Likwid-perfctr

GROUP-3
Bene, Cristian, Sohaib, Timea

Organization

- What is Likwid-perfctr?
- How to use the marker API of Likwid-perfctr?
- What are event groups?
- Problem1: game.cpp
  - What event groups are valid for our problem?
  - Analysis and improvement suggestions for the problem.
- Problem2: dtrmv.cpp
  - Measuring load imbalance by manually calculating FLOPS/core.
  - Measuring load imbalance by using hardware counters.

Likwid-perfctr - Definition

- A framework for measuring an applications' interaction with the hardware using hardware performance counters.

- Uses the msr module (`/dev/cpu/CPUNUM/msr`) to interact with model specific registers in order to read out hardware performance counters.

Likwid-perfctr - Operation

- Can be operated in four modes:
  - Wrapper mode – No modification for application code needed.
  - Stethoscope mode - Measure performance counters for a variable time duration independent of any code running.
  - Timeline mode – Output performance metrics in a specified frequency.
  - Marker-API – Measure specific regions of your code.

Likwid-perfctr – Marker API

- API only read out the counters. Counter config is still handled by wrapper app – `likwid-perfctr`.

- Needs to be linked against the likwid library and pthreads enabled.

```
$ gcc -O3 -fopenmp -pthread -o test dofp.c \
    -DLIKWID_PERFMON -I<PATH_TO_LIKWID>/include \
    -L<PATH_TO_LIKWID>/lib -llikwid -lm
```

Likwid-perfctr – Marker API (2)

```
// This block enables to compile the code with and without the likwid header in place
#ifdef LIKWID_PERFMON
#include <likwid.h>
#else
#define LIKWID_MARKER_INIT
#define LIKWID_MARKER_THREADINIT
#define LIKWID_MARKER_SWITCH
#define LIKWID_MARKER_REGISTER(regionTag)
#define LIKWID_MARKER_START(regionTag)
#define LIKWID_MARKER_STOP(regionTag)
#define LIKWID_MARKER_CLOSE
#define LIKWID_MARKER_GET(regionTag, nevents, events, time, count)
#endif

LIKWID_MARKER_INIT;
LIKWID_MARKER_THREADINIT;

LIKWID_MARKER_START("Compute");
// Your code to measure
LIKWID_MARKER_STOP("Compute");
LIKWID_MARKER_CLOSE;
```

Likwid-perfctr – Marker API (3)

- To reduce influence of LIKWID_MARKER_START preregister a region name.
  ```
  LIKWID_MARKER_REGISTER("Compute")
  ```

- To run a serial process:

  ```
  likwid-perfctr -C S0:0 -g BRANCH -m ./a.out
  ```

Likwid-perfctr - Event Groups

- Event groups are essentially a group of events on which hardware counters are triggered. For example the BRANCH group has counters for `BR_INST_RETIRED_ALL_BRANCHES` and `BR_MISP_RETIRED_ALL_BRANCHES`.

- The type of event groups available depends on the architecture.

- To view the event group available for the architecture:
  `likwid-perfctr -a`

Problem 1 – game.cpp

- Array of bodies

```
RigidBody** bodies = new RigidBody*[N];
```

- Move them serially:

```
for (int n = 0; n < N; ++n) {
    bodies[n]->move(0.0, 0.0, 0.5 * 9.81 * dt * dt);
}
```

Problem 1 - What event groups are relevant?

- Refer to GIT and explain.

Problem 1 – Analysis and improvement

- Of all the counters that we studied we noticed two that were interesting.
  - L2CACHE
  - UOPS (combines UOPS_EXEC/ UOPS_ISSUE/ UOPS_RETIRE

Problem 1 – Analysis and improvement (2)

- L2Cache - About 33% miss ratio.

Metric	Core 0
Runtime (RDTSC) [s]	0.0065
Runtime unhalted [s]	0.0061
Clock [MHz]	3600.3981
CPI	27.9746
L2 request rate	1.3250
L2 miss rate	0.4360
L2 miss ratio	0.3291

Problem 1 – Analysis and improvement (3)

- UOPS –
  Retired UOPS are lower compared to executed ops which means speculative ops were mostly wrong.

Metric	Core 0
Runtime (RDTSC) [s]	0.0066
Runtime unhalted [s]	0.0062
Clock [MHz]	3600.6691
CPI	27.8658
Issued UOPs	4767173
Merged UOPs	4380
Executed UOPs	4958618
Retired UOPs	3487210

Problem 1 – Analysis and improvement (4)

- Problem?
  Non contiguous memory

- Solution: Allocation memory contiguously for better cache usage.

```
RigidBody* bodies = new RigidBody[N];
```

Problem 1 – Analysis and improvement (5)

- L2Cache:
  After improvement

Metric	Core 0
Runtime (RDTSC) [s]	0.0003
Runtime unhalted [s]	0.0002
Clock [MHz]	1200.1153
CPI	0.8610
L2 request rate	0.3470
L2 miss rate	0.0681
L2 miss ratio	0.1963

Problem 1 – Analysis and improvement (6)

- UOPS:
  After improvement

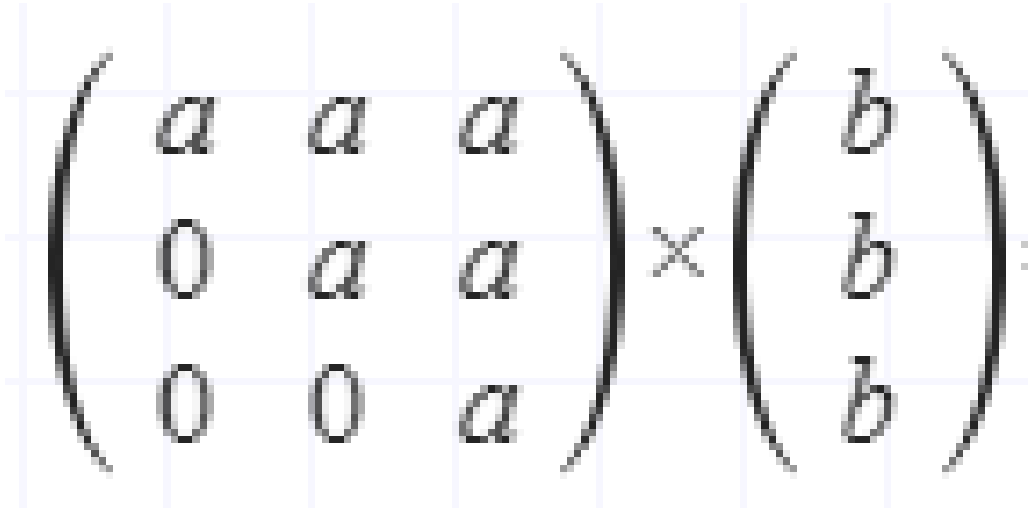
Metric	Core 0
Runtime (RDTSC) [s]	0.0003
Runtime unhalted [s]	0.0002
Clock [MHz]	1200.0367
CPI	0.8447
Issued UOPs	621589
Merged UOPs	4380
Executed UOPs	714472
Retired UOPs	705403

Problem 1 – Analysis and improvement (7)

- Can also use AVX instructions for vectorization (update all three axes at the same time).
  Significant rewrite involved.

Problem 2 - dtrmv.cpp

- Computes upper-triangular matrix times vector multiplication



- Goal: Find and fix load imbalances

Problem 2 – Calculating flops by hand

- Upper-triangular matrix * vector has $O(2 * (\sum_{i=0}^{N} N - i))$ flops
- Total number of flops: 100010000
- Processor has 14 cores -> assuming 14 threads
- loop is divided equally into chunks
- N is 10000
- 4 cores execute 715 loop runs, 10 execute 714 loop runs
- By inserting the corresponding values in the formula the flops can be computed

Problem 2 – Calculating flops by hand (2)

Core	#flops
Core 0	13789490
Core 1	12767040
Core 2	11744590
Core 3	10722140
Core 4	9686838
Core 5	8667246
Core 6	7647654
Core 7	6628062
Core 8	5608470
Core 9	4660388
Core 10	3569286
Core 11	2549694
Core 12	1530102
Core 13	510510

Problem 2 – Using hardware counters

- Haswell has no events to measure FLOPS directly
- It has AVX_INSTS events for AVX & AVX2 256-bit instructions
- The FLOPS_AVX group uses the AVX_INSTS_CALC event for MFLOP/s

```
Packed SP MFLOP/s = 1.0E-06*(AVX_INSTS_CALC*8)/runtime
Packed DP MFLOP/s = 1.0E-06*(AVX_INSTS_CALC*4)/runtime
```

- In total around 50768000 DP instructions
- Different from hand calculation, but imbalance still visible

Problem 2 – Using hardware counters (2)

- Cores clearly execute different amounts of these instructions

Event	Counter	Core 0	Core 1	Core 2	Core 3
INSTR_RETIRED_ANY	FIXC0	137081800	180522400	271493300	371803100
CPU_CLK_UNHALTED_CORE	FIXC1	478382500	398217400	402163600	399593900
CPU_CLK_UNHALTED_REF	FIXC2	427817300	414105400	419766800	417029500
AVX_INSTS_CALC	PMC0	55168750	39543710	23918670	8289290

- This is the case for different numbers of threads

Problem 2 – INSTR_RETIRED_ANY

- INSTR_RETIRED_ANY counts all relevant completed instructions
- Disregards extra instruction executed due to speculative execution
- Instruction count varies greatly for different numbers of threads
- AVX_INSTS_CALC pretty much stays the same

- INTR_RETIRED_ANY seems not really useful in this case

Problem 2 – Balancing the load

```
void dtrmv(double const* A, double const* x, double* y) {
 #pragma omp parallel
 {
  LIKWID_MARKER_START("dtrmv");
  #pragma omp for
  for (int i = 0; i < N; ++i) {
    double sum = 0.0;
    for (int j = i; j < N; ++j) {
      sum += A[i*N + j] * x[j];
    }
    y[i] = sum;
  }
  LIKWID_MARKER_STOP("dtrmv");
 }
}
```

Problem 2 – Balancing the load (2)

- By default the *omp parallel for* schedule is *static*
- Loop is divided into equally sized chunks

- Dynamic schedule: assigns chunks with specified size, has overhead
- Guided schedule: Chunks start relatively large and become smaller, less overhead
- Static schedule with fixed chunk size: workload is decreasing, using a small chunk size could be enough, avoids overhead

Problem 2 – Balancing the load (3)

```
void dtrmv(double const* A, double const* x, double* y) {
 #pragma omp parallel
 {
  LIKWID_MARKER_START("dtrmv");
  //#pragma omp for schedule(dynamic,100)
  //#pragma omp for schedule(guided,100)
  #pragma omp for schedule(static,5)
  for (int i = 0; i < N; ++i) {
    double sum = 0.0;
    for (int j = i; j < N; ++j) {
      sum += A[i*N + j] * x[j];
    }
    y[i] = sum;
  }
  LIKWID_MARKER_STOP("dtrmv");
 }
}
```

Problem 2 – Balancing the load (4)

- Dynamic and guided scheduling improve load balancing and execution speed a little bit
- Static scheduling with a chunk size of 5 works really good

Event	Counter	Core 0	Core 1	Core 2	Core 3
INSTR_RETIRED_ANY	FIXC0	96696990	79239380	78421820	82838940
CPU_CLK_UNHALTED_CORE	FIXC1	369551100	321273900	323575400	322672100
CPU_CLK_UNHALTED_REF	FIXC2	341266400	334607800	332108700	337404200
AVX_INSTS_CALC	PMC0	31779350	31744010	31713530	31683530

Questions?