The Krawczyk Algorithm: Rigorous Bounds for Linear Equation Solution on an FPGA

Christophe Le Lann, David Boland, and George Constantinides

Electrical and Electronic Engineering, Imperial College London,
London, SW7 2AZ, UK

Abstract. In the majority of scientific computing applications, values are represented using a floating point number system. However, this number system only considers an approximate value without any indication of the approximation's accuracy. Interval arithmetic provides a means to ensure that the solution is bounded with absolute certainty.

However, whilst interval arithmetic can be applied to any algorithm to ensure bounds on a solution, the limitations of interval arithmetic can lead to bounds that are not always tight and hence not particularly useful. As a result, some algorithms are specifically designed with interval arithmetic in mind to find high quality bounds on a solution; the Krawczyk algorithm is one such algorithm. The Krawczyk algorithm is targeted towards solving systems of linear equations, which is a common problem in scientific computing and has drawn a wide interest in the FPGA community. We show that by accelerating this algorithm in hardware, developing specialised arithmetic units, it is possible to gain orders of magnitude improvement in execution time over a C implementation.

Keywords: FPGA, Interval Arithmetic, Krawczyk Algorithm.

1 Introduction

On any modern processor, real numbers are usually represented using a floating point number system. However, due to their finite precision, only a subset of real numbers can be accurately represented, meaning most values are approximated to the nearest floating point number. Over the course of an algorithm, errors arising from these approximations accumulate and can cause a significant deviation from the nominal result, and unfortunately it is impossible to directly gain any insight into the size of this deviation, or the accuracy of the output data.

Interval arithmetic is a simple method to obtain a bound on the accuracy, where every number is defined by an interval bounding all possible values it may have taken due to previous approximations. However, any software implementation of interval arithmetic for an algorithm will result in significantly lower performance than the equivalent algorithm in floating point arithmetic [1]. Instead, in this work, we maintain high performance by creating an algorithm-specific hardware implementation using interval arithmetic operators.

A. Koch et al. (Eds.): ARC 2011, LNCS 6578, pp. 287–295, 2011.
© Springer-Verlag Berlin Heidelberg 2011

In order to satisfy this aim, we first needed the relevant hardware blocks for basic operators ($+$, $-$, $\times$, $\div$, $\sqrt{}$) operating on intervals, themselves using operators and comparators ($=$, $>$, $<$) operating on floating point numbers. While hardware implementations for general purpose interval arithmetic processors have previously been created [2, 3], we have created a customisable library of operators, notably in terms of rounding mode, exponent and mantissa size, so as to exploit the full freedom offered by FPGAs in creating a fully-custom datapath.

We then create such a custom datapath to accelerate the Krawczyk algorithm, which is an algorithm to find the solution of a system of linear equations that is specifically tailored towards interval arithmetic. We show that through exploiting the parallelism inherent in these new operators, as well as parallelising the algorithm itself, we gain a significant speed up over software. Though in this work we have only reported results for single precision, we note our customisable library of operators will also be of use for the acceleration of alternative algorithms or to facilitate future exploration into the trade-offs between performance and error for the Krawczyk algorithm in variable precision.

2 Background

2.1 Interval Arithmetic

Interval arithmetic [4] is based upon defining a closed interval as the set of all real numbers lying between its bounds, then propagating these intervals through any operation to ensure that the resulting interval encloses all real numbers that are the result of the operator applied to any two real numbers taken from the respective input operand intervals. We note here that interval arithmetic is an important tool when searching for rigorous results because while it is not possible to represent all reals in floating point, it is possible to create an interval of two floating point values which bound the desired real number.

2.2 Ensuring Correct Rounding

The IEEE-754 standard defines four rounding modes: “round to nearest even”, “round to zero” (truncate) [5], “round towards $+\infty$” (ceil), and “round towards $-\infty$” (floor). Whilst most software is written using the first mode, interval arithmetic requires latter two in order to entirely enclose the solution interval.

In order to ensure correct rounding such that safe bounds are achieved at the output, the floating point unit requires a higher precision in its inner working than that seen at the input and output of a floating point unit. The need for this can easily be seen in the case of subtraction: with a three bit mantissa (plus the leading 1.), one might try to compute the result of $16-15 = 1.000\cdot 2^4 - 1.111\cdot 2^3$. After normalisation, this becomes $(1.000 - 0.111)\cdot 2^4 = 0.001\cdot 2^4 = 1.000\cdot 2^1$. Even though the correct result is representable with three bits $(1.000\cdot 2^0)$, the result returned is incorrect because a bit was lost during normalisation.

The IEEE-754 standard defines three additional bits to avoid such errors: the “guard”, “round” and “sticky” bits [6]. The guard and round bits are used

as classic bits to locally increase the mantissa precision, while the sticky bit represents whether the result is exact or not, to ensure correct rounding.

2.3 The Solution of a System of Linear Equations

The solution to a system of linear equations of the form $Ax = b$ (where A is an $N \times N$ matrix, while x and b are $N \times 1$ vectors) forms the basis of a large number of problems, most notably in the realm of scientific computing. As such, there is a large interest in accelerating algorithms to find this solution using hardware. Some examples include Cholesky [7]; Gauss-Jordan [8]; and Conjugate Gradients [9]. Whilst it is possible to replace every operator with its interval arithmetic equivalent in any of these algorithms to obtain a bounded solution, this may not be the tightest possible bound because interval arithmetic suffers from the so-called "Dependency Problem", where wide bounds arise due to the fact every interval for a given variable is treated independently [4]. A trivial example is the following: for a variable x which lies in the interval $[0, 1]$, perform the operation $x - x$. The interval should be $[0, 0]$, but the result using interval arithmetic would be $[-1, 1]$. Whilst these algorithms contain dependencies and will suffer from this effect, the Krawczyk method, described in Figure 1, attempts to mitigate this problem by iteratively refining the interval solution vector x over a finite number of iterations to get tight bounds for x which satisfy $A \cdot x \supseteq b$ [4].

We define additional interval operations, as in [4], for the absolute value of an interval (1), the mid-point of an interval (2) and the norm of an interval matrix (3).

$$|[\,x\,;\,y\,]| = \max\left(|x|\,,|y|\right) \tag{1}$$

$$\text{mid}\left([\,x\,;\,y\,]\right) = \frac{x+y}{2} \tag{2}$$

$$\|M\| = \max_i \sum_j |M_{ij}| \tag{3}$$

1. $Y = [\text{mid}(A)]^{-1}$ (in floating point arithmetic).
2. $E = I - YA$ (in floating point arithmetic). If $\|E\| \geq 1$ exit with fail
3. $X_0 = \left[-\frac{\|Yb\|}{1-\|E\|}\,;\,\frac{\|Yb\|}{1-\|E\|}\right]$ (in interval arithmetic).
4. Repeat Until Convergence
   $X_{n+1} = (Yb + EX_n) \cap X_n$ (in interval arithmetic).

Fig. 1. Pseudo code for the Krawczyk method to solve $Ax = b$

The Krawczyk method initially finds an approximate solution by a non interval method Y. Moore then showed that under specified conditions, this can be used to create an interval X_0 which both contains the answer and could be refined using Krawczyk's interval version of Newton's method [4].

3 Parameterisable Interval Arithmetic Units

To create interval arithmetic components, we need the relevant floating point operators with parameterisable rounding modes. While many IP cores which

implement floating point arithmetic, including Xilinx LogiCores [10], FloPoCo [11] and Northeastern University's cores [12, 13], none of these libraries directly meet our requirements. In this work, we modify the Northeastern University library, because it is open source and easy to use, with our add rounding mode selection.

3.1 Enabling Correct Rounding

In order to ensure correct rounding, we need to add and handle the guard, round and sticky bits. This is relatively straightforward for addition, subtraction and multiplication. For the first two, we retain the guard and round bits, calculate the sticky bits after denormalisation of the two input operands, perform the operation and re-normalise. As multiplication is simply an integer addition of the exponents and an integer multiplication of the mantissas, we just retain the most significant bits for the desired precision along with the extra guard and round bits, and calculate the sticky bit from the remaining least significant bits.

Division and square root operations, however, are more complex. This is because the Northeastern library, as described in [13], accelerates these operations using Taylor series approximations with table look-up. Unfortunately, this means the sticky bit cannot be determined. We decided to always set the sticky bit to one because while such an approximation may lead to an inaccurate result for a single floating point value, it guarantees safe rounding with respect to interval arithmetic such that the derived bounds will still contain the true solution - albeit bounds that may be slightly wider than the bounds if the correct value for the sticky bit were used. We adopt a similar approach for the square root.

3.2 Parameterizable Interval Unit Implementation

Addition and subtraction. Implementing an ALU on intervals according to interval arithmetic rules [4] is quite straightforward the basic floating point operators, in our work, to maximise the speed, we use separate adders for the lower and upper bound, resulting in the operators shown in Figures 2 and 3.

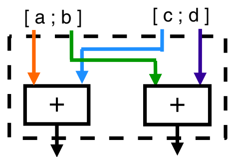

Fig. 2. Interval Adder

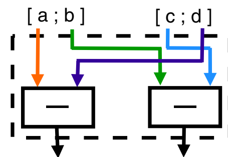

Fig. 3. Interval Subtractor

Multiplication and Division. Unlike addition and subtraction, a direct interval arithmetic implementation of multiplication, requires the calculation and comparison of several possible solutions to find the extremes. However, by examining the input operands' sign, it is possible eliminate several candidate bounds without even computing them. In the most complex case, we require two multipliers per bound, such that the hardware given in Figure 4 is sufficient. For division, we applied a similar process results in only two parallel dividers.

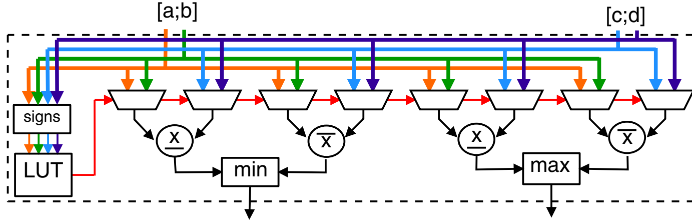

Fig. 4. Interval multiplier

Other operators. To simplify our implementation of the Krawczyk algorithm, we created three additional specialist interval operators: magnitude (1), mid-point (2) and intersection. Interval magnitude is calculated by forcing the sign bit to zero, so that we only need to compare the mantissas and exponents of the two inputs. A mid-point component was made, because this is just an addition and a divide by 2, with the latter operation being just a subtraction of one from the exponent which is cheaper than normal division. Intersecting two intervals is nothing more than selecting the highest lower bound and the lowest upper bound, though we added a check for the case of the null interval for completeness.

4 Krawczyk Algorithm Implementation

Memory Architecture. We decided to initially load and store the entire matrix on-chip and to make use of the large memory bandwidth available on an FPGA. This is a reasonable choice for applications consisting of small to medium size matrices, such as MIMO computation [14], or control system design [15]. Furthermore, as this algorithm consists of many stages, including an iterative refinement stage, we believe it is easily possible to interleave the I/O communication with computation, as discussed in Section 5.1. Furthermore, we also store intermediate matrices in on chip RAM to maintain high performance. Though this comes at a cost of high RAM use, we wish to take advantage of the growing size of the FPGAs, and note that if necessary we could examine RAM saving techniques from existing literature to improve scalability or trade it with performance [16].

Our RAMs are organised such that matrix rows are stored in individual RAMs, allowing us to perform operations for matrix cloumns in parallel.

Computation of $Y = [\text{mid}(A)]^{-1}$. In this block, the mid-point can be computed for each column in parallel by using n instances of our mid-point operator. This operator can be fully pipelined for maximum speed. We then invert this matrix, using the Cholesky decomposition algorithm followed by a lower triangular inversion. We note that this stage does not require interval operators, meaning alternative existing optimised hardware implementations, such as [7], could easily replace our implementation. However, we adopted a simple approach where any operations that are common to any column, such as the division or the subtraction are parallelised.

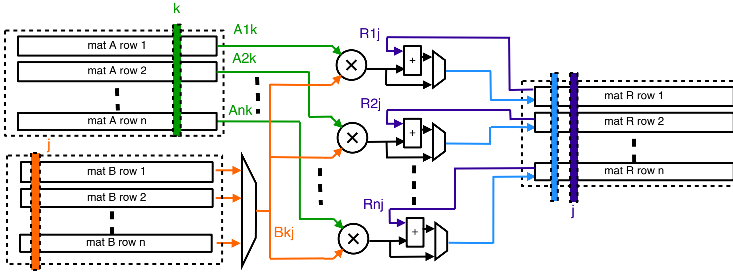

Fig. 5. Matrix multiplication – simplified view

Computation of $\boldsymbol{E = I - YA}$. For the matrix multiplication, we perform parallel multiplication of a column of the Y matrix with the first associated element of the A matrix and store the n temporary results in n RAMs. We then repeat this for all the columns of Y. After this step, the initial values for every element of the result matrix are created, meaning we can the repeat the column multiplication of Y with the next associated element of A and sum this with the temporary results from the RAMs. Provided the system order is greater than the ALU pipeline latency, there will be no data hazards and the system will be fully pipelined; otherwise, we simply stall the pipeline to avoid any problems. This process is summarised in Figure 5. Finally to compute E we then compute the subtraction for each column in parallel, using interval components.

Computation of $\boldsymbol{X_0 = \left[-\frac{||Yb||}{1-||E||};\frac{||Yb||}{1-||E||}\right]}$. The matrix-vector multiplication of Yb, is achieved with an architecture similar to Figure 5, but using only one vector column in the second operand. Pipeline stalls are added between column multiplications if necessary for the accumulator. The matrix norm $||E||$ is calculated with a similar component that replaces the multiplier with an absolute operation. The maximum of a vector $||Yb||$ is an absolute operation and serial comparison of elements over a vector. Finally, as the bounds of X_0 are created by simply changing the sign bit of the result.

Computation of $\boldsymbol{X_{n+1} = (Yb + EX_n) \cap X_n}$. The value EX_n is computed by re-using the matrix-vector multiplier in the previous step. As the matrix-vector multiplier outputs each vector element in series, the summations and intersections are performed serially as results are produced.

5 Results

All these components were written in VHDL and synthesized using Xilinx XST M.63c (Xilinx ISE 12.2) targetted towards a Xilinx Virtex6 XC6VLX760-1. Results for the components of the Krawczyk algorithm in Section 4 for a matrix order of 16 in IEEE single precision floating point are shown in Table 1.

Table 1. Hardware Implementation Results

Description	Regs	LUTs	DSPs	BRAMs	$f_{max}(MHz)$
Middle matrix	5233	0	0	0	301.42
Cholesky decomposition	12798	17990	73	17	154.76
Triangular inversion	8582	16364	40	4	111.14
Matrix by matrix FP multiplier	27105	44284	128	0	154.76
Matrix by vector interval multiplier	28058	45166	128	0	154.76
Matrix sum of magnitudes	6355	10632	0	0	231.75
Vector norm	141	332	0	0	239.26
Krawczyk initialization	1250	1090	8	4	266.81
Krawczyk iteration	1742	2765	0	0	285.39

5.1 Comparison against an MPFR C Simulator

Validation. We first validated both the individual components, and the complete design against the result given by the MPFR C simulator [17] for several different linear systems. We chose the MPFR library as it is directly comparable to our hardware components as it contains parameterisable precision and rounding modes. The design met the expectations, albeit with the small caveat that the Y matrix slightly differs from the one given by the C simulator because of the non IEEE-754 compliance of the floating point dividers and square root operators, as stated in Section 3.2, which in some cases lead to bounds that where slightly wider, although still valid, than the software counterpart.

Performance Comparison. The different latencies corresponding to the different sub-components composing the whole Krawczyk algorithm implementation were used to calculate the total latency of $10.5n^2 + 28.5n + 106$ clock cycles for the data preparation (until completion of the Krawczyk initialization, computing x_0), plus $7n + 15$ clock cycles per iteration of the Krawczyk loop.

For a main clock operating at 100 MHz, a 16th order linear solver completes within less than 4000 clock cycles, or less than 40 μs (assuming the convergence is reached within five iterations). A 100th order linear solver would complete within approximately 1.1 ms. In comparison, the C simulator on a AMD Turion 64 X2 requires around 20 ms for a 16th order system, and 2.1 seconds for a 100th order system: that is 500 to 2000 times slower than the FPGA-based solution. This large speed up is a result of parallelism of the algorithm – we have reduced the computational complexity from $\Theta(n^3)$ to $\Theta(n^2)$, and the efficiency of our specialist parallel interval arithmetic operators in contrast to the software approach of using MPFR. Computing bounds in variable precision with MPFR – as well as with any software capable of finding robust solutions in interval arithmetic with variable precision - is significantly slower than computing in standard single or double precision floating point because all the operations to compute floating point values (ordering the operands, computing the exponent difference, shifting the mantissa if necessary, performing the operation and renormalising) must be simulated in software instead of making use of the floating point hardware components on a computer [1, 17].

Our result estimates are based on a sustained performance, assuming that I/O is interleaved with computation, which we believe is a realistic assumption, given that the amount of data that must be transferred is $\Theta(n^2)$, and the computation time is also $\Theta(n^2)$. However, we do acknowledge that interleaving I/O with computation does increase our RAM requirements to store the A matrix for a subsequent problem whilst computing the current problem.

6 Conclusions

Interval arithmetic constitutes an original method for representing and computing numbers on digital systems that circumvents the inability of floating point to track data inaccuracy by considering an interval bounding a real value with an absolute certainty instead of an approximate value without any indication of the approximation accuracy. In this work, we have modified a pre-existing library of floating point operators to develop a library of interval operators.

We have then applied these to the Krawczyk algorithm, which is specifically designed to solve linear systems of equations based on interval arithmetic, and shown that a dedicated hardware operator which conforms to the computer C simulations to be very efficient: the algorithm optimizations and parallelizations led to a speed-up in orders of magnitude in comparison with a C implementation on modern computers for system orders up to a few dozen. Though we are limited to small orders, we believe by integrating existing techniques which trade performance with scalability from relevant literature for the various operations in the Krawczyk algorithm with our interval components and still gain significant improvements over software. Altogether, we believe that we have both demonstrated the potential for FPGA acceleration of the Krawczyk algorithm and created a library of parameterisable floating point and interval operators that could result in significantly superior performance in many further applications.

References

1. Schulte, M.J., Swartzlander Jr., E.E.: Software and hardware techniques for accurate, self-validating arithmetic. Applications of Interval Computations, 381–404 (1996)
2. Schulte, M.J., Swartzlander Jr., E.E.: A family of variable-precision interval arithmetic processors. IEEE Trans. Comput. 49(5), 387–397 (2000)
3. Kirchner, R., Kulisch, U.: Hardware support for interval arithmetic. Reliable Computing 12, 225–237 (2006), 10.1007/s11155-006-7220-9
4. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
5. IEEE Computer society, IEEE standard for floating-point arithmetic, IEEE Std 754-2008, pp. 1 –58 (August 2008)
6. Koren, I.: Computer arithmetic algorithms. Prentice-Hall, Inc., Upper Saddle River (1993)
7. Salmela, P., Happonen, A., Burian, A., Takala, J.: Several approaches to fixed-point implementation of matrix inversion. In: Proc. Int. Symp. Signals, Circuits and Systems, vol. 2, pp. 497–500 (July 2005)

8. de Matos, G., Neto, H.: On reconfigurable architectures for efficient matrix inversion. In: Proc. Int. Conf. Field Programmable Logic and Applications, pp. 369–374 (August 2006)
9. Roldao, A., Constantinides, G.A.: A high throughput fpga-based floating point conjugate gradient implementation for dense matrices. ACM Trans. Reconfigurable Technol. Syst. 3, 1:1–1:19 (2010)
10. Xilinx, Xilinx logicore, http://www.xilinx.com/ipcenter/
11. de Dinechin, F., Detrey, J., Cret, O., Tudoran, R.: When FPGAs are better at floating-point than microprocessors. In: Proc. Int. Symp. Field Programmable Gate Arrays, p. 260 (2008)
12. Belanovic, P., Leeser, M.: A library of parameterized floating point modules and their use. In: Proc. Int Conf. Field Programmable Logic and Applications, pp. 657–666 (2002)
13. Wang, X., Leeser, M.: Variable precision floating point division and square root. In: Workshop on High Performance Embedded Computing, pp. 47–48 (2004)
14. Biglieri, E., Calderbank, R., Constantinides, A., Goldsmith, A., Paulraj, A., Poor, H.V.: MIMO Wireless Communications. Cambridge University Press, Cambridge (2007)
15. Maciejowski, J.M.: Predictive control with constraints. Prentice Hall, Essex (2002)
16. Boland, D., Constantinides, G.: Optimising memory bandwidth use for matrix-vector multiplication in iterative methods. In: Sirisuk, P., Morgan, F., El-Ghazawi, T., Amano, H. (eds.) ARC 2010. LNCS, vol. 5992, pp. 169–181. Springer, Heidelberg (2010)
17. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. 33(2), 13 (2007)