Praktikum im SS 2022
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

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

Assignment 03 – Due: 17.11.2022

An important aspect in the evaluation of modern architectures and accelerators is the measurement of performance metrics. In previous assignments, you already manually performed primitive measurements for flop rates and memory bandwidth utilization of your code, based on manual counting of flops and memory accesses and time measurements. In this assignment, we take a look at the so-called *performance profiling and analyzing tools* such as LIKWID, which was briefly introduced in Assignment #0. These tools help to characterize the performance of your applications. We will go for a much more detailed analysis of your code, and they help to understand what happens on the system during the execution, both on CPU and GPU.

For detailed analyses, modern hardware provides so-called *hardware performance counters*, which can count various *events* without any overhead after some 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 so-called *sampling* of events. For sampling, the tools configure the counters to generate an interrupt after a given number of events. 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 the previous assignment (the best CPU version of triad) 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**

In this task, you will measure performance events within a code region, instead of the whole program. LIKWID [1] has a C-API (*Marker API*) that allows to measure events occurring within user-defined code regions with `likwid-perfctr` (`-m` command line option).

Listing 1 shows an example of how to use the Marker API. You have to initialize the library, mark begin and end of the measured code region(s), and provide a name.

Listing 1: Likwid Marker API

```
#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 kernel 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) **FLOPS:** Measure the triad floating-point performance in the case of a *single thread*, and *multiple threads* (`#threads = #physical cores`). Afterwards, collect the results and compare them with the measurement in Assignment #1.

(d) **Vectorization:** Investigate whether your code used vector instructions based on the floating-point performance events. Additionally, document the used compiler, compiler version, and the compiler flags on each system.

[1] `https://github.com/RRZE-HPC/likwid/wiki/`

Size	#Threads	FLOPS	Memory BW	L3 BW	L2 BW
$>$ L1 size	1				
$>$ L2 size	1				
$>$ L3 size	1				
$> \sum$ L1 size	max. cores				
$> \sum$ L2 size	max. cores				
$> \sum$ L3 size	max. cores				

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

(e) **Bandwidth (BW) - Cache levels:** Consider appropriate vector sizes corresponding to the cache levels and measure the cache bandwidth utilization. Similarly, run this task in the case of *single thread*, and *multiple threads*.

(f) **Bandwidth (BW) - Memory:** Choose an appropriate vector size to measure the memory bandwidth utilization. Run the task with *single thread*, and *multiple threads*. Finally, together with Task c and Task e, you can complete a table as Table 1. Then, compare this with the measurements obtained from Assignment #1.

Compiling. To compile programs using the Marker API, you must define `LIKWID_PERFMON` and link against the LIKWID library. Additionally, you have to add the include directory of `likwid-markers.h` and the library directory of the LIKWID library. Modify the Makefile accordingly:

```
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 thx2:**

For thx2 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 rome2:**

For rome2 machine we use "`rocprof`" which is provided by ROCm stack. You can find primitive instructions to use these tools in the following:

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

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

```
rocprof --hsa-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 thx2 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 and matrix multiplication 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).