THE ALGORITHMS FOR FPGA IMPLEMENTATION OF SPARSE MATRICES MULTIPLICATION

Ernest Jamro, Tomasz Pabiś, Paweł Russek, Kazimierz Wiatr

AGH University of Science and Technology, Department of Electronics
Al. Mickiewicza 30, 30-059 Krakow, Poland
e-mail: {jamro, russek, wiatr}@agh.edu.pl, pabis.tomek@gmail.com

Abstract. In comparison to dense matrices multiplication, sparse matrices multiplication real performance for CPU is roughly 5–100 times lower when expressed in GFLOPs. For sparse matrices, microprocessors spend most of the time on comparing matrices indices rather than performing floating-point multiply and add operations. For 16-bit integer operations, like indices comparisons, computational power of the FPGA significantly surpasses that of CPU. Consequently, this paper presents a novel theoretical study how matrices sparsity factor influences the indices comparison to floating-point operation workload ratio. As a result, a novel FPGAs architecture for sparse matrix-matrix multiplication is presented for which indices comparison and floating-point operations are separated. We also verified our idea in practice, and the initial implementations results are very promising. To further decrease hardware resources required by the floating-point multiplier, a reduced width multiplication is proposed in the case when IEEE-754 standard compliance is not required.

Keywords: FPGA, sparse matrices, sparse BLAS, matrices multiplication

Mathematics Subject Classification 2010: 65F50

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are alternative computing platform to microprocessors – CPUs (Central Processing Units) or GP-GPUs (General Purpose Graphics Processing Unit). Computation power for floating-point operations of FPGAs are roughly similar to CPUs [1, 2]. Nevertheless, data transmission overheads, a difficult and much more time-consuming designing process, result that FPGAs are rarely employed for floating-point intensive computations. Conversely, FPGAs are often employed for image processing for which limited bit-width (8 or 16-bit) integer operations are commonly employed [3]. Similarly for DSP (Digital Signal Processing) integer operations (especially 16-bit or less), computation power of FPGAs significantly overpass that of CPUs. For example, Xilinx Virtex 7 FPGAs contain up to 3 600 DSP modules, thus they can perform up to 5.3 TMAC/s (MAC – Multiply and ACcumulate) [4], which significantly overpass the computation power of any current CPU or DSP.

For CPUs, in opposite to dense matrix multiplication, sparse matrix multiplication sustained performance is roughly 5-100 times smaller when expressed in GFLOPs [5, 6, 7]. For sparse matrices, processors spend most of the time on comparing matrix indices rather than on performing floating-point multiply and add operations. Therefore, e.g. in [8] indices bit-widths were shortened to 16-bit in order to limit the memory bandwidth overheads. For 16-bit integer operations, i.e. indices comparisons, computational power of the FPGA significantly surpasses that of CPU. According to the authors' knowledge, very few FPGA architectures for sparse matrices multiplication have been published until now. In [9, 10, 11] only sparse matrix-vector multiplication is addressed. However, for matrix-vector operations, a ratio of computation complexity to data transfer size is low, therefore the data transfer overhead often ruins the speed-up offered by the FPGA. The only paper that addresses sparse matrix-matrix multiplication is [12]. This paper considers relatively high matrices density of 100 %, 30 %, 20 % and 10 %, where the 100 %-density is used to represent the dense matrix-matrix multiplication. Nowadays when CPU and GP-GPU employs SIMD/SPMD (Single Instruction Multiple Data/Single Program Multiple Data) operations and are highly optimized for dense matrix-matrix multiplication, density should have a factor of 10 % or lower in order to efficiently implement sparse rather than dense matrix-matrix multiplication algorithm. This holds when computation power and not the memory size is the key limiting factor. For example in [6], DGEMM performance is roughly 180 GFLOPS compared to up to 18 GFLOPS for sparse matrix-vector computations.

Paper [12] considers the power dissipation rather than the computation power of the solution. Every Processing Element (PE) contains arithmetic unit; thus it is not highly optimized for sparse operations, for which, as it will be proved in this paper, indices comparisons dominate the floating-point operations. Therefore a novel architecture is proposed, that is highly parallel, able to carry out 8×8 (or more) indices comparison in a single clock cycle.

The organization of this paper is as follows. Chapter 2 considers how sparsity of matrices influences the integer comparison to floating-point arithmetic operations workload ratio. This novel theoretical study, verified in practice, demonstrates that the larger the matrices sparsity, the more workload should be allocated to indices comparison. Chapter 3 introduces a novel architecture for highly-parallel indices comparison. Chapter 4 presents promising implementation results. Chapter 5 proposes a novel reduce-width floating point multiplier.

2 THE SOFTWARE ALGORITHM FOR SPARSE MATRICES MULTIPLICATION

In this paper, similarly to [7, 9], Compress Sparse Row (CSR) for matrix A and Compress Sparse Column (CSC) for matrix B representations are employed, where $A \times B$ is the performed matrix-matrix multiplication. Examples of the CSR and CSC representations are given in Figure 1.

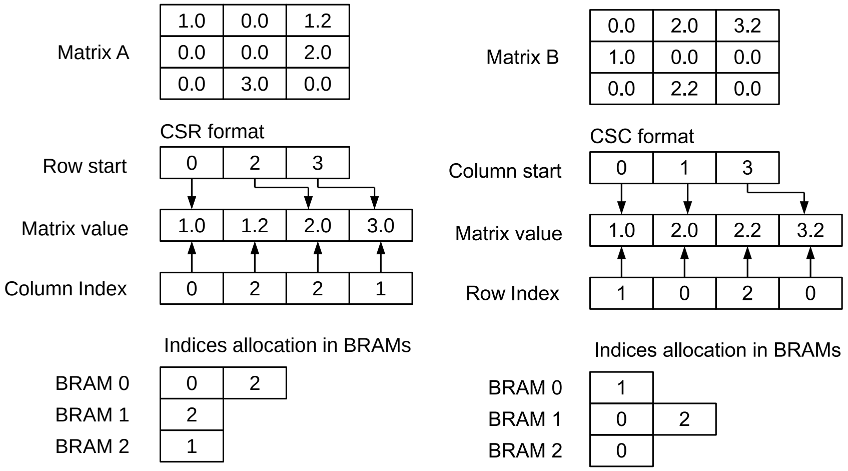

Figure 1. Examples of Compress Sparse Row and Compress Sparse Column representations for $A * B$

In the proposed architecture for matrix A, column indices are stored in internal FPGA memory denoted as Block RAM (BRAM) [4], a separate BRAM memory for each row. Similarly for the matrix B, row indices are stored in a separate BRAM memory for different columns.

Consider the matrices multiplication of $A * B$, where matrix A has density dA and a size $N \times N$ (a square matrix is considered, however the same methodology can be derived for any rectangular size), and matrix B has density dB and size N. Density dA (dB) is defined as the number of nonzero elements to the total number of elements N^2. Consequently, the number of nonzero elements in every row and column is on average equal to $dA * N$ (denoted as daN in Listing 1) for matrix A and $dB * N$ (dbN) for matrix B.

The matrix-matrix multiplication incorporates many vector-vector dot products $a * b$, thus for the simplicity, only a vector-vector operation will be considered in this section. An algorithm for the sparse vector-vector dot product $a * b$ is presented in

Listing 1. The most important part of this algorithm is indices comparison $ia[ja]$ and $ib[jb]$ and taking the next nonzero index (incrementing ja or jb) according to the comparison results. It should be noted that both values ja and jb are incremented and multiplication is performed only if indices $ia[ja]$ and $ib[jb]$ are the same.

```
double a[], b[];// nonzero values of vectors a, b
                // in a compressed form
int ia[], ib[]; // table of indices for nonzero elements in
                // in vector a, b - same as e.g. column index 0 in Figure 1.
double product= 0.0;

// for (almost) every element in ia[] and ib[]
for(int ja=0, jb=0; ja<daN && jb<dbN; )
{
   if (ia[ja] < ib[jb]) // index ia[ja]<ib[jb] so take the next index ia[]
   {
      ja++;
   {
   else
      if (ia[ja] == ib[jb]) // multiplication is performed
      {
         product+= a[ja] * b[jb];
         ja++; jb++; // take the next indices ia[] and ib[]
      }
      else // ia[ja] > ib[jb] - take the next index ib[]
      {
         jb++;
      }
}
```

Listing 1. Sparse vector-vector dot multiplication algorithm

From Listing 1, the following conclusions can be driven. The number of indices comparisons L_N (the loop iterations) is equal to:

$$L_N \approx B * (d_A + d_B) \tag{1}$$

provided that no multiplication is performed (no indices match occurs), where $d_A * N$, $d_B * N$ are the number of nonzero elements in the vector a, b respectively. For nonzero elements located only in the left-part of the vector a, and nonzero elements located only in the right-part of vector b, the number of iterations might be as low as $L_N = d_A * N$. However the assumption is made that nonzero elements in the vector a and b are randomly distributed and therefore (1) is satisfied.

If the L_M is the number of performed multiplications, the total number of the loop iterations L is equal to

$$L \approx L_N - L_M. \tag{2}$$

Equation (2) holds as for the indices match (multiplication is performed), both ja and jb are incremented (see Listing 1); otherwise, either ja or jb is incremented.

Now, let us assume that nonzero elements in the vector a and b are randomly distributed. For an arbitrary taken index $ia[ja]$ of nonzero element of the vector a, the probability that the corresponding element of the vector b (index $ib[jb]$) is also nonzero (i.e. $ia[ja] == ib[jb]$) is equal to d_B, where d_B is the density of the vector b. Consequently, the average total number of nonzero indices matches, i.e. the number of performed multiplications, L_M, is

$$L_M = N * d_A * d_B. \tag{3}$$

The total number of the loop iterations in Listing 1, L, is therefore equal to

$$L \approx L_N - L_M \approx N * (d_A + d_B - d_A * d_B). \tag{4}$$

Concluding the discussion, the ratio R for the number of performed multiplication L_M to the total number of the loop iterations L is equal:

$$R = \frac{L_M}{L} \approx \frac{d_A * d_B}{d_A + d_B - d_A * d_B} \approx \frac{d_A * d_B}{d_A + d_B}. \tag{5}$$

For $d_A = d_B$, (5) can be further simplified to

$$R \approx d_A/2. \tag{6}$$

For example, for $d_A = d_B = 0.1$, the ratio $R \approx 0.05$, thus the number of multiplications L_M is 20 times lower than the total number of the loop iterations L. The architecture of the sparse matrix-matrix multiplication should be able to perform 20 times more indices comparisons to floating-point multiply and add operations. Consequently, indices comparison should be carried out independently to arithmetic operations as it is presented in Figure 2. Thus the indices comparison block should pass to the floating-point arithmetic unit only indices of matrix A and corresponding indices of matrix B for which arithmetic operation are required. In order to separate indices comparisons from floating-point arithmetic, an additional FIFO (First-In First-Out) buffer is employed. Consequently, local grouping of nonzero values, where high number of arithmetic operations are required, may be distributed in time by the FIFO buffer. Thus average (not peak) number of floating-point arithmetic operations should be considered in the arithmetic module.

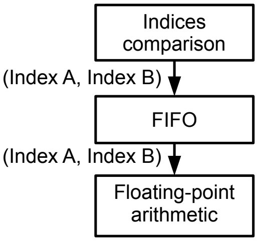

Figure 2. The idea of independent indices comparison and floating-point arithmetic modules

3 ARCHITECTURES FOR PARALLEL MATRICES MULTIPLICATION

A typical (e.g. [13, 14]) architecture for parallel dense matrix-matrix multiplication is given in Figure 3. In this architecture $n \times m$ floating-point multiplications and additions are carried out in a single clock cycle. Row 0 of matrix A is stored in RAM $A0$. Row 1 of Matrix A is stored in RAM $A1$, and so on. Column 0 of matrix B is stored in RAM $B0$ and so on. Consequently $n \times m$ results are calculated at the time. The most important advantage of this architecture is that only $n + m$ memory blocks are required; the output of a single memory block feeds many arithmetic modules. This is very important as in many applications internal memory size limits the level of parallelism.

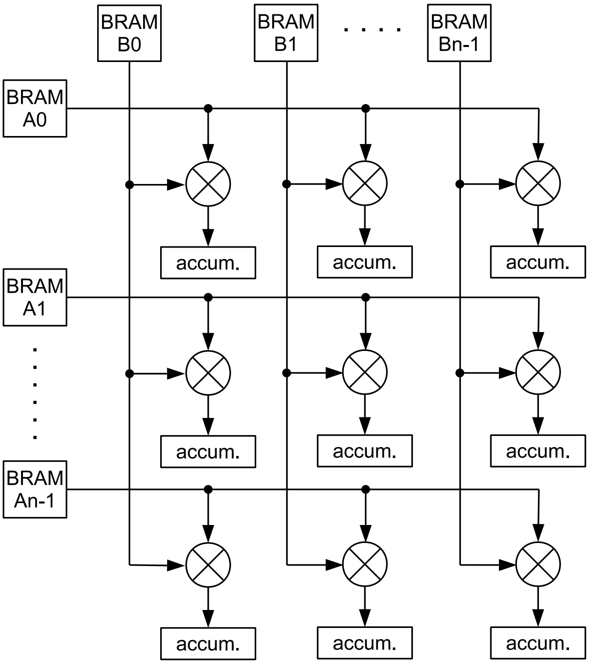

Figure 3. A typical architecture for dense matrix-matrix multiplication

For efficient sparse matrix-matrix multiplication the above architecture cannot be adopted straightforward, as multiplication process is not regular. According to Listing 1 for sparse operations, the memory address ja, jb or both can be incremented in a single iteration. For dense matrix operations both addresses are always incremented. Therefore, novel architectures should be proposed in order to compare indices in parallel.

3.1 Independent Parallel Indices Comparison (IPIC)

For the proposed solution denoted as Independent Parallel Indices Comparison (IPIC), each indices comparison is carried out separately, thus has its own memory and comparison module. A single module should be able to perform a single indices comparison and two independent (index) memory accesses in a single clock cycle. In FPGAs, blocks RAM (BRAM) are the dual port memory; thus two independent memory accesses can be delivered in a single clock cycle. Therefore, in order to better utilize the BRAM memory, the same contents of memory should be used on two ports. Consequently, a basic building block for the IPIC is given in Figure 4. It performs 2×2 comparisons and contains a single BRAM per comparison. In order to further speed up the comparison, tens or hundreds of instances of these blocks should be employed in parallel.

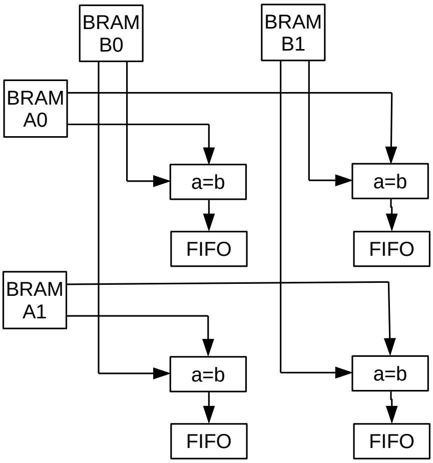

Figure 4. 2×2 Independent Parallel Indices Comparison (IPIC)

For this architecture the level of parallelism is limited by the number of BRAMs. This holds as a comparison process requires insignificant number of FPGA resources. For example, the largest Virtex 7 XC7VX1140T contains 3 760 BRAMs [4], each of them can hold up to 1 024 16-bit words (an assumption is made that indices are 16-bit wide, thus the matrix size $N < 2^{16}$). Considering that only a half the number of memory blocks is assigned to indices comparison (another part of BRAMs is assigned to hold matrices values), up to 1 800 comparisons can be carried out in a single clock cycle. Assuming matrix density $d_A = d_B = 0.1$, i.e. indices comparison to arithmetic ratio is equal to 20, only 90 floating-point multiply and add modules are required. For 200 MHz clock frequency, the FPGAs are able to achieve up to $90 * 2 * 200\,\text{MHz} = 36\,\text{GFLOPS}$.

3.2 FIFO-based Parallel Index Comparison (FPIC)

The IPIC throughput is limited by the amount of BRAM rather than by available FPGA logic resources. Consequently higher level of parallelism for indices comparison is required. Analyzing Figure 4, a conclusion can be drawn that the same data but in different time-slots are read at port A and port B of a single BRAM. Consequently, by insertion of a small FIFO, whose functionality resembles cache memory, a greater number of memory accesses can be obtained. The greater the FIFO depth, the smaller ports dependency, but higher hardware requirements. As FIFOs in FPGAs usually incorporate SRLs (Shift Register LUT) logic and a 32-bit depth SRL fits into a single LUT for Virtex 6 family, a 32-bit FIFO depth is chosen. In the proposed architecture, a single BRAM port feeds 4 FIFOs, thus a single BRAM with additional FIFOs feeds 8 data ports. A block diagram of the proposed indices comparison module for 8×8 parallel comparison is given in Figure 5. In the authors' opinion the proposed architecture has the best ratio of BRAM memory to logic resources, especially for density 10 % or lower. For higher parallelism several 8×8 FPIC modules should be implemented.

4 FPGA IMPLEMENTATION

Table 1 gives implementation results for 8×8 FPIC, for different matrices of size N and density roughly 10 %. As can be seen, the number of required clock cycles is significantly reduced. The effective parallelism is around 43 to 60, where 64 is theoretical maximum value (8×8 parallelism). The higher the matrix size the better the performance. This holds as the matrices were randomly generated, thus statistically the greater the matrix size, the more similar the relative number of nonzero values in each row and column.

N	Density [%]	# clock cycles	effective parallelism
100	10.5	4 179	43.6
250	9.83	56 639	51.0
500	9.67	409 705	55.1
1 000	9.61	3 032 541	59.7

Table 1. Implementation results for FPIC 8×8

The drawback of the FPIC architecture is that it works poorly for non-uniform distribution of nonzero values. The throughput of the entire 8×8 module is limited by the slowest row-column comparison. Consequently, for strongly non-uniform distribution of nonzero matrix values, 2×2 IPIC architecture should be implemented. Alternative solution is to implement 4×4 FPIC, i.e. each BRAM port feeds only two (instead of four for 8×8 FPIC) FIFO buffers.

Typical hardware requirements for different modules employed in the design is given in Table 2. Two different types of floating-point multipliers are considered:

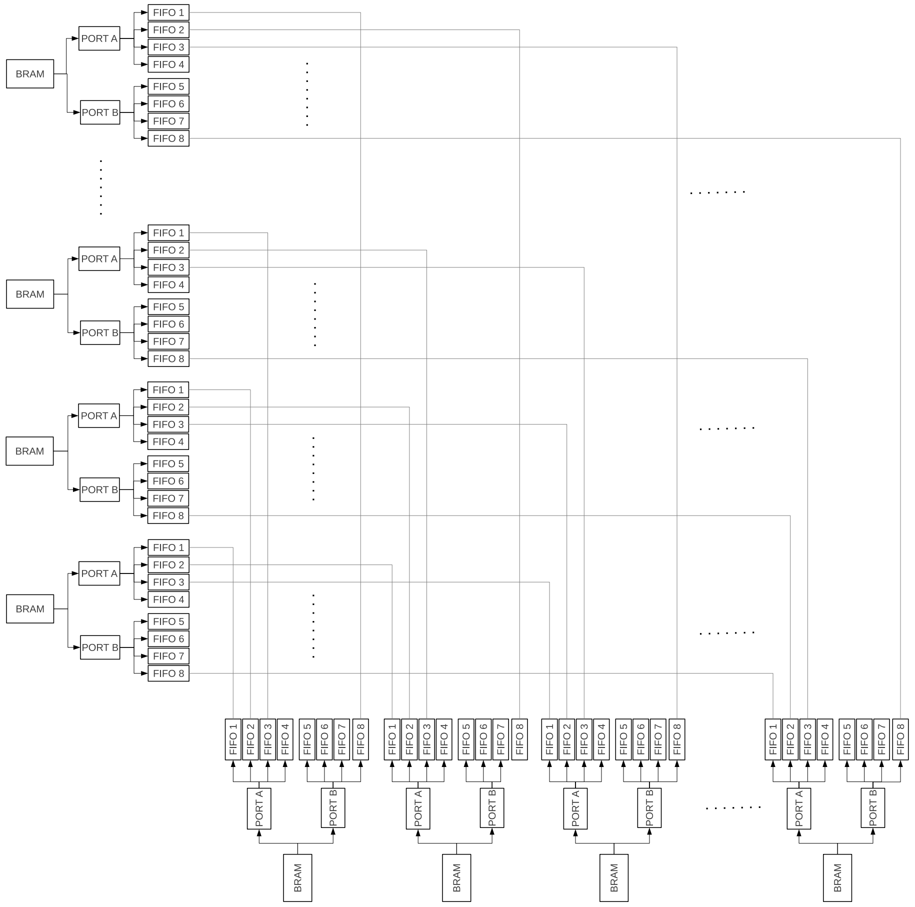

Figure 5. The 8 × 8 FIFO-based Parallel Index Comparison (FPIC)

type 1 – utilization of only logic resources (LUTs), type 2 – utilization of logic and dedicated 25 × 18-bit integer multipliers (DSP48 modules), rows 3 vs. 4, and 5 vs. 6 in Table 2. The largest Xilinx Virtex 7 chip can incorporate roughly 920 single and 370 double precision floating-point multiply and add modules (row 8 and 9 in Table 2); these numbers are theoretical. For double precision operation the number of available DSP48 rather than logic resorces limits the level of parallelism. In real applications, up to roughly 50–80 % of FPGA logic resources should only be used (otherwise the clock frequency drops dramatically) and assumption is made that half of the logic resources is reserved by the indices comparison, external memory and control logic. The DSP48 modules are used only by the floating-point multiplier, therefore, in the final system they do not limit the level of parallelism. Thus the real number of floating-point modules is estimated as 250 single and 150 double precision. Assuming 200MHz clock frequency, the throughput is roughly $250 * 2 * 200\,\text{MHz} = 100\,\text{GFLOPS}$ single and $150{*}2{*}200\,\text{MHz} = 60\,\text{GFLOPS}$ double precision operations. The additional multiplication by 2 is to account for "multiply and add" operations.

	Module	# LUT	# FF	#BRAM 18k	#DSP48
1	32-bit floating-point adder	404	546	0	0
2	64-bit floating-point adder	730	944	0	0
3	32-bit floating-point multiplier1	692	672	0	0
4	32-bit floating-point multiplier2	396	360	0	1
5	64-bit floating-point multiplier1	2 455	2 422	0	0
6	64-bit floating-point multiplier2	463	541	0	9
7	XC7VX1140T	712 000	1 424 000	3 760	3 360
8	max number of 32-bit fmul + fadd row: $7/(1+4)$	921	1 571		3 360
9	max number of 64-bit fmul + fadd row: $7/(2+6)$	569	959		373

Table 2. Implementation results for selected modules

Sparse matrix-vector multiplication throughputs on GP-GPU presented in [15] are 16 and 10 GFLOPS for single and double precision operations, respectively. Similar results were also presented in [6]. It should be noted that FPGAs significantly outperform counterparts in the case when a matrix sparsity is unstructured and randomly distributed. In the case when a matrix structure is known, a dense approach is usually employed, e.g. for a band matrix, the BLAS-General-Band Storage Mode should be used.

In many cases the sparse indexing overheads can be reduced by recognizing patterns in the structure of nonzero coefficients. For example, the matrix may entirely consist of small fixed-size, dense rectangular blocks. Then, we can use a block variant of CSR – a block CSR, or BCSR, format which stores one index per block instead of one per nonzero [7]. As a result, indices comparison overhead is reduced at a cost of increase in the number of unnecessary floating-point operations, i.e. some calculations are performed on zero elements. It should be noted that the BCSR format might be also introduced in the FPGA, however indices comparison overhead in comparison to floating-point operations overhead is less destructive than in the case of CPUs. Summing up, a speed-up factor for CPU and FPGA for sparse matrices multiplication is application-depending, thus cannot be easily generalized. Therefore this paper focuses only on randomly generated sparse matrices. The less dense and more randomly distributed sparse matrices are, the larger is the speed-up factor for FPGA. This holds as the indices comparison to floating-point workload ratio is greater, and FPGAs can significantly outperform CPUs for reduced-width integer operations.

5 FLOATING-POINT MULTIPLIERS

5.1 IEEE-754 Standard Requirements

Both Central Processing Units (CPU) and Graphic Processing Units (GPU) incorporate high throughput floating-point multipliers. These multipliers comply with IEEE-754 standard, either Single-Precision (SP; 32-bit), Double-Precision (DP; 64-bit) or extended double precision (roughly 80-bit). Therefore multipliers structure and bit-width are strictly defined. A processor designer selects a proper architecture of a multiplier. It suits a certain data representation of arguments. A different type of arguments may be programmed on these platforms, nevertheless they will be emulated in software and therefore executed significantly more slowly. It should be noted that the computing performance strongly depends on multiplication precision. CPUs usually doubles their performance for Single-Instruction Multiple-Data (SIMD) operations when multiplication is carried out on the SP instead of DP. For some GPUs a speed-up factor is even higher as they are optimized for the SP operations. For example: AMD Radeon HD 4870 peak performance is 1.2 TFLOPS for SP and only 240 GFLOPS for DP, five times the performance gap.

The selection of the calculation precision is more complicated for FPGA designs for which precision can be selected without restraints. The significant and exponent bit-widths can be custom-defined. For example, Xilinx CORE Generator [16] can generate a floating-point multiplier for fraction widths $4 \ldots 64$, and exponent widths $4 \ldots 16$. Reduction of a data width causes less FPGA resources consumption per multiplier, thus greater number of the parallel multipliers can be implemented and higher computation power is achieved.

Unfortunately, the HPC society is strongly committed to the DP calculations, mostly due to software backward compatibility. Nevertheless, a research is required whether some DP computation can be substituted by the SP or other user-defined precision counterparts. For example, there are a great number of iterative algorithms for which the result precision (thus also calculation precision) increases with an iteration number, e.g. Newton-Raphson method or conjugate gradient method. Therefore initial iterations can be calculated in reduced custom-defined precisions and IEEE-754 standard compliance may be required for final iterations only.

In principle, IEEE-754 standard was defined to allow for effortless migration between different platforms, i.e. the computation results should be the same on different platforms. Nevertheless, no IEEE-754 compatibility is required for computations carried out on e.g. 48-bit data as these computations can be efficiently performed only on FPGA and therefore there is no need to be compatible with different platforms. The same results can be obtained on different FPGAs by implementing the same IP-cores rather than by compiling with the IEEE-754 standard.

Analyzing IEEE-754 standard, a conclusion can be drawn that it was optimized for storage rather than calculation resources. For example, incrementing exponent width by one bit instead of supporting denormalized numbers would significantly increase the represented numbers range and probably reduce resources occupied by arithmetic modules. Nevertheless, one extra bit is required to store such numbers. Therefore, the idea behind this chapter is to optimize intermediate data format in order to reduce arithmetic resources. This holds especially for intermediate data which are not stored in the memory or for data whose width is not compact to the memory interface width (e.g. 50-bit data for 64-bit memory interface). Summing up, the main idea presented in this section is to stop complying with IEEE-754 standard in order to reduce hardware resources. It should be noted that the calculation errors obtained by custom modules may not be greater than for IEEE-754 compliant ones but calculation results might be slightly different.

5.2 Structure of Floating-Point Multipliers

The structure of a floating-point multiplier is described e.g. in [17] and presented in Figure 6. The most area consuming module is the significant multiplier which is a standard unsigned integer multiplier. The inputs bit-width of this multiplier, n, is equal to the significant bit-width plus one (additional hidden '1' of the mantissa). The integer multiplier output bit-width is equal to $2 * n$. Then, the result is normalized and rounded. Additional module "Normalize 2" is required in a case of rounding overflow. The final mantissa width (together with the hidden '1') is equal to n, therefore a great number of the integer multiplier output bits are disregarded. The above scheme is required due to IEEE-754 standard complaints.

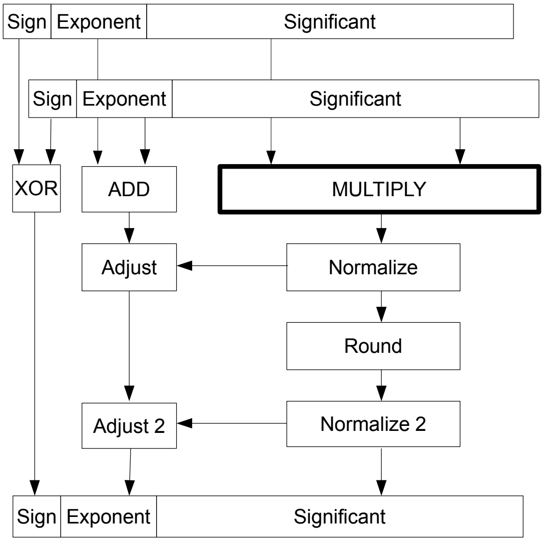

Figure 6. Block diagram of the standard floating point multiplier

The main idea behind this chapter is to design hardware-optimized floating-point multiplier which is not IEEE-754 compliant. The integer multiplier produces

$2n$-bit result, but only $n+2$ MSBs are further used to produce the final significant. These two additional bits $(n+2)$ are required during normalization (where the integer multiplier result might be shifted one bit to the left) and during rounding (assuming rounding-to-nearest-up [17]). Summing up, as $(n-2)$ output bits of the integer multiplier are disregarded, a reduced-width multiplier with a reasonable error might be employed instead of the full-width integer multiplier.

5.3 Reduced Width Multipliers

The idea is to carry out multiplication in such a way that $(n-w)$ LSBs of the results are ignored during the multiplication process where w is a number of guard bits, see Figure 7.

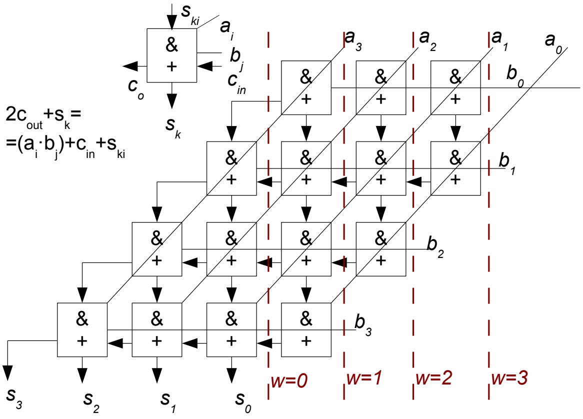

Figure 7. Block diagram of the direct truncated fixed width multiplier

Consequently, a multiplier area is significantly reduced at a cost of an additional calculation error. Analyzing Figure 7 it can be seen that the hardware resources are roughly halved for $w=0$. A truncation error can be significantly decreased by adding an error compensation logic, e.g. [18, 19]. Nevertheless, these error compensation methods are dedicated to ASIC (Application Specific Integrated Circuit) and are not optimal for FPGA designs. An FPGA-optimized reduced width multiplier is presented in [20]. This multiplier does not require additional FPGA resources in comparison to the direct truncation multiplier as the truncated carry path is fed by the input bits a_i or b_j.

In this paper an assumption is made that multipliers are implemented in Configurable Logic Blocks (CLBs) [4]. However, dedicated multipliers incorporated in DSP blocks [4] are now available in FPGAs and often they are preferable to CLB-type of multipliers. Nevertheless, according to the authors' experience and the publication [21], using dedicated multipliers complicates routing and often leads to additional propagation delays. This holds as the dedicated multipliers are located in the particular FPGAs sites. Besides, the number of dedicated multipliers is limited for a given FPGA, and in many cases a combination of dedicated and CLB-type multipliers can increase design functionality. Similarly a combination of CLB-type and dedicated multipliers can be used within a single multiplier, e.g. in a 24 × 24-bit multiplier when 17 × 17-bit (Xilinx Spartan) or 17 × 24 (Xilinx Virtex) unsigned built-in multipliers are available. This combination is especially recommended for reduced-width multipliers.

5.4 Implementation Results

In order to properly evaluate the proposed multiplier architecture, the following calculation error statistics are employed:

- Mean Error: $ME = \frac{\sum_1^N e_i}{N}$
- Mean Absolute Error: $MAE = \frac{\sum_1^N |e_i|}{N}$
- Root Mean Square Error: $RMSE = \sqrt{\frac{\sum_1^N e_i^2}{N}}$
- Maximal error: $E_{\max} = \max(|e_i|)$

where e_i is the difference between the correct result (obtained by the full-width multiplier) and the result obtained for the i^{th} sample. Results are given in Table 3. The error is expressed in Unit Last Places (ULP). The second column corresponds to the type of multiplier: a direct truncation (dir) and the FPGA-optimized truncated multiplier (opt) (see [21]).

Error	Type	$w=0$	$w=1$	$w=2$	$w=3$	$w=4$	$w=5$	$w=6$	round
ME	dir.	5.752	2.751	1.313	0.625	0.297	0.141	0.066	0
	opt.	0.255	0.001	0.063	0.000	0.016	0.000	0.004	0
MAE	dir.	5.752	2.751	1.313	0.625	0.297	0.141	0.066	0.250
	opt.	1.478	0.714	0.350	0.169	0.084	0.040	0.020	0.250
RMSE	dir.	6.031	2.890	1.382	0.660	0.314	0.149	0.071	0.289
	opt.	1.852	0.892	0.439	0.211	0.105	0.050	0.025	0.289
$E_{\max}$	dir.	15.977	7.977	3.774	1.899	0.909	0.440	0.215	0.500
	opt.	9.191	4.477	2.116	1.089	0.497	0.240	0.125	0.500

Table 3. Truncation errors for a direct truncation and the FPGA-optimized error compensation ($n = 24$)

The column 'round' in Table 3 indicates the perfect rounding error. As can be seen from Table 3, truncation errors for reduced-width multipliers and for additional guard bits $w = 4 \dots 6$ are significantly lower than for the perfect rounding error. Consequently, it is recommended to add one extra (guard) bit and to employ reduced-width multiplier rather than to employ a standard full-width multiplier. This holds both for the calculation error and occupied FPGA area.

6 CONCLUSIONS

In this paper, sparse matrix-matrix multiplication was studied. In comparison to the previous works, a high throughput can be achieved in the proposed architectures by separation of the indices comparison and a floating-point arithmetic modules. This is based on the novel theoretical study (verified in practice), that the number of indices comparison to floating-point multiply and add operations ratio is equal to matrix density divided by 2. Therefore for matrix density equal to 10 %, the number of comparison modules should be 20 times greater than the number of multiply and add modules. For lower density the ratio should be even higher. It should be noted that indices comparison requires insignificant FPGA hardware resources; thus highly parallel architectures for indices comparison are proposed. Summing up, the proposed architecture offers significant speed-up in comparison to the CPU and GP-GPU, especially in the case when the matrices sparsity is unstructured and randomly distributed.

The last section is dedicated to floating-point multiplication which can be also significantly optimized in the case when IEEE-754 standard compliance is not required. In this case, a reduced-width multiplier should be employed for which multiplier area is significantly reduced at the cost of additional truncation error. As proved in Table 3, the truncation error is significantly lower than perfect rounding error in the case when 4 . . . 6 extra guard bits are added.

Acknowledgement

This scholarly work was made thanks to POWIEW project. The project is co-funded by the European Regional Development Fund (ERDF) as a part of the Innovative Economy program.

REFERENCES

[1] Strenski, D. et al.: Latest FPGAs Show Big Gains in Floating Point Performance. HPC Wire, April 16, 2012.

[2] Wielgosz, M.—Mazur, G.—Makowski, M.—Jamro, E.—Russek, P.—Wiatr, K.: Analysis of the Basic Implementation Aspects of Hardware-Accelerated Density Functional Theory Calculations. Computing and Informatics, Vol. 29, 2010, No. 6, pp. 989–1000.

[3] Kryjak, T.—Marek, G.: Real-Time Implementation of Moving Object Detection in Video Surveillance Systems Using FPGA. Computer Science, Vol. 12, 2011, pp. 149–162.

[4] Xilinx Inc. 7 Series FPGAs Overview, `www.xilinx.com` DS180 (v1.10), March 2, 2012.

[5] Vuduc, R.—Demmel, J.—Yelick, K.—Kamil, S.—Nishtala, R.—Lee, B.: Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In Proceedings of IEEE/ACM Conference on Supercomputing, November 2002.

[6] `http://www.siliconmechanics.com/files/C2050Benchmarks.pdf`, Tesla C2050 Performance Benchmarks.

[7] Demmel, J.—Dongarra, J.—Eijkhout, V.—Fuentes, E.—Petitet, A.—Vuduc, R.—Whaley, R. C.—Yelick, K.: Self-Adapting Linear Algebra Algorithms and Software. Proceedings of the IEEE, Vol. 93, 2005, No. 2, pp. 293–312.

[8] Martone, M.—Filippone, S.—Paprzycki, M.—Tucci, S.: On the Usage of 16 Bit Indices in Recursively Stored Sparse Matrices. 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC) 2010, pp. 57–64.

[9] Zhuo, L.—Prasanna, V. K.: Sparse Matrix-Vector Multiplication on FPGAs. In Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays. ACM New York, NY, USA 2005, pp. 63–74.

[10] El-Kurdi, Y.—Gross, W. J.—Giannacopoulos, D.: Sparse Matrix-Vector Multiplication for Finite Element Method Matrices on FPGAs. 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines 2006, pp. 293–294.

[11] Sun, J.—Peterson, G.—Storaasli, O.: Sparse Matrix-Vector Multiplication Design on FPGAs. 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines 2007, pp. 349–352.

[12] Lin, C. Y.—Zhang, Z.—Wong, N.: Design Space Exploration for Sparse Matrix-Matrix Multiplication on FPGAs, 2010 International Conference on Field-Programmable Technology (FPT) 2010, pp. 8–10.

[13] Russek, P.—Wiatr, K.: Dedicated Architecture for Double Precision Matrix Multiplication in Supercomputing Environment: Proceedings of the 2007 IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, pp. 321–324.

[14] Yong, D.—Vassiliadis, S.—Kuzmanov, G. K.—Gaydadjiev, G. N.: 64-Bit Floating-Point FPGA Matrix Multiplication. Proceedings 13th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays 2005.

[15] Bell, N.—Garland, M.: Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York 2009.

[16] Xilinx Inc. LogiCORE IP Floating-Point Operator v5.0, Product Specification, DS335, March 1, 2011, `www.xilinx.com`.

[17] Omondi, A. R.: Computer Arithmetic Systems: Algorithms, Architecture and Implementation, Prentice-Hall International 1994.

[18] Strollo, A. G. M.—Petra, N.—DeCaro, D.: Dual-Tree Error Compensation for High Performance Fixed-Width Multipliers. IEEE Trans. on Circuits and Systems-II: Analog and Digital Signal Processing, Vol. 52, 2005, No. 8, pp. 501–507.

[19] Garofalo, V.—Petra, N.—De Caro, D.—Strollo, A. G. M.—Napoli, E.: Low Error Truncated Multipliers for DSP Applications. 15th IEEE International Conference on Electronics, Circuits and Systems 2008, pp. 29–32.

[20] Jamro, E.—Wielgosz, M.—Wiatr, K.: Novel Reduced-Width Multiplier Structure Dedicated for FPGAs. Electrical Review, Vol. 85, 2009, No. 8, pp. 66–69.

[21] Poldre, J.—Tammemae, K.: Reconfigurable Multiplier for Virtex FPGA Family. Int. Workshop on Field-Programmable Logic and Applications, Glasgow 1999, pp. 359–364.

Ernest Jamro received his M. Sc. degree in electronic engineering from the AGH University of Science and Technology (AGH UST), Kraków (Poland) in 1996; his M. Phil. degree from the University of Huddersfield (U.K.) in 1997; his Ph. D. degree from the AGH UST in 2001. He is currently an Assistant Professor in the Department of Electronics, AGH UST. His research interests include reconfigurable hardware (especially Field Programmable Gate Arrays – FPGAs), reconfigurable computing systems, System on Chip design, artificial intelligence.

Tomasz Pabiś received his B. Sc. degree from AGH University of Science and Technology in Kraków (Poland) in 2012. During his studies he worked in the Reconfigurable Computing Systems group. His interest concerns parallel architectures and computing. Recently he joined Cambridge based firm ARM Ltd., where he develops graphics processors.

Pawel Russek received his M. Sc. degree in electronics from AGH University of Science and Technology in 1995 and his Ph. D. degree from the same university in 2003, with research on customized architecture for image compression algorithm in FPGA. He works as an Assistant Professor in the Department of Electronics, AGH UST and as a research fellow in the Academic Computing Center “Cyfronet” AGH. His research interests focus on hardware acceleration in FPGA. He is particularly interested in high performance reconfigurable computing, embedded systems and the digital systems design based on high level synthesis languages. He is the author and a co-author of many publications in that field.

Kazimierz Wiatr received his M. Sc. and Ph. D. degrees in electrical engineering from the AGH University of Science and Technology, Kraków, Poland, in 1980 and 1987, respectively, and his D. Hab. degree in electronics from the University of Technology of Łódź in 1999. He received his Professor degree in 2001. His research interests include design and performance of dedicated hardware structures and reconfigurable processors employing FPGAs for acceleration computing. He received 11 research grants from Polish Committee of Science Research. These works resulted in above 200 publications, including 7 books, the recent one being Acceleration Computing in Video Processing Systems. He is also the author of 5 patents and 35 industrial implementations. He was the reviewer of: IEEE Expert Magazine: Intelligent Systems, IEE Computer and Digital Techniques, IEE Electronic Letters, International Journal Eng. App. of Artificial Intelligence, IEEE Transactions on Neural Networks, Journal of Machine Graphics and Vision, Eurasip Journal on Applied Signal Processing. He is currently the Director of Academic Computing Centre CYFORNET AGH, and the Head of PIONIER council – Polish Optical Internet. Last but not least, he is a member of the Polish parliament (Senate), and the Head of the Senate Science and Education Committee.