Blue Gene Vector Extensions for GCC

Andrew Pochinsky
`avp@mit.edu`

August 30, 2006

Abstract

This white paper describes extensions to gcc for utilizing the Blue-Gene/L Double Hummer floating point unit.

INTRODUCTION

To efficiently utilize BG/L hardware, one needs to use the Double Hummer (DH) FPU for numerical calculations. Currently only IBM's XLC has limited support for it, and GCC users have to resort to `asm()` fragments. Both approaches are not ideal: XLC crashes on SciDAC LQCD codes and GCC instruction scheduling suffers in the presence of explicit `asm()`. To address these issues, `gcc` version 3.4.4 was extended to understand certain aspects of the DH hardware. This release exposes only the vector part of the BG/L architecture through primitive data types and operations, while providing access to the rest of DH instruction set via gcc's built-in mechanism.

Language-level support for complex floating point data types is planned for the future.

The present extension is not compatible with XLC intrinsics—programs written for GCC will not compile with XLC and vice a versa.

COMPILER SWITCHES AND HEADERS

To access DH extensions one needs to use the command line switch `-mbluelight` *and* include the header file `<bluelight.h>`. The compiler adds `__BLUELIGHT__` to predefined symbols if DH support is enabled.

It is possible to mix files compiled with and without DH support in a single program. In fact, `libc` is built without DH extensions.

VECTOR DATA TYPES

There are a couple of special data types, `vector double` and `vector float`, that implement 2-element vectors of doubles and floats respectively. The compiler aligns all data of these types at 16 byte boundaries for `vector double` and 8 byte boundaries for `vector float`. The C library is also aware of extra alignment requirements, e.g., `malloc()` always returns 16-byte aligned memory. Vector objects are almost first class in the language: vector objects could be declared anywhere a declaration is permitted, could be parts of composite data types, could be passed as arguments to procedures and returned as values of functions, their address can be taken, etc. The only restriction is that there is no access to `vector` data via the `stdarg` mechanism due to the complexity of the required changes to GCC and the perceived rarity of use. The compiler will issue a warning if a `vector` argument is passed at an ellipsis position to a function. There will be no warning, however, if the compiler does not know the function prototype as with any other mismatch between the function prototype and its use.

One can combine declarations and initializations for vector data as follows:

```
vector double foo = {1.2, 3.4};
vector float baz = {5.6, 7.8};
```

OPERATIONS

For this section, assume that the following declarations are visible

```
vector double a, b, c, d;
vector float x, y, z, t;
```

In addition to argument passing (e.g., `foo(a,x,10,0,1,b);`) and value return (e.g., `b=bar(); x=baz();`) mentioned above, one can assign values

```
a = b;
x = y;
```

and do arithmetic in corresponding rings:

```
a = b + c * d - a * (b + d);
x = y * z - t;
```

and change precision:

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

However, the conversion must be explicit as there is no implicit conversion such as the C conversions between `float` and `double`.

Note that, though reasonable, vector types do not form a vector space over reals. This limitation is a result of the complexity of the changes to gcc to support it properly.

By default, the compiler will issue fused multiply-add instructions where possible. To disable this optimization, use `-mno-fused-madd`, just as you do to suppress normal PPC fused multiply-add optimizations.

BLUELIGHT BUILT-INS

The remainder of DH functionality is provided via `__builtin` primitives. Since this interface can change in the next release, users are advised to access DH primitives though the `<bluelight.h>` header. Once `<bluelight.h>` is included, the operations listed below are provided.

```
vector double vec_add(vector double A, vector double B);
```

Compute an element-wise sum of arguments:

$$\begin{aligned} R_p &\leftarrow A_p + B_p \\ R_s &\leftarrow A_s + B_s \end{aligned}$$

```
vector double vec_sub(vector double A, vector double B);
```

Compute an element-wise difference of arguments:

$$\begin{aligned} R_p &\leftarrow A_p - B_p \\ R_s &\leftarrow A_s - B_s \end{aligned}$$

```
vector double vec_rec_est(vector double A);
```

Compute an element-wise estimate of the inverse of the argument (the details of what `estimate`() does should be somewhere in the arithmeticPPC documentation):

$$\begin{aligned} R_p &\leftarrow \texttt{estimate}(1/A_p) \\ R_s &\leftarrow \texttt{estimate}(1/A_s) \end{aligned}$$

```
vector double vec_rec_sqrt_est(vector double A);
```

Compute an element-wise estimate of the inverse square root of the argument (details of what `estimate`() does should be somewhere in PPC documentation):

$$\begin{aligned} R_p &\leftarrow \texttt{estimate}(1/\sqrt{A_p}) \\ R_s &\leftarrow \texttt{estimate}(1/\sqrt{A_s}) \end{aligned}$$

```
vector double vec_mul(vector double A, vector double B);
```

Compute an element-wise product of arguments:

$$\begin{aligned} R_p &\leftarrow A_p * B_p \\ R_s &\leftarrow A_s * B_s \end{aligned}$$

```
vector double vec_cross_mul(vector double A,
                            vector double B);
```

Compute an element-wise cross product of arguments:

$$\begin{aligned} R_p &\leftarrow A_s * B_p \\ R_s &\leftarrow A_p * B_s \end{aligned}$$

```
vector double vec_cross_pmul(vector double A,
                             vector double B);
```

Multiply B by A_p:

$$\begin{aligned} R_p &\leftarrow A_p * B_p \\ R_s &\leftarrow A_p * B_s \end{aligned}$$

```
vector double vec_cross_smul(vector double A,
                             vector double B);
```

Multiply B by A_s:

$$\begin{aligned} R_p &\leftarrow A_s * B_p \\ R_s &\leftarrow A_s * B_s \end{aligned}$$

```
vector double vec_madd(vector double A,
                       vector double B,
                       vector double C);
```

Compute an element-wise multiply-add:

$$\begin{aligned} R_p &\leftarrow A_p * B_p + C_p \\ R_s &\leftarrow A_s * B_s + C_s \end{aligned}$$

```
vector double vec_nmadd(vector double A,
                        vector double B,
                        vector double C);
```

Compute an element-wise negative multiply-add:

$$\begin{aligned} R_p &\leftarrow -(A_p * B_p + C_p) \\ R_s &\leftarrow -(A_s * B_s + C_s) \end{aligned}$$

```
vector double vec_msub(vector double A,
                       vector double B,
                       vector double C);
```

Compute an element-wise multiply-substract:

$$\begin{aligned} R_p &\leftarrow A_p * B_p - C_p \\ R_s &\leftarrow A_s * B_s - C_s \end{aligned}$$

```
vector double vec_nmsub(vector double A,
                        vector double B,
                        vector double C);
```

Compute an element-wise negative multiply-substract:

$$\begin{aligned} R_p &\leftarrow -(A_p * B_p - C_p) \\ R_s &\leftarrow -(A_s * B_s - C_s) \end{aligned}$$

```
vector double vec_cross_madd(vector double A,
                             vector double B,
                             vector double C);
```

Compute an element-wise cross multiply-add:

$$\begin{aligned} R_p &\leftarrow A_s * B_p + C_p \\ R_s &\leftarrow A_p * B_s + C_s \end{aligned}$$

```
vector double vec_cross_nmadd(vector double A,
                              vector double B,
                              vector double C);
```

Compute an element-wise negative cross multiply-add:

$$\begin{aligned} R_p &\leftarrow -(A_s * B_p + C_p) \\ R_s &\leftarrow -(A_p * B_s + C_s) \end{aligned}$$

```
vector double vec_cross_msub(vector double A,
                             vector double B,
                             vector double C);
```

Compute an element-wise cross multiply-substract:

$$\begin{aligned} R_p &\leftarrow A_s * B_p - C_p \\ R_s &\leftarrow A_p * B_s - C_s \end{aligned}$$

```
vector double vec_cross_nmsub(vector double A,
                              vector double B,
                              vector double C);
```

Compute an element-wise negative cross multiply-substract:

$$\begin{aligned} R_p &\leftarrow -(A_s * B_p - C_p) \\ R_s &\leftarrow -(A_p * B_s - C_s) \end{aligned}$$

```
vector double vec_cross_cpmadd(vector double A,
                               vector double B,
                               vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow A_p * B_p + C_p \\ R_s &\leftarrow A_p * B_s + C_s \end{aligned}$$

```
vector double vec_cross_cpnmadd(vector double A,
                                vector double B,
                                vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow -(A_p * B_p + C_p) \\ R_s &\leftarrow -(A_p * B_s + C_s) \end{aligned}$$

```
vector double vec_cross_cpmsub(vector double A,
                               vector double B,
                               vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow A_p * B_p - C_p \\ R_s &\leftarrow A_p * B_s - C_s \end{aligned}$$

```
vector double vec_cross_cpnmsub(vector double A,
                                vector double B,
                                vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow -(A_p * B_p - C_p) \\ R_s &\leftarrow -(A_p * B_s - C_s) \end{aligned}$$

```
vector double vec_cross_csmadd(vector double A,
                               vector double B,
                               vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow A_s * B_p + C_p \\ R_s &\leftarrow A_s * B_s + C_s \end{aligned}$$

```
vector double vec_cross_csnmadd(vector double A,
                                vector double B,
                                vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow -(A_s * B_p + C_p) \\ R_s &\leftarrow -(A_s * B_s + C_s) \end{aligned}$$

```
vector double vec_cross_csmsub(vector double A,
                               vector double B,
                               vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow A_s * B_p - C_p \\ R_s &\leftarrow A_s * B_s - C_s \end{aligned}$$

```
vector double vec_cross_csnmsub(vector double A,
                                 vector double B,
                                 vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow -(A_s * B_p - C_p) \\ R_s &\leftarrow -(A_s * B_s - C_s) \end{aligned}$$

```
vector double vec_cross_cpnpma(vector double A,
                                vector double B,
                                vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow -(A_p * B_p - C_p) \\ R_s &\leftarrow A_p * B_s + C_s \end{aligned}$$

```
vector double vec_cross_csnpma(vector double A,
                                vector double B,
                                vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow -(A_s * B_p - C_p) \\ R_s &\leftarrow A_s * B_s + C_s \end{aligned}$$

```
vector double vec_cross_cpnsma(vector double A,
                                vector double B,
                                vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow A_p * B_p + C_p \\ R_s &\leftarrow -(A_p * B_s - C_s) \end{aligned}$$

```
vector double vec_cross_csnsma(vector double A,
                                vector double B,
                                vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow A_s * B_p + C_p \\ R_s &\leftarrow -(A_s * B_s - C_s) \end{aligned}$$

```
vector double vec_cross_cxnpma(vector double A,
                                vector double B,
                                vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow -(A_s * B_s - C_p) \\ R_s &\leftarrow A_s * B_p + C_s \end{aligned}$$

```
vector double vec_cross_cxnsma(vector double A,
                               vector double B,
                               vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow A_s * B_s + C_p \\ R_s &\leftarrow -(A_s * B_p - C_s) \end{aligned}$$

```
vector double vec_cross_cxma(vector double A,
                             vector double B,
                             vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow A_s * B_s + C_p \\ R_s &\leftarrow A_s * B_p + C_s \end{aligned}$$

```
vector double vec_cross_cxnms(vector double A,
                              vector double B,
                              vector double C);
```

Part of complex arithmetic:

$$\begin{aligned} R_p &\leftarrow -(A_s * B_s - C_p) \\ R_s &\leftarrow -(A_s * B_p - C_s) \end{aligned}$$

```
vector double vec_sel(vector double A,
                      vector double B,
                      vector double C);
```

Bluelight select. This instruction is apparently broken in the BG/L:

$$\begin{aligned} R_p &\leftarrow A_p ? B_p : C_p \\ R_s &\leftarrow A_s ? B_s : C_s \end{aligned}$$

```
vector double vec_ctiw(vector double A);
```

Element-wise convert to integer. It appears to be broken as well:

$$\begin{aligned} R_p &\leftarrow \texttt{integer}(A_p) \\ R_s &\leftarrow \texttt{integer}(A_s) \end{aligned}$$

```
vector double vec_ctiwz(vector double A);
```

Element-wise convert to integer toward zero. It appears to be broken as well:

$$\begin{aligned} R_p &\leftarrow \texttt{integer\_toward\_zero}(A_p) \\ R_s &\leftarrow \texttt{integer\_toward\_zero}(A_s) \end{aligned}$$

```
vector double vec_rsp(vector double A);
```

Round to single precision element-wise:

$$\begin{aligned} R_p &\leftarrow \texttt{float}(A_p) \\ R_s &\leftarrow \texttt{float}(A_s) \end{aligned}$$

```
vector double vec_mr(vector double A);
```

Move vector register. There is no real need for this instruction as the compiler does it very well:

$$\begin{aligned} R_p &\leftarrow A_p \\ R_s &\leftarrow A_s \end{aligned}$$

```
vector double vec_neg(vector double A);
```

Compute element-wise negation:

$$\begin{aligned} R_p &\leftarrow -A_p \\ R_s &\leftarrow -A_s \end{aligned}$$

```
vector double vec_abs(vector double A);
```

Compute element-wise absolute value:

$$\begin{aligned} R_p &\leftarrow |A_p| \\ R_s &\leftarrow |A_s| \end{aligned}$$

```
vector double vec_nabs(vector double A);
```

Compute element-wise negative of the absolute value:

$$\begin{aligned} R_p &\leftarrow -|A_p| \\ R_s &\leftarrow -|A_s| \end{aligned}$$

```
vector double vec_secondary_mr(vector double A,
                               vector double B);
```

Some vector mixing:

$$\begin{aligned} R_p &\leftarrow A_p \\ R_s &\leftarrow B_s \end{aligned}$$

```
vector double vec_secondary_neg(vector double A,
                                vector double B);
```

Some vector mixing:

$$\begin{aligned} R_p &\leftarrow A_p \\ R_s &\leftarrow -B_s \end{aligned}$$

```
vector double vec_secondary_abs(vector double A,
                                vector double B);
```

Some vector mixing:

$$\begin{aligned} R_p &\leftarrow A_p \\ R_s &\leftarrow |B_s| \end{aligned}$$

```
vector double vec_secondary_nabs(vector double A,
                                 vector double B);
```

Some vector mixing:

$$\begin{aligned} R_p &\leftarrow A_p \\ R_s &\leftarrow -|B_s| \end{aligned}$$

```
vector double vec_cross_mr(vector double A);
```

Swap vector components:

$$\begin{aligned} R_p &\leftarrow A_s \\ R_s &\leftarrow B_p \end{aligned}$$

```
vector double vec_mk2d(double A, double B);
```

Construct a vector from two doubles:

$$\begin{aligned} R_p &\leftarrow A \\ R_s &\leftarrow B \end{aligned}$$

```
vector double vec_mk2f(float A, float B);
```

Construct a vector from two floats:

$$\begin{aligned} R_p &\leftarrow \texttt{double}(A) \\ R_s &\leftarrow \texttt{double}(B) \end{aligned}$$

```
vector double vec_mk00(vector double A,
                       vector double B);
```

Construct a vector from two vectors:

$$\begin{aligned} R_p &\leftarrow A_p \\ R_s &\leftarrow B_p \end{aligned}$$

```
vector double vec_mk11(vector double A,
                       vector double B);
```

Construct a vector from two vectors:

$$\begin{aligned} R_p &\leftarrow A_s \\ R_s &\leftarrow B_s \end{aligned}$$

```
vector double vec_mk10(vector double A,
                       vector double B);
```

Construct a vector from two vectors:

$$\begin{aligned} R_p &\leftarrow A_s \\ R_s &\leftarrow B_p \end{aligned}$$

```
double vec_get0(vector double A);
```

Extract the primary component of a vector:

$$R \leftarrow A_p$$

```
double vec_get1(vector double A);
```

Extract the secondary component of a vector:

$$R \leftarrow A_s$$

AVAILABILITY

The patched gnu tool chain is available as RPM packages (both source and ppc64 Linux binaries) at `<http://web.mit.edu/bgl/software/>`. It provides its own version of `libc` built from the newlib distribution and a complete set of development tools. Please keep in mind that you will need access to the IBM headers and libraries to do anything interesting on the Blue Gene.