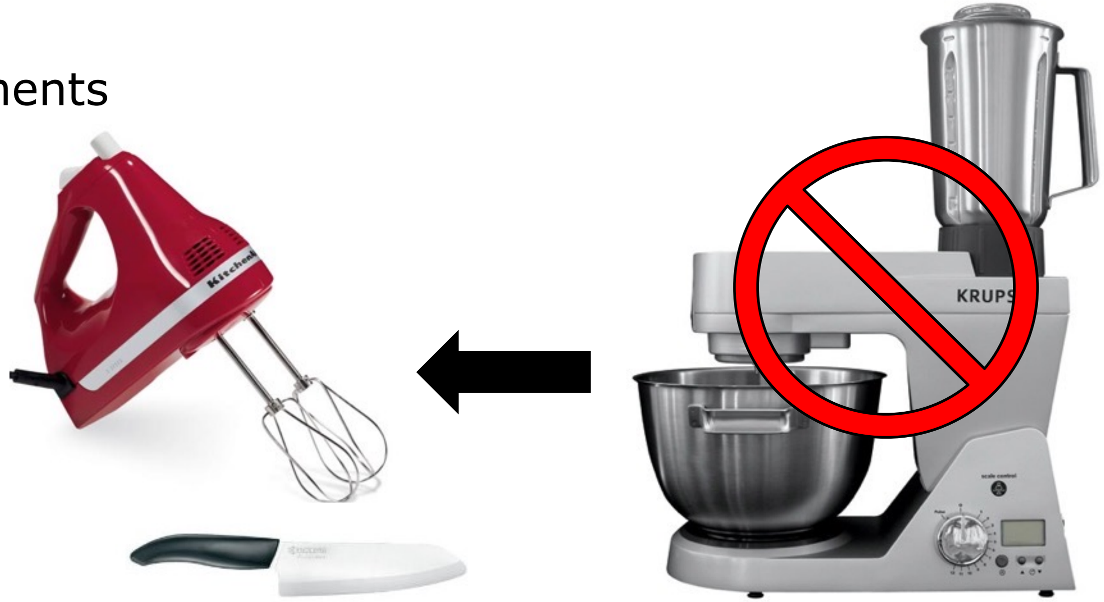
LIKWID
Like I Knew What I'm Doing

Thomas Gruber
thomas.gruber@fau.de
NHR@FAU, University Erlangen-Nürnberg

What is LIKWID?

- Toolbox for performance-oriented programmers and users
- Support for x86 CPUs, ARM CPUs, POWER CPUs, Nvidia GPUs and AMD GPUs

- Tools for the daily life:
  - Read the hardware topology
  - Flexible and comprehensive pinning functionality
  - Easy-to-use hardware performance event measurements
  - Runtime system cleanup
  - System adaption
  - Micro-benchmarking



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

LIKWID Tools Overview

- `likwid-topology` – Read topology of current system
- `likwid-pin` – Pin threads to CPU cores
- `likwid-perfctr` – Hardware performance monitoring (HPM) tool
- `likwid-powermeter` – Measure energy consumption
- `likwid-mpirun` – Pin and hardware performance monitoring for MPI+X applications

- `likwid-memsweeper` – Clean up filesystem cache and LLC
- `likwid-setFrequencies` – Manipulate CPU/Uncore frequency
- `likwid-features` – Manipulate hardware settings (prefetchers)
- `likwid-bench` – Microkernel benchmark tool

likwid-topology

```
$ likwid-topology
--------------------------------------------------------------------------------
CPU name:  Intel(R) Xeon(R) Platinum 8468
CPU type:  Intel SapphireRapids processor
CPU stepping:      8
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:           2
Cores per socket:  48
Threads per core:  1
--------------------------------------------------------------------------------
HWThread      Thread      Core      Die      Socket      Available
0             0           0         0        0           *
1             0           1         0        0           *
2             0           2         0        0           *
[…]
94            0           94        0        1           *
95            0           95        0        1           *
--------------------------------------------------------------------------------
Socket 0:          ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 […] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 )
Socket 1:          ( 48 49 50 51 52 53 54 55 56 57 […] 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 )
--------------------------------------------------------------------------------
```

Basic system layout
No SMT

HW threads
available in CPU set

likwid-topology

```
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   48 kB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 ) […] ( 94 ) ( 95 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   2 MB
Cache groups:           ( 0 ) ( 1 ) ( 2 ) ( 3 ) […] ( 94 ) ( 95 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   105 MB
Cache groups:           ( 0 1 2 3 4 5 6 7 8 9 10 11 12 […] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 )
                        ( 48 49 50 51 52 53 54 55 56 57 […] 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 )
--------------------------------------------------------------------------------
```

L1 and L2 cache private for cores

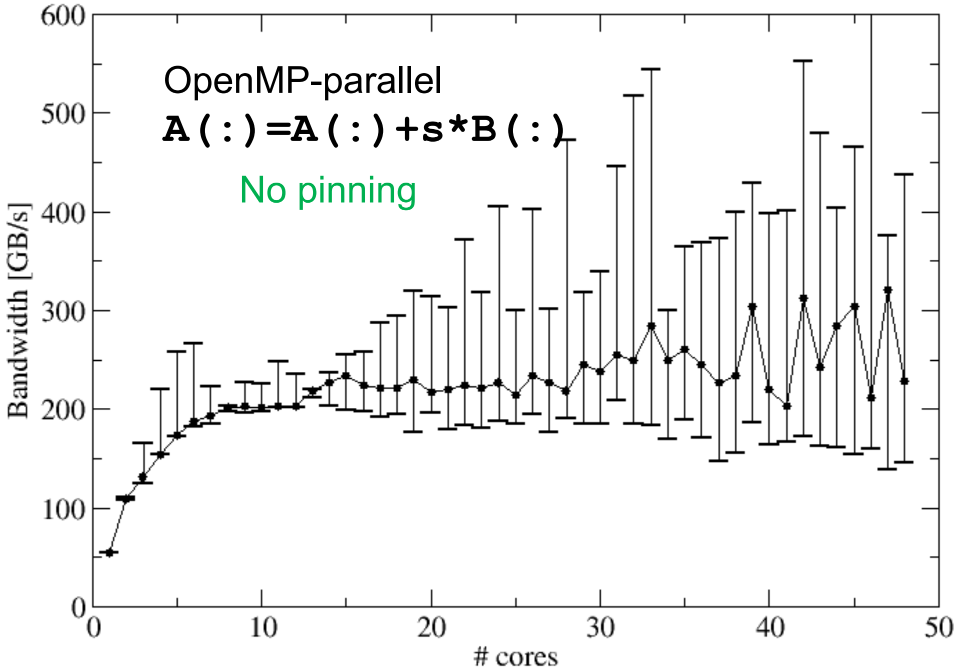
L3 shared by all cores of a CPU socket

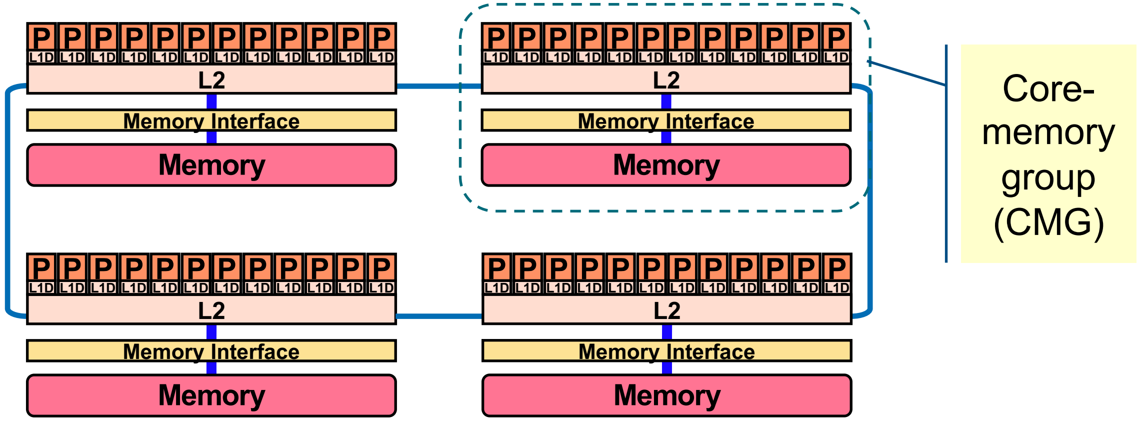
likwid-topology

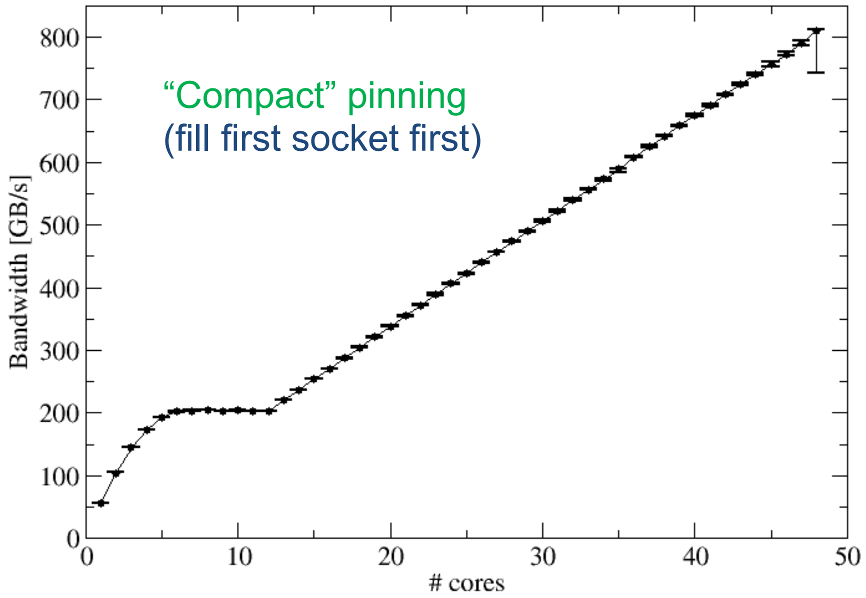
```
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:                2
--------------------------------------------------------------------------------
Domain:                      0
Processors:                  ( 0 1 2 3 4 5 6 7 8 9 10 11 12 […] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 )
Distances:                   10 21
Free memory:                 238216 MB
Total memory:                257091 MB
--------------------------------------------------------------------------------
Domain:                      1
Processors:                  ( 48 49 50 51 52 53 54 55 56 […] 74 75 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 )
Distances:                   21 10
Free memory:                 244817 MB
Total memory:                257977 MB
--------------------------------------------------------------------------------
```

#NUMA nodes == #CPU sockets
→ No SNC mode

DAXPY test on A64FX - *Anarchy vs. thread pinning*



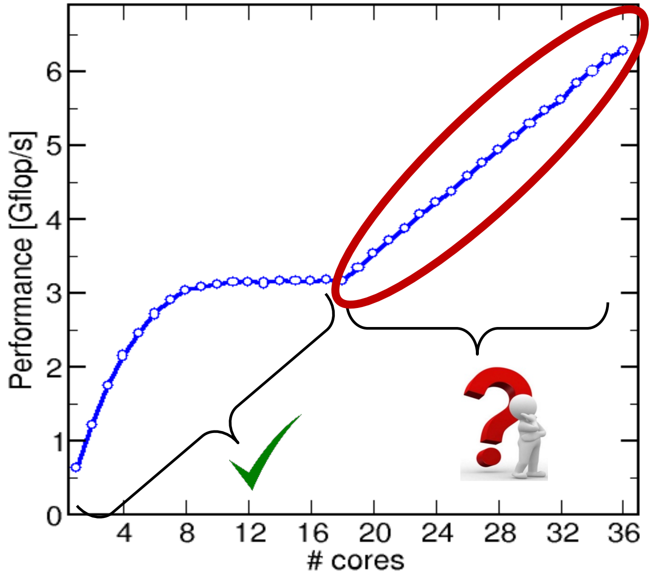




There are several reasons for caring about affinity:

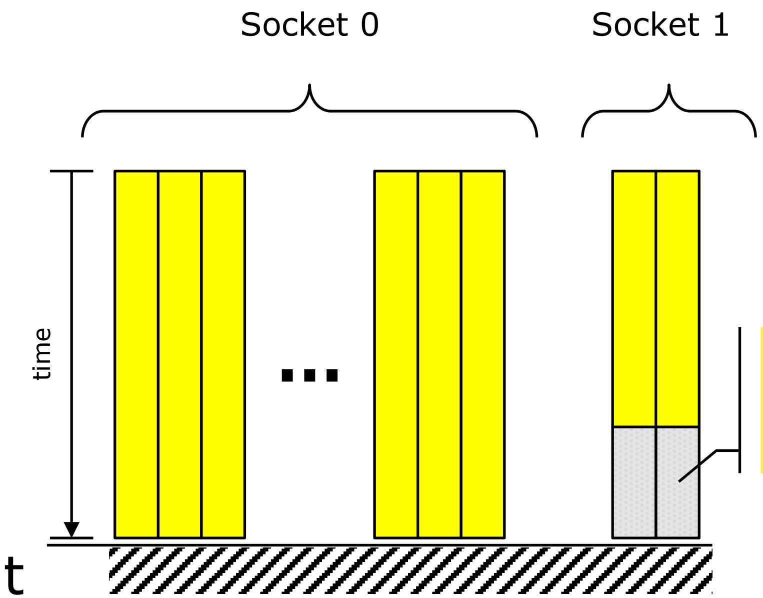
- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention

Interlude: Why the weird scaling behavior?




```
!$omp parallel do schedule(static)
 do i = 1,N
   a(i) = b(i) + s * c(i)
!$omp end parallel do
```

implicit barrier



- Every thread has the same workload
- Performance of left socket is saturated
- Barrier enforces waiting of "speeders" at sync point
- Average performance of each "right" core == average performance of each "left" core
  → linear scaling

likwid-pin

- Pins processes and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Based on combination of wrapper tool together with overloaded pthread library
  → binary must be dynamically linked!
- Supports logical core numbering within topological entities (thread domains)
- Simple usage with physical (kernel) core IDs:

```
$ likwid-pin -c 0-3,4,6  ./myApp parameters
$ OMP_NUM_THREADS=4 likwid-pin -c 0-9 ./myApp params
```

- Simple usage with logical core IDs (“thread groups”):

```
$ likwid-pin -c S0:0-7 ./myApp params
$ likwid-pin –c C1:0-2 ./myApp params
```

LIKWID terminology: Thread group syntax

- The OS numbers all hardware threads (called "processors" in the OS) on a node
- The numbering is enforced at boot time by the BIOS
- LIKWID introduces thread domains consisting of HW threads sharing a topological entity (e.g. socket or shared cache)
- A thread domain is defined by a single character + index

Physical HW threads first!

- Example for likwid-pin:

```
$ likwid-pin -c S0:0-3 ./a.out
```

```
+-----------------------------------------+
| +-------+ +-------+ +------+ +------+ |
| |  0  4 | |  1  5 | | 2  6 | | 3  7 | |
| +-------+ +-------+ +------+ +------+ |
+-----------------------------------------+
```

- Thread group expressions may be chained with @:

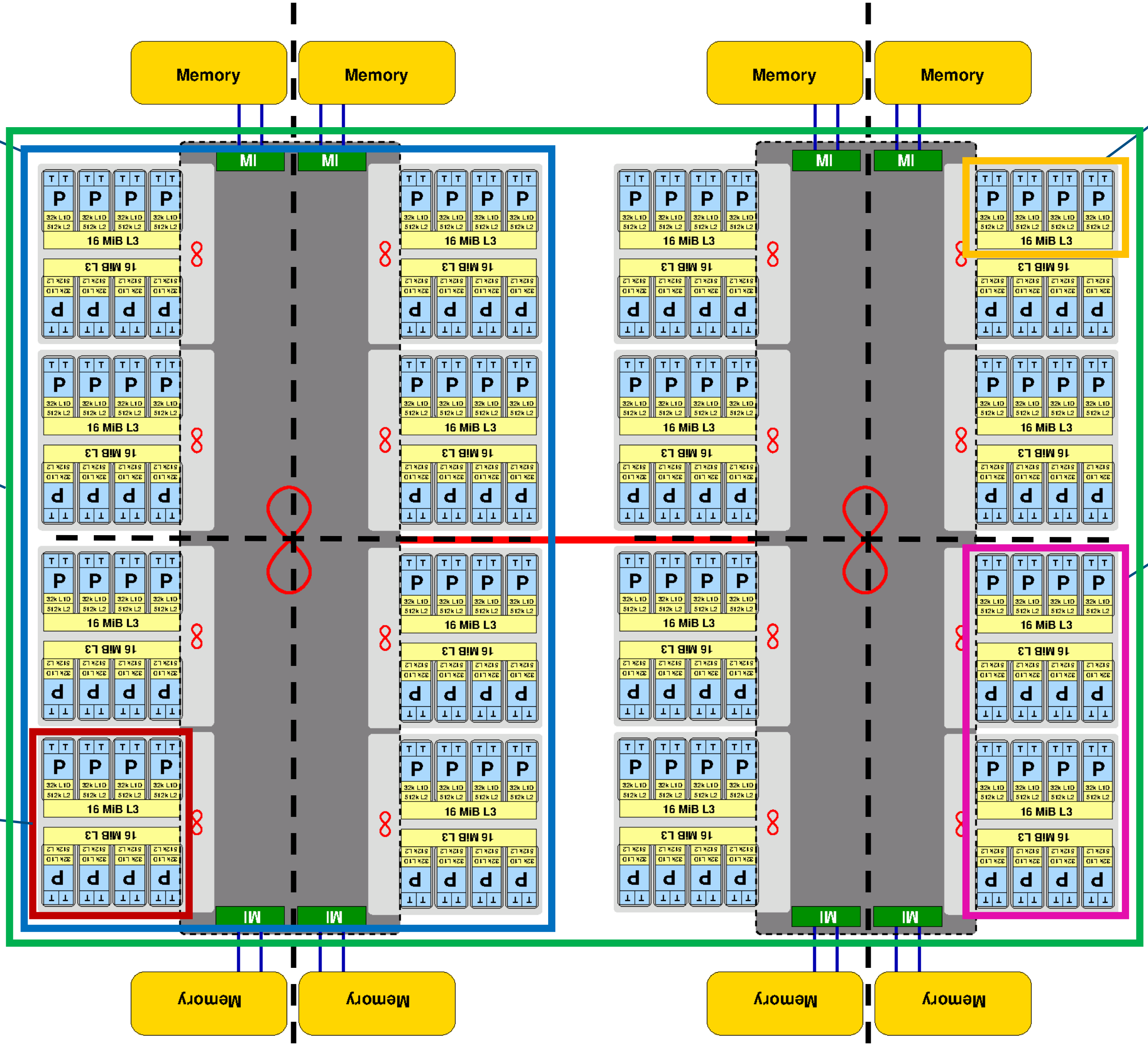
```
$ likwid-pin -c S0:0-2@S1:0-2 ./a.out
```

```
+-----------------------------------------++-----------------------------------------+
| +------+ +------+ +------+ +------+ || +------+ +------+ +------+ +------+ |
| | 0  8 | | 1  9 | | 2 10 | | 3 11 | || | 4 12 | | 5 13 | | 6 14 | | 7 15 | |
| +------+ +------+ +------+ +------+ || +------+ +------+ +------+ +------+ |
+-----------------------------------------++-----------------------------------------+
```

S socket

N node

D die/chip



C outer-level cache group

M ccNUMA domain

AMD Zen

Example: likwid-pin with Intel OpenMP

Running the STREAM benchmark with likwid-pin:

```
$ likwid-pin -c S0:0-3 ./stream
-------------------------------------------------
 Double precision appears to have 16 digits of accuracy
 Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------
 Array size =   20000000
 Offset     =         32
 The total memory requirement is  457 MB
 You are running each test  10 times
 The *best* time for each test is used
 *EXCLUDING* the first and last iterations
[pthread wrapper]
[pthread wrapper] MAIN -> 0
[pthread wrapper] PIN_MASK: 0->1  1->2  2->3
[pthread wrapper] SKIP MASK: 0x0
        threadid 47308666070912 -> core 1 - OK
        threadid 47308670273536 -> core 2 - OK
        threadid 47308674476160 -> core 3 - OK

  [... rest of STREAM output omitted ...]
```

Main PID always pinned

Pin all spawned threads in turn

MPI startup and hybrid pinning: `likwid-mpirun`

- How do you manage affinity with MPI or hybrid MPI/threading?
- In the long run a unified standard is needed
- Till then, likwid-mpirun provides a portable/flexible solution
- The examples here are for Intel MPI/OpenMP programs, but are also applicable to other threading models

Pure MPI:

```
$ likwid-mpirun -np 16 -nperdomain S:2 ./a.out
```

Hybrid:

```
$ likwid-mpirun -np 16 -pin S0:0,1_S1:0,1 ./a.out
$ likwid-mpirun –np 16 –nperdomain S:1 –t 2 ./a.out
```

likwid-perfctr

- How to find out about the performance properties and requirements of a parallel code?
- A coarse overview is often sufficient: `likwid-perfctr`

- Simple end-to-end measurement of hardware performance metrics

  Operating modes:
  - Wrapper
  - Stethoscope
  - Timeline
  - Marker API

- Preconfigured and extensible metric groups, list with `likwid-perfctr -a`

BRANCH	Branch prediction miss rate/ratio
FLOPS_DP	Double Precision MFLOP/s
FLOPS_SP	Single Precision MFLOP/s
L2	L2 cache bandwidth in MBytes/s
L3	L3 cache bandwidth in MBytes/s
MEM	Memory bandwidth in MBytes/s
HBM	HBM bandwidth in MBytes/s
DATA	Load to store ratio
ENERGY	Power and Energy consumption
CLOCK	Clock frequency of cores

likwid-perfctr wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./streamGCC
--------------------------------------------------------------------------------
CPU name:    Intel(R) Xeon(R) Platinum 8468
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
--------------------------------------------------------------------------------
Group 1: L2
+-----------------------+---------+--------------+--------------+--------------+--------------+
|         Event         | Counter | HWThread 48  | HWThread 49  | HWThread 50  | HWThread 51  |
+-----------------------+---------+--------------+--------------+--------------+--------------+
|   INSTR_RETIRED_ANY   |  FIXC0  |   1698636560 |   1394391389 |   1396329060 |   1412310223 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  |   2760330161 |   2662122582 |   2666168561 |   2663045233 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  |   1525237056 |   1470913248 |   1473137400 |   1471431696 |
|     TOPDOWN_SLOTS     |  FIXC3  |  16561980966 |  15972735492 |  15997011366 |  15978271398 |
|    L1D_REPLACEMENT    |   PMC0  |    145791228 |    142482665 |    142482099 |    142652375 |
+-----------------------+---------+--------------+--------------+--------------+--------------+
+-------------------------------+-------------+-------------+-------------+-------------+
|            Metric             | HWThread 48 | HWThread 49 | HWThread 50 | HWThread 51 |
+-------------------------------+-------------+-------------+-------------+-------------+
|      Runtime (RDTSC) [s]      |      0.8560 |      0.8560 |      0.8560 |      0.8560 |
|      Runtime unhalted [s]     |      1.3144 |      1.2677 |      1.2696 |      1.2681 |
|          Clock [MHz]          |   3800.5270 |   3800.6784 |   3800.7078 |   3800.6561 |
|              CPI              |      1.6250 |      1.9092 |      1.9094 |      1.8856 |
| L2D load bandwidth [MBytes/s] |  10900.7659 |  10653.3857 |  10653.3433 |  10666.0748 |
| L2D load data volume [GBytes] |      9.3306 |      9.1189 |      9.1189 |      9.1298 |
+-------------------------------+-------------+-------------+-------------+-------------+
```

Always measured for Intel CPUs

Configured metrics (this group)

Derived metrics

`likwid-perfctr` with MarkerAPI

- The MarkerAPI can restrict measurements to code regions
- The API only reads counters.
  The configuration of the counters is still done by `likwid-perfctr`
- Multiple named regions allowed, accumulation over multiple calls
- Inclusive and overlapping regions allowed

- Caveat: Marker API can cause overhead; do not call too frequently!

```
#include <likwid-marker.h>

LIKWID_MARKER_INIT; // must be called from serial region
. . .
LIKWID_MARKER_START("Compute");
. . .
LIKWID_MARKER_STOP("Compute");
. . .
LIKWID_MARKER_START("Postprocess");
. . .
LIKWID_MARKER_STOP("Postprocess");
. . .
LIKWID_MARKER_CLOSE; // must be called from serial region
```

likwid-perfctr with MarkerAPI: OpenMP code (C)

```
#include <likwid-marker.h>

int main(...) {
  LIKWID_MARKER_INIT;
  #pragma omp parallel
  {
    LIKWID_MARKER_REGISTER("MatrixAssembly");
  }
  ...
  #pragma omp parallel
  {
    LIKWID_MARKER_START("MatrixAssembly");
    #pragma omp for
    for(int i=0; i<N; ++i) { /* Loop */ }
    LIKWID_MARKER_STOP("MatrixAssembly");
  }
  ...
  LIKWID_MARKER_CLOSE;
}
```

Optional: Prepare data structures (reduced overhead on 1st marker call)

Call markers in parallel region if data should be taken on all threads

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

Compiling, linking, and running with marker API

Compile:

```
cc -I /path/to/likwid.h -DLIKWID_PERFMON -c program.c
```

Activate LIKWID macros (C only)

Link:

```
cc -L /path/to/liblikwid program.o -o program -llikwid
```

Run:

```
likwid-perfctr -C <CPULIST> -g <GROUP> -m ./program
```

MPI:

```
likwid-mpirun -np 4 -pin <PINEXPR> -g <GROUP> -m ./program
```

Activate markers

→ One separate block of output for every marked region

So… what should I look at first?

Focus on resource utilization and instruction decomposition!

Metrics to measure:

- Operation throughput (Flops/s)
- Overall instruction throughput (IPC,CPI)
- Instruction breakdown:
  - FP instructions
  - loads and stores
  - branch instructions
  - other instructions
- Instruction breakdown to SIMD width (scalar, SSE, AVX, AVX512 for x86)

- Data volumes and bandwidths to main memory (GB and GB/s)
- Data volumes and bandwidth to different cache levels (GB and GB/s)

Useful diagnostic metrics are:

- Clock frequency (GHz)
- Power (W)

All the above metrics can be acquired using performance groups:

`MEM_DP, MEM_SP, BRANCH, DATA, L2,  L3`

Example: triangular matrix-vector multiplication

```
#define N 10000 // matrix in memory
#define ROUNDS 10
// Initialization
fillMatrix(mat, N*N, M_PI);
fillMatrix(bvec, N, M_PI);

// Calculation loop
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
```

Prevent smart compilers from eliminating benchmark if **cvec** not used afterwards

Example: triangular matrix-vector multiplication

```
#include <likwid-marker.h>
[…] // defines, fillMatrix, init data
LIKWID_MARKER_INIT;
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        LIKWID_MARKER_START("Compute");
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        LIKWID_MARKER_STOP("Compute");
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
LIKWID_MARKER_CLOSE;
```

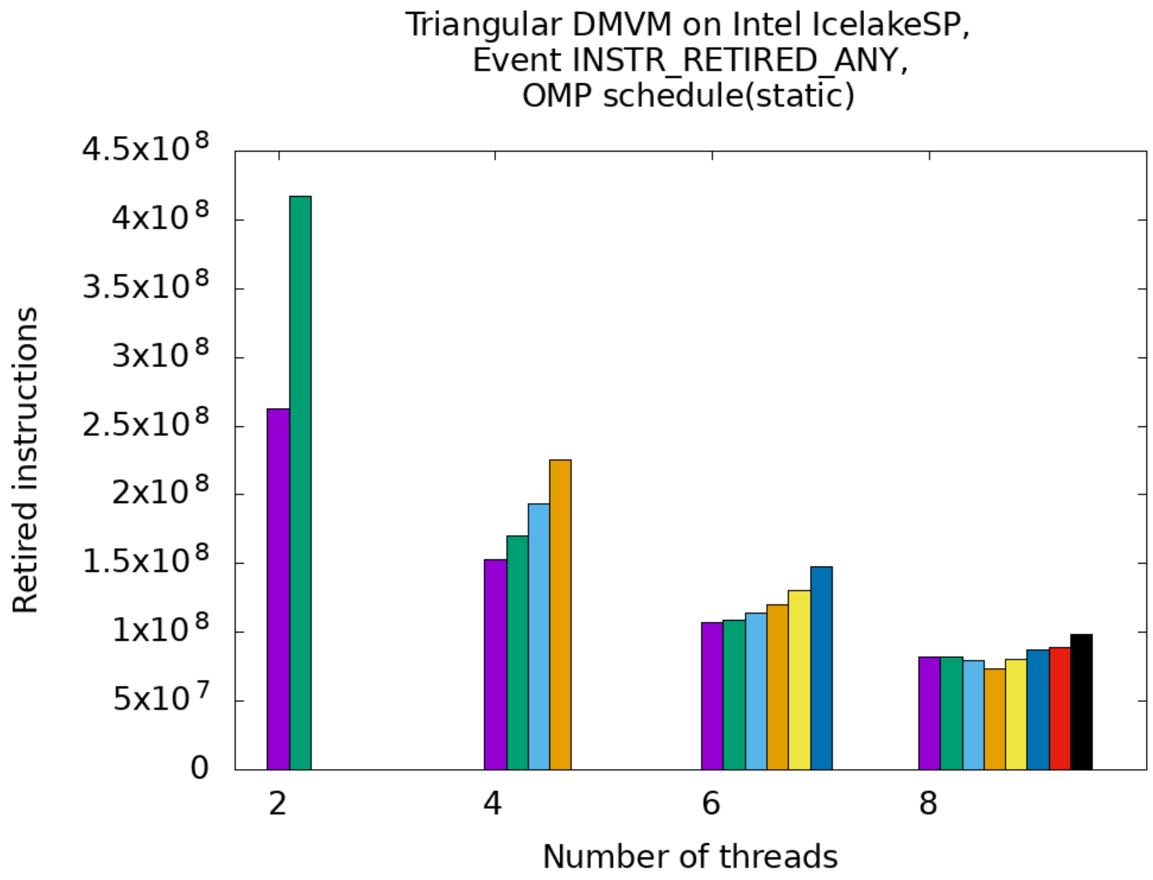
Example: triangular matrix-vector multiplication

```
$ likwid-perfctr -C 0,1,2 -g L2 -m ./a.out
--------------------------------------------------------------------------------
CPU type:  Intel Icelake SP processor
CPU clock: 2.39 GHz
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
Region Compute, Group 1: L2
+-------------------+------------+------------+------------+
|    Region Info    | HWThread 0 | HWThread 1 | HWThread 2 |
+-------------------+------------+------------+------------+
| RDTSC Runtime [s] |   0.198263 |   0.198364 |   0.198246 |
|     call count    |         10 |         10 |         10 |
+-------------------+------------+------------+------------+
+-----------------------+---------+------------+------------+------------+
|         Event         | Counter | HWThread 0 | HWThread 1 | HWThread 2 |
|   INSTR_RETIRED_ANY   |  FIXC0  |  194399400 |  269695800 |  341470000 |   ←???
| CPU_CLK_UNHALTED_CORE |  FIXC1  |  458193600 |  464605300 |  433236300 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  |  473442400 |  469863600 |  465054300 |
|     TOPDOWN_SLOTS     |  FIXC3  | 2290968000 | 2323026000 | 2166181000 |
|    L1D_REPLACEMENT    |   PMC0  |   69660770 |   41754150 |    7610321 |
|    L2_TRANS_L1D_WB    |   PMC1  |      43768 |     263047 |     442018 |
+-----------------------+---------+------------+------------+------------+
```

Example: triangular matrix-vector multiplication

- Retired instructions are misleading!
- Waiting in implicit OpenMP barrier executes many instructions

We need to measure actual work (or use a tool that can separate user from runtime lib instructions)

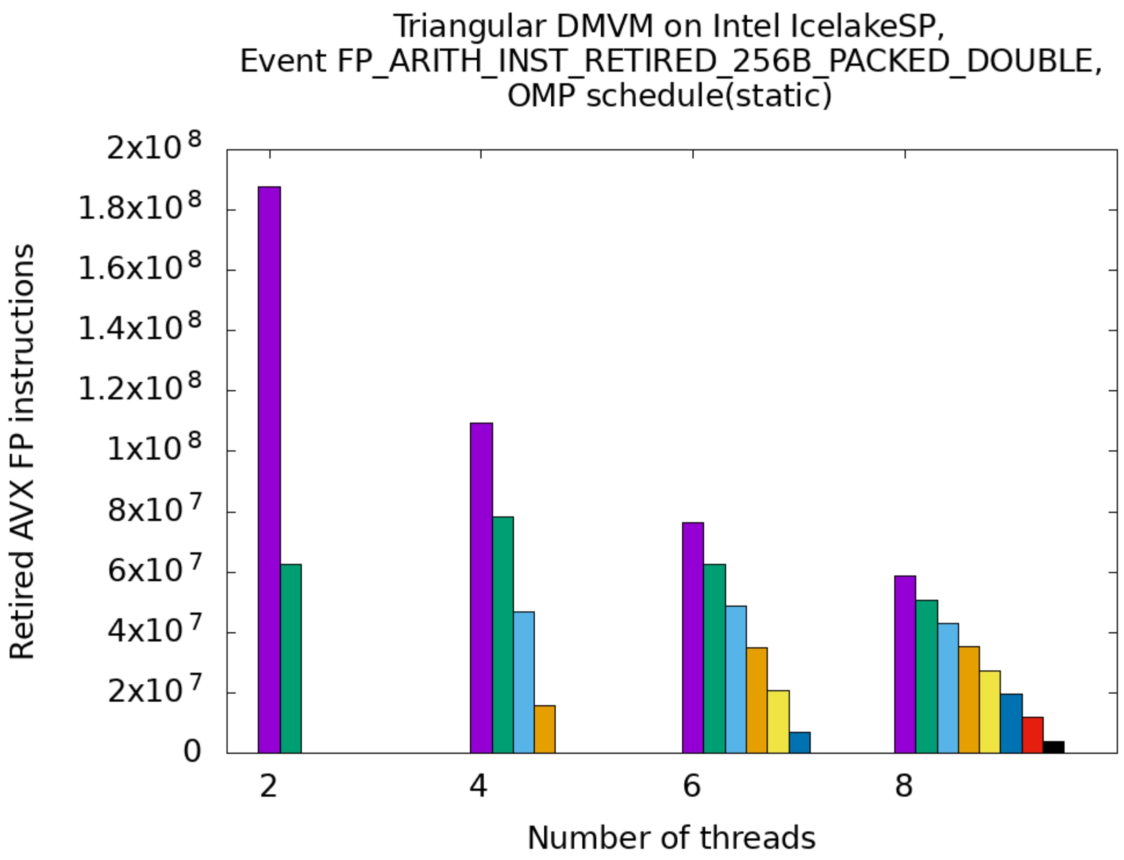


Example: triangular matrix-vector multiplication

- Use a metric for useful work
- Floating-point instructions reliable

Caveats:

- FP instruction counters from SandyBridge to Haswell are only qualitatively correct
- Masked SIMD lanes cannot be counted directly on Intel x86

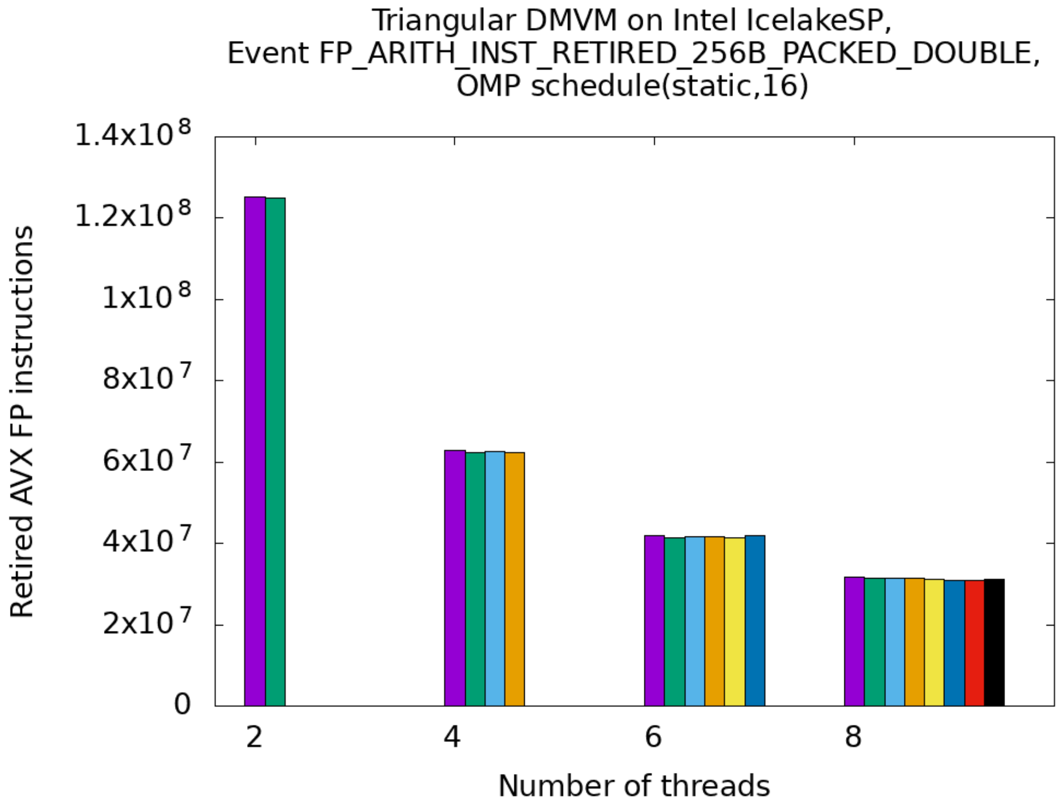


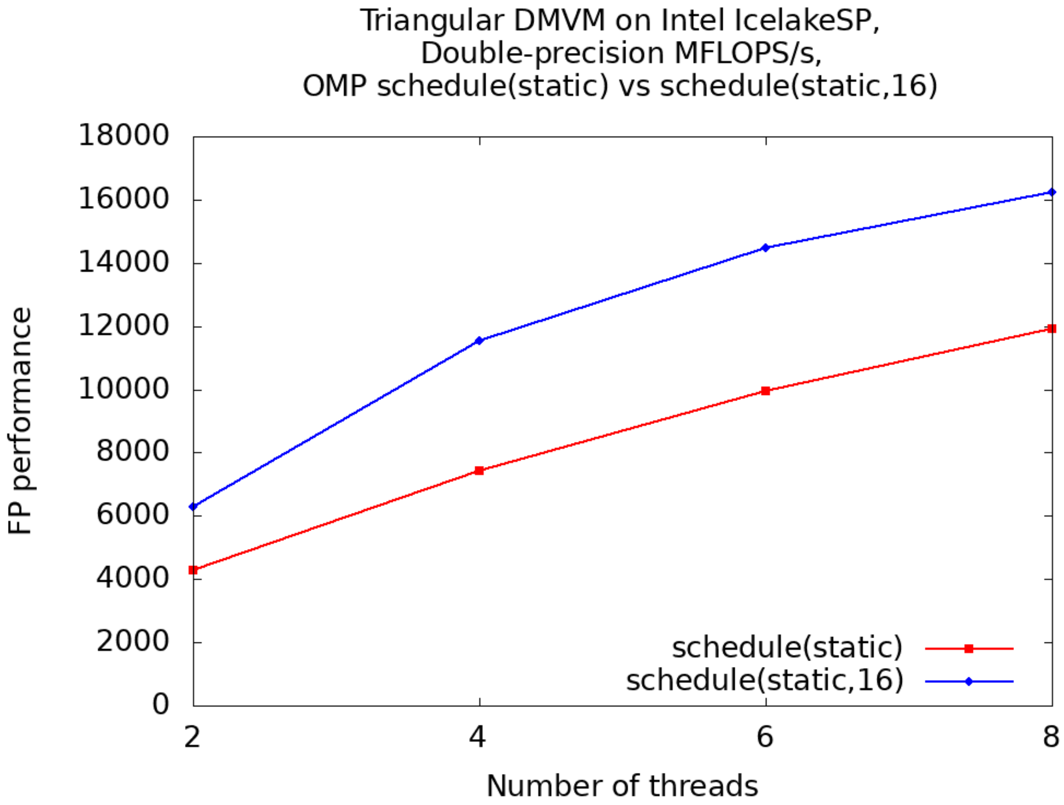
Example: triangular matrix-vector multiplication

Changing OMP schedule to static with chunk size 16 → smaller work packages per thread

No imbalance anymore!

Is it also faster?





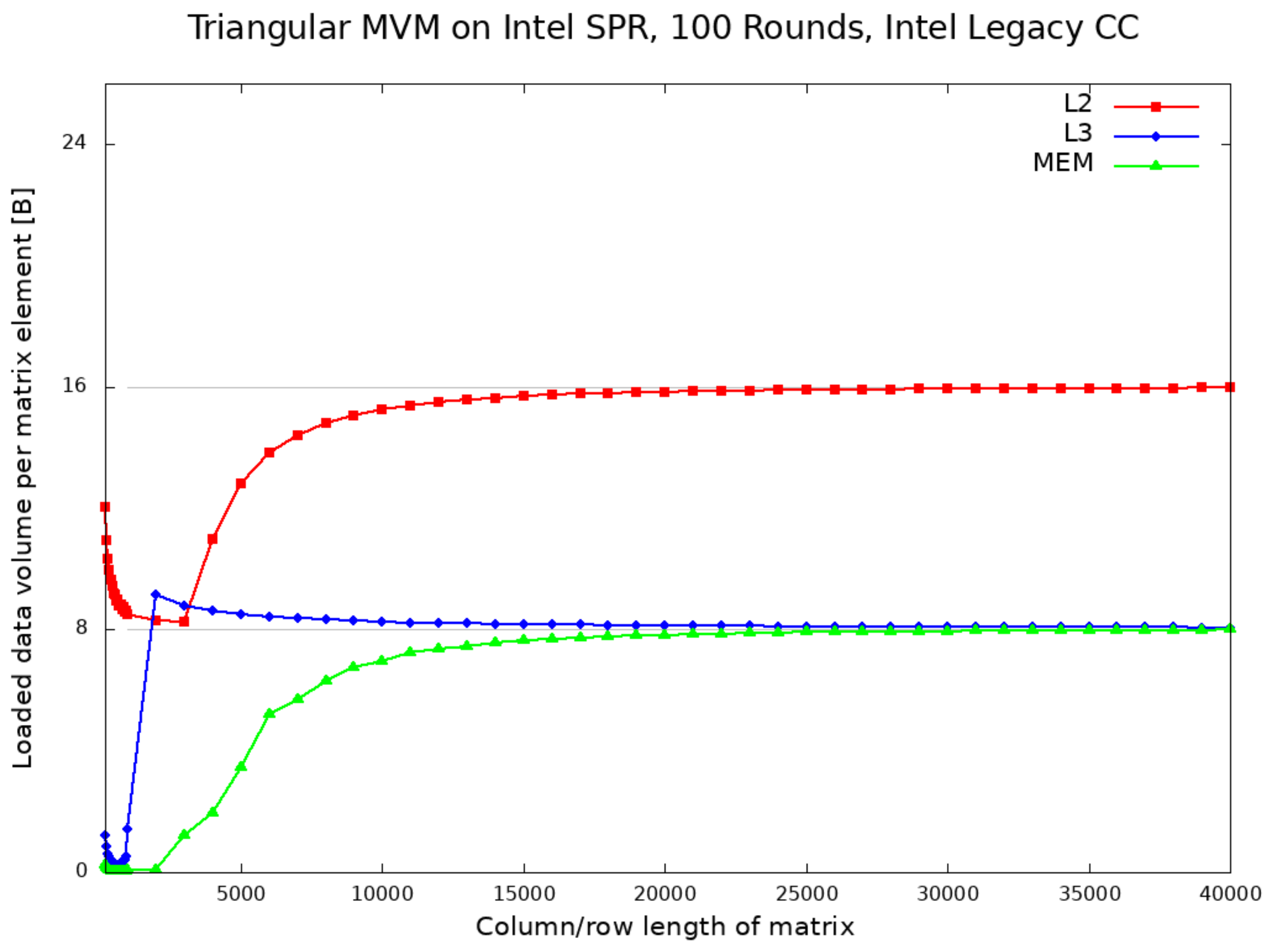
Example: triangular matrix-vector multiplication

```
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
```

- Load data traffic analysis:
  - Matrix elements: $(N * (N + 1))/2$
- Loaded data for each element:
  - Only matrix: 8 Byte
  - Matrix and `bvec`: 16 Byte
  - Matrix, `bvec`, and `cvec`: 24 Byte
- Formula:

  `DVol_in_bytes / (ROUNDS * ELEMENTS)`

Example: triangular matrix-vector multiplication



Summary of hardware performance monitoring

- Useful only if you know what you are looking for
  - Hardware event counting bears the potential of acquiring massive amounts of data for nothing!
- Resource-based metrics are most useful
  - Cache lines transferred, work executed, loads/stores, cycles
  - Instructions, CPI, cache misses may be misleading
- Caveat: Processor work != user work
  - Waiting time in libraries (OpenMP, MPI) may cause lots of instructions
  - → distorted application characteristic
- Another very useful application of PM: validating performance models!
  - Roofline is data centric → measure data volume through memory hierarchy