The bgclang Compiler

Using bgclang on Vesta, Mira and Cetus

The bgclang compiler is a version of the LLVM/Clang compiler customized for the BG/Q supercomputer environment. If you have access to ALCF’s Vesta, Mira and Cetus systems, bgclang is installed for you. You can use the softenv keys:

```bash
+mpiwrapper-bgclang bgclang wrappers and toolchain
+mpiwrapper-bgclang.legacy bgclang.legacy wrappers and toolchain
```

to have the corresponding MPI wrappers added to your path. Experimental MPI version 3 support is also available using the softenv keys:

```bash
+mpiwrapper-bgclang-mpi3 bgclang MPI3 wrappers and toolchain
+mpiwrapper-bgclang-mpi3.legacy bgclang.legacy MPI3 wrappers and toolchain
+mpiwrapper-bgclang-mpi3.legacyndebug bgclang.legacy.ndebug MPI3 wrappers and toolchain
+mpiwrapper-bgclang-mpi3.ndebug bgclang.ndebug MPI3 wrappers and toolchain
```

Other BG/Q systems

If you are working on a non-ALCF BG/Q system, and would like to install the bgclang compiler, please see the bgclang project page: [http://trac.alcf.anl.gov/projects/llvm-bgq](http://trac.alcf.anl.gov/projects/llvm-bgq)

MPI and other wrappers

On an ALCF system (or any other system with a similar setup), the MPI wrapper scripts and other related programs can be easily added to your PATH (see the description of the ALCF softenv keys above). These wrappers are:

```bash
mpicc - The MPI C99 compiler
mpic++ and mpicxx - The MPI C++03 compiler
mpic++11 and mpicxx11 - The MPI C++11 compiler
```

To use bgclang without using the MPI wrappers:

```bash
bgclang (or powerpc64-bqg-linux-clang) - The C99 compiler
bgclang++ (or powerpc64-bqg-linux-clang++) - The C++03 compiler
bgclang++11 (or powerpc64-bqg-linux-clang++11) - The C++11 compiler
```

To compile code using the C++14 standard, use bgclang++11 and pass the `-std=gnu++14` (or `-std=c++14`) flag.

Mailing list and support

ALCF users may e-mail support for help with bgclang-related questions. All bgclang users are encouraged to subscribe to the mailing list: [http://lists.alcf.anl.gov/mailman/listinfo/llvm-bqg-discuss](http://lists.alcf.anl.gov/mailman/listinfo/llvm-bqg-discuss). Users of bgclang on non-ALCF systems should use the mailing list to receive help with bgclang.
General usage

bgclang command-line argument handling is designed to be similar to gcc's command-line argument handling, and where possible, bgclang tries to support the same flags. For more information on Clang, from which bgclang's frontend is derived, see http://clang.llvm.org/.

Like bgxlc and powerpc64-bq-linux-gcc, bgclang defaults to static linking. See the section on dynamic linking below for more information. In general, use of dynamically-linked executables and shared libraries is discouraged on the BG/Q.

OpenMP

bgclang fully supports the OpenMP 3.1 specification, and most OpenMP 4 features are supported. To enable OpenMP support, pass the -fopenmp flag when both compiling and linking.

Note that bgclang's OpenMP runtime library (derived from Intel's open-source implementation) is different from that used by powerpc64-bq-linux-gcc and bgxlc, and linking the OpenMP runtime library from either of those two compilers with an application compiled with bgclang -fopenmp will likely result in runtime failures.

Fast-math optimizations

bgclang supports a number of fast-math optimizations, enabled by passing -ffast-math, which increase performance but violate the relevant IEEE specification on floating-point computation. -ffast-math is conceptually similar to IBM's -qnostrict compiler flag.

Vector (QPX) intrinsics and math functions

bgclang supports the same QPX vector intrinsics (vec_add, etc.) as IBM's compiler, and it understands the vector4double type. No special flags or header files are required to enable this support.

bgclang also comes with a vector math library (derived from Naoki Shibata's SLEEF library). To use this library, include the qpxmath.h header. The bgclang wrapper scripts automatically handle linking to the vector math library, so no special linking flags are required.

#include <qpxmath.h>

the following functions are available (the functions with the _u1 suffix have no more than 1 ulp error):

vector4double xldexp(vector4double x, const int *q);
void xilogb(vector4double d, int *l);
vector4double xsin(vector4double d);
vector4double xcos(vector4double d);
void xsincos(vector4double d, vector4double *ds, vector4double *dc);
vector4double xtan(vector4double d);
vector4double xasin(vector4double s);
vector4double xacos(vector4double s);
vector4double xatan(vector4double s);
vector4double xatan2(vector4double y, vector4double x);
vector4double xlog(vector4double d);
vector4double xexp(vector4double d);
vector4double xpow(vector4double x, vector4double y);
vector4double xsinh(vector4double d);
vector4double xcosh(vector4double d);
vector4double xtanh(vector4double d);
vector4double xasinh(vector4double s);
vector4double xacosh(vector4double s);
vector4double xatanh(vector4double s);
vector4double xcbrt(vector4double d);
vector4double xexp2(vector4double a);
vector4double xexp10(vector4double a);
vector4double xexpm1(vector4double a);
vector4double xlog10(vector4double a);
vector4double xlog1p(vector4double a);
vector4double xsin_u1(vector4double d);
vector4double xcos_u1(vector4double d);
void xsincos_u1(vector4double d, vector4double *ds, vector4double *dc);
vector4double xtan_u1(vector4double d);
vector4double xasin_u1(vector4double s);
vector4double xacos_u1(vector4double s);
vector4double xatan_u1(vector4double s);
vector4double xatan2_u1(vector4double y, vector4double x);
vector4double xlog_u1(vector4double d);
vector4double xcbrt_u1(vector4double d);

plus single precision versions (which are named like the double-precision variants but have an 'f' as a suffix like this):

... vector4double xsinf(vector4double d);
vector4double xcosf(vector4double d);
... vector4double xsinf_u1(vector4double d);
vector4double xcosf_u1(vector4double d);
...

In addition, you can use IBM's SIMD MASS library by including the appropriate header and linking with -lmass_simd. Compared to IBM's SIMD MASS library, bgclang's vector math functions tend to be slower but more accurate. The maximum errors from the qpxmath functions are provided here: qpxmath_max_error.txt.

For convenience, if you define QPXMATH_MASS_SIMD_FUNCTIONS before including the qpxmath.h header, aliases will also be defined for libmass_simd function names (sind4, etc.). Note, however, that libmass_simd provides some functions not provided by bgclang's vector math library. Also, bgclang's vector math library provides vectorized ldexp and ilogb functions (which libmass_simd does not provide).
Autovectorization

bgclang's autovectorization support is enabled by default with the optimization flag \(-O3\). There are two types of autovectorization used by bgclang: Loop autovectorization (which can be disabled using \(-fno-vectorize\)) and SLP autovectorization (which can be disabled using \(-fno-slp-vectorize\)) for the autovectorization of non-loop code.

bgclang can currently transform calls to the following standard library (libm) math functions into calls to its vector math library as part of the autovectorization process: acos, acosh, asin, asinh, atan, atan2, atanh, cbrt, cos, cosh, exp, exp10, exp2, expm1, log, log10, log1p, pow, sin, sinh, tan, tanh, along with the single-precision versions. Also sqrt (and division), but only with \(-ffast\)-math. For sin, cos, tan, asin, acos, atan, atan2, log faster (but slightly less accurate) variants are used with \(-ffast\)-math.

Dynamic linking

Static linking is recommended on the BG/Q, and bgclang will link statically by default. bgclang does support dynamic linking and the creation of shared libraries (.so files). When creating a shared library, you must compile all object files using the \(-fPIC\) flag, and you must link using the \(-shared\) flag. When creating a dynamically-linked application (the executable that uses shared libraries), you must use the \(-dynamic\) flag.

Note: If you're building a dynamically-linked executable using the CMake build system, run cmake with the flag \(-DCMAKE_SKIP_RPATH=ON\); failure to do this might result in the build system stripping necessary RUNPATH attributes from your executable as part of the installation process.

Link-time optimization (LTO)

LTO is a powerful feature of bgclang and its associated toolchain which enables the compiler to perform additional global optimizations, such as function inlining, as part of the final linking process. This can be expensive in terms of compile time, but can yield significant runtime performance gains.

To use LTO you must pass the \(-flto\) flag to bgclang, both when compiling and also when linking. In addition, because the object file produced by bgclang when using LTO is in a custom format, special tools are necessary in order to:

- Combine such object files into static archives: use bgclang-ar (or equivalently powerpc64-bqg-linux-clang-ar) instead of ar (or powerpc64-bqg-linux-ar). Failure to do so will result in errors when attempting to use the static archives during linking; specifically this error (which is misleading in this context):

  error adding symbols: Archive has no index; run ranlib to add one

- Inspect the symbols defined in such object files: use bgclang-nm (or equivalently powerpc64-bqg-linux-clang-nm) instead of nm (or powerpc64-bqg-linux-nm).
bgclang's LTO capability is currently experimental. The are known issues with how debugging data is handled, and you might run into problems using -flto and -g together. We're currently working on fixing these issues.

FAQ

Why do I receive linking errors complaining about multiple definitions of inline functions?

The source code you're compiling probably assumes the GNU semantics for the inline keyword, and not those defined by the C99 standard. Compile your code with the -fgnu89-inline flag to force bgclang to use the non-standard GNU semantics.

I've received an "Application executable ELF header contains invalid value, errno 8 Exec format error" error, why?

This error normally indicates that you've attempted to run a frontend-compiled binary on the compute nodes. This error can also occur if you use the -shared flag with bgclang instead of the -dynamic flag when creating a dynamically-linked executable.

Linking code compiled with bgclang++ together with code compiled with bgclang++11 does not work, why?

Code compiled using bgclang++ uses the same libstdc++ standard template library (STL) implementation as the system-default GNU powerpc64-bgq-linux-g++ compiler. This provides compatibility with C++ libraries, including some system libraries, compiled with the GNU toolchain. This STL implementation, however, cannot provide a conforming C++11 programming environment, and so bgclang++11 uses an up-to-date STL implementation derived from LLVM's libc++. Unfortunately, this STL implementation is incompatible with libstdc++, and so linking errors will result for functions that use STL objects as part of their signatures (i.e. parameter or return types).

I'd like to use bgclang's OpenMP support and also link against code that uses IBM's OpenMP implementation (such as IBM's SMP ESSL library). Can I do that?

No, unfortunately both bgclang's OpenMP library and IBM's OpenMP library define functions of the same name, and using both at the same time is not generally possible. That having been said, if you're willing to play games with how your application is linked, it might be possible, and you should ask for advise on the mailing list.

Is there a corresponding Fortran compiler available?

No, not yet. This is also being worked on.