Automating Optimized Table-with-Polynomial Function Evaluation for FPGAs

Dong-U Lee, Oskar Mencer, David J. Pearce and Wayne Luk

Department of Computing, Imperial College, London, UK
{dong.lee, o.mencer, d.pearce, w.luk}@ic.ac.uk

Abstract. Function evaluation is at the core of many compute-intensive applications which perform well on reconfigurable platforms. Yet, in order to implement function evaluation efficiently, the FPGA programmer has to choose between a multitude of function evaluation methods such as table lookup, polynomial approximation, or table lookup combined with polynomial approximation. In this paper, we present a methodology and a partially automated implementation to select the best function evaluation hardware for a given function, accuracy requirement, technology mapping and optimization metrics, such as area, throughput and latency. The automation of function evaluation unit design is combined with ASC, A Stream Compiler, for FPGAs. On the algorithmic side, MATLAB designs approximation algorithms with polynomial coefficients and minimizes bitwidths. On the hardware implementation side, ASC provides partially automated design space exploration. We illustrate our approach for $\sin(x)$, $\log(1+x)$ and 2^x with a selection of graphs that characterize the design space with various dimensions, including accuracy, precision and function evaluation method. We also demonstrate design space exploration by implementing more than 400 distinct designs.

1 Introduction

The evaluation of functions can often be the performance bottleneck of many compute-bound applications. Examples of these functions include elementary functions such as $\log(x)$ or $\sqrt{x}$, and compound functions such as $(1-\sin^2(x))^{1/2}$ or $\tan^2(x)+1$. Hardware implementation of elementary functions is a widely studied field with many research papers (e.g. [1][10][11][12]) and books (e.g. [2][8]) devoted to the topic. Even though many methods are available for evaluating functions, it is difficult for designers to know which method to select for a given implementation.

Advanced FPGAs enable the development of low-cost and high-speed function evaluation units, customizable to particular applications. Such customization can take place at run time by reconfiguring the FPGA, so that different functions, function evaluation methods, or precision can be introduced according to run-time conditions. Consequently, the automation of function evaluation design is one of the key bottlenecks in the further application of function evaluation in reconfigurable computing. The main contributions of this paper are:

– A methodology for the automation of function evaluation unit design, covering table lookup, table with polynomial, and polynomial-only methods.

– An implementation of a partially automated system for design space exploration of function evaluation in hardware, including:
  - Algorithmic design space exploration with MATLAB.
  - Hardware design space exploration with ASC.
– Method selection results for $\sin(x)$, $\log(1+x)$ and 2^x.

The rest of this paper is organized as follows. Section 2 covers overview and background material. Section 3 presents the algorithmic design space exploration with MATLAB. Section 4 describes the automation of the ASC design space exploration process. Section 5 shows how ASC designs can be verified. Section 6 discusses results, and Section 7 offers conclusion and future work.

2 Overview and Background

We can use polynomials and/or lookup tables for approximating a function $f(x)$ over a fixed range $[a, b]$. On one extreme, the entire function approximation can be implemented as a table lookup. On the other extreme, the function approximation can be implemented as a polynomial approximation with function-specific coefficients. In our work, we use Horner's rule to reduce the number of multiplications.

Between these two extremes, we use a table followed by a polynomial. This table with polynomial method partitions the total approximation into several segments. In this work, we employ uniformly sized segments, which have been widely studied in literature [1][3][5]. Uniform segmentation performs well for functions that are relatively linear, such as the functions we consider in this paper. However, for highly non-linear functions, non-uniform segmentation methods such as the hierarchical segmentation method [4] have been found to be more appropriate.

In [7] the results show that for a given accuracy requirement it is possible to plot the area, latency, and throughput tradeoff and thus identify the optimal function evaluation method. The optimality depends on further requirements such as available area, required latency and throughput. Looking at Figure 1, if one desires the metric to be low (e.g. area or latency), one should use method 1 for bitwidths lower than x1, method 2 for bitwidths between x1 and x2, and method 3 for bitwidths higher than x2. We shall illustrate this approach using Figures 13 to 15, where several methods are combined to provide the optimal implementations in area, latency or throughput for different bitwidths for the function $\sin(x)$.

The contribution of this paper is the design and implementation of a methodology to automate this process. Here, MATLAB automates the mathematical side of function approximation (e.g. bitwidth and coefficient selection), while *A Stream Compiler (ASC)* [6] automates the design space exploration of area, latency and throughput. Figure 2 shows the proposed methodology.

3 Algorithmic Design Space Exploration with MATLAB

Given a target accuracy, or number of output bits so that the required accuracy is one unit in the last place (1 ulp), it is straightforward to automate the design of a sufficiently

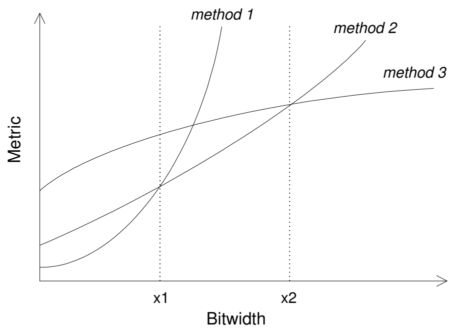

Fig. 1. Certain approximation methods are better than others for a given metric at different precisions.

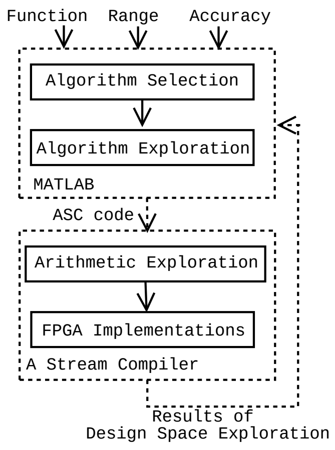

Fig. 2. Block diagram of methodology for automation.

accurate table, and with help from MATLAB, also to find the optimal coefficient for a polynomial-only implementation. The interesting designs are between the table-only and polynomial-only designs – those involving both a table and a polynomial. Three MATLAB programs have been developed: TABLE (table lookup), TABLE+POLY (table with polynomial) and POLY (polynomial only). The programs take a set of parameters (e.g. function, input range, operand bitwidth, required accuracy, bitwidths of the operations and the coefficients and the polynomial degree) and generate function evaluation units in ASC code.

TABLE produces a single table, holding results for all possible inputs; each input is used to index the table. If the input is n bits and the precision of the results is m bits, the size of the table would be $2^n \times m$. It can be seen that the disadvantage of this approach is that the table size is exponential to the input size.

TABLE+POLY implements the table with polynomial method. The input interval $[a, b]$ is split into $N = 2^I$ equally sized segments. The I leftmost bits of the argument x serve as the index into the table, which holds the coefficients for that particular interval. We use degree two polynomials for approximating the segments, but other degrees are possible. The program starts with $I = 0$ (i.e. one segment over the whole input range) and finds the minimax polynomial coefficients which minimize the maximum absolute error. I is incremented until the maximum error over all segments is lower than the requested error. The operations are performed in fixed point and in finite precision with the user supplied parameters, which are emulated by MATLAB.

POLY generates an implementation which approximates the function over the whole input range with a single polynomial. It starts with a degree one polynomial and finds the minimax polynomial coefficients. The polynomial degree is incremented until the desired accuracy is met. Again, fixed point and finite precision are emulated.

4 Hardware Design Space Exploration with ASC

ASC [6] enables a software-like programming of FPGAs. ASC is built on top of the module generation environment PAM-Blox II, which in turn builds upon the PamDC [9] gate library. While [6] shows the details of the design space exploration process with ASC, we now utilise ASC (version 0.5) to automate this process. The idea is to retain user-control over all features available on the gate level, whilst automating many of the tedious tasks involved in exploring the design space. Therefore ASC allows the user to specify the dimensions of design space exploration, e.g. bitwidths of certain variables, optimization metrics such as area, latency, or throughput, and in fact anything else that is accessible in ASC code, which includes algorithm level, arithmetic unit level and gate level constructs. For example, suppose we wish to explore how the bitwidth of a particular ASC variable affects area and throughput. To do this we first parameterize the bitwidth definition of this variable in the ASC code. Then we specify the detail of the exploration in the following manner:

$$\mathtt{RUN0} \quad = \quad -\mathtt{XBITWIDTH} = \{8, 16, 24, 32\} \tag{1}$$

which states that we wish to investigate bitwidths of 8, 16, 24 and 32. At this point, typing ‘make run0’ begins an automatic exploration of the design space, generating a vast array of data (e.g. Number of 4-input LUTs, Total Equivalent Gate Count, Throughput and Latency) for each different bitwidth. ASC also automatically generates graphs for key pieces of this data, in an effort to further reduce the time required to evaluate it.

The design space explorer, or “user”, in our case is of course the MATLAB program that mathematically designs the arithmetic units on the algorithmic level and provides ASC with a set of ASC programs, each of which results in a large number of implementations. Each ASC implementation in return results in a number of design space exploration graphs and data files. The remaining manual step, which is difficult to automate, involves inspecting the graphs and extracting useful information about the variation of the metrics. It would be interesting to see how such information from the hardware design space exploration can be used to steer the algorithmic design space exploration.

One dimension of the design space is technology mapping on the FPGA side. Should we use block RAMs, LUT memory or LUT logic implementations of the mathematical lookup tables generated by MATLAB? Table 1 shows ASC results which substantiate the view that logic minimization of tables containing smooth functions is usually preferable over using block RAMs or LUT memory to implement the table. Therefore, in this work we limit the exploration to combinational logic implementations of tables.

5 Verification with ASC

One major problem of automated hardware design is the verification of the results, to make sure that the output circuit is actually correct. ASC offers two mechanisms for this activity based on a software version of the implementation.

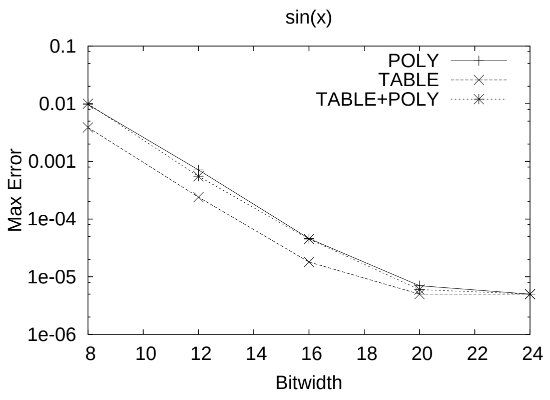

Fig. 3. Accuracy graph: maximum error versus bitwidth for $\sin(x)$ with the three methods.

Table 1. Various place and route results of 12-bit approximations to $\sin(x)$. The logic minimized LUT implementation of the tables minimizes latency and area, while keeping comparable throughput to the other methods, e.g. block RAM (BRAM) based implementation.

ASC optimization	memory type	4-input LUTs	clock speed [MHz]	latency [ns]	throughput [Mbps]
latency	block RAM	**919 + 1BRAM**	17.89	**111.81**	250.41
	LUT memory	1086	15.74	63.51	220.43
	LUT logic	**813**	16.63	**60.11**	232.93
throughput	block RAM	**919 + 1BRAM**	39.49	177.28	**552.79**
	LUT memory	1086	36.29	192.88	508.09
	LUT logic	**967**	39.26	178.29	**549.67**

– **Accuracy Graphs**: graphs showing the accuracy of the gate-level simulation result (SIM) compared to a software version using double precision floating point (SW), automatically generated by MATLAB, plotting:
  `max.error` $= max(|SW - SIM|)$, or
  `max.error` $= max(|SW - FPGA|)$
  when comparing to an actual FPGA output ($FPGA$).
  Figure 3 shows an example graph. Here the precisions of the coefficients and the operations are increased according to the bitwidth (e.g. when bitwidth=16, all coefficients and operations are set to 16 bits), and the output bitwidth is fixed at 24 bits.
– **Regression Testing**: same as the accuracy graph, but instead of plotting a graph, ASC compares the result to a maximally tolerated error and reports only ‘pass’ or ‘fail’ at the end. This feature allows us to automate the generation and execution of a large number of tests.

6 Results

We show results for three elementary functions: $\sin(x)$, $\log(x+1)$ and 2^x. Five bit sizes 8, 12, 16, 20 and 24 bits are considered for the bitwidth. In this paper, we implement designs with n-bit inputs and n-bit outputs. However, the position of the decimal (or binary) point in the input and output formats can be different in order to maximize the precision that can be described. The results of all 400 implementations are post place and route, and are implemented on a Xilinx Virtex-II XC2V6000-6 device.

In algorithmic space explored by MATLAB, there are three methods, three functions and five bitwidths, resulting in 45 designs. These designs are generated by the user with hand-optimized coefficient and operation bitwidths. ASC takes the 45 algorithmic designs and expands them into over 400 implementations in the hardware space. With the aid of the automatic design exploration features of ASC (Section 4), we are able to generate all the implementation results in one go with a single 'make' file. It takes around twelve hours on a dual Athlon XP 2.13GHz PC with 2GB RAM.

The following graphs are a subset of the full design space exploration which we show for demonstration purposes. Figures 4 to 15 show a set of FPGA implementations resulting from a 2D cut of the multidimensional design space.

In Figures 4 to 6, we fix the function and approximation method to $\sin(x)$ and TABLE+POLY, and obtain area, latency and throughput results for various bitwidths and optimization methods. Degree two polynomials are used for all TABLE+POLY experiments in our work.

Figure 4 shows how the area (in terms of the number of 4-input LUTs) varies with bitwidth. The lower part shows LUTs used for logic while the small top part of the bars shows LUTs used for routing. We observe that designs optimized for area are significantly smaller than other designs. In addition, as one would expect, the area increases with the bit width. Designs optimized for throughput have the largest area; this is due to the registers used for pipelining. Figure 5 shows that designs optimized for latency have significantly less delay, and the increase in delay with the bitwidth is lower than others. Designs optimized for area have the longest delay, which is due to hardware being shared in a time-multiplexed manner. Figure 6 shows that designs optimized for throughput perform significantly better than others. Designs optimized for area perform worst, which is again due to the hardware sharing. An interesting observation is the fact that throughput is relatively constant with bitwidth. This is due to increased routing delays as designs get larger with increased precision requirements.

Figures 7 to 9 show various metric-against-metric scatter plots of 12-bit approximations to $\sin(x)$ with different methods and optimizations. For TABLE, only results with area optimization are shown since the results for other optimizations applied are identical. With the aid of such plots, one can decide rapidly what methods to use for meeting specific requirements in area, latency or throughput.

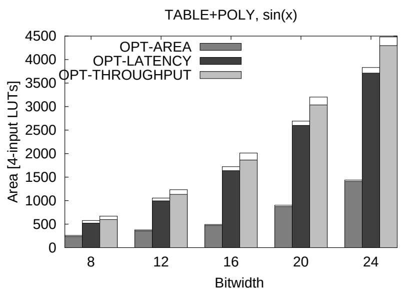

Fig. 4. Area versus bitwidth for $\sin(x)$ with TABLE+POLY. OPT indicates for what metric the design is optimized for. Lower part: LUTs for logic; small top part: LUTs for routing.

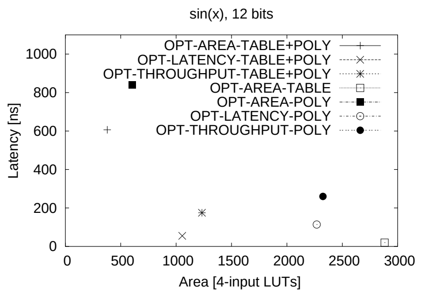

Fig. 7. Latency versus area for 12-bit approximations to $\sin(x)$. The Pareto optimal points in the latency-area space are shown.

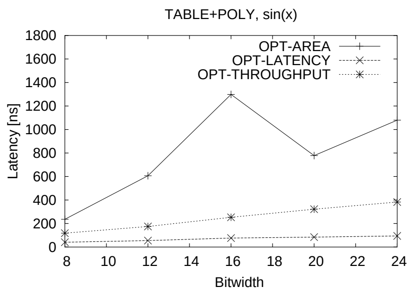

Fig. 5. Latency versus bitwidth for $\sin(x)$ with TABLE+POLY. Shows the impact of latency optimization.

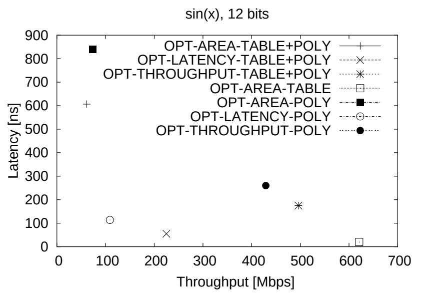

Fig. 8. Latency versus throughput for 12-bit approximations to $\sin(x)$. The Pareto optimal points in the latency-throughput space are shown.

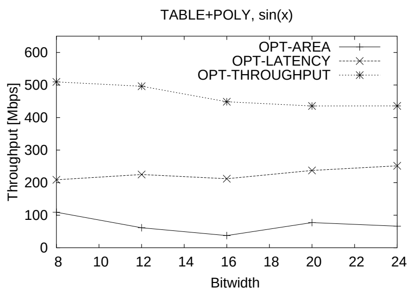

Fig. 6. Throughput versus bitwidth for $\sin(x)$ with TABLE+POLY. Shows the impact of throughput optimization.

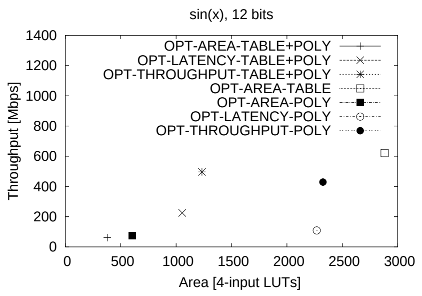

Fig. 9. Area versus throughput for 12-bit approximations to $\sin(x)$. The Pareto optimal points in the throughput-area space are shown.

In Figures 10 to 12, we fix the approximation method to TABLE+POLY, and obtain area, latency and throughput results for all three functions at various bitwidths. Optimum optimization methods are used for all three experiments (e.g. area is optimized to get the area results).

From Figure 10, we observe that $\sin(x)$ requires the most and 2^x requires the least area. The difference gets more apparent as the bitwidth increases. This is because 2^x is the most linear of the three functions, hence requires fewer number of segments for the approximations. This leads to a reduction in the number of entries in the coefficient table and hence less area on the device.

Figure 11 shows the variations of the latency with the bitwidth. We observe that all three functions have similar behavior. In Figure 12, we observe that again the three functions have similar behavior, with 2^x performing slightly better than others for bitwidths higher than 16. We suspect that this is because of the lower area requirement of 2^x, which leads to less routing delay.

Figures 13 to 15 show the main emphasis and contribution of this paper, illustrating which approximation method to use for the best area, latency or throughput performance. We fix the function to $\sin(x)$ and obtain results for all three methods at various bit widths. Again, the optimum optimization is used for a given experiment. For experiments involving TABLE, we have managed to obtain results up to 12 bits only, due to memory limitations of our PCs.

From Figure 13, we observe that TABLE has the least area at 8 bits, but the area increases rapidly making it less desirable at higher bitwidths. The reason for this is the exponential increase in table to the input size for full lookup tables. The TABLE+POLY approach yields the least area for precisions higher than eight bits. This is due to the efficiency of using multiple segments with minimax coefficients for each. We have observed that for POLY, roughly one more polynomial term (i.e. one more multiply-and-add module) is needed every four bits. Hence, we see a linear behavior with the POLY curve.

Figure 14 shows that TABLE has significantly smaller latency than others. We expect that this will be the case for bitwidths higher than 12 bits as well. POLY has the worst delay, which is due to computations involving high-degree polynomials, and the terms of the polynomials increase with the bitwidth. The latency for TABLE+POLY is relatively low across all bitwidths, which is due to the fact that the number of memory accesses and polynomial degree are fixed.

In Figure 15, we observe how the throughput varies with bitwidth. For low bitwidths, TABLE designs result in the best throughput, which is due to the short delay for a single memory access. However, the performance quickly degrades and we predict that at bit widths higher than 12 bits, it will perform worse than the other two methods due to rapid increase in routing congestion. The performance of TABLE+POLY is better than POLY before 15 bits and gets worse after. This is due to the increase in the size of the table with precision, which leads to longer delays for memory accesses.

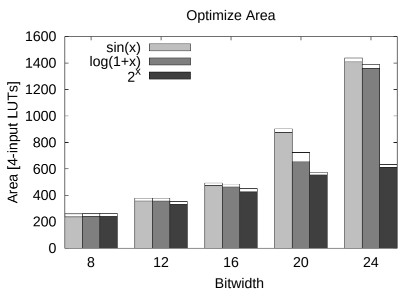

Fig. 10. Area versus bitwidth for the three functions with TABLE+POLY. Lower part: LUTs for logic; small top part: LUTs for routing.

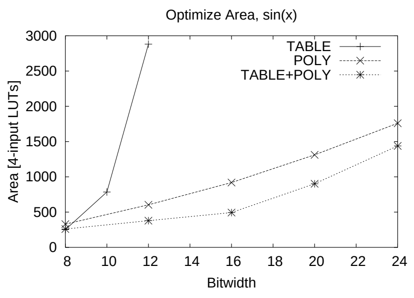

Fig. 13. Area versus bitwidth for $\sin(x)$ with the three methods. Note that the TABLE method gets too large already for 14 bits.

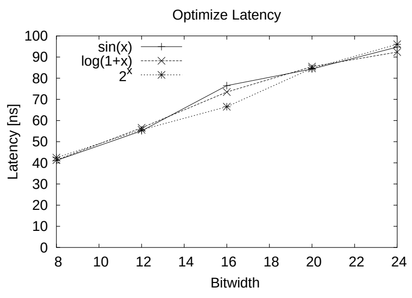

Fig. 11. Latency versus bitwidth for the three functions with TABLE+POLY.

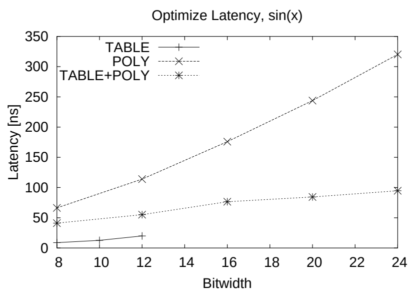

Fig. 14. Latency versus bitwidth for $\sin(x)$ with the three methods.

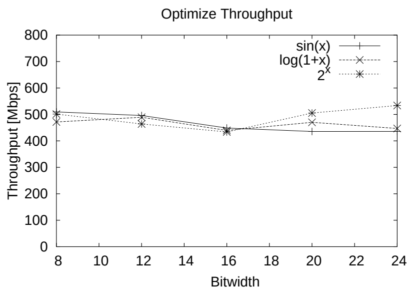

Fig. 12. Throughput versus bitwidth for the three functions with TABLE+POLY. Throughput is similar across functions, as expected.

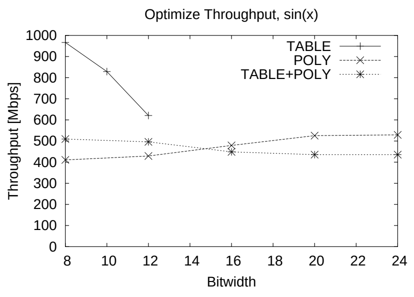

Fig. 15. Throughput versus bitwidth for $\sin(x)$ with the three methods.

7 Conclusions

We present a methodology for the automation of function evaluation unit design, covering table lookup, table with polynomial, and polynomial-only methods. An implementation of a partially automated system for design space exploration of function evaluation in hardware has been demonstrated, including algorithmic design space exploration with MATLAB and hardware design space exploration with ASC.

We conclude that the automation of function evaluation unit design is within reach, even though there are many remaining issues for further study. Current and future work includes optimizing polynomial evaluation, exploring the interaction between range reduction and function evaluation, including more approximation methods, and developing a complete and seamless automation of the entire process.

Acknowledgements

The authors thank George Constantinides and Ray Cheung for their assistance. The support of Xilinx Inc. and the U.K. Engineering and Physical Sciences Research Council (Grant number GR/N 66599, GR/R 55931 and GR/R 31409) is gratefully acknowledged.

References

1. J. Cao, B.W.Y. We and J. Cheng, " High-performance architectures for elementary function generation", *Proc. 15th IEEE Symp. on Comput. Arith.*, 2001.
2. M.J. Flynn and S.F. Oberman, *Advanced Computer Arithmetic Design*, John Wiley & Sons, New York, 2001.
3. V.K. Jain, S.A. Wadecar and L. Lin, "A universal nonlinear component and its application to WSI", *IEEE Trans. Components, Hybrids and Manufacturing Tech.*, vol. 16, no. 7, pp. 656–664, 1993.
4. D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "Hierarchical Segmentation Schemes for Function Evaluation", *Proc. IEEE Int. Conf. on Field-Prog. Tech.*, pp. 92–99, 2003.
5. D.M. Lewis, "Interleaved memory function interpolators with application to an accurate LNS arithmetic unit", *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 974–982, 1994.
6. O. Mencer, D.J. Pearce, L.W. Howes and W. Luk, "Design space exploration with a stream compiler", *Proc. IEEE Int. Conf. on Field-Prog. Tech.*, pp. 270–277, 2003.
7. O. Mencer and W. Luk, "Parameterized high throughput function evaluation for FPGAs", *J. of VLSI Sig. Proc. Syst.*, vol. 36, no. 1, pp. 17–25, 2004.
8. J.M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser Verlag AG, 1997.
9. B. Patrice, R. Didier and V. Jean, "Programmable active memories: a performance assessment", *Proc. ACM Int. Symp. on Field-Prog. Gate Arrays*, 1992.
10. M.J. Schulte and J.E. Stine, "Approximating elementary functions with symmetric bipartite tables", *IEEE Trans. on Comput.*, vol. 48, no. 9, pp. 842–847, 1999.
11. P.T.P. Tang, "Table lookup algorithms for elementary functions and their error analysis", *Proc. IEEE Symp. on Comput. Arith.*, pp. 232–236, 1991.
12. W.F. Wong and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers", *IEEE Trans. on Comput.*, vol. 43, pp. 278–294, 1994.