http://dx.doi.org/10.1049/iet-cdt.2011.0132

Ž. Jovanović, V. Milutinović:

FPGA Accelerator for Floating-Point Matrix Multiplication

Abstract:

This study treats architecture and implementation of a FPGA accelerator for double-precision floating-point matrix multiplication. The architecture is oriented towards minimising resource utilisation and maximising clock frequency. It employs the block matrix multiplication algorithm which returns the result blocks to the host processor as soon as they are computed. This avoids output buffering, and simplifies placement and routing on the chip. The authors show that such architecture is especially well suited for full-duplex communication links between the accelerator and the host processor. The architecture requires the result blocks to be accumulated by the host processor; however, the authors show that typically more than 99% of all arithmetic operations are performed by the accelerator. The implementation focuses on efficient use of embedded FPGA resources, in order to allow for a large number of processing elements (PEs). Each PE uses 8 Virtex-6 DSP blocks. Both adders and multipliers are deeply pipelined and use several FPGA-specific techniques to achieve small area size and high clock frequency. Finally, the authors quantify the performance of accelerator implemented in Xilinx Virtex-6 FPGA, with 252 PEs running at 403 MHz (achieving 203.1 GFLOPS), by comparing it to DGEMM function from MKL, ACML, GotoBLAS and ATLAS libraries executing on Intel Core2Quad and AMD Phenom X4 microprocessors running at 2.8 GHz. The accelerator performs 4.5 times faster than the fastest processor/library pair.

1. Introduction

Today's FPGAs are fast and large enough to allow hardware implementation of various algorithms that work faster compared to their software-only counterparts executing on general-purpose microprocessors [1], [2], [3], [4], [5]. There is a plethora of research efforts regarding the use of FPGA accelerators to speed up critical parts of computationally-intensive programs. They vary in scope and way in which acceleration is accomplished; however, they all rely on some kind of parallelism, and their performance is determined by the number of concurrently working functional units.

Due to its significance in science and engineering, matrix multiplication methods and their optimisations are a very often studied subject in the field of both software and hardware design. Its inherent parallelism is especially interesting from the aspect of various parallel and distributed systems. FPGA-accelerated matrix multiplication became a viable faster alternative to software implementations from the moment when FPGA started to offer a potentially better multiplication performance than microprocessors, that is, when they started to include a dedicated multiplier blocks [6].

1.1. Related work

There are several recent works which treat the problem of performing double-precision floating-point matrix multiplication in FPGA. Architecture by Dou et al. [7] consists of a master processor and a linear array of P processing elements. The master processor divides the input matrices into tiles with dimensions Si x N and N x Sj, respectively, and schedules them to the processing array, which calculates one Si x Sj block of the result before moving to the next one. Each processing element (PE) contains one multiplier and one adder, two register banks with Si/P words of storage for storing elements of the first matrix, and two register banks with Si×Sj/P words, for storing intermediate results. Elements of the second matrix are not reused, and therefore not stored. The total used local storage is $M = 2\times$Si+$2\times S$i×Sj words, and the required input bandwidth at maximum performance is $B = 2\times P/\sqrt{M/2}$ words per clock cycle.

Zhuo and Prasanna [8] give the comprehensive overview of previous works on integer and single-precision matrix multiplication in FPGA, and identify the challenges associated with their possible expansion to double-precision arithmetic. They then introduce two architectures, and three corresponding algorithms. The first two algorithms work with small matrices (those which can fit into accelerator internal memory), which allow them to achieve the optimal latency depending on the available communication bandwidth. Both algorithms divide input matrices into rectangular

blocks, but differ in number of multipliers and adders per processing element, and number of required processing elements for a given size of input matrices. The third algorithm is suitable for larger matrices, and uses the block matrix multiplication algorithm with square blocks. It employs a linear list of processing elements, each with one multiplier and one adder, similarly to architecture by Dou et al. [7]. However, it swaps the execution order of the two inner loops, out of three loops which constitute the matrix multiplication algorithm. In that way, there is no need for two sets of registers, and for a given local memory size M and number of processing elements P, the algorithm requires input bandwidth $B = 2 \times P / \sqrt{M}$.

Architecture by Kumar and al. [9] uses an algorithm for scheduling input data to processing elements which has the same loop execution order as that of Zhuo and Prasanna [8]. However, instead of a systolic array-like structure (in which every PE communicates only with the adjacent ones), it uses broadcast to distribute the same elements of the first matrix simultaneously to all PEs. The elements of second and resultant matrices are exchanged only with the adjacent PEs, as is the case with the other two related works.

All of the three architectures are equivalent to each other, and have the same performance of $2 \times P$ FLOPS per clock cycle. They use the classical block matrix multiplication algorithm, and can multiply two square matrices of order N in N^3/P clock cycles. They have the form of a linear list of processing elements, which allows them to be scalable, that is, easily expandable to larger devices or multiple FPGAs. The architectures are also modular, because they treat floating-point arithmetic blocks as modules, which can be interchanged with modules having different implementation or characteristics, without affecting the properties of the architecture. Finally, all of the architectures have balanced processing power and bandwidth requirements. This represents the optimal use of available resources, as the computing and communication phases completely overlap. The respective papers also discuss trade-offs between local memory size and required communication bandwidth. However, they do not distinguish between input and output bandwidth, and take into account only their aggregate value. This is appropriate when communication channel is bus-based, and therefore half-duplex. However, as the full matrix multiplication has highly asymmetric traffic patterns in inbound and outbound directions, and, as the most backplane and network technologies transition to point-to-point, full-duplex architectures (the notable example being PCI-Express), that leave the communication channel in outbound direction almost unutilised.

Dou et at. [7] implement double-precision floating-point units which are IEEE-754 [12] compliant, with the exception of denormal number support. They use FPGAs with 18x18 integer multiplier blocks, and construct a floating-point multiplier from 9 such blocks. Work of Kumar et al. [9] is more recent, and use FPGAs with 25x18 multipliers. In spite of that, their floating-point

multiplier design require 13 such blocks. The level of IEEE-754 compliance is the same as that of Dou et at. [7]. Zhuo and Prasanna [8] describe three floating-point multiplier and adder implementations with different level of IEEE standard compliance: the "fully-compliant", "moderately-compliant" (similar to those in [7] and [9]) and "least-compliant" (which, in addition to the absence of denormal support, also lack all the rounding modes except "round toward zero" and does not generate exceptions). They specify the number of pipeline stages, area, and clock frequency for adders and multipliers using all three level of compliance. However, they do not include implementation details and number of used multiplier blocks.

The reported single-FPGA maximum performance figures are: 15.6 GFLOPS with 39 PEs at 200 MHz [7], 6.7 GFLOPS with 20 PEs at 170 MHz [8], and 29.8 GFLOPS with 40 PEs at 373 MHz [9] ("algorithm 3", with unspecified level of IEEE standard compliance) for large square matrices. Although it is not possible to directly compare the results, because they were published over a time span of several years and use different FPGA devices, the measured results confirm the predicted performance of $2 \times P$ FLOPS per clock cycle for all of them. Zhuo and Prasanna [9] report the performance of their FPGA accelerator as "comparable with that of the state-of-the-art general-purpose processors" executing matrix-multiplication function from Intel MKL library. However, it is not clear how the results reported in the other two papers compare to the performance of general-purpose microprocessors from the corresponding time periods.

1.2. Essence of the proposed approach

Existing solutions to FPGA-accelerated dense matrix multiplication problem have very similar architectures, because they all depend on the classic block matrix multiplications algorithm. Faster algorithms do exist [10], [11], however, they are much more complex, and generally not suitable for hardware implementation. Since there is not much room for improvements in the architectural domain, it is possible to achieve better performance primarily by implementing larger number of smaller and faster floating-point units.

This paper presents an architecture and implementation of a FPGA accelerator for multiplication of matrices in IEEE double-precision format, which aims to be as fast as possible by using the following techniques, not found in other related works:

- A block matrix multiplication architecture which returns the result blocks as soon as they are computed, and leaves their final accumulation to the host processor. This allows for a less constrained placement and routing on FPGA, and consequently higher clock frequency. Such architecture also exhibits a much more symmetric communication pattern (similar inbound and outbound bandwidth requirements), which make it especially well suited for

full-duplex communication links. We show that the additional load exhibited on the host processor is relatively small, as the accelerator computes all multiplications and almost all (n-1 out of n) additions.

- Implementation of floating-point units in an efficient way, with only 8 embedded FPGA 25x18 integer multiplier blocks per floating-point multiplier. This allows realisation of larger number of processing elements. The floating-point units are also deeply pipelined, which contributes to the high clock frequency, but also in some instances reduces area, because it allows better utilisation of embedded blocks. We show that the latencies associated with the deeper pipelines do not have any negative implications, as they are in any case much smaller than the other latencies present in the system.

After presenting the accelerator architecture and implementation, we quantify the performance gain of doing FPGA-accelerated matrix multiplication, in comparison to software implementations from several high-performance libraries, executing on commodity microprocessors. We take into account the current state of the art in both microprocessor and FPGA technology. Modern microprocessors perform very well, because they have high clock frequencies, multiple cores, and appropriate support in the form of floating-point vector SIMD units. However, we show that FPGA-accelerated matrix multiplication can still achieve several times higher performance.

2. Accelerator architecture

The accelerator consists of a linear list of multiplier-adder processing elements (PE), with memories for buffering input (and possibly output) data spread equally across all PEs (Fig. 1). Although the PE connection pattern in the form of a tree is also possible [13], the linear list has the advantage of a much more regular structure, which allows simpler routing between PEs and consequently the higher clock frequency. After the initial latency, a list of n PEs multiply two n-element vectors in one clock cycle, or two square matrices of order n in n^2 clock cycles. The initial latency consists of time necessary to fill the input buffers and time necessary to fill the pipelines of multipliers and adders. All the subsequent loading of input data overlap with computation on previously loaded data, and all the pipelines of arithmetic units are constantly full. The linear list architecture can also extend to multiple FPGAs. In that case, the number of parallel arithmetic operations and the required communication bandwidth increase linearly with the number of FPGA nodes.

Let $\mathbf{X}$ and $\mathbf{Y}$ be matrices with dimensions $p \times q$ and $q \times r$, respectively, where $p,q,r \geq n$. Matrix $\mathbf{X}$ consists of $i \times j$ and matrix $\mathbf{Y}$ consists of $j \times k$ blocks of order n, where $i = \lceil p/n \rceil$, $j = \lceil q/n \rceil$ and $k =$

$\lceil r/n \rceil$ (we pad the right and bottom of the matrices with zeros as necessary, in order to have integer number of blocks). It is possible to multiply the matrices **X** and **Y** by performing matrix multiplication and addition operation only between the blocks. The result matrix **R**, with dimensions $p \times r$, consists of $i \times k$ blocks. We now consider two multiplication algorithms, one suitable for accelerator architecture shown in Fig. 1a and the other for accelerator architecture shown in Fig. 1b. We refer to the blocks of matrices **X**, **Y** and **R** as $\mathbf{X}_{uv}$, $\mathbf{Y}_{vw}$, $\mathbf{R}_{uw}$, respectively, where $u \in \{1,..., i\}$, $v \in \{1,..., j\}$ and $w \in \{1,..., k\}$.

Algorithm 1: The host computer consecutively sends to the accelerator the blocks of input matrices which correspond to the one complete result block, before starting to send the input data for calculation of the next result block. It starts with the input blocks used to compute the result block $\mathbf{R}_{11}$: $\mathbf{X}_{11}$ and $\mathbf{Y}_{11}$, $\mathbf{X}_{12}$ and $\mathbf{Y}_{21}$, ..., $\mathbf{X}_{1j}$ and $\mathbf{Y}_{j1}$ and so on, and in total sends $b_{\text{in}}^{(1)} = 2 \times i \times j \times k$ input blocks. For the same time period, the accelerator sends back $b_{\text{out}}^{(1)} = i \times k$ result blocks. The ratio of input to output traffic is $s^{(1)} = b_{\text{in}}^{(1)} / b_{\text{out}}^{(1)} = 2 \times j$. This algorithm does not require that the host computer takes part in computation. However, its drawback is the relatively large bandwidth requirement in host – accelerator direction and highly asymmetric traffic pattern.

Algorithm 2: The host computer starts with sending block $\mathbf{X}_{11}$ and than sends all the blocks which should be multiplied with it: $\mathbf{Y}_{11}$, $\mathbf{Y}_{12}$, ..., $\mathbf{Y}_{1k}$. The multiplication continues with block $\mathbf{X}_{21}$: $\mathbf{Y}_{1k}$ is reused, and the row of **Y** enters the accelerator in the opposite direction: $\mathbf{Y}_{1k\text{-}1}$, ..., $\mathbf{Y}_{11}$. Procedure repeats in the same way for all j rows of **X** and corresponding j columns of **Y**. The results of multiplying consecutive input blocks does not represent parts of the same result block. For that reason, the accelerator can not add them together (and does not have a FIFO buffer R and an additional adder for that purpose), but instead send them immediately to the host computer. The host computer must add the each received partial result block to the previously received part of the same result block. This additions take place simultaneously with the results reception and, as we later show, constitute only a small fraction of all arithmetic operations. In total, the host computer sends $b_{\text{in}}^{(2)} = (1+k+(1+k\text{-}1)\times(i\text{-}1))\times j = (i \times k+1) \times j$ input blocks and for the same time period receives $b_{\text{out}}^{(2)} = i \times j \times k$ output blocks. The ratio of input to output traffic is $s^{(2)} = b_{\text{in}}^{(2)} / b_{\text{out}}^{(2)} = 1+1/(i \times k)$. The worst case, $s^{(2)}=2$, occurs for $i=1$ and $k=1$. However, as i or k increase, $s^{(2)}$ decreases rapidly, and $\lim_{i,k \to \infty} s^{(2)} = 1$. Therefore, for a sufficiently large i or k, this algorithm has almost equal bandwidth requirements in both directions.

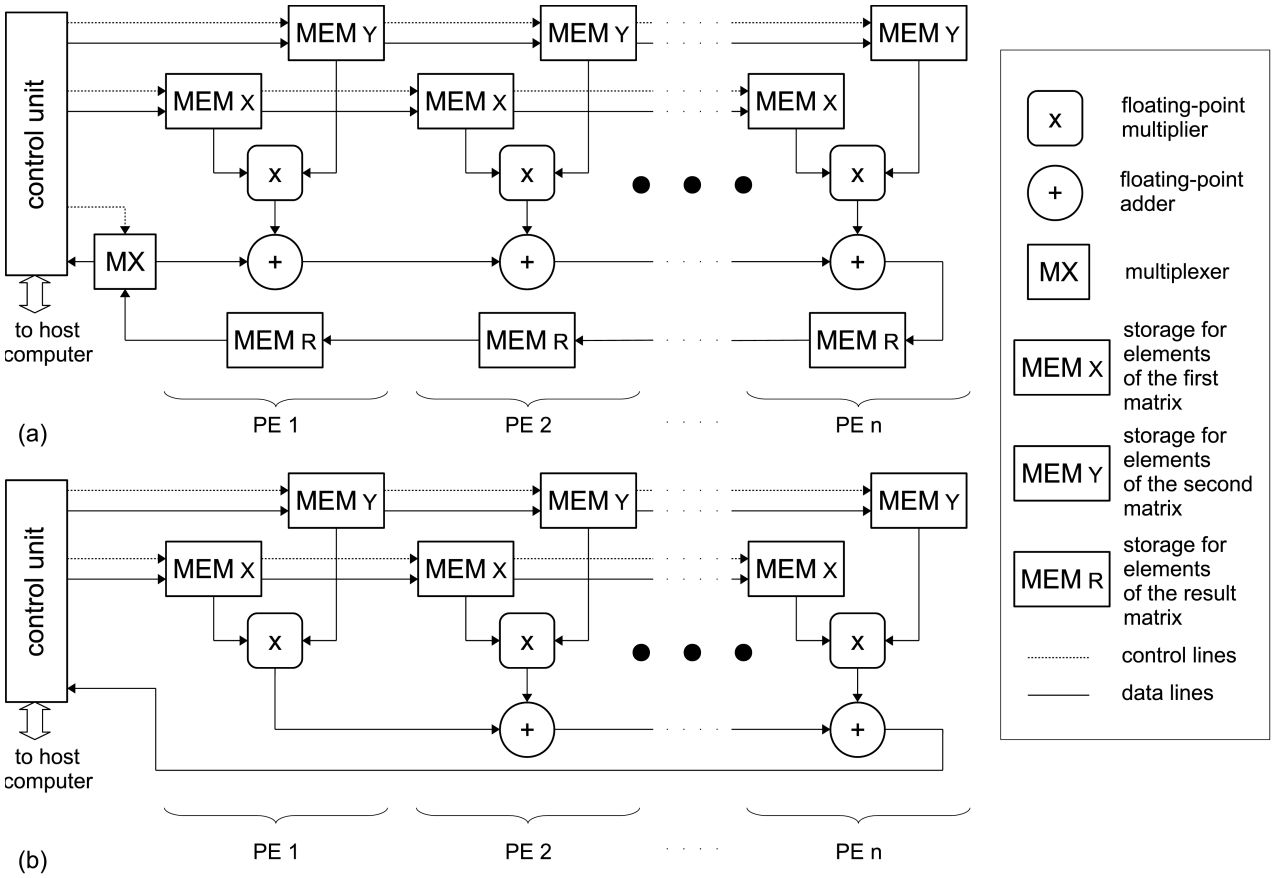

Fig. 1: Two variants of a FPGA accelerator for matrix multiplication, organized as a linear list of n processing elements (PE). Each PE consists of a multiplier, adder and BRAMs X and Y for storing input data. In variant (a), there are also BRAMs R for storing intermediate results, connected as a circular FIFO. Memories X store rows of the first and memories Y columns of the second operand matrix. In each clock cycle, accelerator performs n multiplications, n-1 product accumulations and multiplies two n x n matrices in n^2 clock cycles. The input matrices can be the blocks of larger matrices. The FIFO R in variant (a) facilitates the multiplication of block matrices, by storing a block of the result, until the accelerator multiplies and accumulates all the corresponding input blocks. There is also an additional adder for that purpose. FIFO's total size is n^2 elements, and in each clock cycle elements rotate to the left. When the multiplication of the last block completes, the multiplexer MX allows the FIFO contents to be shifted out and sent to the host computer. Variant (b) immediately returns elements of multiplied matrices without buffering. In the case of block matrix multiplication, it relies on the host computer to add together matrices representing parts of a resultant matrix block. This variant has the advantage of being simpler and easier to place and route, and also uses less BRAM resources.

Let B_{in} and B_{out} be, respectively, necessary input and output communication bandwidth, expressed as number of floating-point words transferred per clock cycle. The total bandwidth is $B_{\text{HD}}=B_{\text{in}}+B_{\text{out}}$ in the case of half-duplex and $B_{\text{FD}}=\max\{B_{\text{in}},B_{\text{out}}\}$ in the case of full-duplex communication link. If the block order is n, and the total communication time is T_{comm} clock cycles, the total number of blocks transferred during time T_{comm} is, for input and output directions, half-duplex and full-duplex links, respectively: b_{in}, b_{out}, b_{HD}, b_{FD}, where $B_{\text{in}} = b_{\text{in}}\times n^2/T_{\text{comm}}$, $B_{\text{out}} = b_{\text{out}}\times n^2/T_{\text{comm}}$, $B_{\text{HD}} = b_{\text{HD}}\times n^2/T_{\text{comm}}$ and $B_{\text{FD}} = b_{\text{FD}}\times n^2/T_{\text{comm}}$. For Algorithm 1, $b_{\text{HD}}^{(1)} = 2\times i\times j\times k + i\times k$ and $b_{\text{FD}}^{(1)} = 2\times i\times j\times k$. For Algorithm 2, $b_{\text{HD}}^{(2)} = (i\times k+1)\times j + i\times j\times k = (2\times i\times k+1)\times j$ and $b_{\text{FD}}^{(2)} = (i\times k+1)\times j$. When utilising half-duplex links, both algorithms have approximately equal bandwidth requirements, $b_{\text{HD}}^{(1)} \cong b_{\text{HD}}^{(2)}$. When utilising a full-duplex link, Algorithm 2 require $b_{\text{FD}}^{(1)}/b_{\text{FD}}^{(2)} = 2\times i\times k\,/(i\times k+1) = 2/s^{(2)}$ times less bandwidth. For k=1, there is no bandwidth reduction. However, for

a sufficiently large k, Algorithm 2 reduces the required bandwidth almost by a factor of two. The rest of this paper discuss Algorithm 2 and the corresponding accelerator architecture shown in Fig. 1b.

Before the matrix multiplication starts, the host computer must send to the accelerator the initial n^2 elements of $\mathbf{X}_{11}$ and n^2 elements of $\mathbf{Y}_{11}$ blocks (these $2\times n^2$ elements are distributed as $2\times n$ elements in each of n PEs). It takes the next n^2 clock cycles to multiply the two blocks. During that time, the accelerator loads n^2 elements of matrix $\mathbf{Y}_{21}$ and multiplication continues according to Algorithm 2. If the new blocks keep coming at the same rate, all the multiplier and adder pipelines work without stopping. The accelerator produces one element of the result block in each clock cycle. The initial latency before producing first result T_i is equal to the time necessary to load initial data and latency through n multipliers and n-1 adders, $T_i = 2\times n^2 + d_m\times n + d_a\times(n\text{-}1)$, where d_m and d_a are, respectively, multiplier and adder latencies. The value of T_i is implementation dependent and we further discuss it in chapter 4.

The accelerator with clock frequency f performs n multiplications and n-1 additions in each clock cycle and therefore the total of $P = (2\times n\text{-}1)\times f$ floating-point operations per second (FLOPS). The one remaining addition is done on the host processor, which, in total, performs $1/n$ of all additions. However, the more useful measure of accelerator performance is the time necessary to multiply two matrices of given size. According to Algorithm 2, the accelerator can multiply matrices $\mathbf{X}$ and $\mathbf{Y}$ in $T_{comp} = i\times j\times k\times n^2$ clock cycles, or $T_{comp}\times f$ seconds. According to the equations for B_{HD} and B_{FD}, $T_{comm} = b_{HD}\times n^2/B_{HD} = (2\times i\times k+1)\times j\times n^2/B_{HD}$ and $T_{comm} = b_{FD}\times n^2/B_{FD} = (i\times k+1)\times j\times n^2/B_{FD}$. Because communication and computation phases overlap, we can calculate the required communication bandwidth B_{HD} and B_{FD} from relation $T_{comm} = T_{comp.}$ $B_{HD} = (2\times i\times k+1)\times j/(i\times j\times k)$ and $B_{FD} = (i\times k+1)\times j/(i\times j\times k)$ words per clock cycle.

In order to compare the proposed architecture to the works which assume square matrices, we now analyse the case when $i=j=k$. The expressions for the required bandwidth become: $B_{HD} = (2\times k^3+k)/k^3$ and $B_{FD} = (k^3\times k)/k^3$. The worst case, B_{HD}=3 and B_{FD}=2, occur for k=1. This is expected, as in that case input data do not represent blocks of larger matrices, and there is no potential for data reuse. However, for k=2, bandwidth requirements are already much lower, at B_{HD}=2.25 and B_{FD}=1.25 words per clock cycle, and $\lim_{k\to\infty}B_{HD} = 2$, $\lim_{k\to\infty}B_{FD} = 1$.

The Algorithm 1 is equivalent to the third algorithm from [8], and algorithms from [7] and [9] when they use minimal local memory size of $\Omega(k^2)$ words. They have the constant bandwidth requirements of B_{HD}=3 and B_{FD}=2 words per clock cycle. All of the related papers further pursue the idea of data reuse by allowing lower input traffic at the expense of larger input buffers. For memory of size M words, and p processing elements, they require $\sqrt{M}/p$ times lower bandwidth

in input direction. This leads to more complex block scheduling algorithms, which use more logic and memory resources. That, in turn, limit the number of PEs which can be implemented on a chip, and also has negative impact on maximum achievable clock frequency. In contrast to this approach, our Algorithm 2 reuse input blocks exactly to the extent needed to perfectly balance traffic in both directions. Thus, it is able to optimally utilise full-duplex links and has a relatively simple architecture. This simplicity, together with an efficient implementation of floating-point units, allows us to place more PEs on a chip than previously possible, and consequently achieve better performance.

3. Implementation of floating-point units

When utilising embedded multipliers and memories, FPGAs use about 20 times more transistors for the same amount of logic than standard-cell-based ASICs [14]. The difference is even larger in comparison to integrated circuits based on full-custom design (such as commodity microprocessors). As a consequence, clock speed of FPGAs is typically an order of magnitude lower than that of microprocessors and the only way to accomplish better performance is by use of relatively high level of parallelism. It is desirable to implement as many functional units as possible and at the same time maximise their clock frequency.

Design techniques for optimal design of arithmetic circuits in FPGA differ significantly from the techniques used in ASIC VLSI design [15]. The reason for this is fixed structure and scarcity of general logic resources in FPGAs. For example, multipliers based on large compressor trees and two-path floating-point adders implemented in FPGAs use too much logic and offer low clock frequency. At the same time, they might not use specialised resources present in modern FPGAs, such as embedded multiplier-adder blocks and fast carry chains. Optimal design of arithmetic circuits in FPGA require careful consideration of space/time trade-offs and the unique features of FPGA technology.

3.1. Design considerations

The accelerator is implemented using Xilinx XC6VSX475T FPGA, because of its large number of embedded multipliers (2016). We report resource utilisation and speed measurements as simulated with this device. However, it should be also pointed out that this FPGA does not exist in the fastest (-3) speed grade, which imposes an additional limit to the maximal achievable clock frequency.

3.2. Addition / subtraction

To perform a two-operand floating-point addition/subtraction, it is necessary to assure that

both operands have the same exponent by right shifting the mantissa of the smaller one and increasing its exponent accordingly, add the mantissas if they have the same sign (effective addition), or subtract the smaller mantissa from the larger otherwise (effective subtraction). Sign of the result is equal to the sign of the larger mantissa. It may then be necessary to normalise the result by shifting it one place to the right (in the case of effective addition) or up to n places to the left (in the case of effective subtraction), where n is the width of the mantissa. Finally, it is necessary to round the result according to the chosen rounding mode. We use this, "classic" algorithm for floating-point addition, because it has the minimal resource requirements. Variants which use dual data paths, leading zero prediction and rounding before final normalisation offer lower latency, at the cost of much greater complexity [16]. They are commonly used in microprocessors and other designs requiring low latency, but they are not practical for FPGA implementation.

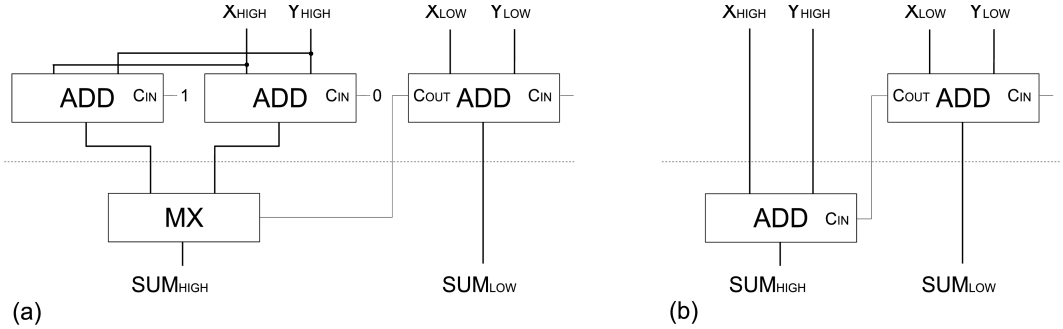

Fig. 2: Using smaller adders to implement a larger one – (a) the carry-select approach, often used in ASICs, assume that multiplexer complexity and propagation delay is much lower than that of an adder. However, in a FPGA, both multiplexer and a ripple-carry adder use one LUT per bit. (b) The simple ripple-carry adder divided into two pipeline stages achieves the same performance, while using approximately two times less resources.

The floating-point adder consists of d_a=10 pipeline stages, A1–A10. In stage A1, we compare mantissas and calculate exponent difference, which determine the number of places smaller mantissa should be shifted to the right. In order to compute correctly rounded results (as defined in [12]), we extend the right side of the operands with three additional bits (guard, round and sticky bit, respectively), which are initially zero. In stage A2, there are three levels of logic, each implementing a funnel shifter. The usual approach is to use six shifters, in order to perform a 32, 16, 8, 4, 2 and 1 bit shifts. However, the six-input LUT architecture of the target FPGA allows combining two shifts in the same LUT, thus reducing the number of required logic levels. We **or** together all the bits which are shifted out, and keep them as the rightmost (sticky) bit. In stage A3, mantissas are swapped if necessary, in order to assure that first mantissa is always the larger one. We perform the addition/subtraction with ripple-carry adder which use one LUT per result bit and a dedicated fast carry-propagation chain. Although the adder is very efficient in terms of used resources, its propagation delay increases linearly with operand size [17] and with 56-bit operands it becomes a clock-limiting factor. Since, for this purpose, low logic count and high clock speed are

more important than low latency, we spread the adder through two pipeline stages, A4 and A5 (Fig. 2b). Stages A6 and A7 contain the leading zero counter. It functions by computing the logic **or** of the adjacent 4-bit groups of the result and then repeating the same procedure on the resultant bits (which correspond to the 16-bit groups in the original result). The calculated values represent the contiguous groups of 4 and 16 zeros in the result. By priority encoding them, it is possible to determine the number of leading zeros. Stage A8 contains the normalisation left shift. The rounding addition takes place in the stages A9 and A10 (with adder split according to Fig. 2b).

The adder supports operations with subnormal numbers. The resource utilisation is 871 LUTs and 1022 flip-flops and achieved clock frequency is 509.94 MHz.

3.3. Multiplication

To perform a floating-point multiplication of two operands, it is necessary to multiply their mantissas, add exponents, and calculate the result sign as **xor** of operand signs. The most complex part is the multiplication of the mantissas. In the case of IEEE double precision format, mantissas are 53 bits wide, and their efficient (both in area and speed) multiplication on FPGA require the use of embedded multiplier-adder blocks. In Xilinx Virtex-5 and Virtex-6 devices, these blocks are called DSP48E and DSP48E1, respectively, and contain a 25×18 bit signed multiplier (24×17 bit unsigned), followed by a 48-bit adder with 17-bit shifter before its other input.

There are several ways to use DSP48E/E1 blocks as "tiles," parts of a larger multiplication parallelogram [18]. Each block compute one partial product, and add it together with a part of previously computed result. We propose the design shown in Fig. 3, to maximise the use of internal 48-bit adders for accumulating partial products. Variant (a) uses fewer DSP48E/E1 blocks, but more LUTs. Because of the overall availability of logic resources in the target device, we use variant (b).

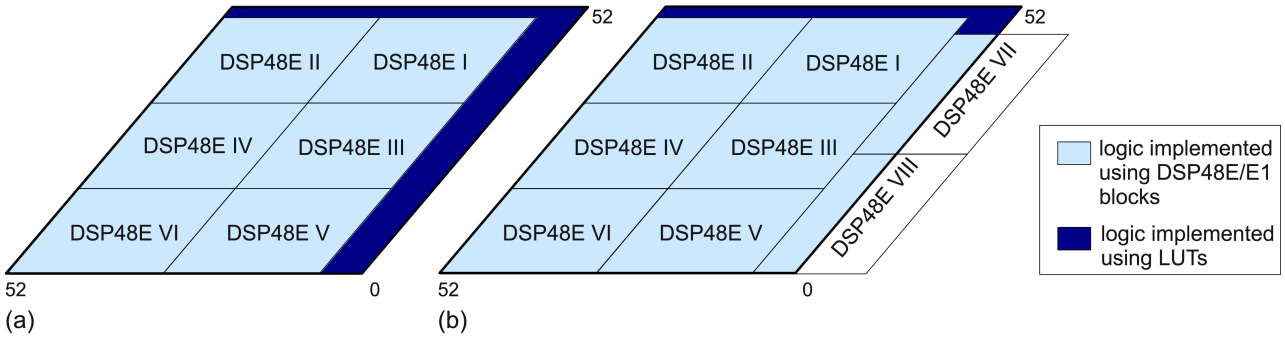

Fig. 3: Implementing 53×53 bit unsigned integer multiplier using Virtex-5 DSP48E or Virtex-6 DSP48E1 blocks. Multiplier (a) uses 6 DSP48E/E1 blocks and more LUTs, while multiplier (b) uses 8 DSP48E/E1 blocks and less LUTs. Variant with 7 DSP48E/E1 blocks is also possible.

The floating-point multiplier consists of d_m=17 pipeline stages, M1–M17. The relatively large number of stages is necessary because DSP48E/E1 blocks require two clock cycles in order to execute multiplication and addition at full speed. For that purpose, they have internal pipeline

registers after both multiplier and adder. We use blocks VII and VIII as 5×24 bit multipliers, and the other six blocks (I, II, III, IV, V and VI) as 24×17 bit multipliers. In stage M1, we calculate the exponent and sign of the result, and also the partial product corresponding to block VII and the partial products which do not use DSP blocks (the dark-coloured areas in Fig. 3b). Multiplications corresponding to blocks I and VIII take place in stage M3 and those corresponding to blocks II, III, IV, V and VI in stages M5, M7, M9, M11 and M13, respectively. The adder in each block accumulate the parts of the result which are located above and to the right from it (as in Fig. 3b) and calculated in previous stages. Those additions take place in stages M2, M4, M6, M8, M10, M12 and M14. Because the rightmost 52 bits of the complete 106-bit product are necessary only to calculate the sticky bit, we can **or** them together as soon as they are computed. In stage M15, we normalise the result mantissa and compute the rounding digit. If we assume that input numbers are normal (MSB=1), the multiplication result can be either normal, or may require a single normalisation left shift. The rounding takes place in the stages M16 and M17 (using a two-stage adder as in Fig. 2b).

The resource utilisation is 447 LUTs and 520 flip-flops and achieved clock frequency is 492.12 MHz. The described multiplier work only with normal operands. It is relatively easy to add support for subnormal numbers by placing leading zero counters and left shifters on multiplier inputs (to pre-normalise the mantissas), while increasing the exponents width from 11 to 17 bits (since $\lceil \log_2(53) \rceil = 6$). It is also necessary to add a right shifter in the last stage, to accommodate for a possible result mantissa conversion to subnormal format. The cost of this additional logic (if we implement it as described in chapter 3.2) is 727 LUTs and 635 flip-flops. We do not implement the subnormal number support, because of the limited space in the target device. Instead, we treat all subnormal input values as zeros

4. Numerical results and discussion

Considering the size of the target device, there are n=252 processing elements in the accelerator. Each PE uses 8 DSP48E1 blocks and four 36 kb BRAMs. The two of them implement buffers X and the other two buffers Y. Each X or Y buffer in each PE can store $4n$=1008 unpacked double-precision floating-point numbers. Although the capacity of $2n$ entries would be sufficient for double-buffering (the accelerator works with one input block while it receives another), we use $4n$ entries to fully utilise the BRAM resources available in the target device. The larger input buffers allow for an easier amortisation of communication speed variations, thus reducing the chance for buffer underflow condition, which would inevitably result in pipeline stall.

The total resource utilisation (excluding PCI-Express related logic) is 290556 LUTs, 433692 flip-flops, 2016 DSP48E1 blocks and 1008 RAMB36E1 blocks. With automatic placement and

routing and “balanced design goal” options, Xilinx ISE toolchain achieves clock frequency of 161 MHz. To obtain better results, we manually floor-plan and partition the whole design, in order to group together the resources used by individual PEs and precisely position the PEs relative to each other. Although the PEs do not form a circular array (ring), we “fold” their linear structure in half, so that PE1 and PE*n* are physically adjacent and located next to the control unit. The achieved clock frequency of the accelerator is f = 403.87 MHz. This corresponds to the performance of p = (2×252-1)×0.40387 = 203.1 GFLOPS. It should be noted that we obtained this results using simulation of the target FPGA device. Due to the factors outside the FPGA, such as communication or software latencies, the performance on the actual hardware could be lower. However, similar works which do include a real-world performance figures (such as [8]), show that this slowdown is typically not significant.

If we consider equation for initial latency T_i, and take into account that n=252, d_m=17, d_a=10, we can see that the first term $2\times n^2$=127008 is much larger then the sum of the other two: $d_m\times n+d_a\times(n-1)$ = 6794. This means that it is possible to use deeply pipelined adders and multipliers, virtually without affecting the total latency.

The target device has an integrated PCI-Express 2.0 8x endpoint which can be used for communication with the host computer. Its bandwidth of 4 GB/s in each direction, with accelerator clock frequency of 403 MHz and 64-bit floating-point words, is equal to B_{FD} = 4096/403/8 = 1.27 word per clock cycle. Considering square matrices of order $k\times n$ ($k\in\mathbb{N}$) and equation for B_{FD}, k must be at least 2 in order for this communication link to not limit the accelerator performance. The simulated execution times are given in Table 1.

In order to compare the accelerator performance to that of microprocessors, we measure the time necessary to execute a double precision matrix multiplication (DGEMM) function from four highly optimised BLAS libraries, with randomly generated square matrices, on two different computers with comparable microprocessors. Both microprocessors are manufactured in 45 nm process and belong to the same technological generation as the 40 nm Virtex-6 FPGA. We observe the similar performance in all processor/library combinations (Table 1). Since this measurements are used only in order to establish a baseline software implementation, we do not analyse the small differences which exist between them.

Table 1: Performance of the proposed accelerator (403 MHz, 252 processing elements) in comparison to software implementations on general-purpose processors. Processor P1 is Intel Core2Quad Q9550 (2.84 GHz, 12 MB cache). Processor P2 is AMD Phenom II X4 925 (2.81 GHz, 8 MB cache). Libraries are, respectively: L1-Intel MKL, L2-GotoBLAS, L3-AMD ACML, L4-ATLAS.

matr.	time for multiplication of two square matrices, seconds		
	our	single-thread library version	four-thread library version

order	FPGA accel.	processor P1				processor P2				processor P1				processor P2			
		L1	L2	L3	L4	L1	L2	L3	L4	L1	L2	L3	L4	L1	L2	L3	L4
2500	0.2	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1	1
5000	1.3	23	23	24	24	25	24	24	25	6	6	6	6	7	6	6	7
7500	4.2	77	78	78	82	86	80	83	85	20	20	20	21	22	20	20	22
10000	10.1	181	184	186	195	200	190	195	199	46	46	47	50	65	53	49	64
12500	19.7	355	358	370	380	395	371	379	390	89	91	92	96	103	111	96	101

It is interesting to notice that, when multiplying two square matrices of order N, all of the four libraries exhibit time complexity of order $O(N^3)$. This indicate that they use the classical block matrix multiplication, and not some of the faster algorithms (such as Strassen's [10], with complexity of $O(N^{2.807})$). We believe this choice is due to the worse numerical stability of such algorithms.

In comparison with software implementations using only one processor core (single-thread library version), the FPGA accelerator is 20 times faster than the slowest and 18 times faster than the fastest processor/library combination. In comparison with software implementations using all four processor cores (multiple-thread library version), the FPGA accelerator is 5.6 times faster than the slowest and 4.5 times faster than the fastest processor/library combination.

The achieved results are consistent with the previously published predictions, according to which FPGAs will continue to offer better floating-point performance than microprocessors [6]. However, we believe that with the current generation of FPGAs and microprocessors, this gap has reached its maximum, and that it will begin to shrink in the future. We base that prediction on the following observations: the recently announced 28 nm Xilinx Virtex-7 FPGA family offer 1.78 times more DSP48E1 blocks compared to Virtex-6, and also some marginal increase in clock frequency (we do not consider other FPGA manufacturers, such as Altera and Lattice, as they do not offer devices of such size). At the same time, 32 nm Intel Sandy Bridge processor offer the vector instruction set (AVX) two times wider than the previous SSE, and also better performance per core compared to the previous generation of processors [19]. There is currently up to 8 cores per chip (in Intel processors), and this number will probably significantly increase in the future [20]. This is not to say that FPGA accelerators will not still be able to achieve better performance in the areas for which microprocessors do not have direct support, but it will become increasingly difficult for FPGAs to outperform them in floating-point arithmetic.

5. Conclusion

This paper has proposed an architecture and corresponding implementation for a FPGA-accelerated floating-point matrix multiplication. To our knowledge, it is the first such work to demonstrate not only comparable, but several times faster performance than that of commodity microprocessors from the same technological generation.

The architecture is as simple as possible, in order to minimise resource utilisation and maximise clock frequency. The employed block matrix multiplication algorithm sends the result blocks to the host processor as soon as they are computed. The consequence of this approach is that, while all multiplication and almost all additions are done in FPGA, $1/n$ of all additions, where n is block order, must be performed on the host processor. However, this number is very small: in our implementation, n=252, and $1/n$=0.4 %. Because the output blocks are not buffered in the accelerator, it generates traffic of similar intensity in both inbound and outbound directions. This makes the architecture well-suited for full-duplex communication links. The drawback is somewhat limited flexibility, as the block order is equal to the number of processing elements, and the used internal memory size is implicitly tied to the communication link speed. As in related works, the performance depends on communication link speed. However, the proposed architecture has the advantage of requiring less bandwidth as the size of input matrices increase. The proposed communication link is 4 GB/s PCI-Express. With the available bandwidth, and considering square matrices, the full speed is achieved for input matrices of order $2\times n$ and larger.

The implementation is also performance-oriented, and focuses on efficient use of embedded FPGA resources. Each PE uses 8 DSP blocks, which allows for a total of 252 PEs. In comparison, a related work based on the same size DSPs [8] require13 DSP blocks per PE. We have also proposed a multiplier design with 6 DSP blocks (which would equal to 336 PEs), however it would require the target device with more general-logic resources. Both adders and multipliers are deeply pipelined and use several FPGA-specific techniques to achieve small area size and high clock frequency. They can be readily reused in any other project related to FPGA floating-point arithmetic.

We have compared the accelerator performance with that of high-end general-purpose microprocessors. In order for comparison to be as objective as possible, we have performed measurements using processors with large amounts of cache memory and high clock frequency, executing matrix multiplication functions from highly optimised libraries. The FPGA accelerator achieved results 18 times better than the fastest single-core, and 4.5 better than the fastest four-core software implementation.

6. References

[1] Todman, T. J., Constantinides, G. A., Wilton, S. J. E., Mencer, O., Luk, W., and Cheung, P. Y. K.: 'Reconfigurable computing: architectures and design methods', IEE Proc. Comput. Digit. Tech., 2005, 152, (2), pp.193–207

[2] Buell, D., El-Ghazawi, T., Gaj, K., Kindratenko, V.: 'High-performance reconfigurable computing', IEEE Computer, 2007, 40, (3), pp. 23-27

[3] Herbordt, M. C., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., and DiSabello D.: 'Achieving high performance with FPGA-based computing', IEEE Computer, 2007, 40, (3), pp. 50-57

[4] El-Ghazawi, T., El-Araby, E., Huang , M., Gaj, K., Kindratenko, V., Buell, D.: 'The promise of high-performance reconfigurable computing ', IEEE Computer, 2008, 41, (2), pp. 69-76

[5] VanCourt, T., Herbordt, M. C.: 'Elements of high-performance reconfigurable computing', Adv. Comput., 2009, 75, pp. 113–157

[6] Underwood, K.: 'FPGAs vs. CPUs: Trends in peak floating-point performance'. Proc. ACM/SIGDA 12th Int. Symp. Field-Programmable Gate Arrays (FPGA), Monterey, CA USA, February 2004, pp. 171-180

[7] Dou, Y., Vassiliadis, S., Kuzmanov, G. K., and Gaydadjiev, G. N.: '64-bit floating-point FPGA matrix multiplication'. Proc. ACM/SIGDA 13th Int. Symp. Field-Programmable Gate Arrays (FPGA), Monterey, CA USA, February 2005, pp. 86-95

[8] Zhuo, L., and Prasanna, V. K.: 'Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems', IEEE Trans. Parallel Distrib. Syst., 2007, 18, (4), pp. 433-448

[9] Kumar, V. B. Y., Joshi, S., Patkar, S. B., and Narayanan, H.: 'FPGA based high performance double-precision matrix multiplication', Int. J. Parallel Programming, 2010, 38, (3-4), pp. 322-338

[10] Strassen, V.: 'Gaussian elimination is not optimal', Numer. Math., 1969, 13, (4), pp. 354-356

[11] Coppersmith, D., Winograd, S.: 'Matrix multiplication via arithmetic progressions', J. Symb. Comput., 1990, 9, (3), pp. 251–280

[12] IEEE Standard 754-2008: 'Standard for Floating-Point Arithmetic', 2008

[13] Zhuo, L., and Prasanna, V. K.: 'High-performance designs for linear algebra operations on reconfigurable hardware', IEEE Trans. Comput., 2008, 57, (8), pp. 1057-1071

[14] Kuon, I., and Rose, J.: 'Measuring the gap between FPGAs and ASICs', IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., 2007, 26, (2), pp. 203-215

[15] Hemmert, K. S., and Underwood, K. D.: 'Fast, efficient floating-point adders and multipliers for FPGAs', ACM Trans. Reconfigurable Technology and Syst., 2010, 3, (3), pp. 11:1–11:30

[16] Seidel, P., and Even, G.: 'Delay-optimized implementation of IEEE floating-point addition', IEEE Trans. Comput., 2004, 53, (2), pp. 97-113

[17] Xilinx Virtex-6 FPGA Configurable Logic Block (UG364), ver. 1.1, http://www.xilinx.com/support/documentation/user_guides/ug364.pdf, accessed September 2009

[18] Banescu S., de Dinechin, F., Pasca, B., and Tudoran R.: 'Multipliers for floating-point double precision and beyond on FPGAs', ACM SIGARCH Comput. Archit. News, 2010, 38, (4), pp. 73-79

[19] Yuffe, M., Knoll, E., Mehalel, M., Shor, J., Kurts, T. : 'A fully integrated multi-CPU, GPU and memory controller 32nm processor '. Dig. Tech. Papers 2011 IEEE Int. Solid-State Circuits Conf. (ISSCC), San Francisco, CA USA, February 2011, pp. 264-266

[20] Patt, Y. N.: 'Future microprocessors: What must we do differently if we are to effectively utilize multi-core and many-core chips', Trans. Internet Research, 2009, 5, (1), pp. 5-9