A High-Performance Area-Efficient Multifunction Interpolator

Stuart F. Oberman and Michael Y. Siu
NVIDIA Corporation
Santa Clara, CA 95050
e-mail: {soberman,msiu}@nvidia.com

Abstract

This paper presents the algorithms and implementation of a high-performance functional unit used for multiple interpolation applications. Graphics processing units (GPUs) frequently perform two classes of floating point interpolation within programmable shaders: per-pixel attribute interpolation and transcendental function approximation. We present a design that efficiently performs both classes of interpolation on a shared functional unit. Enhanced minimax approximations with quadratic interpolation minimize lookup-table sizes and datapath widths for fully-pipelined function approximation. Rectangular multipliers support both sign-magnitude and two's complement inputs of variable widths. Superpipelining is used throughout the design to increase operating frequency and interpolation throughput while maximizing area efficiency.

1 Introduction

The rendering of 3D graphics places extreme demands on computational hardware. Dedicated graphics processing units, or GPUs, are designed to enable high performance 3D graphics and multimedia. Modern GPUs use programmable numerical processors, known as *shaders*, for their core computation. These shaders support general-purpose computing functionality, including single-precision floating-point operations, loops, branches, dynamic flow control and with no fixed program length limits. These programmable shaders can then be used to implement the vertex and pixel processing components of the 3D rendering pipeline in a very powerful and flexible manner.

In a GPU, programmable *vertex shaders* perform various transformations and computation on primitives or triangles. They also compute values for various attributes at each of the triangles' vertices. Example attributes include color, depth and texture coordinates, all represented as single-precision floating-point values [5]. Plane equations are often used to represent the values of computed attributes as they vary across a triangle. These plane equations are determined from the attributes' values at the three vertices. Programmable *pixel shaders* compute the value of each of these attributes at each pixel location through planar interpolation.

Typical instructions sets of modern vertex and pixel shaders support the programming paradigms of various high-level APIs, including OpenGL [10] and Microsoft's DirectX9 [6]. The computational elements of the shaders ISAs therefore include traditional floating-point addition and multiplication, along with support for higher-order functions, including reciprocal, reciprocal square-root, logarithm, exponential, and trigonometric functions. The working precision of modern shaders is typically IEEE single-precision, with 8b exponents and 23b fractions. Thus, vertex and pixel shaders contain hardware to compute high throughput, highly accurate single-precision transcendental functions. Furthermore, a pixel shader contains hardware to perform fast and efficient floating-point planar interpolation of triangle attributes at each pixel location.

This paper discusses a hardware design that supports the computation of both transcendental functions and planar attribute interpolation within the same unit. A traditional pixel shader contains a functional unit to compute transcendental functions. It also contains an attribute interpolating unit to compute the per-pixel attribute values. Each of these functional units consumes chip area and power, adding to the overall GPU design complexity and cost. Since each unit is specialized, we determined that neither unit can be kept completely busy during the course of standard shader operation. We determined that we can merge the two units into a single shared functional unit, as they have very similar computational cores. Even after merging, architectural studies confirmed that the shared functional unit still satisfies the overall pixel shader's throughput requirements for each of these operations. This shared multifunction unit therefore reduces overall chip area and power, and it can

Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH'05)
1063-6889/05 $20.00 © 2005 IEEE

be used more efficiently. Our investigations have also demonstrated that within certain bounds, we are able to further increase area-efficiency by operating the functional unit at higher frequency.

In section 2, we discuss our method for evaluating transcendental functions. In section 3, we discuss the algorithms and implementation of per-pixel attribute interpolation. In section 4, we present the design of our new shared multifunction interpolator, describing which and how components can be shared. We present conclusions in section 5.

2 Transcendental Function Evaluation

2.1 Overview

Several methods exist for evaluating transcendental functions. These methods can be divided into two classes: iterative and non-iterative. The iterative class includes digit-recurrence and functional iteration algorithms. Digit recurrence algorithms use subtraction as the fundamental operator, and they converge linearly to the result [4]. Functional iteration algorithms, including Newton-Raphson and Goldschmidt's algorithms, use multiplication as the primary operator, and they converge quadratically to results [9]. Non-iterative methods include direct table lookup, polynomial and rational approximations, and table-based methods [3, 7, 14, 15].

These two classes of algorithms typically trade off execution time and throughput for area and complexity. Iterative algorithms are very well-suited to forming high-precision function approximations, as approximation precision can be increased either linearly or quadratically with execution time, with only minor impact on implementation complexity. In contrast, non-iterative algorithms such as table lookup require non-linear increases in complexity to support higher-precision approximations.

For lower-precision approximations, both methods can be attractive. In GPUs, the common target precision is 32b IEEE single-precision. In this design space, the complexity required to implement the non-iterative algorithm class is not prohibitive, and the gains in execution time and throughput can be substantial. The class of non-iterative algorithms can be further divided based upon the size of the lookup tables and the computational complexity involved. Tang [15] describes table-based methods which rely on a small table in conjunction with a high-degree polynomial approximation, requiring a significant number of additions and multiplications to produce a function estimate. In contrast, several methods have been proposed which rely on larger tables with less computation. Direct table lookup and bipartite methods [3, 12] require very large initial lookup tables combined with very little computation, typically at most a few integer additions. Function approximation methods based on linear approximations [3, 14] and second-degree interpolation [1, 11, 13] have also been proposed which more closely balance table size with computation. With this balance, these linear and quadratic interpolation methods are very suitable for approximating functions to single-precision floating-point, combining practical hardware requirements with fast execution time.

In this paper, we employ a method for single-precision transcendental function evaluation based on the proposal of Pineiro and Oberman [11]. This method uses quadratic interpolation for approximating the reciprocal, reciprocal square-root, logarithm ($\log_2 X$), exponential (2^X), sin, and cos functions. The primary requirements of our implementation were four-fold:

1. Reciprocal approximation accurate to within 1 ulp and monotonic
2. Reciprocal square root approximation accurate to within 2 ulps and monotonic
3. Area-efficiency maximized
4. Targeted at high operating frequency, fully-pipelined supporting one result per clock

The remaining functions should also have high accuracy. However, their resulting accuracies are based upon that provided by the hardware required to support the primary requirements.

2.2 Quadratic Interpolation Algorithm

2.2.1 Coefficient Generation

Approximating a transcendental function requires three steps [15] : 1) argument reduction to a predetermined input interval, 2) function approximation of the reduced argument, and 3) reconstruction, normalization and rounding of the final result. Our proposed functional evaluator presumes that the input argument has been reduced to the appropriate interval, depending upon the approximated function. A table lookup is used to retrieve optimized coefficients for a quadratic polynomial approximation, followed by a high-speed evaluation of the polynomial.

For a binary input operand X with n-bit significand, the significand is divided into two parts: X_u is the upper part containing m bits, and X_l is the lower part containing $n - m$ bits. The upper m bits X_u are used to consult a set of three lookup tables to return three finite-word coefficients C_0, C_1, and C_2. Each function to be approximated requires a unique set of tables. These coefficients are used to approximate $f(X)$ in the range

$X_u \leq X < X_u + 2^{-m}$ by evaluating the expression

$$f(X) \approx C_0 + C_1 X_l + C_2 X_l^2 \quad (1)$$

The method for forming the coefficients is based on an Enhanced Minimax Approximation, as described in detail in [11]. The three steps of this hybrid algorithm are:

1. Use a numerical software package, such as Maple, to perform the Remez exchange algorithm in order to compute the second-order minimax polynomial [7] for the function to be approximated. The returned minimax polynomial has full-length coefficients a_0, a_1, and a_2 for the degree-0, degree-1, and degree-2 terms respectively. We round the returned degree-1 coefficient a_1 to p bits to form C_1.

2. Given C_1, we form a new function to be approximated by another second-order polynomial:

$$\begin{aligned} f(X) &\approx a_0 + a_1 X_l + a_2 X_l^2 \\ &= a'_0 + C_1 X_l + a'_2 X_l^2 \end{aligned} \quad (2)$$

   which can be rewritten as

$$(a_1 - C_1) X_l = (a'_0 - a_0) + (a'_2 - a_2) X_l^2 \quad (3)$$

   By substitution of $Y = X_l^2$, the equation becomes:

$$(a_1 - C_1) \times \sqrt{Y} = (a'_0 - a_0) + (a'_2 - a_2) Y \quad (4)$$

   This presents a linear approximation of the function $\sqrt{Y}$. In Pineiro [11], it is shown that determining the minimax polynomial coefficients for this particular function using automated techniques is not generally feasible. Instead, the technique of Muller [8] can be used to solve for the coefficient of Y and thus a'_2:

$$a'_2 := a_2 + (a_1 - C_1) \times 2^m \quad (5)$$

   a'_2 is rounded to q bits to form C_2. This step effectively allows C_2 to include not only the effect of a_2 rounded to finite precision, but it also compensates for some of the effects of having rounded C_1 to p bits.

3. Use Maple to find a new minimax degree-2 polynomial to a compensated version of the original function

$$f(X) - C_1 X_l - C_2 X_l^2 \quad (6)$$

   The degree-0 coefficient of the returned minimax polynomial is rounded to t bits to form C_0. This coefficient includes compensation for having rounded both C_1 and C_2 to finite precision.

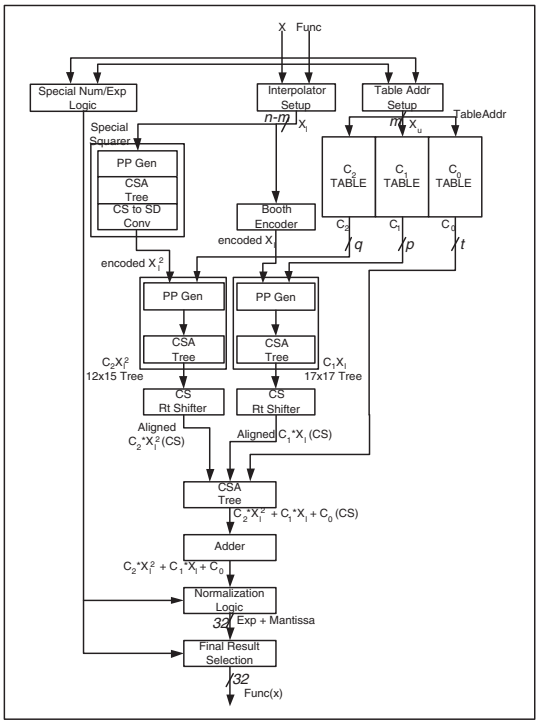

Figure 1. Function evaluator block diagram

The design challenge is to optimize for the values of m, p, q and t such that the desired accuracy is achieved, while minimizing area and delay. After the coefficients are computed, we employ exhaustive simulation across the approximation interval using a C model of the hardware implementation. This is computationally feasible for single-precision operands. This simulation may show that the real maximum error is less than what is mathematically possible, and thus allow us to further adjust the parameters for better area or latency.

2.2.2 Fast Polynomial Evaluation

Once the coefficients are computed, it is necessary to evaluate the polynomial of equation 1. Figure 1 is a block diagram of our method. We employ three special techniques to accelerate the computation. First, we use a specialized squaring unit to evaluate X_l^2, rather than using a standard multiplier. It uses a well-known technique of re-arranging the partial product matrix due to identical multiplier inputs [2]. It also only keeps a subset of the most-significant output bits. We further truncate input bits and partial product bits themselves which contribute toward output bits which would ultimately be truncated. Truncating these bits introduces additional error. For a function with $m = 6$, X_l has 17b. The size of X_l^2 can likely be strictly less than 17b. In [11], it

has been shown that truncating at position 2^{-28} of the squarer output, corresponding to leaving 16b for X_l^2, results in a maximum error of 1.0×2^{-25}. Furthermore, in our design we also add a function-specific bias which accommodates this truncation error. Therefore, we can optimize both squarer truncation and function bias in order to minimize the size of the squarer. The squarer must be designed to handle the largest number of bits $n - m$ possible for all approximated functions. This typically means that the squarer size is governed by the least accurate function approximated, the one with the lowest value of m.

To implement the product $C_2 \times X_l^2$, a straightforward approach involves a conversion from the carry-save output of the squarer to a non-redundant form, followed by Booth-encoding upon input to this multiplier. Instead, we convert the squarer result directly from carry-save representation to radix-4 signed-digit, which can be used to directly select partial products within the multiplier tree. This carry-save to signed-digit converter has area and delay only slightly larger than that of a standard radix-4 Booth recoder unit, but a carry-propagate adder is avoided in the process, reducing overall latency.

Finally, the individual terms $C_2 \times X_l^2$, $C_1 \times X_l$ and C_0 must be summed together. Depending upon the approximated function, the weights of each of the coefficient terms may be different. For a given function, such as reciprocal, the weights of C_0, C_1, and C_2 can be hardwired into the summation tree, such that the results flow into a final CSA tree directly with no required shifting. If the unit supports multiple functions the weights of the same coefficients will be different for different functions. This can be accommodated by proper arrangement of the coefficients in the lookup tables, or by explicit shifters of the resulting products $C_2 \times X_l^2$ and $C_1 \times X_l$. We also introduce a function-specific bias into the summation tree. This bias is determined in advance based upon exhaustive simulation for each function. Its purpose is to center the error distribution, reducing the overall maximum absolute error in the approximation due to all sources.

After the final summation, depending on the function the result may require some amount of normalization. While reciprocal and reciprocal square-root require at most a 1b normalization, other functions such as sin/cos and $\log_2 X$ may require many bits of normalization. These functions take bounded inputs and can produce arbitrarily small outputs.

2.3 Implementation

For our design requirements, we iterated through the hybrid algorithm to form a set of coefficients for each function to be approximated. The goal was to implement all coefficients in a single lookup table ROM. The resulting parameters for our functions are shown in Table 1. The minimum values of m for each function is shown, along with the configurations of the coefficients (widths of C_0, C_1, C_2), the accuracies resulting from these configurations, and the total lookup table sizes.

The first term, accuracy, represents the number of *good bits* in the prenormalized results, as measured by the maximum absolute error in the function approximations over the given input intervals. We also report per function accuracy in ulps, monotonicity, and the percentage of results per function that match the exactly-rounded-to-nearest result. Since sin/cos and $\log_2 X$ outputs could require many bits of normalization for the bounded input intervals, we do not report ulp error or percentage exactly-rounded for these two functions.

The ROM and datapath widths are a function of these parameters. The ROM width, 52b, is limited by the most stringent function, $1/X$. The other functions were then allowed to use all 52b for their coefficients. The $1/X$ function with $m = 7$ requires 128 entries, while the other $m = 6$ functions require 64 entries each. $1/\sqrt{X}$ requires two sets of $m = 6$ tables, one for inputs in the range [1,2), and another for those in [2,4). The total ROM size is therefore 448 x 52b. The signs of the coefficients are functions of the functions to be approximated, and thus are not stored in the tables themselves. However, the multipliers must take into account a sign bit to accommodate the signed multiplication.

The squarer is constrained by the $m = 6$ functions, and must use $23 - 6 = 17$ bits for input. We are able to truncate the squarer output, such that we only require 15 output bits. We compensate for the resulting truncation error through the addition of the function-specific biases.

The $C_1 \times X_l$ multiplier must support the worst case value of X_l, and thus it is a 17x17 signed multiplier. The $C_2 \times X_l^2$ product must support the widest C_2 term plus a sign bit for one dimension, and the maximum width of the squarer output. In our design, this can be implemented with a 12x15 signed multiplier. We accomodate different weights per function for the products $C_2 \times X_l^2$ and $C_1 \times X_l$ by using explicit shifters, which are then used both for function approximation and attribute interpolation.

3 Attribute Interpolation

3.1 Overview

In graphics applications, attributes such as color, depth, and texture are usually specified for vertices of primitives that make up a scene to be rendered. These attributes must then be interpolated in the (x,y) screen space as needed to determine the values of the attributes

Function	Input Interval	m	Configuration	Accuracy (good bits)	ulp error	% exactly rounded	monotonic	Lookup table size
$1/X$	[1,2)	7	26,16,10	24.02	0.98	87%	yes	6.50Kb
$1/\sqrt{X}$	[1,4)	6	26,16,10	23.40	1.52	78%	yes	6.50Kb
2^X	[0,1)	6	26,16,10	22.51	1.41	74%	yes	3.25Kb
$\log_2 X$	[1,2)	6	26,16,10	22.57	n/a	n/a	yes	3.25Kb
sin/cos	$[0, \pi/2]$	6	26,15,11	22.47	n/a	n/a	no	3.25Kb
Total								22.75Kb

Table 1. Quadratic polynomial coefficients parameters for five functions

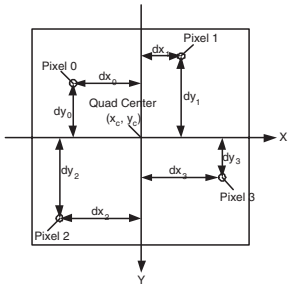

Figure 2. Pixel quad layout

at each pixel location. The value of a given attribute U in an (x,y) plane can be expressed as a plane equation of the form:

$$U(x, y) = A \times x + B \times y + C \tag{7}$$

where A, B, and C are interpolation parameters associated with each attribute U. A GPU often includes a separate fixed-function circuit to compute these interpolation parameters from the values at the vertices. A pixel shading unit must be able to compute the value of each attribute for any pixel location and thus implement in hardware equation 7.

As a hardware optimization, multiple attribute samples, one per pixel, can be computed in parallel with the same hardware unit. For example, Figure 2 illustrates a 2x2 arrangement of pixels, referred to as a pixel *quad*, comprising four pixels that meet at a quad center, which has coordinates (x_c, y_c).

Individual pixel 0 is offset by (dx_0, dy_0), pixel 1 is offset by (dx_1, dy_1), and so forth. For a given sample point with index i ($i = 0, 1, 2, 3$), the revised plane equation to be evaluated is:

$$U(x_i, y_i) \quad = \quad A \times x_c + B \times y_c + \Delta_i \tag{8}$$

where

$$\Delta_i = A \times dx_i + B \times dy_i + C \tag{9}$$

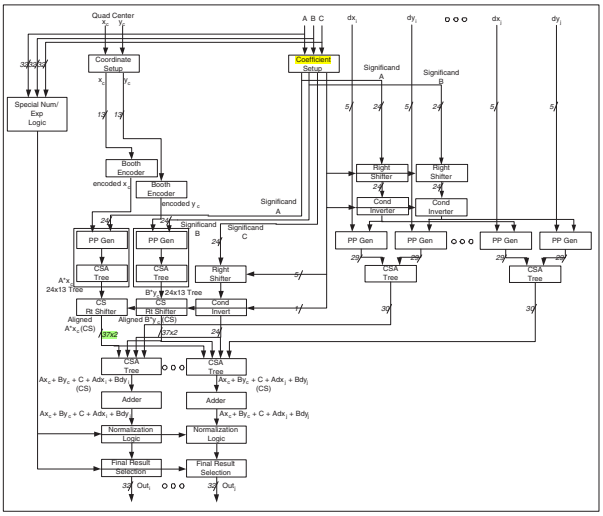

Figure 3. Attribute interpolator block diagram

In this proposed interpolator, the quad center coordinates (x_c, y_c) are 13b two's complement integers, and the coordinate offsets (dx_i, dy_i) are signed 5b fractions restricted to the interval (-1,1). The interpolation parameters A, B, and C are all represented as 32b single-precision floating-point numbers. Each computed sample is also represented by a 32b single-precision floating-point number.

3.2 Implementation

A block diagram of an attribute interpolator is shown in Figure 3.

The three floating-point interpolation parameters are first block-aligned at the top of the unit, by the box labeled *coefficient setup*. This unit determines the maximum exponent of each of the three parameters, and computes the absolute difference of the exponents from this maximum for the other two parameters. These are then

used later to appropriately align the operands for final addition.

There are two main datapaths in this design: the $A \times x_c$ multiplier tree and the $B \times y_c$ multiplier tree. Each tree takes in both a 23b sign-magnitude floating-point fraction and a 13b fixed-point two's complement operand, and thus each is a 24x13 signed multiplier. The resulting product is a 37b two's complement result. Thus, the multiplier trees must convert the sign-magnitude input into two's complement form.

These two products are then appropriately aligned, based upon the results of the initial block alignment. The $A \times x_c$ product is arithmetically right-shifted by the exponent difference computed for A, and the $B \times y_c$ product is arithmetically right-shifted by the exponent difference computed for B. The products could first be carry-assimilated into non-redundant form before shifting. However, this introduces a carry-propagate adder into the critical path. The alternative is to right-align the products directly in carry-save form. Since the products are in two's complement form, special care must be taken to decide which bits to shift-in at the top of the shifters, in order that the proper sign will result after the two terms are added together. Keeping the operands in redundant format in this way reduces the unit's overall latency.

Each product is represented by a 37b $product_{sum}$ term and $product_{carry}$ term. The precomputed sign for the product can be formed based purely upon the sign of the two inputs, referred to as *signprod*. Using this information, we form the two *effsign* bits for the sum and carry terms, as a function of the most-significant bits of the products: msb_{sum} and msb_{carry}.

$$\text{effsign}_{sum} = \begin{cases} 1 & \text{if } \textit{signprod} \\ msb_{sum} \text{ OR } msb_{carry} & \text{otherwise} \end{cases}$$

$$\text{effsign}_{carry} = \begin{cases} msb_{sum} \text{ AND } msb_{carry} & \text{if } \textit{signprod} \\ 0 & \text{otherwise} \end{cases}$$

These two *effsign* bits are then used as the bits to shift into the msb's for the two right shifters. A separate set of these bits is used for each of the two products.

In parallel a separate set of small multipliers are used to form the $A \times x_i + B \times y_i$ products for each of the four sample points. Each sign-magnitude fraction A and B must first be conditionally aligned by the previously computed exponent differences. The fractions are then conditionally two's complemented based upon each operand's true sign. The (dx_i, dy_i) bits are radix-4 Booth encoded and used to select signed multiples of the shifted A and B two's complement fractions. These partial products are reduced to two via a small parallel carry-save adder tree. Similarly the C fraction is aligned and conditional two's complemented. A final set of carry-save adders is used to merge the summed (dx_i, dy_i) results with the aligned C.

Four sets of final carry-save adder trees are used to implement equation 9. This tree takes inputs in redundant form for aligned $A \times x_c$ and $B \times y_c$, and Δ_i and reduces the partial products to two. The tree is sufficiently wide to support all possible cases of overflow.

The redundant result for each sample is carry-assimilated through a carry-propagate adder. Since the two's complement result may be negative, obtaining the true final result could require a two's complement operation. In the proposed implementation, the less expensive one's complement operation is used instead, with little or no impact on overall interpolation accuracy.

The complemented results are then appropriately normalized. Since there could have been catastrophic cancellation in the addition process in the case where the input exponents were all close in magnitude, the output results may require normalization.

4 Shared Multifunction Interpolator

4.1 Overview

Given the need for both a transcendental function evaluator and an attribute interpolator in a pixel shader, it is logical to try to combine the two functions in order to save area and increase utilization. After analyzing the algorithm and implementation for both units, we discovered that the two functions share many operations and physical components. Therefore, we propose the design of a shared multifunction interpolator, which can perform quadratic interpolation for function evaluation, and planar interpolation for per-pixel attribute sampling.

4.2 Implementation

A block diagram of the proposed shared multifunction interpolator is shown in Figure 4.

Comparing Figures 3 and 4, one can notice that the only major additional hardware needed to support functional evaluation on the attribute interpolator unit is the lookup-tables and the special-squarer. The major multiplier trees require a bit of massaging, though, to support all possible configurations. The attribute interpolator requires 24x13 multiplier units for the $A \times x_c$ and $B \times y_c$ trees. However, the multiplicands are in sign-magnitude form, while the multipliers are in two's complement. In contrast, for the function evaluator the 17x17 $C_1 \times X_l$ tree requires a 17b two's complement multiplicand, and a 17b unsigned multiplier. The $C_2 \times X_l^2$ tree at 12x15 is smaller than the $C_1 \times X_l$ tree and thus is not a limiting factor.

A straightforward approach to support the cross-product of the 24x13 and 17x17 functions on the same

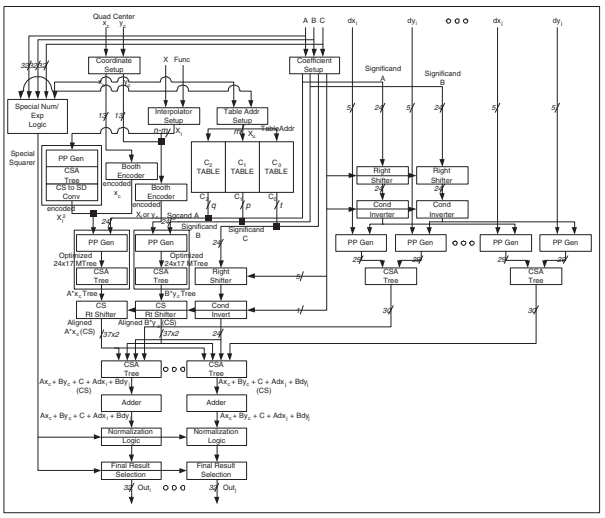

Figure 4. Multifunction interpolator block diagram

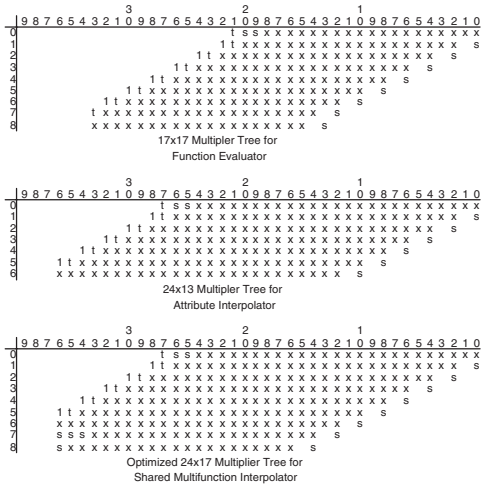

Figure 5. Dot diagrams for shared multiplier tree

hardware would require the design of a 24x17 multiplier. Instead, we took the radix-4 Booth recoded 24x13 tree, and added two partial products which are only as wide as necessary to support left-justification of the final 37b result, creating an optimized 24x17 multiplier tree. This is illustrated in Figure 5.

Supporting both two's complement (13b) and unsigned (17b) formats for the multiplier operand requires no additional logic when Booth recoding is used. However, supporting both sign-magnitude (23b) and two's complement (17b) multiplicands requires that the Booth recoding logic be slightly modified. When selecting a signed-multiple of the multiplicand, for the sign-magnitude case the recoding logic must conditionally invert the sense of the selected sign, depending upon the actual sign of the sign-magnitude multiplicand. As an example, if the Booth logic selects the -2X multiple based purely on multiplier recoding, but the sign bit of the multiplicand is negative, then this overrides the Booth logic such that it selects instead the +2X multiple.

4.3 Testing

We implemented a bit-accurate C model of the multifunction interpolator to verify correctness. We used this C model to exhaustively simulate all single-precision inputs for the five transcendental functions and to determine specific biases which minimize the maximum absolute error for each function. By comparing the results to double-precision results produced by an x86 microprocessor, we were able to validate the accuracy and correctness of our single-precision function evaluator.

We used a combination of random, directed-random, and directed tests to exercise the attribute interpolator. We compared the results of our bit-accurate attribute interpolator C model to those produced by an existing NVIDIA GPU. We validated the interpolator by ensuring our C model results matched those of an existing GPU.

4.4 Area Analysis

We created a register-transfer-language implementation of the interpolator that matched the functionality of our bit-accurate C model. We used this RTL implementation to help derive area estimates for each of the unit's major components. The implementation was designed to minimize both latency and maximize area efficiency. This meant that we did not employ all possible latency reducing techniques, such as severe gate upsizing and replication, as these tend to sacrifice area efficiency. We performed trial designs of each of the major components using a combination of synthesis tools and hand-optimized netlists and an in-house standard-cell library. These areas were normalized to the size of a single full-adder in this standard-cell library. The area results are shown in Table 2.

These area estimates show that about 20% of the area in the multifunction interpolator is dedicated to the support of the five transcendental functions. The remaining area is required in any event for attribute interpolation. This area cost is much less than the total cost required to support a completely independent transcendental function evaluator.

Logic Block	Area (full-adders)
17b squarer	90
CS to radix-4 SD converter	45
lookup table ROM	1380
function overhead total	1515
2 optimized 17x24 mults	945
8 5x24 mults	2040
3 24b right-shifters	280
3 24b two's complementers	110
4 45b right-shifters	840
4 CSA tree	730
4 45b CPA	640
4 normalizers	930
planar interpolation total	6515
multifunction total	8030

Table 2. Area estimates for multifunction interpolator

5 Conclusion

We have presented the algorithms and implementation for a high-performance shared multifunction interpolator supporting both transcendental function approximation and planar attribute interpolation. By sharing a single functional unit for multiple interpolation applications, we were able to increase the unit's area-efficiency and utilization. These algorithms, in conjunction with a sufficiently pipelined implementation, increase total computational throughput while reducing total hardware cost.

6. Acknowledgements

The authors wish to thank Scott Kuusinen for his assistance with physical design and area analysis, and Lars-Erik Harsson for assisting with algorithm verification.

References

[1] J. Cao, B. Wei, and J. Cheng, "High-performance architectures for elementary function generation," in *Proc. 15th IEEE Symposium on Computer Arithmetic*, pp. 136–144, June 2001.

[2] T. C. Chen, "A binary multiplication scheme based on squaring," *IEEE Trans. Computers*, vol. C-20, pp. 678–680, 1971.

[3] D. DasSarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *Proc. 12th IEEE Symposium on Computer Arithmetic*, pp. 12–25, July 1995.

[4] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit Recurrent Algorithms and Implementations*, Kluwer Academic Publishers, 1994.

[5] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice, second edition in C*, Addison-Wesley, 1995.

[6] Microsoft Corporation, *Microsoft DirectX Technology Overview*, available at http://www.microsoft.com/windows/directx.

[7] J. M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser, 1997.

[8] J. M. Muller, "Partially rounded small-order approximations for accurate, hardware-oriented table-based methods," in *Proc. 16th IEEE Symposium on Computer Arithmetic*, pp. 114–121, June 2003.

[9] S. F. Oberman, "Floating point division and square root algorithms and implementations in the AMD-K7 microprocessor," in *Proc. 14th IEEE Symposium on Computer Arithmetic*, pp. 106–115, April 1999.

[10] OpenGL Architecture Review Board, *OpenGL 2.0 Specification*, available at http://www.opengl.org.

[11] J. A. Pineiro, S. F. Oberman, J. M. Muller, and J. D. Bruguera, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Trans. Computers*, vol. 54, no. 3, pp. 304–318, 2005.

[12] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. Computers*, vol. 48, no. 8, pp. 842–847, 1999.

[13] M. J. Schulte and E. E. Swartzlander, "Hardware designs for exactly rounded elementary functions," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 964–973, 1994.

[14] N. Takagi, "Powering by a table look-up and a multiplication with operand modification," *IEEE Trans. Computers*, vol. 47, no. 11, pp. 1216–1222, 1998.

[15] P. T. P. Tang, "Table lookup algorithms for elementary functions and their error analysis," in *Proc. 10th IEEE Symposium on Computer Arithmetic*, pp. 232–236, 1991.