Praktikum im WS 2023/2024
Evaluierung moderner HPC-Architekturen und -Beschleuniger (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Sergej Breiter, MSc., Minh Thanh Chung, MSc., Amir Raoofy, MSc.,
Dr. Karl Fürlinger, Dr. Josef Weidendorfer

Assignment 03 – Due: 9. Nov 2023, 16:00

In this assignment, we take a look at *performance profiling and analysis tools*. Profiling is an essential part of understanding and optimizing performance. These tools help to characterize the performance of your application and measure what really happens.

For detailed analyses, modern hardware provides so-called *hardware performance counters*, which can count various *events* without overhead after configuration. Performance analysis tools use these counters either by just reading them, or for *sampling profiling* to generate an overall statistical overview of performance through *sampling* of events. For sampling, the tools configure the counters to generate an interrupt after a given number of events or time. Thus, they only look at every n-th event and store the context whenever that is happening. E.g. you can ask the tool to check for the *instruction pointer* on every 1000-th memory access. Using debug information, the tool can relate the instruction pointer to a function name and even the source line. This results in a statistical distribution of how memory accesses are done across your source code.

1. **Measurement with Linux Perf**

   First, we look at whole-program performance measurement on CPUs with "`perf`". This is a user-level tool delivered as part of Linux Kernel sources. It uses the Linux kernel support for performance counters across various architectures.

   (a) Run "`perf list`" on the various systems in BEAST. Try to explain the purpose of classes of events available, and what may be the most interesting ones for analyzing the triad microbenchmark (CPU version). Focus on events which allow to do roofline analysis figures. Hint: How do you get the operational intensity for a kernel from measurements?

   (b) For whole program analysis, run "`perf stat <program>`" on your code from previous assignments on an architecture of your choice. With "`-e <event>`", try to measure the events you would need to draw roofline figures. For this, look up the limits on the roofline for the chosen architecture.

   (c) Now use "`perf record`" to do the same via sampling, first over time (that is, cycles), but also using various other event types. Read the `perf` manual to see and explain how can you use "`perf report`" to see the distribution of events annotated to (1) source code

lines, (2) disassembly output. Bonus tip: for an interactive experience of "`perf record / report`", there is "`perf top`" (may not work due to permission, as it shows sampling over the full system).

2. **First-Person Hardware Counter Measurement with Likwid**

Likwid [1] is a set of command line performance analysis tools. The tool `likwid-perfctr` [2] enables configuring and accessing performance counters.

```
$ module load likwid              # required on every login
$ likwid-perfctr -e | less        # list available performance events
$ likwid-perfctr -a               # list available performance groups
```

Likwid provides predefined *performance groups* to facilitate profiling. Multiple events are measured simultaneously in a performance group to derive particular performance metrics.

```
$ likwid-perfctr -g MEM -H        # show memory bandwidth group
$ likwid-perfctr -g FLOPS_DP -H   # show double precision flops group
```

This shows the formulas used by Likwid to calculate memory bandwidth utilization (data volume per time transferred between memory and CPU) and double precision flop rate (floating-point operations per time) – two examples for useful performance metrics – from hardware performance events on the current architecture.

To actually measure performance events, we need to specify the performance group of interest, the thread-to-core pinning, and the cores being measured on (see Listing 1). The option `-C 0` [3] specifies to pin the thread to core 0, and also to measure performance events on core 0.

Listing 1: Measuring memory bandwidth with `likwid-perfctr`

```
$ likwid-perfctr -C 0 -g MEM <program>
```

Likwid Marker API

In this task, you will measure performance events within a code region, instead of the whole program. LIKWID has a C-API (*Marker API*) that allows to measure events within user-defined code regions with `likwid-perfctr` (add the `-m` command line option).

Listing 2 shows an example of how to use the Marker API in C code. You have to initialize the library, mark begin and end of the measured code region(s), and provide a name.

[1] `https://github.com/RRZE-HPC/likwid/wiki/`
[2] see `likwid-perfctr -h` or `man likwid-perfctr`
[3] see `man likwid-pin`

Listing 2: Likwid Marker API Example

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

LIKWID_MARKER_INIT;
#pragma omp parallel
{
        LIKWID_MARKER_THREADINIT;
        LIKWID_MARKER_REGISTER("RegionName"); // optional
}
/*** ... some code ... ***/

#pragma omp parallel
        LIKWID_MARKER_START("RegionName");    // start counting

/*****************************/
/*** Your code to measure ***/
/*****************************/

#pragma omp parallel
        LIKWID_MARKER_STOP("RegionName");     // stop counting

/*** ... some code ... ***/

LIKWID_MARKER_CLOSE;
```

(a) **Marker API Implementation:** Take the vector triad from Assignment 1 and modify the code, enabling measurements of (only) the triad performance with LIKWID.

(b) **Performance Groups:** Find the relevant performance groups to measure the floating-point performance and memory / cache bandwidth utilization. Explain the performance metric formulas, and the performance events used in these formulas for one system of your choice.

(c) **Performance Event Measurement:** In this part, we perform measurements of performance events to complete a table similar to Table 1. Run the tasks with a *single thread*, and with `#threads = #physical cores`. Consider appropriate vector sizes corresponding to the cache levels as shown in the table. Keep in mind that some of the caches are private while others are shared. We measure the following performance metrics:

- **Cache level bandwidths**
- **Memory bandwidth**
- **Floating-point performance**

Since the amount of performance counters is limited, we cannot measure all the required events to simultaneously obtain the metrics. You can make multiple measurements to fill the table. Additionally, document the used compiler, version, and flags.

(d) **Interpretation:** Describe your observations and compare the results to the software-based measurement (program output).

i. Investigate whether your code used vector instructions based on the floating-point performance events.

ii. Take a look at the measurements for vectors that fit into L1 cache and compare the performance to the peak performance. What may be the limiting factor in this case?

iii. Does your system use the write-allocate policy?

Size	#Threads	FLOPS	Memory BW	L3-L2 BW	L2-L1 BW
$<$ L1	1				
$>$ L1	1				
$>$ L2	1				
$>$ L3	1				
$>\sum$ L1	#cores				
$>\sum$ L2	#cores				
$>\sum$ L3	#cores				

Table 1: A reference table for showing the results on each system.

Compiling. To compile programs using the Marker API, you must define `LIKWID_PERFMON` and link the LIKWID library. Additionally, you have to add the include directory of `likwid-markers.h` and the library directory of the LIKWID library (see Listing 3).

Listing 3: Modifications in the Makefile to use the Marker API

```
LIKWID_BASE ?= $(shell dirname $(dir $(shell which likwid-perfctr)))
CFLAGS      += -DLIKWID_PERFMON -I$(LIKWID_BASE)/include
LDFLAGS     := -L$(LIKWID_BASE)/lib -llikwid

triad:
        $(CC) $(CFLAGS) -o triad triad.c $(LDFLAGS)
```

3. **GPU Profiling – Vendor Profilers:**

   We now look at whole-program performance measurement using GPU profiling tools provided by the venders.

   (a) **nsys and ncu on ice1:**

   For ice1 machine we use "`nsys`" and "`ncu`". You can find primitive instructions to use these tools in the following:

   ```
   nsys nvprof <executable and arguments>
   ```

   Use the nsys and gather information about data communication between host and the device. Compare the measurements from nsys with the ones you manually obtained from the previous assignment for vector triad.

   ```
   ncu --metrics=launch__thread_count,launch__grid_size,launch__block_size
       <executable and arguments>
   ```

   Use these tools to acquire high-level statistical summary of kernel launches in you vector triad benchmark. Compare the measurements from ncu to the number of teams and number of threads configuration that you set in your codes.

   (b) **rocprof on milan2:**

   For milan2 machine we use "`rocprof`"[4] which is provided by ROCm stack. You can find primitive instructions to use these tools in the following:

   ```
   rocprof --list-basic
   ```

[4] `https://rocmdocs.amd.com/projects/rocprofiler/en/latest/rocprofv1.html`

```
rocprof --list-derived
```

```
rocprof <executable and arguments>
```

```
rocprof --hsa-trace <executable and arguments>
```

```
rocprof --hip-trace <executable and arguments>
```

```
rocprof --sys-trace <executable and arguments>
```

Investigate the resulting traces of your vector triad benchmark. Try to extract metrics from the traces, similar to metrics you gathered on ice1 machine and compare the results to that.

4. **GPU Tracing – THAPI:** Tracing Heterogeneous APIs (THAPI) is a tracing infrastructure for heterogeneous computing applications. It supports the tracing of CUDA API calls (does not currently support HIP API). We only conduct experiments on '`ice1`' machine.

   Prepare the environment using the following commands:

```
module load llvm
module load lttng-tools
module load babeltrace2
module load thapi-master-gcc-11.2.0-z45tvgb
```

   Read the documentation of THAPI in here: https://github.com/argonne-lcf/THAPI. Run tracing of your vector triad benchmarks and investigate the resulting traces.

   (a) **Gather metrics:** Try to extract metrics from the traces, similar to the metrics you gathered in Section 3 Part a.

   (b) **Timeline plot:** Plot a timeline for the API calls and kernel launches and interpret this timeline (Hint: read the documentation to figure out how to get a timeline of the API calls).