Custom math functions for molecular dynamics

R. F. Enenkel
B. G. Fitch
R. S. Germain
F. G. Gustavson
A. Martin
M. Mendell
J. W. Pitera
M. C. Pitman
A. Rayshubskiy
F. Suits
W. C. Swope
T. J. C. Ward

While developing the protein folding application for the IBM Blue Gene®/L supercomputer, some frequently executed computational kernels were encountered. These were significantly more complex than the linear algebra kernels that are normally provided as tuned libraries with modern machines. Using regular library functions for these would have resulted in an application that exploited only 5–10% of the potential floating-point throughput of the machine. This paper is a tour of the functions encountered; they have been expressed in C++ (and could be expressed in other languages such as Fortran or C). With the help of a good optimizing compiler, floating-point efficiency is much closer to 100%. The protein folding application was initially run by the life science researchers on IBM POWER3™ machines while the computer science researchers were designing and bringing up the Blue Gene/L hardware. Some of the work discussed resulted in enhanced compiler optimizations, which now improve the performance of floating-point-intensive applications compiled by the IBM VisualAge® series of compilers for POWER3, POWER4™, POWER4+™, and POWER5™. The implementations are offered in the hope that they may help in other implementations of molecular dynamics or in other fields of endeavor, and in the hope that others may adapt the ideas presented here to deliver additional mathematical functions at high throughput.

Molecular dynamics

Sequencing the genome has enabled scientists to read the "words" in the building blocks of life. All-atom molecular dynamics is one of the tools in the grand challenge of understanding the stories told by those words.

We want to model the time–series behavior of a covalently bonded structure, such as a protein molecule that is surrounded by water molecules, as it would be in a living cell. We usually imagine a single protein molecule in a cubic box of a few thousand water molecules, and then imagine that there are identical boxes stacked in all directions, rather like atomic-scale synchronized swimming, with the swimmer made up of balls held together with springs. To understand the behavior of a single "spring" would require quantum mechanics, but on the larger scale of wanting to understand the "swimmer," classical mechanics is sufficient.

Most of the forces to be calculated are the long-range electrostatic forces between atoms in separate water molecules, but the interesting behavior is related to the short-range forces along the springs and between various three-atom and four-atom bonded groups. This requires calculation of large quantities of square roots and their reciprocals (for multiplying and dividing by distances); error functions (one way of approaching the electrostatics); angles (between pairs of springs); periodic images (to work out which swimmer a water molecule is nearest to); and polynomials (for Lennard–Jones bonded forces and to softly switch off forces as pairs of atoms move farther apart and fade to the background).

Custom math functions

Source code for the functions presented here can be found at [1].

0018-8646/05/$5.00 © 2005 IBM

Vectorizable[1] math functions

The IBM xlc compiler can schedule instructions flexibly within a basic block, that is, a sequence of code with no conditional branches and no entry points other than the first instruction. This paper explains how to exploit this for functions commonly used in molecular dynamics; if the compiler can be enabled to see a sufficient number of independent instructions, it will schedule instructions to avoid stalls in the floating-point execution pipeline, and so the hardware will run at a high fraction of peak throughput. To make good use of the compiler instruction scheduling facility, the use of branch instructions should be minimized. This means that special cases and error handling should be omitted or done in a way that avoids branches. Therefore, all of these math functions will return a scalar result, will not set *errno*[2], and will not signal a NaN (a not-a-number exception value in IEEE floating-point) in any useful way. Wrapper code could be placed around the functions to produce conventional results for out-of-domain cases, for example, to produce NaN for $\log(-1)$, but for molecular dynamics, we are generally confident that they will not be asked to process out-of-domain cases, and so the extra computation involved in obtaining conventional answers is best skipped.

One way to enhance scheduling opportunities by exposing independent instructions to the compiler is to write each independent computation explicitly in the source code. Another way is to compute the same basic block repeatedly with different arguments in a counted loop and verify that the compiler can see that loop iterations are independent; the compiler then applies loop transformation optimizations, such as unrolling[3] and modulo scheduling[4], to construct the appropriate work itself. Both techniques aim to reduce stalls[5].

Vectorizable log

The function log may be vectorized by appreciating that a floating-point number is represented as an exponent k and a mantissa (also called a fraction) m; i.e., as $m \times 2^k$, for some m in [1.0, 2.0) and for integer k,

$$\ln(m \times 2^k) = \ln(m) + \ln(2^k).$$

The approximation is produced as three terms, which are added together to give the result.

The variable k is extracted as the exponent part of the argument, giving the first term of the result as $k \times \ln(2)$.

The variable m is expressed as $m0 \times m1$, where $m0$ is $1 + (a/16)$ for integer a in (0, 15), and $m1$ is $m/[1 + (a/16)]$.

The variable a is determined by extracting the first four bits after the binary point from m.

The expression $1/[1 + (a/16)]$ is looked up in a 16-element table, and this gives a value for $m1$ roughly between 1 and $[1 + (1/16)]$.

The second term of the result is $\ln(m0)$, which comes from another 16-element table.

The third term of the result comes from a Taylor series for $\ln(1 + x)$. This converges quite rapidly for $x < (1/16)$.

The full result then is

$$\ln(a) \simeq k \times \ln(2) + \text{Lookup}(a) + \text{TaylorSeries}(x).$$

An improvement comes from a slight modification, where $m1$ is arranged to be in the domain $[1 - (1/32), 1 + (1/32))$, and so the Taylor series is used for $|x| < (1/32)$.

Vectorizable exp

The function exp may be vectorized by using the relation

$$\exp(a0 + a1 + a2 + a3) = \exp(a0) \times \exp(a1) \times \exp(a2) \times \exp(a3).$$

The variable $a0$ is extracted as the integer part of the argument; $a1$ is the next four bits; $a2$ is the subsequent four bits; $a3$ is the remaining bits; $a3$ is a number between 0 and (1/256).

The variable $a0$ is shifted into the exponent of the resulting floating-point number; $\exp(a1)$ and $\exp(a2)$ are looked up in 16-element tables; $\exp(a3)$ is estimated by a Taylor series, which converges quite rapidly for $0 < a3 < (1/256)$.

Again, an improvement comes from a slight modification, setting $a3$ in the domain $[-(1/512), +(1/512))$.

IBM PowerPC* and follow-on hardware supports a floating-point "select" instruction that performs the equivalent of

```
double fsel (double a, double b, double c)
{
  if (a >=0 . 0) return b ; return c
}
```

as a single hardware instruction. This can be used to arrange that $\exp(x)$ returns 0 for a sufficiently large negative argument and Inf[6] for a sufficiently large positive argument without causing a branch in the generated code.

[1]On Blue Gene*/L, if code has dependencies such that a, b, and c must be computed in that order, with b depending on a and c depending on b, and no other work is available, the machine will deliver 10% of its theoretical peak performance. Here, the term *vectorizable* stands for assorted techniques to get closer to 100%.

[2]A global variable used to indicate which error has occurred.

[3]Grouping of multiple loop iterations so that the instructions from multiple iterations can be worked on in parallel.

[4]Software pipelining of loops—rearranging them to work on parts of more than one iteration at a time, the way a button is sewn on a shirt.

[5]Situations in which an instruction must wait before entering the processor because the calculations which produce one or more operands have not yet completed.

[6]A bit pattern representing *infinity*, or *larger than the largest representable value*, in IEEE floating-point.

Vectorizable erf/erfc—piecewise Chebyshev

Traditionally in molecular dynamics codes, $\mathrm{erfc}(x)$ has been approximated using the approximation for $\mathrm{erf}(x)$ in Section 7.1.26 of [2], related by $\mathrm{erfc}(x) + \mathrm{erf}(x) = 1$. The Abramowitz and Stegun approximation from the reference is

$$\mathrm{erf}x = 1 - (a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5) e^{-x^2} + \epsilon(x),$$

$$t = \frac{1}{1 + px},$$

$$|\epsilon(x)| \leq 1.5 \times 10^{-7},$$

$$p = 0.32759\,11,$$

$$a_1 = 0.25482\,9592,$$

$$a_2 = -0.28449\,6736,$$

$$a_3 = 1.42141\,3741,$$

$$a_4 = -1.45315\,2027,$$

$$a_5 = 1.06140\,5429.$$

Vectorizable $\exp(x)$ can be used to form vectorizable $\mathrm{erfc}(x)$ in the obvious way, but there is an alternative that can be used to form a more accurate result, which is desirable in molecular dynamics because it should give better energy conservation for a given timestep size or, alternatively, will allow a larger timestep size before numerical instability sets in.

The reciprocal required above is a special case; for molecular dynamics codes, the dividend will be in the single-precision range, and there is no point returning a result much more accurate than the one part in 10^5 of the complete approximation. This leads to a faster expression of reciprocal than the hardware double-precision divide will give (more on this below).

For molecular dynamics, we are interested in erfc to support electrostatics, $\mathrm{erfc}(x)$ for a limited domain of x, typically (–4, 4).

We partition the domain into equal-sized subdomains, say [–4, –3), [–3, –2), …, [3, 4). Represent x as $x0 + x1$, where $x1$ is in [–0.5, 0.5) and $x0$ is an integer that identifies the subdomain. Each subdomain is associated with a polynomial approximator—a set of eight Chebyshev polynomials works well.

Select the appropriate polynomial by using $x0$ to index an array, and $\mathrm{erfc}(x)$ follows.

It is relatively easy to set the polynomials up to give $\mathrm{erfc}(x)$ accurate within 1 ulp[7] over the whole domain. It is desirable to use fsel to avoid misleading results in case the function is used for a value of x outside the designed domain.

It is possible to exploit the symmetry between $\mathrm{erfc}(x)$ and $\mathrm{erfc}(-x)$ to halve the number of tables required.

[7]Ultimate limit of precision—one double-precision unit in the last place of the IEEE fraction part.

The required table for Chebyshev coefficients is machine-generated. The algorithm is shown in [3]. First, the Chebyshev coefficients for $(d/dx)\,\mathrm{erfc}(x)$ are generated using the analytic expression $(-2/\sqrt{\pi})\exp(-x^2)$. Then the coefficients for $\mathrm{erfc}(x)$ are generated by applying the appropriate transformation on these.

Vectorizable derivative erfc

Derivative erfc is $(-2/\sqrt{\pi})\exp(-x^2)$ and may be vectorized using vectorizable $\exp(x)$.

However, for molecular dynamics, it is desirable to have derivative erfc and erfc related accurately as derivative and integral of each other; this results in better reported energy conservation and better accuracy when switch or soft force cutoff is in use.

When the Abramowitz and Stegun approximation for $\mathrm{erfc}(x)$ is in use, we can differentiate the expression analytically. The derivative has an exponential term of the same form as the original, i.e., $\exp(-x^2)$, so a single evaluation of $\exp(X)$ will do duty for both functions when erfc and its derivative are both required in a computation.

When the multiple Chebyshev approach is in use, another set of Chebyshev polynomials can be used to deliver derivative erfc. If these are on the same subdomains, there is a computational economy.

Vectorizable erfc and derivative—piecewise cubic spline

In molecular dynamics, erfc and its derivative are used in the evaluation of electrostatic forces. Another approximation (particle mesh) means that it is not useful to get $\mathrm{erfc}(x)$ more precise than a relative error of about 10^{-5}; the imprecision due to the "particle mesh" approximation dominates.

However, it is important for the values returned for $\mathrm{erfc}(x)$ and its derivative to be continuous and an analytic integral/derivative pair.

This can be satisfied by approximating $(d/dx)\,\mathrm{erfc}(x)$ with a set of cubic splines, matching the $(-2/\sqrt{\pi})\exp(-x^2)$ function and its derivative at the piecewise endpoints and integrating these polynomials to give piecewise-quartic approximations for $\mathrm{erfc}(x)$. A set of 64 piecewise-cubic polynomials and their integrals, for domains [0, (1/16)), [(1/16), (2/16)), …, [(63/16), (64/16)) gives the ability to approximate $\mathrm{erfc}(x)$ and its derivative to the required precision in the domain [0–4).

Vectorizable sin and cos

It is convenient to use a multiple-Chebyshev-polynomial approach for this as well. Divide $\sin(x)$ into domains [–45, 45), [45, 135), [135, 225), and [225, 315) degrees and repeat cyclically.

In domains [–45, 45) degrees and [135, 225) degrees, use a Chebyshev polynomial for $[\sin(x)/x]$, and multiply the result by x. This arranges that the result for small $|x|$ can be within an ulp without requiring an excessive number of terms in the polynomial.

In domains [45, 135) and [225, 315), use a Chebyshev polynomial for $\cos(x)$.

The required Chebyshev polynomials are always even functions, such that $f(x) = f(-x)$. This economizes on the computation.

After the polynomial evaluation, fix up the result using a suitable multiply and add, according to the subdomain.

Since $\cos(x) = \sin(x + 90)$ with angles in degrees, cos and sin are related.

The tables are machine-generated offline, using higher-precision sin and cos functions and the algorithm in [3].

Vectorizable inverse sin and cos

Sometimes an application knows the sin and cos of an angle and wishes to evaluate the angle. Traditional arcsin involves an ambiguity as to the angle (as between 80 degrees or 100 degrees, for example), is ill-conditioned in ranges near 90 and 270 degrees, and usually involves a conditional branch and a square root.

By expressing it as

```
double acossin(double cos_angle, double sin_angle)
```

we can overcome these limitations and produce an implementation without branches.

We want to compute θ such that cosangle $= \cos\theta$ and sinangle $= \sin\theta$. Let $c = |\text{cosangle}|$, $s = |\text{sinangle}|$, and use the fsel instruction to obtain minsc $= \min(c, s)$ and maxsc $= \max(c, s)$. Then $0 < \text{minsc} < \sqrt{0.5}$ and $\sqrt{0.5} < \text{maxsc} < 1$, and there is an angle ϕ in [0, 45] degrees such that minsc $= \sin\phi$ and maxsc $= \cos\phi$.

Then we use the compound angle formula

$$\sin(a - b) = \sin(a)\cos(b) - \cos(a)\sin(b)$$

for $b = 22.5$ degrees to form the sine of an angle in [–22.5, 22.5] degrees, a value approximately in the domain [–0.38, 0.38].

Next, we use the Taylor expansion for $\arcsin(x)$, which converges quite rapidly over this domain, and we multiply by and add suitable constants (according to whether the original parameters were negated and which was smaller) to evaluate the called-for angle.

Vectorizable reciprocal square root

The natural way to express this is

```
double a=1.0/sqrt(x);
```

The IBM xlc compiler "-qnostrict" option causes this to be recognized as an idiom. There is a hardware reciprocal square root estimate instruction that gives a result accurate to five bits (POWER3)[8] or 13 bits (Blue Gene/L)[9] using lookup tables in the same amount of time that a multiply–add instruction would take; and the compiler generates a suitable number of iterations of Newton's method, or a suitable Taylor correction polynomial, to bring the result to double-precision accuracy. This avoids the division operation, and this direct "reciprocal square root" evaluation is faster than "square root" would be.

Newton's iteration is expressed in terms of multiplies and adds. The "divide by b" that seems to be required is replaced with "multiply by estimate of $1/b$." The running estimate of $1/b$ is steadily improving, so quadratic convergence is maintained.

Vectorizable square root

The compiler recognizes the use of $\sqrt{x}$ in a source program

```
double a=sqrt(x)
```

and rather than calling a function, it generates the fsqrt hardware instruction on the POWER3 processor. Blue Gene/L (BG/L) lacks this instruction, so the compiler, in effect, changes the computation to $x/\sqrt{x}$, which it implements with the help of the floating reciprocal square root estimate instruction. However, $x/\sqrt{x}$ on its own will give "not-a-number" for $x = 0$; the compiler generates additional code to handle this case correctly, but it is computationally expensive.

If the source program is not dependent on the result for $x = 0$, it will run better on both POWER3 and BG/L if coded as

```
double a=x/sqrt(x) .
```

Vectorizable nearest_image_in_periodic_volume

Molecular dynamics is frequently run with periodic boundary conditions, i.e., where we imagine that the simulation volume is surrounded by a never-ending sequence of matching simulation volumes and the interaction force between a pair of atoms is calculated as if one of the atoms is influenced by only by the nearest of the 27 images of the other atom.

To find the nearest image vector between a pair of atoms, one algorithm would remap the simulation volume to a unit cube and scale the vector appropriately, drop the integer part of the x, y, and z coordinates of the vector (each of which would be -1, 0, or $+1$), and subtract

$$\begin{Bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{Bmatrix},$$

giving a vector in

[8] Using a stepwise lookup.

[9] Using a piecewise-linear lookup, stepwise for "offset" and "slope," then passing through the multiply–add unit, which would otherwise be idle. It is better precision with the same transistor count.

$$\left\{ \begin{matrix} \pm 0.5 \\ \pm 0.5 \\ \pm 0.5 \end{matrix} \right\},$$

and rescale back to the original coordinate system.

This appears to require divisions, tests, and conditional branches, but can actually be calculated without requiring any of these.

Vectorizable nearest_integer

Vectorizable nearest_integer relies on the IEEE floating-point representation. Double precision takes 64 bits. The top bit is a sign bit, the next 11 bits are a binary exponent, and the remaining 52 bits are a binary mantissa, with an implied leading 1.

IEEE addition, with the hardware in its usual mode, is specified to round to the nearest representable number. Thus, if one takes a double-precision floating-point number and adds $(2^{52} + 2^{51})$, the fractional part is dropped. One can then subtract the $(2^{52} + 2^{51})$ and obtain the integer nearest to the number used to start.

There is a range around 2^{52} in which one obtains the nearest even integer, so this is not applicable in all cases, but is acceptable for molecular dynamics.

The compiler is being asked to generate code for $(x + k) - k$. It is important to prevent the optimizer from reassociating this to $x + (k - k)$ and then optimizing this to $x + 0$, that is, x.

The sample code does this by expressing $(x + k \times k1) \times k1 - k$, where $k1$ is 1.0, but the compiler is unable to tell that $k1$ is a constant. Since the basic floating-point instruction in the IBM Power Architecture* is multiply–add, this does not cause any extra processing cycles.

Vectorizable fragment_in_range

Molecular dynamics is generally concerned with forces between atoms in an imagined simulation box with periodic boundary conditions. Computation of the force between a pair of atoms is skipped if the atoms are more than a threshold distance apart.

For computational convenience, the atoms are grouped into fragments, typically a water molecule or a covalently bonded set of atoms within a larger molecule. The question arises, "Given fragment a, what is the set of fragments $\{b0, b1, \ldots\}$ such that an atom in a is in range of an atom in each bi, accounting for the periodic boundary?" The simulation will be functionally correct if extra fragments b are in the set, because the forces involved will evaluate to zero, but the simulation is more efficient with fewer extra fragments.

There is an algorithm for this that makes 100% use of the floating-point units (FPUs), successively slicing for slab, cylinder, and sphere.

There is another algorithm that does not use the FPUs; instead, it uses the integer units with wrap at 2^{32}, successively slicing for slab, square prism, and cube. It then uses the FPUs to slice for sphere. On POWER3 and BG/L, the integer algorithm is faster. Either algorithm is sufficiently fast that our implementation of the molecular dynamics code does not have to maintain lists of fragments (known as *Verlet lists*) that may be within a "cut-off" distance.

These algorithms show how to do "vector compress," i.e., produce a vector that is a subset of a starting vector, including only those elements matching a selection criterion, without requiring a conditional branch.

A practical example—reciprocal square roots

The reciprocal square root function evaluates the reciprocal square root for each of nine values, as would be needed to support the calculation of distances between atoms in a pair of three-site[10] water molecules.

Figure 1 shows source code and the compiler-generated assembly listing for the BG/L machine architecture. Compiler intermediate code with cycle counts and corresponding listings for POWER3 can be found at [1].

Values are copied into local variables to make it clear to the compiler what is intended if the function is called with source and target overlapping in memory.

POWER3 requires a vector of length at least 6 to keep the FPUs fully busy on this algorithm. BG/L requires a vector of length 10. In each case, the compiler finds an optimal instruction sequence; 100% floating-point utilization for POWER3 and 90% utilization (four "parallel" ops, then a "primary" op) for BG/L.

The "reciprocal square root estimate" instruction of POWER3 gives five bits of precision; that of BG/L gives 13 bits of precision. BG/L requires fewer follow-on instructions to converge the estimate to double precision. POWER3 uses a Newton–Raphson algorithm for convergence; BG/L uses a Taylor expansion.

The theoretical peak rate for each 440 processor core in the BG/L hardware is ten double-precision square roots per 40 clock cycles. By enclosing similar code in a "for" loop, it is possible to get the VisualAge* compiler to generate code that achieves within a few cycles of this rate.

Examining the machine code reveals that when a floating-point value is calculated, there are at least four other floating-point instructions between the calculation and the first use of the result. This keeps the floating-point pipeline full, allowing the FPU to operate at maximum throughput.

[10] A three-site water molecule is a model with electrostatic charges centered on the three atom locations. A five-site model has fractional electron charges at two other locations. Models run this way often match experiment more closely, and always take more computation for a simulation timestep.

```
IBM VisualAge* C++ Version 6.0.0.3 for Linux** on pSeries* ---
> > > > > OPTIONS SECTION < < < < <
IGNERRNO      ARCH=440D      OPT=3      ALIAS=ANSI      ALIGN=LINUXPPC
FLOAT=NOHSFLT:NORNDSNGL:NOHSSNGL:MAF:NORRM:FOLD:NONANS:RSQRT:FLTINT:NOEMULATE
MAXMEM=-1     NOSTRICT       NOSTRICT_INDUCTION    TBTABLE=SMALL    LIST
SHOWINC=NOSYS:NOUSR          SOURCE     STATICINLINE    TMPLPARSE=NO
NOEH
> > > > > SOURCE SECTION < < < < <
         1 | #include <math.h>
         2 | void nineroot(double* f, const double* x)
         3 | {
         4 | double x0 = x[0] ;
         5 | double x1 = x[1] ;
         6 | double x2 = x[2] ;
         7 | double x3 = x[3] ;
         8 | double x4 = x[4] ;
         9 | double x5 = x[5] ;
        10 | double x6 = x[6] ;
        11 | double x7 = x[7] ;
        12 | double x8 = x[8] ;
        13 | double r0 = 1.0/sqrt(x0) ;
        14 | double r1 = 1.0/sqrt(x1) ;
        15 | double r2 = 1.0/sqrt(x2) ;
        16 | double r3 = 1.0/sqrt(x3) ;
        17 | double r4 = 1.0/sqrt(x4) ;
        18 | double r5 = 1.0/sqrt(x5) ;
        19 | double r6 = 1.0/sqrt(x6) ;
        20 | double r7 = 1.0/sqrt(x7) ;
        21 | double r8 = 1.0/sqrt(x8) ;
        22 | f[0] = r0 ;
        23 | f[1] = r1 ;
        24 | f[2] = r2 ;
        25 | f[3] = r3 ;
        26 | f[4] = r4 ;
        27 | f[5] = r5 ;
        28 | f[6] = r6 ;
        29 | f[7] = r7 ;
        30 | f[8] = r8 ;
        31 | }
-qdebug=BGL:PLST3:CYCLES:SHUTUP:HUMMER:LINUX:NEWSCHED1:NEWSCHED2:REGPRES:ADRA:ANTIDEP:
GPR's set/used:   ssuu ssss s--- s--- ---- ---- ---- ----
FPR's set/used:   ssss ssss ssss ss-- ---- ---- ---s ssss
                  ssss ssss ssss ss-- ---- ---- ---s s-s-
CCR's set/used:   ---- ----
   | 000000                      PDEF     nineroot(double *, const double *)
 3 |                             PROC     f,x,gr3,gr4
 0 | 000000 ori          602C0000 1 LR       gr12=gr1
 0 | 000004 addi     3800FFF0 1 LI       gr0=-16
 0 | 000008 stwu     9421FFA0 1 ST4U     gr1,#stack(gr1,-96)=gr1
 0 | 00000C stfpdux  7FEC07DC 1 SFPLU    gr12,#stack(gr12,gr0,0)=fp31,fp63
 0 | 000010 stfpdux  7FCC07DC 1 SFPLU    gr12,#stack(gr12,gr0,0)=fp30,fp62
 0 | 000014 stfpdux  7FAC07DC 1 SFPLU    gr12,#stack(gr12,gr0,0)=fp29,fp61
 0 | 000018 stfpdux  7F8C07DC 1 SFPLU    gr12,#stack(gr12,gr0,0)=fp28,fp60
 0 | 00001C stfpdux  7F6C07DC 1 SFPLU    gr12,#stack(gr12,gr0,0)=fp27,fp59
 4 | 000020 lfd      C9A40000 1 LFL      fp13=(double)(gr4,0)
 5 | 000024 addi     38C00008 1 LI       gr6=8
 7 | 000028 addi     38A00018 1 LI       gr5=24
 5 | 00002C lfsdx    7DA4319C 1 LFL      fp45=(double)(gr4,gr6,0,trap=8)
 9 | 000030 addi     39000028 1 LI       gr8=40
11 | 000034 addi     38C00038 1 LI       gr6=56
13 | 000038 addis    3CE00000 1 LA       gr7=.+CONSTANT_AREA%HI(gr2,0)
 6 | 00003C lfd      C9640010 1 LFL      fp11=(double)(gr4,16)
13 | 000040 addi     38E70000 1 LA       gr7=+CONSTANT_AREA%LO(gr7,0)
 7 | 000044 lfsdx    7D64299C 1 LFL      fp43=(double)(gr4,gr5,0,trap=24)
 8 | 000048 lfd      C9440020 1 LFL      fp10=(double)(gr4,32)
13 | 00004C fprsqrte 0120681E 1 FPRSQRE  fp9,fp41=fp13,fp45
 9 | 000050 lfsdx    7D44419C 1 LFL      fp42=(double)(gr4,gr8,0,trap=40)
31 | 000054 ori      602C0000 1 LR       gr12=gr1
10 | 000058 lfd      C9040030 1 LFL      fp8=(double)(gr4,48)
31 | 00005C addi     38000010 1 LI       gr0=16
11 | 000060 lfsdx    7D04319C 1 LFL      fp40=(double)(gr4,gr6,0,trap=56)
15 | 000064 fprsqrte 00E0581E 1 FPRSQRE  fp7,fp39=fp11,fp43
```

```
13| 000068 addi     38C00020 1 LI       gr6=32
12| 00006C lfd      CBE40040 1 LFL      fp31=(double)(gr4,64)
13| 000070 lfpsx    7F672B1C 1 LFPS     fp27,fp59=+CONSTANT_AREA(gr7,gr5,0,trap=24)
13| 000074 fpmul    01890250 1 FPMUL    fp12,fp44=fp9,fp41,fp9,fp41,fcr
17| 000078 fprsqrte 00C0501E 1 FPRSQRE  fp6,fp38=fp10,fp42
13| 00007C lfs      C3C70004 1 LFS      fp30=+CONSTANT_AREA(gr7,4)
19| 000080 fprsqrte 0080401E 1 FPRSQRE  fp4,fp36=fp8,fp40
13| 000084 lfpsx    7CA7331C 1 LFPS     fp5,fp37=+CONSTANT_AREA(gr7,gr6,0,trap=32)
21| 000088 frsqrte  FFA0F834 1 FRSQRE   fp29=fp31
13| 00008C lfpsx    7C67431C 1 LFPS     fp3,fp35=+CONSTANT_AREA(gr7,gr8,0,trap=40)
13| 000090 addi     38800030 1 LI       gr4=48
15| 000094 fpmul    002701D0 1 FPMUL    fp1,fp33=fp7,fp39,fp7,fp39,fcr
13| 000098 lfpsx    7C47231C 1 LFPS     fp2,fp34=+CONSTANT_AREA(gr7,gr4,0,trap=48)
13| 00009C fpmadd   01ADDB20 1 FPMADD   fp13,fp45=fp27,fp59,fp13,fp45,fp12,fp44,fcr
17| 0000A0 fpmul    00060190 1 FPMUL    fp0,fp32=fp6,fp38,fp6,fp38,fcr
23| 0000A4 addi     38C00008 1 LI       gr6=8
19| 0000A8 fpmul    01840110 1 FPMUL    fp12,fp44=fp4,fp36,fp4,fp36,fcr
21| 0000AC fmul     FF9D0772 1 MFL      fp28=fp29,fp29,fcr
15| 0000B0 fpmadd   002BD860 1 FPMADD   fp1,fp33=fp27,fp59,fp11,fp43,fp1,fp33,fcr
17| 0000B4 fpmadd   014AD820 1 FPMADD   fp10,fp42=fp27,fp59,fp10,fp42,fp0,fp32,fcr
19| 0000B8 fpmadd   0108DB20 1 FPMADD   fp8,fp40=fp27,fp59,fp8,fp40,fp12,fp44,fcr
21| 0000BC fmadd    FFFFDF3A 1 FMA      fp31=fp27,fp31,fp28,fcr
13| 0000C0 fxcpmadd 001E2B64 1 FXPMADD  fp0,fp32=fp5,fp37,fp13,fp45,fp30,fp30,fcr
15| 0000C4 fxcpmadd 019E2864 1 FXPMADD  fp12,fp44=fp5,fp37,fp1,fp33,fp30,fp30,fcr
31| 0000C8 lfpdux   7F6C03DC 1 LFPLU    fp27,fp59,gr12=#stack(gr12,gr0,0)
17| 0000CC fxcpmadd 017E2AA4 1 FXPMADD  fp11,fp43=fp5,fp37,fp10,fp42,fp30,fp30,fcr
19| 0000D0 fxcpmadd 039E2A24 1 FXPMADD  fp28,fp60=fp5,fp37,fp8,fp40,fp30,fp30,fcr
21| 0000D4 fmadd    FCBF2FBA 1 FMA      fp5=fp5,fp31,fp30,fcr
13| 0000D8 fpmadd   000D1820 1 FPMADD   fp0,fp32=fp3,fp35,fp13,fp45,fp0,fp32,fcr
15| 0000DC fpmadd   01811B20 1 FPMADD   fp12,fp44=fp3,fp35,fp1,fp33,fp12,fp44,fcr
17| 0000E0 fpmadd   016A1AE0 1 FPMADD   fp11,fp43=fp3,fp35,fp10,fp42,fp11,fp43,fcr
19| 0000E4 fpmadd   03C81F20 1 FPMADD   fp30,fp62=fp3,fp35,fp8,fp40,fp28,fp60,fcr
21| 0000E8 fmadd    FCBF197A 1 FMA      fp5=fp3,fp31,fp5,fcr
13| 0000EC fpmadd   000D1020 1 FPMADD   fp0,fp32=fp2,fp34,fp13,fp45,fp0,fp32,fcr
31| 0000F0 lfpdux   7F8C03DC 1 LFPLU    fp28,fp60,gr12=#stack(gr12,gr0,0)
15| 0000F4 fpmadd   00611320 1 FPMADD   fp3,fp35=fp2,fp34,fp1,fp33,fp12,fp44,fcr
17| 0000F8 fpmadd   016A12E0 1 FPMADD   fp11,fp43=fp2,fp34,fp10,fp42,fp11,fp43,fcr
19| 0000FC fpmadd   018817A0 1 FPMADD   fp12,fp44=fp2,fp34,fp8,fp40,fp30,fp62,fcr
21| 000100 fmadd    FCBF117A 1 FMA      fp5=fp2,fp31,fp5,fcr
13| 000104 fpmul    000D0010 1 FPMUL    fp0,fp32=fp13,fp45,fp0,fp32,fcr
15| 000108 fpmul    002100D0 1 FPMUL    fp1,fp33=fp1,fp33,fp3,fp35,fcr
17| 00010C fpmul    004A02D0 1 FPMUL    fp2,fp34=fp10,fp42,fp11,fp43,fcr
19| 000110 fpmul    00680310 1 FPMUL    fp3,fp35=fp8,fp40,fp12,fp44,fcr
21| 000114 fmul     FCBF0172 1 MFL      fp5=fp31,fp5,fcr
13| 000118 fpmadd   00094820 1 FPMADD   fp0,fp32=fp9,fp41,fp9,fp41,fp0,fp32,fcr
15| 00011C fpmadd   00273860 1 FPMADD   fp1,fp33=fp7,fp39,fp7,fp39,fp1,fp33,fcr
17| 000120 fpmadd   004630A0 1 FPMADD   fp2,fp34=fp6,fp38,fp6,fp38,fp2,fp34,fcr
19| 000124 fpmadd   006420E0 1 FPMADD   fp3,fp35=fp4,fp36,fp4,fp36,fp3,fp35,fcr
21| 000128 fmadd    FC9DE97A 1 FMA      fp4=fp29,fp29,fp5,fcr
22| 00012C stfd     D8030000 1 STFL     (double)(gr3,0)=fp0
23| 000130 stfsdx   7C03359C 1 STFL     (double)(gr3,gr6,0,trap=8)=fp32
29| 000134 addi     38C00038 1 LI       gr6=56
31| 000138 lfpdux   7FAC03DC 1 LFPLU    fp29,fp61,gr12=#stack(gr12,gr0,0)
24| 00013C stfd     D8230010 1 STFL     (double)(gr3,16)=fp1
25| 000140 stfsdx   7C232D9C 1 STFL     (double)(gr3,gr5,0,trap=24)=fp33
31| 000144 lfpdux   7FCC03DC 1 LFPLU    fp30,fp62,gr12=#stack(gr12,gr0,0)
26| 000148 stfd     D8430020 1 STFL     (double)(gr3,32)=fp2
27| 00014C stfsdx   7C43459C 1 STFL     (double)(gr3,gr8,0,trap=40)=fp34
31| 000150 lfpdux   7FEC03DC 1 LFPLU    fp31,fp63,gr12=#stack(gr12,gr0,0)
28| 000154 stfd     D8630030 1 STFL     (double)(gr3,48)=fp3
31| 000158 addi     38210060 1 AI       gr1=gr1,96,gr12
29| 00015C stfsdx   7C63359C 1 STFL     (double)(gr3,gr6,0,trap=56)=fp35
30| 000160 stfd     D8830040 1 STFL     (double)(gr3,64)=fp4
31| 000164 bclr     4E800020 0 BA lr
  | Instruction count 90
  | Constant Area
  | 000000 BF800000 3E8C0000 BEA00000 3EC00000 BF000000 49424D20
  | 000018 BF800000 BF800000 BEA00000 BEA00000 3EC00000 3EC00000
  | 000030 BF000000 BF000000
```

Figure 1

Code tuned for Blue Gene/L.

Each 440 processor core in BG/L can dispatch two instructions per clock cycle. Each has one floating-point instruction pipeline, one load/store pipeline, two integer pipelines, and one branch pipeline. The 440 does not have "out-of-order" processing capability or "rename" registers; both of these cost transistors and electrical energy, which the BG/L design puts to better use elsewhere. Therefore, we are dependent for good performance on the ability of the compiler to schedule instructions and allocate registers in the optimal patterns for the real hardware. The compiler effort to exploit the two-instructions-per-cycle capability can be seen in the assembly fragment shown in the figure.

IBM Power Architecture defines 32 double-precision floating-point registers. Floating-point operations, in general, work on three operand registers and a result register. For example, "floating-point multiply–add" might evaluate $f_1 = f_2 + (f_3 \times f_4)$ in a single pass through the FPU. The Blue Gene/L chip has an additional 32 double-precision floating-point registers, an additional FPU, and extensions to the instruction decoder to implement "parallel" versions of these, such as

$$f_1 = f_2 + (f_3 \times f_4),\ s_1 = s_2 + (s_3 \times s_4)$$

and various "cross" versions, such as

$$f_1 = f_2 + (f_3 \times f_4),\ s_1 = s_2 + (s_3 \times f_4)$$

or

$$f_1 = f_2 + (f_3 \times s_4),\ s_1 = s_2 + (s_3 \times f_4),$$

and an "antisymmetric" version,

$$f_1 = f_2 + (f_3 \times f_4),\ s_1 = s_2 - (s_3 \times s_4).$$

Several of these can be seen in the assembly fragment shown in Figure 1.

Conclusion

When we started designing the protein folding application, we imagined that we would be unable to fully exploit the floating-point capacity of a modern uniprocessor because of the sequential nature of the scalar library functions, which we expected would limit the performance of the application. This would limit the fraction of peak flops that we would achieve on the massively parallel machine we had in mind.

Working with the life scientists on the actual requirements of the application and with the compiler programmers on optimization capabilities has resulted in techniques for evaluating the required functions and presenting the machine with sufficient independent work that we, in fact, achieved a high fraction of peak flops on a uniprocessor. This is expressible as source code in a high-level language, such as C++; it has not been necessary to hand-code anything in assembler.

Knowing that we can well exploit a uniprocessor, we are motivated to use the machine efficiently as we take the next steps to scale up to the 40,960 processors that will be available to us in our contribution to the "grand challenge" of understanding the Protein Economy.[11]

Blue Gene/L can do more than just lead the world at linear algebra. It takes some thought, but the results are worth the effort.

Acknowledgments

The Blue Gene/L project has been supported and partially funded by the Lawrence Livermore National Laboratory on behalf of the United States Department of Energy under Lawrence Livermore National Laboratory Subcontract No. B517552.

References

1. See *http://www.research.ibm.com/bluegene/jrd_2005/cust_math/index.html*.
2. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, M. Abramowitz and I. A. Stegun, Eds., U.S. Department of Commerce, Washington, DC, 1972.
3. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, 1992.

Received May 5, 2004; accepted for publication August 17, 2004; Internet publication April 12, 2005

[11]Proteomics: the Protein Economy. Deoxyribonucleic acid (DNA) stores information. We know; we have sequenced it. Ribonucleic acid (RNA) copies information. We know; we can make it happen in the laboratory. Ribosome (a protein) reads the RNA and assembles a protein out of amino acids according to the recipe it has read. We know; every living thing on earth works in much the same way. The protein folds into its equilibrium structure, more or less, quickly or slowly. We're curious about this. Function follows form, and all the diversity of life happens.

Robert F. Enenkel *IBM Software Group, Toronto Laboratory, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 (enenkel@ca.ibm.com).* Dr. Enenkel currently works in the Optimizing Compiler Group at the IBM Toronto Laboratory; he was previously a Research Associate at the IBM Centre for Advanced Studies (CAS). He worked at IBM on the development of a C compiler and its math library, and developed parallel methods for random-number generation for Fortran and high-performance Fortran compilers prior to joining the CAS. He received his B.S., M.S., and Ph.D. degrees from the University of Toronto, with thesis work in the area of numerical methods for the parallel solution of initial value problems for ordinary differential equations. He currently performs research and development in numerical computing as it relates to compilers and operating systems, including floating-point arithmetic, mathematical function libraries, and the performance tuning of algorithms. He is also interested in parallel computing and the application of numerical methods to practical problems in various areas of science. He has received two IBM Invention Achievement Awards and several IBM Author Recognition Awards. Dr. Enenkel is a member of the Society for Industrial and Applied Mathematics. More information may be found on his Web page at *https://www-927.ibm.com/ibm/cas/toronto/people/members/renekel.shtml*.

Blake G. Fitch *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (bgf@us.ibm.com).* Mr. Fitch joined the IBM Thomas J. Watson Research Center in 1985 as a student. He received his B.S. degree in computer science from Antioch College in 1987 and remained at IBM to pursue interests in parallel systems. He joined the Scalable Parallel Systems Group in 1990, contributing to research and development that culminated in the IBM scalable parallel system (SP*) product. His research interests have focused on application frameworks and programming models suitable for production parallel computing environments. Practical application of this work includes contributions to the transputer-based control system for the IBM CMOS S/390* mainframes (IBM Boeblingen, Germany, 1994) and the architecture of the IBM Automatic Fingerprint Identification System parallel application (IBM Hursley, UK, 1996). Mr. Fitch joined the Blue Gene project in 1999 as the application architect for Blue Matter, a scalable molecular dynamics package.

Robert S. Germain *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (rgermain@us.ibm.com).* Dr. Germain manages the Biomolecular Dynamics and Scalable Modeling Group within the Computational Biology Center at the IBM Thomas J. Watson Research Center. He received his A.B. degree in physics from Princeton University in 1982 and his M.S. and Ph.D. degrees in physics from Cornell University. He joined the Thomas J. Watson Research Center as a Research Staff Member in the Physical Sciences Department after receiving his doctorate in 1989, and later the VLSI/Scalable Parallel Systems Packaging Department. Dr. Germain was project leader, from 1995 to 1998, for the development of a large-scale fingerprint identification system using an indexing scheme (FLASH) developed at IBM Research. He has been responsible for the science and associated application portions of the Blue Gene project since 2000. His current research interests include the parallel implementation of algorithms for high-performance scientific computing, the development of new programming models for parallel computing, and applications of high-performance computing to challenging scientific problems in computational biology. Dr. Germain is a member of the IEEE and the American Physical Society.

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gustav@us.ibm.com).* Dr. Gustavson leads the Algorithms and Architectures project in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. degree in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute. He joined IBM Research in 1963. One of his primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. Dr. Gustavson has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for the IBM Power Family* of processors for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed and shared memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Invention Achievement Award, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award. He is a Fellow of the IEEE.

Allan Martin *IBM Software Group, Toronto Laboratory, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 (armartin@ca.ibm.com).* Mr. Martin graduated from the University of Toronto with a B.S. degree in engineering science in 1999. He has worked in compiler back-end development since 1999, and has expertise in the area of modulo scheduling and other loop optimizations. He has implemented and continues to develop a version of swing modulo scheduling in the compiler that includes a number of algorithm enhancements that help to achieve near-optimal performance.

Mark Mendell *IBM Software Group, Toronto Laboratory, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 (mendell@ca.ibm.com)* Mr. Mendell graduated from Cornell University in 1980 with a B.S. degree in computer engineering. He received his M.S. degree in computer science from the University of Toronto in 1983. At the University of Toronto, he helped to develop the Concurrent Euclid, Turing, and Turing Plus compilers and worked on the Tunis operating system project. In 1991 he joined IBM, working initially on the AIX* C++ compiler from V1.0 to V5.0. He has been the team leader for the TOBEY Optimizer Group since 2000. Mr. Mendell implemented the automatic compiler support of the dual FPU for the BG/L project.

Jed W. Pitera *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (pitera@us.ibm.com).* Dr. Pitera is a Research Staff Member in the Science and Technology Department at the IBM Almaden Research Center. His research focuses on the use of computer simulation to address questions in biology and chemistry, particularly in the areas of protein folding, molecular recognition, and self-assembly. He received undergraduate training in biology and chemistry at the California Institute of Technology, where he worked in Dr. Pamela Bjorkman's Protein Crystallography Group. He subsequently pursued graduate studies in biophysics at the University of California at San Francisco (UCSF) in the laboratory of Dr. Peter Kollman. Dr. Pitera developed an interest in the use of biomolecular simulation and free-energy calculations in the rational design of proteins and pharmaceuticals while in Dr. Kollman's group. He pursued similar work in a postdoctoral

position with Prof. Dr. Wilfred van Gunsteren at the Swiss Federal Institute of Technology Zurich (ETH), where his research focused on novel methods of calculating free energies for ligand design. He has worked as a member of the IBM Blue Gene Project Science and Application team since February of 2001. Dr. Pitera is also an adjunct assistant professor in the UCSF Department of Pharmaceutical Chemistry.

Michael C. Pitman *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (pitman@watson.ibm.com)*. Dr. Pitman received his Ph.D. degree in chemistry in 1995 from the University of California at Santa Cruz. He joined the Biomolecular Dynamics and Scalable Modeling Group within the Computational Biology Center at the IBM Thomas J. Watson Research Center soon afterward and continued work in the area of computational drug design methods. He began a leading role in the Blue Gene Protein Science program in 2001, focusing on large-scale membrane and membrane protein simulation. His research interests are focused on understanding the nature of protein–membrane interactions. Dr. Pitman conducts large-scale all-atom simulations of membrane proteins in explicit, biologically relevant environments.

Aleksandr Rayshubskiy *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, New York 10598 (arayshu@us.ibm.com)*. Mr. Rayshubskiy received an M.E. degree in computer science from Cornell University in 2002. He worked in the Biomolecular Dynamics and Scalable Modeling Group within the Computational Biology Center at the IBM Thomas J. Watson Research Center in 2000 as an intern, joining the group as a full-time software engineer in 2003. Mr. Rayshubskiy worked primarily on the development of the Blue Matter molecular dynamics package. His current research interests include parallel applications, load balancing, performance tuning, and lower-level hardware interfaces to the application.

Frank Suits *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (suits@us.ibm.com)*. Dr. Suits is a member of the Biomolecular Dynamics and Scalable Modeling Group within the Computational Biology Center at the IBM Thomas J. Watson Research Center. This group is responsible for the software and science involved in the protein simulations that are integral to the Blue Gene project. Although his degree is in optical physics, he has worked on a wide variety of projects at the IBM Thomas J. Watson Research Center, including optical storage, magnetic storage materials, scientific visualization, and queuing systems. At present, Dr. Suits is focusing on the analysis of the protein and membrane simulations currently running on BG/L.

William C. Swope *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (swope@almaden.ibm.com)*. Dr. Swope has been engaged with the IBM Blue Gene Protein Science Project since 2000, with strong emphasis on biomolecular simulation methodology and the development of practical techniques to simulate protein folding kinetics and thermodynamics. He joined the Science and Technology Department in 1992 at the IBM Almaden Research Center, where he has also been involved in scientific software development for computational chemistry applications and in technical data management issues related to life sciences. He began with IBM in 1982 at IBM Instruments, Inc., an IBM subsidiary that developed scientific instrumentation, where he worked in an advanced processor design group. He also worked for six years at the IBM Scientific Center in Palo Alto, California, where he supported scientific customers of IBM in their development of software for numerically intensive computation. He received his undergraduate degree in chemistry and physics from Harvard University and his Ph.D. degree in quantum chemistry from the University of California at Berkeley. He then performed postdoctoral research on the statistical mechanics of solvation and condensed phases in the chemistry department at Stanford University. Dr. Swope maintains a number of scientific relationships and collaborations with academic and commercial scientists involved in the life sciences, specifically related to drug development.

T. J. Christopher Ward *IBM United Kingdom Limited, Hursley House, Hursley Park, Winchester, Hants SO21 2JN, England (tjcw@uk.ibm.com)*. Mr. Ward graduated from Cambridge University in 1982 with a first-class honors degree in electrical engineering. He has worked for IBM in various hardware and software development roles, always finding ways of improving performance of products and processes. He was a member of the IBM Computational Biology Center at the IBM Thomas J. Watson Research Center from 2001 to 2004, arranging for the Blue Gene/L hardware and compilers and the Blue Matter protein folding application to work effectively together and achieve the performance entitlement. Mr. Ward currently works for IBM Hursley as part of the IBM Center for Business Optimization, enabling customers of IBM to take advantage of the opportunities afforded by the rapidly decreasing cost of supercomputing services.