FPGA Based Acceleration of Decimal Operations

Alberto Nannarelli
Dept. Informatics and Mathematical Modelling
Technical University of Denmark
Kongens Lyngby, Denmark
Email: an@imm.dtu.dk

***Abstract*—Field Programmable Gate-Arrays (FPGAs) can efficiently implement application specific processors in non-conventional number systems, such as the decimal (Binary-Coded Decimal, or BCD) number system required for accounting accuracy in financial applications.**

The main purpose of this work is to show that applications requiring several decimal (BCD) operations can be accelerated by a processor implemented on a FPGA board connected to the computer by a standard bus.

For the case of a telephone billing application, we demonstrate that even a basic implementation of the decimal processor on the FPGA, without an advanced input/output interface, can achieve a speed-up of about 10 over its execution on the CPU of the hosting computer.

***Keywords*-FPGA accelerators; decimal arithmetic; financial applications;**

I. Introduction

Technology scaling and increased device density in chips make possible to have dedicated processors, Application Specific Processors (ASPs), which perform a specific set of instructions. These ASP are normally classified as hardware accelerators and operate on vectors of data [1].

However, having hardware accelerators on chip is quite expensive especially if the operations to be accelerated are not popular enough to sustain mass production volumes. As an alternative to keep costs low, accelerators can be placed on a different chip (and board) and connected to the CPU via a standard bus, such as the PCI Express bus. Clearly, in such a set up, the maximum achievable throughput is limited by the speed of communication memory-CPU-accelerator, but in this way it is possible to connect the accelerator to any CPU by dramatically decreasing production costs, and to make the system (CPU plus accelerator) easily upgradable.

The largest group of this connected-by-bus accelerators is probably constituted by Graphics Processing Units, or GPUs. The GPUs were originally introduced to off-load CPUs from graphics, but over the years they have evolved into general-purpose many-core systems [2].

Field Programmable Gate-Arrays (FPGAs) are chips in which the hardware can be programmed (configured) by the user at logic gate level to implement any processor. By connecting FPGAs (and boards) to the CPU via a standard bus, we can implement custom made accelerators at low cost. As in the case of GPUs, FPGA based accelerators can exploit data parallelism and might suffer from the communication bandwidth. Differently from the GPUs, FPGA accelerators can be fine tuned to match exactly the algorithm.

However, the last generations of general-purpose GPUs (GPGPUs) present massive parallel processors, high bandwidth on-board memory (DRAM) which make them as powerful as parallel computers [3]. For example *nVIDIA* Tesla 2nd generation (code-named Fermi [4]) boards are equipped with 448 cores clocked at 1.15 GHz, 3 or 6 GB of on-board DRAM and they can sustain a peak throughput of 1 or more Tera-FLOPS. These performances cannot be achieved by FPGAs which cannot be so densely configured to implement so many floating-point cores and which are normally clocked in the range 100–500 MHz.

However, hardware acceleration implemented on FPGAs can be competitive when processing "non-conventional" data. For example, very long integers and modular arithmetic used in cryptography, or financial (accounting) computation requiring the decimal number system.

In the following, we show that we can accelerate with a decimal processor implemented on FPGA the accounting typically done by telephone companies. As a case study, we consider a telephone billing application, described in Section III.

Results show that the execution of the benchmark on the FPGA based accelerator is about 10 times faster than the execution on the CPU and that the CPU–FPGA communication constitutes the bottleneck of the acceleration, in our case.

Newer reconfigurable platforms in which one or more processors can access the FPGA programmable logic by high-bandwidth interconnects [5], reducing the communication overhead, seems quite suitable for accelerating applications requiring decimal representation.

II. Decimal Arithmetic

Device shrinking, increased densities and the resulting extra space available on silicon dice, made possible to implement decimal processors in hardware.

The main drive for decimal units is the need in financial transaction and accounting for correctly rounded decimals that binary arithmetic cannot guarantee [6]. For example,

Binary Floating-Point (BFP) cannot exactly represent decimal fractions such as

$$10\% = \frac{10}{100} = (0.1)_{10} = (0.0001100110011001100 1 \ldots)_2$$

To overcome this loss of precision, financial applications implement decimal arithmetic operations in software and run 100–1000 times slower than the corresponding binary operations. Alternatively, Decimal Floating-Point (DFP) can be directly implemented in hardware and run order of magnitudes faster than the software computation. Some companies are already commercializing processors which include DFP units [7], [8].

For these reasons, in the revised IEEE standard 754 [9] support for decimal representation was added to the binary one.

In the following parts of this section, we briefly describe the IEEE standard 754 for Decimal Floating-Point (DFP) and the two DFP operations: addition and multiplication.

Moreover, we explain why deviating from the IEEE standard 754 without reducing the required precision, is beneficial in designing FPGA accelerators.

A. IEEE 754 DFP Formats

In the IEEE standard 754, the significand of DFP numbers can be represented in two different encodings: Densely Packed Decimal (DPD) format or Binary Integer Decimal (BID) format. Both encodings can represent significands with precision of 7, 16 and 34 decimal digits and base-10 exponents of 8, 10 and 14 bits (formats *decimal32*, *decimal64*, and *decimal128* respectively) [9]. Differently from binary FP, decimal FP significands are not normalized.

In the DPD format, a word representing the significand of a decimal number is divided into 10 bit fields, called *"declets"*, representing 3 decimal digits each. The conversion from declet to Binary Coded Decimal (BCD) encoding, and vice versa, can be implemented with simple logic [10].

In the BID format, a word represents the significand of a decimal number as an unsigned integer in binary. For example, the decimal floating-point number 0.125 is represented in BID by 125×10^{-3} with significand $0 \ldots 0\ 0111\ 1101_2$ and exponent $-3 + bias$.

The advantage of the BID format is that units designed for binary FP, can be used to perform operations on the significands (some range extension might be necessary). On the other hand, simple operations, such as operand alignment, or scaling by 10^k, require a multiplication on the significands.

Because scaling by 10^k is quite a frequent operation (used in rounding DFP significand, for example), we adopt the DPD/BCD representation in the implementation of the accelerator.

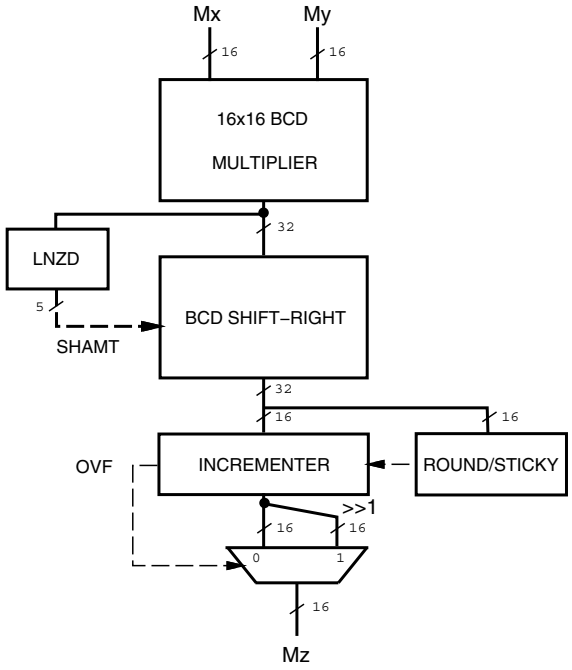

Figure 1. DFP multiplier for *decimal64* (significand computation).

B. DFP Addition

The addition of two floating-point numbers requires the alignment of the decimal point of the significands. In binary, this can be achieved by looking at the exponents of the two FP numbers and by shifting to the right the significand of the smallest number by as many positions as the exponent difference. In DFP, as the significands are not normalized, in addition to the exponent comparison, it is also necessary to detect the position of the first non-zero digit [11].

Once the significands are aligned, and complemented if the effective operation is subtraction, the significand addition is performed. Finally, the result is eventually shifted, rounded and packed (conversion BCD to DPD).

C. DFP Multiplication

To make the description of a DFP multiplier simpler, we consider the case of *decimal64* operands (significand is 16 digits and exponent is 10 bits).

Figure 1 sketches the block diagram of the hardware necessary to implement the significand computation. The scheme is completed by a binary adder to compute the exponent of the product and a XOR gate for its sign.

In Figure 1 wires in solid lines indicate BCD digits and wires in dashed lines binary digits (bits).

Block 16x16 BCD MULTIPLIER performs the BCD multiplication

$$P = M_X \times M_Y \ .$$

Both multiplicand and multiplier are not normalized (may contain leading 0s) and consequently the product is not normalized. The parallel multiplication shift-and-add algorithm

is based on the identity

$$P = M_X \times M_Y = \sum_{i=0}^{15} M_X M_{Y_i} 10^i$$

where for decimal operands M_X is a n-digit BCD vector and $M_{Y_i} \in [0, 9]$ is the BCD digit of M_Y of weight 10^i. To avoid complicated multiples of M_X, the digits of M_Y are recoded (as in [12], for example). The partial products are accumulated by using a carry-save adder tree ([12], [13], [14]), and the final product is computed by a BCD carry-propagate adder.

Block LNZD is a Leading Non-Zero (digit) Detector. The product P may have leading zeros. The LNZD detects the position of the leading non-zero digit and its output gives the shift amount. For example

```
P = 0000 0000 0000 00PP PPPP PPPP PPPP PPPP
                      ↑
                      k= 17
```

in this case the LNZD is in position of weight 10^{17} and P must be shifted SHAMT=k-16-1=2 digits to the right. If $k < 16$ no shift is performed (the product contains less non-zero digits than the representation).

Block BCD SHIFT-RIGHT shifts to the right the BCD digits of P according to SHAMT. The shifted out bits are used for rounding as explained next. For the above example

```
P = 0000 0000 0000 00PP PPPP PPPP PPPP PPPP
```

after the shift (SHAMT=2) we have

P = PPPP PPPP PPPP PPPP	T = PP00 0000 0000 0000
(to INCREMENTER)	(to ROUND/STICKY)

where T is the sticky bit.

The rounding operation is shown in Figure 2. Block INCREMENTER adds 1 to the 16 most-significant digits to perform the rounding. The increment is applied if the digit in round position (R) is greater than 4 (a carry is produced). Moreover, the sticky bits T are used to detect a tie.

If there is an overflow in the final increment, the result must be divided by 10 (1 digit shift right), and the exponent incremented. This is done by a multiplexer (bottom of Figure 1).

For the exponent part (represented in binary), the following computation is made (B is bias):

$$E_Z = (E_X + E_Y - B) + \text{SHAMT} + \text{OVF}$$

Finally, the sign is $S_Z = S_X \oplus S_Y$.

D. Deviation from IEEE 754

Although the main reason for standardization is to ensure that the same operation on the same data produce the same result independently of the processor/architecture, for some specific application deviating from the standard can result in area savings and faster execution time without any loss of precision. This is true for both decimal and binary floating-point.

```
0 PPPP PPPP PPPP PPPP R TTT TTTT TTTT TTTT
                        +5
-------------------------------
S ZZZZ ZZZZ ZZZZ ZZZZ
```

where

- S is the overflow bit (OVF in Figure 1)
- R is the digit to be rounded
- TTT TTTT TTTT TTTT are the digits to compute the sticky-bit (for ties)

Figure 2. Rounding in *decimal64* multiplier.

It might be convenient for a FPGA implementation of a processor, not to comply with the IEEE 754 standard and tailor the hardware to the specific application as long as the necessary precision is guaranteed. On the other hand, processors implemented on non-reconfigurable hardware (e.g. semi or full-custom integrated circuits) must comply with the standard to guarantee the necessary precision for all the possible applications.

Therefore, an FPGA implementation of accelerators gives more freedom in the design as the accelerator can be reconfigured (totally or partially) when the application is changed. This freedom in the design can lead to better processor performance and application optimized designs.

In our case, from the above descriptions, it is clear that both latency (delay) and area (and power dissipation) can be reduced for DFP addition and multiplication by deviating from the standard.

A first observation is that by concatenating two or more operations, we can avoid to pack (BCD→DPD) and un-pack (DPD→BCD) the data at the interface between operations.

However, more importantly, by deviating from the 7, 16, or 34 digit precision defined in the standard, we can significantly improve the performance. For example, in the case of the multiplier, having the exact number of digits necessary for the required precision may reduce the number of partial products, that may result in fewer level in the partial product accumulation tree (reduced delay and area).

Moreover, the hardware necessary for the rounding can be simplified according to the application specifications.

III. The TELCO Benchmark

The *"telco"* benchmark captures the essence of a telephone company billing application [15]. It was developed by IBM to investigate the balance between I/O time and calculation time. The benchmark provides an example of IEEE 754-2008 compliant set of DFP operations (multiplication add addition) and it can be executed on any computer[1]. The benchmark is based on the `decNumber` package [16] an implementation of the IEEE 754 decimal specifications in ANSI C.

We briefly summarize the benchmark's features. The benchmark reads an input file containing a list of telephone

[1]The benchmark is available in C and Java.

Type of call	L-type	D-type
Price rate	0.0013	0.00894
Base tax (Btax) rate	0.0675	
D-type tax (Dtax) rate	-	0.03410

Table I
PRICE AND TAX RATES IN THE TELCO BENCHMARK.

```
if (calltype = L)
   P = duration * Lrate;
else
   P = duration * Drate;
Pr = RoundtoNearestEven(P);
B = Pr * Btax;
C = Pr + Trunc(B);
if (calltype = D) {
   D = Pr * Dtax;
   C = C + Trunc(D);
  }
```

Figure 3. Pseudo-code of *"telco"* benchmark.

call times (in seconds). The calls are of two types (listed as L and D) and to each type of call a rate (price) is applied. Once the call price has been computed, one or two taxes (depending on the type of call) are applied. The price and tax rates, as fractions of some currency, are listed in Table I.

The price of the call must be rounded to the nearest cent (round-to-even in case of a tie), while the tax is computed by truncating to the cent. For example, for the L-type call 50 seconds long, we have[2]:

$$\begin{aligned} \text{Price} &= 50 \times 0.0013 = 0.065 \overset{ROUND}{\rightarrow} 0.06 \\ \text{Btax} &= 0.06 \times 0.0675 = 0.00405 \overset{TRUNC.}{\rightarrow} 0.00 \end{aligned}$$

and the total cost of the call is 0.06, or 6 cents.
For a D-type call of 329 seconds, we have:

$$\begin{aligned} \text{Price} &= 329 \times 0.00894 = 2.94126 \overset{ROUND}{\rightarrow} 2.94 \\ \text{Btax} &= 2.94 \times 0.0675 = 0.1984 \overset{TRUNC.}{\rightarrow} 0.19 \\ \text{Dtax} &= 2.94 \times 0.03410 = 0.10025 \overset{TRUNC.}{\rightarrow} 0.10 \end{aligned}$$

and the total cost of the call is

$$\text{Cost of call} = 2.94 + 0.19 + 0.10 = 3.23.$$

The pseudo-code of the accounting algorithm is listed in Figure 3. The benchmark is completed by computing the totals for costs and taxes.

IV. THE HARDWARE ACCELERATOR

A. The TELCO Processor

The processor to accelerate the execution of the *"telco"* application can be implemented by mapping the algorithm of Figure 3 on the hardware (multipliers and adders) of Figure 4.

As stated before, we avert from the IEEE 754 standard to optimize the latency and the FPGA resources. A detailed description of the design choices follows.

[2]In the price calculation there is a tie: 0.065 is exactly in the middle of the interval $[6, 7]$ cents, and the number is rounded to the even extreme.

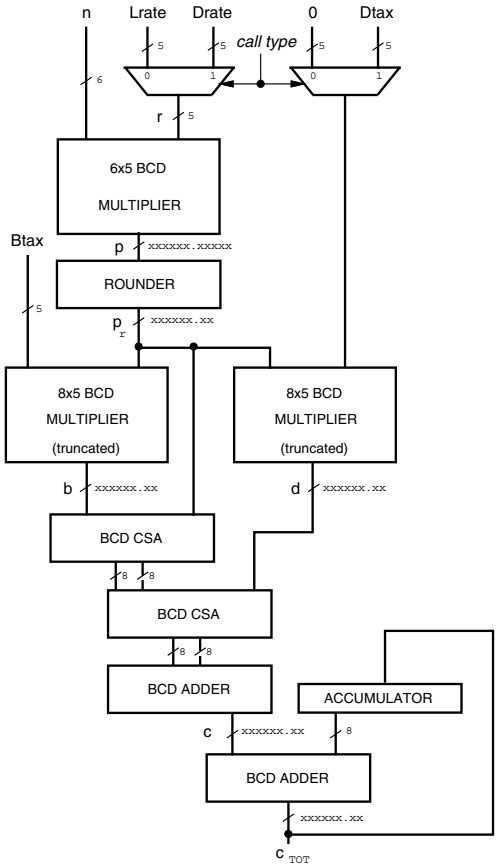

Figure 4. The TELCO processor.

$$\begin{array}{lcl} n & : & \texttt{xxxxxx.00000} \times \\ r & : & \texttt{0.xxxxx} = \\ \hline p & : & \texttt{xxxxxx.xxxxx} + \\ & & \texttt{5} = \\ \hline p_r & : & \texttt{xxxxxx.xx} \end{array} \quad \text{(rounding)}$$

Figure 5. Computation of call price.

1) Computation of call price: For this first step, we assume that the call duration (in seconds) is a 6-digit integer number n in the range $n \in [0, 10^6)$ ($10^6 s$ corresponds to 11.5 days), and a 5-digit fractional number for the call rate r. The call rate is selected by a multiplexer depending on the type of the call. The product $p = n \times r$, computed by a 6×5 BCD multiplier, is a 11 BCD digit number (5 fractional digits) to be rounded to p_r.

The synopsis of the operations in this first stage is in Figure 5[3].

2) Parallel computation of Btax and Dtax: The contribution of the two taxes (b and d) is computed in parallel. Both tax rates are represented by a 5-digit fractional number, and, consequently, two 8×5 BCD multipliers are used to

[3]In the synopses, symbols `x` represent a BCD digit $[0, 9]$, and `y` a bit $[0, 1]$, respectively.

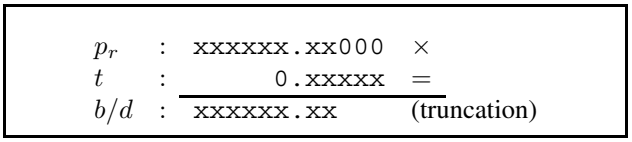

Figure 6. Computation of one of the taxes.

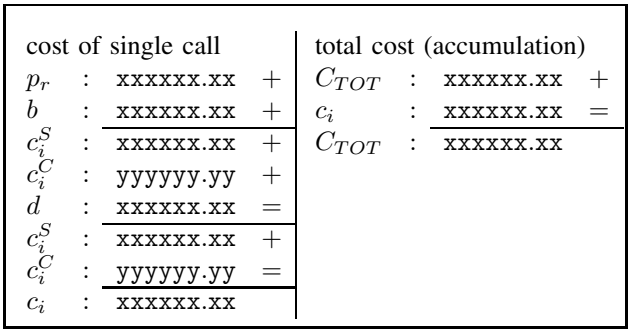

Figure 7. Computation of the total cost.

compute b and d. If the call is an L-type, the cost d is zeroed by forcing $Dtax = 0$. In this step, the results of the multiplications must be truncated to the cent (2nd fractional digit) as shown in the synopsis of Figure 6 (for one of the two multipliers).

3) Computation of the total cost: In the last stage, the two (or three) costs p_t, b (and d) are added to obtain the cost of the single call c_i, and the total cost

$$C_{TOT} = \sum_{i=1}^{\#\text{ calls}} c_i.$$

To speed-up the computation, some intermediate values are in BCD carry-save representation [12]. The synopsis is in Figure 7.

Figure 7 shows that the dynamic range of the integer part of the total cost is one million. This magnitude is considered adequate for the application.

B. The Hardware Platform

The hardware accelerator is implemented on the Alpha Data ADM-XRC-5T2 board (equipped by a Xilinx Virtex-5 LX330T FPGA) connected to the host PC through the PCI Express bus. The CPU of the host PC is the Intel Core2 Duo[4] processor clocked at 3 GHz.

The Alpha Data board is provided with a Software Development Kit (SDK) which is an extensive development platform, including an application-programming interface (API), VHDL functions and examples.

Data transfer host PC to FPGA (and vice versa) are handled by the Direct Memory Access (DMA). The implementation of the local bus interface and the DMA functions are included in the SDK.

C. The Hardware Implementation

The processor of Figure 4 (pipelined into 8 stages) is implemented on the FPGA of the Alpha Data board together with input and output FIFO buffers, a controller and some other glue logic. Data (call durations) are read from the input FIFO into the processor and costs are queued in the output buffer and sent back to the CPU.

[4]There is no support for DFP in this processor.

# calls	I/O time $[ms]$	I/O time $[\mu s]$	comp. time $[ms]$	comp. time $[\mu s]$	exec. time $[ms]$	exec. time $[\mu s]$
$1 \cdot 10^5$	40	(0.4)	210	(2.1)	250	(2.5)
$5 \cdot 10^5$	180	(0.4)	660	(1.3)	840	(1.7)
$1 \cdot 10^6$	300	(0.3)	1200	(1.2)	1500	(1.5)

Table II
BENCHMARK EXECUTION TIME (AND TIME PER CALL) ON HOST PC.

The delay of the critical path is about $8\ ns$ (maximum frequency is 125 MHz), but the processor is clocked at 80 MHz which is the frequency of the DMA.

As the input FIFO can be read every second clock cycle, the effective maximum frequency of operation is 40 MHz ($25\ ns$ per element) and the latency of the processor is $8 \times 25 = 200\ ns$.

The FPGA is quite large and less than 5% of the logic is utilized. However, as the bottleneck is the data transfer from/to the host computer (Section V), any further parallelization, or deeper pipelining, will not enhance the performance if the communication is not improved.

V. RESULTS AND COMPARISONS

In this section, we compare the performance of the software execution of the *"telco"* benchmark on the host PC CPU and the execution on the FPGA accelerator.

A. Host PC CPU Execution

To measure the performance of the benchmark on the CPU, we launched several runs with different data set sizes (number of calls) and computed the average for same sizes. For small sizes the I/O set-up time is dominant, therefore, we opted for large number of calls to estimate the computation time. Table II reports the measured values. The values between () indicate the average time to process a single call.

From the results of Table II, we can derive that for large sets the average I/O time per element is between 0.3–$0.4\ \mu s$ and the decimal computation time between 1.2–$1.3\ \mu s$.

Therefore, the software execution of the benchmark on the CPU is *computation-bound*.

B. FPGA Execution

Because we were mostly interested in measuring the performance of the decimal processor, we opted for the simple I/O scheme with an input and an output FIFO buffer. Unfortunately, this was not an adequate choice for large data sets, and in our testing we are limited to sets with hundreds of calls. We are working on modifying the I/O interface and on using the on-board DRAM to overcome the limitation.

The average execution time for the different data sizes and the time per element, between (), are listed in Table III. The time is read from a timer in the program (written in C)

# calls	execution time $[\mu s]$	$[\mu s]$
n = 100	T_{100} = 162	(1.62)
n = 200	T_{200} = 185	(0.92)
n = 300	T_{300} = 198	(0.66)

Table III
EXECUTION TIME (AND TIME PER CALL) ON FPGA ACCELERATOR.

to execute the benchmark on the FPGA as total execution time (I/O and computation time together).

To estimate the computation time per element (t_c), we assumed the I/O time t_{IO} constant for the three sets of calls, and calculated t_c on the increment from one set to the other[5]. For example, the computing time per element when the number of calls goes from 200 to 300 is

$$t_{c(A)} = \frac{T_{300} - T_{200}}{100} = 0.13\ \mu s = 130\ ns$$

while going from 100 to 300 elements results in

$$t_{c(B)} = \frac{T_{300} - T_{100}}{200} = 0.18\ \mu s = 180\ ns$$

Then, we calculate the I/O time per element as

$$t_{IO} = \frac{T_n - nt_c}{n} \Rightarrow t_{IO(A)} = 0.53\ \mu s,\ t_{IO(B)} = 0.48\ \mu s$$

This estimate shows that the decimal computation takes 130–180 ns per element, while the theoretical throughput is 40M elements per second (25 ns per element). Clearly, the accelerator is slowed down by the I/O communication, and the execution of the benchmark on the FPGA is *I/O-bound*.

VI. CONCLUSION

FPGA implementations of accelerators are attractive for a number of reasons including:

- Applications requiring non-binary number systems can be efficiently implemented as the processor can feature special operators, not present in CPUs and GPUs (e.g. BCD adders and multipliers).
- In FPGAs, deviation from standards (IEEE 754) or other conventional representations can be exploited to optimize performance, area and power dissipation.

In this work, as a case study, we present the implementation of a hardware accelerator for an accounting application requiring the use of decimal arithmetic. Instead of the IEEE 754 standard to represent decimal fractional numbers in the accelerator, we adopted a fixed-point BCD representation which guarantees the correct rounding in the different parts of the algorithm at lower delay and area costs.

By considering the computation time for the decimal part, the accelerator speed-up is about 10 (CPU 1200 ns vs. FPGA 130 ns). However, the FPGA computation time per element is about 5 times longer than the minimum theoretical time per element (25 ns).

[5]This is a conservative assumption as a fraction of the computed t_c time is actually spent in I/O.

Therefore, by redesigning the accelerator I/O interface to alleviate its penalty, the speed-up over the CPU execution should further improve.

ACKNOWLEDGMENT

The author wishes to thank Nicolas Borup and Jonas Dindorp for the data provided on the hardware implementation and on the performed experiments.

REFERENCES

[1] S. Patel and W.-M. W. Hwu, "Accelerator Architectures," *IEEE Micro magazine*, vol. 28, pp. 4–12, July/Aug. 2008.

[2] D. Luebke and G. Humphreys, "How GPUs Work," *IEEE Computer magazine*, pp. 96–100, Feb. 2007.

[3] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro magazine*, vol. 28, pp. 39–55, Mar./Apr. 2008.

[4] NVIDIA. "Fermi. NVIDIA's Next Generation CUDA Compute Architecture". Whitepaper. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[5] Xilinx Inc. "Zynq-7000 Extensible Processing Platform". [Online]. Available: http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm

[6] M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *Proc. of 16th Symposium on Computer Arithmetic*, June 2003, pp. 104–111.

[7] L. Eisen *et al.*, "IBM POWER6 accelerators: VMX and DFU," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 663–684, 2007.

[8] S. Carlough, A. Collura, S. Mueller, and M. Kroener, "The IBM zEnterprise-196 Decimal Floating-Point Accelerator," in *Proc. of 20th IEEE Symposium on Computer Arithmetic*, July 2011, pp. 139–146.

[9] *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society Std. 754, 2008.

[10] M. F. Cowlishaw, "Densely packed decimal encodings," *IEE Proceedings - Computers and Digital Techniques*, vol. 149, no. 3, pp. 102–104, May 2002.

[11] L.-K. Wang, M. J. Shulte, J. D. Thompson, and N. Jairan, "Hardware Design for Decimal Floating-Point Addition and Related Operations," *IEEE Transactions on Computers*, vol. 58, no. 3, pp. 322–335, Mar. 2009.

[12] T. Lang and A. Nannarelli, "A Radix-10 Combinational Multiplier," *Proc. of 40th Asilomar Conference on Signals, Systems, and Computers*, pp. 313–317, Nov. 2006.

[13] L. Dadda, "Multi Operand Parallel Decimal Adders: a mixed Binary and BCD Approach," *IEEE Transactions on Computers*, vol. 56, pp. 1320–1328, Oct. 2007.

[14] A. Vazquez, E. Antelo, and P. Montuschi, "A new family of high-performance parallel decimal multipliers," *Proc. of 18th Symposium on Computer Arithmetic*, pp. 195–204, June 2007.

[15] IBM Corporation. "The "telco" benchmark". [Online]. Available: http://speleotrove.com/decimal/telco.html

[16] International Components for Unicode (ICU). "The `decNumber` package". [Online]. Available: http://download.icu-project.org/files/decNumber/decNumber-icu-368.zip