

# HandsOnPerformanceOptimization-solution

November 8, 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 [1]: !gcc --version
```

```
gcc (GCC) 9.2.0
```

```
Copyright (C) 2019 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

## 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 [4]: `%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 [84]: !make poisson2d_O3
```

```
gcc -c -std=c99 -mcpu=power9 -DUSE_DOUBLE -mvsx -maltivec -O3  poisson2d_reference.c
-o poisson2d_reference.o -lm
gcc -std=c99 -mcpu=power9 -DUSE_DOUBLE -mvsx -maltivec -O3  poisson2d.c
poisson2d_reference.o -o poisson2d -lm
```

```
In [73]: !make run
```

```
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS  time ./poisson2d
Job <24897> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291
4.73user 0.00system 0:04.73elapsed 99%CPU (0avgtext+0avgdata 24256maxresident)k
256inputs+0outputs (0major+480minor)pagefaults 0swaps
```

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 [74]: !make run_perf
```

```
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf stat -e
cycles,instructions ./poisson2d
Job <24898> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
```

```

200, 0.248007
300, 0.247025
400, 0.246050
500, 0.245084
600, 0.244124
700, 0.243173
800, 0.242228
900, 0.241291

```

Performance counter stats for './poisson2d':

```

16264721613      cycles:u
28463907825      instructions:u      #    1.75  insn per cycle

4.738444892 seconds time elapsed

```

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

```

```

bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf record -e cycles
--output=/gpfs/wolf/trn003/scratch/aherten//cycles.data ./poisson2d
Job <24899> is submitted to default queue <batch>.

```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
```

```
Calculate current execution.
```

```

0, 0.249995
100, 0.248997
200, 0.248007
300, 0.247025
400, 0.246050
500, 0.245084
600, 0.244124
700, 0.243173
800, 0.242228
900, 0.241291

```

```
[ perf record: Woken up 3 times to write data ]
```

```
[ perf record: Captured and wrote 0.739 MB
```

```
/gpfs/wolf/trn003/scratch/aherten//cycles.data (19102 samples) ]
```

```
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf report -i
```

```
/gpfs/wolf/trn003/scratch/aherten//cycles.data --stdio
```

```
Job <24900> is submitted to default queue <batch>.
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
# To display the perf.data header info, please use --header/--header-only options.
```

```
#
```

```
#
```

```
# Total Lost Samples: 0
```

```
#
```

```
# Samples: 19K of event 'cycles:u'
```

```
# Event count (approx.): 16254596654
```

```
#
```

```
# Overhead Command Shared Object Symbol
```

```
# ... ..
```

```
#
```

```

65.50% poisson2d poisson2d      [.] 00000038.plt_call.fmax@@GLIBC_2.17
21.21% poisson2d poisson2d      [.] main
 9.18% poisson2d libc-2.17.so    [.] __memcpy_power7
 3.28% poisson2d libm-2.17.so    [.] __fmaxf
 0.74% poisson2d libm-2.17.so    [.] __exp_finite
 0.04% poisson2d poisson2d      [.] 00000038.plt_call.memcpy@@GLIBC_2.17
 0.01% poisson2d libm-2.17.so    [.] __GI___exp
 0.01% poisson2d ld-2.17.so      [.] check_match.10253
 0.01% poisson2d ld-2.17.so      [.] do_lookup_x
 0.00% poisson2d ld-2.17.so      [.] _dl_lookup_symbol_x
 0.00% poisson2d ld-2.17.so      [.] _dl_relocate_object
 0.00% poisson2d ld-2.17.so      [.] strcmp
 0.00% poisson2d ld-2.17.so      [.] _wordcopy_fwd_aligned
 0.00% poisson2d ld-2.17.so      [.] _dl_sysdep_start
 0.00% poisson2d ld-2.17.so      [.] _start

```

```

#
# (Tip: Limit to show entries above 5% only: perf report --percent-limit 5)
#

```

**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 [76]: !make poisson2d_ofast
         !make run

gcc -std=c99 -mcpu=power9 -DUSE_DOUBLE -mvsx -maltivec -Ofast  poisson2d.c
poisson2d_reference.o -o poisson2d -lm
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS  time ./poisson2d
Job <24901> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291
2.41user 0.00system 0:02.41elapsed 99%CPU (0avgtext+0avgdata 24256maxresident)k
256inputs+0outputs (0major+480minor)pagefaults 0swaps

```

Again, run a `perf`-instrumented version:

```

In [77]: !make run_perf

bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf stat -e
cycles,instructions ./poisson2d
Job <24902> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>

```

```

<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291

Performance counter stats for './poisson2d':

      8258991976      cycles:u
    12013091172      instructions:u      #    1.45  insn per cycle

    2.408703909 seconds time elapsed

```

Generate the list of top routines in terms of hotness:

```

In [78]: !make run_perf_recrep

bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf record -e cycles
--output=/gpfs/wolf/trn003/scratch/aherten//cycles.data ./poisson2d
Job <24903> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.382 MB
/gpfs/wolf/trn003/scratch/aherten//cycles.data (9728 samples) ]
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf report -i
/gpfs/wolf/trn003/scratch/aherten//cycles.data --stdio
Job <24904> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 9K of event 'cycles:u'
# Event count (approx.): 8268811890
#

```

```

# Overhead  Command      Shared Object  Symbol
# ... ..
#
81.12% poisson2d poisson2d      [.] main
17.97% poisson2d libc-2.17.so  [.] __memcpy_power7
0.79%  poisson2d libm-2.17.so  [.] __exp_finite
0.04%  poisson2d poisson2d      [.] 00000038.plt_call.memcpy@@GLIBC_2.17
0.02%  poisson2d ld-2.17.so   [.] do_lookup_x
0.01%  poisson2d libc-2.17.so  [.] vfprintf@@GLIBC_2.17
0.01%  poisson2d libc-2.17.so  [.] _dl_addr
0.01%  poisson2d ld-2.17.so   [.] _dl_relocate_object
0.01%  poisson2d ld-2.17.so   [.] check_match.10253
0.01%  poisson2d ld-2.17.so   [.] _dl_lookup_symbol_x
0.01%  poisson2d ld-2.17.so   [.] strcmp
0.00%  poisson2d ld-2.17.so   [.] open_path
0.00%  poisson2d ld-2.17.so   [.] init_tls
0.00%  poisson2d ld-2.17.so   [.] _dl_sysdep_start
0.00%  poisson2d ld-2.17.so   [.] _start

#
# (Tip: For tracepoint events, try: perf report -s trace_fields)
#

```

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 [79]: !make gen_profile

gcc -std=c99 -mcpu=power9 -DUSE_DOUBLE -mvsx -maltivec -Ofast -fprofile-
generate=/gpfs/wolf/trn003/scratch/aherten/ poisson2d.c -o poisson2d_train -lm
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS ./poisson2d_train 100
100 100
Job <24905> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 100 iterations on 100 x 100 mesh
Calculate current execution.
    0, 0.249490
echo `date` > /gpfs/wolf/trn003/scratch/aherten/.profile_generated
gcc -std=c99 -mcpu=power9 -DUSE_DOUBLE -mvsx -maltivec -Ofast -fprofile-
use=/gpfs/wolf/trn003/scratch/aherten/ poisson2d.c -o poisson2d_ref -lm
cp poisson2d_ref poisson2d
```

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

```
In [80]: !make run

bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d
Job <24906> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
    0, 0.249995
   100, 0.248997
   200, 0.248007
   300, 0.247025
   400, 0.246050
   500, 0.245084
   600, 0.244124
   700, 0.243173
   800, 0.242228
   900, 0.241291
2.28user 0.01system 0:02.30elapsed 99%CPU (0avgtext+0avgdata 24192maxresident)k
```



```
256inputs+0outputs (0major+479minor)pagefaults 0swaps
```

Great! It is! In our tests, this shaved off another 5%.

Let's also measure instructions and cycles

```
In [81]: !make run_perf
```

```
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf stat -e
cycles,instructions ./poisson2d
Job <24907> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291
```

```
Performance counter stats for './poisson2d':
```

```
       7925983538      cycles:u
    12253080719      instructions:u      #    1.55  insn per cycle

    2.313471365 seconds time elapsed
```

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 this information to a file, use `-` for example `-fopt-info-all=(SC19_DIR_SCRATCH)/filename`.

```
In [82]: !make poisson2d_0fast_info
```

```
gcc -std=c99 -mcpu=power9 -DUSE_DOUBLE -mvsx -maltivec -Ofast -fopt-info-all
poisson2d.c poisson2d_reference.o -o poisson2d_0fast_info -lm
poisson2d.c:62:14: optimized:   Inlining atoi/24 into main/33 (always_inline).
```

```

poisson2d.c:61:14: optimized:   Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:56:14: optimized:   Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:52:20: optimized:   Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:161:5: missed:     not inlinable: main/33 -> free/38, function body not
available
poisson2d.c:159:5: missed:     not inlinable: main/33 -> free/38, function body not
available
poisson2d.c:158:5: missed:     not inlinable: main/33 -> free/38, function body not
available
poisson2d.c:142:31: missed:    not inlinable: main/33 -> printf/36, function body not
available
poisson2d.c:103:5: missed:     not inlinable: main/33 -> __builtin_puts/37, function
body not available
poisson2d.c:96:5: missed:      not inlinable: main/33 -> printf/36, function body not
available
poisson2d.c:78:29: missed:     not inlinable: main/33 -> exp/35, function body not
available
poisson2d.c:68:41: missed:     not inlinable: main/33 -> malloc/34, function body not
available
poisson2d.c:67:41: missed:     not inlinable: main/33 -> malloc/34, function body not
available
poisson2d.c:65:41: missed:     not inlinable: main/33 -> malloc/34, function body not
available
/usr/include/stdlib.h:280:16: missed:    not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:    not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:    not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:    not inlinable: main/33 -> strtol/39, function
body not available
Unit growth for small function inlining: 207->207 (0%)

```

Inlined 4 calls, eliminated 0 functions

```

consider run-time aliasing test between *_84 and *_87
consider run-time aliasing test between *_92 and *_97
consider run-time aliasing test between *_104 and *_107
consider run-time aliasing test between *_111 and *_115
poisson2d.c:124:13: optimized: Loop 8 distributed: split to 0 loops and 1 library
calls.
poisson2d.c:90:9: optimized: Loop 10 distributed: split to 0 loops and 1 library
calls.
poisson2d.c:108:25: missed: couldn't vectorize loop
poisson2d.c:108:25: missed: not vectorized: loop nest containing two or more
consecutive inner loops cannot be vectorized
poisson2d.c:136:9: missed: couldn't vectorize loop
poisson2d.c:136:9: missed: Loop costings may not be worthwhile.
poisson2d.c:131:9: missed: couldn't vectorize loop
poisson2d.c:131:9: missed: Loop costings may not be worthwhile.
poisson2d.c:122:9: missed: couldn't vectorize loop
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_543, _539,
_549);
poisson2d.c:112:9: missed: couldn't vectorize loop
poisson2d.c:112:9: missed: not vectorized: control flow in loop.
poisson2d.c:114:13: optimized: loop vectorized using 16 byte vectors
poisson2d.c:88:5: missed: couldn't vectorize loop
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_528, 0, _531);
poisson2d.c:72:5: missed: couldn't vectorize loop

```

```

poisson2d.c:78:27: missed: not vectorized: complicated access pattern.
poisson2d.c:74:9: missed: couldn't vectorize loop
poisson2d.c:78:29: missed: not vectorized: relevant stmt not supported: _27 = exp
(_21);
poisson2d.c:43:5: note: vectorized 1 loops in function.
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_528, 0, _531);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_543, _539,
_549);
poisson2d.c:114:13: optimized: loop turned into non-loop; it never loops
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _187 = strtol (_1,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _189 = strtol (_2,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _193 = strtol (_3,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _191 = strtol (_4,
0B, 10);
poisson2d.c:65:41: missed: statement clobbers memory: A_153 = malloc (_7);
poisson2d.c:67:41: missed: statement clobbers memory: Anew_155 = malloc (_7);
poisson2d.c:68:41: missed: statement clobbers memory: rhs_157 = malloc (_7);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_528, 0, _531);
poisson2d.c:96:5: missed: statement clobbers memory: printf ("Jacobi relaxation
calculation: max %d iterations on %d x %d mesh\n", iter_max_130, ny_139, nx_195);
poisson2d.c:103:5: missed: statement clobbers memory: __builtin_puts (&"Calculate
current execution."[0]);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_543, _539,
_549);
poisson2d.c:142:31: missed: statement clobbers memory: printf ("%5d, %0.6f\n",
iter_237, error_219);
poisson2d.c:158:5: missed: statement clobbers memory: free (rhs_202);
poisson2d.c:159:5: missed: statement clobbers memory: free (Anew_124);
poisson2d.c:161:5: missed: statement clobbers memory: free (A_123);
poisson2d.c:65:41: missed: statement clobbers memory: A_144 = malloc (8000000);
poisson2d.c:67:41: missed: statement clobbers memory: Anew_143 = malloc (8000000);
poisson2d.c:68:41: missed: statement clobbers memory: rhs_142 = malloc (8000000);

```

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 [83]: `!make poisson2d_ref_info`

```

gcc -std=c99 -mcpu=power9 -DUSE_DOUBLE -mvsx -maltivec -Ofast -fopt-info-all
-fprofile-use=/gpfs/wolf/trn003/scratch/aherten/ -Ofast -fprofile-
generate=/gpfs/wolf/trn003/scratch/aherten/ poisson2d.c -o poisson2d_train -lm
poisson2d.c:62:14: optimized: Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:61:14: optimized: Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:56:14: optimized: Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:52:20: optimized: Inlining atoi/24 into main/33 (always_inline).
Increasing alignment of decl: __gcov0.main
poisson2d.c:164:1: missed: not inlinable: _GLOBAL__sub_D_00100_1_main/48 ->
__gcov_exit/55, function body not available
poisson2d.c:164:1: missed: not inlinable: _GLOBAL__sub_I_00100_0_main/47 ->
__gcov_init/54, function body not available
poisson2d.c:161:5: missed: not inlinable: main/33 -> free/38, function body not
available

```

```

poisson2d.c:159:5: missed:    not inlinable: main/33 -> free/38, function body not
available
poisson2d.c:158:5: missed:    not inlinable: main/33 -> free/38, function body not
available
poisson2d.c:142:31: missed:   not inlinable: main/33 -> printf/36, function body not
available
poisson2d.c:103:5: missed:    not inlinable: main/33 -> __builtin_puts/37, function
body not available
poisson2d.c:96:5: missed:     not inlinable: main/33 -> printf/36, function body not
available
poisson2d.c:78:29: missed:    not inlinable: main/33 -> exp/35, function body not
available
poisson2d.c:68:41: missed:    not inlinable: main/33 -> malloc/34, function body not
available
poisson2d.c:67:41: missed:    not inlinable: main/33 -> malloc/34, function body not
available
poisson2d.c:65:41: missed:    not inlinable: main/33 -> malloc/34, function body not
available
/usr/include/stdlib.h:280:16: missed:   not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:   not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:   not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:   not inlinable: main/33 -> strtol/39, function
body not available
Unit growth for small function inlining: 295->295 (0%)

```

Inlined 4 calls, eliminated 0 functions

```

consider run-time aliasing test between *_84 and *_87
consider run-time aliasing test between *_92 and *_97
consider run-time aliasing test between *_104 and *_107
consider run-time aliasing test between *_111 and *_115
poisson2d.c:124:13: optimized: Loop 8 distributed: split to 0 loops and 1 library
calls.
poisson2d.c:90:9: optimized: Loop 10 distributed: split to 0 loops and 1 library
calls.
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_64, _135,
_313);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_632, 0, _239);
poisson2d.c:108:25: missed: couldn't vectorize loop
poisson2d.c:108:25: missed: not vectorized: loop nest containing two or more
consecutive inner loops cannot be vectorized
poisson2d.c:136:9: missed: couldn't vectorize loop
poisson2d.c:136:9: missed: Loop costings may not be worthwhile.
poisson2d.c:131:9: missed: couldn't vectorize loop
poisson2d.c:131:9: missed: Loop costings may not be worthwhile.
poisson2d.c:122:9: missed: couldn't vectorize loop
poisson2d.c:122:9: missed: not vectorized: control flow in loop.
poisson2d.c:112:9: missed: couldn't vectorize loop
poisson2d.c:112:9: missed: not vectorized: control flow in loop.
poisson2d.c:114:13: optimized: loop vectorized using 16 byte vectors
poisson2d.c:88:5: missed: couldn't vectorize loop
poisson2d.c:88:5: missed: not vectorized: control flow in loop.
poisson2d.c:72:5: missed: couldn't vectorize loop
poisson2d.c:72:5: missed: not vectorized: control flow in loop.
poisson2d.c:74:9: missed: couldn't vectorize loop
poisson2d.c:78:29: missed: not vectorized: relevant stmt not supported: _27 = exp

```

```

(_21);
poisson2d.c:43:5: note: vectorized 1 loops in function.
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_632, 0, _239);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_64, _135,
_313);
poisson2d.c:114:13: optimized: loop turned into non-loop; it never loops
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _187 = strtol (_1,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _189 = strtol (_2,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _193 = strtol (_3,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _191 = strtol (_4,
0B, 10);
poisson2d.c:65:41: missed: statement clobbers memory: A_153 = malloc (_7);
poisson2d.c:67:41: missed: statement clobbers memory: Anew_155 = malloc (_7);
poisson2d.c:68:41: missed: statement clobbers memory: rhs_157 = malloc (_7);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_632, 0, _239);
poisson2d.c:96:5: missed: statement clobbers memory: printf ("Jacobi relaxation
calculation: max %d iterations on %d x %d mesh\n", iter_max_337, ny_124, nx_286);
poisson2d.c:103:5: missed: statement clobbers memory: __builtin_puts (&"Calculate
current execution."[0]);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_64, _135,
_313);
poisson2d.c:142:31: missed: statement clobbers memory: printf ("%5d, %0.6f\n",
iter_316, error_118);
poisson2d.c:158:5: missed: statement clobbers memory: free (rhs_127);
poisson2d.c:159:5: missed: statement clobbers memory: free (Anew_311);
poisson2d.c:161:5: missed: statement clobbers memory: free (A_122);
poisson2d.c:65:41: missed: statement clobbers memory: A_129 = malloc (8000000);
poisson2d.c:67:41: missed: statement clobbers memory: Anew_132 = malloc (8000000);
poisson2d.c:68:41: missed: statement clobbers memory: rhs_140 = malloc (8000000);
poisson2d.c:136:9: note: considering unrolling loop 7 at BB 53
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:136:9: optimized: loop unrolled 7 times (header execution count 9800)
poisson2d.c:131:9: note: considering unrolling loop 6 at BB 50
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:131:9: optimized: loop unrolled 7 times (header execution count 9800)
poisson2d.c:122:9: note: considering unrolling loop 5 at BB 47
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:122:9: optimized: loop unrolled 3 times (header execution count 9800)
poisson2d.c:118:25: note: considering unrolling loop 13 at BB 33
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:118:25: optimized: loop unrolled 3 times (header execution count 436550)
poisson2d.c:118:25: note: considering unrolling loop 9 at BB 30
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:112:9: note: considering unrolling loop 14 at BB 42
poisson2d.c:43:5: note: considering unrolling loop 4 at BB 40
poisson2d.c:108:25: note: considering unrolling loop 3 at BB 60
poisson2d.c:88:5: note: considering unrolling loop 2 at BB 23
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:88:5: optimized: loop unrolled 3 times (header execution count 100)
poisson2d.c:74:9: note: considering unrolling loop 11 at BB 12

```

```

considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:74:9: optimized: loop unrolled 3 times (header execution count 9604)
poisson2d.c:72:5: note: considering unrolling loop 1 at BB 16
poisson2d.c:164:1: missed: statement clobbers memory: __gcov_init (&*.LPBX0);
poisson2d.c:164:1: missed: statement clobbers memory: __gcov_exit ();
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS ./poisson2d_train 100
100 100
Job <24908> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
libgcov profiling error:/gpfs/wolf/trn003/scratch/aherten//#autofs#nccsopen-svm1_home#
aherten#SC19-Tutorial#3-Optimizing_POWER#Handson#Task1#poisson2d.gcda:overwriting an
existing profile data with a different timestamp
Jacobi relaxation calculation: max 100 iterations on 100 x 100 mesh
Calculate current execution.
    0, 0.249490
echo `date` > /gpfs/wolf/trn003/scratch/aherten//.profile_generated
gcc -std=c99 -mcpu=power9 -DUSE_DOUBLE -mvsx -maltivec -Ofast -fopt-info-all
-fprofile-use=/gpfs/wolf/trn003/scratch/aherten/ poisson2d.c poisson2d_reference.o -o
poisson2d_ref_info -lm
poisson2d.c:62:14: optimized:    Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:61:14: optimized:    Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:56:14: optimized:    Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:52:20: optimized:    Inlining atoi/24 into main/33 (always_inline).
poisson2d.c:161:5: missed:      not inlinable: main/33 -> free/38, function body not
available
poisson2d.c:159:5: missed:      not inlinable: main/33 -> free/38, function body not
available
poisson2d.c:158:5: missed:      not inlinable: main/33 -> free/38, function body not
available
poisson2d.c:142:31: missed:      not inlinable: main/33 -> printf/36, function body not
available
poisson2d.c:103:5: missed:      not inlinable: main/33 -> __builtin_puts/37, function
body not available
poisson2d.c:96:5: missed:      not inlinable: main/33 -> printf/36, function body not
available
poisson2d.c:78:29: missed:      not inlinable: main/33 -> exp/35, function body not
available
poisson2d.c:68:41: missed:      not inlinable: main/33 -> malloc/34, function body not
available
poisson2d.c:67:41: missed:      not inlinable: main/33 -> malloc/34, function body not
available
poisson2d.c:65:41: missed:      not inlinable: main/33 -> malloc/34, function body not
available
/usr/include/stdlib.h:280:16: missed:    not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:    not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:    not inlinable: main/33 -> strtol/39, function
body not available
/usr/include/stdlib.h:280:16: missed:    not inlinable: main/33 -> strtol/39, function
body not available
Unit growth for small function inlining: 207->207 (0%)

Inlined 4 calls, eliminated 0 functions

consider run-time aliasing test between *_84 and *_87
consider run-time aliasing test between *_92 and *_97

```

```

consider run-time aliasing test between *_104 and *_107
consider run-time aliasing test between *_111 and *_115
poisson2d.c:124:13: optimized: Loop 8 distributed: split to 0 loops and 1 library
calls.
poisson2d.c:90:9: optimized: Loop 10 distributed: split to 0 loops and 1 library
calls.
poisson2d.c:108:25: missed: couldn't vectorize loop
poisson2d.c:108:25: missed: not vectorized: loop nest containing two or more
consecutive inner loops cannot be vectorized
poisson2d.c:136:9: missed: couldn't vectorize loop
poisson2d.c:136:9: missed: Loop costings may not be worthwhile.
poisson2d.c:131:9: missed: couldn't vectorize loop
poisson2d.c:131:9: missed: Loop costings may not be worthwhile.
poisson2d.c:122:9: missed: couldn't vectorize loop
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_539, _535,
_544);
poisson2d.c:112:9: missed: couldn't vectorize loop
poisson2d.c:112:9: missed: not vectorized: control flow in loop.
poisson2d.c:114:13: optimized: loop vectorized using 16 byte vectors
poisson2d.c:88:5: missed: couldn't vectorize loop
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_524, 0, _527);
poisson2d.c:72:5: missed: couldn't vectorize loop
poisson2d.c:78:27: missed: not vectorized: complicated access pattern.
poisson2d.c:74:9: missed: couldn't vectorize loop
poisson2d.c:78:29: missed: not vectorized: relevant stmt not supported: _27 = exp
(_21);
poisson2d.c:43:5: note: vectorized 1 loops in function.
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_524, 0, _527);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_539, _535,
_544);
poisson2d.c:114:13: optimized: loop turned into non-loop; it never loops
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _187 = strtol (_1,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _189 = strtol (_2,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _193 = strtol (_3,
0B, 10);
/usr/include/stdlib.h:280:16: missed: statement clobbers memory: _191 = strtol (_4,
0B, 10);
poisson2d.c:65:41: missed: statement clobbers memory: A_153 = malloc (_7);
poisson2d.c:67:41: missed: statement clobbers memory: Anew_155 = malloc (_7);
poisson2d.c:68:41: missed: statement clobbers memory: rhs_157 = malloc (_7);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memset (_524, 0, _527);
poisson2d.c:96:5: missed: statement clobbers memory: printf ("Jacobi relaxation
calculation: max %d iterations on %d x %d mesh\n", iter_max_130, ny_139, nx_195);
poisson2d.c:103:5: missed: statement clobbers memory: __builtin_puts (&"Calculate
current execution."[0]);
poisson2d.c:43:5: missed: statement clobbers memory: __builtin_memcpy (_539, _535,
_544);
poisson2d.c:142:31: missed: statement clobbers memory: printf ("%5d, %0.6f\n",
iter_237, error_219);
poisson2d.c:158:5: missed: statement clobbers memory: free (rhs_202);
poisson2d.c:159:5: missed: statement clobbers memory: free (Anew_124);
poisson2d.c:161:5: missed: statement clobbers memory: free (A_123);
poisson2d.c:65:41: missed: statement clobbers memory: A_144 = malloc (8000000);
poisson2d.c:67:41: missed: statement clobbers memory: Anew_143 = malloc (8000000);
poisson2d.c:68:41: missed: statement clobbers memory: rhs_142 = malloc (8000000);
poisson2d.c:136:9: note: considering unrolling loop 7 at BB 47
considering unrolling loop with constant number of iterations

```

```

considering unrolling loop with runtime-computable number of iterations
poisson2d.c:136:9: optimized: loop unrolled 7 times (header execution count 9800)
poisson2d.c:131:9: note: considering unrolling loop 6 at BB 44
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:131:9: optimized: loop unrolled 7 times (header execution count 9800)
poisson2d.c:122:9: note: considering unrolling loop 5 at BB 40
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:122:9: optimized: loop unrolled 7 times (header execution count 9701)
poisson2d.c:118:25: note: considering unrolling loop 13 at BB 27
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:118:25: optimized: loop unrolled 3 times (header execution count 436550)
poisson2d.c:118:25: note: considering unrolling loop 9 at BB 24
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:112:9: note: considering unrolling loop 14 at BB 37
poisson2d.c:43:5: note: considering unrolling loop 4 at BB 35
poisson2d.c:108:25: note: considering unrolling loop 3 at BB 51
poisson2d.c:88:5: note: considering unrolling loop 2 at BB 18
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:88:5: optimized: loop unrolled 7 times (header execution count 99)
poisson2d.c:74:9: note: considering unrolling loop 11 at BB 9
considering unrolling loop with constant number of iterations
considering unrolling loop with runtime-computable number of iterations
poisson2d.c:74:9: optimized: loop unrolled 3 times (header execution count 9604)
poisson2d.c:72:5: note: considering unrolling loop 1 at BB 14

```

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, lets change directory to that of Task 2

```
In [85]: %cd ../Task2
```

```
/autofs/nccsopen-svm1_home/aherten/SC19-Tutorial/3-Optimizing_POWER/Handson/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 [97]: !make clean
```

```
rm -f poisson2d poisson2d*.o
```

```
In [88]: !make poisson2d CC=gcc
         !make run
         !make l3missstats
```

```
make: `poisson2d' is up to date.
```

```
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d
```

```
Job <24911> is submitted to default queue <batch>.
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
```

```
Calculate current execution.
```

```
0, 0.249995
100, 0.248997
200, 0.248007
300, 0.247025
400, 0.246050
500, 0.245084
600, 0.244124
700, 0.243173
800, 0.242228
900, 0.241291
```

```
2.39user 0.01system 0:02.40elapsed 100%CPU (0avgtext+0avgdata 24256maxresident)k
```

```
0inputs+0outputs (0major+480minor)pagefaults 0swaps
```

```
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf stat -e
```

```
cycles,r168a4 ./poisson2d
```

```
Job <24912> is submitted to default queue <batch>.
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
```

```
Calculate current execution.
```

```
0, 0.249995
100, 0.248997
200, 0.248007
300, 0.247025
400, 0.246050
```

```

500, 0.245084
600, 0.244124
700, 0.243173
800, 0.242228
900, 0.241291

```

Performance counter stats for './poisson2d':

```

      8271503902      cycles:u
      481152478      r168a4:u

```

2.412224884 seconds time elapsed

```

In [98]: !make poisson2d_pref CC=gcc
         !make run
         !make l3missstats

```

```

gcc -std=c99 -DUSE_DOUBLE -Ofast -mcpu=power9 -mvsx -maltivec -fprefetch-loop-
arrays -fprefetch-loop-arrays poisson2d.c -o poisson2d_pref -lm
cp poisson2d_pref poisson2d

```

```

bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d
Job <24919> is submitted to default queue <batch>.

```

<<Waiting for dispatch ...>>

<<Starting on login1>>

Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh

Calculate current execution.

```

0, 0.249995
100, 0.248997
200, 0.248007
300, 0.247025
400, 0.246050
500, 0.245084
600, 0.244124
700, 0.243173
800, 0.242228
900, 0.241291

```

```

1.92user 0.00system 0:01.93elapsed 99%CPU (0avgtext+0avgdata 24256maxresident)k
256inputs+0outputs (0major+480minor)pagefaults 0swaps

```

```

bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf stat -e
cycles,r168a4 ./poisson2d

```

Job <24920> is submitted to default queue <batch>.

<<Waiting for dispatch ...>>

<<Starting on login1>>

Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh

Calculate current execution.

```

0, 0.249995
100, 0.248997
200, 0.248007
300, 0.247025
400, 0.246050
500, 0.245084
600, 0.244124
700, 0.243173
800, 0.242228
900, 0.241291

```

Performance counter stats for './poisson2d':

```
6586609284      cycles:u
459879452       r168a4:u
```

1.925399505 seconds time elapsed

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

```
gcc -std=c99 -DUSE_DOUBLE -O3 -mcpu=power9 -mvsx -maltivec poisson2d.c -o
poisson2d -lm
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d
Job <24923> is submitted to default queue <batch>.
```

<<Waiting for dispatch ...>>

<<Starting on login1>>

Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh  
Calculate current execution.

```
0, 0.249995
100, 0.248997
200, 0.248007
300, 0.247025
400, 0.246050
500, 0.245084
600, 0.244124
700, 0.243173
800, 0.242228
900, 0.241291
```

```
4.73user 0.00system 0:04.73elapsed 99%CPU (0avgtext+0avgdata 24256maxresident)k
256inputs+0outputs (0major+479minor)pagefaults 0swaps
```

```
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf stat -e
cycles,r168a4 ./poisson2d
```

Job <24924> is submitted to default queue <batch>.

<<Waiting for dispatch ...>>

<<Starting on login1>>

Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh  
Calculate current execution.

```
0, 0.249995
100, 0.248997
200, 0.248007
300, 0.247025
400, 0.246050
500, 0.245084
600, 0.244124
700, 0.243173
800, 0.242228
900, 0.241291
```

Performance counter stats for './poisson2d':

```
16445764669      cycles:u
645094089        r168a4:u
```

4.792567763 seconds time elapsed

```

In [101]: !make poisson2d_pref CC=gcc -B
           !make run
           !make l3missstats

gcc -std=c99 -DUSE_DOUBLE -O3 -mcpu=power9 -mvsx -maltivec -fprefetch-loop-arrays
poisson2d.c -o poisson2d_pref -lm
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d
Job <24925> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291
4.74user 0.00system 0:04.74elapsed 99%CPU (0avgtext+0avgdata 24256maxresident)k
0inputs+0outputs (0major+480minor)pagefaults 0swaps
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS perf stat -e
cycles,r168a4 ./poisson2d
Job <24926> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291

Performance counter stats for './poisson2d':

      16239159454      cycles:u
       631061431      r168a4:u

      4.730144897 seconds time elapsed

```

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

Observing the results, we see that SW Prefetching seems to help at `-Ofast` but not at `-O3`. We can use the steps described in the the next section to verify that the compiler has not inserted any SW prefetch operations at `-O3` at all. That is because in the `-O3` binary the time is dominated by `__fmax` call which causes the compiler to come to the conclusion that whatever benefit we obtain

by adding SW prefetch will be overshadowed by the penalty of `fmax()` GCC may add further loop optimizations such as unrolling upon invocation of `-fprefetch-loop-arrays`.

### 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 [114]: !make CC=gcc -B poisson2d_pref
          !objdump -lSd ./poisson2d_pref > poisson2d.dis
```

```
gcc -std=c99 -DUSE_DOUBLE -Ofast -mcpu=power9 -mvsx -maltivec -fprefetch-loop-arrays poisson2d.c -o poisson2d_pref -lm
```

```
In [116]: !grep dcb poisson2d.dis
```

```
10000b28: 2c d2 00 7c      dcbt    0,r26
10000b30: 2c ba 00 7c      dcbt    0,r23
10000b38: 2c b2 00 7c      dcbt    0,r22
10000b50: 2c d2 00 7c      dcbt    0,r26
10000b58: ec b9 00 7c      dcbtst  0,r23
10000b80: 2c d2 00 7c      dcbt    0,r26
10000e64: 2c 92 00 7c      dcbt    0,r18
10000e68: 2c 9a 00 7c      dcbt    0,r19
10000e6c: 2c a2 00 7c      dcbt    0,r20
10000e70: 2c aa 00 7c      dcbt    0,r21
10000e7c: 2c b2 00 7c      dcbt    0,r22
10000e80: 2c d2 00 7c      dcbt    0,r26
10000e94: ec b9 00 7c      dcbtst  0,r23
```

### 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 [117]: !make CC=xlc_r -B poisson2d
          !make run

xlc_r -std=c99 -DUSE_DOUBLE -Ofast -qarch=pwr9 -qtune=pwr9 -DINLINE_LIBS
poisson2d.c -o poisson2d -lm
1500-036: (I) The NOSTRICT option (default at OPT(3)) has the potential to alter
the semantics of a program. Please refer to documentation on the STRICT/NOSTRICT
option for more information.
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d
Job <24927> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 50.149062
 200, 99.849327
 300, 149.352369
 400, 198.659746
 500, 247.773000
 600, 296.693652
 700, 345.423208
 800, 393.963155
 900, 442.314962
2.26user 0.00system 0:02.27elapsed 99%CPU (0avgtext+0avgdata 24256maxresident)k
256inputs+0outputs (0major+477minor)pagefaults 0swaps
```

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

```
In [9]: !make poisson2d_dscr CC=xlc_r -B
        !make run

xlc_r -std=c99 -DUSE_DOUBLE -Ofast -qarch=pwr9 -qtune=pwr9 -DINLINE_LIBS
-qprefetch=dscr=1 poisson2d.c -o poisson2d_dscr -lm
1500-036: (I) The NOSTRICT option (default at OPT(3)) has the potential to alter
the semantics of a program. Please refer to documentation on the STRICT/NOSTRICT
option for more information.
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d
Job <24929> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291
4.58user 0.00system 0:04.59elapsed 99%CPU (0avgtext+0avgdata 24192maxresident)k
0inputs+0outputs (0major+476minor)pagefaults 0swaps
```

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

The DSCR register controls the operation of the HW Prefetcher on POWER9. It can be modified in the command line by `ppc64_cpu --dscr=<value>`. However this needs admin privileges. IBM XL offers a compiler flag to set the value through the compiler. `-qprefetch=dscr=1` turns off the prefetcher. Observing the results we see that the performance without the HW prefetcher is twice as bad as that with default prefetching. So we can conclude that Prefetching helps the Jacobi application.

## 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 [10]: %cd ../Task3
```

```
/autofs/nccsopen-svm1_home/aherten/SC19-Tutorial/3-Optimizing_POWER/Handson/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 [39]: !make poisson2d CC=gcc
```

```
gcc -c -std=c99 -DUSE_DOUBLE -O3 -mcpu=power9 -mvsx -maltivec -fopenmp -lgomp
poisson2d_reference.c -o poisson2d_reference.o -lm
gcc -std=c99 -DUSE_DOUBLE -O3 -mcpu=power9 -mvsx -maltivec -fopenmp -lgomp
poisson2d.c poisson2d_reference.o -o poisson2d -lm
```

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 [40]: !eval $$SC19_SUBMIT_CMD ./poisson2d 1000 1000 1000

Job <24951> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh
Calculate reference solution and time with serial CPU execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291
Calculate current execution.
  0, 0.249995
 100, 0.248997
 200, 0.248007
 300, 0.247025
 400, 0.246050
 500, 0.245084
 600, 0.244124
 700, 0.243173
 800, 0.242228
 900, 0.241291
1000x1000: Ref:   4.7430 s, This:   3.9363 s, speedup:    1.20
```

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 [41]: !OMP_NUM_THREADS=1 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
          | grep speedup

<<Waiting for dispatch ...>>
<<Starting on login1>>
1000x1000: Ref:   4.7288 s, This:   4.9791 s, speedup:    0.95

In [42]: !OMP_NUM_THREADS=2 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
          | grep speedup

<<Waiting for dispatch ...>>
```



```

<<Starting on login1>>
1000x1000: Ref:  4.7125 s, This:  2.4914 s, speedup:    1.89

In [35]: !OMP_NUM_THREADS=4 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup

<<Waiting for dispatch ...>>
<<Starting on login1>>
1000x1000: Ref:  2.1065 s, This:  1.3836 s, speedup:    1.52

In [21]: !OMP_NUM_THREADS=8 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup

<<Waiting for dispatch ...>>
<<Starting on login1>>
1000x1000: Ref:  2.3868 s, This:  0.5272 s, speedup:    4.53

In [22]: !OMP_NUM_THREADS=10 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup

<<Waiting for dispatch ...>>
<<Starting on login1>>
1000x1000: Ref:  2.3912 s, This:  0.4612 s, speedup:    5.18

In [23]: !OMP_NUM_THREADS=20 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup

<<Waiting for dispatch ...>>
<<Starting on login1>>
1000x1000: Ref:  2.3864 s, This:  0.4037 s, speedup:    5.91

In [24]: !OMP_NUM_THREADS=40 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup

<<Waiting for dispatch ...>>
<<Starting on login1>>
1000x1000: Ref:  2.3773 s, This:  0.3045 s, speedup:    7.81

In [25]: !OMP_NUM_THREADS=80 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup

<<Waiting for dispatch ...>>
<<Starting on login1>>
1000x1000: Ref:  2.3819 s, This:  0.3081 s, speedup:    7.73

```

### 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 simulations threads running per core (*SMT*, Simultaneous Multi-Threading).

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

```
In [55]: !eval $SC19_SUBMIT_CMD ppc64_cpu --smt
Job <24465> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
SMT=4
```

There are more sources 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 [36]: !$$SC19_SUBMIT_CMD cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
!$$SC19_SUBMIT_CMD cat /sys/devices/system/cpu/cpu5/topology/thread_siblings_list

Job <24949> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
0-3
Job <24950> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
4-7
```

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?

Running with two different configurations 1) Binding all threads to the same core 2) Binding all threads to different cores, we see a higher speedup in case of binding all threads to different cores.

Using `OMP_PLACES` for binding, and using some magical Python-Bash interplay:

```
In [43]: for affinity in ["{0},{1},{2},{3}", "{0},{5},{9},{13}"]:
          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"
```

```

Affinity: {0},{1},{2},{3}
<<Waiting for dispatch ...>>
<<Starting on login1>>
OMP_PLACES = '{0},{1},{2},{3}'
1000x1000: Ref: 4.7315 s, This: 3.9090 s, speedup: 1.21
Affinity: {0},{5},{9},{13}
<<Waiting for dispatch ...>>
<<Starting on login1>>
OMP_PLACES = '{0},{5},{9},{13}'
1000x1000: Ref: 4.6485 s, This: 1.2829 s, speedup: 3.62

```

In this case, we carry out the same experiment using `GOMP_CPU_AFFINITY` which essentially sets the same environment variable `OMP_PLACES`. Running with two different configurations 1) Binding all threads to the same core 2) Binding all threads to different cores, we see a higher speedup in case of binding all threads to different cores.

```

In [44]: for affinity in ["0,1,2,3", "0,5,9,13"]:
          print("Affinity: {}".format(affinity))
          !eval OMP_DISPLAY_ENV=true GOMP_CPU_AFFINITY=$affinity OMP_NUM_THREADS=4
          $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000 | grep
          "OMP_PLACES\|speedup"

```

```

Affinity: 0,1,2,3
<<Waiting for dispatch ...>>
<<Starting on login1>>
OMP_PLACES = '{0},{1},{2},{3}'
1000x1000: Ref: 2.3964 s, This: 2.1361 s, speedup: 1.12
Affinity: 0,5,9,13
<<Waiting for dispatch ...>>
<<Starting on login1>>
OMP_PLACES = '{0},{5},{9},{13}'
1000x1000: Ref: 2.3925 s, This: 0.7030 s, speedup: 3.40

```

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 [44]: !make CC=xlc_r -B run

xlc_r -c -std=c99 -DUSE_DOUBLE -O3 -qhot -qtune=pwr9 -DINLINE_LIBS -qsmp=omp
poisson2d_reference.c -o poisson2d_reference.o -lm
1500-036: (I) The NOSTRICT option (default at OPT(3)) has the potential to alter
the semantics of a program. Please refer to documentation on the STRICT/NOSTRICT
option for more information.
xlc_r -std=c99 -DUSE_DOUBLE -O3 -qhot -qtune=pwr9 -DINLINE_LIBS -qsmp=omp
poisson2d.c poisson2d_reference.o -o poisson2d -lm
1500-036: (I) The NOSTRICT option (default at OPT(3)) has the potential to alter
the semantics of a program. Please refer to documentation on the STRICT/NOSTRICT
option for more information.
bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d
Job <24956> is submitted to default queue <batch>.
<<Waiting for dispatch ...>>
<<Starting on login1>>
Jacobi relaxation calculation: max 1000 iterations on 1000 x 1000 mesh

```

Calculate reference solution and time with serial CPU execution.

```
0, 0.249995
100, 50.149062
200, 99.849327
300, 149.352369
400, 198.659746
500, 247.773000
600, 296.693652
700, 345.423208
800, 393.963155
900, 442.314962
```

Calculate current execution.

```
0, 0.249995
100, 50.149062
200, 99.849327
300, 149.352369
400, 198.659746
500, 247.773000
600, 296.693652
700, 345.423208
800, 393.963155
900, 442.314962
```

```
1000x1000: Ref: 5.6783 s, This: 2.6528 s, speedup: 2.14
21.56user 6.18system 0:08.37elapsed 331%CPU (0avgtext+0avgdata 23040maxresident)k
3200inputs+0outputs (2major+1098minor)pagefaults 0swaps
```

Run the parallel application with varying number of threads (OMP\_NUM\_THREADS) and note the performance improvement.

Just as in the GCC binary we see a similar speedup with higher number of threads until a certain point beyond which the benefit tapers off.

```
In [28]: !OMP_NUM_THREADS=1 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
1000x1000: Ref: 2.2561 s, This: 2.6432 s, speedup: 0.85
```

```
In [29]: !OMP_NUM_THREADS=2 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
1000x1000: Ref: 2.3071 s, This: 1.5343 s, speedup: 1.50
```

```
In [30]: !OMP_NUM_THREADS=4 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
1000x1000: Ref: 2.2617 s, This: 0.6936 s, speedup: 3.26
```

```
In [31]: !OMP_NUM_THREADS=8 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
1000x1000: Ref: 2.2728 s, This: 0.3402 s, speedup: 6.68
```

```
In [45]: !OMP_NUM_THREADS=10 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
1000x1000: Ref: 2.1678 s, This: 0.2869 s, speedup: 7.56
```

```
In [33]: !OMP_NUM_THREADS=20 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
1000x1000: Ref: 2.2813 s, This: 0.1452 s, speedup: 15.71
```

```
In [34]: !OMP_NUM_THREADS=40 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
1000x1000: Ref: 2.3284 s, This: 0.0981 s, speedup: 23.75
```

```
In [35]: !OMP_NUM_THREADS=80 $$SC19_SUBMIT_CMD -c ALL_CPUS --bind none ./poisson2d 1000 1000 1000
| grep speedup
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
1000x1000: Ref: 2.2918 s, This: 0.1439 s, speedup: 15.92
```

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 [36]: for affinity in ["{0},{1},{2},{3}", "{0},{5},{9},{13}"]:
|         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"
```

```
Affinity: {0},{1},{2},{3}
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>
```

```
OMP_PLACES='{0},{1},{2},{3}' custom
```

```
1000x1000: Ref: 5.9792 s, This: 2.4122 s, speedup: 2.48
```

```
Affinity: {0},{5},{9},{13}
```

```
<<Waiting for dispatch ...>>
```

```
<<Starting on login1>>  
OMP_PLACES='{0},{5},{9},{13}' custom  
1000x1000: Ref: 2.3101 s, This: 0.6884 s, speedup: 3.36
```

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](#).