



PERFORMANCE COUNTERS AND TOOLS

OPENPOWER TUTORIAL, SC19, DENVER

18 November 2019 | Andreas Herten | Forschungszentrum Jülich, Jülich Supercomputing Centre

Outline

Goals of this session

- Get to know Performance Counters
 - Measure counters on POWER9
- Hands-on
- *Additional material in [appendix](#)*

Motivation

Performance Counters

Introduction

General Description

Counters on POWER9

Measuring Counters

perf

PAPI

GPUs

Conclusion

Knuth

*[...] premature optimization is the root of all evil.
Yet we should not pass up our [optimization] opportunities [...]*

– Donald Knuth

Optimization Measurement

Making educated decisions

- Only optimize code **after** measuring its performance

Measure! Don't trust your gut!

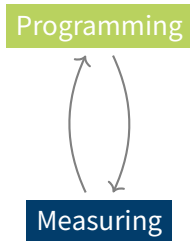
- Objectives

- Run time
- Cycles
- Operations per cycle (FLOP/s)
- Usage of architecture features (\$, (S)MT, SIMD, ...)

- Correlate measurements with code

→ Hot spots/performance limiters

- Iterative process



Measurement

Native
Derived
Software

Two options for insight

Coarse Timestamps to time program / parts of program

»The `printf()` method«

- Only good for first glimpse
- No insight to inner workings

Detailed **Performance counters** to study usage of hardware architecture

- Instructions → IPC, CPI
- Cycles →
- Floating point operations
- Stalled cycles
- Cache misses, cache hits
- Prefetches
- Flushes
- Branches
- CPU migrations
- ...

Performance Counters

Performance Monitoring Unit

Right next to the core

- Part of processor periphery, but dedicated registers
- History
 - First occurrence: Intel Pentium, reverse-engineered 1994 (RDPMC) [2]
 - Originally for chip developers
 - Later embraced for software developers and *tuners*
- Operation: Certain events counted at logic level, then aggregated to registers

Pros

- Low overhead
- Very specific requests possible; detailed information
- Information about CPU core, *nest*, cache, memory

Cons

- Processor-specific
- Hard to measure
- Limited amount of counter registers
- Compressed information content

Working with Performance Counters

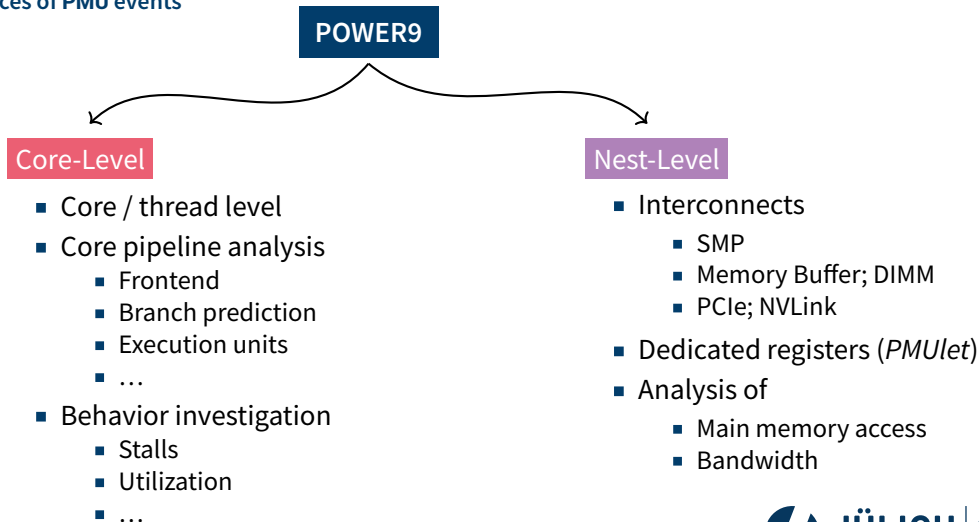
Some caveats

- Mind the clock rates!
 - Modern processors have dynamic clock rates (CPUs, GPUs)
 - Might skew results
 - Some counters might not run at nominal clock rate
- Limited counter registers
POWER9: 6 registers for hardware counters (PMC1 - PMC6) [3]
- Cores, Threads (OpenMP)
 - Absolutely possible
 - Complicates things slightly
 - Pinning necessary
 - `OMP_PROC_BIND`, `OMP_PLACES`; `PAPI_thread_init()`
- Nodes (MPI): Counters independent of MPI, but aggregation tool useful (Score-P, ...)

Performance Counters on POWER9

POWER9 Compartments

Sources of PMU events



POWER9 Performance Counters

Instructions, Stalls

PM_LD_MISS_L1 Load missed L1 cache
Store: PM_ST_MISS_L1; Local L4 Hit: PM_DATA_FROM_LL4

PM_INST_CMPL Instructions completed
Also: PM_RUN_INST_CMPL

PM_VECTOR_FLOP_CMPL Vector FP instruction completed
Also: PM_2FLOP_CMPL

PM_RUN_CYC Total cycles run

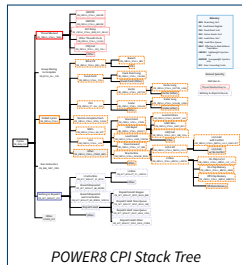
Processor cycles gated by the run latch

PM_CMPLU_STALL Completion stall
Cycles in which a thread did not complete any groups, but there were entries

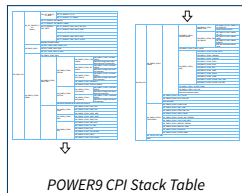
PM_CMPLU_STALL_THRD Completion stall due to thread conflict
Completion stalled because the thread was blocked

PM_CMPLU_STALL_BRU Stall due to BRU
BRU: Branch Unit

PM_CMPLU_STALL_LSU Completion stall by LSU instruction
LSU: Load/Store Unit



POWER8 CPI Stack Tree



POWER9 CPI Stack Table

POWER9 Performance Counters

Instructions, Stalls

PM_LD_MISS_L1 Load missed L1 cache

PM_INST_CMPL

PM_VECTOR_FLOP_CMPL

PM_RUN_CYC

PM_CMPLU_STALL

PM_CMPLU_STALL_THRU

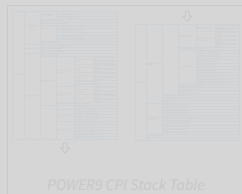
PM_CMPLU_STALL_BR

PM_CMPLU_STALL_LSU Completion stall by LSU instruction

LSU: Load/Store Unit

Number of counters for POWER9:
959

*See appendix for more on counters
(CPI stack; resources)*



Measuring Counters

Overview

`perf` Linux tool (based on `perf_events` Linux interface)

PAPI C/C++ API

Score-P Measurement environment (*appendix*)

Likwid Set of command line utilities for detailed analysis

`perf_event_open()` Linux system call from `linux/perf_event.h`

... Many more solutions, usually relying on `perf`

perf

Linux' own performance tool

- Part of Linux kernel since 2009 (v. 2.6.31)
- Example usage: `perf stat ./app`

```
$ perf stat ./poisson2d
Performance counter stats for './poisson2d':

   65703.208586      task-clock (msec)    #    1.000 CPUs utilized
          355      context-switches        #    0.005 K/sec
           0      cpu-migrations            #    0.000 K/sec
       10,847      page-faults              #    0.165 K/sec
  228,425,964,399    cycles                   #    3.477 GHz           (66.67%)
   299,409,593      stalled-cycles-frontend  #    0.13% frontend cycles idle (50.01%)
  147,289,312,280    stalled-cycles-backend  #   64.48% backend cycles idle (50.01%)
  323,403,983,324    instructions              #    1.42   insn per cycle
                                     #    0.46   stalled cycles per insn (66.68%)
   12,665,027,391    branches                  # 192.761 M/sec          (50.00%)
     4,256,513      branch-misses            #    0.03% of all branches (50.00%)

65.715156815 seconds time elapsed
```

perf

Linux' own performance tool

- Part of Linux kernel since 2009 (v. 2.6.31)
- Usage: `perf stat ./app`
- Raw counter example: `perf stat -e PM_BR_CMPL ./app`

```
$ perf stat -e PM_BR_CMPL ./poisson2d
```

```
Performance counter stats for './poisson2d':
```

```
1638043350      PM_BR_CMPL:u                      (3.44%)
```

```
65.761947405 seconds time elapsed
```


perf

Linux' own performance tool

- Part of Linux kernel since 2009 (v. 2.6.31)
- Usage: `perf stat ./app`
- Raw counter example: `perf stat -e PM_BR_CMPL ./app`
- More in [appendix](#)

PAPI

Measure where it hurts...

- Performance Application Programming Interface
- API for C/C++, Fortran
- Goal: Create common (and easy) interface to performance counters
- Two API layers (*Examples in [appendix!](#)*)
 - High-Level API: Most-commonly needed information capsuled by convenient functions
 - Low-Level API: Access all the counters!
- Command line utilities
 - `papi_avail` List aliased, common counters
 - Use `papi_avail -e EVENT` to get description and options for EVENT
 - `papi_native_avail` List all possible counters, with details
- Extendable by Component PAPI (GPU!)
- **Comparison to perf**: Instrument specific parts of code, with different counters

PAPI

papi_avail

```
$ papi_avail
```

```
Available PAPI preset and user defined events plus hardware information.
```

```
-----  
PAPI version           : 5.7.0.0  
Operating system       : Linux 4.14.0-115.6.1.el7a.ppc64le  
Vendor string and code : IBM (3, 0x3)  
Model string and code  : 8335-GTC (0, 0x0)  
CPU revision           : 2.0000000  
CPU Max MHz            : 3800  
CPU Min MHz            : 2300  
Total cores            : 128  
SMT threads per core   : 4  
Cores per socket       : 16  
Sockets                : 2  
Cores per NUMA region  : 128  
NUMA regions           : 1  
Number Hardware Counters : 5
```

PAPI

papi_avail

```
Max Multiplex Counters    : 384
Fast counter read (rdpmc): no
```

```
-----
=====
PAPI Preset Events
=====
```

Name	Code	Avail	Deriv	Description (Note)
PAPI_L1_DCM	0x80000000	Yes	Yes	Level 1 data cache misses
PAPI_L1_ICM	0x80000001	Yes	No	Level 1 instruction cache misses
PAPI_L2_DCM	0x80000002	Yes	No	Level 2 data cache misses
PAPI_L2_ICM	0x80000003	Yes	No	Level 2 instruction cache misses
PAPI_L3_DCM	0x80000004	Yes	Yes	Level 3 data cache misses
PAPI_L3_ICM	0x80000005	Yes	No	Level 3 instruction cache misses
PAPI_L1_TCM	0x80000006	No	No	Level 1 cache misses
PAPI_L2_TCM	0x80000007	No	No	Level 2 cache misses
PAPI_L3_TCM	0x80000008	No	No	Level 3 cache misses

PAPI

papi_avail

```
$ papi_avail -e PM_DATA_FROM_L3MISS
```

```
Available PAPI preset and user defined events plus hardware information.
```

```
-----  
Event name:          PM_DATA_FROM_L3MISS  
Event Code:          0x40000021  
Number of Register Values: 0  
Description:         |Demand LD - L3 Miss (not L2 hit and not L3 hit).|
```

```
Unit Masks:  
Mask Info:           |:u=0|monitor at user level|  
Mask Info:           |:k=0|monitor at kernel level|  
Mask Info:           |:h=0|monitor at hypervisor level|  
Mask Info:           |:period=0|sampling period|  
Mask Info:           |:freq=0|sampling frequency (Hz)|  
Mask Info:           |:excl=0|exclusive access|  
Mask Info:           |:mg=0|monitor guest execution|
```

PAPI

Notes on usage; Tipps

- Important functions in **High Level API**

`PAPI_num_counters()` Number of available counter registers

`PAPI_flops()` Get real time, processor time, number of floating point operations, and MFLOPs/s

`PAPI_ipc()` Number of instructions and IPC (+rttime/ptime)

`PAPI_epc()` Number of counts of arbitrary event (+rttime/ptime)

- Important functions in **Low Level API**

`PAPI_add_event()` Add aliased event to event set

`PAPI_add_named_event()` Add any event to event set

`PAPI_thread_init()` Initialize thread support in PAPI

- Documentation [online](#) and in man pages (man `papi_add_event`)

- All PAPI calls return status code; check for it! (*Macros in appendix: C++, C*)

- Convert names of performance counters with `libpfm4` (*appendix*)

→ <http://icl.cs.utk.edu/papi/>

GPU Counters

A glimpse ahead

- Counters built right in
- Grouped into *domains* by topic
- NVIDIA differentiates between (*more examples in [appendix](#)*)

Event Countable activity or occurrence on GPU device

Examples: `shared_store`, `generic_load`, `shared_atom`

Metric Characteristic calculated from one or more events

Examples: `executed_ipc`, `flop_count_dp_fma`, `achieved_occupancy`

- Usually: access via `nvprof` / Visual Profiler / Nsight Compute
Exposed via CUPTI for 3rd party

→ **Afternoon session** / [appendix](#)

Conclusions

What we've learned

- Large set of performance counters on POWER9 processors
- Right next to (*inside*) core(s)
- Provide detailed insight for performance analysis on many levels
- Different measurement strategies and tools
 - perf
 - PAPI
 - Score-P
- Also on GPU

Thank you
for your attention!
a.herten@fz-juelich.de

Appendix

Appendix

Knuth on Optimization

POWER9 Performance Counters

perf

PAPI Supplementary

Score-P

GPU Counters

Glossary

References

Appendix

Knuth on Optimization

Knuth on Optimization

The full quote

There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3 %. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified

– Donald Knuth in “Structured Programming with Go to Statements” [4]

Appendix

POWER9 Performance Counters

POWER Performance Counters

- Further information on counters at IBM website
 - *POWER9 Performance Monitor Unit User's Guide* [5]
 - PMU Events for POWER9 in the Linux kernel
 - JSON overview of OpenPOWER PMU events on Github
 - OProfile: [ppc64 POWER8 events](#), [ppc64 POWER9 events](#)
- List available counters on system
 - With PAPI: `papi_native_avail`
 - With `showevtinfo` from [libpfm](#)'s `/examples/` directory

```
./showevtinfo | \
  grep -e "Name" -e "Desc" | sed "s/^.\++: //g" | paste -d'\t' - -
```
- See [next slide](#) for CPI stack visualization
- Most important counters for **OpenMP**: `DMISS_PM_CMPLU_STALL_DMISS_L3MISS`,
`PM_CMPLU_STALL_DMISS_REMOTE`

Extracted from POWER9 Performance Monitor Unit User's Guide [5]

Appendix

perf

perf

Sub-commands

- Sub-commands for perf

- `perf list` List available counters

- `perf stat` Run program; report performance data

- `perf record` Run program; **sample** and save performance data

- `perf report` Analyzed saved performance data (*appendix*)

- `perf top` Like top, live-view of counters



perf

Tipps, Tricks

- Option `--repeat` for statistical measurements

1.239 seconds **time** elapsed (+- 0.16%)

- Restrict counters to certain user-level modes by `-e counter:m`, with `m = u` (user), `= k` (kernel), `= h` (hypervisor)
- `perf` modes: Per-thread (default), per-process (`-p PID`), per-CPU (`-a`)
- Other options

`-d` More details

`-B` Add thousands' delimiters

`-d -d` More more details

`-x` Print machine-readable output

- More info

- http://web.eece.maine.edu/~vweaver/projects/perf_events/
- Gregg's *perf* Examples: <http://www.brendangregg.com/perf.html>

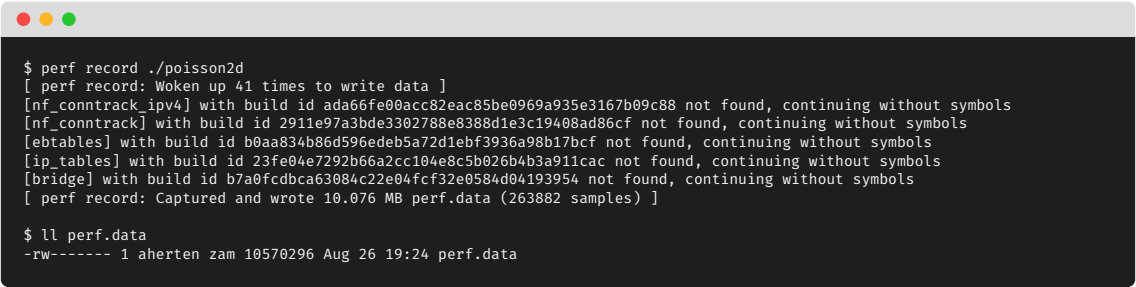
→ <https://perf.wiki.kernel.org/>



Deeper Analysis with perf

perf report: Zoom to main()

Usage: perf record ./app



```
$ perf record ./poisson2d
[ perf record: Woken up 41 times to write data ]
[nf_contrack_ipv4] with build id ada66fe00acc82eac85be0969a935e3167b09c88 not found, continuing without symbols
[nf_contrack] with build id 2911e97a3bde3302788e8388d1e3c19408ad86cf not found, continuing without symbols
[ebtables] with build id b0aa834b86d596edeb5a72d1ebf3936a98b17bcf not found, continuing without symbols
[ip_tables] with build id 23fe04e7292b66a2cc104e8c5b026b4b3a911cac not found, continuing without symbols
[bridge] with build id b7a0fcdbca63084c22e04fcf32e0584d04193954 not found, continuing without symbols
[ perf record: Captured and wrote 10.076 MB perf.data (263882 samples) ]

$ ll perf.data
-rw----- 1 aherten zam 10570296 Aug 26 19:24 perf.data
```

Deeper Analysis with perf

perf report: Zoom to main()

```
Samples: 263K of event 'cycles:ppp', Event count (approx.): 228605603717, Thread: poisson2d
Overhead Command Shared Object Symbol
93.00% poisson2d poisson2d [.] main
4.70% poisson2d libm-2.17.so [.] __fmaxf
1.84% poisson2d poisson2d [.] 00000017.plt_call.fmax@@GLIBC_2.17
0.21% poisson2d libm-2.17.so [.] __exp_finite
0.01% poisson2d [kernel.kallsyms] [k] hrtimer_interrupt
0.01% poisson2d [kernel.kallsyms] [k] update_wall_time
0.01% poisson2d libm-2.17.so [.] __GI___exp
0.01% poisson2d [kernel.kallsyms] [k] task_tick_fair
0.01% poisson2d [kernel.kallsyms] [k] rcu_check_callbacks
0.01% poisson2d [kernel.kallsyms] [k] __hrtimer_run_queues
0.01% poisson2d [kernel.kallsyms] [k] __do_softirq
0.01% poisson2d [kernel.kallsyms] [k] _raw_spin_lock
0.01% poisson2d [kernel.kallsyms] [k] timer_interrupt
0.01% poisson2d [kernel.kallsyms] [k] update_process_times
0.01% poisson2d [kernel.kallsyms] [k] tick_sched_timer
0.01% poisson2d [kernel.kallsyms] [k] rcu_process_callbacks
0.01% poisson2d poisson2d [.] 00000017.plt_call.exp@@GLIBC_2.17
0.01% poisson2d [kernel.kallsyms] [k] ktime_get_update_offsets_now
0.01% poisson2d [kernel.kallsyms] [k] account_process_tick
0.01% poisson2d [kernel.kallsyms] [k] run_posix_cpu_timers
0.00% poisson2d [kernel.kallsyms] [k] trigger_load_balance
0.00% poisson2d [kernel.kallsyms] [k] scheduler_tick
0.00% poisson2d [kernel.kallsyms] [k] clear_user_page
0.00% poisson2d [kernel.kallsyms] [k] update_cfs_shares
0.00% poisson2d [kernel.kallsyms] [k] tick_do_update_jiffies64
```

Deeper Analysis with perf

perf report: Zoom to main()

```
main /gpfs/homeb/zam/aherten/NVAL/OtherProgramming/OpenPOWER-SC17/PAPI-Test/poisson2d
0.00      lwz     r9,100(r31)
          mullw  r9,r10,r9
1.01      extsw  r9,r9
          lwz     r10,140(r31)
0.00      add     r9,r10,r9
          extsw  r9,r9
0.00      rldicr r9,r9,3,60
0.00      ld      r10,184(r31)
14.28     add     r9,r10,r9
0.00      lfd     f12,0(r9)
0.00      lwz     r10,136(r31)
0.00      lwz     r9,100(r31)
0.00      mullw  r9,r10,r9
          extsw  r9,r9
1.32      lwz     r10,140(r31)
0.00      add     r9,r10,r9
0.00      extsw  r9,r9
          rldicr r9,r9,3,60
0.00      ld      r10,168(r31)
          add     r9,r10,r9
22.54     lfd     f0,0(r9)
0.01      fsub   f0,f12,f0
0.00      fabs   f0,f0
0.00      fmr     f2,f0
          lfd     f1,128(r31)
          bl      10000780 <00000017.plt_call.fmax@@GLIBC_2.17>
1.03      ld      r2,24(r1)
Press 'h' for help on key bindings
```

Appendix

PAPI Supplementary

PAPI: High Level API

Usage: Source Code

```
// Setup
```

```
float realTime, procTime, mflops, ipc;  
long long flpins, ins;
```

```
// Initial call
```

```
PAPI_flops(&realTime, &procTime, &flpins, &mflops);  
PAPI_ipc(&realTime, &procTime, &ins, &ipc);
```

```
// Compute
```

```
mult(m, n, p, A, B, C);
```

```
// Finalize call
```

```
PAPI_flops(&realTime, &procTime, &flpins, &mflops);  
PAPI_ipc(&realTime, &procTime, &ins, &ipc);
```



PAPI: Low Level API

Usage: Source Code

```
int EventSet = PAPI_NULL;
long long values[2];

// PAPI: Setup
PAPI_library_init(PAPI_VER_CURRENT);
PAPI_create_eventset(&EventSet);
// PAPI: Test availability of counters
PAPI_query_named_event("PM_CMPLU_STALL_VSU");
PAPI_query_named_event("PM_CMPLU_STALL_SCALAR");
// PAPI: Add counters
PAPI_add_named_event(EventSet, "PM_CMPLU_STALL_VSU");
PAPI_add_named_event(EventSet, "PM_CMPLU_STALL_SCALAR");
// PAPI: Start collection
PAPI_start(EventSet);
// Compute
do_something();
// PAPI: End collection
PAPI_CALL( PAPI_stop(EventSet, values), PAPI_OK );
```

Pre-processor macro
for checking results!
See *next slides!*

PAPI Error Macro: C++

For easier status code checking

```
#include "papi.h"
#define PAPI_CALL( call, success )           \
{                                             \
    int err = call;                          \
    if ( success != err)                     \
        std::cerr << "PAPI error for " << #call << " in L" << __LINE__ << " of " << \
        ↪ __FILE__ << ": " << PAPI_strerror(err) << std::endl; \
}
// Second argument is code for GOOD,
// e.g. PAPI_OK or PAPI_VER_CURRENT or ...
// ...
// Call like:
PAPI_CALL( PAPI_start(EventSet), PAPI_OK );
```

PAPI Error Macro: C

For easier status code checking

```
#include "papi.h"
#define PAPI_CALL( call, success )      \
{                                       \
    int err = call;                     \
    if ( success != err)                 \
        fprintf(stderr, "PAPI error for %s in L%d of %s: %s\n", #call, __LINE__, \
↪    __FILE__, PAPI_strerror(err)); \
}
// Second argument is code for GOOD,
// e.g. PAPI_OK or PAPI_VER_CURRENT or ...
// ...
// Call like:
PAPI_CALL( PAPI_start(EventSet), PAPI_OK );
```

libpfm4

A helper Library

- Helper library for setting up counters interfacing with perf kernel environment
- Used by PAPI to resolve counters
- Handy as translation: Named counters → raw counters
- Use command line utility `perf_examples/evt2raw` to get raw counter for perf



```
$ ./evt2raw PM_CMPLU_STALL_VSU  
r2d012
```

→ http://perfmon2.sourceforge.net/docs_v4.html

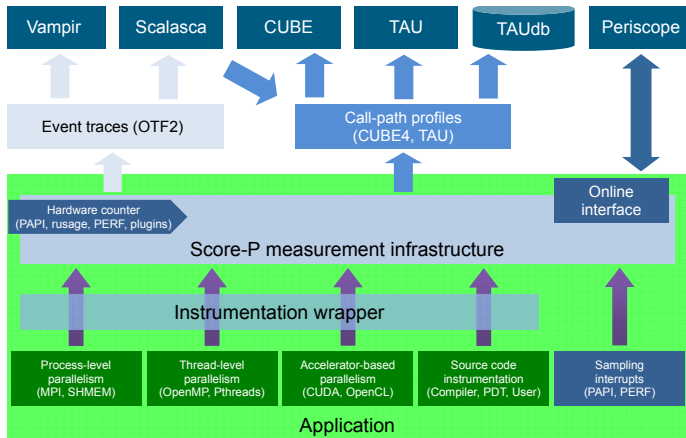
Appendix

Score-P

Score-P

Introduction

- Measurement infrastructure for profiling, event tracing, online analysis
- Output format input for many analysis tools (Cube, Vampir, Periscope, Scalasca, Tau)



Score-P

Howto

- Prefix compiler executable by scorep

```
$ scorep clang++ -o app code.cpp
```

- Adds instrumentation calls to binary
 - Profiling output is stored to file after run of binary
 - Steer with environment variables at **run time**

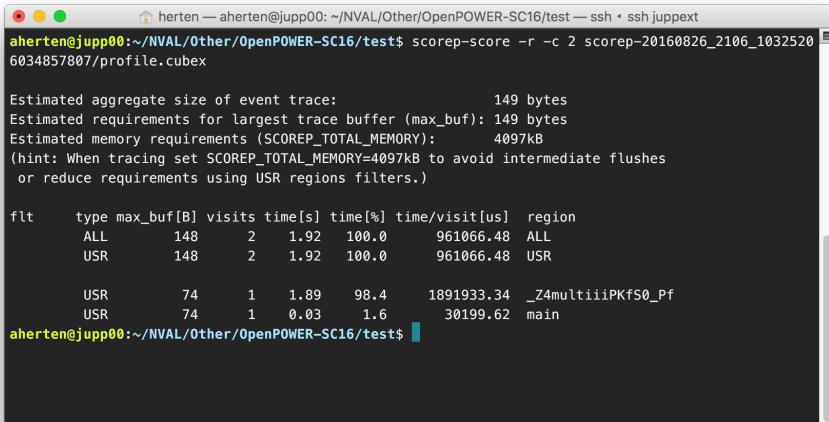
```
$ export SCOREP_METRIC_PAPI=PAPI_FP_OPS,PM_CMPLU_STALL_VSU  
$ ./app
```

- ⇒ Use different PAPI counters per run!
 - Quick visualization with [Cube](#); scoring with scorep-score

Score-P

Performance counter analysis with cube_dump

Usage: scorep-score -r *FILE*



```
aherten@jupp00:~/NVAL/Other/OpenPOWER-SC16/test$ scorep-score -r -c 2 scorep-20160826_2106_1032520
6034857807/profile.cubex

Estimated aggregate size of event trace:          149 bytes
Estimated requirements for largest trace buffer (max_buf): 149 bytes
Estimated memory requirements (SCOREP_TOTAL_MEMORY): 4097kB
(hint: When tracing set SCOREP_TOTAL_MEMORY=4097kB to avoid intermediate flushes
or reduce requirements using USR regions filters.)

flt      type max_buf[B] visits time[s] time[%] time/visit[us] region
      ALL      148       2   1.92  100.0    961066.48  ALL
      USR      148       2   1.92  100.0    961066.48  USR

      USR       74       1   1.89   98.4    1891933.34  _Z4multiiPKfS0_Pf
      USR       74       1   0.03   1.6     30199.62  main

aherten@jupp00:~/NVAL/Other/OpenPOWER-SC16/test$
```


Score-P

Performance counter analysis with cube_dump

Usage: cube-dump -m *METRIC FILE*



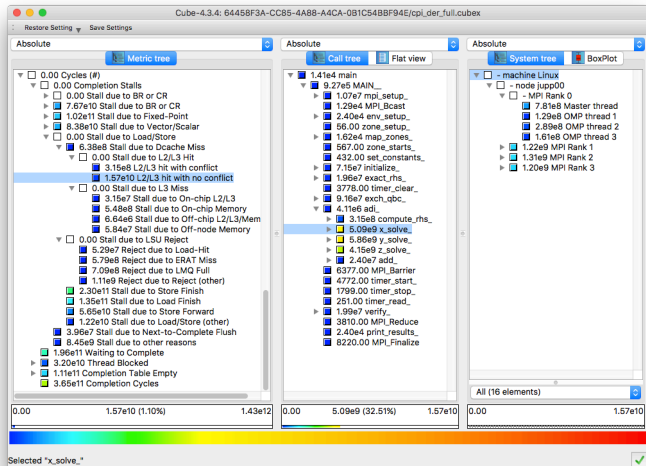
```
herten — aherten@jupp00: ~/NVAL/Other/OpenPOWER-SC16/test — ssh jupp
aherten@jupp00:~/NVAL/Other/OpenPOWER-SC16/test$ cube_dump -m PAPI_FP_OPS scorep-20160830_1138_106367049
09174321/profile.cubex

===== DATA =====
Print out the data of the metric PAPI_FP_OPS

Master thread
-----
main(id=0)          721459
_Z4multiiPKfS0_Pf(id=1) 432004097
aherten@jupp00:~/NVAL/Other/OpenPOWER-SC16/test$
```

Score-P

Analysis with Cube



Appendix

GPU Counters

GPU Example Events & Metrics

<i>NAME</i>	<i>NVIDIA Description (quoted)</i>
<code>gld_inst_8bit</code>	Total number of 8-bit global load instructions that are executed by all the threads across all thread blocks.
<code>threads_launched</code>	Number of threads launched on a multiprocessor.
<code>inst_executed</code>	Number of instructions executed, do not include replays.
<code>shared_store</code>	Number of executed store instructions where state space is specified as shared, increments per warp on a multiprocessor.
<code>executed_ipc</code>	Instructions executed per cycle
<code>achieved_occupancy</code>	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
<code>l1_cache_local_hit_rate</code>	Hit rate in L1 cache for local loads and stores
<code>gld_efficiency</code>	Ratio of requested global memory load throughput to required global memory load throughput.
<code>flop_count_dp</code>	Number of double-precision floating-point operations executed non-predicated threads (add, multiply, multiply-accumulate and special)
<code>stall_pipe_busy</code>	Percentage of stalls occurring because a compute operation cannot be performed because the compute pipeline is busy

Measuring GPU counters

Tools

CUPTI C/C++-API through `cupti.h`

- Activity API: Trace CPU/GPU activity
- Callback API: Hooks for own functions
- Event / Metric API: Read counters and metrics

→ Targets developers of profiling tools

PAPI All PAPI instrumentation through PAPI-C, e.g.
`cuda::device:0:threads_launched`

Score-P Mature CUDA support

- Prefix `nvcc` compilation with `scorep`
- Set environment variable `SCOREP_CUDA_ENABLE=yes`
- Run, analyze

nvprof, Visual Profiler, Nsight Compute NVIDIA's solutions



nvprof

GPU command-line measurements

Usage: `nvprof --events AB --metrics C,D ./app`

```
herten — aherten@JUHYDRA: ~/cudaSamples/NVIDIA_CUDA-7.5_Samples/bin/x86_64/linux/release — .linux/release — ssh juhydra
ixMulCUDA<int=32>(float*, float*, float*, int, int)" (0 of 3)==18158== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, flo
at*, int, int)" (0 of 3)==18158== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
==18158== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (0 of 3)==18158== Replaying kernel "void matr
ixMulCUDA<int=32>(float*, float*, float*, int, int)" (0 of 3)==18158== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, flo
at*, int, int)" (0 of 3)==18158== Replaying kernel "void matrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Performance= 1.69 GFlop/s, Time= 77.513 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
==18158== Profiling application: ./matrixMul
==18158== Profiling result:
==18158== Event result:
Invocations
Device "Tesla K40m (0)"
Kernel: void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
301 threads_launched 204800 204800 204800

==18158== Metric result:
Invocations
Device "Tesla K40m (0)"
Kernel: void matrixMulCUDA<int=32>(float*, float*, float*, int, int)
301 flop_count_sp Floating Point Operations(Single Precisi 131072000 131072000 131072000
301 ipc Executed IPC 1.472345 1.486837 1.480249
301 achieved_occupancy Achieved Occupancy 0.960357 0.989658 0.975385

# aherten @ JUHYDRA in ~/cudaSamples/NVIDIA_CUDA-7.5_Samples/bin/x86_64/linux/release [21:47:45]
$ nvprof --events threads_launched --metrics flop_count_sp,ipc,achieved_occupancy ./matrixMul
```

nvprof

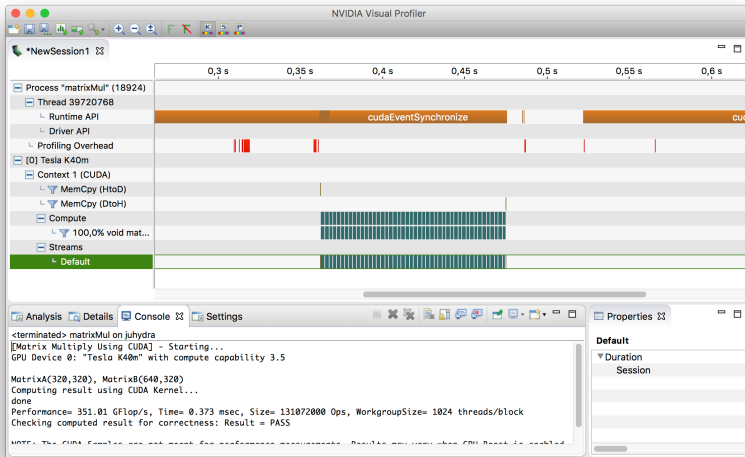
Useful hints

Useful parameters to nvprof

- `--query-metrics` List all metrics
- `--query-events` List all events
- `--kernels name` Limit scope to kernel
- `--print-gpu-trace` Print timeline of invocations
- `--aggregate-mode off` No aggregation over all multiprocessors (average)
 - `--csv` Output a CSV
- `--export-profile` Store profiling information, e.g. for Visual Profiler

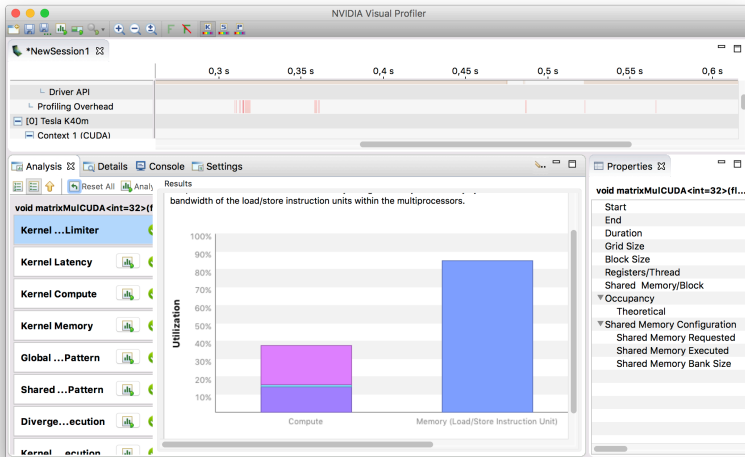
Visual Profiler

Analysis experiments



Visual Profiler

Analysis experiments



Appendix

Glossary & References

Glossary I

CPI Cycles per Instructions; a metric to determine efficiency of an architecture or program. [5](#)

IPC Instructions per Cycle; a metric to determine efficiency of an architecture or program. [5](#)

MPI The Message Passing Interface, a API definition for multi-node computing. [8](#)

NVIDIA US technology company creating [GPUs](#). [23](#), [52](#), [53](#)

OpenMP Directive-based programming, primarily for multi-threaded machines. [8](#)

PAPI The Performance API, a C/C++ API for querying performance counters. [2](#), [14](#), [18](#), [19](#), [20](#), [21](#), [22](#), [24](#)

Glossary II

perf Part of the Linux kernel which facilitates access to performance counters; comes with command line utilities. [2](#), [14](#), [15](#), [16](#), [17](#), [18](#), [24](#)

POWER CPU architecture from IBM, earlier: PowerPC. See also POWER8. [60](#)

POWER8 Version 8 of IBM's **POWER** processor, available also within the OpenPOWER Foundation. [60](#)

POWER9 The latest version of IBM's **POWER** processor. [2](#), [8](#), [9](#), [10](#), [11](#), [12](#), [24](#), [26](#), [29](#), [30](#), [31](#)

Score-P Collection of tools for instrumenting and subsequently scoring applications to gain insight into the program's performance. [8](#), [14](#), [24](#)

References I

- [2] Terje Mathisen. *Pentium Secrets*. URL: http://www.gamedev.net/page/resources/_/technical/general-programming/pentium-secrets-r213 (page 7).
- [3] IBM. *Power ISA™, Version 3.0 B*. Chapter 9. Performance Monitor Facility. 2017. URL: https://wiki.raptorcs.com/w/images/c/cb/PowerISA_public.v3.0B.pdf (page 8).
- [4] Donald E. Knuth. “Structured Programming with Go to Statements”. In: *ACM Comput. Surv.* 6.4 (Dec. 1974), pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640. URL: <http://doi.acm.org/10.1145/356635.356640> (page 28).
- [5] IBM. *POWER9 Performance Monitor Unit User’s Guide*. Version 1.2. Nov. 2018. URL: https://wiki.raptorcs.com/w/images/6/6b/POWER9_PMU_UG_v12_28NOV2018_pub.pdf (pages 30, 32).

References II

- [6] Brandon Gregg. *perf Examples*. URL: <http://www.brendangregg.com/perf.html> (page 35).

References: Images, Graphics I

- [1] Sabri Tuzcu. *Time is money*. Freely available at Unsplash. URL: <https://unsplash.com/photos/r1EwRkl1P1I>.
- [7] Score-P Authors. *Score-P User Manual*. URL: <http://www.vi-hps.org/projects/score-p/>.