Maximizing HPC Application Performance on OpenPOWER systems with IBM XL Compilers and Libraries

Yaoqing Gao, ygao@ca.ibm.com
Senior Technical Staff Member
IBM Canada Lab
Disclaimer

© Copyright IBM Corporation 2015. The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in these materials may change at any time at IBM’s sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. IBM, the IBM logo, Rational, the Rational logo, Telelogic, the Telelogic logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.

Please Note:

- IBM’s statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM’s sole discretion.
- Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision.
- The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract. The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.
Agenda

- XL Compiler Overview
- Performance Tuning with XL Compilers
  - Identify performance bottlenecks
  - Compiler optimizations
  - Most frequently used compiler options, flags, pragma and directives
  - Performance tuning tips
- Summary
Overview of XLC and XLF Compilers for OpenPOWER

▪ Common technology to implement C/C++ and Fortran
  – Target Linux and AIX on Power, Linux on Blue Gene/Q, z/OS and Linux on z Systems

▪ Advanced optimization capabilities
  – Full platform exploitation
  – Loop transformations
  – SIMDization and vectorization
  – Whole program optimization (IPA)
  – Profile Directed Feedback optimization (PDF)

▪ Language standard compliance
  – C99 Standard compliance, selected C11 features
  – C++98, nearly full C++11 compliance on selected platforms and starting C++14 compliance
  – Fortran 2003 Standard compliance, selected Fortran 2008 and TS29113 features
  – OpenMP 3.1 compliance and selected OpenMP 4.0 features

▪ GCC affinity
  – Partial source and full binary compatibility with GCC
  – Option compatibility, makefile consistency

▪ IBM Advance Toolchain support
  – Support for LE Linux on Power AT8
Why XL Compiler on Power?

SPEC benchmark leadership
- Overall 16% on SPEC2006int over GCC
- Overall 57% on SPEC2006fp over GCC
- Published leadership 1.7x on SPECint 1.9x on SPECfloat over Ivybridge on P8
- Critical in keeping Power leadership over Intel via proprietary optimizations

Key contributions to Power middleware
- Strong performance gains for major middleware & ISVs, 10-30% (e.g. DB2, SAP, Oracle)
- Mature whole-program and profile-based optimization are used in production by key middleware

Key contributions to Analytics & Technical Computing
- Delivered significant performance gains contributing to 2x goal for key Analytic workloads
- Up to 40% performance gain for ILOG CPLEX, up to 20% for SPSS
- Aggressive loop transformations for locality and memory hierarchy optimization
- Optimized OpenMP implementation

Power performance depends on XL Compiler exploitation
- SIMD, TM, and future Power features will increasingly rely on compiler optimization
- XL advanced optimization is widely used in production environments by ISVs, and in Technical Computing and HPC
The Latest Release XL C/C++ V13.1.2 and XL Fortran V15.1.2

- Support for RHEL 7.1, Ubuntu 14.04 & 14.10, SLES 12

- Integrate Clang infrastructure into the XL C/C++ compiler front end
  - Improved C/C++ language conformance
  - Improved GCC compatibility
    - support a subset of XL and gcc style options
    - Partial source and full binary compatibility
  - Expressive diagnostics

- Continue to use IBM proprietary backend with advanced optimization technology to deliver industry leading performance

- XLC integrated as the Power8 host compiler with CUDA 5.5 and 7.0 on Ubuntu 14.10 (https://developer.nvidia.com/cuda-downloads-power8)

- Fortran Language Features - TS29113 (Further C Interoperability)
  - Dummy argument of type CHARACTER with assumed length

- Macro compatibility with legacy XLC compilers
  - Ease enablement of applications built with xlC on AIX with q[no]xlcompatmacros
Identify Application Hot Spots and Performance Bottlenecks
Hot Spot and Bottleneck Detection

- Identify hot spots and detect bottlenecks
  - gather the profile information: timing, call frequency, block frequency, frequently used values,
  - performance tools: gprof, oprofile, perf, PIF, compiler instrumentation

- Identify if a workload is computation intensive, memory latency or bandwidth intensive; IO intensive by gathering performance counter information about
  - CPI breakdown
  - FPU/FXU
  - Cache misses
  - Branch mispredictions
  - LSU
  - ...
Profiling with gprof

- **gprof** is a performance analysis tool using a hybrid of instrumentation and sampling.

- **Step 1: Compile an application with XL compiler option –pg**
  - Instrumentation code is inserted into the program code during compilation to gather caller-function data at run-time.
  - `xlc -O3 -pg -o app app.c`

- **Step 2: Run the application**
  - Sampling data is saved in 'gmon.out' or in 'progname.gmon' file just before the program exits.

- **Step 3: Run gprof tool**
  - `gprof app gmon.out > analysis.txt`

- **Step 4: Analyze the profiling information**
  - Each function, who called it, whom it called, and how many times.
  - How many times each function got called, total times involved, sorted by time consumed.
Profiling with operf/oprofile

- **OProfile** is a system-wide statistical profiling tool and operf is the profiler tool provided with Oprofile.

- **Step 1: Select performance events**
  - ophelp to list available events

- **Step 2: Gather the profile information**
  - operf -e event1[,event2[,...]] where event is specified by
    event_name:sampling_rate[:unitmask[:kernel[:user]]]

- **Step 3: Analyze the profiling information**
  - opreport -l
Profiling with perf

- **Perf** is a performance tool that automatically groups events, and cycles through them every N µsecs

- **Step 1: select performance events**
  - perf list to list available events

- **Step 2: Gather the profile information**
  - perf <command>
    , where <command> = { lock, stat, sched, kmem, timechart, top, etc.}
  - Perf record –e event_name| raw_PMU_eventOperf -e event1[,event2[,...]] where event is specified by event_name:sampling_rate[:unitmask[:kernel[:user]]] | \mem:addr[:r][w][x]

- **Step 3: Analyze the profiling information**
  - perf report --source
CPI Analysis

- CPI
  - Cycles Per (completed) Instruction
- CPI Stack
  - Provides a break-down of where cycles are spent for an instruction, on average
- CPI Stall Components
  - Identifies hardware components causing processor pipeline stalls
- Usefulness of CPI Stall Components
  - Focuses performance investigation on critical areas
### POWER8 CPI Stack

**Stalled Cycles**

<table>
<thead>
<tr>
<th>Cause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall due to BR or CR</td>
<td>Stall due to Branch</td>
</tr>
<tr>
<td></td>
<td>Stall due to CR</td>
</tr>
<tr>
<td>Stall due to Fixed Point</td>
<td>Stall due to Fixed-Point long</td>
</tr>
<tr>
<td></td>
<td>Stall due to Fixed-Point (other)</td>
</tr>
<tr>
<td>Stall due to Vector/Scalar</td>
<td>Stall due to Vector long</td>
</tr>
<tr>
<td></td>
<td>Stall due to Vector (other)</td>
</tr>
<tr>
<td>Stall due to Vector</td>
<td>Stall due to Scalar long</td>
</tr>
<tr>
<td></td>
<td>Stall due to Scalar (other)</td>
</tr>
<tr>
<td>Stall due to Scalar</td>
<td>Stall due to Vector/Scalar (other)</td>
</tr>
<tr>
<td>Stall due to Load/Store</td>
<td>Stall due to L2/L3 Hit</td>
</tr>
<tr>
<td></td>
<td>Stall due to L3 Miss</td>
</tr>
<tr>
<td></td>
<td>Stall due to L2/L3 hit with conflict</td>
</tr>
<tr>
<td></td>
<td>Stall due to L2/L3 hit with no conflict</td>
</tr>
<tr>
<td>Stall due to Dcache Miss</td>
<td>Stall due to On-chip L2/L3</td>
</tr>
<tr>
<td></td>
<td>Stall due to On-chip Memory</td>
</tr>
<tr>
<td>Stall due to LSU Reject</td>
<td>Stall due to Off-chip L2/L3</td>
</tr>
<tr>
<td></td>
<td>Stall due to Off-chip Memory</td>
</tr>
<tr>
<td></td>
<td>Stall due to Off-node Memory</td>
</tr>
<tr>
<td>Stall due to Load/Store</td>
<td>Stall due to On-chip Memory</td>
</tr>
<tr>
<td>Stall due to Next-To-Complete Flush</td>
<td>Stall due to Store Finish</td>
</tr>
<tr>
<td>Stall due to Nops</td>
<td>Stall due to Load Finish</td>
</tr>
<tr>
<td>Stall Cycles (other)</td>
<td>Stall due to Store Forward</td>
</tr>
<tr>
<td></td>
<td>Stall due to Load/Store (other)</td>
</tr>
</tbody>
</table>

**Finished Group Waiting to Complete**

**Thread Blocked**

- Blocked due to LWSYNC
- Blocked due to HWSYNC
- Blocked due to ECC Delay
- Blocked due to Other Thread's Flush
- Blocked due to COQ Full
- Thread Blocked (Other)

**Nothing to Dispatch**

- Nothing to Dispatch due to Icache Miss
- Nothing to Dispatch due to Branch Mispredict
- Nothing to Dispatch due to Branch Mispredict + Icache Miss
- Nothing to Dispatch - Dispatch Held
  - Dispatch Held due to Mapper
  - Dispatch Held due to Store Queue
  - Dispatch Held due to Issue Queue
  - Dispatch Held (Other)
- Nothing to Dispatch (Other)
## Focus of Investigation

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Cause</th>
<th>Common Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSU Stalls</td>
<td>Data Cache Misses</td>
<td>- Reduce memory footprint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Optimize data layout</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Add NUMA awareness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Enable or disable prefetching</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Remove false sharing</td>
</tr>
<tr>
<td>LSU Stalls</td>
<td>Address Translation Misses (ERAT, TLB)</td>
<td>- Use large pages</td>
</tr>
<tr>
<td>FXU and VSU Stalls</td>
<td>Instructions taking too long to execute</td>
<td>- Optimize the code path to reduce instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Replace long-latency instructions (div, mult) with low-latency ones (add, shift)</td>
</tr>
<tr>
<td>Empty GCT</td>
<td>ICache Misses</td>
<td>- Code straightening</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Compile PDF or FDPR</td>
</tr>
<tr>
<td>Empty GCT</td>
<td>Branch Mispredictions</td>
<td>- Inlining, Code straightening</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Compile PDF or FDPR</td>
</tr>
</tbody>
</table>
Obtaining a Valid CPI Stack

- **Slice & Dice & Re-Assemble**
  - PMU and tool limitations force this technique
    - **Slice**: Collect data from approx 20 PMU groups on P8
    - **Dice**: Rotate through 20 PMU group every N μsecs (for speed)
    - **Re-Assemble**: Done by PowerLinux SDK to obtain CPI stack

- **Workload Requirements**
  - **Slicing**: Workload must be stable between runs
  - **Dicing**: Workload must be stable within a run
  - Workload must be homogeneous across all hw threads

- **Symptoms of Invalid CPI Stack**
  - Components with values > 100%
  - Components with values < 0%

- **Requirements of PMU Data**
  - **Across slices**: PMU values must have very low variability
  - **Across hw threads within each slice**: PMU values must have low variability

- **Achieving Valid CPI Stack**
  - Use static core frequency because PMU events assume constant M cycles per second
  - Transactional Memory may need to be disabled
  - Unnecessary hypervisor/OS services/daemons should be disabled
Perf with PIF

- For stability, may need to pin application to a specific chip, core, hw thread
  numactl --physcpubind=0 --membind=0 <command_to_run>

- Run following Linux perf tool command to collect data
  perf stat -a -A -e
  r1001C,r2001A,r30004,r4000A,r10036,r2C010,r30006,r4C010,r1E054,r2C012,r30026,r4C012,r100F8,r2C014,r30028,r4C014,r10006,r2C016,r30036,r4C016,r10018,r2C018,r30038,r4C018,r1001A,r2C01A,r30004,r4C01A,r1002C,r2C01C,r30008,r4C01C,r10130,r2D010,r3000C,r4D010,r10134,r2D012,r30012,r4D012,r1C040,r2D014,r3001C,r4D014,r1C042,r2D016,r300F0,r4D016,r1C044,r2D018,r300F6,r4D018,r1E052,r2D01A,r30068,r4D01A,r1E05A,r2D01C,r3E05E,r4D01C,r1E05C,r2D01E,r3E050,r4D01E,r1E15E,r2E01E,r3E052,r4E010,r10066,r200FD,r30066,r40012,r10068,r20016,r3E054,r40132,r1006A,r201E2,r34040,r4013E,r1006C,r2003E,r3012A,r4E052,r100EE,r200F6,r30130,r400F8,r2E010,r500fa,r600f4 -o cpi_events.perf <command_to_run>

- Run following command to process data and generate CPI stack (.html output)
  pif2.0/process_p8_perfstat.sh cpi_events.perf <command_to_run>
Compiler Optimization: Basic and Advanced Optimization
Optimization Capabilities

- **Platform exploitation**
  - `qarch`: ISA exploitation
  - `qtune`: skew performance tuning for specific processor, including `tune=balanced`
  - Large library of compiler builtins and performance annotations

- **Mature compiler optimization technology**
  - Five distinct optimization packages
  - Debug support and assembly listings available at all optimization levels
  - Whole program optimization
  - Profile-directed optimization
Summary of Optimization Levels

- **Noopt,-O0**
  - Quick local optimizations
  - Keep the semantics of a program (-qstrict)

- **-O2**
  - Optimizations for the best combination of compile speed and runtime performance
  - Keep the semantics of a program (-qstrict)

- **-O3**
  - Equivalent to –O3 –qhot=level=0 –qnostrict
  - Focus on runtime performance at the expense of compilation time: loop transformations, dataflow analysis
  - May alter the semantics of a program (-qnostrict)

- **-O3 –qhot**
  - Equivalent to –O3 –qhot=level=1 –qnostrict
  - Perform aggressive loop transformations and dataflow analysis at the expense of compilation time

- **-O4**
  - Equivalent to –O3 –qhot=level=1 –qipa=level=1 -qnostrict
  - Aggressive optimization: whole program optimization; aggressive dataflow analysis and loop transformations

- **-O5**
  - Equivalent to –O3 –qhot=level=1 –qipa=level=2 -qnostrict
  - More aggressive optimization: more aggressive whole program optimization, more precise dataflow analysis and loop transformations
Basic Optimization Techniques

- **Inlining**
  - Replaces a call to a procedure by a copy of the procedure itself. It is done to eliminate the overhead of calling the function, and also to allow specialization of the function for the specific call point.

- **Redundancy detection**
  - Identify computations that are redundant or partially redundant with values previously computed, so their value can be reused rather than recomputed.

- **Platform exploitation**
  - Use a model of the target processor to determine the best mix of instructions to use to implement a certain program sequence.

- **Flow restructuring**
  - Reorganize the code to increase the density of the hot code or to make it less frequent for conditional branches to be taken.
Loop Optimization

- Analyze and transform loops to improve runtime performance
  - Analyze memory access patterns to improve cache utilization
  - Tailor instruction schedule for specific loop and target processor
  - Interleave execution of multiple loop iterations

- Most effective on numerical applications, e.g. analytics, technical computing
  - Depends on loops with regular behavior that can be analyzed and restructured by the optimizer

- Enabled at O3 and above. Aggressive loop optimization with –O3 -qhot
SIMDization/Vectorization

- Supports data types of INTEGER, UNSIGNED, REAL and COMPLEX
- Explicit SIMD programming with –qaltivec (=BE|LE)
- Automatic SIMDization at –O3 –qhot
  - Basic block level SIMDization
  - Loop level aggregation
  - Data conversion
  - Reduction
  - Loop with limited control flow
  - Math SIMDization
  - Partial Loop Vectorization
  - Alignment Handling
MASS and MASSV Libraries

- Libraries of mathematical routines tuned for optimal performance on various POWER architectures
  - General implementation tuned for POWER
  - Specific implementations tuned for specific POWER processors

- Compiler will automatically insert calls to MASS/MASSV routines at higher optimization levels
  - Users can add explicit calls to the library

```
for (i=0;i<n;i++) {
    b[i]=sqrt(a[i]);
}
```

Transformation report

- Loop vectorization was performed.

```
__vsqrt_P8(b,a,n);
```
Parallelization

- **User-driven parallelism**
  - All optimization levels interoperate with POSIX Threads implementation
  - Full OpenMP 3.1 implementation provides simple mechanism to write parallel applications
    - Based on pragmas/annotations on top of sequential code
    - Industry specification, developed by OpenMP consortium (www.openmp.org)

- **Compiler-driven parallelism**
  - Mechanism for the compiler to automatically identify and exploit data parallelism
  - Identify parallelizable loops, performing independent operations on arrays or vectors
    - Best results on loop-intensive, compute-intensive workloads
    - Aided by program annotations, fully interoperable with OpenMP
Inter-Procedural Analysis (IPA)

- Optimize the whole program at module scope
  - Intercept the linker and re-optimize the program at module scope

- Three levels of aggressiveness (-qipa=level=0/1/2)
  - Balance between aggressive optimization and longer optimization time

- Enables additional program optimization
  - Cross-file inlining (including cross-language)
  - Global code placement based on call affinity
  - Global data reorganization

- Reduction in TOC pressure, through data coalescing
Profile-Directed Optimization (PDF)

- Collect program statistics on training run to use on subsequent optimization phase
  - Minor impact on execution time of instrumented program (10% - 50%)
  - Static program information: Call frequencies, basic block execution counts
  - Value profiling: collect histogram of values for expressions of interest
  - Hardware counter information (optional)

- Supports multiple training runs and parallel instances of the program
  - Profiling information from multiple training runs aggregated into single file
  - Locking used to avoid clobbering of the profiling data on file

- Integrated with IPA process (implies ipa=level=0)
  - PDF synchronization point at beginning of link-time optimization phase
  - No need to recompile source files for PDF2, only relink with qpdf2 option

- Tolerates program changes between instrumentation/optimization
  - Compiler skips profile-based optimization for any modified functions
  - Shows an estimate of the relevance of the profiling data
Debugging Optimized Code

- **Debug levels**
  - Tradeoff between compiler optimization and debug transparency
  - Compiler optimizations hide program state from the debugger
    - Users have to choose between full debug at no-opt, or marginal debug at full opt

- **Compiler to provide control over tradeoffs between optimization and debug**
  - Debug levels: -g0 to -g9
    - -g1 minimal debug to maintain full performance
    - -g2 the default to provide maximal performance with unreliable debug
    - -g8 preserves most performance and allows examination of program state through debugger
    - -g9 provides full debug capability, at runtime performance cost
      - Expect better runtime performance from -g9 -O2 than -g -O0
Performance Tuning Tips
Compiler Options, Flags, Directives and Pragmas

- **Frequently used compiler option sets**
  - `-O3` – `qhot` for floating point computation intensive code (e.g. technical computing, Analytics)
  - `-O3` – `qipa` for integer code (e.g., commercial workload)

- **Frequently used compiler directives/pragmas**
  - **Dependency**
    - `#pragma ibm independent_loop`
  - **Frequency**
    - `#pragma execution_frequency`
  - **Alignment**
    - `__alignx`
    - `__attribute__((aligned(16)))`
  - **SIMDization**
    - `#pragma nosimd`
    - `#pragma simd_level`
  - **Unroll**
    - `#pragma unroll`
XML Compiler Transformation Reports

- Generate compilation reports consumable by other tools
  - Enable better visualization and analysis of compiler information
  - Help users do manual performance tuning
  - Help automatic performance tuning through performance tool integration

- Unified report from all compiler subcomponents and analysis
  - Compiler options
  - Pseudo-sources
  - Compiler transformations, including missed opportunities

- Consistent support among Fortran, C/C++

- Controlled under option
  - `-qlistfmt=xml html=inlines` generates inlining information
  - `-qlistfmt=xml html=transform` generates loop transformation information
  - `-qlistfmt=xml html=data` generates data reorganization information
  - `-qlistfmt=xml html=pdf` generates dynamic profiling information
  - `-qlistfmt=xml html=all` turns on all optimization content
  - `-qlistfmt=xml html=none` turns off all optimization content
Performance Tuning with Compiler Transformation Reports

file.c

foo (float *p,  
     float *q,  
     float *r,  
     int n) {
    for (int i=0; i < n; i++) {
        p[i] = p[i] + q[i]*r[i];
    }
}

file.xml

Loop was not SIMD vectorized because a data dependence prevents SIMD vectorization

file.xml

Loop was SIMD vectorized

Tuning

file.c

foo (float * restrict p,  
     float * restrict q,  
     float * restrict r,  
     int n) {
    for (int i=0; i < n; i++) {
        p[i] = p[i] + q[i]*r[i];
    }
}
**SIMDization Tuning**

**Transformation report**

- Loop was SIMD vectorized
- It is not profitable to vectorize
- Data dependence prevents SIMD vectorization
- Memory accesses have non-vectorizable alignment.

**User actions**

- Use `#pragma simd_level(10)` to force the compiler to do SIMDization
- Use fewer pointers when possible
- Use `#pragma ibm independent_loop` if it has no loop carried dependency
- Use `restrict` keyword
- Use `__attribute__((aligned(n)))` to set data alignment
- Use `__alignx(16, a)` to indicate the data alignment to the compiler
- Use array references instead of pointers where possible
SIMDization Tuning

Transformation report

- Loop structure prevents SIMD vectorization

User actions

- Convert while-loops into do-loops when possible
- Limited use of control flow in a loop
- Use MIN, MAX instead of if-then-else
- Eliminate function calls in a loop through inlining

- Loop interchange for stride-one accesses, when possible
- Data layout reshape for stride-one accesses
- Higher optimization to propagate compile known stride information
- Stride versioning

- Do statement splitting and loop splitting

memory accesses have non-vectorizable strides

either operation or data type is not suitable for SIMD vectorization.
Compiler Friendly Code

- **Compiler must be conservative when determining potential side effects**
  - Procedure calls may access or modify any visible variables
  - Accesses through pointers may modify any visible variables

- **Pessimistic side effect analysis prevents compiler optimizations**
  - Must re-compute expressions with operands which may have been modified
  - Must compute values that otherwise might be unneeded

- **Help the compiler identify side effects to improve application performance**
  - Use suitable optimization levels
  - Include appropriate header files for any system routines in use
  - Use local variables to maintain values of global variables across function calls or pointer dereferences
  - Avoid using global variables when local variables are suitable
  - Avoid reusing local variables for unrelated purposes
  - Follow ANSI C/C++ language pointer aliasing rules
    - An object of a certain data type can only be accessed through a pointer of the same (or compatible) data type
Compiler Friendly Code

- Use restrict keyword (XLC supports multiple level and scope restricted pointer) or compiler directives/pragmas to help the compiler do dependence and alias analysis
- Use “const” for globals, parameters and functions whenever possible
- Group frequently used functions into the same file (compilation unit) to expose compiler optimization opportunity (e.g., intra compilation unit inlining, instruction cache utilization)
- Limit exception handling
- Excessive hand-optimization such as unrolling may impede the compiler
- Keep array index expressions as simple as possible for easy dependency analysis
- Consider using the highly tuned MASS and ESSL libraries
Performance Tuning Tips

- Make use of visibility attribute
  - Load time improvement
  - Better code with PLT overhead reduction
  - Code size reduction
  - Symbol collision avoidance

- Inline tuning
  - Call overhead reduction
  - Load-hit-store avoidance

- Whole program optimization by IPA
  - Across-file inlining
  - Code partitioning
  - Data reorganization
  - TOC pressure reduction
Architecture and System Specific Tuning Tips

**System configuration**
- Adjust SMT level: `ppc64_cpu --smt=<level>`
- Adjust hardware prefetch aggressiveness: `ppc64_cpu --dscr=<value>`
- Adjust cpu/memory affinity: `numactl <flags>`
- Set huge pages: `sysctl -w vm.nr_hugepages=<number>`

**POWER8 exploitation**
- POWER8 specific ISA exploitation under `--qarch=pwr8`
- Scheduling and instruction selection under `--qtune=pwr8:SMTn (n=1, 2, 4, 8)`

**Automatic SIMDization at O3 –qhot**
- Limited use of control flow
- Limited use of pointers. Use `independent_loop` directive to tell the compiler a loop has no loop carried dependency; use either `restrict` keyword or `disjoint pragma` to tell the compiler the references do not share the same physical storage whenever possible
- Limited use of stride accesses. Expose stride-one accesses whenever possible

**Data prefetch**
- Automatic data prefetch at O3 –qhot or above.
- `-qprefetch=dscr=N` to control hardware prefetch aggressiveness
Floating-point Computation Control

- Aggressive optimization may affect the results of the program
  - Precision of floating-point computation
  - Handling of special cases of IEEE FP standard (INF, NAN, etc)
  - Use of alternate math libraries

- `qstrict` guarantees identical result to `noopt`, at the expense of optimization
  - Suboptions allow fine-grain control over this guarantee
  - Examples:
    - `qstrict=precision`  Strict FP precision
    - `qstrict=exceptions`  Strict FP exceptions
    - `qstrict=ieeefp`     Strict IEEE FP implementation
    - `qstrict=nans`       Strict general and computation of NANs
    - `qstrict=order`      Do not modify evaluation order
    - `qstrict=vectorprecision`  Maintain precision over all loop iterations

- Can be combined: `-qstrict=precision:nonans`
Summary

This presentation addresses:
- What are frequently used XL compiler options
- What are frequently used XL compiler directives/pragmas
- How to identify program hot spots and detect performance bottlenecks with XL compilers and performance tools
- How to write compiler-friendly code for better performance
- How to do performance tuning with XL compilers and libraries
- How to do POWER8 specific optimization
Thank You
Additional information

- XL C/C++ home page

- C/C++ Café
  http://ibm.biz/Bdx8XR

- XL Fortran home page

- Fortran Café
  http://ibm.biz/Bdx8XX

- IBM SDK Linux – Using the CPI plugin

- PMU events
Compile Application with XLC and XLF

- Get and Install XL C/C++ and Fortran compilers

- Set up the path for IBM XL compilers
  export PATH=/opt/ibm/xlC/13.1.2/bin:/opt/ibm/xlf/15.1.2/bin:$PATH

- Check the compiler release and version
  xlc –qversion
  xlf –qversion

- Compile an application
  xlc for C (add –qlanglvl=stdc11, –qlanglvl=extc1x to enable C11 );
  xlC for C++ (add –qlanglvl=extended0x to enable C++11)
  xlf, xlf90, xlf95, xlf2003, xlf2008 for Fortran

- Specify compile options
  - -O3 –qhot for floating point computation intensive application
  - -O3 –qipa for integer application
  - add –qsmp=omp for OpenMP application
GPU Programming with CUDA C/C++ and XL C/C++

- Check CUDA-capable GPU
  `lspci | grep -i nvidia`

- Verify Linux version
  `uname -m && cat /etc/*release`

- Get and Install CUDA C/C++ and XL C/C++ and Fortran compilers
  CUDA C/C++: https://developer.nvidia.com/cuda-downloads-power8

- Modify host_config.h for XL C/C++
  Change "!=" to ">=" and remove the irrelevant comment.

- Set up the path for IBM XL compilers
  `export PATH=/opt/ibm/xlC/13.1.2/bin:$PATH`

- Environment Setup
  `export PATH=/usr/local/cuda-5.5/bin:$PATH`
  `export LD_LIBRARY_PATH=/usr/local/cuda-5.5/lib64:$LD_LIBRARY_PATH`

- Compile and run
  `nvcc -ccbin xlC -m64 -Xcompiler -O3 -Xcompiler -q64 -Xcompiler -qsmp=omp -gencode arch=compute_20,code=sm_20 -o cudaOpenMP.o -c cudaOpenMP.cu`
  `nvcc -ccbin xlC -m64 -Xcompiler -O3 -Xcompiler -q64 -o cudaOpenMP cudaOpenMP.o -lxlsmp`
Compile and Run You Application

- **Compile single thread code**
  
  `xlC -O3 -qhot`

- **Compile OpenMP code**
  
  `xlC -O3 -qhot -qsmp=omp`

- **Run single thread code**
  
  Your_application

- **Display OpenMP runtime settings**
  
  Export `OMP_DISPLAY_ENV=TRUE`

- **Run OpenMP code with OpenMP environment setting**
  
  export `OMP_WAIT_POLICY=ACTIVE`
  export `OMP_STACKSIZE=8000000B`
  export `OMP_SCHEDULE=static`
  export `OMP_PROC_BIND=TRUE`
  export `OMP_DYNAMIC=FALSE`
  your_application