

# Notebook

November 17, 2019

## 1 Hands-On: Performance Counters

This Notebook is part of the exercises for the SC19 Tutorial »Application Porting and Optimization on GPU-accelerated POWER Architectures«. It is to be run on a POWER9 machine; in the tutorial: on Ascent, the POWER9 training cluster of Oak Ridge National Lab.

This Notebook can be run interactively on Ascent. If this capability is unavailable to you, use it as a description for executing the tasks on Ascent via a shell access. During data evaluation, the Notebook mentions the corresponding commands to execute in case you are not able to run the Notebook interactively directly on Ascent.

### 1.1 Table of Contents

- Section ??
- Section ??
  - Section ??
  - Section ??
  - Section ??
- Section ??
- Section ??

### 1.2 Task 1: Measuring Cycles and Instructions

Throughout this exercise, the core loop of the Jacobi algorithm is instrumented and analyzed. The part in question is

```
for (int iy = iy_start; iy < iy_end; iy++)
{
    for( int ix = ix_start; ix < ix_end; ix++ )
    {
        Anew[iy*nx+ix] = -0.25 * (rhs[iy*nx+ix] - (A[ iy    *nx+ix+1] + A[ iy    *nx+ix-1]
                                                    + A[(iy-1)*nx+ix  ] + A[(iy+1)*nx+ix  ]));
        error = fmaxr( error, fabsr(Anew[iy*nx+ix]-A[iy*nx+ix]));
    }
}
```

The code is instrumented using PAPI. The API routine `PAPI_add_named_event()` is used to add *named* PMU events outside of the relaxation iteration. After that, calls to `PAPI_start()` and `PAPI_stop()` can be used to count how often a PMU event is incremented.

For the first task, we will measure quantities often used to characterize an application: cycles and instructions.

**TASK:** Please measure counters for completed instructions and run cycles. See the TODOs in file `poisson2d.ins_cyc.c`. You can either edit the files with Jupyter capabilities by clicking on the link of the file or selecting it in the file drawer on the left; or use a dedicated editor on the system (vim is available). The names of the counters to be implemented are `PM_INST_CMPL` and `PM_RUN_CYC`.

After changing the source code, compile it with `make task1` or by executing the following cell (we need to change directories first, though).

*(Using the `Makefile` we have hidden quite a few intricacies from you in order to focus on the relevant content at hand. Don't worry too much about it right now – we'll un-hide it gradually during the course of the tutorial.)*

Section ??

```
In [ ]: !pwd
In [ ]: %cd Tasks/
        # Use `%cd Solutions` to look at the solutions for each task
In [ ]: !make task1
```

Before we launch our measurement campaign we should make sure that the program is measuring correctly. Let's invoke it, for instance, with these arguments: `./poisson2d.ins_cyc.bin 100 64 32` – see the next cell. The 100 specifies the number of iterations to perform, 64 and 32 are the size of the grid in y and x direction, respectively.

```
In [ ]: !./poisson2d.ins_cyc.bin 100 64 32
        # alternatively call !make run_task1
```

Alright! That should return a comma-separated list of measurements.

For the following runs, we are going to use Ascent's compute backend nodes which are not shared amongst users and also have six GPUs available (each!). We use the available batch scheduler *IBM Spectrum LSF* for this. For convenience, a call to the batch submission system is stored in the environment variable `$SC19_SUBMIT_CMD`. You are welcome to adapt it once you get more familiar with the system.

For now, we want to run our first benchmarking run and measure cycles and instructions for different data sizes, as a function of `nx`. The Makefile holds a target for this, call it with `make bench_task1`:

```
In [ ]: !make bench_task1
```

Once the run is completed, let's study the data!

This can be done best in the interactive version of the Jupyter Notebook. In case this version of the description is unavailable to you, call the Makefile target `make graph_task1` (either with X forwarding, or download the resulting PDF).

```
In [ ]: import numpy as np
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
import common
%matplotlib inline
sns.set()
plt.rcParams['figure.figsize'] = [14, 6]
```

Execute the following cell if you want to switch to color-blind-safer colors

```
In [ ]: sns.set_palette("colorblind")

In [ ]: plt.rcParams['figure.figsize'] = [14, 6]
df = pd.read_csv("poisson2d.ins_cyc.bin.csv", skiprows=range(2, 50000, 2)) # Read in
the CSV file from the bench run; parse with Pandas
df["Grid Points"] = df["nx"] * df["ny"] # Add a new column of the number of grid points
(the product of nx and ny)
df.head() # Display the head of the Pandas dataframe
```

Let's have a look at the counters we've just measured and see how they scaling with increasing number of grid points.

*In the following, we are always using the minimal value of the counter (indicated by »(min)«) as this should give us an estimate of the best achievable result of the architecture.*

```
In [ ]: fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
df.set_index("Grid Points")["PM_RUN_CYC (min)"].plot(ax=ax1, legend=True);
df.set_index("Grid Points")["PM_INST_CMPL (min)"].plot(ax=ax2, legend=True);
```

Although some slight variations can be seen for run cycles for many grid points, the correlation looks quite linear (as one would naively expect). Let's test that by fitting a linear function!

*The details of the fitting have been extracted into dedicated function, `print_and_return_fit()`, of the `common.py` helper file. If you're interested, go have a look at it.*

```
In [ ]: def linear_function(x, a, b):
        return a*x+b

In [ ]: fit_parameters, fit_covariance = common.print_and_return_fit(
        ["PM_RUN_CYC (min)", "PM_INST_CMPL (min)"],
        df.set_index("Grid Points"),
        linear_function,
        format_uncertainty=".4f"
    )
```

Let's overlay our fits to the graphs from before.

```
In [ ]: fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
for ax, pmu_counter in zip([ax1, ax2], ["PM_RUN_CYC (min)", "PM_INST_CMPL (min)"]):
    df.set_index("Grid Points")[pmu_counter].plot(ax=ax, legend=True);
    ax.plot(
        df["Grid Points"],
        linear_function(df["Grid Points"], *fit_parameters[pmu_counter]),
        linestyle="--",
        label="Fit: {:.2f} * x + {:.2f}".format(*fit_parameters[pmu_counter])
    )
ax.legend();
```

Please execute the next cell to summarize the first task.

```
In [ ]: print("The algorithm under investigation runs about {:.0f} cycles and executes about
        {:.0f} instructions per grid point".format(
            *[fit_parameters[pmu_counter][0] for pmu_counter in ["PM_RUN_CYC (min)",
            "PM_INST_CMPL (min)"]])
    )
```

### Bonus:

The linear fits also calculate a y intersection (»b«). How do you interpret this value?

We are revisiting the graph in a little while.

Section ??

## 1.3 Task 2: Measuring Loads and Stores

Looking at the source code, how many loads and stores from / to memory do you expect? Have a look at the loop which we instrumented.

Let's compare your estimate to what the system actually does!

### 1.3.1 Task A

Please measure counters for loads and stores. See the TODOs in `poisson2d.ld_st.c`. This time, implement `PM_LD_CMPL` and `PM_ST_CMPL`.

Compile with `make task2`, test your program with a single run with `make run_task2`, and then finally submit a benchmarking run to the batch system with `make bench_task2`. The following cell will take care of all this.

Section ??

```
In [ ]: !make bench_task2
```

Once the run finished, let's plot it again in the course of the following cells (non-interactive: `make graph_task2a`).

```
In [ ]: df_ldst = pd.read_csv("poisson2d.ld_st.bin.csv", skiprows=range(2, 50000, 2))
        df_ldst["Grid Points"] = df_ldst["nx"] * df_ldst["ny"]
        df_ldst.head()

In [ ]: fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
        df_ldst.set_index("Grid Points")["PM_LD_CMPL (min)"].plot(ax=ax1, legend=True);
        df_ldst.set_index("Grid Points")["PM_ST_CMPL (min)"].plot(ax=ax2, legend=True);
```

Also this behaviour looks – at a first glance – linear. We can again fit a first-order polynom (and re-use our previously defined function `curve_fit`)!

```
In [ ]: _fit, _cov = common.print_and_return_fit(
        ["PM_LD_CMPL (min)", "PM_ST_CMPL (min)"],
        df_ldst.set_index("Grid Points"),
        linear_function,
        format_value=".4f"
    )
    fit_parameters = {**fit_parameters, **_fit}
    fit_covariance = {**fit_covariance, **_cov}
```

Let's overlay this in one common plot:

```
In [ ]: fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
        for ax, pmu_counter in zip([ax1, ax2], ["PM_LD_CMPL (min)", "PM_ST_CMPL (min)"]):
            df_ldst.set_index("Grid Points")[pmu_counter].plot(ax=ax, legend=True);
            ax.plot(
                df_ldst["Grid Points"],
                linear_function(df["Grid Points"], *fit_parameters[pmu_counter]),
                linestyle="--",
                label="Fit: {:.2f} * x + {:.2f}".format(*fit_parameters[pmu_counter])
            )
        ax.legend();
```

Did you expect more?

The reason is simple: Among the load and store instructions counted by `PM_LD_CMPL` and `PM_ST_CMPL` are vector instructions which can load and store multiple (in this case: two) values at a time. To see how many *bytes* are loaded and stored, we need to measure counters for vectorized loads and stores as well.

### 1.3.2 TASK B

Please measure counters for *vectorized* loads and *vectorized* stores. See the TODOs in `poisson2d.vld.c` and `poisson2d.vst.c` (Note: These vector counters can not be measured together and need separate files and runs). Can you find out the name of the counters yourself, using `papi_native_avail | grep VECTOR_?`

Compile, test, and bench-run your program again.

Section ??

```
In [ ]: !papi_native_avail | grep VECTOR_
```

make `bench_task3` will submit benchmark runs of both vectorized counters to the batch system (as two subsequent runs of the individual files).

```
In [ ]: !make bench_task3
```

Let's plot it again, as soon as the run finishes! Non-interactively, call `graph_task2b`.

Because we couldn't measure the two vector counters at the same time, we have two CSV files to read in now. We combine them into one common dataframe `df_vldvst` in the following.

```
In [ ]: df_vld = pd.read_csv("poisson2d.vld.bin.csv", skiprows=range(2, 50000, 2))
df_vst = pd.read_csv("poisson2d.vst.bin.csv", skiprows=range(2, 50000, 2))
df_vldvst = pd.concat([df_vld.set_index("nx"),
df_vst.set_index("nx")][['PM_VECTOR_ST_CMPL (total)', 'PM_VECTOR_ST_CMPL (min)', '
PM_VECTOR_ST_CMPL (max)']], axis=1).reset_index()

In [ ]: df_vldvst["Grid Points"] = df_vldvst["nx"] * df_vldvst["ny"]
df_vldvst.head()

In [ ]: fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
df_vldvst.set_index("Grid Points")["PM_VECTOR_LD_CMPL (min)"].plot(ax=ax1, legend=True);
df_vldvst.set_index("Grid Points")["PM_VECTOR_ST_CMPL (min)"].plot(ax=ax2, legend=True);
```

Also here seems to be a linear correlation. Let's do our fitting and plot directly.

```
In [ ]: _fit, _cov = common.print_and_return_fit(
    ["PM_VECTOR_LD_CMPL (min)", "PM_VECTOR_ST_CMPL (min)"],
    df_vldvst.set_index("Grid Points"),
    linear_function,
    format_value=".4f",
)
fit_parameters = {**fit_parameters, **_fit}
fit_covariance = {**fit_covariance, **_cov}

In [ ]: fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
for ax, pmu_counter in zip([ax1, ax2], ["PM_VECTOR_LD_CMPL (min)", "PM_VECTOR_ST_CMPL
(min)"]):
    df_vldvst.set_index("Grid Points")[pmu_counter].plot(ax=ax, legend=True);
    ax.plot(
        df_vldvst["Grid Points"],
        linear_function(df["Grid Points"], *fit_parameters[pmu_counter]),
        linestyle="--",
        label="Fit: {:.2f} * x + {:.2f}".format(*fit_parameters[pmu_counter])
    )
    ax.legend();
```

Let's try to make sense of those numbers.

Vector loads and vector stores use two 8 Byte values at a time. When we measured loads and stores with LD\_CMPL and ST\_CMPL in part A of this task, we measured total number of stores and loads; that is: vector and scalar versions of the instructions. In order to convert the load and store instructions into **bytes** loaded and stored, we need to separate them. The difference of total

instructions and vector instructions yield scalar instructions. We multiply the scalar instructions by 8 Byte (double precision) and the vector instructions by 16 Byte (two loads or stores of double precision). That yields the loaded or stored data (or, more precisely, the instruction-equivalent data).

To formalize it, see the following equations, as an example for load ( $ld$ ), with  $b$  denoting data loaded in bytes and  $n$  denoting the number of instructions.

$$b_{ld} = b_{ld}^{\text{scalar}} + b_{ld}^{\text{vector}} \quad (1)$$

$$b_{ld}^{\text{scalar}} = n_{ld}^{\text{scalar}} * 8 \text{ Byte} \quad (2)$$

$$b_{ld}^{\text{vector}} = n_{ld}^{\text{vector}} * 16 \text{ Byte} \quad (3)$$

$$n_{ld}^{\text{scalar}} = n_{ld}^{\text{total}} - n_{ld}^{\text{vector}} \quad (4)$$

$$\Rightarrow b_{ld} = n_{ld}^{\text{scalar}} * 8 \text{ Byte} + n_{ld}^{\text{vector}} * 16 \text{ Byte} \quad (5)$$

$$= (n_{ld}^{\text{scalar}} + 2n_{ld}^{\text{vector}}) * 8 \text{ Byte} \quad (6)$$

$$= (n_{ld}^{\text{total}} - n_{ld}^{\text{vector}} + 2n_{ld}^{\text{vector}}) * 8 \text{ Byte} \quad (7)$$

$$= (n_{ld}^{\text{total}} + n_{ld}^{\text{vector}}) * 8 \text{ Byte} \quad (8)$$

We are going to print this in the next cell. In case you look at this Notebook non-interactively, call `graph_task2b-2`.

```
In [ ]: df_byte = pd.DataFrame()
df_byte["Loads"] = (df_vldvst.set_index("Grid Points")["PM_VECTOR_LD_CMPL (min)"] +
df_ldst.set_index("Grid Points")["PM_LD_CMPL (min)"])*8
df_byte["Stores"] = (df_vldvst.set_index("Grid Points")["PM_VECTOR_ST_CMPL (min)"] +
df_ldst.set_index("Grid Points")["PM_ST_CMPL (min)"])*8
ax = df_byte.plot()
ax.set_ylabel("Bytes");
```

Let's quantify the difference by, again, fitting a linear function to the data.

```
In [ ]: _fit, _cov = common.print_and_return_fit(
    ["Loads", "Stores"],
    df_byte,
    linear_function
)
fit_parameters = {**fit_parameters, **_fit}
fit_covariance = {**fit_covariance, **_cov}
```

Analogously to the proportionality factors, this much is loaded/stored per grid point.

*Not really a TASK C:* We can combine this information with the cycles measured in Task 1 to create a bandwidth of exchanged bytes per cycle.

```
In [ ]: df_bandwidth = pd.DataFrame()
df_bandwidth["Bandwidth / Byte/Cycle"] = (df_byte["Loads"] + df_byte["Stores"]) /
df.set_index("Grid Points")["PM_RUN_CYC (min)"]
```

Let's display it as a function of grid points. And also compare it to the available L1 cache bandwidth in a second (sub-)plot. Non-interactive users, call `make_graph_task2c`.

```
In [ ]: fig, (ax1, ax2) = plt.subplots(nrows=2, sharex=True)
for ax in [ax1, ax2]:
    df_bandwidth["Bandwidth / Byte/Cycle"].plot(ax=ax, legend=True, label="Jacobi
Bandwidth")
    ax.set_ylabel("Byte/Cycle")
```

```
ax2.axhline(2*16, color=sns.color_palette()[1], label="L1 Bandwidth");
ax2.legend();
```

As you can see, we are quite a bit away from the available L1 cache bandwidth. Can you think of reasons why?

## 1.4 Task E1: Measuring FLOps

If you still have time, feel free to work on the following extended task.

**TASK:** Please measure counters for *vectorized* floating point operations and *scalar* floating point operations. The two counters can also not be measured during the same run. So please see the TODOs in [poisson2d.sflops.c](#) and [poisson2d.vflops.c](#). By now you should be able to find out the names of the counters by yourself (*Hint: they include the words »scalar« and »vector«*...).

As usual, compile, test, and bench-run your program.

Section ??

```
In [ ]: !make bench_task4

In [ ]: df_sflop = pd.read_csv("poisson2d.sflop.bin.csv", skiprows=range(2, 50000, 2))
df_vflop = pd.read_csv("poisson2d.vflop.bin.csv", skiprows=range(2, 50000, 2))
df_flop = pd.concat([df_sflop.set_index("nx"),
df_vflop.set_index("nx")][['PM_VECTOR_FLOP_CMPL (total)', 'PM_VECTOR_FLOP_CMPL (min)', '
PM_VECTOR_FLOP_CMPL (max)']], axis=1).reset_index()
df_flop.head()
```

Again, the name of the vector counter is a bit misleading; not floating point operations are measured but floating point instructions. To get *real* floating point operations, each value needs to be multiplied by the vector width (2). We can plot the values afterwards (non-interactive: `make graph_task4`).

```
In [ ]: df_flop["Grid Points"] = df_flop["nx"] * df_flop["ny"]
df_flop["Vector FLOps (min)"] = df_flop["PM_VECTOR_FLOP_CMPL (min)"] * 2
df_flop["Scalar FLOps (min)"] = df_flop["PM_SCALAR_FLOP_CMPL (min)"]

In [ ]: df_flop.set_index("Grid Points")[["Scalar FLOps (min)", "Vector FLOps (min)"]].plot();

In [ ]: _fit, _cov = common.print_and_return_fit(
    ["Scalar FLOps (min)", "Vector FLOps (min)"],
    df_flop.set_index("Grid Points"),
    linear_function
)
fit_parameters = {**fit_parameters, **_fit}
fit_covariance = {**fit_covariance, **_cov}
```

With that measured, we can determine the Arithmetic Intensity; the balance of floating point operations to bytes transmitted:

$$AI^{\text{emp}} = I_{\text{flop}}/I_{\text{mem}}, \quad (9)$$

with  $I$  denoting the respective amount. This is the empirically determined Arithmetic Intensity.

In the non-interactive version of the Notebook, please plot the graph calling `make graph_task4-2` in the terminal.

```
In [ ]: I_flop_scalar = df_flop.set_index("Grid Points")["Scalar FLOps (min)"]
I_flop_vector = df_flop.set_index("Grid Points")["Vector FLOps (min)"]
```

```

I_mem_load    = df_byte["Loads"]
I_mem_store   = df_byte["Stores"]

In [ ]: df_ai = pd.DataFrame()
df_ai["Arithmetic Intensity"] = (I_flop_scalar + I_flop_vector) / (I_mem_load +
I_mem_store)
ax = df_ai.plot();
ax.set_ylabel("Byte/Flop");

```

Thinking back to the first lecture of the tutorial, what Arithmetic Intensity did you expect?

## 1.5 Task E2: Measuring a Larger Range

If you still still have time, you might venture into your own benchmarking adventure.

Maybe you noticed already, for instance in Task 2 C: At the very right to very large numbers of grid points, the behaviour of the graph changed. Something is happening there!

**TASK:** Revisit the counters measured above for a larger range of `nx`. Right now, we only studied `nx` until 1000. New effects appear above that value – partly only well above, though ( $nx > 15000$ ).

You're on your own here. Edit the `bench.sh` script to change the range and the stepping increments.

**Good luck!**

Section ??