Praktikum im WS 2021/2022
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)

Vincent Bode, MSc., Minh Chung, MSc., Dennis-Florian Herr, MSc.,
Bengisu Elis, MSc., Dr. Karl Fürlinger, Amir Raoofy, MSc., Dr. Josef Weidendorfer

Assignment 05 – Due: 09.06.2021

Note: This assignments may look quite long, but do not worry: we want you to know various tools for performance analysis, and the actual amount of work should be on the lower side (in contrast to previous assignments).

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* that help to characterize the performance of your applications. These tools are used for a much more detailed analysis of your code, and they help in understanding of what happens on the system during the execution, both on CPU and GPU.

For the detailed analysis, modern hardware provides so-called *hardware performance counters*, which can count various *events* without any overhead after some configuration. Performance analyzing 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 were observed. 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 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 both the triad and matrix-multiplication microbenchmarks (CPU versions). 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 (best CPU versions of triad and matrix multiplication) 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 `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 PAPI**

PAPI is a cross-platform interface/library that allows access to hardware performance counters on CPUs (and recently GPUs). PAPI also supports *sampling*, but it is most often used for direct measurements. In this setting, sections of code are bracketed with PAPI calls that set up and read hardware event counters; the application can query and analyze its own execution characteristics.

A simple example for this usage is shown in Fig. 1. This example sets up counters for instructions executed (`PAPI_TOT_INS`) and CPU clock cycles (`PAPI_TOT_CYC`), and measures these counters in the code block in between the invocations of `PAPI_start_counters()` and `PAPI_read_counters()`.

The list of predefined events available on a platform can be queried with the command `papi_avail`. PAPI offers access to a platform's native events but also tries to define a set of useful events across platforms. The number of counters that can be used simultaneously and restrictions concerning the sets of events that can be counted simultaneously are platform dependent.

In the following we list the instruction for using PAPI for various systems in this assignments. Mind minor differences in the compilation instruction on each system. Also refer to you previous instructions and your experiences from last assignments to set the compiler and suitable flags.

Usage on ice2:

```
module load papi
# use the environment variables PAPI_INC and PAPI_LIBDIR
# for header and library files, e.g.,
<compiler> <flags> $PAPI_INC -L $PAPI_LIBDIR -lpapi ./test.c -o ./test
```

Usage on thx2:

```
module load papi/5.7.0_cuda
<compiler> <flags> -I $PAPI_INC -L $PAPI_LIB -lpapi ./test.c -o ./test
```

Usage on rome2:

```
module load papi/5.7.0_rocm
<compiler> <flags> -I $PAPI_INC -L $PAPI_LIB -lpapi ./test.c -o ./test
```

(a) Take the vector triad kernel from assignment #1 and and use PAPI to measure some hardware performance counters. Try to find similar counters to those you used section 1 of this assignment. Similar to section 1, determine whether the available counters for floating point operations and the memory system correspond with the observed performance from your manual measurements.

(b) Repeat subtask (a) for the matrix multiplication kernel from assignment #2.

(c) For the kernel you measured, compare the expected FLOP rates and memory bandwidth with the measurements. Explain any discrepancies you observe, and in this case, try to find better event types matching your expectations.

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

```
#include "papi.h"

// how many events to monitor
#define NUM_EVENTS 2

...
int events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_TOT_CYC};
long long values[NUM_EVENTS];

if(PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
  handle_error();
}

if( PAPI_start_counters(events, NUM_EVENTS) != PAPI_OK ) {
  handle_error();
}

/* code you want to measure */

if( PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK ) {
  handle_error();
}

for( i=0; i<NUM_EVENTS; i++ ) {
  char evtname[PAPI_MAX_STR_LEN];
  PAPI_event_code_to_name(events[i], evtname);
  printf("%s : %lld\n", evtname, values[i]);
}
```

Figure 1: Code snippet demonstrating the usage of PAPI.

previous assignments for vector triad and matrix matrix multiplication.

```
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 and matrix multiplication benchmarks. 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 and matrix multiplication benchmarks. 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).