Octuple-precision floating point on Apple G4

R. Crandall, Advanced Computation Group, Apple Computer

and

J. Papadopoulos, University of Maryland College Park

Abstract. We describe herein a G4 Velocity Engine implementation of "oct-precision," i.e. 256-bit floating-point operations. (We speak of 32-bit exponents and 224-bit mantissas.) We present performance benchmarks in comparison to an existing C++ library. The basic result is that Altivec-based oct-precision can run about 4x faster than a scalar implementation of the same precision; a 500 MHz. G4 can therefore perform at 5-10 Mocts (million oct-ops per second).

8 May 2002

1. Introduction

Many problems in number theory and the computational and physical sciences, especially in recent times, require more floating point precision than is commonly available in fundamental computer hardware. For example, the new science of "experimental mathematics," whereby algebraic truths are foreshadowed, even discovered numerically, requires much more than single (32-bit) or double (64-bit) precision.

High-precision floating point is therefore typically implemented as a software library designed to generate correct floating point results as quickly as possible. The computational speed that such a library can attain depends critically on the inherent precision. Most general-purpose microprocessors now contain hardware that implements floating point arithmetic with 53-bit-mantissa double precision. [Briggs] describes a *quad* precision library that works with pairs of doubles at a time and so offers twice as many bits of accuracy. This level of precision is a good compromise, since it is enough for many applications and is only(!) an order of magnitude slower than dedicated hardware.

Applications that demand even higher precision can make use of an *oct*- precision floating point library such as the one described in [Bailey]. Here the basic datatype is a group of four double-precision numbers, allowing 212-bit accuracy but at speeds an order of magnitude slower than that of quad-precision.

In principle, since floating point arithmetic reduces to fixed point operations, any large-integer package can be used to create a corresponding floating point package. In practice, the amount of precision desired is far below the range in which commonly available large-integer packages operate comfortably, and is much too small for the asymptotically faster algorithms that such packages often employ. Applications that perform large amounts of high-precision arithmetic (multiplication of large matrices of high-precision values, say) must therefore be able to take advantage of a custom library that is hardwired to perform floating point arithmetic at a given level of precision.

This paper describes one such floating point library that is written for the Apple G4. The result is a set of oct-precision computational primitives that offers much higher than quad-precision accuracy at three to four times the speed of the C++ implementation described in [Bailey].

2. Design of oct-precision library

The first decision to make regarding such a library is the precision which internal computations will use. One would be hard-pressed to improve upon the implementation of [Briggs] for quad-precision operands, and the scalar floating point unit of the G4 can already achieve high speed at this precision level. The natural next step is therefore to consider an implementation of Bailey's oct-precision algorithm that is tuned to the G4 Altivec unit.

Unfortunately, there are several technical hurdles to such an approach. Bailey uses four double-precision values at a time to achieve 212-bit precision, but the Altivec unit only deals with single-precision floating point (24-bit mantissa). A vector of four single precision values allows 96 bits of precision and an 8-bit exponent, both of which are somewhat limited. Further, while portions of Bailey's algorithm are well suited to a vector architecture, other portions are scalar with many undesirable control dependencies. Finally, Bailey's methods require support for IEEE compliant arithmetic, which is nontrivial to implement.

We therefore elected to use integer operations to emulate floating point arithmetic at high precision. Our 256-bit oct-precision format includes room for a 32-bit exponent and 224-bit mantissa. Rather than emulate conventional IEEE formats described in [HennPat], we use a modified format designed to simplify the implementation.

More precisely, oct-precision numbers are assumed to be of the form $m \cdot 2^e$, where e is a signed 32-bit exponent (unbiased) and m is a 224-bit twos-complement fraction with 222 bits to the right of the binary point. The two bits to the left of the binary point consist of a sign bit and an explicit leading one. These first two bits are 01_2 for positive floating point values and 10_2 for negative values. Zero is represented as a mantissa and exponent of zero. This format allows 222 bits of precision, and a single oct-precision value can fit into two Altivec 128-bit registers.

Let $e(a)$ and $m(a)$ denote the exponent and mantissa, respectively, of a floating point number a. An algorithm for addition of floating point numbers a and b is as follows (subtraction is similar):

```
if  $(e(a) \geq e(b))$
    $c = a$;  $d = b$;
else
    $c = b$;  $d = a$;

shift $m(d)$ right $(e(c) - e(d))$ places                // arithmetic right shift
$m(c) = m(c) + m(d)$;                                    //twos-complement addition

if ( sgn($m(a)$) = sgn($m(b)$) ) {
    if ( sgn($m(c)$) != sgn($m(a)$) ) {                  // signed overflow
        $e(sum) = e(a) + 1$;
        $m(sum) = m(c)/2$;
        // done
    }
}

// possible subtractive cancellation
$s$ = number of leading sign bits in $2 \cdot m(c)$
$e(sum) = e(a) - s$;
$m(sum) = m(c) << s$;
```

The pseudocode above does not include detection of zero operands, and also does not include handling of exceptional conditions (infinity, denormal operands, etc.). Further, the above implies truncation rounding instead of the more common round-to-nearest. Since the exponent and mantissa in our format are both very large, we assume that applications will never reach the point where these trappings of IEEE 754 floating point arithmetic [Kahan] are needed.

Floating point multiplication of oct-precision numbers a and b is simpler but also more computationally intensive. To avoid gratuitous loss of accuracy, assume that the mantissa of *product* below can contain an additional 32 bits to the right of the binary point.

```
//signed multiplication of mantissas
$m(product) = m(a) \cdot m(b)$;              // keep high 256 bits of unsigned product
if  $(m(a) < 0)$
    $m(product) = m(product) - m(b)$;
if  $(m(b) < 0)$
    $m(product) = m(product) - m(a)$;

$s$ = number of leading sign bits in $2 \cdot m(product)$
$e(product) = e(a) + e(b) - s$;
$m(product) = m(product) << s$;              // truncate result to 224 bits
```

The use of a signed mantissa slows down multiplication somewhat but makes addition much faster, since one can avoid having to compare the mantissas of the input operands.

Although not implemented, other primitives such as division and square root extraction can be built up from existing add and multiply primitives combined with a way to start Newton iteration [HennPat].

3. Implementation notes

Our implementation has concentrated only on the primitives that a full-featured oct-precision library would definitely need: addition, subtracton and multiplication of oct-precision numbers in our format. To increase computational speed we deal with arrays of oct-precision values; this allows stack operations and loading of constants to occur only once for a group of floating point operations, and also allows the next add or multiply to begin before the present add or multiply has finished. Asymptotically, this method yields a 30% performance boost that is independent of the G4 Altivec engine.

Within a single add or multiply, we reserve the Altivec engine for operations on the mantissa of oct-precision numbers and reserve the scalar integer unit for operations on the exponent; these can occur in parallel to save time. The Altivec unit is very suitable for the large variable shifts and large multiprecision multiplies that the algorithms require, and the integer unit provides functionality (such as the PowerPC **cntlzw** or count-leading-zeros instruction) which is very useful but awkward to implement using the Altivec vector instructions.

Since there are many possible execution paths for floating point addition, we attempt to optimize those paths that would happen most often at the expense of all others. We assume that floating point operands are positive and negative with approximately equal frequency, and that the magnitude of the exponents is the same or only slightly different. This implies that the most common path through the pseudocode given previously is the one where no cancellation occurs at all (i.e. $s = 0$), followed perhaps by the case of signed overflow during floating point addition. All other cases are removed from the main execution path and placed elsewhere to save time for these common cases. Further, we use multiple PowerPC condition registers to record multiple compare results simultaneously.

To add the 224-bit numbers a and b which reside in **v1:v2** and **v3:v4** respectively, we use the Altivec **vaddcuw** instruction to generate the carry bits out of four parallel 32-bit additions, and then keep adding the carry vector into the sum until the carry vector is zero:

```
    vaddcuw v5, v1, v3
    vaddcuw v6, v2, v4                    // v5:v6 = 256-bit carry vector
add_loop:
    vadduwm v1, v1, v3
    vadduwm v2, v2, v4                           // v1:v2 = new sum
    vsldoi v3, v5, v6, 4
    vsldoi v4, v6, v0, 4                  // v3:v4 = carry vector << 32
    vor v7, v3, v4                    // collapse carry vector to 128 bits
    vaddcuw v5, v1, v3
    vaddcuw v6, v2, v4                        // v5:v6 = new carry vector
    vcmpequw. v7, v0, v7
    bnl cr6, add_loop                       // loop if any carry bits left
```

With some rearranging, the loop above runs in 6 clock cycles on the G4 processor. While five iterations are needed in the worst case to completely propagate carries (carries propagate through five 32-bit words at most), two iterations are usually sufficient to make the carry vector zero and compute the 224-bit sum. Thus the addition takes 14 clock cycles on average; subtraction takes slightly longer since the bits in one operand must be complemented and the carry vector initialized to 1.

Unsigned multiplication of 224-bit integers consumes most of the time needed for floating point multiplication. We can save time by not computing the entire 448-bit product, since almost half of it is thrown away; instead, it is sufficient to compute enough of the "multiplication pyramid" for a 224-bit answer plus a few extra bits for use in normalizing the product. This makes multiplication almost twice as fast.

The algorithm used to calculate the product itself was first described in [Crandall], and takes advantage of the Altivec engine's ability to accumulate products of 16-bit integers very rapidly. Additional savings accrue from hardwiring the code to compute our half-product at the specific precision desired. The geometry of our multiplication pyramid changes as computation progresses, and tailoring the code to that geometry can save several vector permute operations (which happen to be a bottleneck for this algorithm).

Our oct-precision library presently consists of three routines: vector add, vector subtract and vector multiply. Each is implemented in assembly language.

4. Performance

The exact time needed for a single oct-precision floating point operation using our library depends on the input operands. Negative operands will take slightly longer, and rare conditions (such as massive subtractive cancellation after addition) can take up to twice as long to finish. In general, the routines perform fastest when operands are positive and when exponents are close together.

For comparison purposes we have listed the performance obtained by our primitives and also from C++ code that uses Bailey's library to effect the same computations. "Mocts" below refers to a unit of one million oct-precision floating point operations per second. These timings took place on a 500MHz Apple G4 containing an MPC7400 microprocessor and running MacOS X; Bailey's library was compiled using gcc with full optimization and using the multiply-add feature of the PowerPC architecture. Finally, our vectors of input values include both positive and negative numbers; our library would be 5-10% faster if the inputs were all positive.

size[**]	Bailey Mocts[*]			Altivec Mocts		
	ADD	SUB	MUL	ADD	SUB	MUL
1	1.9	1.8	0.9	7.5	8.0	3.0
5	2.2	2.1	1.0	8.6	8.4	4.0
10	2.2	2.1	1.0	9.2	8.6	4.3
20	2.2	2.2	1.0	9.4	8.8	4.4

`* 1 Mocts is one million` *oct-float* `operations per second.`
`** 'size' here is the numbert of oct floats stored contiguously.`

In conclusion, Altivec-enhanced emulation of oct-precision floating point arithmetic can be expected to perform about four times faster than an alternate implementation that uses the scalar floating-point unit of the G4.

Acknowledgments

We are indebted to the Advanced Computation Group, in particular J. Klivington, and to computationalist D. Bailey, for their aid in this research..

References

Hida Y, Li X and Bailey D 2000, “Quad-Double Arithmetic: Algorithms, Implementation and Application”, Technical Report LBNL-46996, Lawrence Berkeley National Laboratory.

Briggs K, doubledouble homepage: http://www-epidem.plantsci.cam.ac.uk/ kbriggs/doubledouble

Patterson D and Hennessy J 1996, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, San Francisco.

Crandall R and Klivington J 1999, “Altivec Implementation of Multiprecision Arithmetic”, Apple Computer, http://developer.apple.com/hardware/ve/acgresearch.html.

Kahan W 1996, ”Lecture Notes on the Status of IEEE Standard 754 for Binary Floating Point Arithmetic”, University of California Berkeley.

Motorola Inc 1997, “PowerPC Microprocessor Family: The Programming Environments for 32-bit Processors”.

Motorola Inc 1999, “Altivec Technology Programming Interface Manual”.

Motorola Inc 1998, “Altivec Technology Programming Environments Manual”.

Motorola Inc 2000, “MPC7400 RISC Microprocessor User’s Manual”.

Motorola Inc 2001, “MPC7450 RISC Microprocessor User’s Manual”.