

A Tightly Coupled Accelerator Infrastructure for Exact Arithmetics

Fabian Nowak and Rainer Buchty

Chair for Computer Architecture
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
{nowak|buchty}@kit.edu

Abstract. Processor speed and available computing power constantly increases, enabling computation of more and more complex problems such as numerical simulations of physical processes. In this domain, however, the problem of accuracy arises due to rounding of intermediate results. One solution is to avoid intermediate rounding by using exact arithmetic. The use of FPGAs as application-specific accelerators can speed up such operations compared to their software implementation. In this paper, we present a system approach employing state-of-the art FPGA and interconnection technology for exact arithmetic with double-precision operands, delivering up to 400M exact MACs/s in total and providing a speedup of up to 88 times over competing software implementations in the case of matrix multiplication.

1 Introduction

With the computation of increasingly complex problems, accuracy issues arose related to rounding effects taking place in current FPU implementations. These may lead to unsatisfying results and even physical damage, if upfront simulations do not indicate certain problems. This problem is typically addressed by certain exact arithmetics built into mathematics and simulation packages. Such arithmetics increase the width of the internal data representation or concatenate several floating-point numbers in order to enlarge the “accuracy window” in which no rounding is necessary. While these software implementations provide a viable workaround, they are magnitudes slower than native hardware support. Custom accelerator hardware can speed up such operations using several design techniques, e.g. pipelined execution and parallelization.

We therefore present an accelerator system for exact arithmetics. Based on a careful examination of the underlying algorithm for implementing exact arithmetics, a hardware solution employing pipelining techniques and exploiting parallelism is proposed delivering up to 400M exact MAC operations per second.

In the remainder, we first outline related work in Section 2. We then present our general architecture design in Section 3 before discussing implementation issues and viable solutions in Section 4. The elaborated design is thoroughly evaluated in Section 5. We conclude this paper by summing up the results and presenting our plans for future work in Section 6.

2 Related Work

Performing floating-point operations in software is quite costly, therefore many FP accelerators exist. Such accelerators were originally the domain of dedicated silicon, but with increasing FPGA capabilities, these gained attention as computation accelerators. In [1], for example, double-precision operations are employed for accelerating physics simulations. Other work focuses on exploiting FPGAs for matrix multiplications with double precision [2], or on the conjugate gradient method [3–5].

The problem of exact computation is well-understood, and hence multiple solutions exist. For certain computations, the effect of errors can be evaluated upfront, allowing the use of lower-precision computation [6]. For computations where knowledge of the included error is required, using so-called Staggered Interval Arithmetics [7] (SIA) is an option. The precision of floating-point operations can also be enhanced by increasing mantissa and exponents as e.g. provided by the GNU MP library.

Our work is motivated by the work of Kulisch et al. [8],[9], who developed a concept for exact arithmetics support in regular floating-point units and also implemented a PCI-attached coprocessor for exact arithmetics. Following Kulisch’s work, a straight-forward implementation supporting single-precision operands demonstrated the general applicability to FPGA technology, but did not yet offer speedup in comparison to software implementations [10].

In order to further support the use of such accelerators and to ease benchmarking, we developed a runtime system enabling dynamic switching from conventional to exact arithmetics implemented in software or hardware [11]. Dynamically applying exact arithmetics only where required, results in the fastest total runtime while still guaranteeing numerical robustness and therefore delivering precise results.

3 Architecture and Design

The need for exact arithmetics in numerical programs arises in most cases for matrix multiplications. Thus, we give a detailed description of how matrix multiplication can be split and organized in the optimal way for data transfers and parallel pipelined processing in this section.

3.1 Exploiting Matrix Multiplication Properties

Matrix multiplications can be regarded as consisting of matrix-vector multiplications that are simply a series of inner vector products:

$$C = A \cdot B = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} \cdot (b_1 \cdots b_o) = (A \cdot b_1 \cdots A \cdot b_o) = \begin{pmatrix} a_1 \bullet b_1 & \cdots & a_1 \bullet b_o \\ \vdots & \ddots & \vdots \\ a_m \bullet b_1 & \cdots & a_m \bullet b_o \end{pmatrix} \quad (1)$$

where A is an $m \times n$ matrix with rows a_1 to a_m , B is an $n \times o$ matrix with columns b_1 to b_o , C is an $m \times o$ matrix, and \bullet denotes the inner product.

When looking at Equation 1, we easily can see that for data reuse we need to stream in one of the two matrices, while reading the other from local memory, because row a_i is used for computing the complete row i of C . This allows to stream the rows of A and read R vectors of the local matrix B concurrently element by element according to the column of A . Hence, we obtain R elements of C at once after all the n elements have been accumulated. Note that reading more than one element of a vector b_j is not useful as only one product can be computed and accumulated per cycle. Thus, individual EAUs have to be used in parallel in order to gain speedup.

However, not only row a_i can be reused, but the same works for the vectors b_j of B as well: having enough bandwidth available or running the different components at different clock speeds allows to receive more than one value of a_i . This can be exploited to stream in $S > 1$ rows of A concurrently in favor to delivering several elements of a_i because of the same reason as mentioned above. As a result, the vectors b_j are the same for all multipliers, while each “plane” is given different rows a_i of matrix A .

In the obvious approach, the elements of A and B are being sent one by one or packed into tuples $(a_{i,k}, b_{j,k})$. In that case, no additional overhead due to data organization occurs. When multiplying several vectors at once in the parallelized approach, we read the k^{th} element of R vectors, i.e. of elements $b_{1,1}, \dots, b_{R,1}, b_{1,2}, \dots, b_{R,2}, \dots, b_{1,k}, \dots, b_{j,k}, \dots, b_{R,k}, \dots, b_{R,n}, \dots, b_{R+1,1}, \dots, b_{o,n}$, where the second index denotes the row of column vector b_j . Thus, B should be transferred into DDR memory in such a way that subsequent memory accesses happen in exactly this order. In case $R = o$, nothing special has to be done, otherwise the matrix has to be reordered. Finally, for the 2-dimensional approach, the elements of matrix A are accessed rather row by row than column by column. Hence, transposing A is already helpful, but as in the second case, data has additionally to be reordered with respect to block size S that denotes the number of “planes”, i.e. the number of elements that are transferred per cycle and number of rows a that are processed in parallel.

3.2 FPGA Integration

Following Sect. 3.1, it is advisable to store one of the two matrices to be multiplied, B , in on-board memory, enabling concurrent computation of different columns of the result matrix C . This requires transferring matrix B from main memory to FPGA memory. Additional control logic applies for controlling initialization and multiplication, resulting in the overall architecture depicted in Fig. 1 where the user logic is clocked with 130 MHz and DDR memory with double the frequency, i.e 260 MHz, as required by the DDR controller.

Our hardware environment consists of the UoH HTX Board [12] containing a Xilinx Virtex-4 FPGA. The board is located in a host PC using the HyperTransport (HT) bus. Resulting from this hardware, DDR memory is clockable at a maximum of 266 MHz and HT either at 200 MHz or at 400 MHz.

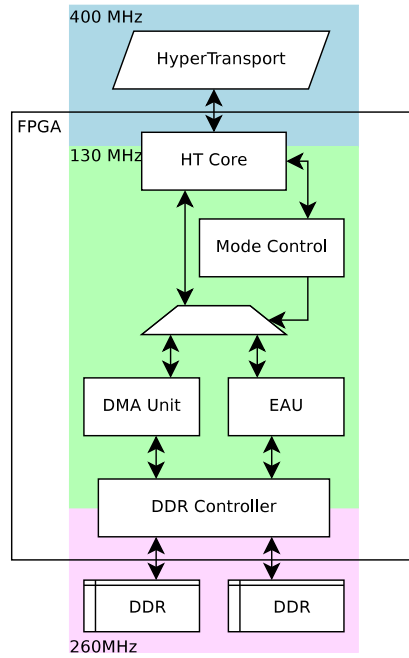


Fig. 1. Integration of the EAU in the UoH HTX Board

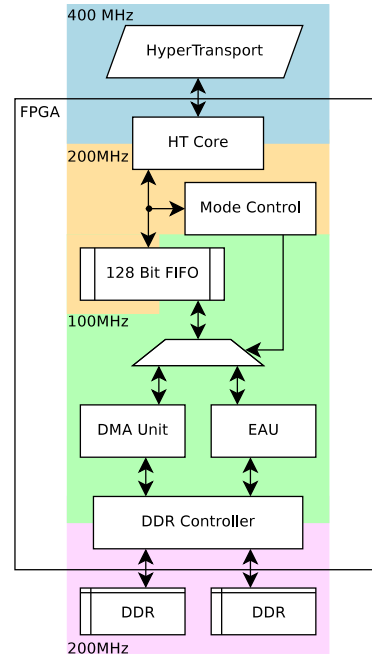


Fig. 2. Enhanced integration of the EAU with intermediate buffering

In order to better exploit the HT-provided bus bandwidth, buffer circuitry can be applied leading to the architecture sketched in Fig. 2, where the buffers decouple the 200 MHz HT backend from the 100 MHz user logic, which in return decreases the DDR memory’s frequency to 200 MHz. Such an architecture is expected to perform better due to operating near full bandwidth usage in contrast to the former.

4 Implementation

4.1 Multiplier

The architecture is designed in conformance with IEEE-754, which is addressed as follows: by storing the results of double-precision multiplications in 106-bit registers, overflows and underflows do not occur. As the input data is sent from software applications with their own safety checks included, we assume that no NaN, explicit underflow or infinity is being sent as input data.

When designing and implementing the multiplier, special focus has been put onto high synthesizable frequency. By splitting the multiplication of the 53 mantissa bits into 14 different fully pipelined stages, we achieved a frequency

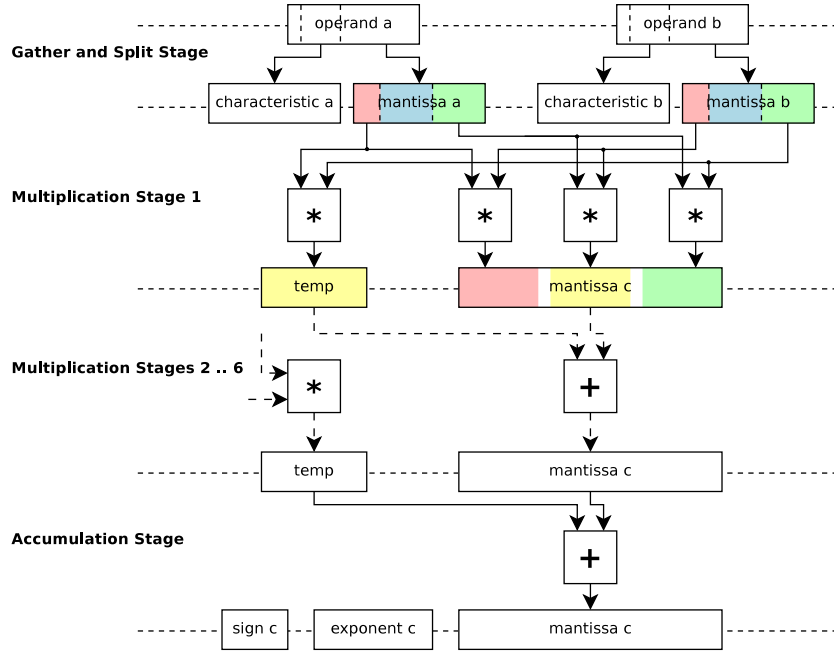


Fig. 3. Simplified sketch of a double-precision multiplier implementation

rate of 176 MHz and a low resource foot print when synthesizing. The results for the employed Xilinx Virtex-4 FX100-11 FF1152 are listed in Table 1.

The implementation follows the scheme depicted in Fig. 3: first, the operands are split into sign, characteristic, and extended 53-bit mantissa in the first gather-and-split stage where the mantissa is split into k blocks requiring k^2 multiplications. We then execute non-overlapping multiplications in parallel, thus initially populating the first pipeline register carrying the result. An additional multiplication can be executed in parallel, but as it overlaps with the other partial results, it must be accumulated onto the mantissa pipeline register in the next stage where the partial product is calculated in parallel. This step repeats for the remaining partial products, resulting in a total of 6 multiplication stages when employing hypothetical 19-bit multipliers. In the final step, the last product has to be accumulated, the sign and new exponent have to be passed.

Resource	Usage	Percentage
Slices	1,946	4%
Flip Flops	2,669	3%
4-Input LUTs	2,874	3%
DSP48s	16	10%

Table 1. Resource usage of the multiplier after synthesis

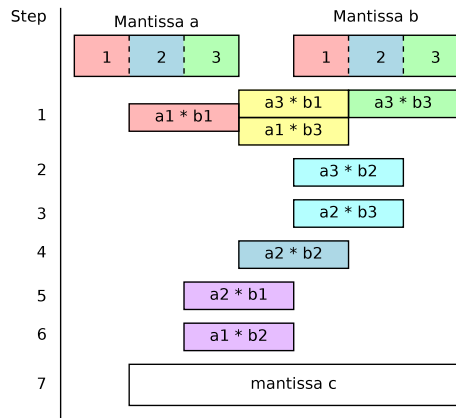


Fig. 4. Simplified sketch of the multiplication pipeline assuming a DSP width of 19

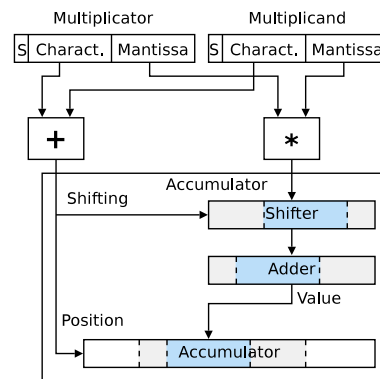


Fig. 5. Product of two operands shifted and added to the correct position of the long accumulator

As the FPGA hardware provides a DSP width of 18 bits with 1 bit used for the sign, we have to divide the mantissa into four blocks of length 13 and 14 bits, respectively, resulting in 16 multiplications totally, with the inner steps being 12 multiplication stages, instead of 6 as in the implementation scheme. Note that the exponent requires one more bit, i.e. 12 bits, now, and the mantissa requires twice the amount of bits in contrast to the original operand mantissa, i.e. $2 * 53 \text{ bits} = 106 \text{ bits}$. In order to report infinity, a minor amount of logic has been added, which sets all exponent bits to '1' to signal infinity. Figure 4 visualizes the pipelining of the multiplication according to Fig. 3 and the former description.

4.2 Accumulation Unit

The unmodified result of the multiplication, i.e. unshorted and with extended exponent, is handed to the accumulation unit, which shifts the product, determines the position where the product is to be added onto the large accumulation register, and finally adds the shifted product to these positions, as is shown in Figure 5. This way, no loss of precisions with regard to the input data happens. The accumulation register is capable of accurately storing the products of both largest and smallest numbers by value with a total width of $64 * 67 = 4288 \text{ bits}$; additional space is used to avoid overflows (c.f. [13]).

The accumulation unit was designed as a pipelined unit. Data is accumulated in several steps and track of the blocks' values is kept in order to resolve any occurring carries [13]. The flow of operation within this pipeline involves an atomic read-modify-write operation. In hardware, this can either be addressed as one massive pipeline stage limiting the overall pipeline frequency, or a more fine-grained pipeline where after each add/subtract command an atomic read-modify-write scheme is executed, stalling the pipeline for the time of execution.

Resource	Usage	Percentage
Slices	17,094	40%
Flip Flops	5,808	6%
4-Input LUTs	32,481	38%
DSP48s	1	1%

Table 2. Resource usage of the accumulation unit after synthesis

Resource	Usage	Percentage
Slices	17,838	42%
Flip Flops	8,449	10%
4-Input LUTs	34,207	40%
DSP48s	16	10%

Table 3. Resource usage of multiplication and accumulation units after synthesis

For the sake of reaching a maximum clock rate, we chose the latter approach leading to a latency of 3 cycles for add/subtract operations.

After accumulation, the pipeline determines the range where valid data is stored in the accumulator, reads its value and converts it to a double-precision floating-point value. Due to the two stall cycles, we can read the accumulator value two cycles after it has been written. The additional approach of assigning each operation an individual ID allows tracking intermediate results, as will be explained in more detail in Sect. 4.3. Table 2 shows the resource usage of the accumulation unit, Table 3 of the combination with a multiplier to one design without additional control or interface instances.

4.3 Communication Principles and Controller

As already indicated above, the accumulator is not only given mantissa and exponent, but also an identification number (ID) and the operation itself. The CLEAR operation resets the internal state and accumulator; the accumulator can then be used for the next series of exact MAC operations. For each operation, it is also necessary to indicate whether the command is valid in order to separately control the multipliers and accumulator units.

The second part of the communication occurs from the accumulator to its data input site, indicating that it is ready for data (`rfd`). There is a delay of one cycle between the input of an operation and the `rfd=0`, which requires to buffer the subsequent command if indicated as valid operation for automatic processing after the current operation.

4.4 Data Transfers

To begin with, data of one of the matrices has to be brought onto the HTX board's DDR RAM. After successful completion, operation can commence: data needs to be streamed via DMA from host system memory to the HTX board, where it is multiplied and accumulated with the data residing in the board-local memory. Finally, the accumulated result has to be converted to double-precision floating-point format and brought to the host-system, where it can be processed further (Figs. 1, 2).

With the proposed design in mind, the central control instance becomes responsible for controlling DDR RAM operations, especially with regard to initialization and burst transfers, for mode control where mode is one of initialization,

in/out transfer phase, regular operation with read accesses for the operands and possibly write accesses for the results, and for future extensions that modify individual blocks of the accumulation units.

From the client side, it is important that after requesting a medium-grained operation (inner product, matrix multiplication) the relevant data is sent. This ideally happens by writing operand data to distinct memory-mapped addresses of the HTX board while respecting the data order according to Sect. 3.

Recalling the capabilities of the hardware platform and the resulting mode of operation, reading the accumulated result can be accomplished in one of two basic ways: explicitly offered `read` operations in the accumulator that are started upon read requests on the HyperTransport posted queue, and streamed, uninterrupted write-out of the accumulator to main memory. Explicit read operations in the accumulator pipeline violate the concept of pipelining, making it even more complicated and potentially decreasing maximum execution speed. The second approach, i.e. to always compute the accumulator's value in double-precision and to uninterruptedly send these values to the system's main memory, using the IDs as least-significant parts of the address, looks fine at first sight. While it does not interfere with the read bandwidth of the EAU on the HTX board, it however does not allow to send the results of more than 3 EAUs, as each EAU produces a new result each 3 cycles.

To circumvent the afore-mentioned limitations, we suggest to combine the two approaches: each EAU converts its accumulation values and returns them together with the IDs that led to these results. A separate controller cares for incoming read requests, buffers the read address that contains the accumulation unit and the ID, and returns the next value of the specified EAU that has the same ID. This task perfectly fits the controller required for initializing the local memory. Still, there is the choice between continuing execution while waiting for the result, or rather stalling the accumulation unit. When choosing the latter, no problems due to subsequent modifications can occur. While the number of bits for the ID has already been set to 4 for a sufficient amount of flexibility, the number of bits for different EAUs could be chosen arbitrarily. According to our experience with a single-precision implementation [10], we can say that a total of 16 EAUs will be enough, so that 4 bits of the address should suffice for choosing a specific EAU. This scheme can be enhanced further by having the controller automatically write back the final results of each completed accumulation.

4.5 Putting it all together

In Sect. 4.2, we saw that the accumulation unit accepts operands each 3 cycles due to otherwise conflicting accesses. By demultiplexing the products of the multipliers onto different accumulation units as in Figure 6, we can circumvent these disadvantages, allowing to compute uninterruptedly. When we even consider that upon each cycle we can obtain a value of the left input matrix A , we come up with a throughput of 3 exact MAC operations per cycle, which is 300M exact MAC operations per second (eMACs/sec) when running at a frequency of 100 MHz.

Resource	1 Mult, 1 Accu		2 Mults, 2 Accus	
	Usage	Percentage	Usage	Percentage
Occupied Slices:	19,973	47%	36,315	86%
Slices containing only related logic:	19,973	100%	36,315	10%
Slices containing unrelated logic:	0	0%	0	0%
Total Number of 4 input LUTs:	34,336	40%	68,892	81%
Number used as logic:	33,701		61,480	
Number used as a route-thru:	551		7,248	
Number used as Shift registers:	84		184	
Number of BUFG/BUFGCTRLs:	1	3%	2	6%
Number used as BUFGs:	1		2	
DSP48s:	16	10%	32	20%

Table 4. Resource usage of multiplication and accumulation units after mapping process. Percentages are given for the Xilinx Virtex-4 FX100.

5 Evaluation

5.1 Performance Estimations

We present the final formula for the overall execution time in Table 5. Here, bw denotes the bandwidth to board memory, f the user frequency. The execution time is composed of transfer to memory, streamed computation, and transfer from memory back to the host system. Note that the transfer to memory is limited by the HTX bandwidth bwh rather than by the memory bandwidth bwr .

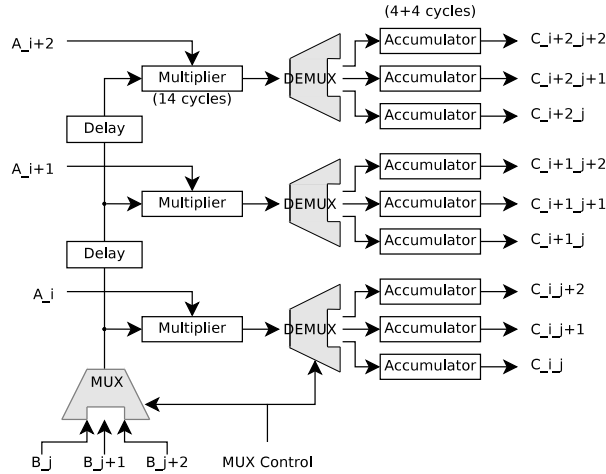


Fig. 6. Demultiplexing multiplier output onto accumulation units when multiplying rows i to $i + 2$ of matrix A with columns j to $j + 2$ of Matrix B . The iterator index k is not shown.

Transfer to RAM	$n * o * 8B/bw$
Transfer from RAM	$m * o * 8B/bw$
Computation	$m * n * o/tp$
Overall time	$(m + n) * o * 8B/bw + m * n * o/tp = (2m + m * n + 2n) * o/tp$
130 MHz Design	$(2m + m * n + 2n) * o/260M/s$
100 MHz Design	$(2m + m * n + 2n) * o/400M/s$

Table 5. Overview of the different components involved in maximum bandwidth usage depending on matrix dimensions n , m , o , throughput tp , and transport bandwidth bw

Dimensions	130 MHz	100 MHz
1000	3.86 sec	2.51 sec
4000	246 sec	160 sec
8000	3847 sec	2501 sec

Table 6. Best-case execution times of two different architectures for exact arithmetics on UoH HTX board

	GMP	C-XSC	DP
Runtime (ticks)	566,480,613	640,445,526	6,403,510
Runtime (μs)	177,089	200,211	2,001
M eMACs/sec	5.921	5.237	523.810 ¹

Table 7. Comparison between 256-Bits GMP, C-XSC, and double precision (DP), runtime in clock ticks. ¹ see text

Table 6 provides numbers for different matrix dimensions $m = n = o$ of 1000, 4000, and 8000, which would result in matrix sizes of 8 MB, 128 MB, and 512 MB, respectively.

Following Tables 5 and 6, we see a preference for the 100 MHz design; in contrast to the 130 MHz design, however, performance tightly depends on the streamed data and puts more pressure on the data transport layer of the hardware design.

Due to resource constraints on the chosen FPGA and due to the chose implementation of the exact accumulation unit, 2 accumulation units and one multiplier are the maximum number of resources to deliver maximum throughput of 2 eMACs each 3 cycles, i.e. 88M eMACs/sec for the 130 MHz design (Fig. 1) and 66M eMACs/sec for the 100 MHz design (Fig. 2), respectively.

5.2 Technology Scaling

As the architecture and current implementation do not look very promising, we conduct a technology scaling onto the Xilinx Virtex-6 LX240T, which is available for example on the Xilinx ML605 evaluation board. This FPGA provides enough space for 6 exact MAC units, so that at least 2 exact MAC operations can be started per cycle. Furthermore, due to the available increased logic speed, some pipeline stages of the time-critical read-modify-write scheme may be mergable, resulting in even 3 exact MAC operations, and hence in a maximum throughput rate of 300M eMACs/sec. Besides, the wider DSP units allow to reduce the multiplier pipeline length to less than 12 multiplication stages; the look-up tables with 6 inputs in contrast to 4 inputs might further reduce area requirements.

5.3 Comparison

In order to evaluate our architecture, we performed 2^{20} multiply-accumulates of the value $v = 2^{-20}$ onto an init value of 2^{20} using our hardware and two software solutions. For the latter, we employed an implementation based on the GNU Multiprecision Library (GMP) using a 256 bits wide internal floating-point representation and one based on C-XSC where a 4288 bits wide fixed-point accumulator is employed. Individual runtimes were measured on an Intel Pentium 4 3,2 GHz over 128 runs. All programs were compiled with `-O2`. Subtracting the initialization value afterwards we obtained the correct result 2^{-20} , which fails when using regular double-precision floating-point operations.

The measurements can be found in Table 7. We additionally provide the execution time with regular double-precision, which is about 100 times less compared to the highly accurate libraries. Our architecture concept is capable of executing exact multiply-accumulate operations in about the same order of time.

Based on the maximum achievable throughput for HT400, we can derive a theoretical maximum speedup of $\frac{400}{5.237} = 76.38$ over state-of-the-art software implementations of exact arithmetics. Following Sect. 5.1, limitations of the target platform still allow a maximum speedup of $\frac{88}{5.237} = 16.80$.

6 Results and Future Work

As of now, our system allows processing two MAC operation each 3 cycles while running at a frequency of 100 MHz, which leads to a theoretical processing speed of 66M MAC operations per second. Note that these operations are carried out in exact arithmetics, avoiding rounding and windowing errors. We showed in Section 3 how a multitude of accumulation units can help to constantly process streamed data and how using the DDR memory can further speed up the overall performance up to 400 MAC operations per second. A technology scaling showed that this architecture is a viable approach to exploit the available bandwidth of the HyperTransport bus. For better results, the implementation of the accumulator unit will be revised, promising a latency of 1 and even less resource usage.

Our future work consists of integrating the DDR memory and DMA controllers in order to carry out detailed measurements with numerical software where computation with exact arithmetics enables solving more complex problems. We also plan to extend our system infrastructure by numerical error measurements so that functions be directed to their exact arithmetics implementation automatically, sacrificing speed over robustness and numerical stability.

Similar to [1], the approach will be extended towards microprogramming capabilities so that (parts of) algorithms can be completely loaded onto the accelerator, thus supporting sparse matrix multiplications. Finally, the architecture is designed to deliver coarse-grained function acceleration of numerical algorithms such as the Lanczos algorithm.

References

1. Danese, G., Loporati, F., Bera, M., Giachero, M., Nazzicari, N., Spelgatti, A.: An Accelerator for Physics Simulations. *Computing in Science and Engineering* **9**(5) (2007) 16–25
2. Kumar, V.B.Y., Joshi, S., Patkar, S.B., Narayanan, H.: FPGA Based High Performance Double-Precision Matrix Multiplication. In: *VLSID '09: Proceedings of the 2009 22nd International Conference on VLSI Design*, Washington, DC, USA, IEEE Computer Society (2009) 341–346
3. DuBois, D., DuBois, A., Boorman, T., Connor, C., Poole, S.: An Implementation of the Conjugate Gradient Algorithm on FPGAs. In *Pocek, K.L., Buell, D.A., eds.: FCCM*, IEEE Computer Society (2008) 296–297
4. Morris, G.: Floating-Point Computations on Reconfigurable Computers. In: *HPCMP-UGC '07: Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference*, Washington, DC, USA, IEEE Computer Society (2007) 339–344
5. Strzodka, R., Göddeke, D.: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006)*. (April 2006) 259–268
6. Herbordt, M., Sukhwani, B., Chiu, M., Khan, M.A.: Production Floating Point Applications on FPGAs. online (July 2009) *Symposium on Application Accelerators in High Performance Computing (SAAHPC'09)*.
7. Kulisch, U.W.: Complete Interval Arithmetic and Its Implementation on the Computer. In: *Numerical Validation in Current Hardware Architectures*. Volume 5492/2009 of *Lecture Notes in Computer Science.*, Springer Berlin / Heidelberg (April 2009) 7–26
8. Bierlox, N.: Ein VHDL Koprozessor-kern für das exakte Skalarprodukt. PhD thesis, Universität Karlsruhe (November 2002) <http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1053>.
9. Kirchner, R., Kulisch, U.: Accurate arithmetic for vector processors. *Journal of Parallel and Distributed Computing* **5**(3) (1988) 250–270
10. Nowak, F., Buchty, R., Kramer, D., Karl, W.: Exploiting the HTX-Board as a Coprocessor for Exact Arithmetics. In: *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA2009)*, Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg (February 2009) 20–29
11. Buchty, R., Kramer, D., Kicherer, M., Karl, W.: A Light-Weight Approach to Dynamical Runtime Linking Supporting Heterogenous, Parallel, and Reconfigurable Architectures. In: *Architecture of Computing Systems – ARCS 2009*. Volume 5455 of *Lecture Notes in Computer Science.*, Springer Berlin / Heidelberg (February 2009) 60–71
12. Fröning, H., Nüssle, M., Slognat, D., Litz, H., Brüning, U.: The HTX-Board: A Rapid Prototyping Station. *Proceedings of the 3rd Annual FPGA World Conference* (November 2006)
13. Kulisch, U.W.: *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2002)