NUMERICAL FUNCTION GENERATORS USING BILINEAR INTERPOLATION

Shinobu Nagayama[1], *Tsutomu Sasao*[2], *Jon T. Butler*[3]

[1] Department of Computer Engineering, Hiroshima City University, Japan
[2] Department of Computer Science and Electronics, Kyushu Institute of Technology, Japan
[3] Department of Electrical and Computer Engineering, Naval Postgraduate School, U.S.A.

ABSTRACT

Two-variable numerical functions are widely used in various applications, such as computer graphics and digital signal processing. Fast and compact hardware implementations are required. This paper introduces the bilinear interpolation method to produce fast and compact numerical function generators (NFGs) for two-variable functions. This paper also introduces a design method for symmetric two-variable functions. This method can reduce the memory size needed for symmetric functions by nearly half with small speed penalty. Experimental results show that the bilinear interpolation method can significantly reduce the memory size needed for two-variable functions, and the speed of NFGs based on the bilinear method is comparable to that of NFGs based on tangent plane approximation. For a complicated function, our NFG is faster and more compact than a circuit designed using a one-variable NFG.

1. INTRODUCTION

The availability of large quantities of logic in LSIs and the need for high-speed arithmetic function computation in modern data processing applications have created a unique research opportunity [7]. High-speed arithmetic function computation will almost certainly result in significant changes in the way engineers perform data processing tasks. For example, image recognition tasks will more likely be performed at the site of the image collection, such as onboard reconnaissance vehicles.

High-speed computation of one-variable arithmetic functions (e.g. $\sin(x)$ and $\log(x)$) has been extensively studied [2, 6, 8, 10–12]. However, significantly less work has been done on the high-speed implementation of multi-variable functions (e.g. $\sqrt{x^2+y^2+z^2}$ and $\arctan(x/y)$) [3, 4, 13]. The existing design approaches apply a different method for each function realized. As far as we know, no systematic design method for generic multi-variable functions has been presented.

A straightforward design method for an arbitrary multi-variable function is to use a single memory in which the address is a combination of values of variables and the content of that address is the corresponding value of function. This method is fast, but requires a 2^{mn}-word memory to implement an m-variable function with n bits for each variable. Thus, unlike one-variable functions, this method is impractical even for low-precision applications.

To produce a practical implementation, multi-variable functions can be designed using a combination of one-variable function generators, multipliers, and adders [3, 4]. This design can reduce the required memory size. However, depending on the function, it can produce a slow implementation because of its complex architecture. Also, complex architecture makes error analysis harder.

This paper proposes a systematic design method for two-variable functions. Since our design method is based on a piecewise polynomial approximation, hardware architecture is simple even for complex functions. However, polynomial approximation methods tend to require large memory size. For multi-variable functions, using higher-order polynomials is not always effective to reduce the memory size. This is because, for multi-variable polynomials, higher polynomial order requires many more polynomial coefficients. Also, higher-order polynomials produce slower NFGs. Thus, for polynomial approximation methods, reducing memory size with a small speed penalty is a key issue. To accomplish this, this paper introduces the bilinear interpolation method. This paper also introduces a design method and an architecture for symmetric two-variable functions. Error analysis for our NFGs is omitted because it is almost the same as [11]. It is guaranteed that the maximum error of our fixed-point NFGs is smaller than 2^{-m} (i.e., m-bit accuracy NFGs), where m is the number of fractional bits for the inputs and the output.

2. POLYNOMIAL APPROXIMATION USING BILINEAR INTERPOLATION

Bilinear interpolation is an extension of linear interpolation. It interpolates two-variable functions $f(X,Y)$ using four points. Let the four points be (X_1,Y_1), (X_1,Y_2), (X_2,Y_1), and (X_2,Y_2), and let $f_{11} = f(X_1,Y_1)$, $f_{12} = f(X_1,Y_2)$, $f_{21} = f(X_2,Y_1)$, and $f_{22} = f(X_2,Y_2)$. Then, the bilinear interpolation $g(X,Y)$ is given by:

$$g(X,Y) = \frac{f_{11}(X_2-X)(Y_2-Y)+f_{21}(X-X_1)(Y_2-Y)}{(X_2-X_1)(Y_2-Y_1)} + \frac{f_{12}(X_2-X)(Y-Y_1)+f_{22}(X-X_1)(Y-Y_1)}{(X_2-X_1)(Y_2-Y_1)}$$

By expanding and rearranging this, we obtain the following form: $g(X,Y) = C_{xy}XY + C_xX + C_yY + C_0$, where

$$C_{xy} = \frac{f_{11}-f_{21}-f_{12}+f_{22}}{(X_2-X_1)(Y_2-Y_1)},$$

$$C_x = \frac{-f_{11}Y_2+f_{21}Y_2+f_{12}Y_1-f_{22}Y_1}{(X_2-X_1)(Y_2-Y_1)},$$

$$C_y = \frac{-f_{11}X_2+f_{21}X_1+f_{12}X_2-f_{22}X_1}{(X_2-X_1)(Y_2-Y_1)}, \text{ and}$$

$$C_0 = \frac{f_{11}X_2Y_2 - f_{21}X_1Y_2 - f_{12}X_2Y_1 + f_{22}X_1Y_1}{(X_2 - X_1)(Y_2 - Y_1)}.$$

To approximate a given two-variable function using bilinear interpolation, we first partition a given domain of the function into segments. For each segment, we approximate the function using bilinear interpolation. In this case, the memory size and speed of an NFG are strongly dependent on the efficiency of the segmentation algorithm. Thus, effective segmentation algorithms are needed to achieve fast and compact NFGs. We use a *recursive planar segmentation algorithm* [9] (based on bilinear interpolation).

The algorithm begins by computing a bilinear interpolation using the four corner points of the given domain. This is an initial approximation. If that approximation error is larger than the given acceptable error, then the domain is partitioned into four equal-sized square segments. For each segment, a bilinear interpolation is computed using the four corner points of the segment. The same process is recursively repeated until all segments have an acceptable approximation error. Note that this algorithm creates a segment of size $w_i \times w_i$, where $w_i = 2^{h_i} \times 2^{-m_{in}}$, m_{in} is the number of fractional bits for X and Y, and h_i is an integer. That is, all the segmentation points P_i and Q_i are restricted to values such that the least significant h_i bits are 0 (i.e., $P_i = (\ldots\ p_{-j+1}\ p_{-j}\ 00\ \ldots\ 0)_2$, where $j = m_{in} - h_i$).

In this algorithm, to reduce the approximation error, the maximum positive error max_{fg} and the maximum negative error min_{fg} are equalized by a vertical shift of $g(X,Y)$ with correction value $v = (max_{fg} + min_{fg})/2$. Thus, the approximation error is $(max_{fg} - min_{fg})/2$, and the approximating polynomial is $g(X,Y) + v$.

For each segment $\{[B_x, E_x), [B_y, E_y)\}$, since $B_x \le X < E_x$ and $B_y \le Y < E_y$ hold, we can offset X and Y by B_x and B_y to compute the approximating polynomial $g(X,Y) + v$. By using the offset inputs $(X - B_x)$ and $(Y - B_y)$ instead of X and Y, we reduce the size of multipliers needed to compute $g(X,Y) + v$. By substituting $X - B_x + B_x$ and $Y - B_y + B_y$ for X and Y respectively, we transform $g(X,Y) + v$ as follows:

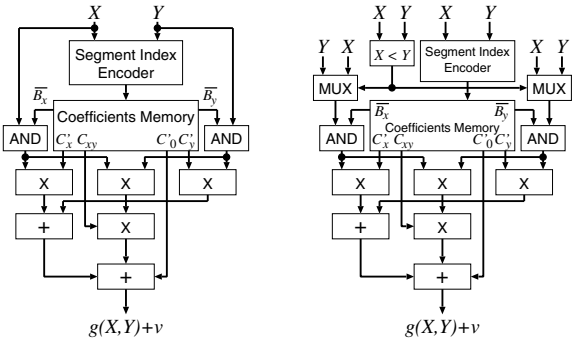
$$\begin{aligned} g(X,Y) + v &= C_{xy}(X - B_x + B_x)(Y - B_y + B_y) + C_x(X - B_x + B_x) \\ &\quad + C_y(Y - B_y + B_y) + C_0 + v \\ &= C_{xy}(X - B_x)(Y - B_y) + (C_x + C_{xy}B_y)(X - B_x) \\ &\quad + (C_y + C_{xy}B_x)(Y - B_y) + C_{xy}B_xB_y + C_xB_x + C_yB_y + C_0 + v \\ &= C_{xy}(X - B_x)(Y - B_y) + C'_x(X - B_x) + C'_y(Y - B_y) + C'_0, \quad (1) \end{aligned}$$

where $C'_x = C_x + C_{xy}B_y$, $C'_y = C_y + C_{xy}B_x$, and $C'_0 = C_{xy}B_xB_y + C_xB_x + C_yB_y + C_0 + v$.

3. ARCHITECTURE BASED ON BILINEAR INTERPOLATION

Fig. 1 shows architectures for two-variable NFGs realizing (1). The **Segment Index Encoder** converts values of X and Y into a segment number. This, in turn, is applied as the address input of the **Coefficients Memory**. The coefficients are applied to adders and multipliers to form the polynomial value $g(X,Y) + v$. Note the use of bitwise ANDs in Fig. 1 to compute $X - B_x$ and $Y - B_y$. In recursive segmentation, we can realize $X - B_x$ and $Y - B_y$ using AND gates driven on one side by $\overline{B_x}$ and $\overline{B_y}$, respectively [8].

The segment index encoder realizes the segment index function: $\{0,1\}^n \times \{0,1\}^n \to \{0,1,\ldots,k-1\}$ shown in

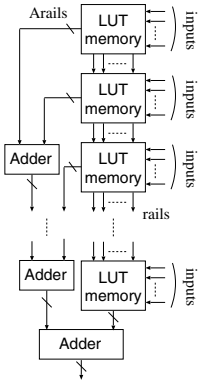


(a) For general functions. (b) For symmetric functions.

Fig. 1. Architectures for two-variable NFGs based on bilinear interpolation.

Segments	Index
$X_b \le X < P_0$ $Y_b \le Y < Q_0$	0
$X_b \le X < P_0$ $Q_0 \le X < Q_1$	1
$\vdots$	$\vdots$
$P_{r-1} \le X < Y_e$ $Q_{r-1} \le X < Y_e$	$k-1$

(a) Segment index function.



(b) LUT cascade and adders [8].

Fig. 2. Segment index encoder.

Fig. 2(a), where X and Y have n bits, and k denotes the number of segments. We realize this function with the architecture shown in Fig. 2(b). We use an edge-valued BDD (EVBDD) [5] to design this architecture. In this architecture, the interconnecting lines between adjacent LUT memories determine the position in the EVBDD (labeled *rails*), and the outputs from each LUT memory to the adders tally the function value (labeled *Arails*). Consider the design of the LUT cascade and adders in Fig. 2(b), given the segmentation produced by the algorithm in [9].

We begin by representing the segment index function using a multi-terminal BDD (MTBDD) [1]. Then, we convert the MTBDD into an EVBDD. By decomposing the EVBDD, we obtain the architecture in Fig. 2(b). For more detail on this architecture, see [8].

In our architecture, the coefficients memory and the LUT memories of the segment index encoder are implemented by RAMs. Thus, by changing the data for the coefficients memory and the LUT memories, a wide class of two-variable functions can be realized by a single architecture.

4. DESIGN METHOD FOR SYMMETRIC FUNCTIONS

Definition 1 *A two-variable function $f(X,Y)$ is **symmetric** if $f(X,Y) = f(Y,X)$.*

Table 1. Number of segments needed for three design methods.

Function $f(X,Y)$	Domain X	Domain Y	X and Y: 12-bit accuracy (Approximation error: 2^{-14}) Tangent	Bilinear	Sym.
$\sin(\pi X)\sqrt{Y}$	$[0,1)$	$(0,1)$	38,773	29,875	N/A
$\sin(\pi XY)$	$[0,1)$	$[0,1)$	26,122	8,389	4,232
X^4Y^5	$[0,1)$	$[0,1)$	8,179	3,592	N/A
$1/\sqrt{X^2+Y^2}$	$(0,1)$	$(0,1)$	173,552	103,046	51,687
$XY/\sqrt{X^2+Y^2}$	$(0,1)$	$(0,1)$	6,523	4,114	2,104
WaveRings	$[0,\pi]$	$[0,\pi]$	28,377	16,278	8,202
Sombrero	$(0,8)$	$(0,8)$	32,371	18,664	9,398
Gaussian	$(0,1)$	$(0,1)$	141,113	86,633	N/A
$\sqrt{X^2+Y^2}$	$(0,1)$	$(0,1)$	6,160	4,093	2,083
$\sqrt[3]{X^3+Y^3}$	$(0,1)$	$(0,1)$	6,790	3,955	2,027

Sym.: Two symmetric segments are counted as one segment.

Symmetric functions are commonly found in practical applications of NFGs. For example, $\sqrt{X^2+Y^2}$, which is used in converting from rectangular to polar coordinates, is symmetric. This section presents a design method and an architecture taking advantage of the function's symmetry.

Definition 2 *A segmentation is* ***symmetric*** *if for every segment* $\{[B_{x1},E_{x1}),[B_{y1},E_{y1})\}$ *such that* $B_{x1} \neq B_{y1}$ *or* $E_{x1} \neq E_{y1}$*, there is another segment* $\{[B_{x2},E_{x2}),[B_{y2},E_{y2})\}$ *such that* $B_{x1} = B_{y2}$*,* $E_{x1} = E_{y2}$*,* $B_{y1} = B_{x2}$*, and* $E_{y1} = E_{x2}$*.* ***Symmetric segments*** *are a pair of such segments.*

Lemma 1 *Let* $f(X,Y)$ *be a symmetric function, and let* $g_1(X,Y)$ *and* $g_2(X,Y)$ *be bilinear interpolations of* $f(X,Y)$ *for symmetric segments. Then,* $g_1(X,Y) = g_2(Y,X)$*.*

Theorem 1 *The segmentation of a symmetric function produced by the recursive planar segmentation algorithm is symmetric.*

From Lemma 1 and Theorem 1, we can use only one bilinear interpolation to approximate the given symmetric function in symmetric segments. By assigning the same segment index to symmetric segments, we can reduce the size of coefficients memory by nearly half.

Fig. 1(b) shows an architecture for symmetric functions. Here, the coefficients memory stores only data for segments such that $X \leq Y$. For other segments, approximated values are computed using Lemma 1. Since the comparator and multiplexers operate in parallel with the segment index encoder, there is no speed penalty due to these additional circuits.

5. EXPERIMENTAL RESULTS

5.1. Number of Segments and Memory Sizes

Table 1 compares the number of segments needed for the bilinear interpolation method with that for the tangent plane approximation[1] [9] for various functions. For those functions that are symmetric, Table 1 shows the number of symmetric segments. In this table, *WaveRings*, *Sombrero*, and *Gaussian* are:

$$\textit{WaveRings} = \frac{\cos\left(\sqrt{X^2+Y^2}\right)}{\sqrt{X^2+Y^2+0.25}}$$

[1]In the tangent plane approximation, the function is realized using $g(X,Y) = C_xX + C_yY + C_0$.

Table 2. Total memory sizes needed for 12-bit accuracy NFGs based on three design methods.

$f(X,Y)$	Tangent	Bilinear	Sym.
$\sin(\pi X)\sqrt{Y}$	2,030,356	2,167,788*	N/A
$\sin(\pi XY)$	1,313,684	701,311	356,164
X^4Y^5	516,230	266,644	N/A
$1/\sqrt{X^2+Y^2}$	13,054,030	9,412,758	4,698,276
$XY/\sqrt{X^2+Y^2}$	369,189	293,330	153,176
WaveRings	1,886,924	1,559,560	797,279
Sombrero	2,051,615	1,487,068	757,238
Gaussian	11,345,482	8,285,179	N/A
$\sqrt{X^2+Y^2}$	316,128	287,868	153,291
$\sqrt[3]{X^3+Y^3}$	405,576	294,328	154,309

*Bilinear is worse than tangent for only 1 case.

$$\textit{Sombrero} = \frac{\sin\left(\sqrt{X^2+Y^2}\right)}{\sqrt{X^2+Y^2}}, \quad \textit{Gaussian} = \frac{1}{Y\sqrt{2\pi}}e^{-\frac{X^2}{2Y^2}}$$

Table 1 shows that, for all functions, the bilinear interpolation method requires fewer segments than the tangent plane approximation. And, the number of symmetric segments is much smaller. Thus, the bilinear interpolation method and the symmetric technique significantly reduce the number of words in the coefficients memory. For $\sin(\pi X)\sqrt{Y}$, the bilinear interpolation method is also superior to the tangent plane method. However, the number of segments needed in the bilinear interpolation method is only slightly smaller.

Table 2 compares the total memory sizes needed for NFGs based on the three design methods. Note that our NFGs have two kinds of memories: coefficients memory and LUT memory; the memory size shown is the sum of the two.

Table 2 shows that, for all functions except for $\sin(\pi X)\sqrt{Y}$, the bilinear method uses less memory than the tangent plane approximation, even though the bilinear interpolation requires more polynomial coefficients. That is, for many functions, the reduction in the number of segments due to bilinear interpolation is sufficient to compensate for the increase of polynomial coefficients. Especially for symmetric functions, using the symmetric technique shown in Section 4 reduces further the memory size.

5.2. FPGA Implementation Results

We implemented 12-bit accuracy NFGs based on the three design methods using the Altera Stratix III FPGA. Since the FPGA has *adaptive look-up tables (ALUTs)* that can realize fast adders, synchronous memory blocks, and dedicated DSPs (multipliers), our NFGs are efficiently implemented by those hardware resources in the FPGA. Table 3 compares the FPGA implementation results for 12-bit accuracy. In this table, the columns "Delay" show the total delay time of each NFG from the input to the output, in nanoseconds.

The NFGs based on tangent plane approximation are faster because they require fewer multipliers and fewer polynomial coefficients. However, for the $1/\sqrt{X^2+Y^2}$ and *Gaussian* functions, the memory needed for NFGs based on tangent plane approximation is so large that they could not be implemented in the FPGA. On the other hand, NFGs based on the bilinear method require less memory, and their speed is comparable to the speed of the NFGs based on tangent plane approximation. Since the symmetric technique significantly reduces the memory size, it is easier to implement with an FPGA. Table 3 shows that the symmetric

Table 3. FPGA implementation of 12-bit accuracy NFGs based on three design methods.

FPGA device: Altera Stratix III (EP3SL340F1517C4)						Logic synthesis tool: Synplify Pro Ver. 8.8									
Function $f(X,Y)$	Tangent plane					Bilinear interpolation					Symmetric method				
	#ALUTs	#DSPs	Freq. [MHz]	#stages	Delay [ns]	#ALUTs	#DSPs	Freq. [MHz]	#stages	Delay [ns]	#ALUTs	#DSPs	Freq. [MHz]	#stages	Delay [ns]
$\sin(\pi X)\sqrt{Y}$	270	4	243	12	49	394	6	262	15	57	N/A	N/A	N/A	N/A	N/A
$\sin(\pi XY)$	131	4	285	7	25	274	14	245	9	37	351	14	245	10	41
X^4Y^5	235	4	243	10	41	286	14	251	10	40	N/A	N/A	N/A	N/A	N/A
$1/\sqrt{X^2+Y^2}$	–	–	–	17	–	592	7	249	18	72	543	7	252	16	64
$XY/\sqrt{X^2+Y^2}$	182	4	285	10	35	206	7	262	10	38	293	7	262	11	42
WaveRings	341	4	243	12	49	474	14	262	13	50	489	14	256	13	51
Sombrero	221	4	243	10	41	281	14	245	11	45	296	14	245	11	45
Gaussian	–	–	–	17	–	639	14	252	17	67	N/A	N/A	N/A	N/A	N/A
$\sqrt{X^2+Y^2}$	147	4	285	8	28	195	7	262	10	38	256	7	253	11	43
$\sqrt[3]{X^3+Y^3}$	193	4	285	10	35	201	13	272	10	37	242	12	244	10	41

–: NFGs cannot be mapped into the FPGA due to insufficient memory blocks.
#ALUTs: Number of ALUTs. #DSPs: Number of 9-bit × 9-bit DSP units. Freq. : Operating frequency. #stages : Number of pipeline stages.

Table 4. FPGA implementation of various NFGs for $XY/\sqrt{X^2+Y^2}$.

FPGA device:	Altera Stratix III (EP3SL340F1517C4)
Logic synthesis tool:	Synplify Pro Ver. 8.8

NFGs	Memory [bits]	#ALUTs	#DSPs	Freq. [MHz]	#stages	Delay [nsec.]
	12-bit accuracy					
One-variable	269,136	273	16	195	14	72
Tangent	369,189	182	4	285	10	35
Bilinear	293,330	206	7	262	10	38
Symmetric	153,176	293	7	262	11	42

technique has some speed penalty. However, it is reasonable. To show this, we designed $XY/\sqrt{X^2+Y^2}$ using one-variable NFG for $1/\sqrt{X}$, two squaring circuits, an adder, and two multipliers. The one-variable NFG was realized by the method shown in [8], which is based on linear approximation and non-uniform segment lengths. Table 4 compares the results with our NFGs.

Our NFGs require fewer ALUTs and DSPs than the one-variable implementation, and have much shorter delay. Especially, the NFG designed by the symmetric method requires less memory and shorter delay than the one-variable NFG. This shows that the speed penalty of our methods is small.

6. CONCLUDING REMARKS

We have proposed a design method and a programmable architecture for two-variable numerical function generators using the bilinear interpolation. To realize a two-variable function in hardware, we partition the given domain of the function into segments, and approximate the given function using the bilinear interpolation in each segment. In this paper, we also presented a design method and an architecture for two-variable symmetric functions. Experimental results show that the proposed method can significantly reduce the memory size needed for two-variable functions with small speed penalty.

7. ACKNOWLEDGMENTS

This research is partly supported by the Grant in Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS), Knowledge Cluster Initiative (the second stage) of MEXT (Ministry of Education, Culture, Sports, Science, and Technology) a contract with the National Security Agency, and the MEXT Grant-in-Aid for Young Scientists (B), 20700051, 2008.

8. REFERENCES

[1] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," *Proc. of 30th ACM/IEEE Design Automation Conference*, pp. 54–60, June 1993.

[2] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," *16th IEEE Inter. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP'05)*, pp. 328–333, 2005.

[3] R. Gutierrez and J. Valls, "Implementation on FPGA of a LUT based $atan(y/x)$ operator suitable for synchronization algorithms," *Proc. of the IEEE Conf. on Field Programmable Logic and Applications*, pp. 472–475, Aug. 2007.

[4] Z. Huang and M. D. Ercegovac, "FPGA implementation of pipelined on-line scheme for 3-D vector normalization," *Proc. of the 9th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'01)*, pp. 61–70, Apr. 2001.

[5] Y-T. Lai and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," *Proc. of 29th ACM/IEEE Design Automation Conference*, pp. 608–613, 1992.

[6] D.-U. Lee, W. Luk, J. Villasenor, and P. Y. K. Cheung, "Hierarchical segmentation schemes for function evaluation," *Proc. of the IEEE Conf. on Field-Programmable Technology*, Tokyo, Japan, pp. 92–99, Dec. 2003.

[7] J.-M. Muller, *Elementary Function: Algorithms and Implementation*, Birkhauser Boston, Inc., New York, NY, second edition, 2006.

[8] S. Nagayama, T. Sasao, and J. T. Butler, "Design method for numerical function generators using recursive segmentation and EVBDDs," *IEICE Trans. Fundamentals*, Vol. E90-A, No. 12, pp. 2752–2761, Dec. 2007.

[9] S. Nagayama, J. T. Butler, and T. Sasao, "Programmable numerical function generators for two-variable functions," *EUROMICRO Conference on Digital System Design (DSD-2008)*, Sep. 2008 (accepted).

[10] J.-A. Piñeiro, S. F. Oberman, J.-M. Muller, and J. D. Bruguera, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Trans. on Comp.*, Vol. 54, No. 3, pp. 304–318, Mar. 2005.

[11] T. Sasao, S. Nagayama, and J. T. Butler, "Numerical function generators using LUT cascades," *IEEE Transactions on Computers*, Vol. 56, No. 6, pp. 826–838, Jun. 2007.

[12] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. on Comp.*, Vol. 48, No. 8, pp. 842–847, Aug. 1999.

[13] N. Takagi and S. Kuwahara, "A VLSI algorithm for computing the Euclidean norm of a 3D vector," *IEEE Transactions on Computers*, Vol. 49, No. 10, pp. 1074–1082, Oct. 2000.