Extended-Precision Floating-Point Numbers for GPU Computation

Andrew Thall, *Alma College*

***Abstract*— Double-float (df_{64}) and quad-float (qf_{128}) numeric types can be implemented on current GPU hardware and used efficiently and effectively for extended-precision computational arithmetic. Using unevaluated sums of paired or quadrupled f_{32} single-precision values, these numeric types provide approximately 48 and 96 bits of mantissa respectively at single-precision exponent ranges for computer graphics, numerical, and general-purpose GPU programming. This paper surveys current art, presents algorithms and Cg implementation for arithmetic, exponential and trigonometric functions, and presents data on numerical accuracy on several different GPUs. It concludes with an in-depth discussion of the application of extended precision primitives to performing fast Fourier transforms on the GPU for real and complex data.**

[Addendum (July 2009): the presence of IEEE compliant double-precision hardware in modern GPUs from NVidia and other manufacturers has reduced the need for these techniques. The double-precision capabilities can be accessed using CUDA or other GPGPU software, but are not (as of this writing) exposed in the graphics pipeline for use in Cg-based shader code. Shader writers or those still using a graphics API for their numerical computing may still find the methods described herein to be of interest.]

***Index Terms*— floating-point computation, extended-precision, graphics processing units, GPGPU, Fast Fourier Transforms, parallel computation**

Manuscript date of submission: March 15, 2007

I. Introduction

MODERN GPUs have wide data-buses allowing extremely high throughput, effecting a stream-computing model and allowing SIMD/MIMD computation at the fragment (pixel) level. Machine precision on current hardware is limited, however, to 32-bit nearly IEEE 754 compliant floating-point. This limited precision of fragment-program computation presents a drawback for many general-purpose (*GPGPU*) applications. Extended-precision techniques, developed previously for CPU computation, adapt well to GPU hardware; as programmable graphics pipelines its IEEE compliance, extending precision becomes increasingly straightforward; as future generations of GPUs move to hardware support for higher precision, these techniques will remain useful, leveraging parallelism and sophisticated instruction sets of the hardware (e.g., vector-parallelism, fused multiply-adds, etc.) to provide greater efficiencies for extended-precision than has been seen in most CPU implementations.

Higher precision computations are increasingly necessary for numerical and scientific computing (see Bailey [1], Dinechin et al [2]). Techniques for extended and mixed precision computation have been available for general purpose programming through myriad software packages and systems, but there have been only limited attempts to apply these methods for *general-purpose graphics-processing-unit* (*GPGPU*) computation, which have been hampered by having at best f_{32}single-precision floating point capability. Some success has been achieved in using augmenting GPU based computation with double-precision CPU correction terms: Göddeke et al [3] mix GPU computation with CPU-based double-precision defect correction in Finite-Element simulation, achieving a $2\times$ speedup over tuned CPU-only code while maintaining the same accuracy.

Myer and Sutherland [4] coined the term *wheel of reincarnation* to describe the evolution of display processor technology as a never-ending series of hardware innovations are created as add-on special purpose systems. Such esoteric hardware is always in a race-against-time against generic processors; advanced capabilities developed for special-purpose systems are invariably folded back into commodity CPU technology as price-performance breakpoints allow. We are currently in a unique time vis-a-vis the CPU/GPU dichotomy, where the stream programming model inherent in GPU hardware has allowed the computational power of GPUs to rocket past that of the relatively static performance of the single-processor CPU over the near past and as projected for the near future. GPU-based stream processors avoids a major pitfall of prior parallel processing systems in being nearly ubiquitous; because of the relative cheapness of GPUs and their use in popular, mass-marketed games and game-platforms, inexpensive systems are present and in on most commodity computers currently sold. Because of this omnipresence, these processors show programmability—quality and choices of APIs, multiple hardware platforms running the same APIs, stability of drivers—far beyond that of the special-purpose hardware of previous generations.

These trends are expected to continue: when innovative breakthroughs are made in CPU technology, these can be expected to be applied by the GPU manufacturers as well. CPU trends to multi-core systems will provide hardware parallel of the classic MIMD variety; these can be expected to have the same weaknesses of traditional parallel computers: synchronization, deadlock, platform-specificity and instability of applications and supporting drivers under changes in hardware, firmware, and OS capabilities. While systems such as OpenMP and OpenPVM provide platform and OS independence, it remains the case that unless it is worth the developer's time, advanced capabilities and algorithms will remain experimental, brittle, platform-specific curiosities. The advantage of the stream-based computational model is its simplicity.

This paper will survey prior and current art in extended-precision computation and in application of this to GPUs. It will then describe an implementation of a df_{64} and qf_{128} library for current generation GPUs, show data on numerical accuracy for basic operations and elementary functions, and discuss limitations of these techniques for different hardware platforms.

II. Background: Arbitrary Precision Arithmetic

Techniques for performing extended-precision arithmetic in software using pairs of machine-precision numbers have a long history: Dekker [5] is most often cited on the use of unevaluated sums in extended-precision research prior to the IEEE 754 standard, with Linnainmaa [6] generalizing the work of Dekker to computer independent implementations dependent on faithful rounding. Such methods are distinct from alternative approaches exemplified by Brent [7], Smith [8], and Bailey [9] that assign special storage to exponent and sign values and store mantissa bits in arbitrary-length integer arrays. A general survey of arbitrary precision methods is Zimmermann [10]

Priest [11] in 1992 did a full study of extended-precision requirements for different radix and rounding constraints. Shewchuk [12] drew basic algorithms from Priest but restricted his techniques and analyses to the IEEE 754 floating-point standard [13], radix-2, and exact rounding; these allowed relaxation of normalization requirements and led to speed improvements. These two provided the theoretical underpinnings for the *single-double* FORTRAN library of Bailey [14], the *doubledouble* C++ library of Briggs [15] and the C++ *quad-doubles* of Hida et al [16], [17]. The use and hazards of double- and quad-precision numerical types is discussed by Li et al [18], in the context of extended- and mixed-precision BLAS libraries.

For methods based on Shewchuk's and Priest's techniques, faithful rounding is crucial for correctness. Requirements in particular of IEEE-754 compliance create problematic difficulties for older graphics display hardware; latest generation GPUs are much more compliant with the IEEE standard.

A. Extended-Precision Computation

The presentation here will follow that of Shewchuk [12]. Given IEEE 754 single-precision values, with exact rounding and round-to-even on ties, for binary operations $* \in \{+, -, \times, /\}$, the symbols $\circledast \in \{\oplus, \ominus, \otimes, \oslash\}$ represent b-bit floating-point version with exact-rounding, i.e.,

$$a \circledast b \equiv fl(a * b) = a * b \ + \mathrm{err}(a \circledast b), \tag{1}$$

where err$(a \circledast b)$ is the difference between the correct arithmetic and the floating-point result; exact rounding guarantees that

$$|\mathrm{err}(a \circledast b)| \leq \frac{1}{2}\mathrm{ulp}(a \circledast b). \tag{2}$$

Extended-precision arithmetic using unevaluated sums of single-precision numbers rests on a number of algorithms (based on theorems by Dekker [5] and Knuth [19]) used for defining arithmetic operations precisely using pairs of non-overlapping results.

Theorem 1 ***Dekker [5]*** *Let a and b be p-bit floating-point numbers such that $|a| \geq |b|$. Then the following algorithm will produce a nonoverlapping expansion $x+y$ such that $a+b = x+y$, where x is an approximation to $a+b$ and y represents the roundoff error in the calculation of x.*

Two floating-point numbers x and y are *nonoverlapping* if the least-significant non-zero bit of x is more significant than the most-significant non-zero bit of y, or vice-versa. An expansion

Algorithm 1 Fast-Two-Sum

```
Require: $|a| \geq |b|$, $p$-bit floating point numbers
1: procedure Fast-Two-Sum($a$, $b$)
2:     $x \leftarrow a \oplus b$
3:     $b_{virtual} \leftarrow x \ominus a$
4:                 ▷ $b_{virtual}$ is the actual value added to $x$ in 2
5:     $y \leftarrow b \ominus b_{virtual}$
6:     return $(x, y)$
7:                 ▷ $x + y = a + b$, where $y$ is roundoff error on $x$
8: end procedure
```

is non-overlapping if all of its components are mutually non-overlapping. We will also define two numbers x and y as *adjacent* if they overlap, if x overlaps $2y$ or if y overlaps $2x$. An expansion is *nonadjacent* if no two of its components are adjacent.

Theorem 2 ***Knuth [19]*** *Let a and b be p-bit floating-point numbers where $p \geq 3$. Then the following algorithm will produce a nonoverlapping expansion $x+y$ such that $a+b = x+y$, where x is an approximation to $a+b$ and y represents the roundoff error in the calculation of x.*

Algorithm 2 Two-Sum

```
Require: $a, b$, $p$-bit floating point numbers, where $p \geq 3$
1: procedure Two-Sum($a$, $b$)
2:     $x \leftarrow a \oplus b$
3:     $b_{virtual} \leftarrow x \ominus a$
4:     $a_{virtual} \leftarrow x \ominus b_{virtual}$
5:     $b_{roundoff} \leftarrow b \ominus b_{virtual}$
6:     $a_{roundoff} \leftarrow a \ominus a_{virtual}$
7:     $y \leftarrow a_{roundoff} \oplus b_{roundoff}$
8:     return $(x, y)$
9:                 ▷ $x + y = a + b$, where $y$ is roundoff error on $x$
10: end procedure
```

The following corollary, forming the heart of multiple-term expansions as numerical representations:

Corollary 3 *Let x and y be the values returned by* Fast-Two-Sum *or* Two-Sum. *On a machine whose arithmetic uses round-to-even tiebreaking, x and y are nonadjacent.*

Given these building blocks, Shewchuk describes *expansion-sum* algorithms for adding arbitrary-length numbers each represented by sequences of nonadjacent terms to get another such nonadjacent sequence as the exact sum. For the purpose of double- and quad-precision numbvers, the ones of importance regard pairs and quads of nonadjacent terms that form the df_{64} and qf_{128} representations.

Multiplication of multiple-term expansions also rests on a few basic theorems, one by Dekker [5] and another that he attributes to G. W. Veltkamp.

Theorem 4 ***Dekker [5]*** *Let a a p-bit floating-point number where $p \geq 3$. Choose a* splitting point *s such that $\frac{p}{2} \leq s \leq p-1$. Then the following algorithm will produce a $(p-s)$-bit value a_{hi} and a nonoverlapping $(s-1)$-bit value a_{lo}, such that $|a_{hi}| \geq |a_{lo}|$ and $a = a_{hi} + a_{lo}$.*

Sherchuk emphasizes the strangeness of representing a p-bit number with two numbers that together have only (p - 1)-significant bits; this is possible because a_{lo} is not necessarily

TABLE I
GPU Floating-Point Paranoia Results on NVidia Architecture. Bounds are in *ulps* for $s23e8$ representation.

Operation	Exact rounding (IEEE-754)	chopped	NV35 fp30	NV40 & G70 fp40	G80 gp4fp
Addition	[-0.5, 0.5]	(-1.0, 0.0]	[-1.000, 0.000]	[-1.000, 0.000]	[-0.5, 0.5]
Subtraction	[-0.5, 0.5]	(-1.0, 0.0)	[-1.000, 1.000]	[-0.75, 0.75]	[-0.5, 0.5]
Multiplication	[-0.5, 0.5]	(-1.0, 0.0]	[-0.989, 0.125]	[-0.78125, 0.625]	[-0.5, 0.5]
Division	[-0.5, 0.5]	(-1.0, 0.0]	[-2.869, 0.094]	[-1.19902, 1.37442]	[-1.58222, 1.80325]

Algorithm 3 Split

```
Require: $a$, a $p$-bit floating point number, where $p \geq 3$
Require: $s$, $\frac{p}{2} \leq s \leq p-1$
 1: procedure SPLIT($a$, $s$)
 2:     $c \leftarrow (2^s + 1) \otimes a$
 3:     $a_{big} \leftarrow c \ominus a$
 4:     $a_{hi} \leftarrow c \ominus a_{big}$
 5:     $a_{lo} \leftarrow a \ominus a_{hi}$
 6:     return $(a_{hi}, a_{lo})$
 7: end procedure
```

of the same sign as a_{hi}—the result of Alg. 3 is that the sign-bit of a_{lo} stores the extra bit needed for a p-bit significand. (This is especially important for splitting IEEE double-precision numbers, for which $p = 53$, and which, though odd, can be split into two $\lfloor \frac{p}{2} \rfloor$-bit values without loss of precision.)

Using this splitting algorithm, a precise multiplication algorithm can now be created.

Theorem 5 ***by Veltkamp [5]*** *Let a and b be p-bit floating-point numbers where $p \geq 6$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $ab = x + y$, where x is an approximation to ab and y represents the roundoff error in the calculation of x. Furthermore, if round-to-even tiebreaking is used, x and y are nonadjacent.*

Algorithm 4 Two-Product

```
Require: $a, b$, $p$-bit floating point numbers, where $p \geq 6$
 1: procedure TWO-PRODUCT($a$, $b$)
 2:     $x \leftarrow a \otimes b$
 3:     $(a_{hi}, a_{lo}) = \text{SPLIT}(a, \lceil \frac{p}{2} \rceil)$
 4:     $(b_{hi}, b_{lo}) = \text{SPLIT}(b, \lceil \frac{p}{2} \rceil)$
 5:     $err_1 \leftarrow x \ominus (a_{hi} \otimes b_{hi})$
 6:     $err_2 \leftarrow err_1 \ominus (a_{lo} \otimes b_{hi})$
 7:     $err_3 \leftarrow err_2 \ominus (a_{hi} \otimes b_{lo})$
 8:     $y \leftarrow (a_{lo} \otimes b_{lo} \ominus err_3$
 9:     return $(x, y)$
10: end procedure
```

B. Extended Precision on GPUs

The hardware support for vector operations in fragment programs makes them well-suited for double- and quad-precision computation. Preliminary exploration of the feasibility and error-characteristics of GPU double-floats has been done using the Cg language by Meredith et al [20] and Thall [21], and, using the Brook language, by Da Graça and Defour [22]. The Gödekke article [3] present a survey that includes these and related experiments in the GPGPU community.

C. Numerical Capabilites and Limitations of GPUs: GPU Paranoia

Marginal IEEE 754 compliance until recently, when became pretty good on G80/NV8800. Rounding-modes and characteristics of computation not fully compliant.

Testing accuracy and precision of hardware systems is problematic; for a survey see Cuyt et al [23]. The *Paranoia* system [24], developed by Kahan in the early 1980s, was the inspiration for *GPU Paranoia* by Hillesland and Lastra [25], a software system for characterizing the floating point behavior of GPU hardware. GPU Paranoia provides a test-suite estimating floating point arithmetic error in the basic function, as well as characterizing apparent number of guard digits and correctness of rounding modes. Recent GPU hardware has shown considerable improvement over earlier. For the purposes of extended-precision computation, 32-bit floats are required. For Nvidia GPUs, the 6800 series and above were the first to have sufficient precision and IEEE compliance for extended-precision numerical work; the 7800/7900/7950 series has better IEEE compliance, allowing all but the transcendental functions to be computed; for computations involving Taylor series, rather than self-correcting Newton-Raphson iterations, the superior round-off behavior and guard-digits of the 8800 series are necessary.

Test run with Hillesland's GPU Paranoia generate results shown in Table I.

D. FMAD *or Just Angry?*

Extended precision arithmetic can be made much more efficient on architectures that implement a *fused-multiply-add* (*FMAD* or *FMA*) in hardware. An FMAD instruction allowing the computation of $a \times b + c$ to machine precision with only a single round-off error. This allows a much simpler $twoProd()$ function to be used in place of Alg. 4:

Theorem 6 ***Hida [17], Pg. 6*** *Let a and b be p-bit floating-point numbers where $p \geq 6$. Then, on a machine implementing an FMAD instruction, the following algorithm will produce a nonoverlapping expansion $x + y$ such that $ab = x + y$, where x is an approximation to ab and y represents the roundoff error in the calculation of x. Furthermore, if round-to-even tiebreaking is used, x and y are nonadjacent.*

Algorithm 5 Two-Product-FMA

```
Require: $a, b$, $p$-bit floating point numbers, where $p \geq 6$
 1: procedure TWO-PRODUCT-FMA($a$, $b$)
 2:     $x \leftarrow a \otimes b$
 3:     $y \leftarrow \text{FMA}(a \times b - x)$
 4:     return $(x, y)$
 5: end procedure
```

The attractiveness of this algorithm is obvious; it depends, however, both on the correct error bounding of the hardware operation and on the ability of compilers to use the FMAD if and only if the programmer requires it. Even on hardware that correctly implements the FMAD instruction, an overly helpful compiler is all too likely to return a value $y = 0$. This is especially problematic for a language such as Cg that is designed to optimize heavily even at the expense of accuracy, and where compilation is only to an intermediate assembly language that hides critical architectural details. The current df_{64} and qf_{128} implementations therefore use the longer Alg. 4.

III. Implementing GPU df_{64} and qf_{128} Floating Point

This discussion begin with df_{64} methods and then take up the challenges of qf_{128} implementation on current GPUs. While the techniques for this generalize from the CPU methods of Shewchuck [12] and Hida et al [16], the difficulty of implementing these in the limiting environment of GPU programming domain is not to be dismissed.

A. GPU Implementation of df_{64} Numbers

A df_{64} number A is an unevaluated sum of two f_{32} numbers represented the value of the floating point result and its error term.

$$A_{df64} = [a_{hi}, a_{lo}] \tag{3}$$
$$A = a_{hi} + a_{lo} \tag{4}$$

Arithmetic operations on these numbers are done exactly using a_{lo} as an error term for a_{hi} term. This allows 48-bits of precision available in the pair of values (23 bits +1 hidden per mantissa). (It is misleading, however, to call this a 48-bit floating point number, since numbers such as $2^{60} + 2^{-60}$ are exactly representable.)

Discussion of GPU-based arithmetic using these primitives will be divided into basic operations:

1) convertion between d_{64} and df_{64} representations,
2) *twoSum* and *df64_add* operations,
3) *splitting*, *twoProd*, and *df64_mult* operations,
4) division and square-root functions,
5) comparison operations,
6) exponential, logarithmic, and trigonometric functions, and,
7) modular-division and remainder operations.

The algorithms described and implemented here have been tested on Nvidia graphics hardware: Geforce 6800, 7900, and 8800 consumer grade cards, and depend as noted on the single-precision floating point characteristics of these and related platforms. Unless otherwise noted, routines will run on the least powerful of these platforms and under Cg 1.5 using fp40.

1) Conversion between Representations: Single-precision f_{32} converts trivially to and from df_{64} precision by respectively setting the low-order bits of the df_{64} number to zero, or assigning the high-order bits to the f_{32} number.

To convert between d_{64} and df_{64} numbers on Intel-based CPUs, the extended-precision internal format (doubles with 64-bit mantissa, rather than 53 bits) must be disabled d_{64} precision in order to get correct splitting of a d_{64} value into the nearest f_{32} value and corrector that form the df_{64} pair (see Hida et al. [16]).

```
const double SPLITTER = (1 << 29) + 1;
void df64::split(double a, float *a_hi, float *a_lo) {

    double t = a*SPLITTER;
    double t_hi = t - (t - a);
    double t_lo = a - t_hi;

    *a_hi = (float) t_hi;
    *a_lo = (float) t_lo;
}
```

Conversion from df_{64} to d_{64} requires only the summation of the high and low order terms.

2) TwoSum *and* df64_add *operations:* Addition in df_{64} depends on helper functions to sum f_{32} components and return df_{64} results. The `twoSum()` function computes the sum and error terms for general cases; the `quickTwoSum()` function assumes $a > b$.

```
float2 quickTwoSum(float a, float b) {
    float s = a + b;
    float e = b - (s - a);
    return float2(s, e);
}

float2 twoSum(float a, float b) {
    float s = a + b;
    float v = s - a;
    float e = (a - (s - v)) + (b - v);
    return float2(s, e);
}

float2 df64_add(float2 a, float2 b) {

    float2 s, t;
    s = twoSum(a.x, b.x);
    t = twoSum(a.y, b.y);
    s.y += t.x;
    s = quickTwoSum(s.x, s.y);
    s.y += t.y;
    s = quickTwoSum(s.x, s.y);
    return s;
}
```

This addition satisfies IEEE style error bound

$$a \oplus b = (1 + \delta)(a + b) \tag{5}$$

due to K. Briggs and W. Kahan.

On vector-based GPU hardware (in this study, pre-NV8800 GPUs), there is there is a $2\times$ speedup in performing operations simultaneously on two-vector and four-vector components. The above `df64_add()` can in this way be done with only a single call to the `twoSum()` which can be used as well to add real and imaginary components of single-precision complex numbers (f_{32} real and imaginary pair) simultaneously.

```
float4 twoSumComp(float2 a_ri, float2 b_ri) {

    float2 s = a_ri + b_ri;
    float2 v = s - a_ri;
    float2 e = (a_ri - (s - v)) + (b_ri - v);
    return float4(s.x, e.x, s.y, e.y);
}
float2 df64_add(float2 a, float2 b) {

    float4 st;
    st = twoSumComp(a, b);
    st.y += st.z;
    st.xy = quickTwoSum(st.x, st.y);
    st.y += st.w;
    st.xy = quickTwoSum(st.x, st.y);
    return st.xy;
}
```

3) splitting, twoProd, and df64_mult operations: At the heart of the extended-precision multiply is the splitting of f_{32} values into high and low order terms whose products will fit into a f_{32} values without overflow. The `split()` function takes an f_{32} float as input and returns the high and low order terms as a pair of floats. As for `twoSum()` above, vector-operation-based GPUs can achieve a 2× speedup by performing two splits simultaneously.

```
float2 split(float a) {

    const float split = 4097; // (1 << 12) + 1;
    float t = a*split;
    float a_hi = t - (t - a);
    float a_lo = a - a_hi;
    return float2(a_hi, a_lo);
}

float4 splitComp(float2 c) {

    const float split = 4097; // (1 << 12) + 1;
    float2 t = c*split;
    float2 c_hi = t - (t - c);
    float2 c_lo = c - c_hi;
    return float4(c_hi.x, c_lo.x, c_hi.y, c_lo.y);
}
```

The `twoProd()` function multiplies two f_{32} values to produce a df_{64} result, using f_{32} multiplication to produce the high-order result p and computing the low-order term as the error using split-components of the input a and b. Thus,

$$\begin{aligned} a &= a_{hi} + a_{lo} && (6)\\ b &= b_{hi} + b_{lo} && (7)\\ \Longrightarrow & && (8)\\ ab &= (a_{hi} + a_{lo}) \cdot (b_{hi} + b_{lo}) && (9)\\ &= a_{hi}b_{hi} + a_{hi}b_{lo} + a_{lo}b_{hi} + a_{lo}b_{lo} && (10)\\ & && (11) \end{aligned}$$

and the error term of the single-precision $p = a \otimes b$ can be computed by subtracting p from the split-product, beginning with the most-significant term.

```
float2 twoProd(float a, float b) {
    float p = a*b;
    float2 aS = split(a);
    float2 bS = split(b);
    float err = ((aS.x*bS.x - p)
                + aS.x*bS.y + aS.y*bS.x)
                + aS.y*bS.y;
    return float2(p, err);
}
```

As before, the pair of splits can be coalesced into a single double-wide split on vector-based GPUs.

Tests with the Cg 1.5 compiler on G70 hardware show that the `FMA-twoprod()`

```
float2 FMA-twoProd(float a, float b) {
    float x = a*b;
    float y = a*b - x;
    return float2(x, y);
}
```

does not correctly yield the df_{64} product and error term, due as expected to the aggressive optimization of the Cg compiler.

Thus armed, the `df64_mult()` can now be computed:

```
float2 df64_mult(float2 a, float2 b) {
    float2 p;

    p = twoProd(a.x, b.x);
    p.y += a.x * b.y;
    p.y += a.y * b.x;
    p = quickTwoSum(p.x, p.y);
    return p;
}
```

Special cases: squaring of df_{64} values can be done more efficiently with special `twoSqr()` and `df64_sqr()` methods; multiplication—and division—by powers-of-two is trivial, since the high and low-order terms can be multiplied or divided independently.

4) Division and Square-Root Functions: Division and square-root functions can be implemented using Newton-Raphson iteration. Since these methods are quadratically convergent in the neighborhood of their roots, the single-precision quotient or square-root serves as an excellent initial estimator; since these methods are self-correcting, the precision of the initial estimate is also not critical, and only one or two iterations thereafter are needed to produce df_{64} or qf_{128} precision.

Further and quite substantial speedups can be achieved using *Karp's method* (Karp and Markstein [26], used by Briggs [15], Hida [16]), which reduces the number of extended-precision operations needed in the Newton iteration. The algorithm for Karp's division algorithm is given in 6. This allows a double precision

Algorithm 6 Karp's Method for High-Precision Division

```
Require: $A \neq 0$, $B$, double-precision floating point values
1: procedure KARP'S DIVISION($B$, $A$)
2:     $x_n \leftarrow 1/A_{hi}$                      ▷ single ← single/single
3:     $y_n \leftarrow B_{hi} * x_n$                  ▷ single ← single*single
4:     Compute $Ay_n$                                 ▷ double ← double*single
5:     Compute $(B - Ay_n)$                           ▷ single ← double*double
6:     Compute $x_n(B - Ay_n)$                        ▷ double ← single*single
7:     $q \leftarrow y_n + x_n(B - Ay_n)$             ▷ double ← single+double
8:     return $q$                                     ▷ the double-precision $B/A$
9: end procedure
```

division with only a low-precision division, a single double-double multiply, and four single- or mixed-precision operations. A Cg implementation of this is straightforward. Karp gives a similar mixed-precision algorithm for square-roots (see 7). The

Algorithm 7 Karp's Method for High-Precision Square-Root

```
Require: $A > 0$, double-precision floating point value
1: procedure KARP'S SQUARE-ROOT($A$)
2:     $x_n \leftarrow 1/\sqrt{A_{hi}}$               ▷ single ← single/single
3:     $y_n \leftarrow A_{hi}x_n$                     ▷ single ← single*single
4:     Compute $y_n^2$                                ▷ double ← single*single
5:     Compute $(A - y_n^2)_{hi}$                     ▷ single ← double−double
6:     Compute $x_n(A - y_n^2)_{hi}/2$                ▷ double ← single*single/2
7:     $q \leftarrow y_n + x_n(A - y_n^2)/2$          ▷ double ← single+double
8:     return $q$                                     ▷ the double-precision $+\sqrt{A}$
9: end procedure
```

Cg implementation for this shows similarly substantial speedups over naive implementations. (The reciprocal square-root in Step 2 is a single operation on common graphics hardware and in

```
float2 df64_div(float2 B, float2 A) {

    float xn = 1.0f/A.x;
    float yn = B.x*xn;
    float diff = (df64_diff(B, df64_mult(A, yn))).x;
    float2 prod = twoProd(xn, diffTerm);

    return df64_add(yn, prodTerm);
}

float2 df64_sqrt(float2 A) {

    float xn = rsqrt(A.x);
    float yn = A.x*xn;
    float2 ynsqr = df64_sqr(yn);

    float diff = (df64_diff(A, ynsqr)).x;
    float2 prod = twoProd(xn, diff)/2;

    return df64_add(yn, prodTerm);
}
```

Fig. 1. Cg implementations of Karp's mixed-precision division and square-root algorithms

```
bool df64_eq(float2 a, float2 b) {
  return all(a == b);
}

bool df64_neq(float2 a, float2 b) {
  return any(a != b);
}

bool df64_lt(float2 a, float2 b) {
  return (a.x < b.x || (a.x == b.x && a.y < b.y));
}
```

Fig. 2. Cg implementations of basic df_{64} conditional tests. Note that comparisons between vector-based data are done component-wise.

the Cg language.) Figure 1 shows Cg implementations of both algorithms.

5) Comparison Operations: The common Boolean comparisons $\{=, \neq, <, \leq, >, \geq\}$ can be implemented in double-float precision by "short-circuit" conditional tests, reducing the likely number of single-float conditional tests to be no greater than a single test by comparing first the high-order terms of the two numbers. Current Cg profiles do not allow short-circuit comparisons, however, since conditional branching is much more costly than conditional testing, which can be performed in parallel on vector GPUs. The special case of comparison to zero always involves only a single comparison. Examples of Boolean comparisons are given in Fig. 2.

6) Exponential, Logarithmic, and Trigonometric Functions: Evaluation of extended-precision transcendental functions on the GPU shares the range of challenges common to all numerical techniques for software evaluation of these functions. Typically, range-reduction is used to bring the value to be evaluated into an acceptable range, then an algorithmic method based on Newton iteration, Taylor series, or table-based polynomial or Padé (rational) approximations are used to compute the answer to the necessary precision, followed by modification based on the range-reduction to produce the correct value. See [27] for a discussion of many different methods. A good discussion of Padé approximation is found in [28]. Correct rounding of transcendentals is often problematic even in conventional CPU libraries; for a discussion of issues and methods for full-precision results, see Gal [29] and Brisbarre et al [30].

```
float2 df64_expTAYLOR(float2 a) {

    const float thresh = 1.0e-20*exp(a.x);
    float2 t;  /* Term being added. */
    float2 p;  /* Current power of a. */
    float2 f;  /* Denominator. */
    float2 s;  /* Current partial sum. */
    float2 x;  /* = -sqr(a) */
    float m;

    s = df64_add(1.0f, a);  // first two terms
    p = df64_sqr(a);
   m = 2.0f;
    f = float2(2.0f, 0.0f);
    t = p/2.0f;
    while (abs(t.x) > thresh) {
        s = df64_add(s, t);
        p = df64_mult(p, a);
       m += 1.0f;
        f = df64_mult(f, m);
        t = df64_div(p, f);
    }

    return df64_add(s, t);
}
```

Fig. 3. Cg implementations of $\exp(x)$ using a Taylor series, for $|x| < \frac{\ln 2}{128}$. Range-reduction is handled in an enclosing routine.

```
float2 df64_log(float2 a) {

    float2 xi = float2(0.0f, 0.0f);

    if (!df64_eq(a, 1.0f)) {
        if (a.x <= 0.0)
            xi = log(a.x).xx;    // return NaN
        else {
            xi.x = log(a.x);
            xi = df64_add(df64_add(xi,
                   df64_mult(df64_exp(-xi), a)), -1.0);
        }
    }
    return xi;
}
```

Fig. 4. Cg implementations of $\ln(x)$ using Newton-Raphson iteration. The initial approximation is provided by a single-precision result.

Particular challenges to the GPU occur when inaccuracies in rounding of basic operations make range-reduction and evaluation of series solutions problematic. Inaccurate range-reduction followed by nearly correct approximation followed by inaccurate restoration to final value can quickly chew up a substantial fraction of the extended precision one hoped for. In general, methods based on Newton iteration fare better than those based on Taylor series or table-based approximation.

In terms of hardware capability, pre-8800 NVidia hardware can be used for exponential functions but computation of accurate logarithms, sines and cosines has been reliably achieved only on G80 (8800) cards, with their markedly better IEEE 754 compliance.

7) Modular Division and Remainder Operations: Major hassles. Cg 2.0 essential. 8800 essential.

```
float4 df64_sincosTAYLOR(float2 a) {

    const float thresh = 1.0e-20 * abs(a.x) * ONE;
    float2 t;  /* Term being added. */
    float2 p;  /* Current power of a. */
    float2 f;  /* Denominator. */
    float2 s;  /* Current partial sum. */
    float2 x;  /* = -sqr(a) */
    float m;

    float2 sin_a, cos_a;
    if (a.x == 0.0f) {
        sin_a = float2(ZERO, ZERO);
        cos_a = float2(ONE, ZERO);
    }
    else {
        x = -df64_sqr(a);
        s = a;
        p = a;
        m = ONE;
        f = float2(ONE, ZERO);
        while (true) {
            p = df64_mult(p, x);
            m += 2.0f;
            f = df64_mult(f, m*(m-1));
            t = df64_div(p, f);
            s = df64_add(s, t);
            if (abs(t.x) < thresh)
                break;
        }

        sin_a = s;
        cos_a = df64_sqrt(df64_add(ONE, -df64_sqr(s)));
    }
    return float4(sin_a, cos_a);
}
```

Fig. 5. Cg implementations of $\sin(x)$ and $\cos(x)$ using Taylor series expansion.

B. GPU Implementation of qf_{128} Numbers

The implementation and discussion here once again follows that of Hida [17] on efficient quad-double implementation. A qf_{128} number A is an unevaluated sum of four f_{32} numbers allowing 96-bits of precision in single-precision exponent ranges. In this representation,

$$\begin{aligned} A_{qf128} &= [a_1, a_2, a_3, a_4] \\ A &= a_1 + a_2 + a_3 + a_4 \end{aligned}$$

for single-precision a_n, where a_1, a_2, a_3, a_4 store successively lower-order bits in the number. Arithmetic operations on qf_{128} numbers depends on the same splitting, two-sum and two-prod operations as df_{64} computation; an additional renormalization step is also required to....[xxAT ensure a consistent representation?]. A five-input version of this is shown in Fig. 6.

Given this renormalization function, it is also necessary to have n-Sum functions taking different numbers of inputs and producing different numbers of outputs. With these in hand, a basic `qf_add()` function can be implemented as in Fig. 7. This method of addition does not satisfy IEEE-style error bound but rather

$$a \oplus b = (1+\delta_1)a + (1+\delta_2)b, \text{with } |\delta_1|, |\delta_2| \le \varepsilon_{qf128} \qquad (12)$$

For proofs of these bounds for quad-doubles, see Hida [17]. IEEE bounds on qf_{128} addition can be achieved using an algorithm by Shewchuk [12] and Boldo [31]; this requires costly sorting of terms and a float-accumulate operation (with a factor of 2–3.5x

```
float4 qf_renormalize(float a0, float a1, float a2,
                      float a3, float a4) {

    float s;
    float t0, t1, t2, t3, t4;
    float s0, s1, s2, s3;
    s0 = s1 = s2 = s3 = 0.0f;

    s = quickTwoSum(a3, a4, t4);
    s = quickTwoSum(a2, s, t3);
    s = quickTwoSum(a1, s, t2);
    t0 = quickTwoSum(a0, s, t1);

    float4 b = float4(0.0f, 0.0f, 0.0f, 0.0f);

    int count = 0;
    s0 = quickTwoSum(t0, t1, s1);
    if (s1 != 0.0) {
        s1 = quickTwoSum(s1, t2, s2);
        if (s2 != 0.0) {
            s2 = quickTwoSum(s2, t3, s3);
            if (s3 != 0.0)
                s3 += t4;
            else
                s2 += t4;
        }
        else {
            s1 = quickTwoSum(s1, t3, s2);
            if (s2 != 0.0)
                s2 = quickTwoSum(s2, t4, s3);
            else
                s1 = quickTwoSum(s1, t4, s2);
        }
    }
    else {
        s0 = quickTwoSum(s0, t2, s1);
        if (s1 != 0.0) {
            s1 = quickTwoSum(s1, t3, s2);
            if (s2 != 0.0)
                s2 = quickTwoSum(s2, t4, s3);
            else
                s1 = quickTwoSum(s1, t4, s2);
        }
        else {
            s0 = quickTwoSum(s0, t3, s1);
            if (s1 != 0.0)
                s1 = quickTwoSum(s1, t4, s2);
            else
                s0 = quickTwoSum(s0, t4, s1);
        }
    }
    return float4(s0, s1, s2, s3);
}
```

Fig. 6. Cg implementations of a qf_{128} renormalization function.

```
float4 qf_add(float4 a, float4 b) {

    float4 s, err;
    // We evaluate 4 twoSums in parallel
    s = qf_twoSum(a, b, err);
    float c0, c1, c2, c3, c4, err1, err2, err3, err4;
    c0 = s.x;
    c1 = twoSum(s.y, err.x, err1);
    c2 = threeSum(s.z, err.y, err1, err2, err3);
    c3 = threeSum(s.w, err.z, err2, err4);
    c4 = threeSum(err.w, err3, err4);

    return qf_renormalize(c0, c1, c2, c3, c4);
}
```

Fig. 7. Cg implementation of a qf_{128} add-function. In addition to the twoSum, there are multiple threeSum functions taking different numbers of inputs and producing different numbers of outputs.

slowdown, according to Hida), but gives

$$a \oplus b = (1+\delta)(a+b), \text{ with } |\delta| \le 2\varepsilon_{qf128} \quad (13)$$

Multiplication, shown in Fig. 8, requires additional n-Sum functions. The 4×4 product terms can be simplified by using single-precision multiplication when error terms will be too small for the representation.

Division can be computed using an iterative long-division method followed by a renormalization of the quotient terms (Hida [17]); the current Cg implementation instead uses Newton-Raphson iteration and exploits Karp's methods for reducing the necessary precision at each step.

C. Compilation Issues vis-a-vis the Cg Language

Cg 1.5 vs. Cg 2.0 (not available on Cg website, only with NVidia OpenGL SDK). Geometry programs and advanced fragment program capabilities are only available on Geforce 8800 and equivalent Quadra cards, and require Cg 2.0 to access the advanced features. (Cg 2.0 is not currently [xxAT] posted on the NVidia Cg developer's page, but can downloaded bundled with their most recent OpenGL SDK.) Cg 1.5 allows limited extended precision computation for df_{64} primitives including floating-point division and square-root (Mandelbrot set example), but precision is not adequate for computation of series summations for trigonometric, exponential, and logarithmic functions. These can be computed using Taylor series, Newton iteration, or Padé approximation on the 8800 series cards.

Computation using qf_{128} representations requires 8800-level hardware for correct renormalization. Cg 2.0 is required to make use of the advanced machine instructions available in the advanced fragment program specifications, allowing robust looping, branching, and arbitrary length fragment programs.

D. Case Study: a GPU Implementation of the Fast Fourier Transform in df_{64}

A discrete Fourier transform is a linear transformation of a complex vector:

$$\mathcal{F}(\vec{x}) = W\vec{x}$$

where, for complex $\vec{x}$ of length n, $W_{kj} = \omega_n^{kj}$, for

$$\begin{aligned}\omega_n &= \cos(2\pi/n) - i\sin(2\pi/n)\\ &= e^{-2\pi i/n}.\end{aligned}$$

"Fast" versions of this tranformation allow the matrix product to be computed in $\mathcal{O}(n\log n)$ operations rather than $\mathcal{O}(n^2)$, and play pivotal roles in signal processing and data analysis.

A general discussion of FFTs on GPU hardware can be found in Moreland et al. [32]. A discussion of issues involved in parallel computation of FFTs is Bailey [33]. The GAMMA group at UNC-Chapel Hill has released the GPUFFTW library for single-precision FFTs on real and complex input [xxAT].

This implementation uses a transposed Stockham autosort framework for the FFT; a discussion such methods is found in Van Loan [34]. The Stockham autosort methods avoid bit-reversal rearrangement of the data at the cost of a temporary storage array (necessary for a GPU-based algorithm in any case) and a varying index scheme for each of the $\ln n$ iterations. The GAMMA group at UNC employ such a framework in their single-precision GPUFFTW library.

A basic algorithm for a transposed Stockham transform is shown in Fig. 8). This presentation uses a precomputed W_{long} vector containing the $n-1$ unique Fourier-basis coefficients for the transform. This was done in the GPU implementation to avoid costly extended-precision trigonometric computations on the fragment processors; for single-precision GPU transforms, the difference between the texture-fetch of precomputed values and the use of hardware-based sine and cosine functions for *direct call* by fragment programs is negligible. This algorithm

Algorithm 8 Sequential Transpose Stockham Autosort FFT

```
Require: $X$ is a complex array of power-of-two length $n$
Require: $W_{long} = [\omega_2^0, \omega_4^0, \omega_4^1, \omega_8^0, \omega_8^1, \omega_8^2, \omega_8^3, \omega_0^4, \dots, \omega_n^{n/2-1}]$
 1: procedure TS-FFT($X, n$)
 2:     $t \leftarrow \log_2 n$
 3:     for $q \leftarrow 1, t$ do
 4:         $L \leftarrow 2^q$
 5:         $L^* \leftarrow L/2$
 6:         $r \leftarrow n/L$
 7:         $Y \leftarrow X$
 8:         for $k \leftarrow 0, r-1$ do
 9:             for $j \leftarrow 0, L^*-1$ do
10:                 $\tau \leftarrow W_{long}[L^*+j-1] \cdot Y[(k+r)L^*+j]$
11:                 $X[kL+j] \leftarrow Y[kL^*+j] + \tau$
12:                 $X[kL+L^*+j] \leftarrow Y[kL^*+j] - \tau$
13:             end for
14:         end for
15:     end for
16:     return $X$                                ▷ the DFT of the input
17: end procedure
```

can be implemented using a Cg fragment program to perform the inner two loops, storing the initial X vector as a `dataWidth*dataHeight RGBA32` floating point texture in a *framebuffer object* (*FBO*), writing into a buffer in the FBO, and *ping-ponging* between input and output buffers after each of the $log_2 n$ iterations of the outer loop. Algorithm 9 shows the CPU support structure for a data-streaming GPU implementation; Algorithm 10 shows the necessary work on the GPU. These assume df_{64} format, with each pixel storing a single complex value as four 32-bit floats: $[real_{hi}, real_{lo}, imag_{hi}, imag_{lo}]$. Modifications for qf_{128} make it simpler to use pairs of parallel images containing real and imaginary components; a fragment program can read from both input images and write to two output buffers, or separate fragment programs can be run to compute the real and imaginary results separately. (GPU systems can take a performance hit for dual-buffer output from a single shader; this must be weighed against the cost of running the fragment program twice.) An implementation can *ping-pong* between pairs of read and write buffers just as for the single-input, single-output buffer case. This implementation achieves highest efficiency when it minimizes the data load and readback to and from GPU; once loaded, data can be kept in *framebuffer objects* (*FBOs*) on the GPU and available for spatial and frequency domain operations with minimal bus-traffic to and from the CPU. Appendix I contains C++ and Cg code implementing these algorithms: Figure 9 shows the CPU-side scaffolding code; Figure 10 shows the Cg fragment program.

If a precomputed W_{long} vector is provided in an input texture, the FFT can be computed on a 6800/7800/7900 card using an $fp40$ profile under Cg 1.5. If direct-computation of the Fourier

```
float4 qf_mult(float4 a, float4 b) {

    float p00, p01, p02, p03, p10, p11, p12, p13, p20, p21, p22, p30, p31;
    float q00, q01, q02, q03, q10, q11, q12, q20, q21, q30;

    // as below, can replace 10 function calls with 3, using correct swizzling
    p00 = twoProd(a.x, b.x, q00);
    p01 = twoProd(a.x, b.y, q01);
    p02 = twoProd(a.x, b.z, q02);
    p03 = twoProd(a.x, b.w, q03);
    p10 = twoProd(a.y, b.x, q10);
    p11 = twoProd(a.y, b.y, q11);
    p12 = twoProd(a.y, b.z, q12);
    p20 = twoProd(a.z, b.x, q20);
    p21 = twoProd(a.z, b.y, q21);
    p30 = twoProd(a.w, b.x, q30);

    p13 = a.y*b.w;
    p22 = a.z*b.z;
    p31 = a.w*b.y;

    float c0, c1, c2, c3, c4, errEoutHi, errEoutLo, errE2outHi, errE2outLo, errE3out;

    c0 = p00;
    c1 = threeSum(p01, p10, q00, errEoutHi, errEoutLo);
    c2 = sixThreeSum(p02, p11, p20, q01, q10, errEoutHi, errE2outHi, errE2outLo);
    c3 = nineTwoSum(p03, p12, p21, p30, q02, q11, q20, errEoutLo, errE2outHi, errE3out);
    c4 = nineOneSum(p13, p22, p31, q03, q12, q21, q30, errE2outLo, errE3out);

    return qf_renormalize(c0, c1, c2, c3, c4);
}
```

Fig. 8. Cg implementation of a qf_{128} multiplier.

TABLE II

GPU PRECISION FOR df_{64} FLOATING-POINT COMPUTATION. BOUNDS ARE IN *ulps* FOR 48 CONTIGUOUS SIGNIFICANT BITS. COMPARISONS ARE TO d_{64} CPU COMPUTATIONS PERFORMED ON df_{64}-PRECISION VALUES

Operation	Operand range	G80 max $\|error\|$	G80 RMS $error$
Addition	$[-1,1]$	1.1	0.12
Subtraction	$[-1,1]$	1.1	0.12
Multiplication	$[-1,1]$	2.5	0.33
Division	$[-1,1]$	4.1	0.48
Reciprocal	$[-1,1]$	3.1	0.40
Recip. sqrt	$(0,1)$	4.4	0.55
Square root	$[0,1]$	4.5	0.46
$\exp(x)$	$[-1,1]$	10.6	1.7
$\ln(1+x)$	$[1,2]$	11.0	1.7
$\sin(x)$	$[-\frac{\pi}{2},\frac{\pi}{2}]$	7.8	0.94
$\cos(x)$	$[-\frac{\pi}{2},\frac{\pi}{2}]$	241.3	6.0

basis-coefficients is needed, an 8800 card and Cg 2.0 is required.

E. Timings and Accuracy of GPU-Based FFT Routines

[Addendum (July 2009): Results from this section have been deleted.]

The timing and accuracy tests were done on input vectors of length 2^m, for even values of m. This allowed the use of square $2^{m/2} \times 2^{m/2}$ image-textures for data transfer and storage on the GPU. (Since modern APIs allow non-square textures, it requires only trivial modifications of the current code to allow odd power-of-two vectors.) Timings were done on forward transformations of complex-to-complex data. Precision experiments were made, as per Bailey [33] by comparison of root-mean-square error (RMSE) of pseudo-random complex vector component magnitudes and phases with inverse transformations of their forward transforms:

$$\begin{aligned} \text{RMSE} &= \sqrt{\frac{1}{2^m}\sum_{i=0}^{2^m-1}[M(x[i]) - M(x'[i])]^2}, && (14)\\ \text{where } \vec{x}' &= \mathcal{F}^{-1}(\mathcal{F}(\vec{x})) && (15)\\ \text{and } M(a) &= \text{mag}(a) \text{ or phase}(a). && (16) \end{aligned}$$

Peak signal-to-noise ratios were also computed [xxAT]:

$$\text{Peak } s/n \quad = \quad 10\log_{10}\frac{I_{max}^2}{\text{MSE}} \qquad (17)$$

where MSE is the mean-square-error and I_{max} is the maximum signal strength.

Computations of GFLOPs were made by dividing the time for a single forward FFT by the $5m \cdot 2^m$ approximation of the floating point operations required by the transpose Stockham algorithm (see van Loan [34]).

Algorithm 9 Sequential (CPU) Portion of Parallel Transposed Stockham FFT

```
Require: $I$: a complex array of power-of-two length $N$; each element is a sequence of four 32-bit floats: $[real_{hi}, real_{lo}, imag_{hi}, imag_{lo}]$.
Require: $RB$, $WB$: $n \times m = N$ element GPU pixel-buffers
Require: $W_{long}$ an $n \times m$ texture storing the $(N-1)$ $\omega$ coefficients
1: procedure TS-FFT-CPU($I$, $N$, $n$, $m$)
2:     $WB \leftarrow I$                                  ▷ load data to GPU
3:     $t \leftarrow \log_2 N$
4:     for $q \leftarrow 1, t$ do
5:         $L \leftarrow 2^q$
6:         $RB \leftrightarrow WB$                        ▷ swap buffers on the GPU
7:         Framebuffer $\leftarrow WB$
8:         $data \leftarrow RB$
9:         Framebuffer $\leftarrow$ TS-FFT-GPU($data, W_{long}, L, N, m$)
10:    end for
11:    return readback from $WB$ (last output Framebuffer)
12: end procedure
```

1) Results using Precomputed W_{long} Values: Tables [DELETED] show timing and accuracy comparisons for df_{64}-based FFTs as described. Timings were made on two platforms: an NVidia Geforce 8800 GT running on a 3.2 GHz single-processor Pentium IV front-end; an NVidia 7900go [xxAT] running on a Centrino Duo front-end. Both platforms used the Windows XP operating system with the Intel C++ 9.0 compiler; the NV7900 vector hardware code was compiled under Cg 1.5; the NV8800 scalar hardware code was compiled under Cg 2.0.

Timing and accuracy comparisons were against $d_{64}64$ transposed Stockham FFTs implemented on the CPU on the respective front-end machines. While the sequential FFT code was not heavily optimized, neither was the GPU code: a radix-4 implementation as described in Bailey [33] could be expected to give a 2x to 4x improvement in runtime and potential improvements in accuracy as well.

2) Results using Direct Computation of W_L^{jk} Values: Table [DELETED] shows similar timings and accuracy to the above, using direct call to the extended-precision df_{64} sine-cosine function on the NV8800 GPU. While these results produce marked slowdowns over the use of pre-computed values, they illustrate the accuracy of the library functions and their usefulness in situations that might require few calls.

Appendix I
C++ and Cg Implementation of the Transposed Stockham FFT

Fig. 9 and Fig. 10 present examples of C++ and Cg implementations of the texture-based streaming version of the transposed Stockham FFT.

Acknowledgment

The author would like to thank Dr. David Bremer (LLNL) and Dr. Jeremy Meredith (ORNL) for discussions of their experiments, Dr. Mark Harris for his information on GPU floating-point performance and other GPGPU issues, and Dr. Dinesh Manocha and his colleagues in UNC's GAMMA group.

Algorithm 10 Streaming (GPU) Portion of Parallel Transpose Stockham FFT

```
Require: input as described above in Alg. 9
Require: integer texture coordinates $(c_x, c_y)$ of the fragment
1: procedure TS-FFT-GPU($data, W_{long}, L, N, m$)
2:     $R \leftarrow N/L$
3:     $L^* \leftarrow L/2$
4:     $i \leftarrow c_y m + c_x$                   ▷ array index in $I$ of fragment
5:     $k \leftarrow i/L$
6:     $j \leftarrow i - Lk$
7:     if $j \geq L^*$ then            ▷ distinguish $j < L^*$ from $j > L^*$
8:         $j \leftarrow j - L^*$
9:         $s_\tau = -1.0$                        ▷ sign multiplier for $\tau$
10:    else
11:        $s_\tau = 1.0$
12:    end if
13:    $d_1 \leftarrow L^* + j - 1$                 ▷ index of $\omega$ in Wlong
14:    $\omega \leftarrow$ lookup($W_{long}, d_1/m, d_1 \mod m$)
15:    $d_2 \leftarrow kL^* + R + j$
16:    $Y \leftarrow$ lookup($data, d_2/m, d_2 \mod m$)
17:    $\tau = s_\tau \omega Y$
18:    $d_3 \leftarrow kL^* + j$
19:    $X \leftarrow$ lookup($data, d_3/m, d_3 \mod m$)
20:    return $X + \tau$
21: end procedure
```

References

[1] D. H. Bailey, "High-precision floating-point arithmetic in scientific computation," *Computing in Science and Eng.*, vol. 7, no. 3, pp. 54–61, 2005.

[2] F. de Dinechin, "High precision numerical accuracy in physics research," Workshop presentation, Advanced Computing and Analysis Techniques in Physics Research (ACAT 2005), Zeuthen, Germany, May 2005.

[3] D. Göddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, Jan. 2007.

[4] T. H. Myer and I. E. Sutherland, "On the design of display processors," *Commun. ACM*, vol. 11, no. 6, pp. 410–414, 1968.

[5] T. J. Dekker, "A floating point technique for extending the available precision," *Numerische Mathematik*, vol. 18, pp. 224–242, 1971.

[6] S. Linnainmaa, "Software for doubled-precision floating-point computation," *ACM Trans. on Mathematical Software*, vol. 7, no. 3, pp. 272–283, September 1981.

[7] R. P. Brent, "A FORTRAN multiple-precision arithmetic package," *ACM Trans. on Math. Software*, vol. 4, no. 1, pp. 57–70, March 1978.

[8] D. Smith, "A Fortran package for floating-point multiple-precision arithmetic," *ACM Transactions on Mathematical Software*, vol. 17, no. 2, pp. 273–283, June 1991.

[9] D. H. Bailey, "A Fortran 90-based multiprecision system," *ACM Trans. Math. Softw.*, vol. 21, no. 4, pp. 379–387, 1995.

[10] P. Zimmermann, "Arithmétique en précision arbitraire," LORIA/INRIA, Lorraine, Fr., rapport de recherche INRIA 4272, Septembre 2001.

[11] D. M. Priest, "On properties of floating point arithmetics: Numerical stability and the cost of accurate computations," Ph.D. dissertation, University of California, Berkeley, 1992.

[12] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete & Computational Geometry*, vol. 18, no. 3, pp. 305–363, October 1997.

[13] IEEE, "An American national standard: IEEE standard for binary floating-point arithmetic," *ACM SIGPLAN Notices*, vol. 22, no. 2, pp. 7–18, 1997.

[14] D. H. Bailey, "A Fortran-90 double-double precision library," Technical Report," Libraries available at http://crd.lbl.gov/~dhbailey/mpdist/index.html, 2001.

[15] K. Briggs, "Doubledouble floating point arithmetic," http://keithbriggs.info/doubledouble.html, Tech. Rep., 1998.

```
void TransStockhamFFT::update()
{
    myFBO.Bind();
    glViewport(0, 0, dataWidth, dataHeight);

    readID = 0;
    writeID = 1;
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB,
                  texID[writeID]);

    glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,
                    0, 0, 0,
                    dataWidth, dataHeight,
                    GL_RGBA, GL_FLOAT, inputIm);

    int N = dataWidth*dataHeight;
    int T = intLog2(N);
    float L = 1;
    float R = N;
    for (int q = 1; q <= T; q++) {

        int temp = readID;
        readID = writeID;
        writeID = temp;
        L *= 2;
        R /= 2;

        glDrawBuffer(frameBuffers[writeID]);

        cgGLBindProgram(transStockhamFP);
        cgGLEnableProfile(g_cgProfile);

        cgGLSetTextureParameter(texParam,
                                texID[readID]);
        cgGLEnableTextureParameter(texParam);
        cgGLSetTextureParameter(wLongParam,
                                texID[3]);
        cgGLEnableTextureParameter(wLongParam);
        cgGLSetParameter1f(fwdFFTParam, 1.0f);
        cgGLSetParameter4f(LRNIParam, L, R,
                           (float) N,
                           (float) dataWidth);
        glBegin(GL_QUADS);
        {
            glTexCoord2f(0,0);
            glVertex2f(0, 0);
            glTexCoord2f(dataWidth, 0);
            glVertex2f(1, 0);
            glTexCoord2f(dataWidth, dataHeight);
            glVertex2f(1, 1);
            glTexCoord2f(0, dataHeight);
            glVertex2f(0, 1);
        }
        glEnd();
        cgGLDisableTextureParameter(texParam);
    }

    cgGLDisableProfile(g_cgProfile);
    FramebufferObject::Disable();
}
```

Fig. 9. CPU-side code for the GPU-based FFT

```
#include "extPrecision.cg"

float4 mainTransStockham(float2 coords : TEX0,
                uniform samplerRECT texture,
                uniform samplerRECT Wlong,
                uniform float FWDFFT,
                uniform float4 LRNI) : COLOR
{
    int locX = (int) coords.x;
    int locY = (int) coords.y;

    int L = (int) floor(LRNI.x);
    int R = (int) floor(LRNI.y);
    int N = (int) floor(LRNI.z);
    int IMAGESIZE = (int) floor(LRNI.w);

    int LStar = L/2;
    int i = locY*IMAGESIZE + locX;
    int k = i / L;
    int j = i - L*k;

    float tauMult = 1.0f;
    if (j >= LStar) {
        j = j - LStar;
        tauMult = -1.0f;
    }

    int dex = LStar + j - 1;
    int dex1 = dex/IMAGESIZE;
    int dex2 = dex - IMAGESIZE*dex1;
    float4 W = texRECT(Wlong,
                   float2(dex2, dex1));

    dex = k*LStar + j;
    dex1 = dex/IMAGESIZE;
    dex2 = dex - IMAGESIZE*dex1;
    float4 src = texRECT(texture,
                     float2(dex2, dex1));

    dex += R*LStar;
    dex1 = dex/IMAGESIZE;
    dex2 = dex - IMAGESIZE*dex1;
    float4 Y = texRECT(texture,
                   float2(dex2, dex1));

    float4 tauPlain = cdf64_mult(Y, W);
    float4 tau = tauPlain*tauMult;

    float4 dst = cdf64_add(src, tau);

    if (FWDFFT == -1.0 && L == N)
        dst /= N;

    return dst;
}
```

Fig. 10. Cg code for the GPU fragment code. Slight speedups are achievable by replacing conditional branches with calls to separate fragment programs. Counterintuitively, elimination of the **floor()** functions produces slight slowdowns. (This is with the Cg 1.5 compiler. See discussion of Cg 1.5 vs. 2.0 differences in Sec. III-C

[16] Y. Hida, X. S. Li, and D. H. Bailey, “Algorithms for quad-double precision floating point arithmetic,” in *Proceedings of the 15th Symposium on Computer Arithmetic (ARITH ’01)*, N. Burgess and L. Ciminiera, Eds. Washington, DC: IEEE Computer Society, 2001, pp. 155–162. [Online]. Available: http://citeseer.ist.psu.edu/hida01algorithms.html

[17] ——, “Quad-double arithmetic: algorithms, implementation, and application,” Lawrence Berkeley National Laboratory, Berkeley, CA, Tech. Rep. LBNL-46996, October 30 2000. [Online]. Available: citeseer.ist.psu.edu/article/hida00quaddouble.html

[18] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, “Design, implementation and testing of extended and mixed precision BLAS,” *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 152–205, 2002.

[19] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, 2nd ed. Reading, Massachusetts: Addison Wesley, 1981, vol. 2.

[20] J. Meredith, D. Bremer, L. Flath, J. Johnson, H. Jones, S. Vaidya, and R. Frank, “The GAIA Project: evaluation of GPU-based programming environments for knowledge discovery,” Presentation, HPEC ’04, Boston, MA, 2004.

[21] A. L. Thall, “Extended-precision floating-point numbers for GPU computation,” Poster Session, ACM SIGGRAPH ’06 Annual Conference, Boston, MA, August 2006.

[22] G. Da Graça and D. Defour, “Implementation of float-float operators on graphics hardware,” *preprint*, may 2006.

[23] B. Verdonk, A. Cuyt, and D. Verschaeren, “A precision and range independent tool for testing floating-point arithmetic I: basic operations, square root and remainder,” *ACM Transactions on Mathematical Software*, vol. 27, no. 1, pp. 92–118, 2001.

[24] R. Karpinski, “Paranoia: a floating-point benchmark,” *Byte Magazine*, vol. 10, no. 2, pp. 223–235, February 1985.

[25] K. Hillesland and A. Lastra, “GPU floating-point Paranoia,” in *Pro-*

ceedings of the ACM Workshop on General Purpose Computing on Graphics Processors. Los Angeles: ACM, August 2004, Code available at http://www.cs.unc.edu/~ibr/projects/paranoia/, pp. C–8.

[26] A. H. Karp and P. Markstein, "High-precision division and square root," *ACM Trans. Math. Softw.*, vol. 23, no. 4, pp. 561–589, 1997.

[27] J.-M. Muller, *Elementary Functions: Algorithms and Implementation, 2nd Edition*. Birkhauser, 2005.

[28] G. A. Baker, *Essentials of Padé Approximation*. New York: Academic Press, 1975.

[29] S. Gal, "An accurate elementary mathematical library for the ieee floating point standard," *ACM Trans. Math. Softw.*, vol. 17, no. 1, pp. 26–45, 1991.

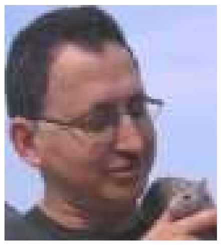
[30] N. Brisebarre, D. Defour, and N. Revol, "A new range-reduction algorithm," *IEEE Trans. Comput.*, vol. 54, no. 3, pp. 331–339, 2005, member-Peter Kornerup and Senior Member-Jean-Michel Muller.

[31] S. Boldo and J.-M. Muller, "Some functions computable with a fused-mac," *arith*, vol. 00, pp. 52–58, 2005.

[32] K. Moreland and E. Angel, "The FFT on a GPU," in *Graphics Hardware 2003*, M. Doggett, W. Heidrich, W. Mark, and A. Schilling, Eds. Eurographics, 2003.

[33] D. H. Bailey, "A high-performance FFT algorithm for vector supercomputers," *Int. Journal of Supercomputer Applications*, vol. 2, no. 1, pp. 82–87, 1988.

[34] C. van Loan, *Computational Frameworks for the Fast Fourier Transform*, ser. Frontiers in Applied Mathematics. SIAM, 1992, Another wonderful book from Charlie van Loan. Let's send this guy lots of money.



Andrew Thall is Assistant Professor of Mathematics and Computer Science at Alma College. He received his Ph.D. (2004) in Computer Science from the University of North Carolina at Chapel Hill, where his main areas of research were in medical-image analysis and medial-based geometric surface modeling. His current research interests include numerical computation, GPGPU, and computer science education.