bgclang: Creating an Alternative, Customizable, Toolchain for the Blue Gene/Q

Hal Finkel
Leadership Computing Facility
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439
Email: hfinkel@anl.gov

I. Introduction

The IBM Blue Gene/Q (BG/Q) supercomputer, the third generation in the IBM Blue Gene line, has once again demonstrated the viability of using a non-commodity hardware to deliver scalability, power efficiency and good system reliability while providing high computational performance. One unfortunate disadvantage of this custom-hardware approach, however, is that the software ecosystem made available on these highly specialized machines pales in comparison to what is available to users of commodity hardware. This is especially true for software development tools: compilers, debuggers, profiling tools, etc. because these tools require non-trivial customization to work well on custom hardware. Not only does this restricted software ecosystem often make it difficult to compile, debug and profile applications, but it also limits the ability of researchers and engineers to experiment with alternative techniques across large codebases. This is because the usual open-source tools, which are not only widely supported by applications but also provide a solid platform for research, are often not available, or don't work well, on non-commodity machines.

A. bgclang

In this poster, bgclang, a compiler toolchain based on the LLVM/Clang compiler infrastructure [13], but customized for the IBM Blue Gene/Q (BG/Q) supercomputer, is described. By enhancing LLVM with support for the BG/Q's QPX vector instruction set, bgclang inherits from LLVM/Clang a high-quality auto-vectorizing optimizer, C++11 frontend, and many other associated tools. This allows bgclang to provide both capabilities not otherwise available on the BG/Q and also a much-needed platform for experimentation. The fact that a fully C++11-compliant programming environment, supporting OpenMP and QPX autovectorization, is available on the BG/Q only because of bgclang is probably bgclang's most important contribution to computational science as a whole. Furthermore, the author of this poster has played a major role in the development of LLVM's autovectorization infrastructure, and has contributed significantly to the PowerPC backend. However, for the sake of brevity, and to focus only on novel development of direct relevance to the BG/Q, this poster will present two facets of bgclang that highlight its unique capabilities: First, bgclang's Address Sanitizer feature, which provides efficient instrumentation-based memory-access validation that can be applied at scale; Second, experiments in loop optimization techniques, including a software prefetching optimization, within bgclang. These loop optimizations can produce 2x speedups over code produced by other available compilers for loops in the extended TSVC bechmark suite [15].

II. Address Sanitizer on the BG/Q

The Address Sanitizer project provides instrumentation-based memory-access bounds checking, a debugging technique essential for finding buffer-overflow errors in production-level applications with realistic working sets. Because scientific applications often contain large arrays, and use explicit index-calculation code to access those arrays, buffer overflow errors are fairly common, often due to "off-by-one" programming mistakes. Address Sanitizer can also find use-after-free (a.k.a. dangling-pointer) errors. There are two widely used open-source tools for catching these kinds of errors: Valgrind [23] and Address Sanitizer. A prototype of Valgrind is currently available on the BG/Q, however the slowdown factor imposed by Valgrind is much higher than that from Address Sanitizer. The magnitude of the slowdown is important, because to find an error a realistic working set may need to be provided to the application, and the error might not occur until some amount of computation has already been performed, so the application must be able to reach the erroneous state in some time window acceptable to the relevant supercomputing facility. On modern out-of-order server-level cores, the slowdown from Address Sanitizer is approximately 2x, while the slowdown from Valgrind is approximately 20x [16]. Valgrind on the BG/Q produces slowdowns of, for example, 178x for the first TSVC benchmark loop.

The performance of Address Sanitizer on the BG/Q is good enough to find errors while running production-level workloads. This is a unique capability provided by bgclang.

III. Loop Optimizations

The optimization of loops has long been of interest in high-performance computing, and is of paramount importance to efficient use of the BG/Q. There are several important factors governing loop performance on the BG/Q described on the poster. The poster focuses on a couple of issues, demonstrating how bgclang serves as an effective research platform for the development of beneficial loop transformations in a general-purpose compiler.

A. Software Prefetching

The hardware prefetcher (L1P) on the BG/Q does not place data directly into the L1 cache, but rather, holds it in a dedicated L1P buffer. Accessing data from this buffer takes almost 5x as long as reading data stored in the L1 cache itself. While there are sometimes 30 instructions in the loop body that can be scheduled in between a load and its first use, this is often not the case, and mitigating this excess latency can be accomplished via multiple mechanisms, including:

- Running with multiple hardware threads
- Loading data in each loop iteration for use by some future loop iteration[1]
- Issuing explicit prefetch instructions (which do bring the data into the L1 cache)

A custom loop transformation was developed for bgclang which inserts explicit prefetch instructions for loads in inner loops. This transformation needs to predict the number of cycles each loop iteration will require and, from that, determine for how many iterations ahead to prefetch data. This yields large performance benefits, as demonstrated, for many classes of loops.

B. Pre-Increment Load and Store Preparation

Many loops access elements of one or more data arrays demonstrating the idiom "load from or store to this pointer, increment (or decre ment) the pointer by some amount, and then load from or store to it again." Less common is the idiom "take some pointer,increment (or decrement) the pointer by some amount, and then load from or store to it", however, the PowerPC ISA (including the QPX vector instruc tions supported by the BG/Q) provide load and store instructions that embody this second idiom. Taking advantage of these instructions on the BG/Q is important, especially for loops with small bodies, because they allow transforming a two-cycle load/store-add pair into a single instruction that dispatches in a single cycle. bgclang includes a custom transformation used to prepare loops for pre-increment instructions.

For 48 of the first 105 benchmark loops, bgclang generated faster code than bgxlc with aggressive prefetching enabled. That number drops to 24 when bgclang's prefetching is disabled and drops to 36 when bgclang's pre-increment load/store preparation is disabled.

We can thus conclude that these two loop transformations, pre-increment preparation and prefetch generation, are useful for generating code for the BG/Q. Moreover, the fact that bgclang allowed us to explore this possibility in a general purpose compiler and, although not discussed here, over a wide array of both benchmarks and application codes, demonstrates the significant utility of the project as a whole. New transformations can be tested and deployed as part of an autovectorizing compiler on the BG/Q using bgclang, a capability that would not otherwise exist given only IBM's provided compilers.

[1] IBM's bgxlc compiler has been observed to apply this technique.

IV. Conclusion

This poster describes bgclang, a combination of a customized LLVM/Clang compiler with other toolchain components, that together create a complete, open-source HPC toolchain for the BG/Q. Not only does bgclang fulfill specific needs not otherwise met by the vendor-supplied software (an autovectorizing C++11 compiler, instrumentation-based memory-access validation, etc.), but crucially, bgclang provides a fertile platform for future research built on top of those production-quality components. Two examples, the Address Sanitizer feature, and the loop optimizations, were presented as examples, not only of useful features that bgclang provides, but of how bgclang provides a much-needed platform for BG/Q-specific research in compilers and tools. Many of the lessons leaned from the development of bgclang are expected to be transferable to future HPC architectures, and will not only influence what capabilities LLVM/Clang might provide in the future, but also what features vendors might provide in the future, and the future directions of the C++ language itself. In short, bgclang has succeeded in making the BG/Q a more-productive place to conduct scientific research in many different areas.

References

[1] Adelstein-Lelbach, B., Anderson, M., Kaiser, H., C++Now! (2012).

[2] IBM, *A2 Processor User's Manual for Blue Gene/Q*, version 1.3 (2012).

[3] IBM, *QPX Architecture* (2012).

[4] IBM, *Blue Gene/Q Application Development*, IBM Redbooks (2013).

[5] http://bluebrain.epfl.ch/

[6] http://clang.llvm.org/

[7] http://clang-omp.github.io/

[8] Callahan, D., Kennedy, K., Porterfield, A., ACM SIGOPS Rev **25** (1991).

[9] Haring, R.A., M. Ohmacht, T.W. Fox, M.K. Gschwind, P.A. Boyle, N.H. Christ, C. Kim, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, and G.L. Chiu, IEEE Micro **32**, 48 (2012).

[10] http://www.openmprtl.org/

[11] Grosser, T., Groesslinger, A., Lengauer, C., Parallel Processing Letters **22**, 4 (2012).

[12] Habib, S., Morozov, V., Finkel, H., Pope, A., Heitmann, K., Kumaran, K., Peterka, T., Insley, J., Daniel, D., Fasel, P., Frontiere, N., Lukic, Z., SC'12 (2012).

[13] Lattner, C., Adve, V., CGO'04 (2004)

[14] http://libcxx.llvm.org/

[15] Maleki, S., Gao, Y., Garzarán, M., Wong, T., Padua, D., PACT'11 (2011).

[16] http://code.google.com/p/address-sanitizer/wiki/ComparisonOfMemoryTools

[17] https://code.google.com/p/address-sanitizer/wiki/AddressSanitizerAlgorithm

[18] http://www.openfoam.com/

[19] Poulson, J., Marker, B., van de Geijn, R., Hammond, J., Romero, N., ACM Trans. Math Soft. (2013).

[20] Shibata, N., Computer Science - Research and Development **25** 1 (2010).

[21] Schnetter, E., https://bitbucket.org/eschnett/vecmathlib

[22] https://code.google.com/p/qmcpack/

[23] http://valgrind.org/

[24] Wang, A., Gaudet, M., Peng, W., Amaral, J., Ohmacht, M., Barton, C., Silvera, R., Michael, M., PACT'12 (2012).