Multicore Performance and Tools

Part 2: Performance Analysis with hardware metrics

Probing performance behavior

- How do we find out about the performance properties and requirements of a parallel code? Profiling via advanced tools is often overkill
- 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
CACHE: Data cache miss rate/ratio
CLOCK: Clock frequency of cores
DATA: Load to store ratio
FLOPS_DP: Double Precision MFlops/s
FLOPS_SP: Single Precision MFlops/s
L2: L2 cache bandwidth in MBytes/s
L2CACHE: L2 cache miss rate/ratio
L3: L3 cache bandwidth in MBytes/s
L3CACHE: L3 cache miss rate/ratio
MEM: Main memory bandwidth in MBytes/s
TLB: TLB miss rate/ratio
ENERGY: Power and energy consumption
```

`likwid-perfctr` wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
--------------------------------------------------------------------------------
CPU name:  Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz[…]
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
--------------------------------------------------------------------------------
Group 1: L2
+-----------------------+---------+--------------+--------------+--------------+--------------+
|         Event         | Counter | HWThread 36  | HWThread 37  | HWThread 38  | HWThread 39  |
+-----------------------+---------+--------------+--------------+--------------+--------------+
|   INSTR_RETIRED_ANY   |  FIXC0  |   1409713380 |   1393263859 |   1394342491 |   1388917034 |
| CPU_CLK_UNHALTED_CORE |  FIXC1  |   2095261718 |   2088036330 |   2075539220 |   2058287996 |
|  CPU_CLK_UNHALTED_REF |  FIXC2  |   2103679392 |   2121235200 |   2100479808 |   2075658144 |
|     TOPDOWN_SLOTS     |  FIXC3  |  10476308590 |  10440181650 |  10377696100 |  10291439980 |
|    L1D_REPLACEMENT    |   PMC0  |    142720376 |    142481840 |    142482162 |    142434419 |
|     L2_TRANS_L1D_WB   |   PMC1  |     54986306 |     54864382 |     54868339 |     54815549 |
|  ICACHE_64B_IFTAG_MISS |  PMC2  |       381869 |         2094 |         7399 |         7718 |
+-----------------------+---------+--------------+--------------+--------------+--------------+
[… statistics output omitted …]
+----------------------------------+--------------+--------------+--------------+--------------+
|              Metric              | HWThread 36  | HWThread 37  | HWThread 38  | HWThread 39  |
+----------------------------------+--------------+--------------+--------------+--------------+
|        Runtime (RDTSC) [s]       |       1.0092 |       1.0092 |       1.0092 |       1.0092 |
|       Runtime unhalted [s]       |       0.8751 |       0.8721 |       0.8669 |       0.8597 |
|            Clock [MHz]           |    2384.7406 |    2356.8484 |    2365.8917 |    2374.2844 |
|                CPI               |       1.4863 |       1.4987 |       1.4885 |       1.4819 |
|   L2D load bandwidth [MBytes/s]  |    9050.5857 |    9035.4589 |    9035.4794 |    9032.4518 |
|   L2D load data volume [GBytes]  |       9.1341 |       9.1188 |       9.1189 |       9.1158 |
|  L2D evict bandwidth [MBytes/s]  |    3486.9462 |    3479.2144 |    3479.4653 |    3476.1177 |
|  L2D evict data volume [GBytes]  |       3.5191 |       3.5113 |       3.5116 |       3.5082 |
|     L2 bandwidth [MBytes/s]      |   12561.7480 |   12514.8061 |   12515.4139 |   12509.0589 |
|      L2 data volume [GBytes]     |      12.6777 |      12.6303 |      12.6309 |      12.6245 |
+----------------------------------+--------------+--------------+--------------+--------------+
```

Always measured for Intel CPUs

Configured metrics (this group)

Derived metrics

`likwid-perfctr` stethoscope mode

- likwid-perfctr counts events on cores; it has no notion of what kind of code is running (if any)

  This allows you to "listen" to what is currently happening, without any overhead:

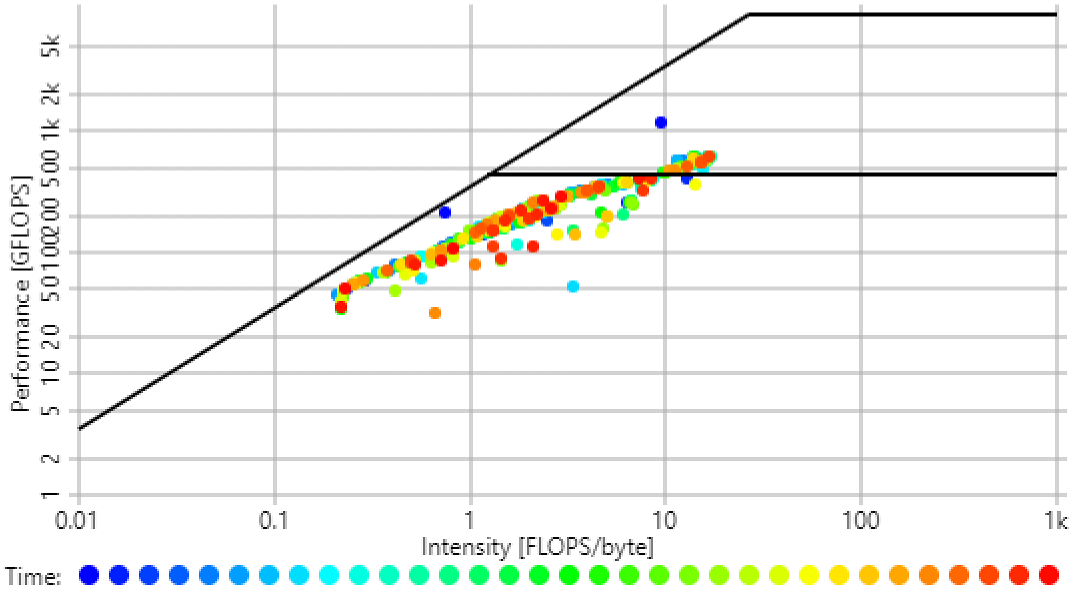
  ```
  $ likwid-perfctr -c N:0-11 -g FLOPS_DP  -S 10s
  ```

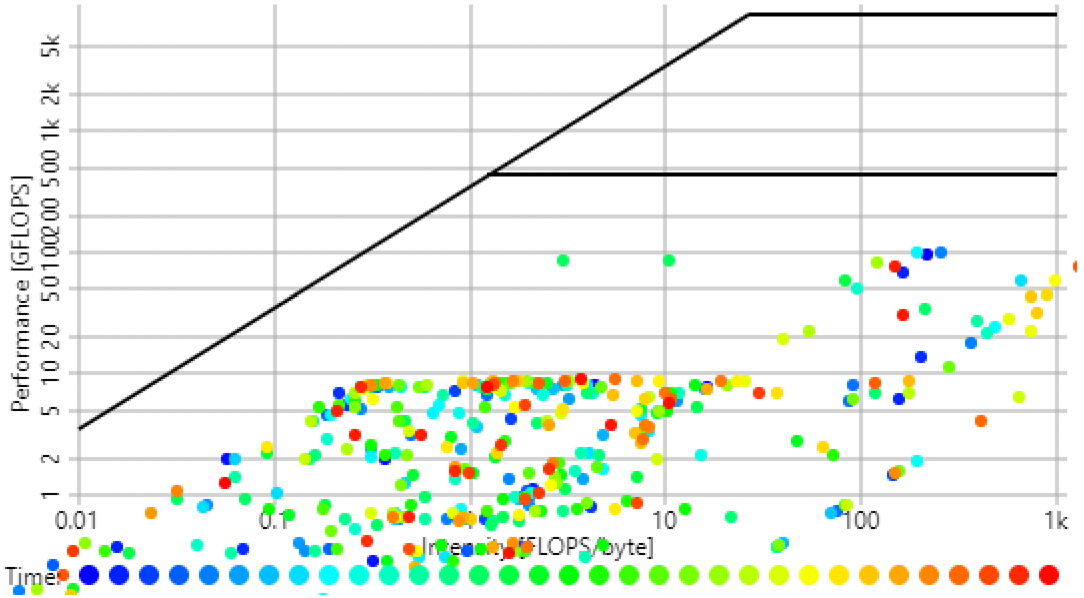
- Can be used as cluster/server monitoring tool

- Frequent use: monitor a long-running parallel application from outside

Cluster monitoring with likwid-perfctr

Two jobs on the NHR@FAU “Fritz” cluster





https://github.com/ClusterCockpit

`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

`likwid-perfctr` with MarkerAPI: OpenMP code (Fortran)

```
program p
  use likwid
  call likwid_markerInit
  !$omp parallel
    call likwid_markerRegisterRegion("MatrixAssembly")
  !$omp end parallel
  ...
  !$omp parallel
    call likwid_markerStartRegion("MatrixAssembly")
    !$omp do
    do i=1,N
     ! Loop
    enddo
    !$omp end do
    call likwid_markerStopRegion("MatrixAssembly")
  !$omp end parallel
  ...
  call likwid_markerClose
end program p
```

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/TutorialMarkerF90

likwid-perfctr with MarkerAPI: source code transformations

```
#pragma omp parallel for
    <loop>
```

```
#pragma omp parallel
{
    LIKWID_MARKER_START(“Compute”);
    #pragma omp for
    <loop>
    LIKWID_MARKER_STOP(“Compute”);
}
```

```
some_parallel_f()
```

```
#pragma omp parallel
{
    LIKWID_MARKER_START(“foo”);
}
some_parallel_f()
#pragma omp parallel
{
    LIKWID_MARKER_STOP(“foo”);
}
```

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
```

Activate markers

MPI:

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

→ 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:

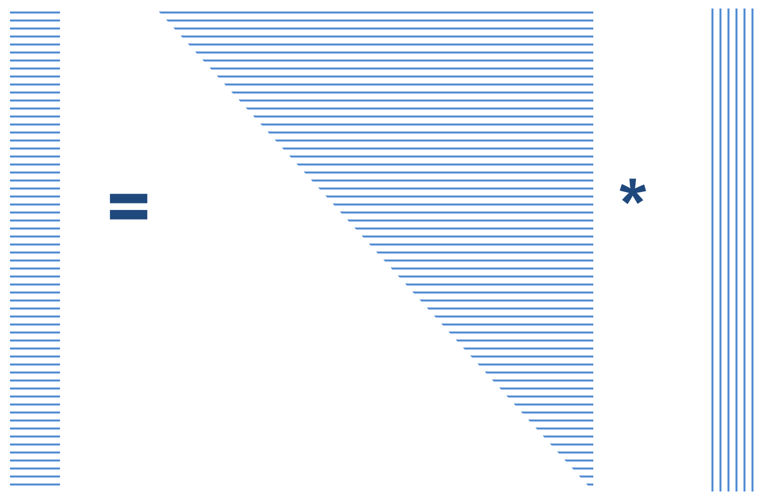
- Clock frequency (GHz)
- Power (W)

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

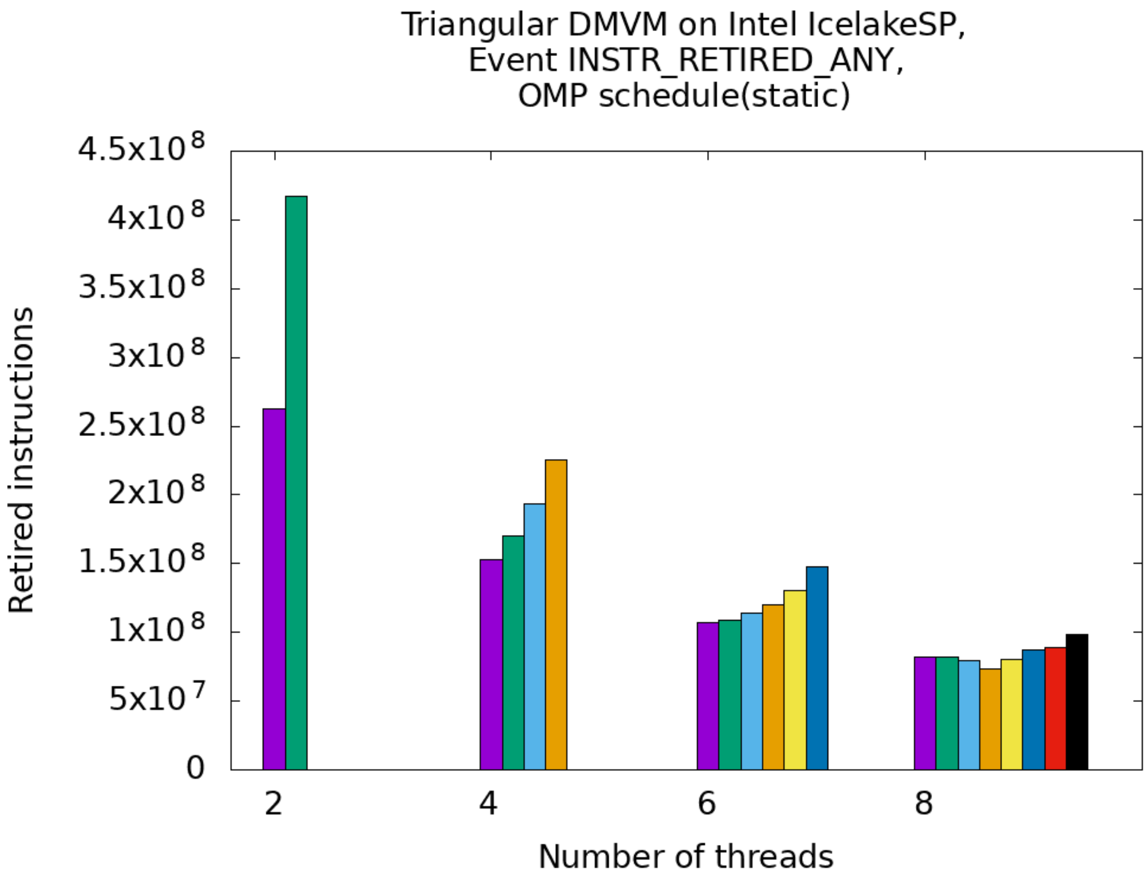
```
$ likwid-perfctr -C 0,1,2 -g L2 -m ./a.out
--------------------------------------------------------------------------------
CPU type:  Intel Icelake SP processor
CPU clock: 2.39 GHz
--------------------------------------------------------------------------------
<<<< PROGRAM OUTPUT >>>>
Region Compute, 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 |
| ICACHE_64B_IFTAG_MISS |   PMC2  |       9698 |      11399 |      11571 |
+-----------------------+---------+------------+------------+------------+
```

???

Example: triangular matrix-vector multiplication

Retired instructions are misleading!

Waiting in implicit OpenMP barrier executes many instructions



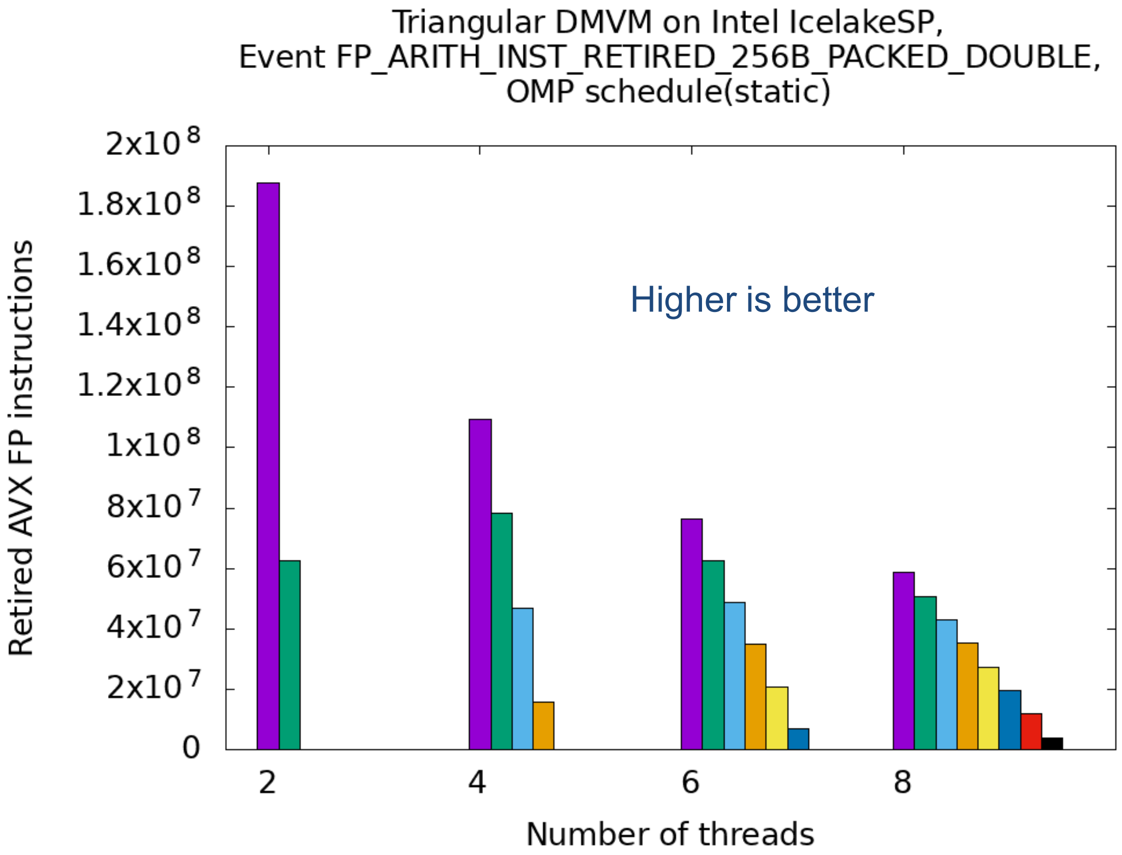
We need to measure actual work (or use a tool that can separate user from runtime lib instructions)

Example: triangular matrix-vector multiplication

Floating-point instructions reliable ↔ useful work metric

Caveats:

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

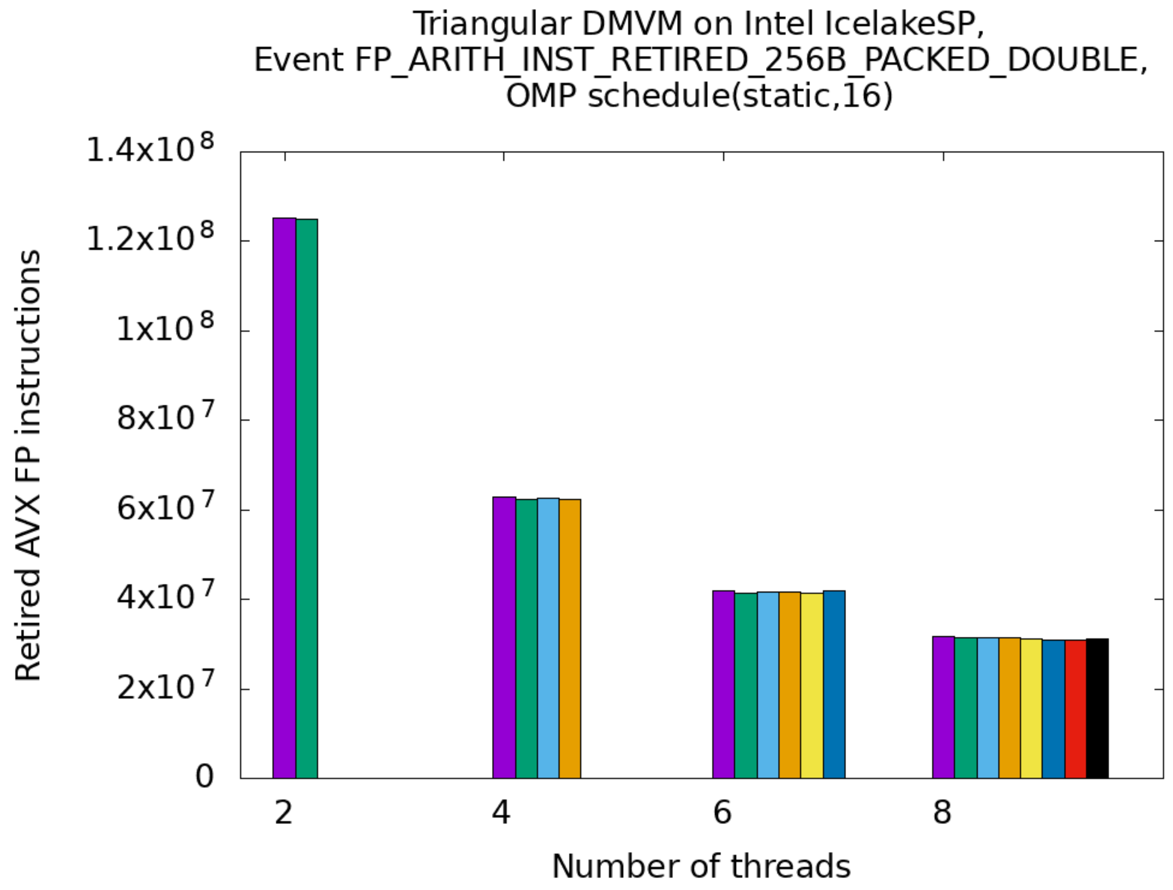


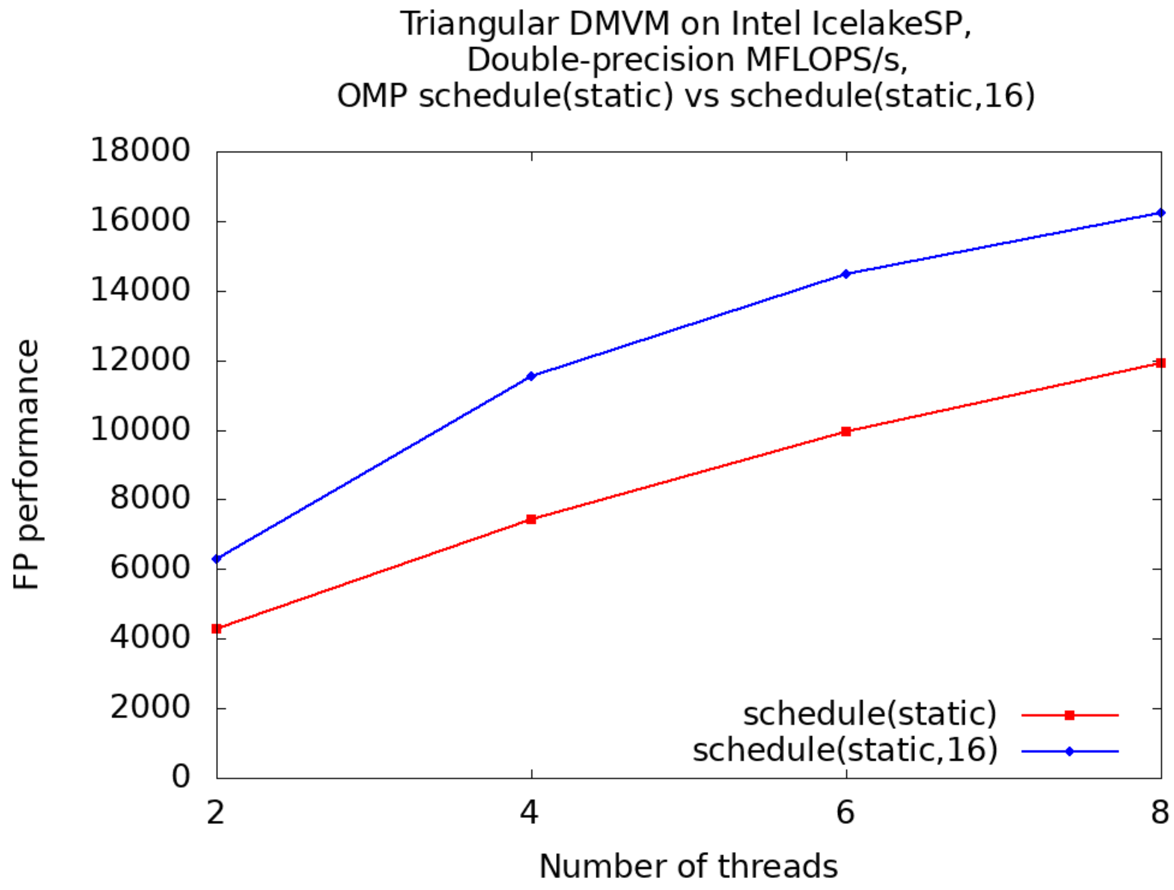
Example: triangular matrix-vector multiplication

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

No imbalance anymore!

Is it also faster?





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
  - Some tools can distinguish runtime instructions from user code instructions

- Another very useful application of PM: validating performance models!
  - Roofline is data centric → measure data volume through memory hierarchy