Friedrich-Alexander-Universität
Technische Fakultät

Computer Architecture Practical Exercise

5 Tools

Kenan Gündogan[1] **Philipp Gündisch**[1]

[1]Friedrich-Alexander Universität Erlangen-Nürnberg, Chair of Computer Science 3 (Computer Architecture)

November 29, 2024

Tools

Performance Engineering

Optimizing code is based on understanding the underlying hardware and adapting the code for it. The following tools help us understand and control how the code is performing.

- **Processor Information**
  To retrieve number of cores and cache hierarchy
- **Static Code Analysis**
  To reduce code overhead
- **Runtime Profiling**
  To identify critical part of code
- **Performance Analysis**
  To understand critical part of code
- **Data and Thread Placement**
  To map threads to cores and data to caches

`LIKWID` Toolkit

Like I Knew What I'm Doing

The LIKWID tool kit is developed by the RRZE and consists of many useful command line tools. In this exercise we will use the following tools:

- `likwid-topology`
- `likwid-perfctr`
- `likwid-pin` (later)

Task 5.1: Processor Information

with `likwid-topology`

Log in to cluster and run the `likwid-topology` command. Crosscheck the results with `lstopo` (with `lstopo -of svg > fritz.svg` the overview from slide 0 is created).

- Note down the cache sizes (in kB) for each level
- What jacobi grid size (in cells) fits in each cache level?
- What vector length (in elements) can be summed with `vec_sum()` per cache level?
- Fill the following table

	Cache Size	Jacobi Grid Size	Vector Length
L1			
L2			
L3			

Task 5.2: Runtime Profiling

with gprof

To evaluate the program with `gprof` you need to compile the program with `-pg` and run it.

```
$ icx -pg ...                   // Works also with gcc
$ ./bin/vec_sum 1000 1000       // This step produces a gmon.out file
$ gprof ./bin/vec_sum ./gmon.out
```

`gprof` shows a high level overview about which functions consume what fraction of time.
Once a high running function was identified we can use `likwid-perfctr`.

Task 5.3: Runtime Profiling

with `likwid-perfctr` (1/3)

To evaluate the program with `likwid-perfctr` you need to adapt your C code. An example how to use it is provided below. Additional `#ifdef` pragmas might be helpful.

```c
#include <likwid-marker.h>

...

LIKWID_MARKER_INIT;
for(runs = 1u; actual_runtime_us < minimal_runtime_us; runs = runs << 1u) {
        start = get_time_micros();
        LIKWID_MARKER_RESET("MARKER_NAME"); // Ignore error from first iteration
        LIKWID_MARKER_START("MARKER_NAME");
        for (i=0; i < runs; ++i) {
                jacobi(grid_old, grid_new, X, Y);
                // ... swap
        }
        LIKWID_MARKER_STOP("MARKER_NAME");
        stop  = get_time_micros();
        actual_runtime_us = stop - start;
}
LIKWID_MARKER_CLOSE;
runs /= 2u;
```

Task 5.3: Runtime Profiling

with likwid-perfctr (2/3)

Additionally, the compiler needs to know about the LIKWID library location.

```
# Makefile Update
CFLAGS_LIKWID := -I/apps/likwid/5.3.0/include -DLIKWID_PERFMON
LDLAGS_LIKWID := -pthread -L/apps/likwid/5.3.0/lib/ -llikwid
```

Also sbatch needs to know about the hardware profiling:

```
# In SBATCH script
...
#SBATCH --constraint=hwperf

module load likwid/5.3.0

# ... for loop etc.
srun likwid-perfctr
        -O --stats \ # print in parseable format
        -o <file> \  # path to output file
        -C 0 \       # measure only on core 0
        -c 0 \       # pin program to core 0
        -f \         # force overwrite registers
        -m \         # only measure instrumented part
        -g MEM_LOAD_RETIRED_L1_ALL:PMC1,MEM_LOAD_RETIRED_L1_HIT:PMC2 \
        ./jacobi $X $Y
```

Further information can be found in the LIKWID manual.

Task 5.3: Runtime Profiling

with `likwid-perfctr` (3/3)

- Update your project to run jacobi with likwid
- Choose the naive version of jacobi for reference
- Implement a new version of the naive jacobi but iterate column wise
- Measure the presented performance counters for each implementation
- (Optional) Try different performance counters (*see* `likwid-perfctr -e`)
- Extract the relevant information from the perfctr log files
- Plot a graph to show the difference of the two implementations

Task Overview

- E 5.1: Processor Information
  - Use `likwid-topology` and `lstopo`
  - Fill the table
- E 5.2: Runtime Profiling `gprof`
  - Try out `gprof` on an existing implementation
- E 5.3: Runtime Profiling `likwid-perfctr`
  - Analyze two `jacobi` implementations with `likwid`
  - Plot the results to show the difference
  - Interpret the results

Appendix: CSV Data Extraction

There are multiple ways to extract the csv data such that you can plot it.

- Import both files in Excel or similar
- Use Python csv module
- Use `cat, cut and grep` in the bash script to select a value e.g.
  `LOADS=$(cat <file> | grep MEM_LOAD_RETIRED_L1_ALL| grep STAT | cut -d , -f 5)`

Appendix: Checklist

Performance Optimization (1/2)

During the timeline of this class new bullet points will be added. Recently added entries are bold.

- Compiling
  - Choice of the compiler (`icx`)
  - Compiler flag to optimize aggressively (e.g. `-O3`)
  - Compiler flag to adapt for specific hardware (e.g. `-xHost`)
- Programming Techniques (if applicable)
  - Use `#define` and `const` instead of variables
  - Data type aware programming
  - Use aligned memory (e.g. with `_mm_malloc()` or `posix_memalign()`)
  - Consecutive address iteration
  - Variable declarations outside of loops
  - Reduce function calls
  - Use intrinsics (to utilize SIMD)

Appendix: Checklist

Performance Optimization (2/2)

During the timeline of this class new bullet points will be added. Recently added entries are bold.

- Measurement
  - Reasonable benchmark time
  - Reasonable benchmark workload
  - Reduce interference factors to a minimum
- Optimization Process
  - **Check assembler code while optimizing**
  - **Check performance gains while optimizing**
  - **Use profiling tools**
  - **Ensure correctness of code**
  - **Optimize iteratively**