Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

A New High Precision Floating Point Library for the Cell B.E. Processor

Jonathan Coles, François Gaignat, Mauro Calderara, Denis Nordmann and Wesley P. Petersen

Research Report No. 2008-06
May 2008

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

A New High Precision Floating Point Library for the Cell B.E. Processor

Jonathan Coles, François Gaignat, Mauro Calderara, Denis Nordmann and Wesley P. Petersen

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

Research Report No. 2008-06 May 2008

Abstract

We have developed a high precision library based on a proposed IEEE format for quad-precision numbers. The library uses the unique vector machine instructions found on the IBM Cell B.E. processor to efficiently manipulate quad word data. Usage, performance, and verification of results are described.

1 Introduction

The need for higher precision is often encountered in fields ranging from astrophysics to experimental mathematics, where both dynamic range and high precision are often required. In particular, condition numbers $\kappa(A)$ of matrices often grow quickly with the problem size. For example, a simple finite difference discretization of the second order differential $-y''(t)$ produces a tridiagonal matrix with 2 for each diagonal element and -1 for each sub- and super-diagonal element. The condition $\kappa(A)$ of this matrix for large n grows like $n^2/2+n$, so for $n =$ a few million, the number of digits lost solving a linear system $Ay = b$ with this matrix, roughly $\log_{10}(\kappa(A))$, quickly ruins the total accuracy of double precision (~ 16 digits). Unfortunately, this is relatively slow growth. More ill-conditioned problems are common, as we will revisit in one of our test examples below. Standard hardware rarely supports floating point numbers with more than fifty three bits precision (64-bit double). In response to the demand, several software libraries have been created to allow for higher or arbitrary precision. Notably, the DD [6] and QD [6] libraries combine two or four double precision numbers, respectively. The ARPREC [1] library allows for arbitrarily precise numbers. Both are highly portable and rely on the underlying machine architecture for handling the floating point computations. Some Sun machines provide a quad precision software library and the GNU MP library [5] has functions for both arbitrary precision and extended precision numbers.

The Cell processor provides a unique, high performance platform for scientific computing, but there is little hardware support for floating point numbers numbers with higher precision than single. We are impressed with the computational power of the Cell processor for scientific purposes, but note that its potential for ever larger problems compels us to consider higher precision arithmetic.

We have developed a quad precision number library specifically targeted for the Cell processor based on an IEEE format. The computation is performed entirely in software, but uses the unique vector instruction set only available on the Cell. The library is competitive with other libraries mentioned before, while often providing more precision for a given number of bits.

The Cell Broadband Engine (CBE), as it is technically known, is an important innovation in the current trend of multi-core CPUs. The Cell consists of a master PowerPC Processor Element (PPE) which controls eight Synergistic Processor Units (SPUs) [2]. The PPE is a modified version of the PowerPC found in IBM servers. The SPUs are optimized units designed for manipulating large amounts of data, particularly when that data is in the form of vectors. Although the SPUs forgo out-of-order execution, branch prediction, and register renaming for flexible dynamic instruction execution, their ability to process vector data is very attractive for scientific purposes.

2 Technical Aspects

The high-precision floating point number library HFNLIB has implemented the four basic operations of addition, subtraction, multiplication, and division. The next version will have square root, exponentiation, logarithm, and trigonometric functions. The *hfn* format is 128 bits wide and fits into a single 128-bit Cell vector register, so we are able to use many of the bit manipulation intrinsics that exist on the processor [8].

In an attempt to use as much of the IEEE compatible double precision as possible, *hfn* numbers are stored in the format depicted in Figure 1. There is one sign bit, an eleven bit exponent (as in IEEE double), an assumed normalization bit, and a 116-bit mantissa. If a is an *hfn* then a_s, a_e, and a_m represent the sign, exponent, and mantissa, respectively.

We do not implement all the features typical of an IEEE floating point standard. In particular, there is no support for `NaN`, `Inf`, overflow, underflow, and un-normalized numbers. The rounding mode is "round to zero", so the last bits may not be accurate. This is similar to the behavior found on many graphics cards today and consistent with the Cell single precision 32-bit arithmetic. The trade-off in a slight loss of accuracy boosts the overall performance.

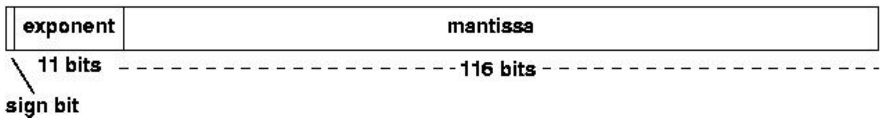

Figure 1: The format of an *hfn* is compatible with the IEEE format for double precision floating point numbers. Because it is 128 bits wide, the number fits completely into a 128-bit vector register on the Cell processor.

The core library is implemented in C but we make heavy use of the intrinsics [8] that are provided for the Cell processor. In this way, we are able to make use of the 128-bit opcodes as well as the unique shuffle opcodes.

We also provide a C++ class wrappers that overloads the standard arithmetic operators. Software that is already written using either standard floating pointing numbers or another high precision library should be able to switch to the *hfn* library with minimal code changes. There is no FORTRAN interface available at this time but we plan to insert our library into the framework provided by the DD library, which provides FORTRAN wrapper functions.

The software was developed on the Cell processors found in the Sony Playstation 3 game console. This was a cheap alternative to the Cell blade servers. For software development this proved to be an adequate platform, but there are serious limitations for more serious work. Namely, the system has limited memory and only a gigabit ethernet network connection. Also, the processor itself has only seven of the eight SPUs available. None of these limitations affected the current software development, but for future work that will parallelize the library either over all of the SPUs or across machines via MPI, a new more advanced machine will be necessary.

2.1 Addition and Subtraction

The addition or subtraction of two *hfn*'s a and b is fairly straightforward. We assume that $|a| > |b|$. If this is not the case then the values are swapped.

To add two numbers the exponents must match. The bits of b_m are shifted to the right by $a_e - b_e$ and the hidden one inserted explicitly into b_m. Bits shifted off the right side are lost. Using SIMD instructions the mantissas are added

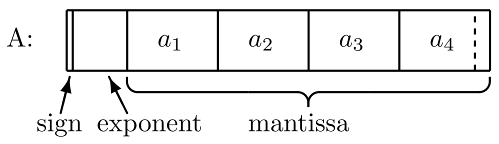

Figure 2: Partitioning of the *hfn* word for computation of multiplication.

or subtracted. If there is a carry or borrow, the exponent is incremented or decremented by one. When we subtract two numbers ($a_s \neq b_s$), then there may be leading zeros in the mantissa. In this case, the mantissa is shifted to the left until the first one becomes the hidden normalization bit. This renormalization also requires the exponent to be adjusted.

2.2 Multiplication

Multiplication is much more involved, but the basic idea of the multiplication is to split the mantissa, multiply the parts and add the products. These multiplications and additions involve smaller numbers and can be done directly in hardware if the parts are small enough.

The sign and the exponent of the result are not difficult to compute: the sign is the last bit of the sum of both signs and the exponent is also the sum of both exponents, or this number +1 (we will see later which condition determines this number).

The mantissa is divided into 4 parts. Each part is stored in a 32-bit variable. The first part (a_1, b_1 or c_1) contains the 31 most significant bits (including the *implicit* normalization bit) and is therefore smaller than 2^{31}. The other parts contain 30 bits, thus smaller than 2^{30}. The least significant part will contain four 0's at the end (in binary notation), because the mantissa in an *hfn* number just contains 117 bits (116 bits + one hidden normalization bit = 117 bits). The four parts of the mantissa contain 121 bits: the four extra bits at the end are just used for the calculation and are not stored in the *hfn* format. Figure 2 shows the representation.

The parts of the mantissa $a_1, a_2, \ldots b_4$ are extracted from the number with shift and logical AND operations: the shift operation aligns the bits and the AND operation clears all the bits that are not needed. The first part (a_1) also needs an other operation: a logical OR to set a bit, the *hidden 1* normalization.

Then the products are computed. The first part of the number A (a_1) is multiplied with all parts of B: $a_1 * b_1, a_1 * b_2, a_1 * b_3$ and $a_1 * b_4$. The second part of A is just multiplied with b_1, b_2 and b_3, the third part of A with b_1 and b_2 and the fourth part of A with b_1.

The next step is to combine the products in order to get the bits that will form the mantissa of the result.

If the mantissa is represented as an integer, the product of both mantissas is given by the following expression:

$$\sum_{i=2}^{8} \sum_{j=1}^{4} a_j * b_{i-j} * 2^{30(8-i)}$$

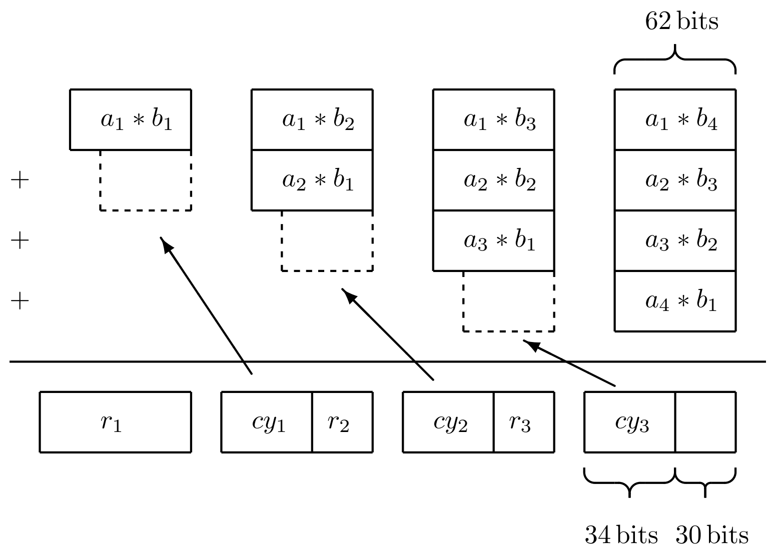

Figure 3: Carry operations in multiplication.

where b_5, b_6 and b_7 are 0. The sum of the indices of both factors of a product determine the weight of the product in the result.

The result that will be computed is just an approximation of the exact result. The low order terms will not be computed. that means that the number that will be computed just contains the high-order bits of the product of both mantissas. The last bits of this number may be wrong because the carry of the low-order part is not computed.

The products with a given weight are added and the resulting sum is split into two parts: the low-order bits are a part of the result and the high-order bits will be added to the next sum as shown in Figure 3.

Whenever new terms are added, the 34 most significant bits of the sum are added to the next sum. These bits are represented by cy_i in the picture ($i = 1, \ldots 3$). They play the role of a carry. The other bits are a part of the final mantissa. The bits of the result are represented by the number $r := r_1 * 2^{60} + r_2 * 2^{30} + r_3$.

The numbers a_1 and b_1 are at least $O(2^{30})$, which means that the number r is at least $2^{60} * 2^{60} = 2^{120}$. And the number r contains at most 122 bits, because $a_1, b_1 < 2^{31} \Rightarrow a_1 * b_1 < 2^{62}$.

The most significant 117 bits, beginning with the most significant 1, will determine the mantissa of the result. Because $r \geq 2^{120}$, the most significant 1 is the bit number 121 or 122 (if the bits are numbered from 1 to 122, from the least significant one to the most significant one). The value of the bit 122 will be called d. It will be necessary for computing the mantissa and the exponent of the result.

If d is 1, the bits will be shifted one position to the right, so that the two most significant bits (122 and 121) are 0 and 1. The 1 in the bit 121 is the

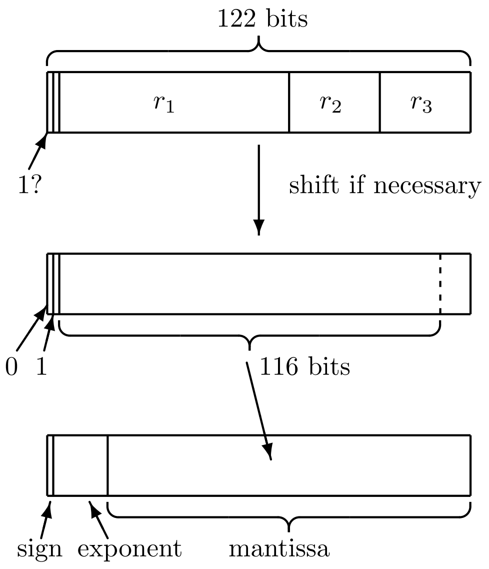

Figure 4: Final assembly of the *hfn* floating point number.

hidden 1. It will not be stored, but the next bits will: the 116 bits that will be written in the mantissa are the bits number 5 to 120. The 4 bits that will not be stored should "absorb" the error that is caused by the lower-order bits that are not computed.

The value of the exponent depends on d. If it is 1, then the exponent of the result is $e_A + e_B + 1$, otherwise, if $d = 0$, the exponent is $e_A + e_B$ (where e_A is the exponent of A and e_B the exponent of B, without the bias).

It is easier to compute the sign of the result: it is just $s_A \oplus s_B$ (where s_A is the sign of A, s_B the sign of B). The symbol $\oplus$ represents the XOR operation.

The exponent and sign are computed as shown in Figure 5.

2.3 Accuracy of Multiplication

Both mantissas are multiplied by multiplying the parts of A and B and adding the results. Since each part contains a number smaller than 2^{31}, the product of two such numbers is smaller than 2^{62} and can therefore be stored using at most 62 bits. It is possible to add 4 such products and store the result in 64 bits: $4 \cdot 2^{62} = 2^{64}$.

The number coded in the 121 bits of the extended mantissa used for the calculation (for the number A) is $a_1 2^{90} + a_2 2^{60} + a_3 2^{30} + a_4$. The product with the corresponding number for B is:

$$(a_1 2^{90} + a_2 2^{60} + a_3 2^{30} + a_4)(b_1 2^{90} + b_2 2^{60} + b_3 2^{30} + b_4)$$

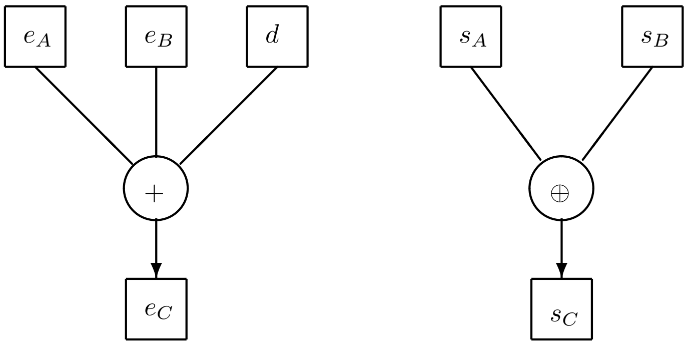

Figure 5: Exponent and sign construction.

$$= \sum_{i,j=1}^{4} a_i b_j 2^{30(8-i-j)}$$

This is the exact product. For performance reasons, not all the terms will be computed: only the 117 most significant bits will be stored in the *hfn* number. Therefore it is possible not to compute the least significant terms without making an error.

The sum can be split into two sums:

$$\sum_{i,j=1}^{4} a_i b_j 2^{30(8-i-j)} = \sum_{\substack{i,j=1 \\ i+j\leq 5}}^{4} a_i b_j 2^{30(8-i-j)} + \sum_{\substack{i,j=1 \\ i+j>5}}^{4} a_i b_j 2^{30(8-i-j)} \tag{1}$$

The sum on the left hand side (that will be called S) is bounded by $2^{240} \leq S < 2^{242}$. This sum needs 241 or 242 bits to be stored. The mantissa of an *hfn* number contains 117 bits. These bits are the most significant bits, i.e. the bits 126 to 242 or 125 to 241 of S (the bits will be numbered from 1 for the least significant bit to 242 for the most significant bit).

It is possible to find an upper bound for the second sum on the right hand side of equation (1). $1 \leq i,j \leq 4$ implies that $i+j > 5 \Rightarrow i,j \geq 2 \Rightarrow a_i b_j < 2^{60}$. The following equation shows an upper bound:

$$\sum_{\substack{i,j=1 \\ i+j>5}}^{4} a_i b_j 2^{30(8-i-j)} < 4 \cdot 2^{120} \tag{2}$$

The maximal error if the second sum on the right hand side is neglected is: $e_{max} < 4 \cdot 2^{120} = 2^{122}$. This means that the maximal error is smaller than half the weight of the least significant stored bit. If the result would be “rounded to nearest” the maximal error would be less than the weight of the last bit, but that does not mean that the result would always be exact.

In the current implementations, the result is “rounded to zero” the value in the extension of the mantissa is ignored. Therefore, the maximum difference between the stored number and the exact result is the sum of the error calculated above and the maximal value that can be stored in the extra bits. That means that the maximal error after a multiplication is less than twice the weight of the least significant bit stored in the *hfn* format.

2.4 Division

Because $z/y = z \cdot (1/y)$, division is straightforward using a Newton method and reciprocal approximation to get $1/y$. Writing

$$f(x) = x^{-1} - y,$$

the solution to $f(x) = 0$ will be $x = y^{-1}$, the desired reciprocal. Using the Cell IEEE double precision to compute $x_0 \approx y^{-1}$, two Newton steps in *hfn* of

$$x_{k+1} = 2 \cdot x_k - x_k \cdot y \cdot x_k$$

easily suffices to yield $x = x_2$ accurate to 116 bit precision. This iteration only requires *hfn* addition and multiplication. It should be noted that the initial approximation only needs the first half of the *hfn* number y since the exponent format is that of double.

3 Verification and Performance

We also developed performance and verification suites. This enables us to compare new technical improvements to the code with previous versions and competing libraries while ensuring that the code is correct.

The performance tests show that the HFNLIB compares favorably with the DD library. DD uses two double precision numbers to represent a single number. Unfortunately, the bits used for the exponent in the second double are unused so DD does not quite achieve quad precision like the HFNLIB. Nevertheless, DD still provides a good baseline for our comparisons, since it is often used in production code. All our results are normalized to the performance of hardware double. This shows immediately the performance hit a code will take using HFNLIB or any other software library. We compare the addition and multiplication with DD. The results are shown in Figure 6.

Our test suite covers the following three areas:

1. Testing specific corner cases where specially crafted input values trigger code flow changes or special case handling.

2. Testing operations using random numbers as input and comparing the result to a reference-implementation. Obviously, this leads to a dependence on the correctness of a third party's work but provides a simple sanity check. We tested against the DD library.

3. Testing operations with specially crafted input where the results are known or simple to verify. Examples include $a - a = 0$, $b/b = 1$, $b(a_1 + a_2) = ba_1 + ba_2$ for all a and b, This method was used for correctness testing in [1, 6].

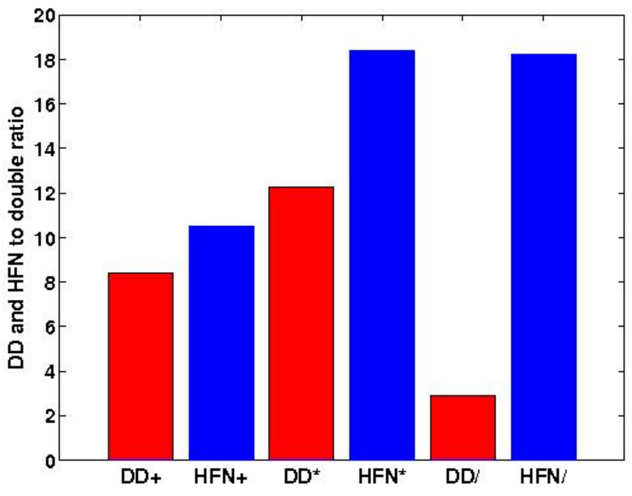

Figure 6: Timing performance of the HFNLIB and DD [6] relative to double precision. Tests were conducted on the PS-3. To scale to Mflops, use the reciprocal of these factors to rescale the double Mflops rates: 24.5 (sum), 27.2 (multiply), and 1.16 (divide). Library DD has approximately 106 bits of precision, HFNLIB has 116 bits.

4. Consistent with our objective of evaluating the Cell processor for scientific purposes, a simple iterative refinement test was implemented [7]. The idea is to solve a poorly conditioned linear system in double precision, followed by a couple of iterative refinement steps combining double and *hfn* precision. Although Moler's analysis shows linear convergence only when matrix A is not horribly conditioned, the method nevertheless shows improvement even for the Hilbert matrix. This is not a contrived example. If the first n moments of a distribution function, say $p(x)$, $0 \leq x \leq 1$, are known, call them $m_0, m_1, \ldots, m_{n-1}$, where $m_k = \int p(x)x^k dx$, trying to determine the coefficients $\{c_k\}$ to approximate $p(x) = \sum_{k<n} c_k x^k$ gives the Hilbert matrix ($A_{j,k} = 1/(j+k+1)$, for $0 \leq j,k < n$). One first solves $Ac = m$ in double precision, then successively computes refinements $c \to c + \delta c$, where $A\delta c = \delta m$ is solved in double precision. The right hand side $\delta m = m - Ac$ is computed using *hfn* from the previous iteration c. This test is also available in our test suite: it is nearly the Linpack benchmark, in C, modified to use rank-1 updates instead of DAXPY operations [3, 4], plus iterative refinement.

The 128-bit random numbers used in the test suite were generated by a relatively crude algorithm that generates three 64-bit random numbers, normalizes them to a certain range of order of magnitude each and then adding them within the high-precision libraries. It is worth investigating whether this algorithm generates suitable random numbers.

4 Summary and Future Work

We have implemented a 128-bit IEEE format floating point library for the IBM Cell processor. Using the unique features of the Cell, we have implemented fast and efficient functions for the basic arithmetic operations. With these functions alone, all other higher level functions can be constructed. It will be necessary, however, to provide more optimized versions to be of more practical utility. Although the 128-bit vector registers were used, more optimization is needed. In particular, because the SPUs have 128 such vector registers, multiple operand pairs can be computed concurrently without register spill. We also plan to implement exponentiation and logarithm, square root, and trigonometric functions. Although many optimizations of Hfnlib remain to be implemented, it does provide 116 bits of precision.

On ftp://ftp.math.ethz.ch/users/wpp/hfnlib the latest version of Hfnlib will be updated as optimizations and features are added.

5 Acknowledgments

The authors would like to thank Professors George Lake, Kaspar Nipp, and Rolf Jeltsch for funding, support, and encouragement. Particular thanks also go to Dr. David Bailey and John Shalf at the Lawrence Berkeley Laboratory for their time, expertise, as well as their kind hospitality at NERSC.

References

[1] David H. Bailey, Yozo Hida, Xiaoye S. Li, and Brandon Thompson. Arprec: An arbitrary precision computation package. *LBNL-53651*, September 2002.

[2] Ibm cell broadband engine programming handbook, version 1, April 19, 2006. http://www-128.ibm.com/developerworks/cell/.

[3] J.J. Dongarra. Performance of various computers using standard linear equations software. http://netlib.org/benchmark/performance.ps.

[4] J.J. Dongarra, J. Du Croz, S. Hammerling, and R.J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM TOMS*, 14:1–17, 1988.

[5] Gmp, 2007. http://gmplib.org/.

[6] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. *LBNL-46996*, October 2000.

[7] Cleve B. Moler. Iterative refinement in floating point. *JACM*, 2:316–321, April 1967.

[8] W. P. Petersen and P. Arbenz. *Introduction to Parallel Computing.* Oxford Univ. Press, 2nd printing edition, 2004.