Efficient Floating Point Precision Tuning for Approximate Computing

Nhut-Minh Ho[1], Elavarasi Manogaran[1], Weng-Fai Wong[1], and Asha Anoosheh[2]

[1]School of Computing, National University of Singapore
e-mail: {minhhn,elamano,wongwf}@comp.nus.edu.sg
[2]Dept. of Electrical Engineering & Computer Sciences, University of California, Berkeley
e-mail: asha@berkeley.edu

Abstract— This paper presents an automatic tool-chain that efficiently computes the precision of floating point variables down to the bit level of the mantissa. Our toolchain uses a distributed algorithm that can analyze thousands of variables. We successfully used the tool to transform floating point signal processing programs to their arbitrary precision fixed-point equivalent, obtaining about 82% and 66% average reduction in resources when compared to the double precision and single precision versions, respectively.

I. Introduction

Floating-point arithmetic is ubiquitous and convenient in computing. However, with generally only two sizes, i.e., single- and double-precision, available for selection, it can lead to wastage in compute cycles (and hence energy), as well as storage space. Recognizing this, the latest processors are starting to introduce shorter precision floating point formats. Nvidia's upcoming general purpose graphic processing unit (GPGPU) computing architecture will support half-precision floating-point arithmetic natively to target certain application domains [4]. The FPGA community has also recognized this, and the use of lower precision choices in FPGA devices including fixed-point and shorter floating-point arithmetic [9, 14] have been suggested. Furthermore, there have been active research on variable precision formats, especially for field programmable processors [11], giving rise to the field of *approximate computing*.

Utilizing all the multiple precision choices effectively necessitate *precision analysis*. However, without proper tool support, precision analysis can be complex and tedious, and in particular, difficult to scale to larger and more complex applications. Unfortunately, today's precision analysis tools either produce conservative results [3], or can only deal with very small code sizes [13, 17]. In this paper, we will introduce an efficient tool-chain to support precision tuning not only for floating-point programs running on CPU but also for fixed-point programs on different architectures. It is a distributed search that support multi-purpose precision tuning. The toolchain can yield result in less than a second for more than half of our experiments. The result is for arbitrary precision and hence can be used for any precision formats available in the actual hardware. Apart from comparing the result with the precision tuning for large programs, we also showed how our toolchain can be applied to fixed-point conversion.

II. Related work

Approximate computing is well etablished, and we refer the reader to detailed surveys of the whole field [24]. We shall only examine a small subset of works that are most relevant to the work described here.

One direction of work in the reduction of floating point precision uses code profiling to track the range of variables [10] or in search algorithms [17]. These techniques are either conservative or failed to scale up to large programs. Recently, using a similar profiling approach, Precimonious [23] became the first tool to be able to analyze considerably large programs. The tool was later augmented with *blame analysis* [22] to further improve performance. Most of these tools automatically determine the best mix of single or double precision for code. We differ from these techniques in being able to tune for an arbitrary number of precision bits for the floating point mantissa.

Approximation in circuit synthesis attempts to minimize area, and energy under some error constraints. In the early days, when floating point units were too resource heavy to be implemented in many devices including FPGAs, fixed point arithmetic was used. Apart from search-based wordlength optimization proposed in [8], Gaffar et al. proposed a unified approach for bitwidth analysis for both floating point and fixed point designs through the use of automatic differentiation [13]. Techniques and tools such as [19] use interval arithmetic and affine arithmetic to calculate the range and precision of floating point variables. The most recent work in high level synthesis of approximate computing circuits that are related to ours includes [20], [21]. Most of these techniques perform conservative approximations on computations which could lead to an over-estimation and over-provisioning of hardware resources. Our tool can be seen as a uniform solution for both software as well as circuit synthesis.

III. Overview

Given an application's source code, we first rewrite the program into one that uses a arbitrary-precision version that uses

GNU MPFR (GNU Multiple Precision Floating-Point Routines) [12]. We could have also used any similar library. The rewriting process itself is done using a modified version of the C syntax parser and code generator `pycparser`[1]. Because intermediate results are present during the execution, we also introduced an intermediate variable for each binary operation that has MPFR operands. The rewritten program will obtain the precision value of each variable from a configuration file at the beginning of each execution to initialize all the required MPFR variables. This is then used in the search which is implemented in Python. As the search proceeds, only the external configuration file will be modified before each run. After obtaining the search result, we can choose to further refine the result based on some statistical features, or end the search. The whole process is automatic and parallelizable, with little to no human interaction, depending on the application.

IV. Arbitrary-precision tuning

This section presents the key components of our toolchain for arbitrary-precision tuning. In this paper, by 'floating-point precision' we are referring to the precision of the mantissa.

A. Problem formulation

A given program has N variables $X = \{x_1, x_2, \dots, x_N\}$, each having the (same) upper-bound U and lower-bound L of precision. For our work, we set the $L = 4$ and $U = 53$ (double precision). Assigning each x_i to a precision p_i bits in its mantissa, $L \leq p_i \leq U, i \in 1, N$, we get a vector $P = \{p_1, p_2, \dots, p_N\}$. For convenience, we shall use P to denote the entire array where $P[i] = p_i$. Let δ be the error induced by running the program with precision P. The δ value is calculated differently depending on the original program. We define the whole process of running a program with precision P, and deriving the output error as the function $\delta = F(P)$. The target of the search is to find the smallest precision possible for each variable while keeping $\delta \leq \epsilon$, some user given bound. Before describing the algorithm, we need to introduce the concept of an *influence group*.

Definition IV.1 *Consider the dependency graph of a program, the influence group $\mathcal{G}[i]$ is the list of variables along some program path from x_i to the last variable that is affected by the value of x_i.*

$\mathcal{G}[i]$ is a simple way of capturing the impact of changing the precision of variable x_i on other variables.

B. Distributed search algorithm

Our search algorithm consists of 2 phases, both parallelizable. In the pseudo-code (Algorithm 1), we marked the parallelizable loops as *MPI_Parallel*. The number of parallel MPI threads is set to N. To discuss the theoretical upper-bound of run time of each step in the following explanation, we shall assume that the network delay and synchronization time of the algorithm is negligible compared to the execution time of the target program. This assumption is true in practice as we target large running programs that take minutes or hours to complete one execution. The time unit referred in this section is the execution time of the target program.

[1]https://github.com/eliben/pycparser

Algorithm 1 Heuristic precision tuning

```
 1: procedure ITERATIVE SEARCH                          ▷ main procedure
 2:     $\mathrm{MWL}_0 \leftarrow \{L_1, L_2, \dots, L_N\}$                       ▷ initialize
 3:     $P_0 \leftarrow \{U_1, U_2, \dots, U_N\}$                         ▷ initialize
 4:     repeat
 5:         $\mathrm{MWL}_k \leftarrow$ ISOLATED DOWNWARD($\mathrm{MWL}_{k-1}, P_{k-1}$)
 6:         $P_k \leftarrow$ GROUPED UPWARD($\mathrm{MWL}_k$)
 7:     until Converged
 8:     return $P_k$
 9: end procedure
10: procedure ISOLATED DOWNWARD(MWL, $P$)
11:     $P_{\text{temp}} \leftarrow P$
12:     for $i \leftarrow 1, N$ do                                ▷ MPI_Parallel
13:         $P_{temp}[i] \leftarrow$ BINARYSEARCH($MWL[i], P[i], P, i$)
14:     end for
15:     return $P_{temp}$
16: end procedure
17: procedure GROUPED UPWARD($P$)
18:     $\delta_{min} \leftarrow F(P)$
19:     $\Delta \leftarrow \{0, 0, \dots, 0\}$          ▷ results from parallel threads
20:     $P_{min} \leftarrow P$
21:     repeat
22:         for $i \leftarrow 1, N$ do                            ▷ MPI_Parallel
23:             $P_{\text{temp}} \leftarrow$ INCGROUPPREC($P_{\text{min}}, i$)
24:             $\Delta[i] \leftarrow F(P_{\text{temp}})$
25:         end for
26:         $\delta_{\text{min}}, I_{\text{min}} \leftarrow$ min value and its index in $\Delta$
27:         $P_{\text{min}} \leftarrow$ INCGROUPPREC($P_{\text{min}}, I_{\text{min}}$)
28:     until $\delta_{\text{min}} \leq \epsilon$
29:     return $P_{\text{min}}$
30: end procedure
```

We use a similar idea of *minimum word length* (MWL) described in `Min+b` algorithm [5] for the first step, then exploiting the internal structure of the target program to improve the second step. The first call to ISOLATED DOWNWARD(MWL_0, P_0) in our algorithm corresponds to the first phase in Min+b, except that we use binary search combined with *MPI_Parallel* at the variable-level to reduce the maximum search time to $\lfloor log_2(U - L) + 1 \rfloor$. At each parallel thread, BINARYSEARCH($\mathrm{MWL}[i], P[i], P, i$) will find the minimum required precision for x_i while setting other variables at the precision given in P. It uses binary search strategy in the range from $\mathrm{MWL}[i]$ to $P[i]$. The second phase of our algorithm is a competition between different influence groups (and there are N groups) to gain 1 bit for all members. The group that yields the most error reduction is chosen (lines 26-27 in Algorithm 1). This is iterated until the program satisfies the accuracy requirement. At this point (after the first iteration in $IterativeSearch$), one can simply stop the search to return a workable result P_1 with some redudant bits.

To reduce the redundant bits, we chose to reuse the ISOLATED DOWNWARD(MWL_1, P_1) to find MWL_2, which is the minimum precision for each variable when the other precisions are in the P_1 just obtained. We then compute another feasible $P_2 =$ GROUPED UPWARD(MWL_2). The process iterates until convergence, i.e., when $\text{MWL}_k = P_k$ or $Sum(P_k) = Sum(P_{k-1})$.

Unlike `Min+b`, we use actual error propagation information from the source code. This is how we form the *Grouped Upward* procedure. In the *Grouped Upward* procedure, instead of running the program $\binom{N}{b}$ times to get P_{min}, we only need to run N copies that can be parallelized. The cost for this simplification is that redundant bits will be added to some of the variables (because $\textsc{IncGroupPrec}(P, i)$ will increase precision of the whole influence group $\mathcal{G}[i]$ in P by 1 bit), which necessitates the iterative process to eliminate the redundant bits.

To get a better idea of the quality of our search result, we reimplemented a version of *Max-1* in [5], and used it to test on the DSP programs in our benchmarks set as that algorithm does not scale for the others of our benchmarks. The average number of bits reported by our algorithm is 6% fewer than *Max-1* in the cases we tested. In other words, our search result is comparable to an established search algorithm for wordlength optimization. Parallel search strategies have also been implemented on GPUs [15] with additional heuristics for pruning. The major issue of their approach is that GPU threads are lightweight and cannot handle the complex programs that we are targeting. The programs we tested are larger than the three largest bechmarks in [15], and yet our algorithm only needs less than a second to complete.

C. Statistically guided refinement for DSP programs

Using the above algorithm, the required precision vector for a given input can be computed. For real programs, inputs are often come from specific ranges instead of single fixed values. Intuitively, the worse input in the given range should be used for tuning. However, finding the worse input in a given range is nontrivial, even if automated [6]. We shall now present a statistically guided process to refine the result obtained from our search algorithm that does not need the worst case input.

C.1 Percentile and average error refinement

We further refine the search result iteratively to ensure one of two common statistical features, namely either the average *Signal-to-Quantization-Noise Ratio* (SQNR), or the 5th percentile SQNR. Instead of spending effort on rare corner cases of the inputs, we try to ensure the majority of the outputs will not fall below a certain quality. The important parameter for this refinement process is M, the size of the training set for the iterative process. We choose M to be not so large that the entire process is too slow. Neither can it be so small that the results are unreliable when generalizing beyond the training set.

The process begins with selecting a random input and using our search procedure to get the precision vector P. We then run the program using the P to extract the features of M inputs to find out which is the worst input in the training set that caused the lowest SQNR. Let's call that *worst_input_1*. After that, we invoke another search on the *worst_input_1* to get P_{temp} which satisfies *worst_input_1*. Then we form the new P_{refined} by comparing P and P_{temp} and keeping the maximum precision for each variable. Using the new refined precision, we repeat the process using M inputs again, and search for the worst input using the current precision. In the Algorithm 2, the *worst_seed* corresponds to *worst_input* because we use the seed value to generate input. In the DSP programs we tested, with M = 100, the refinement process finished after 5-6 epochs. The refined precision still satisfied the statistical feature when we tested against 100,000 other random inputs beyond the training set.

Algorithm 2 Average SQNR refinement

```
1:  procedure AVERAGE REFINEMENT($M$)
2:      $Worst\_seed \leftarrow Random\_value\_in[0, 1, \ldots, M-1]$
3:      $P_{\text{refined}} \leftarrow [0, 0, \ldots, 0]$        ▷ assign 0 for each variable
4:      repeat
5:          $Program_k \leftarrow Program(Worst\_seed)$
6:          $P_{\text{temp}} \leftarrow IterativeSearch$ on $Program_k$
7:          for $i \leftarrow 0, Length(P_{\text{temp}})$ do
8:              $P_{\text{refined}}[i] \leftarrow Max(P_{\text{refined}}[i], P_{\text{temp}}[i])$
9:          end for
10:         for $j \leftarrow 0, M-1$ do                ▷ MPI_Parallel
11:             Run Program(j), Record $SQNR_j$
12:         end for
13:         Find $Worst\_seed \leftarrow$ j causes lowest SQNR
14:     until $Average(SQNR_j) \geq Expected\_Avg$
15:     return $P_{\text{refined}}$
16: end procedure
```

V. Applications and evaluation

A. Software precision tuning

First, we shall compare the search result with the enhanced version of *Precimonious* [22]. Because our analysis produces the precision value down to the number of bits, we mapped the results back to either single or double precision for the comparison. We choose to reproduce the result of 5 programs have the analysis results published online by the *Precimonious* team[2]. We used the same input files and error metrics to analyze the same set of variables for each program. For a fairer comparison, we did not use the refinement process for this first set of experiments. To test the scalability of the algorithm, we also analyze larger and more complicated programs from SPEC CPU2006, PARSEC and Rodinia benchmarks. The required number of double-precision variables for 4 different error thresholds are presented in Table I. The *Init* column is the number of double-precision variables in the original code, the *B+P* columns contain the result of the latest version of *Precimonious*. Our search results are in the *D* columns. For all the tables in this section, the cells with '-' indicate that either (1) the original benchmarks do not have that high accuracy, or (2) the exact results are provided in the benchmarks and the original version needs more precision than `double` to satisfy the particular accuracy requirement. Speedup value entries of '-' indicates that the original version performed better or just as well as our tuned versions. Only the results of those programs where gain performance were obtained are shown in Table II because in some mixed precision versions of the programs, the compiler needs to implicitly add data type conversion from `float` to `double`, and vice versa. This adds to the overhead at the machine code level and may diminish, or even eliminate the speedup gained in other single-precision operations.

[2]https://github.com/corvette-berkeley

TABLE I

: Number of double variables

		$\epsilon = 10^{-4}$		$\epsilon = 10^{-6}$		$\epsilon = 10^{-8}$		$\epsilon = 10^{-10}$	
	Init	B+P	D	B+P	D	B+P	D	B+P	D
ep	45	42	12	42	13	42	19	-	19
cg	32	2	0	13	3	16	7	16	19
polyroots	31	13	0	13	4	13	13	13	13
sum	34	11	0	11	6	11	8	24	17
blas	17	0	0	0	0	10	5	10	9
lbm	30	-	1	-	10	-	8	-	8
myocyte	417	-	7	-	19	-	96	-	189
blackscholes	35	-	2	-	6	-	-	-	-

TABLE II

: Speedup (%) of the tuned programs

	$\epsilon = 10^{-4}$		$\epsilon = 10^{-6}$		$\epsilon = 10^{-8}$		$\epsilon = 10^{-10}$	
	B+P	D	B+P	D	B+P	D	B+P	D
ep	-	-	-	-	-	-	-	-
cg	7.1	15.4	7.9	15.4	7.3	17.1	7.9	-
polyroots	-	4.8	-	-	-	-	-	-
sum	41.5	49.5	41.5	44.3	41.5	44.3	-	31.7
blas	5.1	5.1	5.1	5.1	-	-	-	-
lbm	-	63.1	-	-	-	-	-	-

Table II shows the speedup of the tuned version that had a reduced number of double-precision variables. To measure the speedup, we ran each program 5 times and took the average execution time. All the program are compiled by `gcc -O2`. We observe that the speedup results vary within the range of $\pm 1\%$ of the means. We also run the version of the programs tuned by Precimonious using the same setup. The results show that our tuned programs performed just as well as, and sometimes even better than, *Precimonious*. The performance impact on the floating-point programs is similar across the tools. Some programs can take advantage of our tool to gain as much as 63.1% in performance.

B. Discussion

In Figure 1, we present the aggregated precision result across all the programs we tested. We group the precision of all the variables into 12 bins. A bin labelled p covers the precision range of $[p, p+3]$ bits. The vertical axis gives the number of variables has precision results falling into the range associated with the bins' label on the horizontal axis. We plotted four series with all the values from 4 different error thresholds reported. The total number of variables is 2,666. This graph yields an interesting observation on the mantissa precision required for floating point programs. The majority of the variables only needs the precision of 4 to 7 bits ($> 50\%$ for $\epsilon = 10^{-4}$, $\approx 30 - 40\%$ for other thresholds). From our experiments, the following conclusion can be drawn: as the accuracy requirement increases, the precision of only a small number of variables needs to be increased. For the rest of the variables, there is no need for anything beyond single-precision. Another interesting observation is that, if we have the hardware support half-precision for these programs, there will be around 66%, 52%, 38% and 31% of variables can be converted to half-precision with an accuracy of 10^{-4},10^{-6},10^{-8} and 10^{-10}, respectively. The insights from the aggregated result point to a

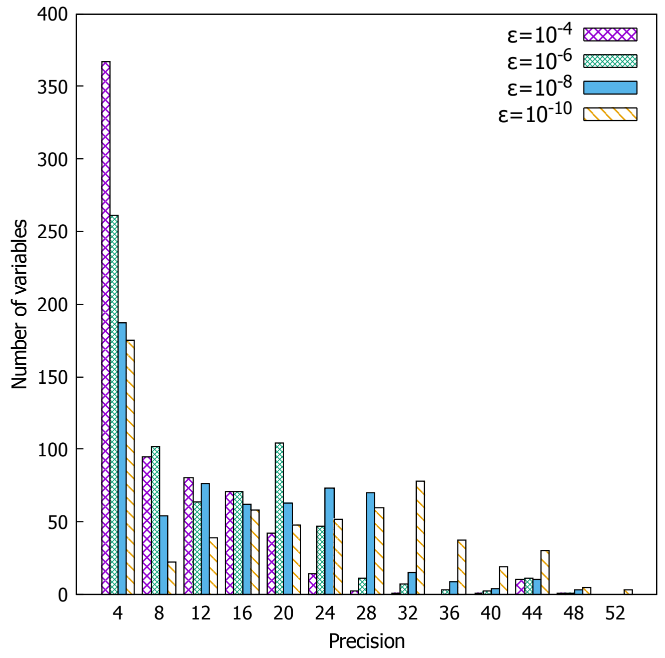

Fig. 1.: Histogram of the precision required by all floating-point benchmarks we tested on for various error threshold (ϵ)

promising future for approximate computing, especially using variable floating-point hardware.

C. Overhead Cost

To measure the searching overhead, we ran the search on a cluster of servers has $2\times$ Intel® Xeon® E5-2690. We used the number of processors equals number of variables. We first measured T_{mpfr}, the execution time of the MPFR versions used for the search algorithm as the baseline and we measured the overhead by the ratio of the total searching time to T_{mpfr}. Except for the largest program, namely `myocyte`, the algorithm needs, on average, to run the input program 25.8 times to yield the final precision result. The fastest run takes only $9 \times T_{\text{mpfr}}$. In case of `myocyte` which has 417 variables, the algorithm converges after $110.5 \times T_{\text{mpfr}}$ on average. The overhead result shows that we can expect the overhead to be approximately $\leq N \times T_{\text{mpfr}}$ in practice. When the search algorithm is integrated into the toolchain, we have to take the overhead of MPFR arithmetic into account when considering the running time of the toolchain. Although we observed an insignificant overhead of MPFR arithmetic on most of the experiments, this overhead very much depends on the original code. Among 14 programs we tested (including 6 DSP programs in Section D), there are 9 programs had the total tuning time less than 1 second. Two experiments belonging to `Blackscholes` completed in 2.2 seconds, on average. The `myocyte` experiments ran in 1 minute 45 seconds on average. The three programs, namely `ep`, `cg` and `lbm`, had T_{mpfr} compared to the native program were $58\times$ slower ($T_{\text{mpfr}} \approx 11.5$ minutes) on average. The overhead can be attributed to our automatic transformation process preventing the compiler from otherwise optimizing the code. Because of the overhead in MPFR arithmetic, the average processing time of `lbm`, `cg`, `ep` were 303 minutes on

TABLE III

: Comparison of resources consumed and execution time (clock cycles x clock period) for single (SP), double (DP) and arbitrary precision (AP) implementations.

Case study	Hardware resource consumption				Execution time (ns)			SNR
	HW	SP	DP	AP	SP	DP	AP	(dB)
FFT16	DSPs	72	90	10				
	FFs	7991	8351	1356	430	437	182	62
	LUTs	8150	10014	3365				
Chebyshev	DSPs	19	17	2				
	FFs	4186	4084	647	1269	1742	1019	65
	LUTs	5559	5661	1096				
2x2 Matrix multiply	DSPs	24	62	7				
	FFs	2761	5495	366	187	233	61	60
	LUTs	2389	6564	813				
FIR filter	DSPs	5	14	2				
	FFs	543	1097	98	2346	3149	1039	61
	LUTs	476	1213	171				
IIR filter	DSPs	2	3	1				
	FFs	231	450	20	31	35	8	63
	LUTs	215	782	98				
8x8 DCT	DSPs	31	79	10				
	FFs	3395	6744	1833	5198	6178	2958	56
	LUTs	3474	8843	4604				
Average	all	6587	9927	2417	1577	1962	877	62
Ave. AP Impr. (%)		66%	82%		53%	62%		

average. It is possible to optimize the MPFR programs to reduce the arithmetic overhead. Alternatively, we can use other more efficient libraries. This is left as future work.

D. Fixed-point tuning

We also extended our tool-chain for fixed point conversion via the Xilinx Vivado HLS Design Suite. Successful fixed point precision tuning suggests that our search technique can be used uniformly across floating point and fixed point designs. We evaluated the hardware resource consumption using six case studies in Table III by converting the floating point to fixed point designs targeting the Xilinx Kintex 7 xc7k160tfbg484-1 FPGA. As future work, we plan to use our tool for RTL synthesis of floating point operators from Xilinx LogiCORE IP [2] or FloPoCo [11]. The bitwidth (BW) of a fixed point number is the sum of the integer bitwidth (IBW) and the fractional bitwidth (FBW). IBW is calculated using a profile-based statistical scaling procedure as described in [16] where the dynamic range information is computed by measuring the mean, variance and standard deviation of the variables. These are then propagated up in a bottom up fashion. The dynamic range is estimated using the relation,

$$R(x) = max\{(|\mu(x)| + n \times \sigma(x)), Amax|x|\}$$

where $\mu(x)$, $\sigma(x)$ and $Amax|x|$ is the average, standard deviation and absolute maximum value, respectively, of a given variable x. Unlike [16] where a larger value of n that overestimates the range is used, we set $n = 1$. Our measured ranges for up to 1000 random numbers generated within our predefined input range showed that no overflow will occur. For FFT16,FIR and IIR filters, we use the input dynamic range of $(-1, 1)$, and the input to the 2×2 matrix are in the range $(0, 1)$. For Chebyshev approximation, we implemented the approximation of the function $\sin(x)$, the coefficients and the value of x are in the range $(-1, 1)$ and $[0, \pi]$ respectively. The IBW of the fixed point designs are calculated from the range results using,

$$\text{IBW}_x = \lceil log_2(\text{R}(x)) \rceil + \alpha, \alpha = \begin{cases} 1, & \text{frac}(log_2(x_{max})) \neq 0 \\ 2, & \text{frac}(log_2(x_{max})) = 0 \end{cases}$$

where x_{max} is the maximum value of the variable x observed during profiling [18]. The precision results obtained from our presented search algorithm and average SQNR refinement process are used as the FBW. The SQNR error metric is used to evaluate the accuracy of the arbitrary precision output against the original double precision code. Unlike most of previous works [7] that aims to achieve a higher SQNR value by using higher target precisions, our aim is to reduce the resource consumption whilst staying within a given accuracy bound for the results. Most previous works used an average SQNR value of 60 to 80dB. For this reason, we fixed the error bound of our floating point precision search to be within these two values. After conversion to fixed point, we obtained an SQNR of around 56 to 65 dB.

VI. CONCLUSIONS

We presented an algorithm for finding the minimum mantissa precision in floating point code assuming bounds on the output error are given. The algorithm's novelty lies in the use of program's high-level structure information to guide the blackbox search in such a way that is both scalable and yet produces high quality results. The proposed search algorithm is not only fast and parallelizable, but also produces results comparable to that obtained by fine-grain wordlength optimization methods. It has been implemented in a toolchain, and we have shown how its arbitrary precision results can be used in both optimizing floating point code as well as for fixed point conversion. This provides a new alternative for the latter. For smaller digital signal processing programs, our tool runs in less than a second. We plan to extend the work using more efficient multiprecision libraries, as well as other optimizations to further improve the tool's end-to-end delay. The tool is available under MIT license [1].

REFERENCES

[1] `https://github.com/minhhn2910/fpPrecisionTuning`.

[2] LogiCORE IP floating-point operator v7.0.

[3] A. W. Brown, P. H. Kelly, and W. Luk. Profiling floating point value ranges for reconfigurable implementation. In *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, pages 6–16, 2007.

[4] I. Buck. nVidia's next-gen pascal gpu architecture to provide 10x speedup for deep learning apps., 2015.

[5] M.-A. Cantin, Y. Savaria, and P. Lavoie. A comparison of automatic word length optimization procedures. In *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, volume 2, pages II–612. IEEE, 2002.

[6] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *ACM SIGPLAN Notices*, volume 49, pages 43–52. ACM, 2014.

[7] J. Chung and L.-W. Kim. Bit-width optimization by divide-and-conquer for fixed-point digital signal processing systems. *Computers, IEEE Transactions on*, 64(11):3091–3101, 2015.

[8] G. A. Constantinides, P. Y. Cheung, and W. Luk. Wordlength optimization for linear digital signal processing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(10):1432–1442, 2003.

[9] M. Courbariaux, Y. Bengio, and J.-P. David. Low precision arithmetic for deep learning. *arXiv preprint arXiv:1412.7024*, 2014.

[10] F. De Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *Computers, IEEE Transactions on*, 60(2):242–253, 2011.

[11] F. De Dinechin and B. Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 4(28):18–27, 2011.

[12] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[13] A. A. Gaffar, O. Mencer, and W. Luk. Unifying bit-width optimisation for fixed-point and floating-point designs. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 79–88. IEEE, 2004.

[14] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *arXiv preprint arXiv:1502.02551*, 2015.

[15] N. Kapre and D. Ye. GPU-accelerated high-level synthesis for bitwidth optimization of FPGA datapaths. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 185–194. ACM, 2016.

[16] S. Kim, K.-I. Kum, and W. Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 45(11):1455–1464, 1998.

[17] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 369–378. ACM, 2013.

[18] D.-U. Lee, A. A. Gaffar, R. C. Cheung, O. Mencer, W. Luk, G. Constantinides, et al. Accuracy-guaranteed bit-width optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):1990–2000, 2006.

[19] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk. MiniBit: bit-width optimization via affine arithmetic. In *Proceedings of the 42nd annual Design Automation Conference*, pages 837–840. ACM, 2005.

[20] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu. Joint precision optimization and high level synthesis for approximate computing. In *Proceedings of the 52nd Annual Design Automation Conference*, page 104. ACM, 2015.

[21] K. Nepal, Y. Li, R. Bahar, and S. Reda. Abacus: A technique for automated behavioral synthesis of approximate computing circuits. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 361. European Design and Automation Association, 2014.

[22] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1074–1085. ACM, 2016.

[23] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 27. ACM, 2013.

[24] Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate computing: A survey. *IEEE Design & Test*, 33(1):8–22, 2016.