

Application Performance Tuning with Compilers on POWER9- Hands On Summary of Tasks

Archana Ravindar

Summary of Tasks

- The Tasks are designed to familiarize the users with various ways of tuning performance.
- There are three Tasks organized into Task-1, Task-2, Task-3
 - Task1 – Basic Compiler Optimizations and Annotations
 - Task2 – Impact of Software Prefetching
 - Task3 – Impact of OpenMP parallelism and Binding threads to cores
- All exercises work with a single Makefile.
- Most Exercises are designed with GCC compiler. Some also use the IBM XL compiler. In such cases, the users can define CC variable that directs the makefile to use appropriate flags; For ex: `make CC=gcc <target>`, `make CC=xlc_r <target>`
- The user can run
 - Makefile Targets
 - “run”: which will invoke `bsub -W 60 -nnodes 1 -Is -P TRN003 jsrun -n 1 -c 1 -g ALL_GPUS time ./poisson2d <niter> <ny> <nx>`
 - Alternatively, the same command can be invoked as `!${SC19_SUBMIT_CMD} time ./poisson2d <niter> <ny> <nx>`
 - The default values make run will invoke is (niter, nx and ny = 1000)
 - “run_perf” (prints cycles, instructions), “run_perf_recrep” (displays top routines execution time wise)
 - Other Additional targets are defined as we go along with the tasks
- Helpful references for each Task
- Jupyter notebook : `HandsOnPerformanceOptimization_Task.ipynb` that describes the tasks that need to be done
 - `HandsOnPerformanceOptimization_Solution.ipynb` that describes the solutions and results
- Solution/ directory contains the resultant binaries, makefiles, source files needed for the solutions

Task 1 (A) : Flag Tuning and Annotations

- This exercise uses the same Jacobi application that computes the 2-d poisson equation using the Jacobi solver. The default parameters are set to the following- NITER=1000, NY=1000, NX=1000.
- This hands-on exercise illustrates the impact of the Ofast flag
 - Ofast enables `-ffast-math` option that implements the same math function in a way that does not require guarantees of IEEE / ISO rules or specification and avoids the overhead of calling a function from the math library
- CFLAGS is the variable defined in Makefile.gcc that controls the compilation flags with which the application is compiled; You may see flags such as
 - `-mcpu=power9` : This leverages POWER9 ISA
 - `-mvsx` generates instructions that use vector/scalar instructions
 - `-maltivec` generates code that uses altivec instructions
- Add `-O3` to CFLAGS_O3 and compile the application to create the O3 binary (Makefile target: poisson2d_O3)
- Similarly Add `-Ofast` to CFLAGS_Ofast and compile the application to create the Ofast binary
- Measure performance of both the binaries with `make run` and `make runstats` and compare them
- View the performance profile of both binaries with `make perfgenerate` and compare them
- Why do you think one is faster than the other

Task 1 (B): Profile Directed Feedback Optimization

- Profile directed feedback is an advanced way of optimizing an application
 - Involves using profile information to optimize the performance of the application
 - The flags for profile directed feedback are `–fprofile-generate` and `–fprofile-use`
 - The steps to create a profile directed binary are (make runpdf)
 - Build using `fprofile-generate` (poisson2d_train is built)
 - Run binary with a smaller input workload
 - This generates a .gcda file
 - Build using `fprofile-use` (poisson2d_ref is built)
 - This uses information in the .gcda file and the final binary is generated
- Run the Ofast binary and Ofast binary optimized using profile directed feedback and compare their performance
 - make run

Task 1 (C): Compiler Annotations

- Compiler adds annotations in optimization passes to communicate the optimization decisions it has made during the compilation process.
- These annotations are very useful in order to understand why an optimization was made or why it was missed
- Newer versions of GCC introduce flags such as `-fopt-info-all=<filename>` that captures these annotations in a text file which can be read; `*all*` indicates all the optimization annotations
- In order to view only the missed ones, we can specify `-fopt-info-missed=<filename>` that captures the optimizations that could not be done either because the compiler thought it was not profitable to do in the given situation or the code characteristic is preventing it from making the optimization.
- Use `poisson2d_Ofast_record` target to generate the annotations file for compilation at Ofast level; The annotations are generated in file `opt-record` in `SC19_DIR_SCRATCH`.
- Use `runpdf.record` target to generate the annotations file for profile directed feedback optimization
- What differences do you see between annotations at Ofast and Ofast with profile directed feedback ?
- The steps to create a profile directed binary with annotations (generated in `$(SC19_DIR_SCRATCH)/filename` are make `runpdf.record`
 - Build using `fprofile-generate` (`poisson2d_train` is built)
 - Run binary with a smaller input workload
 - This generates a `.gcda` file
 - Build using `fprofile-use` and `-fopt-info-all=<filename>` (`poisson2d_ref_record` is built),
 - This uses information in the `.gcda` file and optimizes the binary and generates `filename`
- The annotations are in `filename`

Task 2(A): Impact of Software Prefetching

- This hands-on exercise illustrates ways in which we can enable SW Prefetching and through compiler flags also control the values of the Data Stream Control Register that directs the HW prefetcher
- Compiling with a prefetch flag enables the compiler to analyze the code and insert `__dcbt` and `__dcbtst` instructions into the code *“if it is determined to be beneficial by the compiler”*
- `__dcbt` and `__dcbtst` instructions prefetch memory values into L3
- The prefetch flag for GCC is `-fprefetch-loop-arrays`
- Add this flag to CFLAGS and compile the application with O3 and Ofast to create the binary with and without SW prefetching enabled. For this purpose we have defined target `poisson2d_pref` that is the binary with SW prefetching
- Check if the compiler has added the prefetch instructions by observing the disassembly using `objdump -lSd ./binary > assemblyfile.dis`
- Check the performance and L3 misses of O3(with and without prefetch) and Ofast(with and without prefetch) using target `“l3missstats”`
- At which optimization level does the compiler do prefetching ?

Task 2(B): Controlling contents of DSCR by compiler flags

- POWER9 has an inbuilt hardware prefetcher that is transparent to the programmer
- If the application accesses regular memory patterns, the hardware prefetcher automatically starts prefetching data
- The contents of DSCR register determine how aggressive or laid back is the hardware prefetcher in its operation
- The contents of the register can be changed on the OS command line using the command `ppc64_cpu -dscr=<value>` but needs administrator privileges
- Alternatively IBM XL provides a flag `-qprefetch=dscr=<value>` that sets the DSCR with `<value>`
- Default value of DSCR=0, DSCR=1 turns off hardware prefetching. DSCR=7 enables prefetching with larger depth with other default settings. And so on.
- Add `-qprefetch=dscr=1` to CFLAGS and compile the application
- For this purpose we have defined the target `poisson2d_dscr` in the Makefile
- Compare performance of the default binary with binary compiled with `-qprefetch=dscr=1`
- Does Hardware prefetching help the Jacobi application ?

Task 3(A) : OpenMP parallelization and Binding Threads to Cores

- This hands-on exercise illustrates performance tuning with OpenMP parallelization and it consists of two aspects.
 - First we need to parallelize the application using compiler flags. The hands on exercise will show you how to do it for GCC and IBM XL compilers.
 - Second we need to decide the best binding and run the application
- The poisson application is modified to execute two sets of loops that are identical copies of each other- The main loop is parallelized with openMP pragmas. The reference loop is not parallelized. The application computes the speedup in execution between the main and the reference loops.

Task 3(A) : OpenMP parallelization and Binding Threads to Cores

- Part 1: Parallelize the application
- Identify loops whose iterations are independent of each other
- Look for TODOs in poisson2d.c for placing `#pragma omp parallel` before any such loop such as
`#pragma omp parallel for`
`for (int iy = 1; iy < ny-1; iy++) { }`
- Look for TODOs to place appropriate compiler option to generate parallel binary
- Compile the application with GCC or XL using `make CC=gcc` or `make CC=xlc_r` to generate the parallel binary
- Run the parallel binary with `OMP_NUM_THREADS=1, 2,4,8,10,20,40` etc and compare speedup with reference loop

Task 3(B): Impact of Binding Threads to Cores

- Part 2: Determine the Beneficial Binding and run the Application
- Run `lscpu` to determine the number of sockets, cores per each socket
- You will see the setting as `SMT=4` on the system; You can verify by running `ppc64_cpu -smt` on the command line
- Run `Cat /proc/cpuinfo` to determine the total number of threads, cores in the system
- Obtain the thread sibling list of CPU0, CPU1 etc.. Reading the file `/sys/devices/system/cpu/cpu0/topology/thread_siblings_list`
- Referring to the sibling list, Set `n1, .. n4` to threads in same core and run for example-

`$(SC19_SUBMIT_CMD) time OMP_NUM_PLACES="{0},{1},{2},{3}" OMP_NUM_THREADS=4 ./poisson2d 1000 1000 1000`
- Set `n1, .. n4` to threads in different cores and run for example-

`$(SC19_SUBMIT_CMD) time OMP_NUM_PLACES="{0},{5},{9},{13}" OMP_NUM_THREADS=4 ./poisson2d 1000 1000 1000`
- Compare Speedups; Which one is higher?
- Both GCC and XL binaries can use the same binding variables to obtain speedups as `OMP_NUM_PLACES` and `OMP_NUM_THREADS` are variables specific to OpenMP. `GOMP_CPU_AFFINITY` is specific to GCC binaries.

Application Performance Tuning with Compilers on POWER9- Hands On Summary of Tasks

Archana Ravindar

Task 1(A,B) Result : Tuning with Compiler Flags

- This hands-on exercise illustrates the impact of the Ofast flag
 - Ofast enables `-ffast-math` option that implements the same math function in a way that does not require guarantees of IEEE / ISO rules or specification and avoids the overhead of calling a function from the math library
- It also illustrates the impact of profile directed feedback
 - Profile directed feedback involves optimizing the binary around

hot paths

– Top functions in the Profile of O3 binary

This can be obtained by running the following :

```
> perf record -e cycles ./poisson2d.O3; perf report;
```

```
65.6% poisson2d.O3 libm-2.26.so    [.] __fmax
21.21% poisson2d.O3 poisson2d.O3  [.] main
9.18%  poisson2d.O3 libc-2.17.so   [.] __memcpy_power7
```

– Top functions in the Profile of fast binary

```
81.12% poisson2d.fast poisson2d.fast [.] main
9.18%  poisson2d.fast libc-2.17.so    [.] __memcpy_power7
```

Niter, x,y=1000	Runtime (s)
O3 binary	4.73
fast binary	2.4
speedup with Ofast	1.9x

Niter,x,y=1000	Runtime (s)
fast binary	2.4
fast with PDF	2.3
% speedup with PDF	4%

Task 1(C) Result : Compiler Annotations

Compiler Annotations provide a wealth of information in understanding why certain optimizations were done or missed
The following annotations and more are obtained by building the application at Ofast with `-fopt-info-all=<path>/<file-name>`

```
poisson2d.c:38:5: note: vectorized 1 loops in function.  
poisson2d.c:127:9: missed: couldn't vectorize loop  
poisson2d.c:127:9: missed: Loop costings may not be worthwhile.  
poisson2d.c:107:9: missed: couldn't vectorize loop  
poisson2d.c:107:9: missed: not vectorized: control flow in loop.  
poisson2d.c:110:13: optimized: loop vectorized using 16 byte vectors  
poisson2d_reference.c:43:13: optimized: loop turned into non-loop; it never loops
```

Generating annotations with profile directed feedback generates additional annotations that contain details of execution frequency which determine many of the optimization parameters as shown below

```
poisson2d.c:114:25: optimized: loop unrolled 3 times (header execution count 436550)  
poisson2d.c:83:5: optimized: loop unrolled 7 times (header execution count 99)
```

Task 2(A) Result : Impact of Software Prefetching

- Compiling with a prefetch flag enables the compiler to analyze the code and insert `__dcbt` and `__dcbtst` instructions into the code *if it is beneficial*
- `__dcbt` and `__dcbtst` instructions prefetch memory values into L3 ; `__dcbt` is for load and `__dcbtst` is for store
- POWER9 has prefetching enabled both at HW and SW levels
- At HW level, prefetching is “ON” by default
- At the SW level, you can request the compiler to insert prefetch instructions ; However the compiler can choose to ignore the request if it determines that it is not beneficial to do so.
- You will find that the compiler generates prefetch instructions when the application is compiled at the Ofast level but not when It is compiled at the O3 level
- 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`

Niter, nx, ny=1000	Runtime (s)	Niter, nx, ny=500	Runtime(s)
Plain Ofast	2.4	Plain O3	4.72
Prefetch Ofast	1.9	Prefetch O3	4.72
% speedup	21%	% speedup	0

L3 cache misses	482546857
L3 cache misses with prefetch	457517865
% Reduction	6%

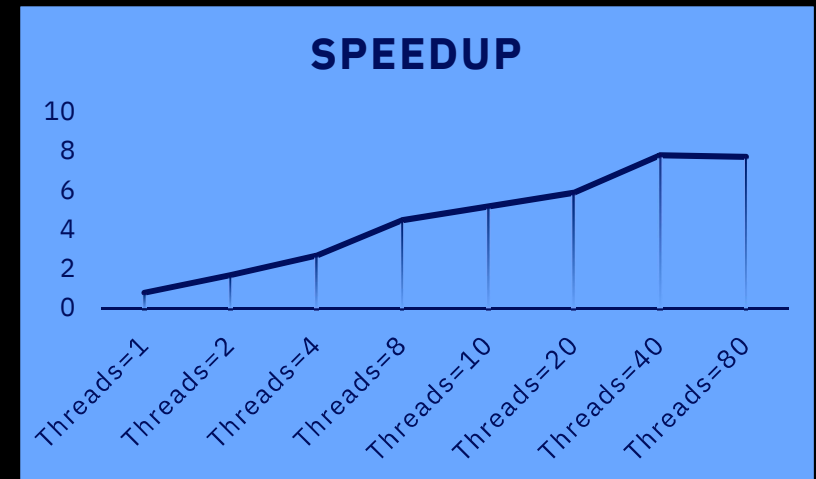
Task 2(B) Impact of hardware prefetcher

Niter, nx, ny=1000	Runtime (s)
Plain Ofast	2.26
Ofast with prefetching turned off	4.57
Degradation	~2x

- POWER9 has an inbuilt hardware prefetcher
- The parameters of the prefetcher are stored in a special register “Data Stream Control Register (DSCR)”
- The DSCR has various fields such as depth, ramp, urgency, load prefetch enable, store prefetch enable, stride prefetch etc
- At the command line the user can set it using
 - `ppc64_cpu -dscr=<val>`
 - However this requires admin privileges usually
- Alternatively certain compilers such as IBM XL has an option to set the value during compile time, `-qprefetch=dscr=<val>`
- This internally generates the instruction `mtudscr r[n]` or `mtspr 3, r[n]` that sets the DSCR to value in register `r[n]`
- By compiling with `-qprefetch=dscr=1`, the hardware prefetcher can be disabled
- Comparing the performance of the binary with and without the hardware prefetcher we see that this particular application benefits with prefetching

Task 3 Result : OpenMP Parallelization

- Parallelizing an application involves two steps
 - Using openMP pragmas to parallelize loops, we see that increasing the number of threads, the speedup also scales upto a certain point
 - Using the correct binding
- Binding all threads to the same core causes bottlenecks in execution thereby slowing down the application
- Binding all threads to different cores removes the bottleneck and improves performance
- Though not compiler related, this hands-on illustrates that system level tuning is also equally important to obtain performance improvements
- Results are similar in the XL binary as in the GCC binary



Binding for OMP_THREADS=4	Speedup
{0},{1},{2},{3} (OMP_PLACES)	1.1x
{0},{5},{9},{13} (OMP_PLACES)	3.4x