Institute of Physics Publishing Journal of Physics: Conference Series **46** (2006) 157–160
doi:10.1088/1742-6596/46/1/022 SciDAC 2006

The Blue Gene, GCC and lattice QCD: a case study

Andrew Pochinsky
Massachusetts Institute of Technology, 77 Mass. Ave., Cambridge, MA 02139
E-mail: avp@mit.edu

Abstract. An vector extension to the C programming language utilizing Blue Gene/L floating point hardware is presented. The extensions are implemented in the GNU compiler collection toolchain and are available as a cross compiler.

1. Introduction

The Blue Gene can potentially provide considerable computational power. It has a high performance floating point unit and a network interface tightly coupled to the processor. This very promising combination is difficult to exploit efficiently, however, because of lack of a programming abstraction that is both close enough to the hardware to allow for an efficient implementation and independent enough from the machine details to facilitate writing portable code.

This work attempts to build a better abstraction of the floating point unit than currently provided by the `C` and `C++` compilers available for the machine.

2. Blue Gene Architecture

The BG/L processor core includes the Double Hummer floating point (DH) unit that provides several kinds of operations. The following are important for us:

- Parallel floating point operations on pairs of doubles. The usual PPC set of multiplications, additions and fused multiply-adds is provided.
- Loads and stores both in single and double precision. The processor generates an alignment exception if the data is not naturally aligned, but the compute node kernel (CNK) provides a handler for it.
- Cross operations that allow one to compute a complex multiplication in two instructions.
- Other operations: various cross multiplications, moves and rounding to single precision.

The network interface is sufficiently decoupled from the register file, that for our current purposes it could be viewed as a set of memory mapped I/O ports that are accessed by loading from and storing to them the contents of the vector registers. At the present stage we do not attempt to abstract communications any further than that.

To simplify porting of existing codes, an abstraction exposing parallel operations is implemented for the `C` and `C++` programming languages using GNU GCC as a starting point.

3. GCC Vector Extensions

Vector extensions proved a useful abstraction both with x86 SSE and PPC AltiVec. For the BG/L, a similar approach has been taken. The DH registers are abstracted as `vector double` and `vector float` data types and the usual compliment of floating point operations is provided on them. Since the compiler already handles fused floating point operations, extending them to the new data types is a natural step.

Following the AltiVec interface, one needs to specify the `-mbluelight` flag to the compiler and add

```
#include <bluelight.h>
```

into a source file to access BG/L extensions. After that, declarations like the following will work

```
vector double foo;
vector float bar;
```

Both `vector float` and `vector double` contain two elements. This makes it possible to cast between vector types:

```
vector double a;
vector float b = (vector float)a;
```

However, automatic casts are not provided. It is also an error to cast non-vectors to vectors and vice a versa.

For both single and double precision, full sets of arithmetic operations are provided in spite the fact that the hardware does only double precision operations. The single precision rounding requirements of IEEE 754 are not enforced by the compiler, however. Since the processor does not implement rounding exceptions anyway this behavior does not seem an issue. The single precision is provided as a convenience since many applications are memory bound and a factor of the memory footprint is significant.

The initializers are provided

```
vector float f = {3.1415, 2.7182};
```

Also, the usual arithmetic operations of addition, subtraction and multiplication that act on vectors elementwise are presented:

```
vector double a, b, c, d;
a = b + c * d;
b += -a * d;
...
```

One can pass vector data to the function as arguments and return them as results.

Combinations of vectors multiply and additions will be converted into fused instructions unless the `-mno-fused-madds` is given at the command line.

All Double Hummer operations are exposed in the `<bluelight.h>` header file. One should use `vec_` macros since the implementation via GNU builtins is subject to change. Semantics of the operations are described in the GCC info documentation.

4. Implementation Details

GCC version 3.4.4 was the latest of the 3.x tree at the time the project started. It was chosen as a starting point for the present work.

All builtins are expanded inline by the compiler and, since their resource requirements are known, are efficiently scheduled. In fact, the compiler fusion pass will consider combining

multiplies and additions even if they were written as `vec_add(a,vec_mul(b,c))` instead of `a+b*c`. This is important if the GCC is used as a back-end for high level code generating tools.

To avoid complications with automatic variables, the stack is aligned at 128 bits, so that local data requires no special handling.

The standard library is built so that `malloc()` returns 128 bit aligned memory.

The vector argument passing is supported in the same way it is done for floats: the first eight arguments are passed via registers, and the remaining arguments on the stack.

The only part of the procedure call mechanism that treats vector types differently from other primitive types, is the `<stdarg.h>`. In particular, the compiler does not support passing vectors in the ellipsis position and retrieving them via the `va_arg()` macro. In this case an error is signaled by the compiler. This restriction is due to complexity of changes necessary to implement varargs properly and perceived lack of use for it in the target application area.

The fact that the vector registers overlap the floating point registers required some special care. The problem concerns handling the register file overflow; the solution is to treat all FPRs as VRs when dealing with spills. This approach required minimal changes in the internals of the compiler and allowed one to freely intermix object files compiled with the BlueLight extensions with files compiled without it.

Changes to the RTL were surprisingly straightforward. The DH instructions are orthogonal to other operations and register dependencies are handled in the machine specific files in `gcc/config/rs6000` directory.

The instruction scheduler can still benefit from some tuning to the BG/L processor, but it is easy to do once more information about the processor timing is known.

No attempt has been made to implement complex arithmetics via the DH instructions, because GCC converts complex arithmetic into real operations too early in the compilation process and reassembling it back is not feasible in GCC version 3.x. There are conceptual problems with `complex double` and `complex float` as defined by the C99 standard, however. ISO 9899:1999 requires that complex floats should have the same alignment requirements as corresponding real floats. Adhering to the standard's requirements will impose a severe performance penalty, so there is a strong pressure to break the standard at this point. While this violation of the standard is not an issue for in most cases, there are situations when subtle errors could be very confusing.

5. SciDAC Lattice QCD Codes

Level III routines for inverting the Domain Wall Fermion Dirac operator from the SciDAC Lattice QCD project are being ported to the Blue Gene using the vector extensions described here. The BG/L specific parts are limited to ten single-line inlined functions and is not different in this respect from other ports (x86 SSE and PPC AltiVec).

The work is underway to implement QLA and QDP level II routines on top of the vector extensions. This will allow the whole suite of the SciDAC LQCD codes to run more efficiently on the Blue Gene.

6. From LQCD to Molecular Dynamics

Both classical and quantum MD codes can greatly benefit from the vector extensions, though for different reasons. The CMD codes of interest operate on small matrices and, since BG/L vector size is 2, the primitive matrix operations could be compiled efficiently if written in terms of vector data types.

The QMD codes are essentially data parallel operating on very long vectors. This maps naturally into the vector abstraction provided by the compiler.

7. Code Availability

The patched compiler is available from

<http://web.mit.edu/bgl/software/>

in both source and binary forms. It is distributed under the GPL.

The toolchain contains everything needed to produce BG/L executables. It could be used as a cross compiler, it is possible to compile BG/L codes on both big-endian (e.g., PPC) and little-endian (e.g., x86) platforms.

We plan to support for the next generation of the Blue Gene architecture in the compiler as the details of its floating point hardware become available.

Acknowledgments

This work is supported the U. S. Department of Energy under grant DE-FC02-01ER41193. The implementation is based on the GNU Compiler Collection provided by the Free Software Foundation.