An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products

Florent de Dinechin, Bogdan Pasca, Octavian Creţ, Radu Tudoran

AN FPGA-SPECIFIC APPROACH TO FLOATING-POINT ACCUMULATION AND SUM-OF-PRODUCTS

Florent de Dinechin, Bogdan Pasca *

LIP (CNRS/INRIA/ENS-Lyon/UCBL)
École Normale Supérieure de Lyon
Université de Lyon
Florent.de.Dinechin@ens-lyon.fr

Octavian Creţ, Radu Tudoran

Computer Science Department
Technical University of Cluj-Napoca
Octavian.Cret@cs.utcluj.ro

ABSTRACT

Floating-point operators on FPGAs do not have to be identical to the ones available in processors. This article studies two common situations where the flexibility of FPGAs allows one to design application-specific floating-point operators. First, for applications involving the addition of a large number of floating-point values, an ad-hoc accumulator is proposed. By tailoring its parameters to the numerical requirements of the application, it can be made arbitrarily accurate, at an area cost comparable in practice to a standard floating-point adder, and at a higher frequency. The second example is the sum-of-product operation, which is the building block of matrix computations. An architecture is proposed based on the previous accumulator and an ad-hoc rounding-free multiplier. These architectures are implemented within the FloPoCo generator, freely available under the GPL.

1. INTRODUCTION

Most general-purpose processors have included floating-point (FP) units since the late 80s, following the IEEE-754 standard. The feasibility of FP on FPGA was studied long before it became a practical possibility [14, 8, 10]. As soon as the sizes of FPGAs made it possible, many libraries of floating-point operators were published (see [1, 7, 9, 13] among other). FPGAs could soon provide more FP computing power than a processor in single precision [9, 13], then in double-precision [15, 4, 3]. Here single precision (SP) is the standard 32-bit format consisting of a sign bit, 8 bits of exponent and 23 bits of significand (or mantissa), while double-precision (DP) is the standard 64-bit format with 11 bits of exponent and 52 significand bits. Since then, FPGAs have increasingly been used to accelerate scientific, financial and other FP-based computations. This acceleration is essentially due to massive parallelism: Basic FP operators in an FPGA are typically slower than their processor counterparts by one order of magnitude.

Most of the aforementionned applications are very close, from the arithmetic point of view, to their software implementations. They use the same basic operators, although the internal architecture of the operators may be highly optimized for FPGAs [11]. Most published FP libraries are fully parameterizable in significand length and exponent length, but applications that exploit this flexibility are rare [13, 15].

The FloPoCo project[1] studies how the flexibility of the FPGA target can be better exploited in the floating-point realm. In particular, it looks for operators which are radically different from those present in microprocessors. In the present article, such operators are presented for the ubiquitous operation of floating-point accumulation, and applied to sums of products.

All the results in this article are obtained for Virtex4, speedgrade -12, using ISE9.1, but the goal of FloPoCo is to produce portable VHDL.

2. ACCUMULATION

Summing many independent terms is a very common operation. Scalar product, matrix-vector and matrix-matrix products are defined as sums of products. Numerical integration usually consists in adding many elementary contributions. Monte-Carlo simulations also involve sums of many independent terms. Many other applications involve accumulations of floating-point numbers.

If the number of summands is small and constant, one may build trees of adders, but to accomodate the general case, it is necessary to design an iterative accumulator, illustrated by Figure 1.

It is a common situation that the error due to the computation of one summand is independent of the other summands and of the sum, while the error due to the summation

*This work was partly supported by the XtremeData university programme, the ANR EVAFlo project and the Egide Brâncuşi programme 14914RL.

[1] www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

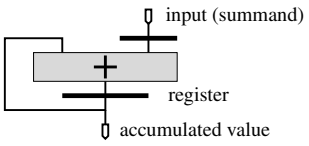

Fig. 1. Iterative accumulator

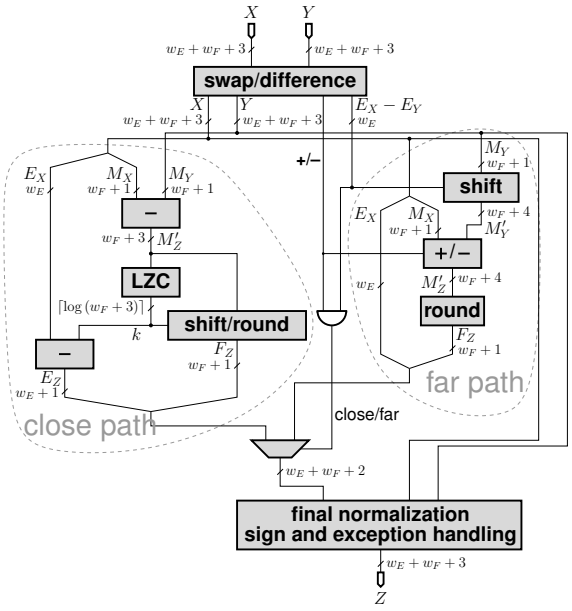

Fig. 2. A typical floating-point adder (w_E and w_F are respectively the exponent and significand sizes)

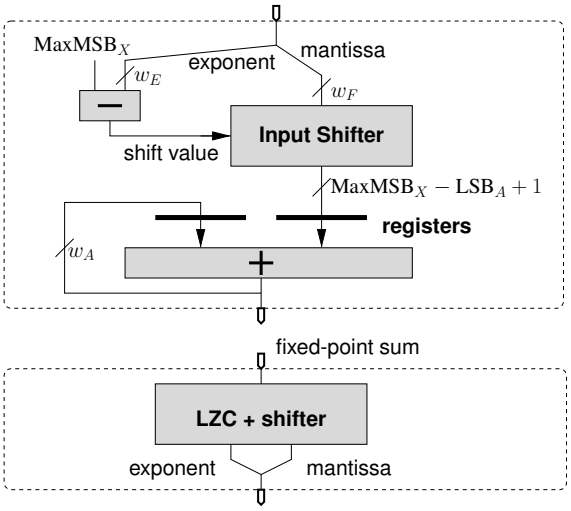

Fig. 3. The proposed accumulator (top) and post-normalisation unit (bottom). Only the registers on the accumulator itself are shown. The rest of the design is combinatorial and can be pipelined arbitrarily.

grows with the number of terms to sum. This happens in integration and sum of products, for instance. In this case, it makes sense to have more accuracy in the accumulation than in the summands.

A first idea is to use a standard FP adder, possibly with a larger significand than the summands. The problem is that an FP adder has a long latency: typically $l = 3$ cycles a processor, up to tens of cycles in an FPGA. This is explained by the complexity of their architecture, illustrated on Figure 2. This means that an accumulator based on them will either add one number every l cycle, or compute l independent sub-sums which have then to be added together somehow.

Luo and Martonosi [12] have described an architecture that avoids this problem. The techniques presented in the sequel are both a simplification and a generalization of their pioneering work. The main idea is that an accumulator built out of a floating-point adder is inefficient because the significand of the accumulator has to be shifted, sometimes twice (first to align both operands and then to normalize the result, see Figure 2). These shifts are in the critical path of the loop of Figure 1. A solution to avoid shifting the accumulated result is to keep it in fixed-point. Luo and Martonosi use such accumulators, of size 64-bit, for large intervals of exponents, chosen in such a way that the accumulator is never shifted. However it may sometimes overflow, in which case their architecture stalls. To avoid this overflow detection in the critical path, they suggest to impose a limit on the number of summands to add. The main problem with this design is that it scales poorly beyond single-precision.

In this paper, we suggest to build a fixed-point accumulator which is tailored to the numerics of each application in order to ensure that 1/ its significand never needs to be shifted, 2/ it never overflows and 3/ it eventually provides a result that is as accurate as the application requires. We also show that it can be clocked to any frequency that the FPGA supports.

3. AN APPLICATION-SPECIFIC ACCUMULATOR

3.1. Overall accumulator architecture

The proposed accumulator architecture, depicted on Figure 3, removes all the shifts from the critical path of the loop by keeping the current sum as a large fixed-point number. Figure 4 illustrates the accumulation of several floating-point numbers (represented by their significands shifted by their exponent) into such an accumulator.

There is still a loop, but it is now a fixed-point addition for which current FPGAs are highly efficient. Specifically, the loop involves only the most local routing, and the fast-carry logic of current FPGAs provides good performance up

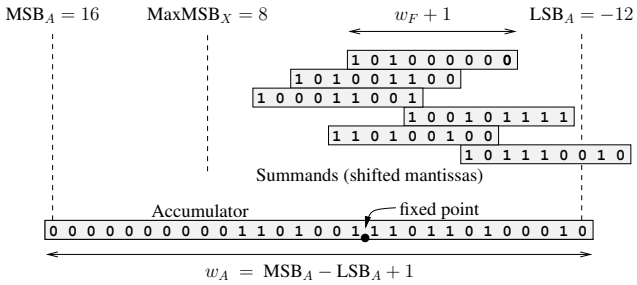

Fig. 4. Accumulation of floating-point numbers into a large fixed-point accumulator

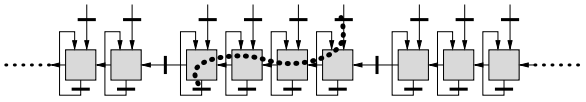

Fig. 5. Accumulator with 4-bit partial carry-save. The boxes are full adders, bold dashes are 1-bit registers, and the dots show the critical path.

to 64-bits (or better than DP accuracy). For instance, a Virtex4 with speed grade -12 runs such an accumulator at more than 220MHz, while consuming only 64 CLBs. Section 3.2 will show how to reach even larger frequencies and/or accumulator sizes.

The shifters now only concern the summand (see Figure 3), and, being combinatorial, can be pipelined as deep as required by the target frequency. The FloPoCo VHDL generator uses a simplified model of the target FPGA to build such a pipeline automatically for a given frequency.

The accumulated result needs to be converted back to floating-point. This normalization consists, as in the FP adder, in leading-zero counting and shifting, followed by rounding. It may be performed at each cycle and pipelined arbitrarily. However, most applications won't need all the intermediate sums. Some will output the fixed-point accumulator and leave the final normalization to software. Another common case is that a single post-normalization unit may be shared by several accumulators (matrix operations can be scheduled this way). Therefore, it makes sense to provide this final normalizer as a separate component, as shown by Figure 3.

Note that an accumulator based on an FP adder of latency l exposes a similar issue: it will also need either a tree of FP adders, or a lot of extra logic and registers to add together the l sub-sums it computes. However this final sum may also be computed in software at no extra hardware cost. To sidestep this discussion, in the following, we choose to provide area results only for the accumulators themselves. As a rule of thumb, the size of a post-normalisation unit is roughly equal to that of its accumulator.

Many details are missing from Figure 3. The accumulator stores a two's complement number while the summands use a sign/magnitude representation, and thus need to be converted to two's complement, which involves a carry propagation. The post-normalization unit also has to convert back to sign/magnitude. The main point is that none of this is on the critical path of the loop.

Let us now discuss, with the help of Figure 4, the parameters of this architecture. w_E and w_F are the exponent and significand size of the summands The weight MSB$_A$ of the most-significant bit (MSB) of the accumulator has to be larger than the maximal expected result, which will ensure that no overflow ever occurs. The weight LSB$_A$ of its least-significant bit can be arbitrarily low, and will determine the final accuracy as Section 3.3 will show. MaxMSB$_X$ is the maximum expected weight of the MSB of a summand. MaxMSB$_X$ may be equal to MSB$_A$, but very often one is able to tell that each summand is much smaller in magnitude than the final sum. In this case, providing MaxMSB$_X <$ MSB$_A$ will save hardware in the input shifter.

The main claim of the present article is the following: For most applications accelerated using an FPGA, values of MSB$_A$ and LSB$_A$ can be determined a priori, using a rough error analysis or software profiling, that will lead to an accumulator smaller and more accurate than the one based on an FP adder. This claim will be justified in 3.3, and 3.4 will illustrate it on actual applications.

This claim sums up the essence of the advantage of FPGAs over the fixed FP units available in processors, GPUs or dedicated floating-point accelerators: we advocate an accumulator specifically tailored for the application to be accelerated, something that would not be possible or economical in an FPU.

3.2. Fast accumulator design

If the fast-carry logic of the FPGA is not enough to reach the target frequency, a partial carry-save representation allows to reach any arbitrary frequency supported by the FPGA. As illustrated by Figure 5, the idea is to cut the large carry propagation into smaller chunks of k bits ($k = 4$ on the figure), simply by inserting $\lfloor(\text{MSB}_A - \text{LSB}_A)/k\rfloor$ registers. The critical path is now that of a k-bit addition, and the value of k can therefore be chosen to match the target frequency. The additional hardware cost is just the few additional registers – 1/4 more in our figure. This is a classical technique which was in particular suggested by Hossam, Fahmy and Flynn [5] for use as an internal representation in processor FPUs. For $k = 1$ one obtains a standard carry-save representation, but larger values of k are prefered as they take advantage of fast-carry logic while reducing the register overhead.

Of course a drawback of the partial carry-save accumu-

lator is that it holds its value in a non-standard redundant format. To convert to standard notation, there are two options. One is to dedicate $\lfloor(\text{MSB}_A - \text{LSB}_A)/k\rfloor$ cycles at the end of the accumulation to add enough zeroes into the accumulator to allow for carry propagation to terminate. This comes at no hardware cost. The other option, if the running value of the accumulator is needed, is to perform this carry propagation in a pipelined way before the normalisation. The important fact is again that this carry propagation is outside of the critical loop.

3.3. Setting up the accumulator's parameters

Let us now justify our claim. First note that a designer has to provide a value for MSB_A and MaxMSB_X, but these values do not have to be accurate. For instance, adding 10 bits of safety margin to MSB_A has no impact on the latency and very little impact on area. Now from the application point of view, 10 bits mean 3 orders of magnitude: it is huge. A designer in charge of implementing a given computation on FPGA is expected to understand it well enough to bound the expected result with a margin of 3 orders of magnitude. An actual example is detailed below in 3.4. As another example, in Monte-Carlo simulations of financial markets, each accumulation computes an estimate of the value of a share. No share will go beyond, say, $100,000 before something happens that makes the simulation invalid anyway.

It may be more difficult to evaluate MaxMSB_X. In doubt, $\text{MaxMSB}_X = \text{MSB}_A$ will do, but the accumulator will be larger than needed. A better option is profiling: a typical instance of the problem may be run in software, instrumented to output the max and min of the absolute values of summands. Again, the possibility of adding 10 bits of more of margin makes this approach both safe and easy.

In some cases, the application will dictate MaxMSB_X but not MSB_A. In this case, one has to consider the number n of terms to add. Again, one will usually be able to provide an upper bound, be it the extreme case of 1 year running at 500MHz, or 2^{53} cycles. In a worst-case scenario on such simulation times, this suggests the relationship $\text{MSB}_A = \text{MaxMSB}_X + 53$ to avoid overflows. For comparison, 53 is the precision of a DP number, so the cost of this worst case scenario is simply a doubling of the accumulator itself, *but not of the input shifter* which shifts up to MaxMSB_X only. It will cost just slightly more than 53 slices in the accumulator. It may cost more in the post-normalisation unit if one is built.

The last parameter, LSB_A, allows a designer to manage the tradeoff between precision and area. First, remark that if a summand has its LSB higher than LSB_A (case of the 5 topmost summands on Figure 4), it is added exactly, entailing no rounding error. Therefore, the proposed accumulator will be exact (entailing no roundoff error whatsoever) if the accumulator size is large enough so that its LSB is smaller than those of all the inputs. Conversely, if a summand has an LSB smaller than LSB_A (case of the bottommost summand on Figure 4), adding it to the accumulator entails a rounding error of at most 2^{LSB_A-1}. In the worst case, when adding n numbers, this error will be multiplied by n and invalidate the $\log_2 n$ lower bits of the accumulator – note that this worst-case situation is antagonist to the one leading to the worst-case overflow. A designer may lower LSB_A to absorb such errors, an example is given in next section. Again, a practical maximum is an increase of 53 bits for 1 year of computation at 500MHz. However, in a global sense, it doesn't make much sense to consider this worst case situation, because when a summand is added exactly, it is usually the result of some rounding, so it carries an error of the order of its LSB, which it adds to the accumulator (by not adding bits lower than its LSB). These errors, which are not due to the accumulator, will typically dwarf the rounding errors due to the accumulator. They can be reduced by increasing w_F, but this is of course outside of the scope of this article.

All considered, it is expected that no accumulator will need to be designed larger than 100 bits.

Finally, the proposed accumulator has sticky output bits for overflows in the summands and in the accumulator, so the validity of the result can be checked a posteriori.

3.4. Case studies

In the inductance computation of [2], physical expertise tells that the sum will be less than 10^5 (using arbitrary units due to factoring out some physical constants), while profiling showed that the absolute value of a summand was always between 10^{-2} and 2.

Converting to bit weight, and adding two orders of magnitude (or 7 bits) for safety in all directions, this defines $\text{MSB}_A = \lceil\log_2(10^2 \times 10^5)\rceil = 24$, $\text{MaxMSB}_X = 8$ and $\text{LSB}_A = -w_F - 15$ where w_F is the significand width of the summands. For $w_F = 23$ (SP), we conclude that an accumulator stretching from $\text{LSB}_A = -23 - 15 = -38$ (least significant bit) to $\text{MSB}_A = 24$ (most significant bit) will be able to absorb all the additions without any rounding error: no summand will add bits lower than 2^{-38}, and the accumulator is large enough to ensure it never overflows. The accumulator size is therefore $w_A = 24 + 38 + 1 = 63$ bits.

Remark that only LSB_A depends on w_F, since the other parameters (MSB_A and MaxMSB_X) are related to physical quantities, regardless of the precision used to simulate them. This illustrates that LSB_A is the parameter that allows one to manage the accuracy/area tradeoff for an accumulator.

Table 1 compares for accuracy and performance the proposed accumulator to one built using a floating-point adder from FPLibrary[2]. To evaluate the accuracies, we computed the exact sum using multiple-precision software on a small

[2] http://www.ens-lyon.fr/LIP/Arenaire/Ware/FPLibrary/

	accuracy	area	frequency
SP FP adder acc	$1.2 \cdot 10^{-3}$	425 sl	214MHz
DP FP adder acc	$2.8 \cdot 10^{-15}$	901 sl	140MHz
proposed acc	$2 \cdot 10^{-16}$	266 sl	200 Mz

Table 1. Compared performance and accuracy of different accumulators for SP summands from [2].

sum size	rel. error for unif[0, 1]		rel. error for unif[-1, 1]	
	FP adder	long acc.	FP adder	long acc.
1000	-5.76e-05	1.05e-07	-1.59e-05	1.40e-04
10,000	-2.74e-04	1.07e-08	-3.04e-04	2.36e-04
100,000	-4.31e-04	1.07e-09	2.54e-03	-2.73e-04
1,000,000	-0.738	-3.57e-09	3.18e-03	-4.47e-05

Table 2. Accuracy of accumulation of FP(7,16) numbers, using an FP(7,16) adder, compared to using the proposed accumulator with 32 bits ($MSBA = 20$, $LSBA = -11$).

run (20,000,000 summands), and the accuracy of the different accumulators was computed with respect to this exact sum. The proposed accumulator is both smaller, faster and more accurate than the ones based on FP adders. This table also shows that for production runs, which are 1000 times larger, a single-precision FP accumulator will not offer sufficient accuracy.

Table 2 provides other examples of the final relative accuracy, with respect to the exact sum, obtained by using an FP adder, and using the proposed accumulator with twice as large a significand. In the first column, we are adding n numbers uniformly distributed in [0,1]. The sum is expected to be roughly equal to $n/2$, which explains that the result becomes very inaccurate for $n = 1,000,000$: as soon as the sum gets larger than 2^{17}, any new summand in [0,1] is simply shifted out and counted for zero. This problem can be anticipated by using a larger significand, or a larger MSBA in the accumulator as we do. In the second column, numbers are uniformly distributed in [-1,1]. The sum grows as well (it is a random walk) but much more slowly. As we have taken a fairly small accumulator ($\text{LSB}_A = -11$), for the first sums floating-point addition is more accurate: while the sum is smaller than 1, its LSB is smaller than -16. However, as more numbers are added, the sum grows. More and more of the bits of a summand are shifted out in the FP adder, but kept in the long accumulator, which becomes more accurate. Note that by adding only 5 bits to it ($\text{LSB}_A = -16$ instead of -11), the relative error becomes smaller than 10^{-10} in all cases depicted in Table 2: Again, LSB_A is the parameter allowing to manage the accuracy/area tradeoff.

We have discussed in this section only the error of the long fixed-point accumulator itself (the upper part of Fig. 3). If its result is to be rounded to an FP(7,16) number using the post-normalisation unit of Figure 3, there will be a relative rounding error of at most $2^{-17} \approx 0.76e-5$. Comparing this value with the relative errors given in Table 2, one concludes that the proposed accumulator, with the given parameters, always leads to a result accurate to the two last bits of an FP(7,16) number.

(w_E, w_F)	adder	long acc, $2w_F$	long acc, $3w_F$
(7,16)	318 sl, 216 MHz	146 sl, 373 MHz	182 sl, 317 MHz
(8,23) SP	425 sl, 214 MHz	169 sl, 265 MHz	265 sl, 214 MHz
(10,37)	644 sl, 153 MHz	300sl, 209 MHz	379 sl, 181 MHz
(11,52) DP	901 sl, 140 MHz	396 sl, 182 MHz	585 sl, 146 MHz

Table 3. Compared synthesis results for an accumulator based on FP adder versus proposed accumulator.

Finally, Table 3 illustrates the performance of the proposed accumulator (using simple carry-propagate architecture, without resorting to the techniques presented in 3.2) compared to one built using a floating-point adder. The results are given for accumulators with twice and three times the significand size of the summands, and with $\text{MaxMSB}_X = (\text{MSB}_A - \text{LSB}_A)/2$. These results are for illustration only: Again, an accumulator should be built in an application-specific way.

4. ACCURATE SUM-OF-PRODUCTS

We now extend the previous accumulator to a highly accurate sum-of-product operator. The idea is simply to accumulate the exact results of all the multiplications. To this purpose, instead of standard multipliers, we use *exact* multipliers which return all the bits of the exact product: for $1 + w_F$-bit input significand, they return an FP number with a $2 + 2w_F$-bit significand. Such multipliers incur no rounding error, and are actually *cheaper* to build than the standard (w_E, w_F) ones. Indeed, the latter also have to compute $2w_F + 2$ bits of the result, and in addition have to round it. In the exact FP multiplier, results do not need to be rounded, and do not even need to be normalized, as they will be immediately sent to the fixed-point accumulator. There is an additional cost, however, in the accumulator, which requires twice a larger input shifter.

This idea was advocated by Kulisch [6] for inclusion in microprocessors, but a generic DP version requires a 4288 bits accumulator, which manufacturers always considered too costly to implement. On an FPGA, one may design an application-specific version with an accumulator of 100-200 bits only. This is being implemented in FloPoCo, and Table 4 provides preliminary synthesis results for single and double precision input numbers, with the same $3w_F$ accumulators as in Table 3. Work is in progress to improve the frequency of the multiplier using partial carry save. Then, the benefits of this approach remain to be tested in actual applications.

SP inputs	DP inputs
409 sl + 3 DSP48, 183 MHz	1557 sl + 9 DSP48, 140 MHz

Table 4. Results for the proposed Sum-Of-Product operator.

5. CONCLUSION AND FUTURE WORK

The accumulator design presented in this article perfectly illustrates the philosophy of the FloPoCo project: Floating-point on FPGA should make the best use of the flexibility of the FPGA target, not re-implement operators available in processors. By doing so, one comes up with a fairly simple design which can be tailored to be arbitrarily faster and arbitrarily more accurate than a naive floating-point approach, without requiring more resources.

This is done in an application-specific manner, and requires the designer to provide bounds on the orders of magnitudes of the values accumulated. We have shown that these bounds can be taken lazily. In return, the designer gets not only improved performance, but also a provably accurate accumulation process. We believe that this return is worth the effort, especially considering the overall time needed to implement a full floating-point application on an FPGA.

The challenge is now to integrate such advanced operators in the emerging C-to-FPGA compilers. A profiling-based approach might be enough to set up the proposed accumulator. In parallel, FloPoCo should be extended with tools that help a designer set up a complete floating-point datapath, managing synchronisation issues but also accuracy ones. Such tools are still at the drawing board.

6. REFERENCES

[1] P. Belanović and M. Leeser. A library of parameterized floating-point modules and their use. In *Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 657–666. Springer, 2002.

[2] O. Creţ, I. Trestian, R. Tudoran, L. Creţ, L. Văcariu, and F. de Dinechin. FPGA-based acceleration of the computations involved in transcranial magnetic stimulation. In *Southern Programmable Logic Conference*. IEEE, Mar. 2008.

[3] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Field-Programmable Gate Arrays*, pages 75–85. ACM, 2005.

[4] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Field-Programmable Gate Arrays*, pages 86–95. ACM, 2005.

[5] H. A. H. Fahmy and M. J. Flynn. The case for a redundant format in floating point arithmetic. In *16th Symposium on Computer Arithmetic*, pages 95–102. IEEE Computer Society, 2003.

[6] U. W. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, 2002.

[7] B. Lee and N. Burgess. Parameterisable floating-point operators on FPGAs. In *36th Asilomar Conference on Signals, Systems, and Computers*, pages 1064–1068, 2002.

[8] Y. Li and W. Chu. Implementation of single precision floating point square root on FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 56–65, Apr. 1997.

[9] G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *FPGAs for Custom Computing Machines*. IEEE, 2002.

[10] W. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[11] J. Liu, M. Chang, and C.-K. Cheng. An iterative division algorithm for FPGAs. In *ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 83–89. ACM, 2006.

[12] Z. Luo and M. Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions on Computers*, 49(3):208–218, 2000.

[13] E. Roesler and B. Nelson. Novel optimizations for hardware floating-point units in a modern FPGA architecture. In *Field Programmable Logic and Applications*, volume 2438 of *LNCS*, pages 637–646. Springer, Sept. 2002.

[14] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machine. In *FPGAs for Custom Computing Machines*, pages 155–162. IEEE, 1995.

[15] L. Zhuo and V. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In *Reconfigurable Architecture Workshop, Intl. Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2004.