Performance analysis with hardware metrics

Likwid-perfctr
Best practices
Hardware performance metrics

- ... 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
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 (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
## Example usage with preconfigured metric group

$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -g FLOPS_DP ./stream.exe

---

**CPU type:** Intel Core Lynnfield processor  
**CPU clock:** 2.93 GHz

---

### Measuring group FLOPS_DP

---

#### YOUR PROGRAM OUTPUT

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INSTR_RETIRED_ANY</strong></td>
<td>1.97463e+08</td>
<td>2.31001e+08</td>
<td>2.30963e+08</td>
<td>2.31885e+08</td>
</tr>
<tr>
<td><strong>CPU_CLK_UNHALTED_CORE</strong></td>
<td>9.56999e+08</td>
<td>9.58401e+08</td>
<td>9.58637e+08</td>
<td>9.57338e+08</td>
</tr>
<tr>
<td><strong>FP_COMP_OPS_EXE_SSE_FP_PACKED</strong></td>
<td>4.00294e+07</td>
<td>3.08927e+07</td>
<td>3.08866e+07</td>
<td>3.08904e+07</td>
</tr>
<tr>
<td><strong>FP_COMP_OPS_EXE_SSE_FP_SCALAR</strong></td>
<td>882</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION</strong></td>
<td>4.00303e+07</td>
<td>3.08927e+07</td>
<td>3.08866e+07</td>
<td>3.08904e+07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [s]</td>
<td>0.326242</td>
<td>0.32672</td>
<td>0.326801</td>
<td>0.326358</td>
</tr>
<tr>
<td>CPI</td>
<td>4.84647</td>
<td>4.14891</td>
<td>4.15061</td>
<td>4.12849</td>
</tr>
<tr>
<td>DP MFlops/s (DP assumed)</td>
<td>245.399</td>
<td>189.108</td>
<td>189.024</td>
<td>189.304</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>122.698</td>
<td>94.554</td>
<td>94.5121</td>
<td>94.6519</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>0.00270351</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SP MUOPS/s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DP MUOPS/s</td>
<td>122.701</td>
<td>94.554</td>
<td>94.5121</td>
<td>94.6519</td>
</tr>
</tbody>
</table>

---

**Always measured**  
**Derived metrics**  
**Configured metrics** (this group)
likwid-perfctr
Best practices for runtime counter analysis

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
- # 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
Instructions retired / CPI may not be a good indication of useful workload – at least for numerical / FP intensive codes...

Floating Point Operations Executed is often a better indicator

Waiting / “Spinning” in barrier generates a high instruction count

```
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, I
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO

!(c) RRZE 2014
Practical Performance Analysis
```
likwid-perfctr
... and load-balanced codes

env OMP_NUM_THREADS=6 likwid-perfctr -t intel -C S0:0-5 -g FLOPS_DP ./a.out

Higher CPI but better performance

%!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, N
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
%!$OMP END PARALLEL DO

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

This enables to listen on what currently happens without any overhead:

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

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**

*Timeline mode*

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

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

![Timeline diagram](chart.png)
Marker API

- A marker API is available to 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

```c
#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`
likwid-perfctr

Group files

- Groups are architecture-specific
- They are defined in simple text files
- Code is generated on recompile of likwid
- likwid-perfctr -a outputs list of groups
- For every group an extensive documentation is available

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.
DEMO
Measuring energy consumption with LIKWID
Measuring energy consumption

*likwid-powermeter* and *likwid-perfctr -*g ENERGY*

- Implements Intel RAPL interface (Sandy Bridge)
- RAPL = “Running average power limit”

---

**CPU name:** Intel Core SandyBridge processor  
**CPU clock:** 3.49 GHz

---

**Base clock:** 3500.00 MHz  
**Minimal clock:** 1600.00 MHz

**Turbo Boost Steps:**
- C1 3900.00 MHz
- C2 3800.00 MHz
- C3 3700.00 MHz
- C4 3600.00 MHz

---

**Thermal Spec Power:** 95 Watts  
**Minimum Power:** 20 Watts  
**Maximum Power:** 95 Watts  
**Maximum Time Window:** 0.15625 micro sec

---

(c) RRZE 2014  
Practical Performance Analysis
Example:
A medical image reconstruction code on Sandy Bridge

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8 cores, plain C</td>
<td>90.43</td>
<td>90</td>
<td>8110</td>
</tr>
<tr>
<td>8 cores, SSE</td>
<td>29.63</td>
<td>93</td>
<td>2750</td>
</tr>
<tr>
<td>8 cores (SMT), SSE</td>
<td>22.61</td>
<td>102</td>
<td>2300</td>
</tr>
<tr>
<td>8 cores (SMT), AVX</td>
<td>18.42</td>
<td>111</td>
<td>2040</td>
</tr>
</tbody>
</table>

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

Faster code \(\rightarrow\) less energy