

# HandsOnPerformanceOptimization-task

November 11, 2019

## 1 Hands-On Performance Optimization

*Supercomputing 2019 Tutorial “Application Porting and Optimization on GPU-Accelerated POWER Architectures”, November 18th 2019*

---

As for the first task of this tutorial, also this task is primarily designed to be executed as an interactive Jupyter Notebook. However, everything can also be done using an SSH connection to Ascent (or any other POWER9 computer) in your terminal.

### 1.1 Jupyter notebook execution

When using Jupyter, this Notebook will guide you through the steps. Note that if you execute a cell multiple times while optimizng the code the output will be replaced. You can however duplicate the cell you want to execute and keep its output. Check the *edit* menu above.

You will always find links to a file browser of the corresponding task subdirectory as well as direct links to the source files you will need to edit as well as the profiling output you need to open locally.

If you want you also can get a terminal in your browser; just open it via the »New Launcher« button (+).

### 1.2 Terminal fallback

The tasks are place in directories named `Task[1-3]`.

Makefile targets are created to cover everything, from compile, to run and profile. Please take a look at the cells containing the make calls as a guide also for the non-interactive version of this description.

### 1.3 Setup

We are using some very fresh compiler features and use GCC 9.2.0 because of that. It should already be in your environment. Let's check!

```
In [ ]: !gcc --version
```

### 1.4 Tasks

This session comes with multiple tasks, each one to be found in the respective sub-directory `Task[1-3]`. In each of these directories you will also find Makefiles that are set up so that you can compile and submit all necessary tasks.

Please choose from the task below.

- Section ??: **Basic compiler optimization flags and compiler annotations**

Improve performance of the CPU Jacobi solver with compiler flags such as `Ofast` and profile-directed feedback. Learn about compiler annotations.

- Section ??: **Optimization via Prefetching controlled by compiler**

Improve performance of the CPU Jacobi solver with software prefetching. Some compilers such as IBM XL define flags that can be used to modify the aggressiveness of the hardware prefetcher. Learn to modify the DSCR value through XL and study the impact on application performance. \*

#### Section ??: **Optimization via OpenMP controlled by compiler and the system**

Parallelize the CPU Jacobi solver and determine the right binding to be used for optimal performance.

- Section 2 Please remember to take the survey !

### 1.4.1 Make Targets

For all tasks we have defined the following make targets.

- **poisson2d:**  
build `poisson2d` binary (default)
- **run:**  
run `poisson2d` with default parameters

Section ??

---

## 1.5 Task 1: Basic compiler optimization flags and compiler annotations

### 1.5.1 Overview

The goal of this task is to understand different options available to optimize the performance of the CPU Jacobi solver

Your task is to:

- Optimize performance with `-Ofast` flag
- Verify the cause for performance improvement by viewing perf profiles of `O3` and `Ofast` binaries
- Optimize performance with profile directed feedback
- Generate compiler annotations/remarks to understand the optimizations done by the compiler with and without profile directed feedback

First, change the working directory to `Task1`.

```
In [ ]: %cd Task1
```

### 1.5.2 Part A: `-Ofast` vs. `-O3`

We are to compare the performance of the binary being compiled with `-Ofast` optimization and with `-O3` optimization. As in the previous task, we use a `Makefile` for compilation. The `Makefile` targets `poisson2d_O3` and `poisson2d_Ofast` are already prepared.

**TASK:** Add `-O3` as the optimization flag for the `poisson2d_O3` target by using the corresponding `CFLAGS` definition. There are notes relating to this Task 1 in the header of the `Makefile`. Compile the code using `make` as indicated below and run with the `Make` targets `run`, `run_perf` and `run_perf_recrep`.

```
In [ ]: !make poisson2d_O3
```

```
In [ ]: !make run
```

Let's have a look at the output of the `Makefile` target `run_perf`. It invokes the GNU `perf` tool to print out details of the number of instructions executed and the number of cycles taken by POWER9 to execute the program. Feel free to add further counter to this call to `perf`.

```
In [ ]: !make run_perf
```

Next we run the `makefile` with target `run_perf_recrep` that prints the top routines of the application in terms of hotness by using a combination of `perf record ./app` and `perf report`.

```
In [ ]: # run_perf_recrep displays the top hot routines
        !make run_perf_recrep
```

**TASK:** Now add the optimization flag `Ofast` to the `CFLAGS` for target `poisson2d_Ofast`. Compile the program with the target `poisson2d_Ofast` and run and analyse it as before with `run`, `run_perf` and `run_perf_recrep`.

What difference do you see?

```
In [ ]: !make poisson2d_Ofast
        !make run
```

Again, run a `perf`-instrumented version:

```
In [ ]: !make run_perf
```

Generate the list of top routines in terms of hotness:

```
In [ ]: !make run_perf_recrep
```

If `perf` is unavailable to you on other machines, you can also study the disassembly with `objdump`: `objdump -lSd ./poisson2d > poisson2d.dis` (feel free to experiment with this in the Notebook as well, just prefix the command with a `!` to execute it.)

**Interpretation** Depending on the application requirement, if a high precision of results is not mandatory, one can compile an application with `-Ofast` which enables `-ffast-math` option that implements the same math function in a relaxed manner very similar to how general mathematical expressions are implemented and avoids the overhead of calling a function from the math library. Comparing the files, you will see that the `-Ofast` binary natively implements the `fmax` function using instructions available in the hardware. The `-O3` binary makes a library call to compute `fmax` to follow a stricter *IEEE* requirement for accuracy.

### 1.5.3 Part B: Profile-directed Feedback

For the first level of optimization we see that `Ofast` cut the execution time of the `O3` binary by almost half.

We can optimize the performance further by using profile-directed feedback optimization.

To compile using profile-directed feedback with the GCC compiler we need to build the application in three stages:

1. Instrument binary;
2. Run binary with training, gather profile information;
3. Use profile information to generate optimized binary.

Step 1 is achieved by compiling the binary with the correct flag – `-fprofile-generate`. In our case, we need to specify an output location, which should be `$(SC19_DIR_SCRATCH)`.

Step 2 consists of a usual, albeit shorter run of the instrumented binary. The can be very short, though the parameters need to be representative of the actual run. After the binary ran, an output file (with file extension `.gcda`) is written to the directory specified during compilation.

For Step 3, the binary is once again compiled, but this time using the `gcda` profile just generated. The according flag is `-fprofile-use`, which we set to `$(SC19_DIR_SCRATCH)` as well.

In our `Makefile` at hand, we prepared the steps already for you in the form of two targets.

- `poisson2d_train`: Will compile the binary with profile-directed feedback
- `poisson2d_ref`: Will take a generated profile and compile a new, optimized binary

By using dependencies, between these two targets a profile run is launched.

**TASK:** Edit the `Makefile` and add the `-fprofile-*` flags to the `CFLAGS` of `poisson2d_train` and `poisson2d_ref` as outline in the file.

After that, you may launch them with the following cells (`gen_profile` is a meta-target and uses `poisson2d_train` and `poisson2d_ref`). If you need to clean the generated profile, you may use `make clean_profile`.

```
In [ ]: !make gen_profile
```

If the previous cell executed correctly, you now have your optimized executable. Let's see if it even fast than before!

```
In [ ]: !make run
```

Let's also measure instructions and cycles

```
In [ ]: !make run_perf
```

What is your speed-up? Feel free to run with larger problem sizes (mesh; iterations)

### 1.5.4 Part C: Compiler annotations/Remarks

Usually, all compilers provide an option to emit annotations or remarks by the compiler. These remarks summarize the optimizations done in detail, the location in source where these optimizations were done. There exist options that also indicate optimizations that were missed and the reason why they could not be done.

To generate compiler annotations using GCC, one uses `-fopt-info-all`. If you only want to see the missed options, use the option `-fopt-info-missed` instead of `-fopt-info-all`. See also the [documentation of GCC regarding the flag](#).

**TASK:** Have a look at the `CFLAGS` of the `Makefile` target `poisson2d_0fast_info`. Add the flag `-fopt-info-all` to the list of flags. This will print optimisation information to stdout. If you rather want to print to this information to a file, use – for example – `-fopt-info-all=$(SC19_DIR_SCRATCH)/filename`.

```
In [ ]: !make poisson2d_0fast_info
```

Let's compare this with the output during compilation when using profile-directed feedback from Task 1 B.

**TASK:** Adapt the CFLAGS of `poisson2d_ref_info` to include `-fopt-info-all` **and** the profile input of `-fprofile-use=...` here. (*Be advised: Long output!*)

```
In [ ]: !make poisson2d_ref_info
```

Comparing the annotations generated of a plain `-Ofast` optimization level and the one generated at `-Ofast` and profile directed feedback, we observe that many more optimizations are possible due to profile information.

For instance you will see annotations such as

```
poisson2d.c:114:25: optimized: loop unrolled 3 times (header execution count 436550)
```

The execution count indicates the dynamic execution count of the node at runtime. This information determines which paths are hotter and subsequently facilitate additional optimizations.

## References

1. <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>
2. <https://perf.wiki.kernel.org/index.php/Tutorial>

Section ??

---

## 1.6 Task 2: Impact of Prefetching on Performance

### 1.6.1 Overview

- Study the difference of program execution time of different optimization levels with and without software prefetching.
- Verify the impact by measuring cache counters with and without prefetching.
- Learn how to modify contents of DSCR (*Data Stream Control Register*) using IBM XL compiler and study the impact with different values to DSCR.

But first, let's change directory to that of Task 2

```
In [ ]: %cd ../Task2
```

### 1.6.2 Part A: Software Prefetching

**TASK:** Look at the Makefile and work on the TODOs.

- First generate a `-Ofast`-optimised binary and note down the performance in terms of cycles, seconds, and L3 misses. This is our baseline!
- Modify the `Makefile` to add the option for software prefetching (`-fprefetch-loop-arrays`). Compare performance of `-Ofast` with and without software prefetching

```
In [ ]: !make clean
```

```
In [ ]: !make poisson2d CC=gcc
        !make run
        !make l3missstats
```

```
In [ ]: !make poisson2d_pref CC=gcc
        !make run
        !make l3missstats
```

**TASK:** Repeat the experiment with the `-O3` flag. Have a look at the `Makefile` and the outlined `TODO`. There's a position to easily adapt `-Ofast`→`-O3`!

```
In [ ]: !make poisson2d CC=gcc -B
        !make run
        !make l3missstats
```

```
In [ ]: !make poisson2d_pref CC=gcc -B
        !make run
        !make l3missstats
```

Do you notice the impact difference with optimization levels? At what optimization level does software prefetching help the most?

### 1.6.3 Part B: Analysis of Instructions

Compilation of the `-Ofast` binary with the software prefetching flag causes the compiler to generate the `dcb*` instructions that prefetch memory values to L3.

**TASK:** Run `$(SC19_SUBMIT_CMD) objdump -lSd` on each binary file (`-O3`, `-Ofast` with prefetch/no prefetch). Look for instructions beginning with `dcb`. At what optimization levels does the compiler generate software prefetching instructions?

```
In [ ]: !make CC=gcc -B poisson2d_pref
        !objdump -lSd ./poisson2d_pref > poisson2d.dis
In [ ]: !grep dcb poisson2d.dis
```

### 1.6.4 Part C: Changing Values of DSCR via compiler flags

This task requires using the IBM XL compiler. It should be already in your environment.

We saw the impact of software prefetching in the previous subsection. In certain cases, tuning the hardware prefetcher through compiler options can also help improve performance. In this exercise we shall see some compiler options that can be used to modify the DSCR value which controls aggressiveness of prefetching. It can be also used to turn off hardware prefetching.

IBM XL compiler has an option `-qprefetch=dscr=<val>` that can be used for this purpose. Compiling with `-qprefetch=dscr=1` turns off the prefetcher. One can give various values such as `-qprefetch=dscr=4`, `-qprefetch=dscr=7` etc. to control aggressiveness of prefetching.

For this exercise we use `make CC=xlc_r` to illustrate the performance impact.

**Task** Generate a XL-compiled binary by compiling using the following cells. After you've generated a baseline, start editing the `Makefile`: Add `qprefetch=dscr=1` to the `CFLAGS` and rebuild the application and note the performance. Which one is faster?

In general, applications benefit with the default settings of hardware DSCR register (`-qprefetch=dscr=0`). However, certain applications also benefit with prefetching turned off.

It is to be noted that DSCR values are highly sensitive to the application. One value that works well for Application A may not help Application B.

Measure performance of the application compiled with XL at default DSCR value

```
In [ ]: !make CC=xlc_r -B poisson2d
        !make run
```

Measure performance of the application compiled with XL with DSCR value turned off

```
In [ ]: !make poisson2d_dscr CC=xlc_r -B
        !make run
```

Does Hardware prefetcher help this application? How much impact do you see when you turn off the hardware prefetcher?

## References

1. <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>
2. <https://www.gnu.org/software/gcc/projects/prefetch.html>
3. [https://openpowerfoundation.org/?resource\\_lib=power-isa-version-3-0](https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0)

Section ??

---

## 1.7 Task 3: OpenMP

### 1.7.1 Overview

We add OpenMP shared-memory parallelism to the application. Also, we study the effect of binding the multi-thread processes to certain cores on the resulting application performance. We do this study for both GCC and XL compilers in order to learn about the appropriate options that need to be used. First, we need to change directory to that of Task3. For Task 3 we modify `poisson2d.c` to invoke an exact copy of the main jacobi loop which is `poisson2d_reference`. We parallelize only the main loop but not `poisson2d_reference`. The speedup is the performance gain seen in the main loop as compared to the reference loop.

```
In [ ]: %cd ../Task3
```

### 1.7.2 Part A: Implement OpenMP Pragas; Compilation

**Task:** Please add the correct OpenMP directives to `poisson2d.c` and compilation flags in the Makefile to enable OpenMP with GCC and XL compilers.

- **Directives:** Look at the TODOs in `poisson2d.c` to add OpenMP parallelism. The pragmas in question are `#pragma omp parallel for` (and once it's `#pragma omp parallel for reduction(max:error)` – can you guess where?)
- **Compilation:** Please add compilation flags enabling OpenMP in GCC and XL to the Makefile. For GCC, we need to add `-fopenmp` and the application needs to be linked with `-lgomp`. For XL, we need to add `-qsmp=omp` to the list of compilation flags.

Afterwards, compile and run the application with the following commands.

```
In [ ]: !make poisson2d CC=gcc
```

The command to submit a job to the batch system is prepared in an environment variable `$SC19_SUBMIT_CMD`; use it together with `eval`. In the following cell, it is shown how to invoke the application using the batch system.

```
In [ ]: !eval $SC19_SUBMIT_CMD ./poisson2d 1000 1000 1000
```

In order to run the parallel application, we need to set the number of threads using `OMP_NUM_THREADS`. What is the best performance you can reach by setting the number of threads via `OMP_NUM_THREADS=N` with `N` being the number of threads? Feel free to play around with the command in the following cell, using 1 thread as an example.

We added `--bind none` to prevent `jsrun`, the scheduler of Ascent, from overlaying binding options. Also, we use `-c ALL_CPUS` to make all CPUs on the compute nodes available to you.

```
In [ ]: !eval OMP_NUM_THREADS=N $SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000
1000
```

### 1.7.3 Part B: Bindings

Different CPU architectures and models come with different configuration of cores. The configuration plays an important role in the run time of the application. We need to optimize for it!

There are applications which can be used to determine the configuration of the processor. Among those are:

- `lscpu`: Can be used to determine the number of sockets, number of cores, and number of threads. It gives a very good overview and is available on most Linux systems.
- `ppc64_cpu --smt`: Specifically for POWER, this tool can give information about the number of simulation threads running per core (*SMT*, Simultaneous Multi-Threading).

Run `ppc64_cpu --smt` to find out about the threading configuration of Ascent!

```
In [ ]: !eval $SC19_SUBMIT_CMD ppc64_cpu --smt
```

There are more sources of information available

- `/proc/cpuinfo`: Holds information about virtual cores, including model and clock speed. Available on most Linux system. Usually used together with `cat`
- `/sys/devices/system/cpu/cpu0/topology/thread_siblings_list`: Holds information about thread siblings for given CPU core (`cpu0` in this case). Use it to find out which thread is mapped to which core.

```
In [ ]: !$$$SC19_SUBMIT_CMD cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
!$$$SC19_SUBMIT_CMD cat /sys/devices/system/cpu/cpu5/topology/thread_siblings_list
```

There are various environment variables available within OpenMP (some specific to GCC) that hold across compilers to specify binding of threads to cores. See, for instance, the [OMP\\_PLACES environment Variable](#). We also have a GNU specific variable which can also be used to control affinity - `GOMP_CPU_AFFINITY`. Setting `GOMP_CPU_AFFINITY` is specific to GCC binaries but it internally serves the same function as setting `OMP_PLACES`.

**Task:** Run the application enabled with OpenMP from Part A with different binding configurations. Make sure to at least run a) binding all threads to a single core and b) binding threads to different cores.

Adapt the following command with your configuration – or follow along accordingly in the non-interactive version of the Notebook.

What's your maximum speedup?

```
In [ ]: !eval OMP_DISPLAY_ENV=true OMP_PLACES="{X},{Y},{Z},{A}" OMP_NUM_THREADS=4
$$$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 100 | grep
"OMP_PLACES\|speedup"
```



```
In [ ]: !eval OMP_DISPLAY_ENV=true GOMP_CPU_AFFINITY="X,Y,Z,A" OMP_NUM_THREADS=4
        $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 100 | grep
        "OMP_PLACES\|speedup"
```

Great!

If you still have time: The same experiments can be repeated with the IBM XL compiler. The corresponding compiler flag to enable OpenMP parallelism that needs to be used for XL is `-qsmp=omp`

**Task:** In the Makefile add the OpenMP flag and generate XL binaries with OpenMP and run the application with various number of threads and note the performance speedup.

```
In [ ]: !make CC=xlc_r -B run
```

Run the parallel application with varying number of threads (`OMP_NUM_THREADS`) and note the performance improvement.

```
In [ ]: !eval OMP_NUM_THREADS=N $SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000
        1000
```

Now we repeat the exercise of using the right binding of threads for the XL binary. `OMP_PLACES` pertains to the XL binary as well as it is an OpenMP variable. `GOMP_CPU_AFFINITY` is specific to GCC binary so that cannot be used to set the binding.

**Task:** Run the application enabled with OpenMP from Part A with different binding configurations. Make sure to at least run a) binding all threads to a single core and b) binding threads to different cores.

Adapt the following command with your configuration – or follow along accordingly in the non-interactive version of the Notebook.

We are mixing Python with Bash (!) here, so don't get confused (because of this, if we want to use Bash environment variables, we need to use two `$$`)

What's your maximum speedup?

```
In [ ]: for affinity in [{"X},{Y},{Z},{A}", "{P},{Q},{R},{S}"]:
        print("Affinity: {}".format(affinity))
        !eval OMP_DISPLAY_ENV=true OMP_PLACES=$affinity OMP_NUM_THREADS=4 $$SC19_SUBMIT_CMD
        -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000 | grep "OMP_PLACES\|speedup"
```

Likewise we see a higher speedup when we bind the threads to different cores rather than to a single core. This hands-on illustrates that apart from compiler level tuning, system level tuning is also equally important to obtain performance improvements

## References

1. [https://gcc.gnu.org/onlinedocs/libgomp/GOMP\\_005fCPU\\_005fAFFINITY.html](https://gcc.gnu.org/onlinedocs/libgomp/GOMP_005fCPU_005fAFFINITY.html)
2. <https://www.openmp.org/spec-html/5.0/openmpse53.html>

Section ??

## 2 Survey

Please remember to take some time and fill out the [survey](#).