A Special-Purpose Compiler for Look-Up Table and Code Generation for Function Evaluation

Yuanrui Zhang[1], Lanping Deng[2], Praveen Yedlapalli[1],
Sai Prashanth Muralidhara[1], Hui Zhao[1], Mahmut Kandemir[1],
Chaitali Chakrabarti[2], Nikos Pitsianis[3,4], Xiaobai Sun[4]
[1]Department of Computer Science and Engineering, Pennsylvania State University, USA
[2]School of Electrical, Computer and Engineering, Arizona State University, USA
[3]Department of Electrical and Computer Engineering, Aristotle University, Greece
[4]Department of Computer Science, Duke University, USA

***Abstract*—Elementary functions are extensively used in computer graphics, signal and image processing, and communication systems. This paper presents a special-purpose compiler that automatically generates customized look-up tables and implementations for elementary functions under user given constraints. The generated implementations include a C/C++ code that can be used directly by applications running on multicores, as well as a MATLAB-like code that can be translated directly to a hardware module on FPGA platforms. The experimental results show that our solutions for function evaluation bring significant performance improvements to applications on multicores as well as significant resource savings to designs on FPGAs.**

I. Introduction

Elementary functions such as Gaussian, Bessel, B-spline and trigonometric functions, as well as their combinations are extensively used in computer graphics, digital signal and image processing (DSIP), and communication systems. The evaluation of these functions is an essential and indispensable component in the embedded systems designed for those applications, and directly dictates the system performance, power, resource utilization, and accuracy.

The most popular method of evaluating elementary functions is to use look-up table (LUT) based polynomial approximation [5]-[10], especially the Talyor expansion [7]. Typically, the input data range is split into equally-spaced or unequally-spaced intervals indicated by uniform or non-uniform sampling points, and the coefficients for different polynomial orders at each sample point are calculated and stored in a table. When evaluating a sufficiently smooth function, the Talyor expansion together with table lookup is applied to obtain the approximated result. Different accuracies can be achieved by adjusting the sampling density (LUT length), the approximation degree (LUT width) and the bit-width in the data representation of LUT entries (LUT depth).

A variety of software and hardware implementations for this approach have been developed in the embedded systems with real-time applications and on-line data processing. While the prevalent platform for hardware implementations is field programmable gate arrays (FPGAs), which provide large flexibility for rapid DSIP system prototyping, the popular platform to run software implementations is multicores, which provide rich parallelism for embedded computing to deal with large input data sizes. Although the LUT-based implementations are fairly common now for function evaluation, the LUTs are usually calculated and generated manually with respect to certain accuracy requirement and resource constraints.

To substitute this manual process, we have developed a special-purpose compiler to assist the user in generating customized LUTs automatically together with software and hardware implementations for elementary functions, targeting both multicore and FPGA platforms. For FPGA-based hardware design, the compiler tries to find the "best" solution which satisfies the accuracy with minimal area and memory, and produces a resource estimation report to help user identify the problem and redistribute the resources among different modules if necessary. Distinguished from previous efforts on function evaluation, our tool can handle individual elementary functions, like $\sin(3\pi x)$, as well as arbitrary arithmetic expressions composed of different elementary functions, such as $10 + \cos(\sin(\pi + x)) \times e^{-2x}$, through function composition. The class of elementary function supported in our work is broad, including Gaussian, exponential, Bessel, central B-splines, certain cylinder and trigonometric functions, with both fixed-point and floating-point (single and double precision) data representations. The experimental results demonstrate that our tool is efficient in finding optimal solutions for function evaluation under certain constraints. In particular, the results on multicores show that our solutions improve the application performance significantly compared to the standard library implementations, while the results on FPGAs show that our solutions save hardware resources significantly in comparison with direct implementation methods.

The rest of this paper is organized as follows. Section II introduces the background of LUT-based Taylor polynomial approximation and its error analysis. Section III describes the related work of function evaluation. Section IV presents an overview of our compiler, while Section V and Section VI give the detailed descriptions of its front end and back end, respectively. Finally, experimental results are discussed in Section VII, and conclusion and future work are given in Section VIII.

978-3-9810801-6-2/DATE10 © 2010 EDAA

II. Background

The LUT-based Talyor polynomial approximation is a well-known technique for function evaluation. A truncated Taylor expansion for a sufficiently smooth function $f(x)$ is as follows:

$$T_n(x) = f(c) + \sum_{k=1}^{d} \frac{f^{(k)}(c)}{k!}(x-c)^k, \quad (1)$$

where $f(c)$ and $f^{[k]}(c)$ are the values of the function and its derivatives at a reference point c, x is the evaluation point, and d is the degree of the Taylor polynomial. It can be implemented efficiently using the Horner's rule [25].

Typically, m equally spaced reference points or sample points from a pre-specified range of input x are chosen. The values of $f(c)$ and $f^{[k]}(c)$ at each sample point are calculated and stored beforehand in a table, with first column indicating sample points themselves. If k ranges from 0 to d, the resulting LUT contains m entries, each having a row of $d+2$ columns, and we say that the LUT is of length m and width $(d+2)$. The space between two consecutive points is called the *sampling interval*. To evaluate $f(x)$, one needs to find the nearest reference point c first, and then substitute the coefficients in Eq. 1 with certain values by looking up into the table.

We consider two types of errors of the Taylor polynomial approximation in this work, the *truncation error* brought by the finite expansion order in the algorithm, and the *round-off error* due to the finite data precision supported by the architecture. In addition, the round-off error can be further broken down into the error caused by coefficients stored in the LUT and the error introduced during the polynomial evaluation. To find a solution satisfying the accuracy requirement with consideration of both analytical truncation error and architectural round-off error, we conduct a search in a three-dimensional design space, where the total error can be adjusted by changing the sampling interval (LUT length), the approximation degree (LUT width), and the bit-width in the data representation of LUT entries (LUT depth).

III. Related Work

There has been a fair amount of prior work on LUT-based polynomial approximation. The work in [6] applies a second-order polynomial to power function x^p, and is limited to single-precision floating-point data. The work in [3] supports a larger set of functions with small-order polynomial, including sine, cosine, exponential, Gamma and Bessel functions, but for fixed-point data. In particular, the Taylor expansion polynomial is employed in the following literature, [1], [2], [5], [7], [8], [9], and [10]. Takagi [5] presents an optimization method for power function x^p by utilizing one multiplier, while Cody et al [7] extend the work to exponential, central Bessel and Gamma functions with consideration of truncation error. Tang et al in [1] and [8] take into account both truncation and round-off errors in exponential, 2^x, $\log x$ and sine functions for floating-point data. Cao [2], Sobti [9] and Bui [10], on the other hand, concentrate on techniques for reducing LUT size to save hardware resources. Apart from these, a most recent effort described in [4], presents a systematic way of using linear piecewise approximation with non-uniform sample points to support trigonometric, logarithmic, sigmoidal and square root functions for fixed-point data. However, all these efforts target individual elementary functions.

As opposed to previous work, we support combined (or merged) LUT generation as well as code generation through function composition, for arbitrary numerical expressions composed of different elementary functions. To the best of our knowledge, our work is the first in this direction.

IV. An Overview of Our Special-Purpose Compiler

While a conventional compiler accepts, say C code, and outputs binary file for a target architecture, our special-purpose (SP) compiler is designed to generate LUTs and implementations (codes) for arithmetic expressions based on elementary functions. The generated C/C++ code can be used directly by applications running on multicores, and the MATLAB-like code can be translated directly to a hardware module on FPGA-based platforms.

Despite this special functionality, the SP compiler is structured the same way as a conventional compiler, including two parts, the *front end* and the *back end* [16], as depicted in Figure 1. The front end parses a legal source code (also MATLAB-like) consisting of arithmetic expressions of elementary functions and converts it into an intermediate representation (IR). The back end then performs a series of optimizations and error checking on IR, generates LUT tables and codes, and also outputs an FPGA resource estimation report for the user. Among all these, the important aspect of the flow resides in the *look-up table generation phase*, which employs a systematic search strategy to find a solution that meets user-specified accuracy and resource constraints. If no solution is found, a near-optimal (best-effort) solution satisfying the accuracy is returned with minimal additional area and memory.

We also developed a *grammar generator* to give the user flexibility of defining elementary function formats in the input code. It generates both lexical and grammatical specifications according to user's requirements, which will then be fed to ANTLR [17], a commercial parser generator, to create the SP compiler front end automatically. In other words, the front end is modular and adaptable to user's demands.

V. SP Compiler Front End

The SP compiler front end accepts a MATLAB-like source code as shown in Figure 2, and translates it into an IR, an acyclic graph (or a DAG). Currently, eight elementary functions are supported in the input code, including sine $\sin(x)$, cosine $\cos(x)$, tangent $\tan(x)$, exponential $a \cdot e^{-b \cdot x}$, Gaussian $a \cdot e^{-b \cdot x^2}$, central B-spline $\beta^n(x)$, Sinc $sinc^n(x)$ and Bessel function of the first kind $J_0(x)$. The example formats of these functions are listed in Table I. In the parameter list, a and b stand for the coefficients of some particular functions, n denotes the order, and $[min, max]$ represents the dynamic range of input data. In case the user does not provide the dynamic range, we calculate it based on the architectural (finite

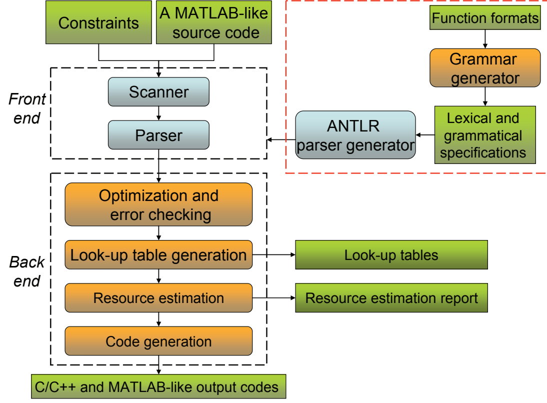

Fig. 1. High-level overview of SP compiler and its interaction with the grammar generator.

TABLE I
EXAMPLES OF INPUT FORMATS FOR EIGHT SUPPORTED ELEMENTARY FUNCTIONS.

Function Name	Function Parameters
Fantom_exp	(x, a, b, min, max)
Fantom_gaussian	(x, a, b, min, max)
Fantom_bessel	(x, min, max)
Fantom_sinc	(x, n, min, max)
Fantom_bspline	(x, n)
Fantom_sin	(x, min, max)
Fantom_cos	(x, min, max)
Fantom_tan	(x, min, max)

representation) limitations. The final dynamic range applied to LUTs is the overlapped region of all ranges. Note that in an elementary function $f(x)$, x could be a constant, a free variable, an elementary function $h(x)$, or an arbitrary arithmetic expression, such as $x+3$ or $x+3*h(x)$. In this way, we support recursive and nested function calls. Nevertheless, we do not support multiple free variables in the current version (it is in our future research agenda). For instance, if the user inputs $y = z$ instead of $y = x$ in the sample code given by Figure 2, yielding two free variables x (from the parameter) and z, the parser will report an error.

```
function  out = filter_test (x)
y = x;                    // y=z will make input illegal
n = 10;
pi = 3.1415;
t = n * 2;
t1 = fantom_gaussian (x*x, 1, 1, 0, 15);
t2 = fantom_cos (1, 0, 2*pi) + fantom_gaussian (t1, 1, 7, 0, 10) *
fantom_bessel(10, 1, 8) / fantom_gaussian ( fantom_sin (t, 0,
pi/2), 1, 1, 0.9, 10);
out = fantom_bspline (t2, n) / fantom_exp (x+2, 2, 6, 0.3, 9);
```

Fig. 2. A sample input MATLAB-like code.

Once the code is confirmed to be legal, an expression DAG is created. In the DAG, each node represents either a constant, a variable, an arithmetic operator (or a sign), or an elementary function. Different types of nodes have different numbers of children, e.g., a function node has multiple children (for parameters), whereas an operator node has one or two children. Also, a node may have more than one ”parent”, indicating a common subexpression. We keep one copy for each variable even it appears at multiple places in the DAG, and eliminate the redundant expressions, for example, variable y in expression $y = x$ in the sample code will not appear in the created DAG, since y is not used by output out.

Grammar Generator

The grammar generator is a stand-alone tool, designed to facilitate the user in defining the input formats of elementary functions. It accepts a file describing the name and parameters of each function, very similar to Table I, and outputs a file containing both lexical (regular expressions) and grammatical specifications, which will be given to ANTLR to generate scanner and parser automatically. This gives the user great flexibility to adjust parameters of a function, as well as add new elementary functions to the tool.

VI. SP COMPILER BACK END

The SP compiler back end presents the innovations of our work. As depicted in Figure 1, it has four phases: *optimization and error checking*, *look-up table generation*, *resource estimation*, and *code generation*. The rest of this section discusses these phases in detail.

A. Optimization and Error Checking

Once an expression DAG is created, the optimization and error checking will be performed first. The former includes constant folding and constant propagation [16]. Constant folding is further divided into two categories, algebraic transformations and elementary function simplification. While algebraic transformations evaluate arithmetic operations with two constant operands or one constant operand having special values like 0, 1, and -1, elementary function simplification evaluates functions $f(x)$ where x is constant, e.g., fantom_bessel(10, 1, 8) in the sample code, by invoking the tool developed in [18]. Constant propagation includes sign propagation. Take $x = -(50 - 30)$ for example. After constant propagation, instead of an operator node denoting the negative sign, a constant node with value -20 is obtained as child of the variable node x. The error checking examines division by zero, square root of negative numbers, and function parameter errors. Constant folding, constant propagation and error checking are interleaved iteratively until no more changes or errors occur, to achieve simplification, reduction, and verification.

B. Look-up Table Generation

The next phase is the LUT generation. There are two steps in this stage. The first step is to generate individual LUT for each

elementary function that appears in the original code. To do this, the total area and memory resources are distributed among multiple functions first, with consideration of variations in the requirements of different functions. Then, a three-dimensional search in the design space of data precision, sampling interval and Taylor approximation degree is conducted using the tool developed in [18], to find the "best" design satisfying the accuracy with minimal area and memory resources for each function. The tool takes into account both truncation error and round-off error, and estimates the accuracy against standard MATLAB implementations.

The second step is to generate merged LUTs through function composition, which can save hardware resources and bring performance improvements for function evaluation. It carries out on the expression DAG in a bottom-up order via post-order depth-first graph traversal. Generally speaking, the composition procedure tries to merge LUTs of multiple functions by manipulating the corresponding coefficients based on some rules. More specifically, the composition action takes place in one of the two following situations, 1) at an operator node, if at least one of its children is a function node; 2) at an elementary function node.

In the first case, we have the form $f(x)$ OP $t(x)$ or $t(x)$ OP $f(x)$, where OP represents an operator, "+", "-", "*" or "/", and $t(x)$ denotes a constant, a variable expression, or a function.

- If $t(x)$ is a constant, the new LUT is generated by keeping the original sample points as the first column, while calculating the new coefficients in other columns according to the arithmetic operation OP on the original coefficients and the constant.
- If $t(x)$ is a variable expression, a temporary LUT for $t(x)$ is created first by using the same sample points and polynomial order as in the LUT for $f(x)$, and then, the new LUT table is generated by computing the coefficients based on the arithmetic operation on the corresponding coefficients in the two tables. For example, in the case that OP is multiplication, the coefficient of order n in the new table at a sample point c is calculated by Eq. 2, where $\frac{f^{(k)}(c)}{(k)!}$ and $\frac{t^{(k)}(c)}{k!}$ are coefficients in the original LUTs, respectively.
- If $t(x)$ is a function, the new coefficients are computed directly in a similar way as the previous case without temporary LUT creation. Note that, under this circumstance, the function $f(x)$ or $t(x)$ can be either an elementary function listed in Table I, or an already composed function. The LUT size mismatch may happen, where the sample points and/or polynomial orders of the two tables are not the same. Our solution is to regenerate the LUT for each function with matching size in the original subgraph rooted by this node first, through sampling interval tuning or zero appending, and then perform composition again on this subgraph in a bottom-up order until reaching the node.

In the second case, we have the form $f(t(x))$, with $t(x)$ being a variable expression. The situation $t(x)$ being a constant is already eliminated in the optimization phase, and no composition is performed for $t(x)$ being a function. Even if $t(x)$ is a variable expression, the combined LUT generation is still quite complicated. Currently, we only handle the scenario where $t(x)$ is a first order polynomial of form $a \cdot x + b$ (a and b are constants), and the coefficient of order n in the new table at a sample point c is calculated by Eq. 3.

$$n_c = \sum_{k=0}^{n} \frac{f^{(n-k)}(c)}{(n-k)!} \cdot \frac{t^{(k)}(c)}{k!}, \tag{2}$$

$$n_c = \frac{(f^{(n)}(a \cdot c+b) \cdot a^n)}{n!} \tag{3}$$

The composition may return one of the four possible status at a node, 1) successfully composed, 2) constant value obtained, 3) cannot be composed, and 4) accuracy not satisfied. Status 1) indicates that a composed function with its combined LUT is successfully generated without violating any accuracy or resource constraint. Status 2) occurs when $t(x)$ in $f(t(x))$ at a function node is evaluated to be a constant, e.g., $t(x) = (x+1-x)$, the detection of which is impossible during the optimization phase. Status 3) happens when composition cannot be accomplished, for example, $f(t(x))$ and $f(x)/t(x)$ where $f(x)$ and $t(x)$ are elementary functions. In this situation, we simply keep two separate tables for individual functions. Status 4) is returned if the generated LUT at a node does not meet the accuracy requirement under assigned resource constraints. In this case, we recover the original subgraph with all the descendants of this node, regenerate the LUT for each function in the subgraph with higher accuracy, and redo the composition in a bottom-up order until this node. Because of the assigned resource constraints, the accuracy may not be satisfied still. We make three trials before giving it up.

It is worth pointing out that Status 3) has transitivity property. If a node cannot be composed, neither can its parent. Using this characteristic, we check the status of children before applying function composition to any node. Like individual elementary functions, the accuracy evaluation for composite functions is also conducted via the tool developed in [18].

C. Code Generation

The code generation phase is relatively simple. Figure 3 gives a sample MATLAB-like code output by our compiler, which can be fed to the MATLAB-to-HDL tool developed in [19] to obtain the hardware design for function evaluation automatically. On the other side, the generated C/C++ code can be executed directly by applications on multicores.

In the sample code, 'lut_function' stands for either an elementary function or a composed function that is implemented using Horner's rule in both hardware design and software code. The numbers in the parameter list of each function represent the order n, the sampling interval, the data bits, and the address bits, respectively, for hardware purposes. Different function instantiations have different look-up tables, even they are of the same type, e.g., the two exponential functions in the sample code have separate LUTs. Besides basic operations, we

```
function out = kernel (x)
t0 = x * x;
t1 = lut_function (t0, 'exponential_1', 1, 64, 32, 12);
t2 = lut_function (t1, 'gaussian_2', 1, 2048, 32, 12);
t3 = t2 * (-0.499268);
t4 = lut_function (t3, 'exponential_3', 10, 1, 32, 4);
t5 = t4 * x;
t6 = float2fixed (t5);
t7 = lut_function (t6, 'composed_4', 1, 1024, 32, 11);
t8 = t5 / t7;
out = t8;
```

Fig. 3. A sample output MATLAB-like code.

support some other functions specific for format converting in hardware designs, including float2fixed(x), fixed2float(x) and float2float(x).

D. Resource Estimation

In addition to the LUT and code generation, our SP compiler back end also outputs FPGA area and memory resource estimations. The area is measured by two types of resources, *Slice* and *Multiplier*, whereas the memory is measured in terms of block random access memories (*BRAMs*). The estimation at each node employs an FPGA modeling tool presented in [14], and the total resource consumption is the sum of resource consumption of all nodes in the DAG. The report aims at helping the user adjust the high-level resource assignments among different modules in a hardware design.

VII. Experimental Evaluation

Our SP compiler is implemented in C, and interacts with the tool [18] implemented in MATLAB (for the LUT generation of individual elementary functions) through MATLAB Engine [23]. We have chosen the following kernel functions that are used in many signal and image processing applications for the experiments. These include the *Gaussian filter* $\sqrt{a/\pi}e^{-a \cdot x^2}$, *Gabor filter* [21] (Eq. 4 and Eq. 5), *Gammatone filter* [22] (Eq. 6) and the arithmetic combinations of Gabor and Gammatone filters, including Gabor+Gammatone and Gabor*Gammatone. Since our current tool supports only one free variable in the input code, all the variables in the equations are set to constants except for x, with a dynamic range [0, 2].

$$g(x, y) = \exp(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}) \cdot \cos(2\pi\frac{x'}{\lambda} + \psi) \tag{4}$$

$$\begin{aligned} x' &= x \cdot \cos\theta + y \cdot \sin\theta \\ y' &= -x \cdot \sin\theta + y \cdot \cos\theta \end{aligned} \tag{5}$$

$$g(x) = a \cdot x^{n-1} \cdot \cos(2\pi f x + \varphi) \cdot \exp(-2\pi b x) \tag{6}$$

The kernels are applied to Non-Uniform FFT (NUFFT) convolution computation [20] with an image containing $10K \times 10K$ non-uniform source samples and a local convolution window of size 5×5 on the target Cartesian grid. The free variable x in the kernel functions stands for the distance between a pair of source and target points. The application with the generated C code for kernel functions is compiled by Intel Compiler and evaluated with LUTs on Intel HaperTown [24] multicore machine, which has dual quad-core connected to private L1 caches and shared L2 caches for each pair of cores. We employ the IEEE double-precision floating point data format, and compare the accuracy against the code using standard C library routines for kernel functions.

Figure 4 shows the execution time comparison between our solution and library version for NUFFT convolution with different kernels, for an accuracy of 10^{-15}. The versions indicated with "(Lib)", use C library calls for exponential, cosine and sine functions in the kernel implementations, whereas the versions marked with "(Ours)" use our generated code and LUTs for the kernel functions. It can be seen that the performance improvements brought by our approach are significant, 42%, 15%, 9%, 52% and 53% for Gaussian, Gabor, Gammatone, Gabor+Gammatone, and Gabor*Gammatone kernels, respectively, irrespective of the number of cores, as the application is fully parallelized. The improvements are achieved mainly by merging multiple functions as well as merging variable expressions into the function LUTs to eliminate the evaluation of these expressions in the computation. However, the improvement with the first kernel, which has only one elementary function, is the result of a further optimization on the generated code and its LUT. More specifically, we observe that the dynamic data range in the application is so small [0, 0.073] that only a few LUT entries are referenced; therefore, we encode the actual numbers in the LUT into the code directly, to reduce memory accesses.

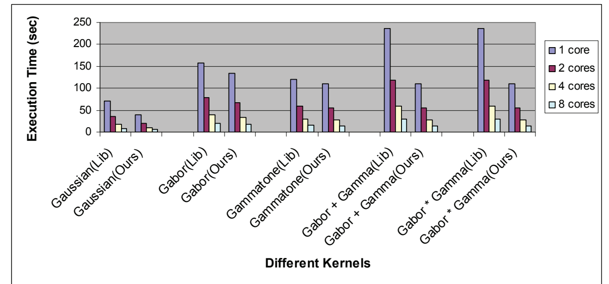

Fig. 4. Performance comparison between our solution and library version for NUFFT convolution with different kernels on Intel HarperTown.

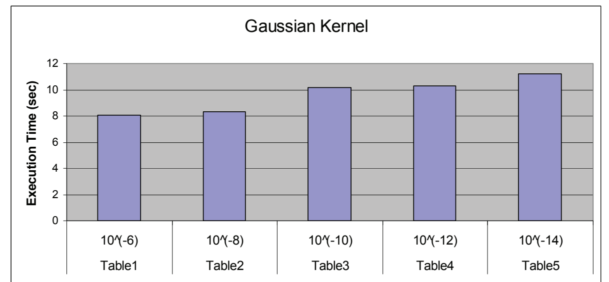

Fig. 5. Execution time of NUFFT convolution with Gaussian kernel, using different LUTs on 8-core Intel HarperTown.

Figure 5 depicts the execution time of NUFFT convolution

with Gaussian kernel on 8-core HarperTown, using different LUTs corresponding to different accuracy requirements. In this experiment, we decrease the dynamic data range from [0, 2] to [0, 0.1] in the input (to SP compiler) based on profiling to make fully utilization of the entries in the generated LUTs. Observe that, the execution time increases as the accuracy is higher, however, the increase is not linear, since there are two sudden jumps between Table 2 and 3, and Table 4 and 5. Here, Table 1 and 2 have a degree of 2, Table 3 and 4 have a degree of 3, and Table 5 has a degree of 4, despite their different sampling intervals. In fact, both sampling interval and polynomial degree affect the LUT size and thus cache misses on multicores (the data format is fixed), while the latter also affects the computation time. Figure 5 tells us that large LUT size is not a big problem with today's huge L2 or L3 caches on multicores, but high polynomial order has a significant effect on the timing due to expensive floating point operations. There is a tradeoff between accuracy and performance for applications, which can be explored by our compiler.

We also conducted experiments on an Xilinx Vertex2Pro-100 FPGA platform using the IEEE single-precision floating point data format, for an accuracy of 10^{-6}, by translating our generated MATLAB-like code into hardware designs through the tool developed in [19]. Table II presents the actual resource utilization of the solutions provided by our compiler and by direct implementation methods, for Gabor, Gammatone, Gabor+Gammatone and Gabor*Gammatone kernels, respectively. The direct implementation of Gabor kernel requires 3 sine/cosine, 1 Gaussian, 3 adders/subtractors and 8 multipliers, and the direct implementation of Gammatone kernel requires 1 cosine, 1 exponential, 1 adders/subtractor and 7 multipliers. We can see that our solutions bring huge resource savings via function composition, which require, on average, 85.7% fewer slices and 40.2% fewer BRAMs than direct implementation methods.

TABLE II
RESOURCE UTILIZATION OF KERNEL FUNCTIONS BY DIRECT IMPLEMENTATION METHOD AND OUR SOLUTION.

Filters		Gabor	Gammatone	Gab+Gam	Gab*Gam
Direct Method	#Slice	5064	2900	8316	8145
	#BRAM	12	45	57	57
Our Solution	#Slice	639	639	639	639
	#BRAM	3	42	42	42

VIII. Conclusion and Future Work

This paper presents a useful tool for generating customized LUTs and implementations for function evaluation. It brings significant performance improvements for applications on multicores as well as significant resource savings on FPGAs, using function composition. The flow can be potentially part of FPGA synthesis, or MATLAB/C++ and re-targetable compilers. As our future work, we plan to extend this tool to support multiple free variables as well as multidimensional functions in the input, and test it on a larger set of applications to demonstrate its applicability to complex functions.

Acknowledgments

This work is supported in part by the DARPA DESA program, the NSF grants CNS #0720645, CCF #0811687, CCF #0702519, CNS #0202007 and CNS #0509251, as well as a grant from Microsoft Corporation.

References

[1] P. Tang, "Table-lookup algorithms for elementary functions and their error analysis", *Proceedings of IEEE Symposium on Computer Architecture*, pp. 232-236, 1991.
[2] J. Cao and et al., "High-performance architectures for elemenatry function generation", *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pp. 136-144, 2001.
[3] J. Muller, "Partially rounded small-order approximations for accurate, hardware-oriented, table-based methods", *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pp. 144-121, 2003.
[4] T. Sasao and et al., "Numerical function generators using LUT cascades", *IEEE Transactions on Computers* **56**, pp. 826-838, 2007.
[5] N. Takagi, "Powering by a table look-up and a multiplication with operand modification", *IEEE Transactions on Computers* **47**, pp. 1216-1222, 1998.
[6] J. Pineiro and et al., "Faithful powering computation using table look-up and a fused accumulation tree", *Proceedings of IEEE Symposium on Computer Architecture*, pp. 40-47, 2001.
[7] W. Cody and L. Stoltz, "The use of Taylor series to test accuracy of function programs", *ACM Transacitons on Mathematical Software (TOMS)* **17**, pp. 55-63, 1991.
[8] P. Tang, "Table-driven implementation of the exponential function in IEEE floating-point arithmetic", *ACM Transactions on Mathmatical Software (TOMS)* **15**, pp. 144-157, 1989.
[9] K. Sobti and et al., "Efficient function evaluations with lookup tables for structured matrix operation", *Proceedings of IEEE Workshop on Signal Processing Systems*, pp. 463-468, 2007.
[10] H. Bui and S. Tahar, "Design and synthesis of an IEEE-754 exponential function", *IEEE Candian Conference on Electrical and Computer Engineering* **1**, pp. 450-455, 1999.
[11] J. Hauser and C. Purdy, "Approximating functions for embedded and ASIC applications", *Proceedings of the 44th IEEE Midwest Symposium on Circuits and Systems* **1**, pp. 478-481, 2001.
[12] F. Hildebrandt, "Introduction to numerical analysis", *McGraw-Hill Book Company*, 1956.
[13] J. Wilkinson, "Rounding errors in algebraic processes", *Prentice-Hall Inc.*, 1964.
[14] L. Deng and et al. "Accurate models for estimating area and power of FPGA implementations", *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 1417-1420, 2008.
[15] M. Unser, "Splines: a perfect fit for signal and image processing", *IEEE Signal Processing Magazine* **16**, pp. 22-38, 1999.
[16] J. Davidson and S. Jinturkar, "Compiler construction", *SrpingerLink*, 1996.
[17] T. Parr and R. Quong, "ANTLR: a predicated-LL(k) parser generator", *Software-Parctice and Expereience*, pp. 789-810, 1995.
[18] L. Deng and et al., "Automated function implementation under accuracy, area, and memory constraints", *Proceedings of SPIE*, 2009.
[19] J. Kim and et al., "An automated framework for accelerating numerical algorithms on reconfigurable platforms using algorithmic/architectural optimization", *IEEE Transactions on Computers*, pp. 1654-1667, 2009.
[20] Y. Zhang and et al, "Exploring Parallelization Strategies for NUFFT Data Translation", *International Conference on Embedded Software*, 2009.
[21] J. Movshon, "Tutorial on gabor filters", 2009.
[22] M. Slaney, "An efficient implementation of the Patterson-Holdsworth auditory filter bank", 1993.
[23] MathWorks, "Using the MATLAB engine to call MATLAB software from C", 2009.
[24] http://en.wikipedia.org/wiki/Xeon.
[25] W. Pankiewicz, "Algorithms: algorithm337: calculation of a polynomial and its drivative values by Horner scheme", *Communications of the ACM*, pp. 633, 1968.