

# **Performance analysis with hardware metrics**

**Likwid-perfctr**

**Best practices**

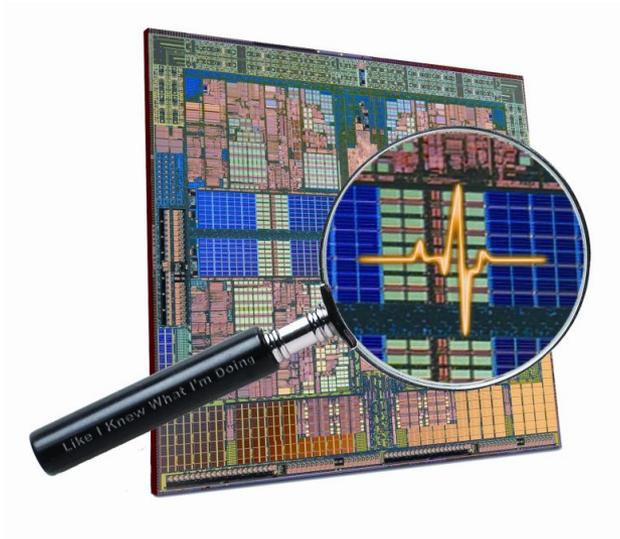
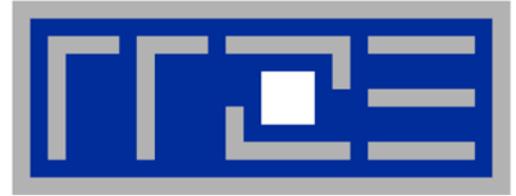
**Energy consumption**



- ... are ubiquitous as a starting point for performance analysis (including automatic analysis)
- ... are supported by many tools
- ... are often reduced to cache misses (what could be worse than cache misses?)

## Reality:

- Modern parallel computing is **plagued by bottlenecks**
  - There are typical **performance patterns** that cover a large part of possible performance behaviors
    - HPM signatures
    - Scaling behavior
    - Other sources of information
- } **“Performance pattern”**



## Using hardware performance metrics

`likwid-perfctr`



- 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** (similar to “perfex” on IRIX, “hpmcount” on AIX, “lipfpm” on Linux/Altix)
  - 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 of cores**  
DATA: Load to store ratio  
**FLOPS\_DP: Double Precision MFlops/s**  
**FLOPS\_SP: Single Precision MFlops/s**  
FLOPS\_X87: X87 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



```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
```

```
-----
CPU name:      Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz [...]
```

```
-----
<<<< PROGRAM OUTPUT >>>>
```

```
Group 1: L2
```

Event	Counter	Core 14	Core 15	Core 16	Core 17
INSTR_RETIRED_ANY	FIXC0	1298031144	1965945005	1854182290	1862521357
CPU_CLK_UNHALTED_CORE	FIXC1	2353698512	2894134935	2894645261	2895023739
CPU_CLK_UNHALTED_REF	FIXC2	2057044629	2534405765	2535218217	2535560434
L1D_REPLACEMENT	PMC0	212900444	200544877	200389272	200387671
L2_TRANS_L1D_WB	PMC1	112464863	99931184	99982371	99976697
ICACHE_MISSES	PMC2	21265	26233	12646	12363

Always measured

Configured metrics (this group)

```
[... statistics output omitted ...]
```

Metric	Core 14	Core 15	Core 16	Core 17
Runtime (RDTSC) [s]	1.1314	1.1314	1.1314	1.1314
Runtime unhalted [s]	1.0234	1.2583	1.2586	1.2587
Clock [MHz]	2631.6699	2626.4367	2626.0579	2626.0468
CPI	1.8133	1.4721	1.5611	1.5544
L2D load bandwidth [MBytes/s]	12042.7388	11343.8446	11335.0428	11334.9523
L2D load data volume [GBytes]	13.6256	12.8349	12.8249	12.8248
L2D evict bandwidth [MBytes/s]	6361.5883	5652.6192	5655.5146	5655.1937
L2D evict data volume [GBytes]	7.1978	6.3956	6.3989	6.3985
L2 bandwidth [MBytes/s]	18405.5299	16997.9477	16991.2728	16990.8453
L2 data volume [GBytes]	20.8247	19.2321	19.2246	19.2241

Derived metrics



### Things to look at (in roughly this order)

- **Load balance** (flops, instructions, BW)
- **In-socket memory BW saturation**
- **Shared cache BW saturation**
- **Flop/s, loads and stores per flop metrics**
- **SIMD** vectorization
- **CPI** metric
- **Clock speed**
- **# of instructions**, branches, mispredicted branches

### Caveats

- **Load imbalance** may not show in CPI or # of instructions
  - **Spin loops** in OpenMP barriers/MPI blocking calls
  - Looking at “top” or the Windows Task Manager does not tell you anything useful
- **In-socket performance saturation** may have various reasons
- **Cache miss metrics are overrated**
  - If I really know my code, I can often *calculate* the misses
  - Runtime and resource utilization is much more important



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

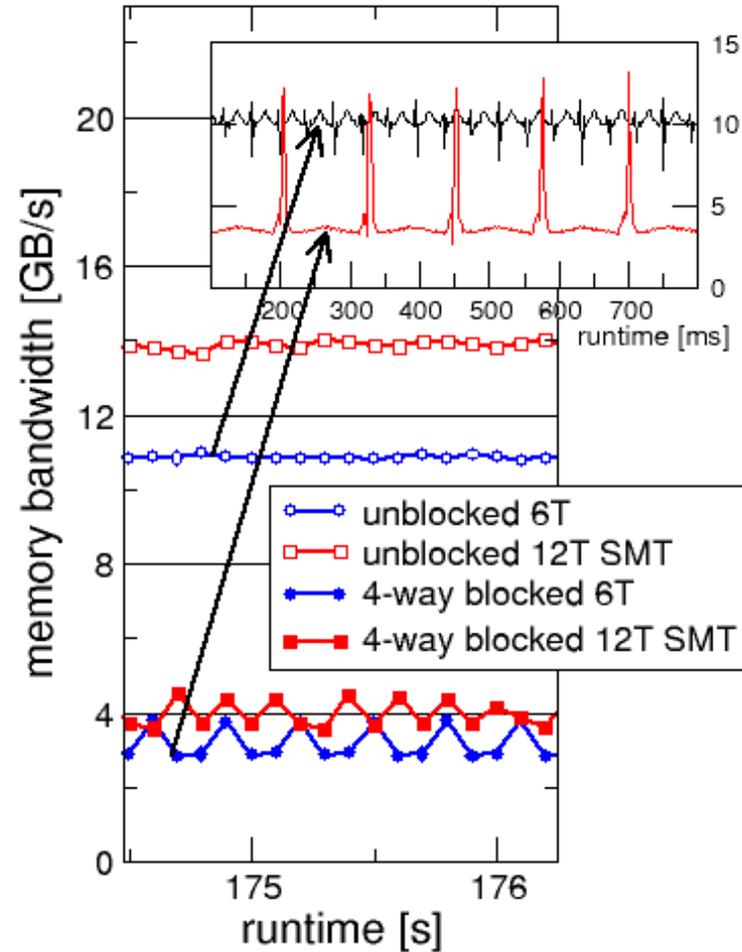
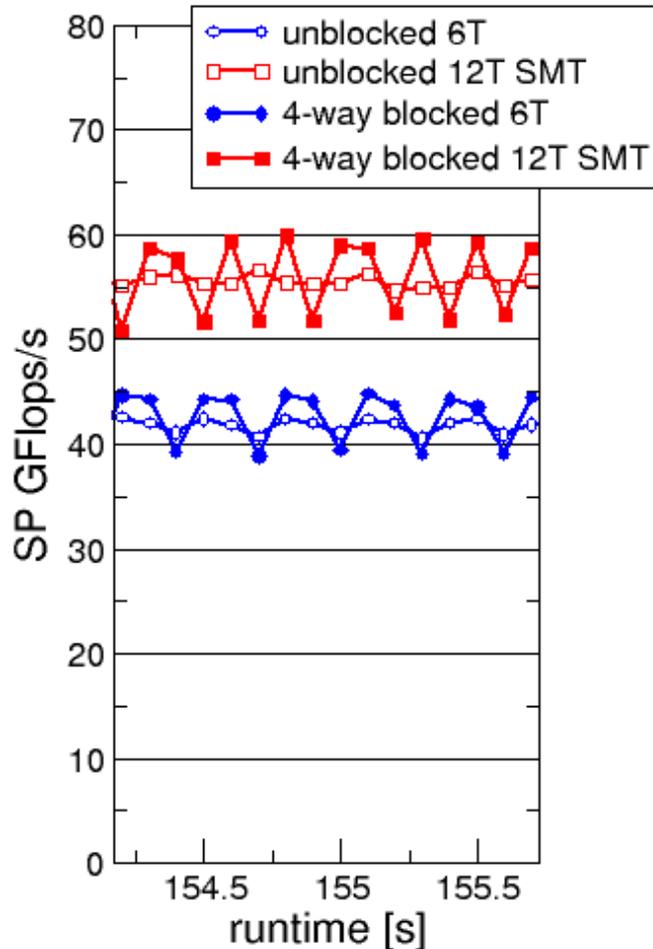
This allows you to “listen” to what is currently happening, without any overhead:

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

- It can be used as cluster/server monitoring tool
- A frequent use is to measure a certain part of a long running parallel application from outside

likwid-perfctr supports time-resolved measurements of full node:

```
$ likwid-perfctr -c N:0-11 -g MEM -t 50ms > out.txt
```





- The marker API can restrict measurements to code regions
- The API only turns counters on/off. The configuration of the counters is still done by likwid-perfctr
- Multiple named regions support, accumulation over multiple calls
- Inclusive and overlapping regions allowed

```
#include <likwid.h>
. . .
LIKWID_MARKER_INIT;           // must be called from serial region
#pragma omp parallel
{
    LIKWID_MARKER_THREADINIT; // only reqd. if measuring multiple threads
}
. . .
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
```

- Activate macros with `-DLIKWID_PERFMON`
- Run `likwid-perfctr` with `-m` switch to enable marking
- See <https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90> for Fortran example



## Compile:

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

## Link:

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

## Run:

```
likwid-perfctr -C <MASK> -g <GROUP> -m ./a.out
```

- One separate block of output for every marked region
- Caveat: marker API can cause overhead; do not call too frequently!



SHORT PSTI

### EVENTSET

```
FIXC0 INSTR_RETIRED_ANY
FIXC1 CPU_CLK_UNHALTED_CORE
FIXC2 CPU_CLK_UNHALTED_REF
PMC0 FP_COMP_OPS_EXE_SSE_FP_PACKED
PMC1 FP_COMP_OPS_EXE_SSE_FP_SCALAR
PMC2 FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION
PMC3 FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION
UPMC0 UNC_QMC_NORMAL_READS_ANY
UPMC1 UNC_QMC_WRITES_FULL_ANY
UPMC2 UNC_QHL_REQUESTS_REMOTE_READS
UPMC3 UNC_QHL_REQUESTS_LOCAL_READS
```

### METRICS

```
Runtime [s] FIXC1*inverseClock
CPI FIXC1/FIXC0
Clock [MHz] 1.E-06*(FIXC1/FIXC2)/inverseClock
DP MFlops/s (DP assumed) 1.0E-06*(PMC0*2.0+PMC1)/time
Packed MUOPS/s 1.0E-06*PMC0/time
Scalar MUOPS/s 1.0E-06*PMC1/time
SP MUOPS/s 1.0E-06*PMC2/time
DP MUOPS/s 1.0E-06*PMC3/time
Memory bandwidth [MBytes/s] 1.0E-06*(UPMC0+UPMC1)*64/time;
Remote Read BW [MBytes/s] 1.0E-06*(UPMC2)*64/time;
```

### LONG

Formula:

```
DP MFlops/s = (FP_COMP_OPS_EXE_SSE_FP_PACKED*2 + FP_COMP_OPS_EXE_SSE_FP_SCALAR) / runtime.
```

- Groups are architecture-specific
- They are defined in simple text files
- Code is generated on recompilation of likwid
- `likwid-perfctr -a` prints a list of groups
- An extended documentation (help text) is available for every group

# Example: triangular dense MVM



## Multiplication of upper triangular matrix with vector

```
#define N 10000 // large enough to use memory
#define ROUNDS 10

[...]

// 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;}
    }
}
```



## LIKWID marker API

```
#include <likwid.h>
[...]
LIKWID_MARKER_INIT;
#pragma omp parallel
{
    LIKWID_MARKER_THREADINIT;
}
#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;
```

Expectation: less work for high thread IDs

# Example: triangular dense MVM



```
$ likwid-perfctr -C 0,1,2 -g L2 -m ./a.out
```

```
-----  
CPU type:      Intel Core SandyBridge EN/EP processor  
CPU clock:    3.09 GHz  
-----
```

```
=====  
Group 1: Region Compute  
=====
```

Region Info	Core 0	Core 1	Core 2
RDTSC Runtime [s]	0.161382	0.161365	0.161365
call count	10	10	10

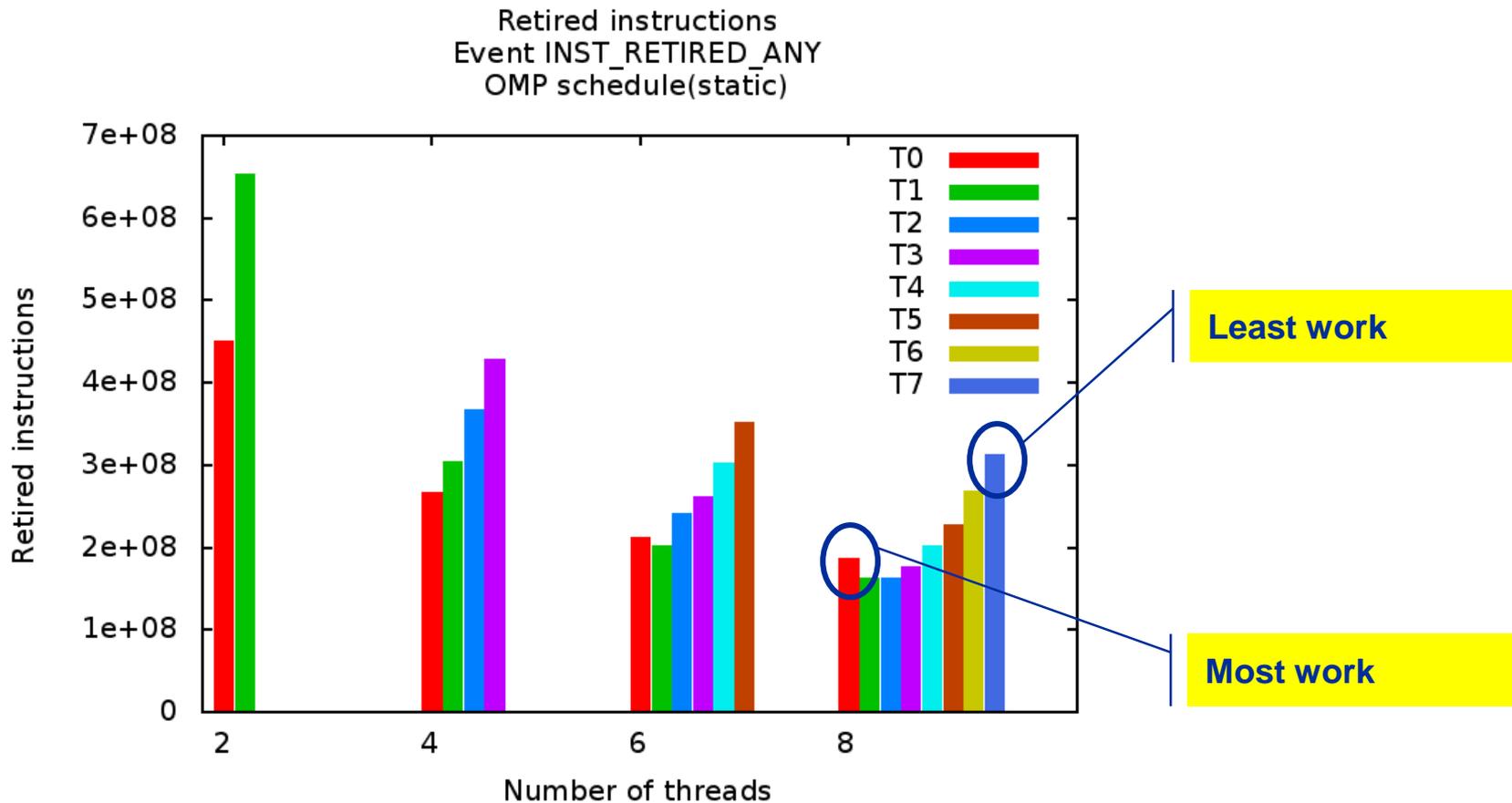
Event	Counter	Core 0	Core 1	Core 2
INSTR_RETIRED_ANY	FIXC0	2.626800e+08	3.187585e+08	3.780255e+08
CPU_CLK_UNHALTED_CORE	FIXC1	4.972802e+08	4.961411e+08	4.933711e+08
CPU_CLK_UNHALTED_REF	FIXC2	4.972801e+08	4.961404e+08	4.933714e+08
L1D_REPLACEMENT	PMC0	5.490278e+07	3.927353e+07	2.364295e+07
L1D_M_EVICT	PMC1	2.920200e+04	2.876600e+04	2.861000e+04
ICACHE_MISSES	PMC2	4.649000e+03	4.984000e+03	5.321000e+03

????

# Example: triangular dense MVM



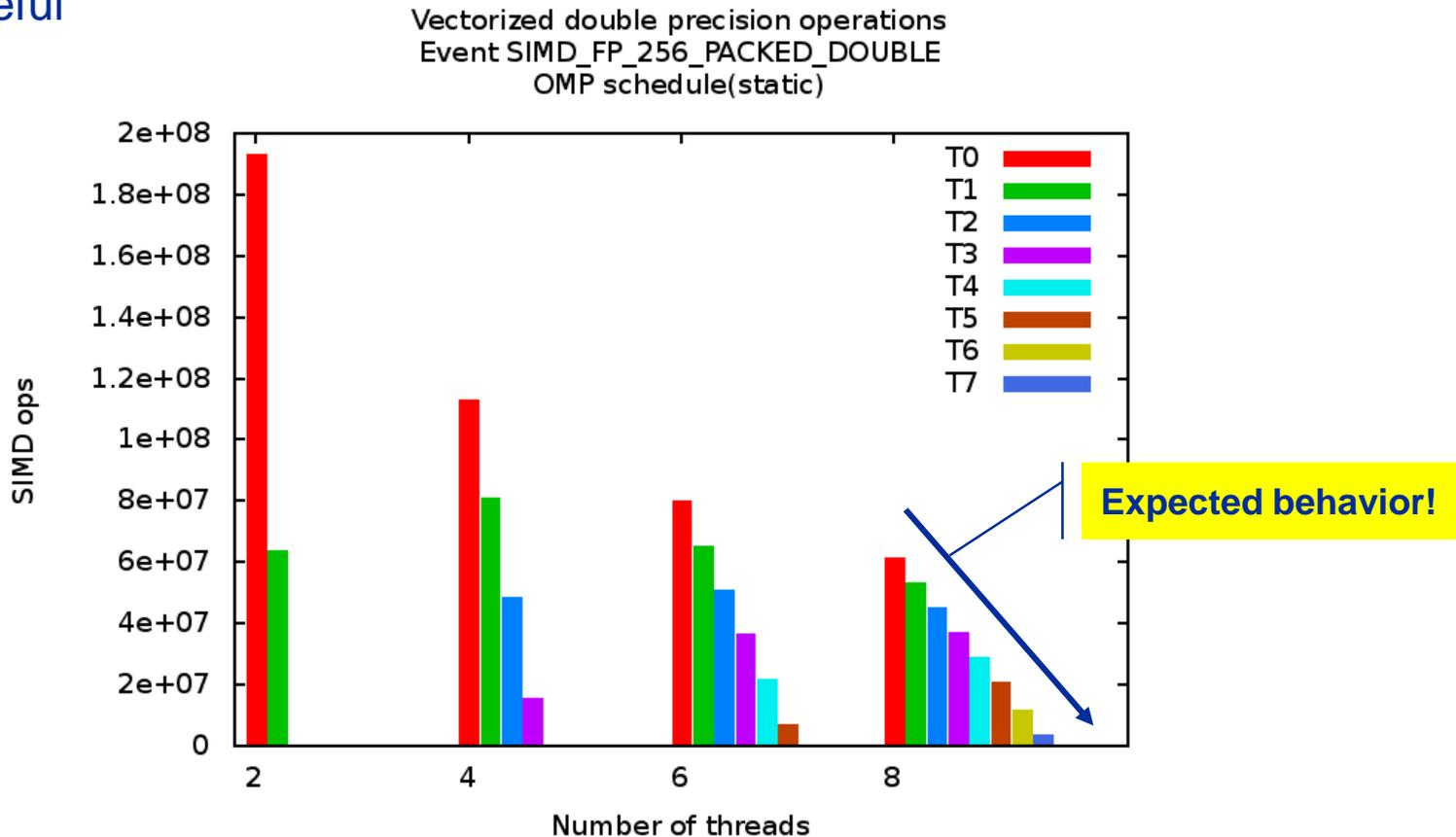
- Retired instructions are misleading
- Waiting in implicit OpenMP barrier issues many instructions



# Example: triangular dense MVM



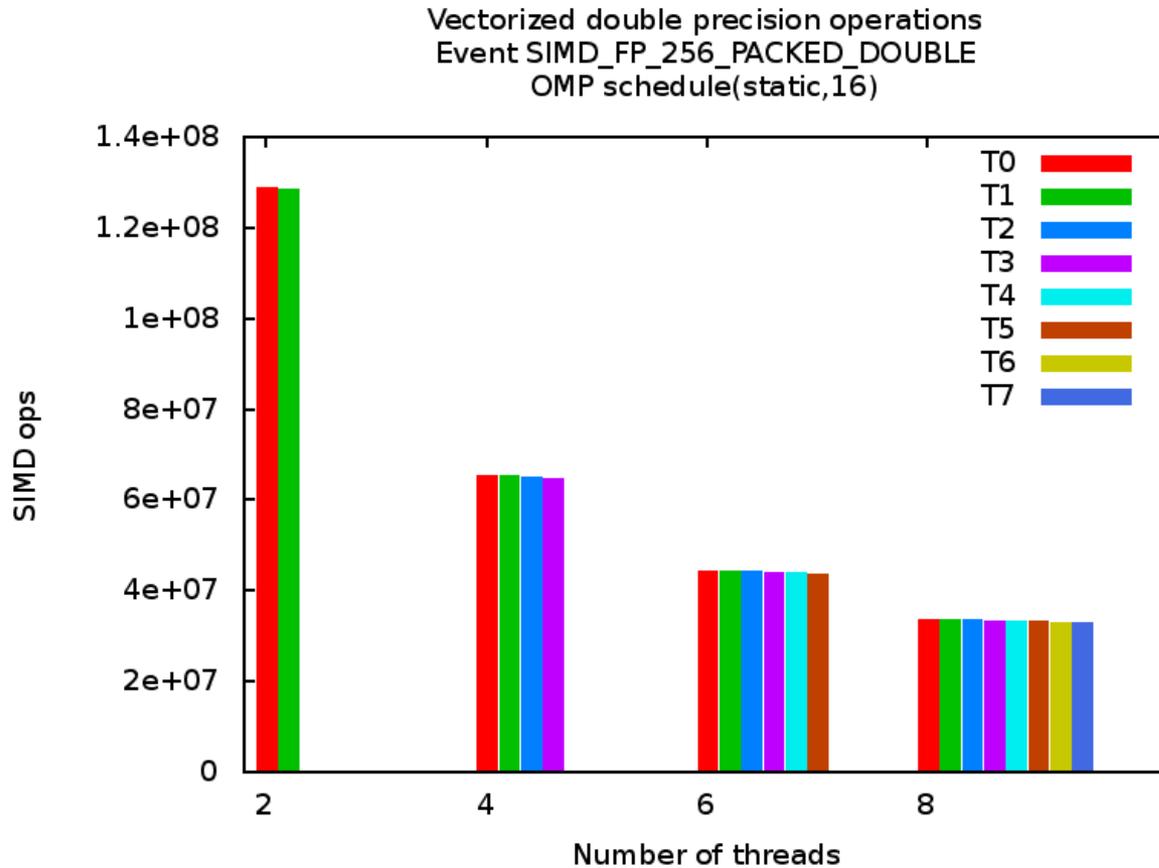
- Floating point instructions are reliable ↔ useful work metric!
- But floating point instr. counters suffer from miscounting since SandyBridge (corrected in Broadwell)
  - Still useful



# Example: triangular dense MVM



- Changing **static** schedule to **static,16** → load balanced



# Energy with likwid-perfctr in wrapper mode



```
$ likwid-perfctr -g ENERGY -C S0:0-1 ./a.out
```

[...]

Metric	Core 0	Core 1
Runtime (RDTSC) [s]	2.362547e+01	2.362547e+01
Runtime unhaltd [s]	3.347379e+01	3.348318e+01
Clock [MHz]	3.284280e+03	3.284278e+03
CPI	6.105052e-01	6.116845e-01
Temperature [C]	33	33
Energy [J]	1.423615e+03	0
Power [W]	6.025761e+01	0
Energy PP0 [J]	0	0
Power PP0 [W]	0	0
Energy DRAM [J]	89.424216	0
Power DRAM [W]	3.785076e+00	0

Package (socket)

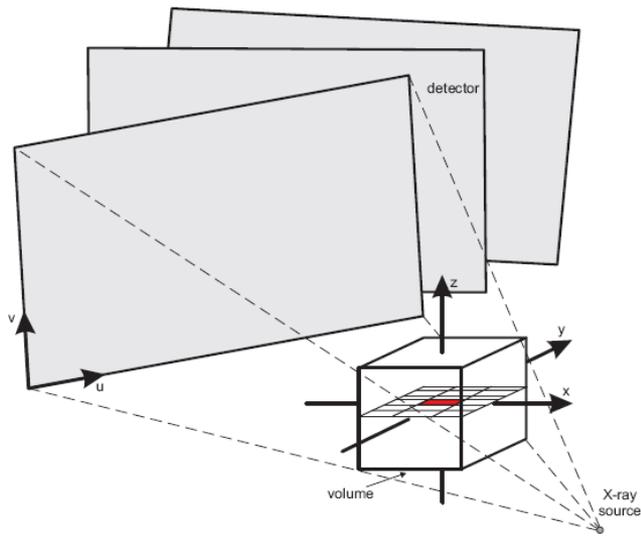
Cores only (if available)

DRAM (DIMMs)

[...]

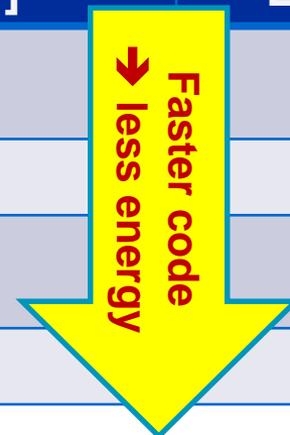
# Example:

A medical image reconstruction code on Sandy Bridge



## Sandy Bridge EP (8 cores, 2.7 GHz base freq.)

Test case	Runtime [s]	Power [W]	Energy [J]
8 cores, plain C	<b>90.43</b>	90	8110
8 cores, SSE	29.63	93	2750
8 cores (SMT), SSE	22.61	102	2300
8 cores (SMT), AVX	<b>18.42</b>	111	2040





- **Useful only if you know what you are looking for**
  - PM bears 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 incur 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

# Monitoring “live” jobs on a cluster in the Roofline diagram



Ganglia Data / Roofline (04. Feb. 2016 - 14:12:24)

Click and drag to zoom in. Hold down shift key to x-pan.

