IBM Education Assistance for z/OS V2R2

Item: XL C/C++ Enhancements
Element/Component: XL C/C++

Material current as of May 2015

Agenda

- Trademarks
- Presentation Objectives
- Overview
- Usage & Invocation
- Interactions & Dependencies
- Migration & Coexistence Considerations
- Installation
- Presentation Summary
- Appendix

Trademarks

- See url http://www.ibm.com/legal/copytrade.shtml for a list of trademarks

Presentation Objectives

- Discuss the major new enhancements to the C and C++ compilers in the following areas:
  - Usability:
    - -M enhancements for better dependency file generation
    - Inline assembly
    - Metal C enhancements
    -
  - Performance:
    - MASS and ATLAS libraries
    - Hardware exploitation with new ARCH/TUNE
    - Architecture sections
    - SIMD – Vector programming support and auto-SIMD
    -
  - Debugging:
    - Capture all source
    - Non-XPLINK CDA runtime

USABILITY ENHANCEMENTS

- -M enhancements for better dependency file generation
- Inline assembly
- Metal C enhancements

Overview: -M Enhancements

- Problem Statement / Need Addressed
  - 'make' dependency file generation through -M did not include non-existent headers or allow named targets
    - Ex. Non-existent headers can be ones generated later in the build
    - Ex. Output targets can be for source reuse
- Solution
  - Add in -MT, -MQ and -MG, and -qmakedep=gcc/pponly options
- Benefit / Value
  - MT allows setting the dependent target name
  - MG allows missing header files to be included in the dependency list
  - MQ is MT but also escapes 'make' special characters for easier dependency file usage
  - The pponly suboption does not generate objects allowing a dependency file generation only compilation

Usage & Invocation: -M Enhancements

- The M flags are only available on USS as they generate dependency files suitable for 'make'

```
> xlc -M -MT t1.o -MT t2.o t.c
t1.o t2.o : t.c
t1.o t2.o : t1.h
t1.o t2.o : t2.h
```

- If used with -qmakedep=gcc or -qmakedep=pponly all targets appear on a single line containing all dependencies

```
> xlc -M -MT t1.o -MT t2.o -qmakedep=gcc t.c
t1.o t2.o : t.c \
t1.h \
t2.h
```

- Special characters can be automatically escaped with -MQ

```
> xlc -MQ '$(prefix)t.o' -qmakedep=gcc t.c
$$(prefix)t.o : t.c \
t1.h \
t2.h
```

Interactions & Dependencies: -M Enhancements

- This feature is only available for the xlc utility

Migration & Coexistence Considerations: -M Enhancements

- None
  - Existing clean xlc invocations have the same behavior
  - Using the new options is the only way to change the behavior

Overview: Inline Assembly

- Problem Statement / Need Addressed
  - Running HLASM statements in LE enabled C or C++ programs was not possible without function level separation and multiple compiles
    - Metal C does not have the full LE so it is a restricted solution for this problem
- Solution
  - Allow HLASM statements in C and C++ code
- Benefit / Value
  - Like the GCC inline assembly feature, this allows specialized HLASM code to be inserted into C and C++ code
    - Use specialized instructions not normally generated by C/C++
  - Use the rich C/C++ libraries and functionality with assembler programs

Usage & Invocation: Inline Assembly

- New options/sub-options: ASM, KEYWORD(ASM), ASMLIB
  - ASM: Causes `__asm` and `__asm__` to be asm statements in the same way as Metal C
  - KEYWORD(ASM): Gives `asm` the same semantics as `__asm`
  - ASMLIB: Specifies the macro libraries to be used when assembling the inline assembler source code
    - Ex. -qasmlib=A -qasmlib=B will result in the following ASMLIB DD allocation:

      ```
      //ASMLIB DD DISP=SHR,DSN=A
      //       DD DISP=SHR,DSN=B
      ```

- Assembler messages will be reported by the compiler under compiler message CCN1148

Usage & Invocation: Inline Assembly

- Example JCL invocation:

```
//jobname JOB acctno,name...
//COMPILE EXEC PGM=CCNDRVR,
// PARM='/SEARCH(''CEE.SCEEH.+'') NOOPT SO OBJ ASM KEYWORD(ASM) ASMLIB(//SYS1.MACLIB)'
//STEPLIB DD DSNAME=CEE.SCEERUN,DISP=SHR
// DD DSNAME=CEE.SCEERUN2,DISP=SHR
// DD DSNAME=CBC.SCCNCMP,DISP=SHR
// DD DSNAME=SYS1.SASMMOD1,DISP=SHR
//SYSLIN DD DSNAME=MYID.MYPROG.OBJ(MEMBER),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN DD DATA,DLM=@@
#include <stdio.h>
   ...
int main(void) {
/* comment */
  int x=5, y=1;
  asm(" AR %0,%1\n":"+r"(x):"r"(y):);
}
@@
//SYSUT1 DD DSN=...
   ...
//*
```

Interactions & Dependencies: Inline Assembly

- Software Dependencies
  - z/OS V2R1 High Level Assembler with APAR PM79901, or later
  - High Level Assembler library SASMMOD1 is included in STEPLIB concatenation of the compiler step if any of the assembler macros in there are used

Migration & Coexistence Considerations: Inline Assembly

- None
  - Any existing usage of the `__asm` and `__asm__` keywords in source code is already invalid as the identifiers are reserved
  - KEYWORD(ASM) is not the default. It must be explicitly specified so existing code using the `asm` keyword will not break.
  - The feature is only active if the ASM option is used

Overview: DSAUSER Enhancements

- Problem Statement / Need Addressed
  - The existing DSAUSER feature only provides space for a pointer which is good for new code, but not ideal for code expecting larger storage when ported from other compilers
- Solution
  - Reserve a user described larger amount of space
- Benefit / Value
  - Easier porting of code
  - Easier usage of specialized linkage

Usage & Invocation: DSAUSER Enhancements

- The DSAUSER option is now enhanced to accept a suboption
  - The suboption specifies the number of words to reserve
  -
- Ex. xlc -qdsauser=12
  - Reserves 48 bytes instead of the default of 4 (default of 8 in 64-bit mode) if only -qdsauser was specified

Interactions & Dependencies: DSAUSER Enhancements

- Software Dependencies
  - Only applicable to Metal C

Migration & Coexistence Considerations: DSAUSER Enhancements

- None
  - The default values are preserved if no sub-option is specified.

Overview: Redefining macro (C++ only)

- Problem Statement / Need Addressed
  - The C++ compiler does not allow redefining a macro to a different value
    - Generates an error which prevents successful compilation
- Solution
  - Introduce a compiler option LANGLVL(REDEFMAC) to turn the error into the warning to allow different value of already defined macro
  - The C compiler already allows redefinition of a macro by default
- Benefit / Value
  - The new option will allow to compile the source with `#define` directive which redefines macro specified with -D compiler option

Usage & Invocation: Redefining macro (C++ only)

- Example of macro redefinition with a different value

```
#define _XOPEN_SOURCE 600

int main() { return 0; }
```

- Invocation:

```
xlC -Wc,'LANGLVL(REDEFMAC)' -D_XOPEN_SOURCE=500 -+ t.c

CCN5848 (W) The macro name "_XOPEN_SOURCE" is already defined with a different
definition.
```

- The value of the macro will be `600` and the definition specified on -D option will be ignored with a warning
- Other source without `#define` will have the macro value of 500

Interactions & Dependencies: Redefining macro (C++ only)

- This option is available in C++ compiler only
  - The C compiler generates warning message by default

Overview: Operator new function calls optimization

- Problem Statement / Need Addressed
  - The C++11 ANSI Standard requires checking the pointer returned from placement operator new and new[], and performs the initialization only when the pointer is not null
  - The check for null pointer returned from other operators new and new[] is not required but performed
  - Both null pointer checks hurt run-time performance
- Solution
  - New compiler options `LANGLVL(NOCHECKPLACEMENTNEW)` and `NOCHECKNEW` can be use to speed the program by eliminating null pointer checks
- Benefit / Value
  - Run-time performance should improve when recompiling with `LANGLVL(NOCHECKPLACEMENTNEW)`

Usage & Invocation: Operator new function calls optimization

- Example of placement new

```
#include <new>

struct A {};

void construct(char* buf) {

 A* p1 = (A*) new(buf) A();

 A* p2 = new A;

}
```

- Compiling with `LANGLVL(NOCHECKPLACEMENTNEW)` will remove the check of return pointer from placement operator new
- The `LANGLVL([NO]CHECKPLACEMENTNEW)` compiler option is available in V2R1M1 and V2R2

Usage & Invocation: Operator new function calls optimization

- Example of placement new

```
#include <new>

void * operator new[](__typeof__ (sizeof 0)); // define your operator new

struct A {

A() { x = 0;}

int x; };

int main() { new A[2]; }
```

- Compiling with V2R2 the check of return pointer from operator new[] will be eliminated
  - The option `[NO]CHECKNEW` is available in V2R2 only
- The default is `NOCHECKNEW` and the option applies only to throwing versions of operator new and new[]

Migration & Coexistence Considerations: Operator new function calls optimization

- If your program redefines operator new or new[] and it can throw an exception, make sure it does not return null pointer
  - Otherwise, you might get exception at run-time
- Example:

```
void * operator new[](__typeof__ (sizeof 0)) { return 0; }
```

- Solution:
  - Compile with `CHECKNEW` option
  - Change definition of operator new so it does throw rather then return 0
  - Add `throw()` exception specification to make a non-throw version of operator new

Overview: Name Mangling change

- Problem Statement / Need Addressed
  - Equivalent Typeid symbols do not compare equal. See Example 1.
  - Unnamed namespace does not include a path of the source causing ambiguous signatures. See Example 2.
- Solution
  - The cv-qualification in function parameters in typeid will not be part of the signature
  - Encoding file directory along with the file name will disambiguate symbols residing in unnamed namespaces in the sources having the same name but in different directories
- Benefit / Value
  - Programs will be able to use typeid and multiple source files having the same name and using unnamed namespaces

Usage & Invocation: Name Mangling change

- Example 1: typeid

```
#include <typeinfo>
#include <stdio.h>
int main() {
if (typeid(void (*)(int)) == typeid(void (*)(const int))) printf("success\n");
else printf("failure\n");
return 0; }
```

- Compiling with `NAMEMANGLING(zOSV2R1M1_ANSI)` will output `success`

Usage & Invocation: Name Mangling change

- Example 2: unnamed namespace

```
#include <stdio.h>

namespace {

 struct C {

  C(){ printf("%s\n", "C() of c1"); }

 }c1;

}
```

- Compiling with `NAMEMANGLING(zOSV2R1M1_ANSI)` 1/test.cpp and 2/test.cpp will generate `__ct__Q2_10`**`1`**`_test_cpp1CFv` and `__ct__Q2_10`**`2`**`_test_cpp1CFv` symbols respectively
  - Note: Source file paths need to be present in the input file specification as part of the compilation invocation command for this to work

Migration & Coexistence Considerations: Name Mangling change

- The ANSI suboption is equivalent to the latest level of name mangling scheme implemented in a given release
  - For V2R1: `NAMEMANGLING(ANSI)` == `NAMEMANGLING(zOSV2R1_ANSI)`
  - For V2R1M1: `NAMEMANGLING(ANSI)` == `NAMEMANGLING(zOSV2R1M1_ANSI)`
- All source files should be recompiled with the desired sub-option or compile only new source files with sub-option equivalent to sub-option used for old objects
  - If old objects were generated with `NAMEMANGLING(zOSV2R1_ANSI)` using the V2R1 compiler, new objects should be generated with `NAMEMANGLING(zOSV2R1_ANSI)` when V2R1M1 or V2R2 compiler is used
- Note: In V2R2 we did not introduce any new sub-option

Overview: C89 enhancements

- Problem Statement / Need Addressed
  - Lower case characters specified on the SYMTRACE or EP options are automatically converted to upper case
  - Option values are truncated if they are longer then 9 characters or contain '_'
  - C89 does not pass any environment variables to the binder
- Solution
  - Allow '_' and mixed case character names longer then 9 characters
  - Enable c89 to pass environment variables to binder
- Benefit / Value
  - The new enhancements will improve trace capability of the binder via SYMTRACE binder option or using IEWBIND_OPTIONS environment variable

Usage & Invocation: C89 enhancements

- Example how to use IEWBIND_OPTIONS

```
int one_TWO_three() { return 1+2+3; }
```

- Invocation

```
1)export IEWBIND_OPTIONS="SYMTRACE=one_TWO_three"

2)/c390/archive/zosdev/latest/util/bin/c89 -Wl,'MSGLEVEL=0' t.c >o 2>&1

3)grep one_TWO_three o

 IEW2420I A61B SYMTRACE: SYMBOL one_TWO_three IS DEFINED IN SECTION $PRIV000010

 IEW2422I A61D SYMTRACE: SYMBOL one_TWO_three DEFINITION ORIGINALLY CAME FROM
```

- Other environment variables which binder responds to can be passed in the same way as shown above
- These enhancements were done in V2R2

PERFORMANCE ENHANCEMENTS

- MASS and ATLAS libraries
- Hardware exploitation with new ARCH/TUNE
- Architecture sections
- SIMD – Vector programming support and auto-SIMD

Overview: MASS library

- Problem Statement / Need Addressed
  - Elementary/special functions (e.g. exp, log, sin, etc.) are an important class of common mathematical functions but not always fast
    - Often heavily used and performance critical in numerical applications such as business analytics
- Solution
  - MASS (Mathematical Acceleration Sub-System) provides comprehensive libraries of scalar, vector, and SIMD functions, tuned for high performance on zEC12/zBC12 and z13 processors
- Benefit / Value
  - Extensive set of functions, both single- and double-precision
    - 59 scalar, 77 vector, 8 SIMD
  - Significant performance gains
    - Up to 6.8x avg speedup for key z13 vector MASS functions vs. zEC12 runtime library

Overview: MASS library

- MASS provides 3 kinds of libraries
  - Scalar
    - e.g. double exp (double in)
    - Easiest to use in existing code since names match existing runtime library functions
  - Vector
    - e.g. void vexp (double out[], double in[], int *vector_length)
    - Generally provides the highest performance, provided vector_length is sufficient (approximately >2 to >10 depending on the function)
  - SIMD
    - e.g. vector double expd2 (vector double in)
    - z13 only
    - Convenient for code written to use z13 vector datatypes and built-in functions

Usage & Invocation: MASS library

- To compile a program that calls MASS functions, use the following compiler options
  - FLOAT(IEEE)
  - ARCHITECTURE(10) - the minimum required ARCH level
  - ARCHITECTURE(11) - required if you use any MASS SIMD functions
  - VECTOR - required if you use any MASS SIMD functions
  - NOEXH - required for C++ applications only
- Do not change the rounding mode from the default value of ROUND(N)
- Include the appropriate header file(s) in the calling program
  - If using scalar MASS, include both math.h and mass.h
  - If using vector MASS, include massv.h
  - If using SIMD MASS, include mass_simd.h

Usage & Invocation: MASS library

- Linking MASS in USS
  - For scalar MASS, use
    - -l mass.arch10 for zEC12/zBC12
    - -l mass.arch11 for z13
  - For vector MASS, use
    - -l massv.arch10 for zEC12/zBC12
    - -l massv.arch11 for z13
  - For SIMD MASS, use
    - -l mass_simd.arch11 for z13
- Example of linking all MASS libraries when compiling for z13
  - xlc main.c -qarch=11 -qfloat=ieee -qvector -l mass.arch11 -l massv.arch11 -l mass_simd.arch11

Usage & Invocation: MASS library

- Linking MASS in MVS batch mode
- Prepend appropriate MASS library dataset(s) to SYSLIB concatenation
  -
  - For zEC12/zBC12
    - CBC.SCCNM10 for MASS functions in math.h
    - CBC.SCCNN10 for all other MASS functions
    -
  - For z13
    - CBC.SCCNM11 for MASS functions in math.h
    - CBC.SCCNN11 for all other MASS functions

Usage & Invocation: MASS library

- Linking MASS in MVS batch mode – Examples
- zEC12/zBC12, 31-bit mode

```
//SYSLIB   DD DSN=CBC.SCCNM10,DISP=SHR
//         DD DSN=CBC.SCCNN10,DISP=SHR
//         DD DSN=CEE.SCEELKEX,DISP=SHR
//         DD DSN=CEE.SCEELKED,DISP=SHR
//         DD DSN=CBC.SCCNOBJ,DISP=SHR
```

–

- z13, XPLINK or LP64 mode

```
//SYSLIB   DD DSN=CBC.SCCNM11,DISP=SHR
//         DD DSN=CBC.SCCNN11,DISP=SHR
//         DD DSN=CEE.SCEEBND2,DISP=SHR
//         DD DSN=CBC.SCCNOBJ,DISP=SHR
```

Usage & Invocation: MASS library

MASS calling program and USS compile examples

MASS scalar function example: rsqrt for zEC12/zBC12

```
// Compile command: xlc -qARCH=10 -qFLOAT=IEEE -c sample1.c
#include <math.h>
#include <mass.h>  // rsqrt is declared in <mass.h>, not <math.h>
int main(void) {
  double input = 16;
  double output;
   output = rsqrt(input);
   // Code to use the results in output goes here
}
```

Usage & Invocation: MASS library

MASS scalar function example: pow for zEC12/zBC12

```
// Compile command: xlc -qARCH=10 -qFLOAT=IEEE -c sample2.c
#include <math.h> // pow is declared in <math.h>, not <mass.h>
#include <mass.h>
int main(void) {
  double base = 3;
  double exponent = 2;
  double output;
  output = pow (base, exponent);
  // Code to use the results in output goes here
}
```

Usage & Invocation: MASS library

MASS vector function example: vlog2 for z13

```
// Compile command: xlc -qARCH=11 -qFLOAT=IEEE -c sample3.c
#include <massv.h>
int main(void) {
  int size = 1000;
  double input[size];
  double output[size];
  input[0] = 8;
  input[1] = 16;
  ...
  input[999] = 42;
  vlog2 (output, input, &size);
  // Code to use the results in output[] goes here
}
```

Usage & Invocation: MASS library

MASS SIMD function example: powd2 for z13

```
// Compile command: xlc -qARCH=11 -qFLOAT=IEEE -qVECTOR -c sample4.c

#include <mass_simd.h>

int main(void) {

  vector double bases = {0, 1};

  vector double exponents = {2, 2};

  vector double output;

  output = powd2 (bases, exponents);

  // Code to use the results in output goes here

}
```

Usage & Invocation: MASS library

MASS scalar/vector function example: pow, rsqrt, vlog for zEC12/zBC12

```
// Compile command:      xlc -qARCH=10 -qFLOAT=IEEE -c sample5.c
#include <math.h>    // This includes the prototype for pow
#include <mass.h>    // This includes the prototype for rsqrt
#include <massv.h>   // This includes the prototype for vlog2
int main(void) {
  int size = 1000;
  double input[size], result[size];
  int i;
  for (i = 0; i < size; i++) {
    input[i] = i;  // initialize input vector
  }
  vlog (result, input, &size);
  double output = pow (result[27], result[525]);
  output = rsqrt(output);
  // Code to use the results in output goes here
}
```

Interactions & Dependencies: MASS library

- Software Dependencies
  - None
- Hardware Dependencies
  - The arch10 MASS libraries are tuned for zEC12/zBC12 HW, and can run on zEC12/zBC12 or z13 HW
  - The arch11 MASS libraries are tuned for z13, and run on z13 HW only

Migration & Coexistence Considerations: MASS library

- Use of MASS is optional
  - Do not include MASS headers
  - Do not link MASS libraries (USS) or prepend MASS datasets to SYSLIB concatenation (MVS)

Overview: ATLAS library

- Problem Statement / Need Addressed
  - Linear algebra is a performance-critical part of many numerical applications in fields such as business analytics
- Solution
  - ATLAS (automatically tuned linear algebra software) provides high-performance versions of all the BLAS (basic linear algebra subprograms) routines, and a subset of the LAPACK (linear algebra package) routines
- Benefit / Value
  - Tuned for high performance on zEC12/zBC12 and z13 processors
    - z13 ATLAS up to 44% throughput improvement vs zEC12/zBC12
    - Up to 88% on key SIMD-accelerated routines
  - Single and multi-threaded versions

Usage & Invocation: ATLAS library

- ATLAS is provided on z/OS for USS only
- Supplied libraries
  - ATLAS main libraries
    - ATLAS specific variants of the BLAS, CBLAS, and LAPACK routines
  - CBLAS libraries
    - C interface versions of the BLAS routines
  - LAPACK libraries
    - C interface versions of the LAPACK routines
  - Fortran BLAS libraries
    - Fortran 77 interface versions of the BLAS routines
  - Supports 31-bit C linkage, 31-bit XPLINK, and 64-bit XPLINK
  - Callable from C/C++

Usage & Invocation: ATLAS library

- Library and header file locations
  - /usr/lpp/cbclib/lib/atlas/lib*.a libraries
  - /usr/lpp/cbclib/include/atlas/*.h header files
- ATLAS main libraries and header files
  - libatlas.arch10.a zEC12/zBC12 single-threaded library
  - libatlas.arch11.a z13 single-threaded library
  - libtatlas.arch10.a zEC12/zBC12 multi-threaded library
  - libtatlas.arch11.a z13 multi-threaded library
  - atlas_*.h header files
- CBLAS libraries and header file
  - libcblas.arch10.a zEC12/zBC12 single-threaded library
  - libcblas.arch11.a z13 single-threaded library
  - libtcblas.arch10.a zEC12/zBC12 multi-threaded library
  - libtcblas.arch11.a z13 multi-threaded library
  - cblas.h header file

Usage & Invocation: ATLAS library

- LAPACK C libraries and header file
  - liblapack.arch10.a zEC12/zBC12 single-threaded library
  - liblapack.arch11.a z13 single-threaded library
  - libtlapack.arch10.a zEC12/zBC12 multi-threaded library
  - libtlapack.arch11.a z13 multi-threaded library
  - clapack.h header file
- Fortran BLAS libraries and header files
  - lib77blas.arch10.a zEC12/zBC12 single-threaded library
  - lib77blas.arch11.a z13 single-threaded library
  - libt77blas.arch10.a zEC12/zBC12 multi-threaded library
  - libt77blas.arch11.a z13 multi-threaded library
  - atlas_*f77*.h header files

Usage & Invocation: ATLAS library

- The following C/C++ compiler options are required to compile and link a program that utilizes ATLAS functionality:
  - FLOAT(IEEE)
  - ROUND(N) - this is enabled by default when FLOAT(IEEE) is enabled
  - ARCHITECTURE(10) - the minimum required ARCH level
  - ARCHITECTURE(11) - required if you want to enable ATLAS vector functionality in your program
  - VECTOR - required if you want to enable ATLAS vector functionality in your program
  - TARGET(zOSV2R1) - the minimum required TARGET level (target system must have the SPE for z13)

Usage & Invocation: ATLAS library

- Example: Call CBLAS version of ATLAS function DGEMM

```
#include <time.h>
#include <stdlib.h>
#include <cblas.h>

void init(double* matrix, int row, int column) {
  for (int j = 0; j < column; j++){
    for (int i = 0; i < row; i++){
      matrix[j*row + i] = ((double)rand())/RAND_MAX;
    }
  }
}

void print(const char * name, const double* matrix,
                int row, int column) {
  printf("Matrix %s has %d rows and %d columns:\n",
                name, row, column);
  for (int i = 0; i < row; i++){
    for (int j = 0; j < column; j++){
      printf("%.3f ", matrix[j*row + i]);
    }
    printf("\n");
  }
  printf("\n");
}
```

```
int main(int argc, char * argv[]) {
  int rowsA, colsB, common;
  int i,j,k;
  if (argc != 4) {
    printf("Using defaults\n");
    rowsA = 2; colsB = 4; common = 6;
  } else {
    rowsA = atoi(argv[1]); colsB = atoi(argv[2]);
    common = atoi(argv[3]);
  }
  double A[rowsA * common]; double B[common * colsB];
  double C[rowsA * colsB]; double D[rowsA * colsB];
  enum CBLAS_ORDER order = CblasColMajor;
  enum CBLAS_TRANSPOSE transA = CblasNoTrans;
  enum CBLAS_TRANSPOSE transB = CblasNoTrans;
  double one = 1.0, zero = 0.0;
  srand(time(NULL));
  init(A, rowsA, common); init(B, common, colsB);
  cblas_dgemm(order,transA,transB, rowsA, colsB, common, 1.0, A,
              rowsA ,B, common ,0.0, C, rowsA);
  for(i=0;i<colsB;i++){
    for(j=0;j<rowsA;j++){
      D[i*rowsA+j]=0;
      for(k=0;k<common;k++){
        D[i*rowsA+j]+=A[k*rowsA+j]*B[k+common*i];
      }
    }
  }
  print("A", A, rowsA, common); print("B", B, common, colsB);
  print("C", C, rowsA, colsB); print("D", D, rowsA, colsB);
  return 0;
}
```

Usage & Invocation: ATLAS library

- Compile/link commands for example program
- To compile the program for zEC12/zBC12

  xlc -c -qfloat=ieee -qround=n -qarch=10 -qtarget=zosv2r1 -I /usr/lpp/cbclib/include/atlas -qfloat=ieee -o sample.o sample.c
- To compile the program for z13

  xlc -c -qfloat=ieee -qround=n -qarch=11 -qtarget=zosv2r1 -I /usr/lpp/cbclib/include/atlas -qfloat=ieee -o sample.o sample.c
- To link the program for zEC12/zBC12

  xlc sample.o -L /usr/lpp/cbclib/lib/atlas -lcblas.arch10 -latlas.arch10 -qfloat=ieee -o sample
- To link the program for z13

  xlc sample.o -L /usr/lpp/cbclib/lib/atlas -lcblas.arch11 -latlas.arch11 -qfloat=ieee -o sample

Usage & Invocation: ATLAS library

- For detailed information on ATLAS, including lists of functions provided, see the external ATLAS web site
  http://math-atlas.sourceforge.net

Interactions & Dependencies: ATLAS library

- Software Dependencies
  - None
- Hardware Dependencies
  - arch10 ATLAS is tuned for zEC12/zBC12 and can also run on z13
  - arch11 ATLAS is tuned for z13 and runs only on z13

Overview: New ARCH/TUNE sub-option

- Problem Statement / Need Addressed
  - Exploit the latest hardware
  - Target the latest micro-architecture
- Solution
  - Indicate ARCH(11) at compile invocation
  - Indicate TUNE(11) at compile invocation
- Benefit / Value
  - Take advantage of the latest hardware features
  - Produce faster code
  - Utilize the hardware better

Usage & Invocation: New ARCH/TUNE sub-option

- xlc -qARCH=11 -qTUNE=11 a.c

Interactions & Dependencies: New ARCH/TUNE sub-option

- Hardware Dependencies
  - A z13 machine for the running code

Migration & Coexistence Considerations: New ARCH/TUNE sub-option

- Users can indicate the TUNE(11) with lower ARCH levels
  - The compiler will attempt to order the instruction to be best suited for the z13 micro-architecture feature but still run on the lower ARCH level hardware

Overview: Architecture Sections

- Problem Statement / Need Addressed
  - To avoid run time exceptions due to running on an older level machine models, lower ARCHITECTURE levels are used
    - Loss of opportunity to use the hardware instructions that could improve execution time
- Solution
  - Add a new pragma directive: #pragma arch_section(<*architecture*>)
    - The pragma indicates the start of a section of the source intended for the machine indicated by <*architecture*>
  - The compiler switches to architecture specified, and at the end of the section switches back to the previous architecture
- Benefit / Value
  - Allow specialized code paths and algorithms to take advantage of newer hardware if available

Usage & Invocation: Architecture Sections

```
>xlC -O2 -qlanglvl=extended -qlist=./a.lst a.C -qARCH=7
#include <builtins.h>
#include <stdio.h>
inline void fetch(int* n) { // Should be guarded under a run time architecture check (see associated feature)
#pragma arch_section(8)
 {                          <----- switches to arch 8
  __dcbt(n);                <----- generates the Prefetch instruction, z10, inline and ensures code stays within section
 }                          <----- end of section for ARCH(8), back to ARCH(7)
}
int main() {
  int myArray[100];
  int i;
  fetch(myArray);
  for (i=0; i< sizeof(myArray)/sizeof(int); ++i)
    myArray[i] = i*2;
  return 55;
}
```

One executable for both architectures!

Interactions & Dependencies: Architecture Sections

- Supported architectures are 5 and higher
- Supply the pragma arch_section in increasing order
- It is the programmers responsibility to check machine level
  - There are compiler-supplied routines (see Runtime Check feature) to aid in checking
- The specified TUNE option should be equal to or greater than the highest architecture section specified

Overview: Runtime Architecture Check

- Problem Statement / Need Addressed
  - Programs may require checking the machine model before doing some processing
  - Users of the `#pragma arch_section` must check the machine model before switching to a higher level ARCHITECTURE
- Solution
  - Provide built-in functions that programmers can call to find out information about the machine their code is running on
    - `__builtin_cpu_init(void)`
    - `__builtin_cpu_is(const char* cpumodel)`
    - `__builtin_cpu_supports(const char* feature)`
    - Note: `__builtin_cpu_init` must be called at first
- Benefit / Value
  - Allow specialized code paths for maximum runtime hardware exploitation

Usage & Invocation: Runtime Architecture Check

- __builtin_cpu_init(void)
  - Runs the CPU detection code, and saves the CPU information in a compiler defined/managed buffer
  - Must be called before calls to the other two built-ins
- __builtins_cpu_is(const char* *cpumodel*)
  - Returns 1 if the CPU is of type *cpumodel*
  - *cpumodel* values: "5" and higher
- __builtin_cpu_supports(const char* *feature*)
  - Returns 1 if the CPU has support for the *feature* indicated
  - *feature* values: `"longdisplacement", "etf2", "etf3", "dfp", "prefetch", "storeclockfast", "loadstoreoncond", "popcount", "interlocked", "tx", "dfpzoned", "vector128", "5", ..., "11"`
- These builtins are recognized without the need to include a header file

Interactions & Dependencies: Runtime Architecture Check

- Software Dependencies
  - The builtins should be used where `#pragma arch_section` are used for safe code. Ex.

```
int main(void) {
  __builtin_cpu_init();
  if (__builtin_cpu_supports("dfp")) {
    #pragma arch_section(7) {
    ....
  }
  return SUCCESS;
}
```

Overview: Packed Decimal Optimization (C only)

- Problem Statement / Need Addressed
  - Packed decimal calculations can be done faster using the DFP and z13 hardware
- Solution
  - Use of the following hardware instructions (new in z13) where possible to improve the run-time performance Packed Decimal operations
  - Convert from Packed
    - CDPT – Convert to long DFP
    - CXPT – Convert to extended DFP
  - Convert to Packed
    - CPDT – Convert from long DFP
    - CPXT – Convert from extended DFP
- Benefit / Value
  - Improved performance of packed decimal operations

Usage & Invocation: Packed Decimal Optimization (C only)

- Where the Packed Decimal type is used, compile with ARCH(11)
  - Gives the compiler permission to generate these new hardware instructions where it sees fit

Interactions & Dependencies: Packed Decimal Optimization (C only)

- Only available on z13 model hardware

Overview: Vector programming support

- Problem Statement / Need Addressed
  - Provides programmers direct access to the SIMD instructions from the z13 Vector Facility for z/Architecture
- Solution
  - Provide vector programming support
    - A language extensions based on the AltiVec Programming Interface specification with suitable changes and extensions
    - Includes vector data types and operators, macros, and built-in functions
- Benefit / Value
  - Empowers developers to write code that takes advantage of the substantial potential boost in performance from the new SIMD unit
  - Generally portable vector code from other AltiVec implementers

Usage & Invocation: Vector programming support

- Options:
  - ARCH(11)
  - FLOAT(AFP(NOVOLATILE))
  - TARGET(ZOSV2R1) or above
  - **VECTOR(TYPE)**
- Macro:
  - __VEC__

Usage & Invocation: Vector programming support

- Vector data types:
  - {vector, __vector} {bool, signed, unsigned} {char, short, int, long long}
  - {vector, __vector} double
    - vector {bool, signed, unsigned} long long require use of the LANGLVL(LONGLONG) compiler option
    - Alternative spelling '__vector', instead of 'vector', can be used with VECTOR(NOTYPE)
- Various language extensions on the vector types for natural usage just like for the native types
  - Ex.
    - Assignment operator (=)
    - Address operator (&)
    - Pointer arithmetic
    - Unary operators (++, –, +, -, ~)
    - Binary operators (+, -, *, /, %, &, I, ^, <<, >>)
    - Relational operators (==, !=, <, >, <=, >=)

Usage & Invocation: Vector programming support

- Vector built-in functions
  - Comprehensive set of vector built-in functions for access and manipulate individual vector elements
  - In the following high-level categories:
    - Arithmetic
    - Compare
    - Compare ranges
    - Find any element
    - Gather and scatter
    - Generate Mask
    - Isolate Zero
    - Load and Store
    - Logical
    - Merge
    - Pack and unpack
    - Replicate
    - Rotate and shift
    - Rounding and conversion
    - Test
    - All Predicates
    - Any Predicates

Usage & Invocation: Vector programming support

- Example:

```
#include <builtins.h>
#include <stdio.h>
int main() {
  vector signed int a = {-1, 2, -3, 4};  // declare and initialize a vector with 4 signed integer elements
  vector signed int b = {-5, 6, -7, 8};
  vector signed int c, d;               // declare vectors with 4 signed integer elements

  c = a + b;           // Generates VAF
  d = vec_abs(c);      // Generates VLPF

  printf("d[0] = %d\n",d[0]); // prints 6 -- d[0] extract the 1st element from the vector
  printf("d[1] = %d\n",d[1]); // prints 8
  printf("d[2] = %d\n",d[2]); // prints 10
  printf("d[3] = %d\n",d[3]); // prints 12

  return 0;
}

> xlc -qVECTOR -qARCH=11 a.c
```

Note:
- FLOAT(AFP(NOVOLATILE)) and TARGET(ZOSV2R2) are defaults
- VECTOR implies VECTOR(TYPE)

Interactions & Dependencies: Vector programming support

- Hardware Dependencies
  - z13

Migration & Coexistence Considerations: Vector programming support

- There is a common subset of functionality provided by both the vector programming support and the AltiVec Programming Interface specification
  - Helpful for writing portable code across different platforms
  - For full details, please consults the z/OS V2R1.1 C/C++ Programming Guide (Chapter 35. Using vector programming support), and the AltiVec Programming Interface Manual

Overview: Auto-SIMD

- Problem Statement / Need Addressed
  - A way for programs to make use of the SIMD instructions from the z13 Vector Facility for z/Architecture with no source code changes
- Solution
  - Auto-SIMDization
    - i.e. compiler to convert the scalar code to vector code
- Benefit / Value
  - No changes required on source code
  - Takes advantage of the substantial potential boost in performance from the new SIMD unit

Usage & Invocation: Auto-SIMD

- Required Options:
  - ARCH(11)
  - FLOAT(AFP(NOVOLATILE))
  - TARGET(ZOSV2R1) [or higher]
  - **VECTOR(AUTOSIMD)**
- VECTOR(AUTOSIMD) is on by default when:
  - ARCH(11)
  - HOT
  - FLOAT(AFP(NOVOLATILE))
  - TARGET(ZOSV2R2)
- Usage:

  > xlc -qVECTOR(AUTOSIMD) -qARCH=11 -qHOT b.c

  Note:
  - FLOAT(AFP(NOVOLATILE)) and TARGET(ZOSV2R2) are defaults
  - HOT implies OPTIMIZE(2)

Interactions & Dependencies: Auto-SIMD

- Hardware Dependencies
  - z13

DEBUGGING ENHANCEMENTS

- Capture all source for demand load
- Non-XPLINK CDA runtime

Overview: Capture all source for demand load

- Problem Statement / Need Addressed
  - "dbgld" only capture the source files with executable statements
  - Debug Tool users may also want to see variable declaration in the header file even when the header file contains only the variable declaration
- Solution
  - Add new dbgld option to capture all source files
- Benefit / Value
  - Allow showing declarations in otherwise empty files

Usage & Invocation: Capture all source for demand load

- Invocation in USS:

```
dbgld -cf a.out
```

- Specify option "CAPSRC(FULL)" in JCL

Overview: Non-XPLINK CDA runtime

- Problem Statement / Need Addressed
  - Applications using NOXPLINK CDA runtime cannot build and run with system CDA runtime
- Solution
  - Ship NOXPLINK CDA runtime as part of LE extensions
- Benefit / Value
  - Downstream applications can build and run with system CDA runtime

Overview: Function Entry Events

- Problem Statement / Need Addressed
  - It is hard to know when a function has been entered if there are a lot of functions
    - Source code changes may be prohibitive
    - Debug HOOK's may be too expensive
- Solution
  - Issue a function entry event when functions of interest are entered
- Benefit / Value
  - Allows viewing specific functions of interest for lower cost than a full HOOK

Usage & Invocation: Function Entry Events

- The FUNCEVENT(ENTRYCALL) option allows generating function entry events for all or specific functions in a compilation unit
  - Ex. For a source file containing functions x, y and z: entry event calls are generated for functions x and y, but not for z
    - `xlc -qfuncevent=entrycall=x:y`
- Calls the LE CEL4CASR or CELHCASR routines for each specified function entry based on the linkage, Non-XPLINK or XPLINK
  - CEL4RGSR can be used in LE to register a listener for function events

Interactions & Dependencies: Function Entry Events

- Software Dependencies
  - LE APAR PI12415

Installation

- None (beyond what was already mentioned)
- You can download XL C/C++ V2R1M1 web deliverable from

  http://www.ibm.com/systems/z/os/zos/downloads

- Download material:

1) XLC211.README.txt
   - contains a sample job to unpax XLC211.pax.Z and executes the SMP/E RECEIVE to receive the FMIDs

2) XLC211.pax.Z
   - contains the SMP/E MCS and the associated RELFILEs

3) Program Directory
   - contains information about the material and procedures associated with the installation

Presentation Summary

- Provided high level information on the major new enhancements to the C and C++ compilers:
  - Usability:
    - -M enhancements for better dependency file generation
    - Inline assembly
    - Metal C enhancements
    -
  - Performance:
    - MASS and ATLAS libraries
    - Hardware exploitation with new ARCH/TUNE
    - Architecture sections
    - SIMD – Vector programming support and auto-SIMD
    -
  - Debugging:
    - Capture all source
    - Non-XPLINK CDA runtime

Appendix

- z/OS XL C/C++ Messages (GC14-7305-01)
- z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer (GC14-7306-01)
- z/OS XL C/C++ User's Guide (SC14-7307-01)
- z/OS XL C/C++ Language Reference (SC14-7308-01)
- Standard C++ Library Reference (SC14-7309-00)
- Common Debug Architecture User's Guide (SC14-7310-00)
- Common Debug Architecture Library Reference (SC14-7311-01)
- DWARF/ELF Extension Library Reference (SC14-7312-01)
- z/OS Metal C Programming Guide and Reference (SC14-7313-01)
- z/OS XL C/C++ Runtime Library Reference (SC14-7314-01)
- z/OS XL C/C++ Programming Guide (SC14-7315-01)
-
- z/OS Internet Library: http://www.ibm.com/systems/z/os/zos/bkserv/
- C/C++ Cafe - Community & Forum: http://www.ibm.com/rational/community/cpp