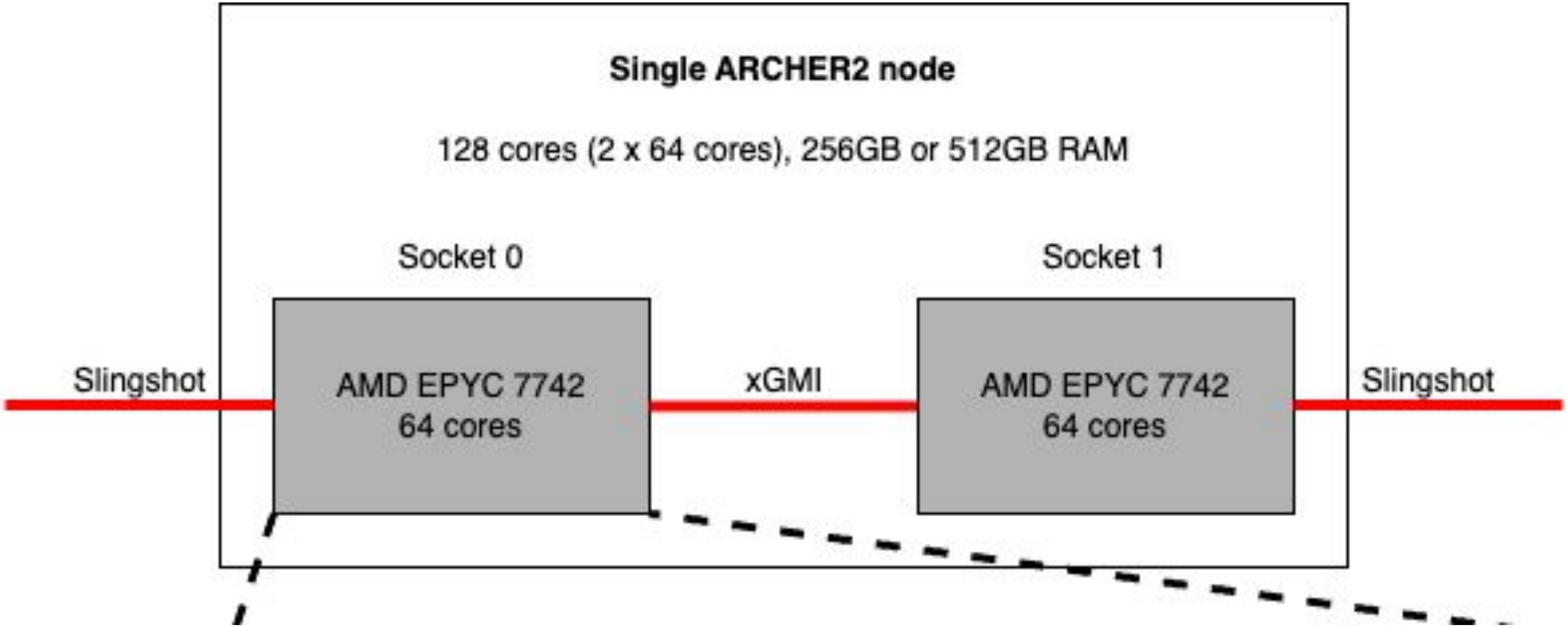
- Probe hardware topology
  - **likwid-topology** - print thread, cache and NUMA topology
- Process & thread placement control
  - **likwid-pin** - pin application threads
  - **likwid-mpirun** - pin application processes
- Hardware performance monitoring
  - **likwid-perfctr** - measure hardware performance counters
  - **likwid-mpirun** - likwid-perfctr for MPI applications
- Microbenchmarking
  - **likwid-bench** - evaluate upper performance bounds using assembly microkernels

LIKWID 5 Tools Architecture



Topology

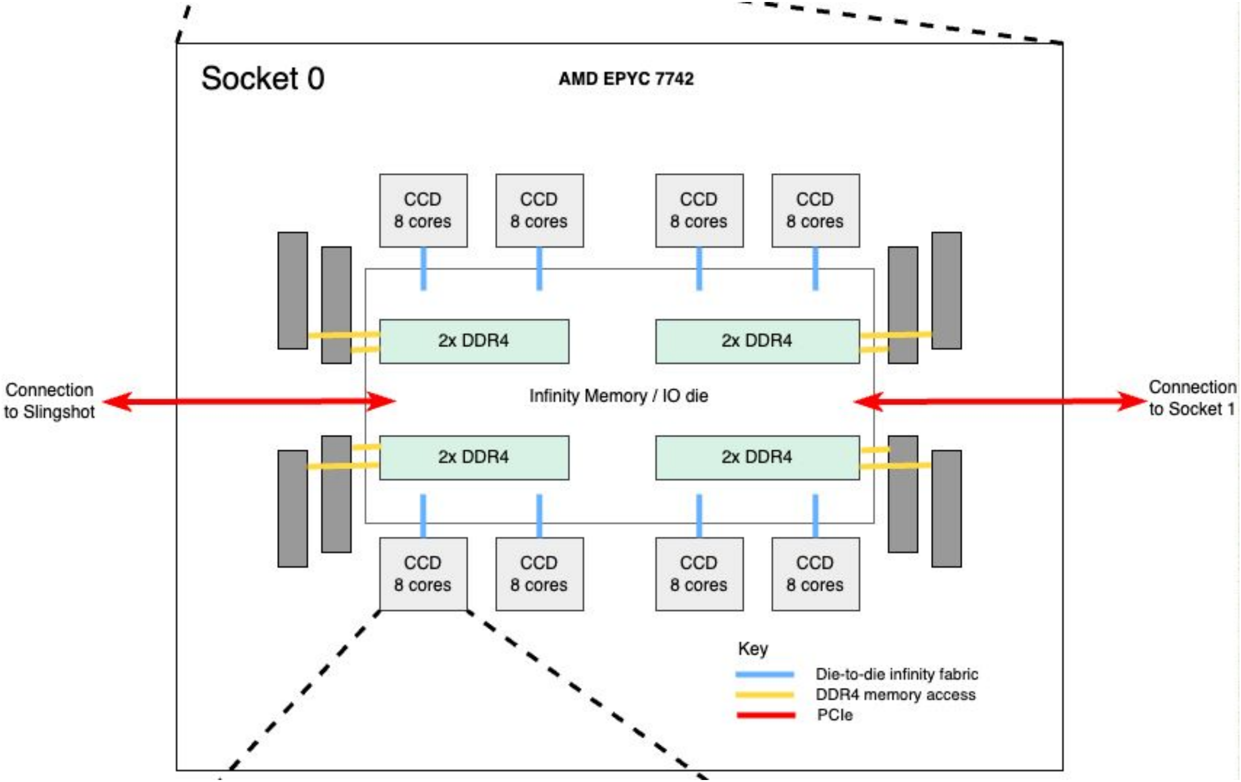
ARCHER2 Node



likwid-topology - hardware threads

```
--------------------------------------------------------------------------------
CPU name:         AMD EPYC 7742 64-Core Processor
CPU type:         AMD K17 (Zen2) architecture
CPU stepping:     0
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:          2
CPU dies:                 2
Cores per socket:64
Threads per core:2
--------------------------------------------------------------------------------
HWThread        Thread        Core        Die        Socket        Available
0               0             0           0          0             *
1               0             1           0          0             *
2               0             2           0          0             *
.
.
.
126             0             126         0          1             *
127             0             127         0          1             *
128             1             0           0          0             *
129             1             1           0          0             *
130             1             2           0          0             *
.
.
.
254             1             126         0          1             *
255             1             127         0          1             *
```

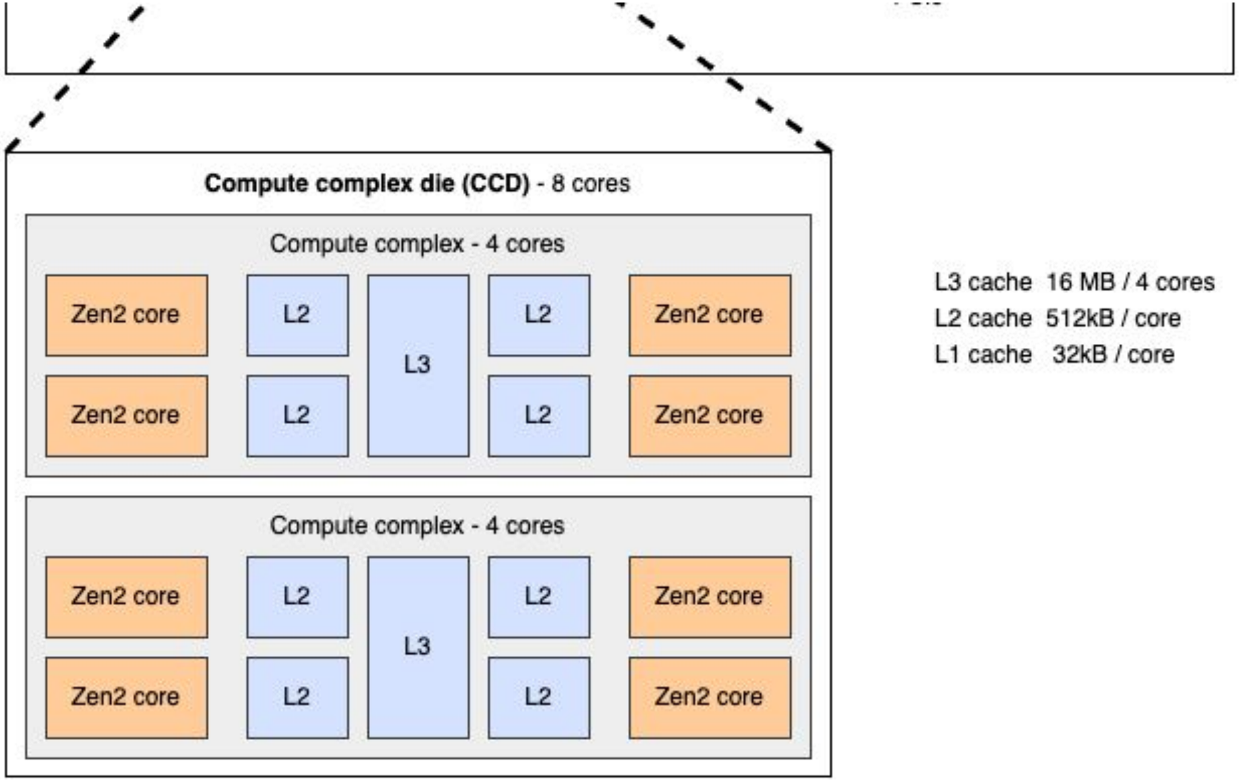
ARCHER2 processor



likwid-topology - NUMA

```
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:            8
--------------------------------------------------------------------------------
Domain:                  0
Processors:              ( 0 128 1 129 2 130 3 131 4 132 5 133 6 134 7 135 8 136 9 137 10 138 11 139 12 140 13 141 14 142 15 143 )
Distances:               10 12 12 12 32 32 32 32
Free memory:             29215.3 MB
Total memory:            63749.9 MB
--------------------------------------------------------------------------------
Domain:                  1
Processors:              ( 16 144 17 145 18 146 19 147 20 148 21 149 22 150 23 151 24 152 25 153 26 154 27 155 28 156 29 157 30 158 31 159 )
Distances:               12 10 12 12 32 32 32 32
Free memory:             27638.5 MB
Total memory:            64504.1 MB
--------------------------------------------------------------------------------
.
.
--------------------------------------------------------------------------------
Domain:                  6
Processors:              ( 96 224 97 225 98 226 99 227 100 228 101 229 102 230 103 231 104 232 105 233 106 234 107 235 108 236 109 237 110 238
111 239 )
Distances:               32 32 32 32 12 12 10 12
Free memory:             29890.9 MB
Total memory:            64504.1 MB
--------------------------------------------------------------------------------
Domain:                  7
Processors:              ( 112 240 113 241 114 242 115 243 116 244 117 245 118 246 119 247 120 248 121 249 122 250 123 251 124 252 125 253 126
254 127 255 )
Distances:               32 32 32 32 12 12 12 10
Free memory:             32468.4 MB
Total memory:            64494.1 MB
--------------------------------------------------------------------------------
```

ARCHER2 Compute Complex Die (CCD) = 2 Core Complexes (CCX)



L3 cache 16 MB / 4 cores
L2 cache 512kB / core
L1 cache 32kB / core

likwid-topology - cache

```
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Type:                   Data cache
Associativity:          8
Number of sets:         64
Cache line size: 64
Cache type:             Non Inclusive
Shared by threads:      2
Cache groups:           ( 0 128 ) ( 1 129 ) ... ( 126 254 ) ( 127 255 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   512 kB
Type:                   Unified cache
Associativity:          8
Number of sets:         1024
Cache line size: 64
Cache type:             Non Inclusive
Shared by threads:      2
Cache groups:           ( 0 128 ) ( 1 129 ) ... ( 126 254 ) ( 127 255 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   16 MB
Type:                   Unified cache
Associativity:          16
Number of sets:         16384
Cache line size: 64
Cache type:             Non Inclusive
Shared by threads:      8
Cache groups:           ( 0 128 1 129 2 130 3 131 ) ( 4 132 5 133 6 134 7 135 ) ... ( 120 248 121 249 122 250 123 251 ) ( 124 252 125 253 126
254 127 255 )
--------------------------------------------------------------------------------
```

Pinning

likwid-pin - thread pinning

Pin threads to cores based on:

1. OS: numbering according to the OS
   `likwid-pin -c 0,2,4,6`
2. Node: logical numbering over whole node (**N:<list>**)
   `likwid-pin -c N:0-3`
3. Socket: logical numbering within each socket (**S#:<list>**)
   `likwid-pin -c S0:0-3@S1:0-3`
4. Die: logical numbering within each die (**D#:<list>**)
   `likwid-pin -c D0:0-63`
5. Cache group: logical numbering within each last level cache groups (**C#:<list>**)
   `likwid-pin -c C0:0@C1:0@C2:0@C3:0`
6. Memory domain: logical numbering within each NUMA domain (**M#:<list>**)
   `likwid-pin -c M0:0-15`
7. Cpuset: logical numbering inside each Linux cpuset (**L prefix**)

Various list syntax options

likwid-pin - thread domains on ARCHER2

```
> likwid-pin -p

Domain N: 0,128,1,129 … 126,254,127,255

Domain S0: 0,128,1,129 … 62,190,63,191
Domain S1: 64,192,65,193 … 126,254,127,255

Domain D0: 0,128,1,129 … 62,190,63,191
Domain D1: 64,192,65,193 … 126,254,127,255

Domain C0: 0,128,1,129,2,130,3,131
...
Domain C31: 124,252,125,253,126,254,127,255

Domain M0: 0,128,1,129 .. 14,142,15,143
...
Domain M7: 112,240,113,241 .. 126,254,127,255
```

Note: `srun --cpus-per-task=128 --hint=nomultithread likwid-pin -p` would exclude 128-255

Performance counter measurement

AMD Zen2 event counters

Core-local counters

3 fixed-purpose:

Counter name	Event name
FIXC0	INST_RETIRED_ANY
FIXC1	ACTUAL_CPU_CLOCK
FIXC2	MAX_CPU_CLOCK

6 general-purpose: PMC0-PMC5, programmable event types

Socket-wide counters

Energy counters:

Counter name	Event name
PWR0	RAPL_CORE_ENERGY
PWR1	RAPL_PKG_ENERGY

Derived metrics and performance groups

Metrics of interest are derived from architecture-specific counters according to LIKWID-provided recipes

Grouped into "performance groups" (event sets & derived metrics):

Event set	Function
FLOPS_DP	Double Precision MFlops/s
FLOPS_SP	Single Precision MFlops/s
L2	L2 cache bandwidth in MBytes/s
L3	L3 cache bandwidth in MBytes/s
MEM	Main memory bandwidth in MBytes/s
CACHE	L1 Data cache miss rate/ratio
L2CACHE	L2 Data cache miss rate/ratio
L3CACHE	L3 Data cache miss rate/ratio
DATA	Load to store ratio
BRANCH	Branch prediction miss rate/ratio
TLB	Translation lookaside buffer miss rate/ratio

AMD Zen2 FLOPS_DP eventset & metrics

```
SHORT Double Precision MFLOP/s

EVENTSET
FIXC1 ACTUAL_CPU_CLOCK
FIXC2 MAX_CPU_CLOCK
PMC0  RETIRED_INSTRUCTIONS
PMC1  CPU_CLOCKS_UNHALTED
PMC2  RETIRED_SSE_AVX_FLOPS_ALL
PMC3  MERGE

METRICS
Runtime (RDTSC) [s] time
Runtime unhalted [s]   FIXC1*inverseClock
Clock [MHz]  1.E-06*(FIXC1/FIXC2)/inverseClock
CPI   PMC1/PMC0
DP [MFLOP/s]   1.0E-06*(PMC2)/time

LONG
Formulas:
CPI = CPU_CLOCKS_UNHALTED/RETIRED_INSTRUCTIONS
DP [MFLOP/s] = 1.0E-06*(RETIRED_SSE_AVX_FLOPS_ALL)/time
```

likwid-perfctr - counter measurement

- Wrapper mode (default):
  - wrap application launch, measure performance counters during execution
  - typically measure on all cores where your application executes
  - can instrument using Marker API to measure region(s) - kernel(s) - only

- Stethoscope mode
  - "listen to" whatever is running on any specified cores for specified duration
  - output: aggregate statistics over specified cores and duration
  - difficult to relate to what application code is being executed

- Timeline mode
  - periodically output aggregate statistics over specified cores & time window
  - can gain insight into application phases but still difficult to correlate with exactly what application code is being executed

No code changes / recompilation needed except for Marker API

likwid-mpirun - likwid-perfctr with MPI

- likwid-perfctr supports threaded applications but not MPI
  - could launch likwid-perfctr using srun (see LIKWID MPI Tutorial)
  - statistics not aggregated over processes (could be useful for load imbalance)

- likwid-mpirun supports MPI-parallel (incl. MPI + threaded) apps:
  - wrapper mode only (incl. marker API)
  - generates srun commands to launch likwid-perfctr
  - aggregates statistics over processes and threads

likwid-mpirun options

-n <count>: total number of processes

-t <count>: number of threads per process

-pin <list>: pinning of processes (and their threads if relevant)
- (mostly) follows likwid-pin syntax

-g/--group <perf>: performance group (e.g. FLOPS_DP, MEM, etc.)

--nocpubind: disable process binding through (ARCHER2 custom option, central LIKWID install only)
- recommended for convenience - control process binding as usual on ARCHER2

-s/--skip <hex>: skip shepherd threads
- always use -s 0x0 on ARCHER2

-d/--debug: more information, e.g. on generated srun command
- very useful to ensure desired application placement and measurement

--mpiopts: pass any desired arguments to generated srun commands

-nperdomain: only useful for pure MPI, can accomplish similar using -pin

-m: activate marker API mode

Example job scripts & outputs

Example job script

Pure MPI, 2 fully populated nodes (2 x 128 processes)

```
#!/bin/bash

#SBATCH --account=z19
#SBATCH --partition=standard
#SBATCH --qos=short
#SBATCH --time=00:05:00
#SBATCH --nodes=2
#SBATCH --tasks-per-node=128
#SBATCH --cpus-per-task=1
#SBATCH --hint=nomultithread
#SBATCH --distribution=block:block

module load likwid
module load xthi

export OMP_NUM_THREADS=1
export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK

likwid-mpirun -n $SLURM_NTASKS --nocpubind -s 0x0 -g FLOPS_DP xthi_mpi &> xthi_mpi.out
```

Example job script

Pure MPI, 2 nodes, 2 ranks per node, 1 per socket (processor)

```
#!/bin/bash

#SBATCH --account=z19
#SBATCH --partition=standard
#SBATCH --qos=short
#SBATCH --time=00:05:00
#SBATCH --nodes=2
#SBATCH --tasks-per-node=2
#SBATCH --cpus-per-task=64
#SBATCH --hint=nomultithread
#SBATCH --distribution=block:block

module load likwid
module load xthi

export OMP_NUM_THREADS=1
export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK


likwid-mpirun -n $SLURM_NTASKS -pin N:0_N:64 --nocpubind -s 0x0 -g FLOPS_DP --debug xthi_mpi &> xthi_mpi.out

# Alternatively:
likwid-mpirun -n $SLURM_NTASKS -pin S0:0_S1:0 --nocpubind -s 0x0 -g FLOPS_DP --debug xthi_mpi &> xthi_mpi.out
```

Application (xthi) output

Pure MPI, 2 nodes, 2 ranks per node, 1 per socket (processor)

```
Node summary for    2 nodes:
Node    0, hostname nid001068, mpi   2, omp   1, executable xthi_mpi
Node    1, hostname nid001079, mpi   2, omp   1, executable xthi_mpi
MPI summary: 4 ranks
Node    0, rank    0, thread   0, (affinity =    0)
Node    0, rank    1, thread   0, (affinity =   64)
Node    1, rank    2, thread   0, (affinity =    0)
Node    1, rank    3, thread   0, (affinity =   64)
```

likwid-mpirun results for application (cp2k H2O-32)

Pure MPI, 2 nodes, 2 ranks per node, 1 per socket (processor)

Event	Counter	nid001068:0:0	nid001068:1:64	nid001079:2:0	nid001079:3:64
ACTUAL_CPU_CLOCK	FIXC1	272843810277	273383588225	273224152138	273645884841
MAX_CPU_CLOCK	FIXC2	307238142692	307671324996	307777646157	308110758689
RETIRED_INSTRUCTIONS	PMC0	639095695379	612418860306	618138505835	624705982405
CPU_CLOCKS_UNHALTED	PMC1	4325299091000	4334074578504	4329865216924	4331611334344
RETIRED_SSE_AVX_FLOPS_ALL	PMC2	379776213171	345443299984	362489794174	378163333211
MERGE	PMC3	4644337115725824	4644337115725824	4785074604081152	4644337115725824

Event	Counter	Sum	Min	Max	Avg
ACTUAL_CPU_CLOCK STAT	FIXC1	1093097435481	272843810277	273645884841	2.732744e+11
MAX_CPU_CLOCK STAT	FIXC2	1230797872534	307238142692	308110758689	3.076995e+11
RETIRED_INSTRUCTIONS STAT	PMC0	2494359043925	612418860306	639095695379	6.235898e+11
CPU_CLOCKS_UNHALTED STAT	PMC1	17320850220772	4325299091000	4334074578504	4330212555193
RETIRED_SSE_AVX_FLOPS_ALL STAT	PMC2	1465872640540	345443299984	379776213171	366468160135
MERGE STAT	PMC3	18718085951258624	4644337115725824	4785074604081152	4679521487814656

Metric	nid001068:0:0	nid001068:1:64	nid001079:2:0	nid001079:3:64
Runtime (RDTSC) [s]	148.7140	148.7171	147.6779	147.6825
Runtime unhalted [s]	121.4966	121.7369	121.6676	121.8564
Clock [MHz]	1994.2933	1995.4260	1993.5457	1994.4471
CPI	6.7678	7.0770	7.0047	6.9338
DP [MFLOP/s]	2553.7354	2322.8217	2454.5974	2560.6509

Metric	Sum	Min	Max	Avg	%ile 25	%ile 50	%ile 75
Runtime (RDTSC) [s] STAT	592.7915	147.6779	148.7171	148.1979	147.6779	147.6825	148.7140
Runtime unhalted [s] STAT	486.7575	121.4966	121.8564	121.6894	121.4966	121.6676	121.7369
Clock [MHz] STAT	7977.7121	1993.5457	1995.4260	1994.4280	1993.5457	1994.2933	1994.4471
CPI STAT	27.7833	6.7678	7.0770	6.9458	6.7678	6.9338	7.0047
DP [MFLOP/s] STAT	9891.8054	2322.8217	2560.6509	2472.9514	2322.8217	2454.5974	2553.7354

Debug output

Pure MPI, 2 nodes, 2 ranks per node, 1 per socket (processor)

```
EXEC (Rank 0): likwid-perfctr -s 0x0 -C L:N:0-0 -g
ACTUAL_CPU_CLOCK:FIXC1,MAX_CPU_CLOCK:FIXC2,RETIRED_INSTRUCTIONS:PMC0,CPU_CLOCKS_UNHALTED:PMC1,RETIRED_SSE_AVX_FLO
PS_ALL:PMC2,MERGE:PMC3 -o temporary1.csv cp2k

EXEC (Rank 1): likwid-perfctr -s 0x0 -C L:N:0-0 -g
ACTUAL_CPU_CLOCK:FIXC1,MAX_CPU_CLOCK:FIXC2,RETIRED_INSTRUCTIONS:PMC0,CPU_CLOCKS_UNHALTED:PMC1,RETIRED_SSE_AVX_FLO
PS_ALL:PMC2,MERGE:PMC3 -o temporary2.csv cp2k

EXEC: srun --ntasks-per-node=2 --ntasks=4 --mpi=cray_shasta --nodes=2 .likwidscript_122134.txt
```

.likwidscript_122134.txt (bash script):

myNodeLocalRank = 0 or 1 on each node

if myNodeLocalRank == 0 then EXEC (Rank 0)
if myNodeLocalRank == 1 then EXEC (Rank 1)

Example job script
Pure OpenMP, fully populated node (128 threads)

```
#!/bin/bash

#SBATCH --account=z19
#SBATCH --partition=standard
#SBATCH --qos=short
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=128
#SBATCH --hint=nomultithread
#SBATCH --distribution=block:block

module load likwid
module load xthi

export OMP_NUM_THREADS=128
export OMP_PLACES=cores
export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK

likwid-mpirun -n 1 -t 128 --nocpubind -s 0x0 -g FLOPS_DP --debug xthi &> xthi.out
```

Debug output
Pure OpenMP, fully populated node (128 threads)

```
EXEC (Rank 0): likwid-perfctr -s 0x0 -C L:N:0-127 -g
ACTUAL_CPU_CLOCK:FIXC1,MAX_CPU_CLOCK:FIXC2,RETIRED_INSTRUCTIONS:PMC0,CPU_CLOCKS_UNHALTED:PMC1,RETIRED_SSE_AVX_FLO
PS_ALL:PMC2,MERGE:PMC3 -o temporary1.csv xthi

EXEC: srun  --ntasks=1 --mpi=cray_shasta --ntasks-per-node=1 --nodes=1 .likwidscript_231548.txt
```

Don't need likwid-mpirun!

Example job script using likwid-perfctr
Pure OpenMP, fully populated node (128 threads)

```
#!/bin/bash

#SBATCH --account=z19
#SBATCH --partition=standard
#SBATCH --qos=short
#SBATCH --time=00:05:00
#SBATCH --nodes=1

module load likwid
module load xthi

export OMP_NUM_THREADS=128
export OMP_PLACES=cores

likwid-perfctr -C N:0-127 -s 0x0 -g FLOPS_DP xthi &> xthi.out

# Alternatively:
likwid-perfctr -C E:N:128:1:2 -s 0x0 -g FLOPS_DP xthi &> xthi.out
```

Example job script
Pure OpenMP, 4 threads, 1 per CCX

```
#!/bin/bash

#SBATCH --account=z19
#SBATCH --partition=standard
#SBATCH --qos=short
#SBATCH --time=00:05:00
#SBATCH --nodes=1

module load likwid
module load xthi

export OMP_NUM_THREADS=4
export OMP_PLACES=cores

likwid-perfctr -C 0,4,8,12 -s 0x0 -g FLOPS_DP xthi &> xthi.out

# Alternatively:
likwid-perfctr -C C0:0@C1:0@C2:0@C3:0 -s 0x0 -g FLOPS_DP xthi &> xthi.out

# Alternatively:
likwid-perfctr -C E:N:4:1:8 -s 0x0 -g FLOPS_DP xthi &> xthi.out
```

Application (xthi) output
Pure OpenMP, 4 threads, 1 per CCX

```
Node summary for    1 nodes:
Node    0, hostname nid001380, mpi   1, omp   4, executable xthi
MPI summary: 1 ranks
Node    0, rank    0, thread   0, (affinity =    0)
Node    0, rank    0, thread   1, (affinity =    4)
Node    0, rank    0, thread   2, (affinity =    8)
Node    0, rank    0, thread   3, (affinity =   12)
```

likwid-perfctr results for application (cp2k H2O-32)

Pure OpenMP, 4 threads, 1 per CCX

Event	Counter	HWThread 0	HWThread 4	HWThread 8	HWThread 12
ACTUAL_CPU_CLOCK	FIXC1	288225142494	277342582674	276504761957	275708683034
MAX_CPU_CLOCK	FIXC2	324520870098	312204162837	311334613420	310553053741
RETIRED_INSTRUCTIONS	PMC0	642746850133	597585919597	590879312097	591524771907
CPU_CLOCKS_UNHALTED	PMC1	4579460146688	4429904890720	4409955410336	4404089955280
RETIRED_SSE_AVX_FLOPS_ALL	PMC2	380268064831	368436346965	360781276795	359314236216
MERGE	PMC3	5348024557502464	5348024557502464	5207287069147136	5207287069147136

Event	Counter	Sum	Min	Max	Avg
ACTUAL_CPU_CLOCK STAT	FIXC1	1117781170159	275708683034	288225142494	2.794453e+11
MAX_CPU_CLOCK STAT	FIXC2	1258612700096	310553053741	324520870098	314653175024
RETIRED_INSTRUCTIONS STAT	PMC0	2422736853734	590879312097	642746850133	6.056842e+11
CPU_CLOCKS_UNHALTED STAT	PMC1	17823410403024	4404089955280	4579460146688	4455852600756
RETIRED_SSE_AVX_FLOPS_ALL STAT	PMC2	1468799924807	359314236216	380268064831	3.672000e+11
MERGE STAT	PMC3	21110623253299200	5207287069147136	5348024557502464	5277655813324800

Metric	HWThread 0	HWThread 4	HWThread 8	HWThread 12
Runtime (RDTSC) [s]	158.7059	158.7059	158.7059	158.7059
Runtime unhalted [s]	128.3473	123.5013	123.1282	122.7737
Clock [MHz]	1994.5010	1994.9081	1994.4366	1993.6993
CPI	7.1248	7.4130	7.4634	7.4453
DP [MFLOP/s]	2396.0556	2321.5044	2273.2701	2264.0263

Metric	Sum	Min	Max	Avg
Runtime (RDTSC) [s] STAT	634.8236	158.7059	158.7059	158.7059
Runtime unhalted [s] STAT	497.7505	122.7737	128.3473	124.4376
Clock [MHz] STAT	7977.5450	1993.6993	1994.9081	1994.3863
CPI STAT	29.4465	7.1248	7.4634	7.3616
DP [MFLOP/s] STAT	9254.8564	2264.0263	2396.0556	2313.7141

Example job script

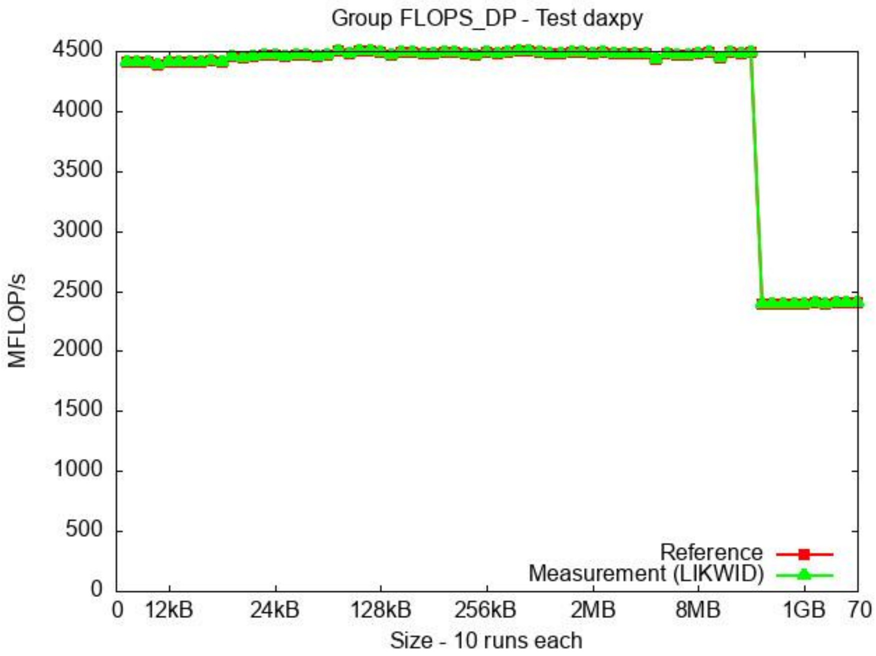
Hybrid MPI + OpenMP, 1 node, 2 ranks per node, 64 threads per rank

```
#!/bin/bash

#SBATCH --job-name=likwid
#SBATCH --account=z19
#SBATCH --partition=standard
#SBATCH --qos=short
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --tasks-per-node=2
#SBATCH --cpus-per-task=64
#SBATCH --hint=nomultithread
#SBATCH --distribution=block:block

module load likwid
module load xthi

export SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export OMP_PLACES=cores

likwid-mpirun -n $SLURM_NTASKS -t 64 --nocpubind -s 0x0 -g FLOPS_DP --debug xthi &> xthi.out

# Alternative:
likwid-mpirun -n $SLURM_NTASKS -pin N:0-63_N:64-127 --nocpubind -s 0x0 -g FLOPS_DP --debug xthi &> xthi.out
```

Microbenchmarking

Validation through microbenchmarking

- likwid-bench provides a microbenchmark suite
- Tests for different performance groups
- Assembly microkernels with known number of instructions for validation of values measured with likwid-perfctr
- Provide estimates of performance upper bounds based on concrete hardware performance rather than theoretical, input to roofline analysis
- Can be useful for validating during prototyping of own assembly kernels



Marker API

Using LIKWID MARKER API

- Main use case: *monitor regions of interest at fine granularity*
- Need to recompile with `-DLIKWID_PERFMON`
- and with `-I $LIKWID_DIR/include`
- Link with `-L $LIKWID_DIR/lib -llikwid` `and` adjust `LD_LIBRARY_PATH`
- Uses the `LD_PRELOAD` mechanism, so the application must be *dynamically linked*
- Run likwid-perfctr or likwid-mpirun with the **-m** flag
- Avoid calling very frequently: calls add overhead
- Can define multiple regions

Using LIKWID MARKER API

- Core C API (API also define for Fortran and a few other language)

```
… // other includes
#include <likwid-marker.h>
…
LIKWID_MARKER_INIT; // macro call to setup measurement system
…
LIKWID_MARKER_REGISTER("myregion"); // to reduce overhead
// if OpenMP is used: called the above and start/stop
// in a parallel region
LIKWID_MARKER_START("myregion");
… // code region of interest
LIKWID_MARKER_STOP("myregion");
…
LIKWID_MARKER_CLOSE;
```

Serial hello world example output (w -g MEM)

```
--------------------------------------------------------------------------------
CPU name:    AMD EPYC 7742 64-Core Processor
CPU type:    AMD K17 (Zen2) architecture
CPU clock:    2.25 GHz
--------------------------------------------------------------------------------
Hello world
--------------------------------------------------------------------------------
Region hello, Group 1: MEM
+-------------------+------------+
|    Region Info    | HWThread 0 |
+-------------------+------------+
| RDTSC Runtime [s] |   0.000013 |
|     call count    |          1 |
+-------------------+------------+

+-----------------------+---------+------------+
|         Event         | Counter | HWThread 0 |
+-----------------------+---------+------------+
|    ACTUAL_CPU_CLOCK   |  FIXC1  |     114358 |
|      MAX_CPU_CLOCK    |  FIXC2  |      75082 |
|  RETIRED_INSTRUCTIONS |   PMC0  |       7011 |
|   CPU_CLOCKS_UNHALTED |   PMC1  |      14666 |
|     DRAM_CHANNEL_0    |   DFC0  |        447 |
|     DRAM_CHANNEL_1    |   DFC1  |        524 |
+-----------------------+---------+------------+

+------------------------------+--------------+
|            Metric            |  HWThread 0  |
+------------------------------+--------------+
|      Runtime (RDTSC) [s]     | 1.315996e-05 |
|      Runtime unhalted [s]    |       0.0001 |
|          Clock [MHz]         |    3426.9943 |
|              CPI             |       2.0919 |
|  Memory bandwidth [MBytes/s] |    4722.2028 |
|  Memory data volume [GBytes] |       0.0001 |
+------------------------------+--------------+
```

Roofline analysis

Roofline (Williams et al., 2009, CACM 52(4):65-76)

Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures*

Samuel Williams, Andrew Waterman, and David Patterson

Parallel Computing Laboratory, 565 Soda Hall, U.C. Berkeley, Berkeley, CA 94720-1776, 510-642-6587
samw, waterman, pattrsn@eecs.berkeley.edu

ABSTRACT
We propose an easy-to-understand, visual performance model that offers insights to programmers and architects on improving parallel software and hardware for floating point computations.

1. INTRODUCTION
Conventional wisdom in computer architecture led to homogeneous designs. Nearly every desktop and server computer uses caches, pipelining, superscalar instruction issue, and out-of-order execution. Although the instruction sets varied, the microprocessors were all from the same school of design.

The switch to multicore means that microprocessors will become more diverse, since there is no conventional wisdom yet for them. For example, some offer many simple processors versus fewer complex processors, some depend on multithreading, and some even replace caches with explicitly addressed local stores. Manufacturers will likely offer multiple products with differing number of cores to cover multiple price-performance points, since the cores per chip will likely double every two years [4].

While diversity may be understandable in this time of uncertainty, it exacerbates the already difficult job of programmers, compiler writers, and even architects. Hence, an easy-to-understand model that offers performance guidelines could be especially valuable.

A model need not be perfect, just insightful. For example, the 3Cs model for caches is an analogy [19]. It is not a perfect model,

limited by the serial portion of a parallel program. It has been recently applied to heterogeneous multicore computers [4][18].

3. THE ROOFLINE MODEL
We believe that for the recent past and foreseeable future, off-chip memory bandwidth will often be the constraining resource[23]. Hence, we want a model that relates processor performance to off-chip memory traffic.

Towards that goal, we use the term *operational intensity* to mean operations per byte of DRAM traffic. We define total bytes accessed as those that go to the main memory *after* they have been filtered by the cache hierarchy. That is, we measure traffic between the caches and memory rather than between the processor and the caches. Thus, *operational intensity* suggests the DRAM bandwidth needed by a kernel on a particular computer.

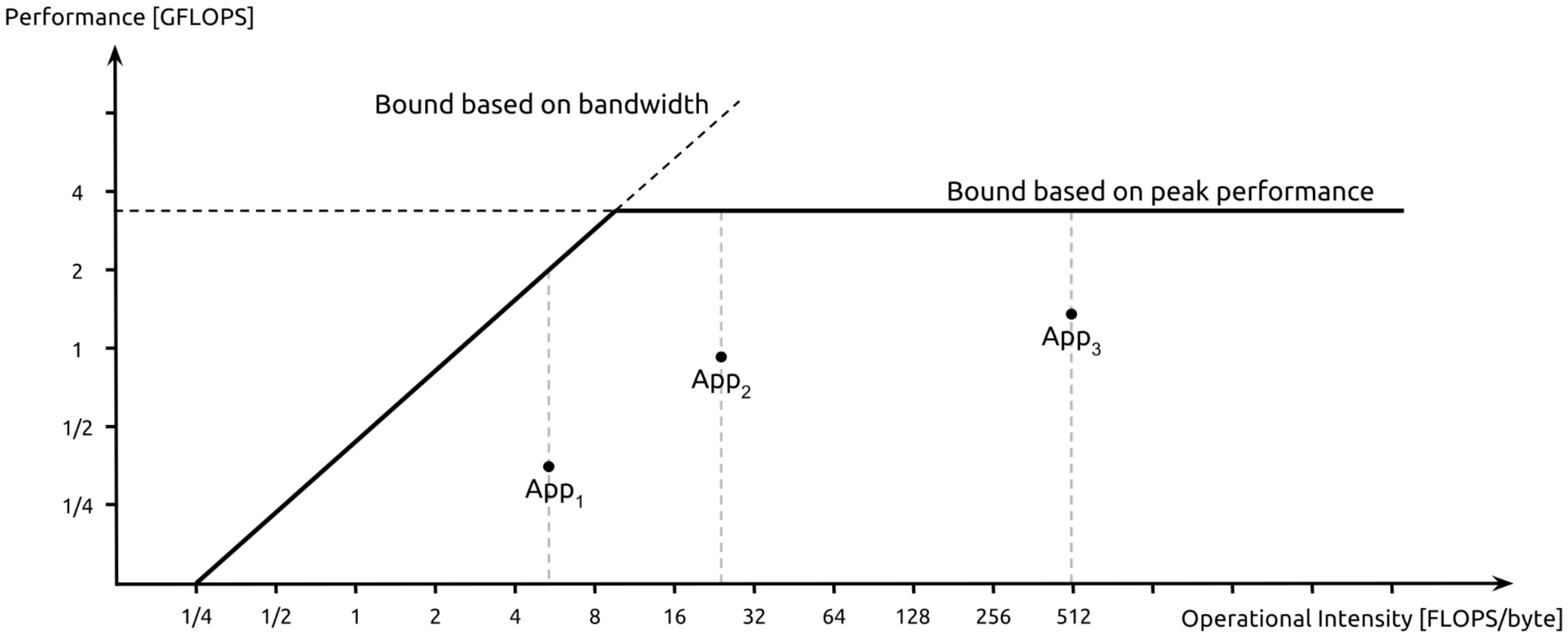
We use operational intensity instead of the terms *arithmetic intensity* [16] or *machine balance* [8][11] for two reasons. First, arithmetic intensity and machine balance measure traffic between the processor and cache, whereas we want to measure traffic between the caches and DRAM. This subtle change allows us to include memory optimizations of a computer into our bound and bottleneck model. Second, we think the model will work with kernels where the operations are not arithmetic (see Section 7), so we needed a more general term than arithmetic.

The proposed model ties together floating-point performance,

Roofline Analysis with LIKWID

- Roofline model is an intuitive visual performance model
- Performance is limited by peak attainable memory performance (e.g. obtained by running the STREAM benchmark) and by peak FLOPS which define the limiting Rooflines
- Y-axis: performance (FLOPS)
- X-axis operational intensity of the application (FLOPS/byte)
- Can plot applications, functions or loops as points
- Distance from the Roofline represents optimisation potential
- There can be multiple Rooflines (e.g. for single vs double precision)

Roofline Analysis with LIKWID



[image from: https://en.wikipedia.org/wiki/Roofline_model]

Using likwid-bench for Roofline analysis

- Roofline does only tell that a loop is memory or compute bound and shows roughly optimisation potential, but does not tell what those optimisations are

- Get memory Roofline by running a version of STREAM or estimate from number of DRAM channels and frequency (or use likwid-bench)

- Get compute Roofline: can roughly estimate by using number of cores times number of operations (from CPU frequency)

- Measure your application (as further above) or region of interest (with Marker API)

- Select optimisations based on the resulting position of the point

Conclusions

- LIKWID is a powerful yet lightweight and portable tool suite with focus on performance engineering
- Allows to measure hardware performance counters on ARCHER2
- We encourage you to give it a go!
- To get started use the ARCHER2 documentation: https://docs.archer2.ac.uk/data-tools/likwid/ and The official LIKWID Wiki for more in depth information: https://github.com/RRZE-HPC/likwid/wiki

Q: overheads?

Counters implemented in hardware so virtually no overheads, but possible overheads in (post-execution?) aggregation - not sure how scales to very large process counts

https://ieeexplore.ieee.org/document/7103452 (also includes comparison with PAPI)

Overhead from marker API might be significant if very frequent calls (entry/exit of instrumented regions)

Q: LIKWID vs PAPI?

https://arxiv.org/pdf/1004.4431

	LIKWID	PAPI
Dependencies	Needs system headers of Linux 2.6 kernel. No other external dependencies.	Needs kernel patches depending on platform and architecture. No patches necessary on Linux kernels $> 2.6.31$.
Installation	Build system based on make only. Install documentation 10 lines. Build configuration in a single text file (21 lines).	Install documentation is 582 lines (3.7.2) and 397 lines (4.0.0). The installation of PAPI for this comparison was not without problems.
Command line tools	Core is a collection of command line tools which are intended to be used standalone.	Collection of small utilities. These utilities are not supposed to be used as standalone tools. There are many PAPI-based tools available from other sources.
User API support	Simple API for configuring named code regions. API only turns counters on and off. Configuration of events and output of results is still based on the command line tool.	Comparatively high-level API. Events must be configured in the code.
Library support	While it can be used as library this was not initially intended.	Mature and well tested library API for building own tooling.
Topology information	Listing of thread and cache topology. Results are extracted from cpuid and presented in and accessible way as text and ASCII art. Nondata caches are omitted. No output of TLB information.	Information also based on cpuid. Utility outputs all caches (including TLBs). No output of shared cache information. Thread topology only as accumulated counts of HW threads and Cores. No mapping from processor Ids to thread topology.
Thread and process pinning	There is a dedicated tool for pinning processes and threads in a portable and simple manner. This tool is intended to be used together with likwid-perfCtr	No support for pinning.
Multicore support	Multiple cores can be measured simultaneously. Binding of threads or processes to correct cores is the responsibility of the user.	No explicit support for multicore measurements.
Uncore support	Uncore events are handled by applying socket locks, which prevent multiple measurements in threaded mode.	No explicit support for measuring shared resources.
Event abstraction	Preconfigured event sets (so-called event groups) with derived metrics.	Abstraction through papi_events, which map to native events.
Platform support	Supports only x86-based processors on Linux with 2.6 kernel.	Supports a wide range of architectures on various platforms (dedicated support for HPC systems like BlueGene or Cray XT3/4/5) with various operating systems (Linux, FreeBSD, and Windows).
Correlated measurements	LIKWID can measure performance counters only	PAPI-C can be extended to measure and correlate various data like, e.g., fan speeds or temperatures.

Table I: Comparison between LIKWID and PAPI