Compiler and Performance Optimizations

Pidad D'Souza (pidsouza@in.ibm.com)
IBM, Systems & Technology Group

IBM

Using the Compiler

- Compiler Invocations
- Program Checking Options
- Program Behavior Options
- Floating Point Control
- Optimization Levels
- Target Machines
- Compile Code for SMP

Controlling Language Level: Fortran

- Compiler invocations for standard compliant compilations
  - `xlf` or `f77`: Fortran 77
  - `xlf90`: Fortran 90
  - `xlf95`: Fortran 95
  - `xlf2003`: Fortran 2003

- Finer control through `-qlanglvl`, `-qxlf77` and `-qxlf90` options
  - Slight tweaks to I/O behavior
  - Intrinsic function behavior
  - `-qlanglvl` can be used for additional diagnostics

MPI and Threaded Code

- Threaded code:
  - append `_r` to the corresponding invocation for sequential code: for instance, `xlf90_r`
  - add `-qsmp=omp` for OpenMP code; you may need to add `-qnosave` for Fortran77 OpenMP code.
- Pure MPI code:
  - Prepend `mp` to the corresponding sequential invocation: for instance, `mpxlf90`.
- Threaded MPI code:
  - Append `_r` to the corresponding invocation for pure MPI code: for instance, `mpxlf90_r`;
  - add `-qsmp=omp` for OpenMP code; you may need to add `-qnosave` for Fortran77 OpenMP code.

Note: `mpxlf90` and `mpxlf90_r` are exactly the same now

Controlling Language Level: C/C++

- Compiler invocations for standard compliant compilations
  - `cc`: “traditional” K&R C
  - `xlc` or `c89`: ANSI89 standard C
  - `xlC`: ANSI98 standard C++
  - `c99`: ANSI99 standard C
  - `gxlc`: “gcc-like” command line
  - `gxlC`: “g++-like” command line
- Finer control through `-qlanglvl`
  - strict conformance checking
  - lots of C++ language variations
  - gcc compatibility control

MPI and Threaded Code

- Threaded code:
  - append `_r` to the corresponding invocation for sequential code: for instance, `c89_r`
  - add `-qsmp=omp` for OpenMP code
- Pure MPI code:
  - `mpcc, mpCC`
- Threaded MPI code:
  - `mpcc_r , mpCC_r`
  - add `-qsmp=omp` for OpenMP code

Note: `mpcc` and `mpcc_r` are exactly the same now

Mixed Language Programming

- Use Fortran compiler invocations to link object files that are generated with both the Fortran and C compilers
- Use C++ compiler invocations to link object files that are generated with both the C++ and C compilers
- To link object files generated with all three compilers, use the C++ compiler and explicitly list the Fortran libraries.
  - Use –v to figure out what libraries to list explicitly

mpxlf -v -o mpi_hello_f mpi_hello_f.o

```
xlf_r -F:mpxlf_r -v -o mpi_hello_f mpi_hello_f.o
-I/usr/lpp/ppe.poe/include/thread -I/opt/rsct/lapi/include
-llapi_r
exec: export(export,XL_CONFIG=/etc/xlf.cfg.53:mpxlf_r,NULL)
exec: /bin/ld(ld,-b32,/lib/crt0.o,-bpT:0x10000000,-
bpD:0x20000000,-binitfini:poe_remote_main,-bh:4,-
o,mpi_hello_f,mpi_hello_f.o,-llapi_r,
-L/usr/lpp/ppe.poe/lib/threads,-L/usr/lpp/ppe.poe/lib,-
L/lib/threads,-lmpi_r, -lxlf90,-L/usr/lpp/xlf/lib,-lxlopt,-lxlf,-
lxlomp_ser,-lpthreads,-lm,-lc,NULL)
```

Using GNU Compilers

- `Start with latest version of GNU`
- `-mcpu=power7 -mtune=power7`
  - Produce code to exploit power7 hardware
  - Optimization tuned for power7
- `-maltivec -mvsx`
  - Recognize vector types and formating extensions of C
  - Use vector scalar data types

Checking Program Correctness

- `-qcheck`
  - In Fortran, does bounds checking on array references, array sections and character substrings
  - In C/C++, checks for NULL pointers, for divide by zero and for array indices out of bounds
- `-qextchk`, `-btypchk`
  - Generates type hash codes so that the AIX linker can check type
  - consistency across files (also done by `-qipa`)
- `-qinitauto`
  - Generates extra code to initialize stack storage
  - Can be done bytewise or wordwise

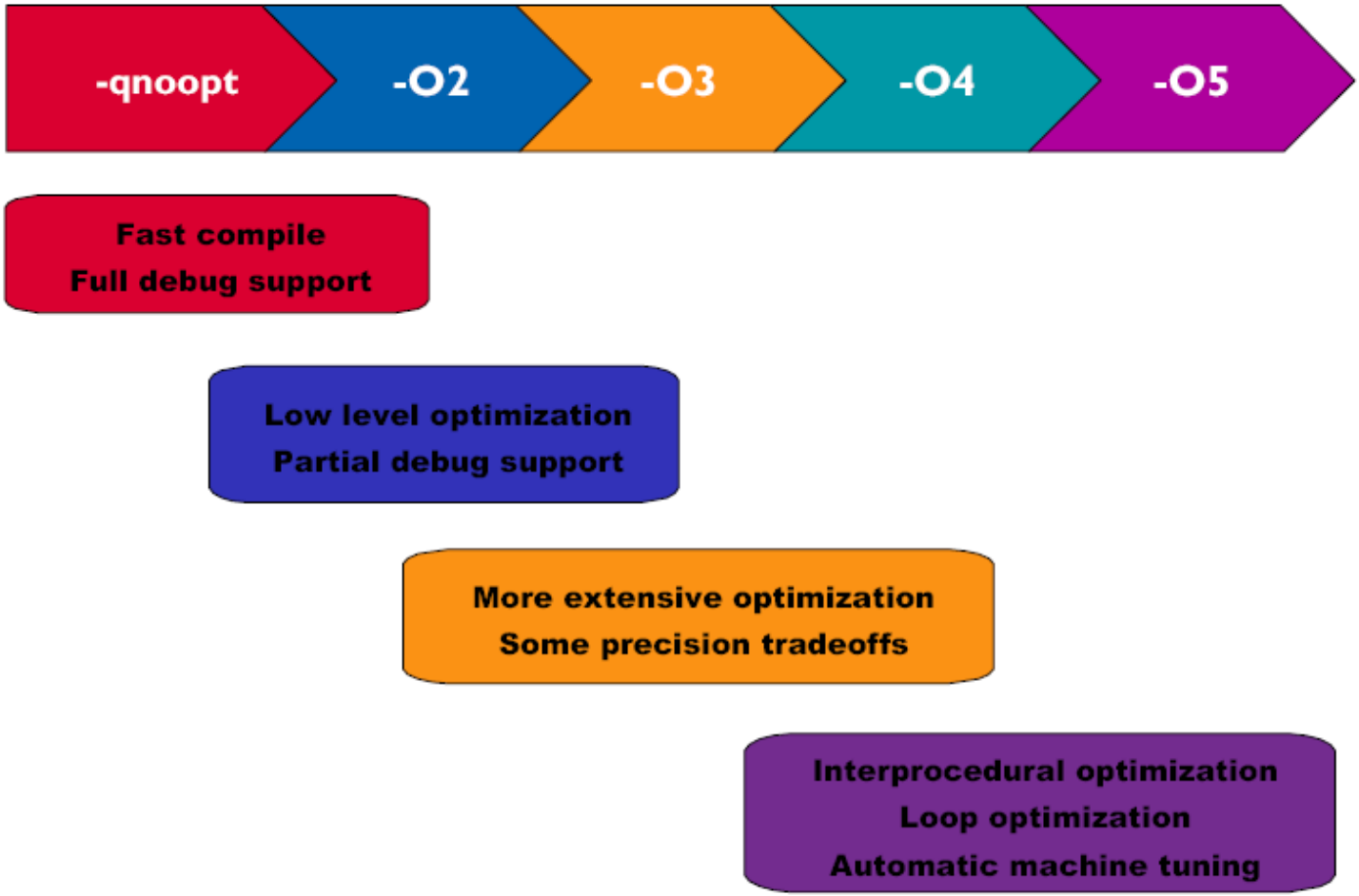
Program Behavior Options (-qstrict)

- `-q[no]strict`
  - Default is `-qstrict` with `-qnoopt` and `-O2`, `-qnostrict` with `-O3`, `-O4`, `-O5`
  - `-qnostrict` allows the compiler to reorder floating point calculations and potentially excepting instructions
  - Use `-qstrict` when your computation legitimately involves NaN, INF or denormalized values
  - Use `-qstrict` when exact compatibility is required with another IEEE compliant system
  - Note that `-qstrict` disables many potent optimizations so use it only when necessary and consider applying it at a file or even function level to limit the negative impact

Floating Point Trapping (-qflttrap)

- Enables software checking of IEEE floating point exceptions
- Usually more efficient than hardware checking since checks can be executed less frequently
- Specified as `-qflttrap=imprecise|enable`
  - `-qflttrap=imprecise`: check for error conditions at procedure entry/exit, otherwise check after any potentially excepting instruction
  - `-qflttrap=enable`: enables generation of checking code, also enables exceptions in hardware
  - `-qflttrap=overflow:underflow:zerodivide:inexact`: check given conditions
- In the event of an error, SIGTRAP is raised
  - As a convenience the `-qsigtrap` option will install a default handler which dumps a stack trace at the point of error (Fortran only)

Optimization Levels



Optimization Level –O2 (same as –O)

- Comprehensive low-level optimization
  - Global assignment of user variables to registers
  - Strength reduction and effective usage of addressing modes
  - Elimination of unused or redundant code
  - Movement of invariant code out of loops
  - Scheduling of instructions for the target machine
  - Some loop unrolling and pipelining
- Partial support for debugging
  - Externals and parameter registers visible at procedure boundaries
  - Snapshot pragma/directive creates additional program points for storage visibility
  - `-qkeepparm` option forces parameters to memory on entry so that they can be visible in a stack trace

Optimization Level –O3

- More extensive optimization
  - Deeper inner loop unrolling
  - Loop nest optimizations such as unroll-and-jam and interchange (`-qhot` subset)
  - Better loop scheduling
  - Additional optimizations allowed by `-qnostrict`
  - Widened optimization scope (typically whole procedure)
  - No implicit memory usage limits (`-qmaxmem=-1`)
- Some precision tradeoffs
  - Reordering of floating point computations
  - Reordering or elimination of possible exceptions (e.g., divide by zero, overflow)
- `-qoptdebug`
  - Improves the ability of debuggers to work with optimized code

Tips for getting the most out of –O2 and –O3

- If possible, test and debug your code without optimization before using `-O2`
- Ensure that your code is standard-compliant. Optimizers are the ultimate conformance test!
- In Fortran code, ensure that subroutine parameters comply with aliasing rules
- In C code, ensure that pointer use follows type restrictions (generic pointers should be `char*` or `void*`)
- Ensure all shared variables and pointers to same are marked volatile
- Compile as much of your code as possible with `-O2`
- If you encounter problems with `-O2`, consider using `-qalias=noansi` or `-qalias=nostd` rather than turning off optimization
- Next, use `-O3` on as much code as possible
- If you encounter problems or performance degradations, consider using `-qstrict`, `-qcompact`, or `-qnohot` along with `-O3` where necessary
- If you still have problems with `-O3`, switch to `-O2` for a subset of files/subroutines but consider using `-qmaxmem=-1` and/or `-qnostrict`

High Order Transformations (-qhot)

- Supported for all languages
- Specified as `-qhot[=[no]vector|arraypad[=n]|[no]simd]`
- Optimized handling of F90 array language constructs (elimination of temporaries, fusion of statements)
- High level transformation (e.g., interchange, fusion, unrolling) of loop nests to optimize:
  - memory locality (reduce cache/TLB misses)
  - usage of hardware prefetch
  - loop computation balance (typically ld/st vs. float)
- Optionally transforms loops to exploit MASS vector library (e.g., reciprocal, sqrt, trig) — may result in slightly different rounding
- Optionally introduces array padding under user control — potentially unsafe if not applied uniformly
- Optionally transforms loops to exploit VMX unit with `–qarch=pwr6 -qenablevmx`

Tips for getting the most out of -qhot

- Try using `-qhot` along with `-O2` or `-O3` for all of your code. It is designed to have neutral effect when no opportunities exist.
- If you encounter unacceptably long compile times (this can happen with complex loop nests) or if your performance degrades with the use of `-qhot`, try using `-qhot=novector`, or `-qstrict` or `-qcompact` along with `-qhot`
- If necessary, deactivate `-qhot` selectively, allowing it to improve some of your code.
- When `–qarch=pwr6`, the default with `–qhot` is to perform SIMD vectorization.
- You can specify `–qhot=nosimd` to disable SIMD vectorization
- Two levels of `–qhot` supported via `–qhot=level=x` where x is 0 or 1. Default is `–qhot=level=1` when `–qhot` is specified.
- `-qhot=level=0` is the default when `–O3` is specified
- Read the transformation report generated using `–qreport`. If your hot loops are not transformed as you expect, try using assertive directives such as `INDEPENDENT` or `CNCALL` or prescriptive directives such as `UNROLL` or `PREFETCH`.

Link-time Optimization (-qipa)

- Supported for all languages
- Can be specified on the compile step only or on both compile and link steps ("whole program" mode)
- Whole program mode expands the scope of optimization to an entire program unit (executable or shared object)
- Specified as `-qipa[=level=n | inline= |` fine tuning`]`
  - `level=0`: Program partitioning and simple interprocedural optimization
  - `level=1`: Inlining and global data mapping
  - `level=2`: Global alias analysis, specialization, interprocedural data flow
  - `inline=`: Precise user control of inlining
  - fine tuning: Specify library code behavior, tune program partitioning, read commands from a file

Tips for getting the most out of -qipa

- When specifying optimization options in a makefile, remember to use the compiler driver (`cc`, `xlf`, etc.) to link and repeat all options on the link step:

```
LD = xlf
OPT = -O3 -qipa
FFLAGS=...$(OPT)...
LDFLAGS=...$(OPT)...
```

- `-qipa` works when building executables or shared objects but always compile `main` and exported functions with `-qipa`
- It is not necessary to compile everything with `-qipa` but try to apply it to as much of your program as possible

Target Machines

- `-qarch`
  - Specifies the target machine or machine family on which the generated program is expected to run successfully
  - `-qarch=ppc` targets any PowerPC (default with XLF V11.1)
  - `-qarch=pwr6` targets POWER6 specifically
  - `-qarch=auto` targets the same type of machine as the compiling machine
- `-qtune`
  - Specifies the target machine on which the generated code should run best
  - Orthogonal to `-qarch` setting but some combinations not allowed
  - `-qtune=pwr6` tunes generated code for POWER6 machines
  - `-qtune=auto` tunes generated code to run well on machines similar to the compiling machine
- `-qtune=balanced` tunes generated code to run well on POWER5 and POWER6 (Default with XLF V11.1)

Getting the most out of target machine options

- `-qarch=pwr7`
  - Utilize POWER7-specific instructions. Compiling with `-qarch=pwr7 -qtune=pwr7` should yield optimal performance on the POWER7. Note compiling with `-qarch=pwr7` will generate an executable that will only run on POWER7 or later processors.
- `-qtune=pwr7`
  - Instructs the compiler to schedule instructions for POWER7 optimization. This can be used with different –qarch options, but most commonly used with `-qarch=pwr7`
- `-qtune=balanced`
  - When used with `-qarch=pwr6` (or `pwr6x`) this option will generate a binary that runs on both POWER6 and POWER7 systems, but with scheduling improvements that should improve POWER7 performance.
- `-qfloat=norngchk`
  - This option produces faster software divide and square root sequences. It eliminates control flow in the software div/sqrt sequence by not checking for some boundary cases in input values. The optimization is used by default at `-O3` unless `-qstrict` is also specified.

The –O4 and –O5 Options

- Optimization levels 4 and 5 automatically activate several other optimization options as a package
- Optimization level 4 (`-O4`) includes:
  - `-O3`
  - `-qhot`
  - `-qipa`
  - `-qarch=auto`
  - `-qtune=auto`
  - `-qcache=auto`
- Optimization level 5 (`-O5`) includes everything from `-O4` plus:
  - `-qipa=level=2`

Compiling Code for SMP

- Use the *reentrant* compiler invocations ending in `_r` such as `xlf90_r` or `xlC_r`
- The `-qsmp` option is used to activate parallel code generation and optimization
- Specify `-qsmp=omp` to compile OpenMP code
  - `-qsmp=omp:noopt` will disable most optimizations to allow for full debugging of OpenMP programs
  - Controls are also available to change default scheduling, allow nested parallelism or safe recursive locking
  - Enables `-O2 –qhot`, disablesables `-qsmp=auto`
- Specify `-qsmp=auto` to request automatic loop parallelization
  - Disables `-qsmp=omp`
  - Use `-qsmp=omp:auto` to mix automatic loop parallelization with OpenMP

OpenMP vs. Automatic Parallelization

- OpenMP is recommended for those who are able to expend the effort of annotating their code for parallelism
  - More flexible than automatic parallelization
  - Portable
- Automatic parallelization is recommended as a means of doing some parallelization without code changes
- Automatic parallelization along with `-qreport` can be helpful for identifying parallel loop opportunities for an OpenMP programmer
- `-qsmp=threshold=n` to specify the amount of work required in a loop before the compiler considers it for automatic parallelization

Auto-vectorization

- C;
  - `-qarch=pwr7 -qtune=pwr7` -O3 -qhot -qaltivec -qsimd=auto
- Fortran:
  - -qarch=pwr7 -qtune=pwr7 -O3 -qhot -qsimd=auto

Compiler Flag Tuning Summary

- Choose the correct architecture and tuning flags
  - `-qarch=pwr7 -qtune=pwr7` (`-qarch=auto -qtune=auto`)
  - `-q64` (recommended for best parallel environment performance)
- Start with lower optimization levels and work your way up
- `-O2` ... `-O3 -qstrict` ... `-O3` ... `-O3 -qhot` ... `-O3 -qhot -qipa=level=2`
- Profile with tprof at each optimization level
  - Compare ticks on individual profile level
  - Select the best compiler option for each subroutine and source file
    - But keep an eye on overall runtime, too
- Make sure frequently called functions are properly inlined
  - If they no longer show up in the profile, that's good