Floating-Point Computation with Just Enough Accuracy

Hank Dietz, Bill Dieter, Randy Fisher, and Kungyen Chang

University of Kentucky, Department of Electrical & Computer Engineering
hankd@engr.uky.edu, dieter@engr.uky.edu,
randall.fisher@ieee.org, kchan0@engr.uky.edu

Abstract. Most mathematical formulae are defined in terms of operations on real numbers, but computers can only operate on numeric values with finite precision and range. Using floating-point values as real numbers does not clearly identify the precision with which each value must be represented. Too little precision yields inaccurate results; too much wastes computational resources.

The popularity of multimedia applications has made fast hardware support for low-precision floating-point arithmetic common in Digital Signal Processors (DSPs), SIMD Within A Register (SWAR) instruction set extensions for general purpose processors, and in Graphics Processing Units (GPUs). In this paper, we describe a simple approach by which the speed of these low-precision operations can be speculatively employed to meet user-specified accuracy constraints. Where the native precision(s) yield insufficient accuracy, a simple technique is used to efficiently synthesize enhanced precision using pairs of native values.

1 Introduction

In the early 1990s, the MasPar MP1 was one of the most cost-effective supercomputers available. It implemented floating-point arithmetic using four-bit slices, offering much higher performance for lower precisions. Thus, Dietz collected production Fortran programs from various researchers and analyzed them to see if lower precisions could be used without loss of accuracy. The discouraging unpublished result: using the maximum precision available, static analysis could not *guarantee* that even one digit of the results was correct! The insight behind the current paper is that most results were acceptably accurate despite using insufficient precision. Why not deliberately use fast low precision, repeating the computation at higher precision only when a dynamic test of result accuracy demands it?

Bit-slice floating-point arithmetic is no longer in common use, but the proliferation of multimedia applications requiring low-precision floating-point arithmetic has produced DSP (Digital Signal Processor), SWAR (SIMD Within A Register)[1], and GPU (Graphics Processing Unit) hardware supporting *only* 16-bit or 32-bit floating-point arithmetic. Scientific and engineering applications often require accuracy that *native* multimedia hardware precisions cannot guarantee, but by using relatively slow (synthesized) higher-precision operations *only* to recompute values that did not meet accuracy requirements, the low cost and high performance of multimedia hardware can be leveraged.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part I, LNCS 3991, pp. 226–233, 2006.

Section 2 overviews our method for synthesizing higher precision operations using pairs of native values and gives microbenchmark results for native-pair arithmetic optimized to run on DSPs, SWAR targets, and GPUs. Section 3 presents a very simple compiler/preprocessor framework that supports speculative use of lower precision, automatically invoking higher precision recomputations only when dynamic analysis of the result accuracy demands it. Conclusions are summarized in Section 4.

2 Multi-precision Arithmetic Using Error Residuals

There are many ways to synthesize higher-precision operations [2]. The most efficient method for the target multimedia hardware is what we call *native-pair* arithmetic, in which a pair of native-precision floating point values is used with the `lo` component encoding the residual error from the representation of the `hi` component. We did not invent this approach; it is well known as *double-double* when referring to using two 64-bit doubles to approximate quad precision [3, 4]. Our contributions center on tuning the analysis, algorithms, data layouts, and instruction-level coding for the multimedia hardware platforms and performing detailed microbenchmarks to bound performance.

More than two values may be used to increase precision, however, successive values reduce the exponent range by at least the number of bits in the mantissa extensions. Ignoring this effect was rarely a problem given the number of exponent bits in an IEEE 754[5] compliant 64-bit binary floating-point `double`, but a 32-bit `float` has a 24-bit mantissa and only an 8-bit exponent. A `float` pair will have twice the native mantissa precision only if the exponent of the low value is in range, which implies the high value exponent must be at least 24 greater than the native bottom of the exponent range; thus, we have lost approximately 10% of the dynamic range. Similarly, treating four `float` as an extended-precision value reduces the effective dynamic range by at least 3×24, or 72 exponent steps – which is a potentially severe problem. Put another way, precision is limited by the exponent range to less than 11 `float` values.

One would expect, and earlier work generally assumes, that the exponents of the `lo` and `hi` components of a native-pair will differ by precisely the number of bits in the mantissa. However, values near the bottom of the dynamic range have a loss of precision when the `lo` exponent falls below the minimum representable value. A component value of 0 does not have an exponent per se, and is thus a special case. For non-zero component values, normalization actually ensures only that the exponent of `lo` is *at least* the number of component mantissa bits less than that of `hi`. Using `float` components, if the 25th bit of the higher-precision mantissa happens to be a 0, the exponent of `lo` will be at least 25 less – not 24 less. In general, a run of k 0 bits logically at the top of the lower-half of the higher-precision mantissa are absorbed by reducing the `lo` exponent by k. For this reason, some values requiring up to k bits more than twice the `native` mantissa precision can be precisely represented! However, this also means that, if the `native` floating-point does not implement denormalized arithmetic (many implementations do not[6, 7]), a run of k 0 bits will cause `lo` to be out of range (i.e., represented as 0) if an exponent of k less than that of `hi` is not representable; in the worst case, if the `hi` exponent is 24 above the minimum value and k=1, the result has only 25 rather than 48 bit precision. Earlier work[8] is oblivious to these strange numerical properties; our runtime accuracy checks are a more appropriate response.

Space does not permit listing our optimized algorithms in this paper. Although we used C for some development and testing, most compilers cannot generate good code for the routines in C because they tend to "optimize" the floating-point operations in a way that does not respect precision constraints. Further, significantly higher performance may be obtained by careful use of instruction set features and architecture-specific data layouts. The following subsections summarize the microbenchmark performance of our machine-specific, hand-optimized, assembly-level code for each target architecture.

2.1 Performance Using Host Processor Instructions

If native-pair operations using attached multimedia processors are too slow to be competitive with higher-precision operations on the host processor, then these operations should be performed on the host or can be divided for parallel execution across the host and multimedia hardware. Table 1 lists the official clock-cycle latencies for host processor native (X87) floating point operations using an AMD ATHLON[9] and INTEL PENTIUM 4[10].

Table 1. Performance, in clock cycles, of host processor instructions

type	processor	add	sub	mul	sqr	div	sqrt
32-bit `float`	ATHLON	4	4	4	4	16	19
32-bit `float`	PENTIUM 4	5	5	7	7	23	23
64-bit `double`	ATHLON	4	4	4	4	20	27
64-bit `double`	PENTIUM 4	5	5	7	7	38	38
80-bit extended	ATHLON	4	4	4	4	24	35
80-bit extended	PENTIUM 4	5	5	7	7	43	43

Native-pair operations constructed using these types are approximately an order of magnitude slower, so it is fairly obvious that pairing 32-bit `float` values is not productive. However, pairing 64-bit `double` values or 80-bit extended values is useful (although loading and storing 80-bit values is relatively inefficient). Thus, a host processor can effectively support at least five precisions, roughly corresponding to mantissas of 24, 53, 64, 106, and 128 bits with a separate sign bit.

2.2 DSP Targets

There are many different types of DSP chips in common use, most of which do not have floating-point hardware. Of those that do, nearly all support only precisions less than 64 bits. Our example case is the Texas Instruments TMS320C31[6], which provides non-IEEE 754 floating-point arithmetic using an 8-bit exponent and 24-bit mantissa, both represented in 2's complement. This DSP has specialized multiply-add support, which accelerates the multiply, square, and divide algorithms, but neither add nor subtract. Table 2 gives the experimentally-determined clock cycle counts for each of the `native` and `nativepair` operations.

Table 2. Cycle counts and instructions required for DSP operations

(a) Opertaion cycle counts

type	add	sub	mul	sqr	div	sqrt
native	1	1	1	1	42	51
nativepair	11	11	25	19	112	119

(b) Instructions required

type	add	sub	mul	sqr	div	sqrt
native	1	1	1	1	33	39
nativepair	11	11	25	19	64	99

In general, an order of magnitude slowdown is incurred for `nativepair` operations, but divide and square root do better because they require executing many instructions for `native` operands. It is worth noting that the additional code size for `nativepair` operation sequences is modest enough to allow their use in embedded systems even if ROM space is tight; the number of instruction words for each operation is summarized in Table 2.

2.3 SWAR Targets and SWAR Data Layout

The most commonly used floating-point SWAR instruction sets are 3DNOW![11, 7], SSE[12] (versions 1, 2, and 3[13] and the AMD64 extensions[14]), and ALTIVEC[15]. These instruction sets differ in many ways; for example, 3DNOW! uses 64-bit registers while the others use 128-bit registers. However, there are a few common properties. The most significant commonality is that all of these SWAR instruction sets use the host processor memory access structures. Thus, the ideal data layout is markedly different from the obvious layout assumed in earlier multi-precision work.

Logically, the `hi` and `lo` parts of a `nativepair` may together be one object, but that layout yields substantial alignment-related overhead for SWAR implementations even if the `nativepair` values are aligned: different fields within the aligned objects have to be treated differently. The ideal layout separates the `hi` and `lo` fields to create contiguous, aligned, interleaved, vectors of the appropriate length. For example, 32-bit 3DNOW! works best when pairs of `nativepair` values have their components interleaved as a vector of the two `hi` fields and a vector of two `lo` fields; for SSE and ALTIVEC, the vectors should be of length four. The creation of separate, Fortran-style, arrays of `hi` and `lo` components is not as efficient; that layout makes write combining ineffective, requires rapid access to twice as many cache lines, and implies address accesses separated by offsets large enough to increase addressing overhead and double the TLB/page table activity.

Given the appropriate data layout, for 3DNOW! the primary complication is that the instruction set uses a two-register format that requires move instructions to avoid overwriting values. Table 3 the experimentally-determined cycle counts using the cycle count performance register in an AMD ATHLON XP.

All measurements were taken repeating the operation within a tight loop, which did allow some parallel overlap in execution (probably more than average for `swarnative` and less for `swarnativepair`). All counts given are for operations on two-element SWAR vectors of the specified types; for example, two `nativepair_add` operations are completed in 24 clock cycles. Although 3DNOW! offers twice the 32-bit floating-point performance of the X87 floating-point support within the same processor, the

Table 3. Cycle counts for 3DNow! and SSE operations

	type	add	sub	mul	sqr	div	sqrt
3DNow!	`swarnative`	1	1	1	1	9	9
3DNow!	`swarnativepair`	24	28	27	14	57	40
SSE	`float swarnativepair`	51	50	148	129	173	199
SSE	`double swarnativepair`	45	48	48	42	50	-

X87 `double` arithmetic is faster than 3DNOW! `nativepair` for all operations except reciprocal and square root.

The SSE code is very similar to that used for 3DNOW!, differing primarily in data layout: there are four 32-bit values or two 64-bit values in each `swarnative` value. Thus, the float version produces twice as many results per `swarnativepair` operation. The code sequences were executed on an INTEL PENTIUM 4 and the cycle counter performance register was used to obtain the cycle counts in Table 3, which show that `float swarnativepair` does not compete well with host `double`, but `double swarnativepair` is very effective.

2.4 GPU Targets

DSP parts tend to be slow, but can function in parallel with a host processor; SWAR is fast, but does not work in parallel with the host. The excitement about GPU targets comes from the fact that they offer both the ability to operate in parallel with the host and speed that is competitive with that of the host.

Although there are many different GPU hardware implementations, all GPUs share a common assembly-language interface for vertex programs[16, 17] and for fragment (pixel-shading) programs[16, 17]. All GPUs use SWAR pixel operations on vectors of 4 components per register (corresponding to the red, green, blue, and alpha channels), with relatively inefficient methods for addressing fields within registers. Thus, the optimal data layout and coding for native-pair operations is very similar to that used for SSE. Oddly, the precision of GPU arithmetic is not standardized, ranging from 16-bit to 32-bit. For our latest experiments, we purchased a $600 NVIDIA GEFORCE 6800 ULTRA, which was then the fastest commodity GPU with roughly IEEE-compliant 32-bit floating point support; Table 4 shows the performance results .

Table 4. Relative cost of GPU operations

type	add	sub	mul	sqr	div	sqrt
`swarnative`	1	1	*	*	4	20
`swarnativepair`	11	11	18	10	35	28

To obtain the above numbers, it was necessary to resort to fractional factorial experimental procedures that timed combinations of operations and used arithmetic methods to extract times for individual operations. Each experiment was repeated 220 times

to determine a 95% confidence interval for each time, which was then used to compute upper and lower bound times for the individual operations. Quirks of the NVIDIA GEFORCE 6800 ULTRA GPU and its assembler yielded inconsistent timing for some combinations of operations; there were insufficient consistent timings for the multiplication and squaring operations to determine the execution cost (probably about the same as add). All the other costs are listed in the above table relative to the cost of a `swarnative` add; in no case was the 95% confidence error greater than 1 unit. These results also are generally consistent with preliminary experiments we performed using an ATI RADEON 9800 XT with a less sophisticated timing methodology.

3 Compiler and Language Support

As mentioned earlier in this paper, traditional compiler technology is somewhat incompatible with the already awkward `nativepair` codings. Assembly-level coding is not viable for implementing complicated numerical algorithms. Implementation and benchmarking are beyond the scope of this paper, but we suggest that the compilation system should explicitly manage precision, including support for *speculative precision*.

Analysis and code generation for explicit precisions allows the compiler not only to maintain correctness while optimizing finite-precision computations, but also to select the fastest implementation for each operation individually – for example, using 3DNOW! `swarnativepair` for square root and X87 double for other operations. Precision directives have been used to preprocess Fortran code to make use of an arbitrary-precision arithmetic package[8]. Better, the notation used in our SWARC[18] dialect of C can be extended: `int:5` specifies an integer of at least five bits precision, so `float:5` could specify a floating-point value with at least five mantissa bits. A less elegant notation can be supported using C++ templates. Requirements on dynamic range, support of IEEE 754 features like NAN and INFINITY, etc., are more complex and less commonly an issue; they can be specified using a more general, if somewhat awkward, syntax.

Speculative precision is based on specifying accuracy constraints. Accuracy requirements can be specified directly or, more practically, as both an accuracy required and a functional test to determine the accuracy of a result. The compiler would generate multiple versions of the speculative-precision code, one for each potentially viable precision. A crude but effective implementation can be created for C++ using a simple preprocessor with straightforward directives like:

`#faildef` ***failure_code*** *Defines* ***failure_code*** *as the code to execute when all precision alternatives fail; this definition can be used for many speculative blocks, not just one*

`#specdef` ***name(item1, item2, ...)*** *Defines name as the ordered sequence of types* ***item1***, ***item2***, *etc.; this definition can be used for many speculative blocks, not just one*

`#speculate` ***name1 name2 ...*** *Defines the start of a region of code which is to be speculatively executed for* ***name1***, ***name2***, *etc. taking each of the values specified in sequence*

`#fail` *Defines the position at which the failure action should be applied*

`#commit` *Defines the position at which the speculate region ends*

In practice, the accuracy check would be a much cheaper computation than the speculative computation; for example, Linpack and many other math libraries compute error terms that could be examined, but we prefer a short example for this paper. Suppose that there are both `float` and `double` versions of `sqrt()`, overloaded using the usual C++ mechanisms, and our goal is to use the cheapest version that can pass a simple accuracy test `mytest()`. Our short example could be coded as:

```
#faildef exit(1);
#specdef fd(float, double)
#speculate fd
  fd a = x; double b = sqrt(a); if (!mytest(b, x)) {
#fail
  } y = b;
#commit
```

Which would be preprocessed to create C++ code like:

```
#define faildef { exit(1); }
#define fd float
{ fd a = x; double b = sqrt(a);
if (!mytest(b, x)) {goto fail0_0;} y = b; } goto commit0;
#define fd double
fail0_0: ; { fd a = x; double b = sqrt(a);
if (!mytest(b, x)) { faildef } y = b; } commit0: ;
```

The syntax could be prettier, but this speculation mechanism has very little overhead and is general enough to handle many alternative-based speculations, not just speculation on types. Further, although the sample generated code simply tries the alternatives in the order specified (which would typically be lowest precision first), one could use a history mechanism resembling a branch predictor to intelligently alter the type sequence based on past behavior.

Another possible improvement is to optimize the higher-precision computations to incrementally improve the precision of the already-computed results, but this is much more difficult to automate. For example, using the `native` results as the initial top-halves of the `nativepair` computations may be cheaper than computing `nativepair` results from scratch, but the computation is changed in ways far too complex to be managed by a simple preprocessor.

4 Conclusions

Although floating-point arithmetic has been widely used for decades, the fact that it is a poor substitute for real numbers has continued to haunt programmers. Unexpected accuracy problems manifest themselves far too frequently to ignore, so specifying excessive precision has become the norm. Even the highest precision supported by the hardware sometimes proves insufficient. This paper suggests a better way.

Rather than statically fixing guessed precision requirements in code, we suggest a more dynamic approach: code to try the lowest potentially viable precision and try

successively higher precisions only when the accuracy is (dynamically at runtime) determined to be inadequate. Thanks to demand in the multimedia community, lower-precision floating-point arithmetic is now often implemented with very high performance and very low cost. The techniques we have developed for extending precision using optimized native-pair arithmetic are commonly an order of magnitude slower than native. However, the large speed difference actually makes speculation more effective. Even if speculating lower precision usually fails to deliver the desired accuracy, an occasional success will reap a significant speedup overall. Only experience with a range of applications will determine just how often speculation succeeds.

References

1. Dietz, H.G., Fisher, R.J.: Compiling for SIMD within a register. In Chatterjee, S., Prins, J.F., Carter, L., Ferrante, J., Li, Z., Sehr, D., Yew, P.C., eds.: Languages and Compilers for Parallel Computing. Springer-Verlag (1999) 290–304
2. Bailey, D.H., Hida, Y., Jeyabalan, K., Li, X.S., Thompson, B.: Multiprecision software directory. http://crd.lbl.gov/ďhbailey/mpdist/ (2006)
3. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. **18** (1971) 224–242
4. Linnainmaa, S.: Software for doubled-precision floating-point computations. ACM Trans. Math. Softw. **7**(3) (1981) 272–283
5. IEEE: IEEE Standard for Binary Floating Point Arithmetic Std. 754-1985. (1985)
6. Texas Instruments: TMS320C3x User's Guide. (2004)
7. Advanced Micro Devices: 3DNow! Technology Manual. (2000)
8. Bailey, D.H.: Algorithm 719; Multiprecision translation and execution of FORTRAN programs. ACM Trans. Math. Softw. **19**(3) (1993) 288–319
9. Advanced Micro Devices: AMD Athlon Processor x86 Code Optimization Guide. (2002)
10. Intel: Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual. (2002)
11. Advanced Micro Devices: AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions. (2003)
12. Klimovitski, A.: Using SSE and SSE2: Misconceptions and reality. Intel Developer UPDATE Magazine (2001)
13. Smith, K.B., Bik, A.J.C., Tian, X.: Support for the Intel Pentium 4 processor with hyper-threading technology in Intel 8.0 compilers. Intel Technology Journal **08**(ISSN 1535-864X) (2004)
14. Advanced Micro Devices: AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions. (2003)
15. Freescale Semiconductor: AltiVec Technology Programming Interface Manual. (1999)
16. Microsoft: DirectX Graphics Reference. (2006)
17. Silicon Graphics, Inc: OpenGL Extension Registry. (2003)
18. Fisher, R.J., Dietz, H.G.: The Scc Compiler: SWARing at MMX and 3DNow. In Carter, L., Ferrante, J., eds.: Languages and Compilers for Parallel Computing. Springer-Verlag (2000) 399–414