Volume : 4 | Issue : 12 | December 2015 • ISSN No 2277 - 8179

Research Paper

Matrix Multiplier Design and Simulation for Floating Point, Negative Value and Real Numbers

Engineering

KEYWORDS : VHDL(very high speed integrated circuit hardware descriptive language), Verilog HDL(verilog hardware descriptive language), FPGA(Field programmable gate array)

Ms. Swati C. Hadke Researcher, M.Tech VLSI, GHRAET, RTMN University, Nagpur, India

Prof. Sanjay Tembhurne Electronics & Communication Engineering, GHRAET, RTMN University, Nagpur, India

ABSTRACT

Matrix multiplication is the kernel operation used in many image and signal processing applications. In this paper, we present the design and Field Programmable Gate Array (FPGA) implementation of matrix multiplier architectures for use in image and signal processing applications. The designs are optimized for speed which is the main requirement in these applications. First design involves computation of dense matrix vector multiplication which is used in image processing application. The design has been implemented on Virtex-4 FPGA and the performance is evaluated by computing the execution time on FPGA. Implementation results demonstrate that it can provide a throughput of 16970 frames per second which is quite adequate for most image processing applications.

Our proposed scheme improves the energy efficiency of the baseline architecture by double precision floating point matrix multiplication. Matrix multiplication computation is one of the important special purpose arithmetic operations for solving large numerical problems. It is very inefficient if the computation is done by software on the core central processing unit. The hardware implementation of such a processor is desirable but inevitably faces the limitation of the amount of VLSI area available. An excessive amount of VLSI area usage for such a processor would impact both cost and performance.

Introduction

State-of-the-art FPGAs offer high operating frequency, unprecedented logic density and a host of other features. As FPGAs are programmed specifically for the problem to be solved, they can achieve higher performance with lower power consumption than general-purpose processors. Therefore, FPGA is a promising implementation technology for computationally intensive applications such as signal, image, and network processing tasks. Matrix multiplication is one of the key kernels in computing. Several architectures and algorithms [24], [4] for matrix multiplication on FPGA have been proposed over the years. For performing matrix multiplication using on-chip resources only, they can achieve up to 300 GFlops=sec for single precision matrices on a state-of-the-art device [13]. However, the limited on-chip memory restricts the matrix sizes that can be supported.

Recently, Field Programmable Gate Arrays (FPGAs) have become a platform of choice for hardware realization of computation-intensive applications [1-13]. Especially, when the design at hand requires very high performance, designers can benefit from high density and high performance FPGAs instead of costly multicore Digital Signal Processing (DSP) systems [1]. FPGAs enable a high degree of parallelism and can achieve orders of magnitude speedup over GPPs [7]. This is as a result of the increasing embedded resources on FPGA. FPGA have the benefits of the hardware speed and the software flexibility; also they have a price/performance ratio much more favorable than Application Specific Integrated Circuits (ASICs). Since the major resources for implementing computation-intensive algorithms are embedded on FPGA, latency associated with device communication has been eliminated. However, these embedded resources are limited hence it is important to use these resources optimally.

Reconfigurable devices, in particular, Field-Programmable Gate Arrays (FPGAs), offer the design flexibility of software, but with time performance closer to Application Specific Integrated Circuits (ASICs). Due to their low computing density, early reconfigurable devices were mainly used for fixed point applications. However, with rapid advances in technology, current reconfigurable devices contain significantly more hardware resources than their predecessors. Hence, they are now feasible for a broader range of applications, including those requiring floating-point operations. Some researchers have even suggested that current reconfigurable devices can perform better than general-purpose processors in terms of peak and sustained floating-point performance [1].

Floating-point matrix multiplication is a basic operation in many scientific computing applications. Its implementations on reconfigurable computing systems are able to achieve high performance for several reasons. First, a single reconfigurable device contains enough configurable slices to hold multiple floating-point units that can perform computations concurrently. Secondly, current reconfigurable fabrics provide large amounts of on-chip memory as well as abundant number of I/O pins; this can alleviate performance bottleneck due to processor memory bandwidth. These two features enable designs on reconfigurable computing systems to exploit the inherent parallelism of matrix multiplication. Furthermore, current reconfigurable devices contain a large number of multiplexers and embedded fixed-point multipliers. These primitives can be leveraged to build high speed floating-point adders and multipliers. Although there have been some studies on FPGA-based fixed-point matrix multiplication [6], they cannot be applied directly to the floating-point counterpart. Due to the complexity of floating-point adders/multipliers, the design space of floating-point matrix multiplication is much larger. Recently some work has been done on FPGA-based floating-point matrix multiplication [1], [7]. However, to the best of our knowledge, none of the previous work has discussed the design tradeoffs among the area, the total storage size, the latency and the required memory bandwidth.

Tri-Matrix Multiplication: Design and Implementation

In this section we will present the design of tri-matrix multiplier which is commonly used in DSP applications [52]. Matrix T can be written as

T = XYZ (4)

where, X and Z are rectangular matrices given by (5) and (6) respectively. Y is a diagonal square matrix, where *n* = 0, 1, ..., N-1.

$$X = \begin{bmatrix} 0 & 0 & \cdots & 0 & x_1(0) & \cdots & x_1(N-k-1) \\ 0 & \cdots & \cdots & x_1(0) & x_1(1) & \cdots & x_1(N-k) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & x_1(0) & \cdots & \cdots & \cdots & \cdots & x_1(N-2) \\ x_1(0) & x_1(1) & \cdots & \cdots & \cdots & \cdots & x_1(N-1) \\ x_1(1) & x_1(2) & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_1(k) & x_1(k+1) & \cdots & x_1(2k-1) & x_1(2k) & \cdots & 0 \end{bmatrix}$$

Volume : 4 | Issue : 12 | December 2015 • ISSN No 2277 - 8179

$$Z = \begin{bmatrix} 0 & 0 & \cdots & x_2(0) & x_2(1) & \cdots & x_2(k) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & x_2(0) & \cdots & \cdots & \cdots & \cdots & x_2(2k-1) \\ x_2(0) & x_2(1) & \cdots & \cdots & \cdots & \cdots & x_2(2k) \\ x_2(1) & x_2(2) & \cdots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_2(N-k-1) & x_2(N-k) & \cdots & x_2(N-1) & 0 & \cdots & 0 \end{bmatrix}$$

Floating Point Numbers

The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point, that is the decimal point can float. There are also representations in which the number of digits before and after the decimal point is set, called fixed-point representation. Floating Point Numbers are numbers that can contain a fractional part. For e.g. following numbers are the floating point numbers: 3.0, -111.5, ½, 3E-5 etc. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point data type; computers from PC" s to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. A number representation (called a numeral system in mathematics) specifies some way of storing a number that may be encoded as a string of digits. In computing, floating point describes

Input 1 (32 bit)
Input 2 (32 bit)
Output 64 bit

Multiplier

A system for numerical representation in which a string of digits (or bits) represents a rational number. The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated separately in the internal representation, and floating-point representation can thus be thought of as a computer realization of scientific notation. Over the years, several different floating-point representations have been used in computers; however, for the last ten years the most commonly encountered representation is that defined by the IEEE 754 Standard. The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values. For example, a fixed point representation that has seven decimal digits, with the decimal point assumed to be positioned after the fifth digit, can represent the numbers 12345.67, 8765.43, 123.00, and so on, whereas a floating-point representation (such as the IEEE 754 decimal32 format) with seven decimal digits could in addition represent 1.234567, 123456.7, 0.00001234567, 1234567000000000, and so on. The floating-point format needs slightly more storage (to encode the position of the radix point), so when stored in the same space, floating-point numbers achieve their greater range at the expense of slightly less precision. There are several mechanisms by which strings of digits can represent numbers. In common mathematical notation, the digit string can be of any length, and the location of the radix point is indicated by placing an explicit "point" character (dot or comma) there. If the radix point is omitted then implicitly assumed to lie at the right (least significant) end of the string (that is, the number is an integer). In fixed-point systems, some specific assumption is made about where the radix point is located in the string. For example, the convention could be that the string consists of 8 decimal digits with the decimal point in the middle, so that "00012345" has a value of 1.2345. A floating-point number consists of A signed digit string of a given length in a given base (or radix). This is known as the significant or sometimes the mantissa (see below) or coefficient. The radix point is not explicitly included, but is implicitly assumed to always lie in a certain position within the significant often just after or just before the most significant digit, or to the right of the rightmost digit. This article will generally follow the convention that the radix point is just after the most significant (leftmost) digit. The length of the significant determines the precision to which numbers can be represented. A signed integer exponent also referred to as the characteristic or scale, which modifies the magnitude of the number. The significant is multiplied by the base raised to the power of the exponent, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent to the right if the exponent is positive or to the left if the exponent is negative. Using base-10 (the familiar decimal notation) as an example, the number 152853.5047, which has ten decimal digits of precision, is represented as the significant 1528535047 together with an exponent of 5 (if the implied position of the radix point after the first most significant digit, here 1).

Significance of IEEE 32 Bit Single Precision Format

For every decimal number example 2, 3, 7, 13 it is possible to write their equivalent binary representation. Binary representation of such numbers can be store inside digital storing elements like flip flops, different memories like RAM, ROM, PROM etc. Now when it is tattle to store floating numbers like 3.2, 4.5, 15.9 etc then simply round it & bring the numbers to the integer value. But in this method the original status of floating number lost. So in order to retain the original status of such floating numbers the IEEE 32 bit single precision format is used. So the single precision format provides us the standard numerical platform to work with floating numbers. The caption deals with high speed floating multiplier which means multiplication of two numbers having floating values. Therefore the IEEE single precision format substantiates to be an aiding hand for caption.

SIGN	EXPONENT	MANTISSA
1	8	23

<------------------------ 32 bits ------------------------→

IEEE 32 bit single precision format

Conversion of Floating Number Into IEEE Single Precision Format

Ex:- Convert 6.75 into single precision format
Solution - 6=110 in binary
.75 * 2 = 1.5
.5 * 2 = 1.0
.0 * 2 = 0.0
.0 * 2 = 0.0
110. 11000000000000000000000 =1.
10110000000000000000000000 * 2
Exponent = 127 + 2= 129 or 10000001 in binary
Mantissa = 10110000000000000000000
6.75 in 32 bit floating point IEEE representation:-

SIGN	EXPONENT	MANTISSA
0	10000001	10110000000000000000000

<--1bit-→<------8bit---→<----------23bits----------→

←------------------32bit---------------------------------→

IEEE 32 bit single precision format for given example

Proposed Plan

Design methodology for the hardware realization of computation intensive algorithm is a combined effort of Electronic Design Automation (EDA) tools, methods and FPGA technology that enables to produce the optimized circuit for the end applications. A right combination of FPGA hardware, designed IP core and EDA tools will definitely enhance the efficiency of the design methodology. By design methodology, we imply the step-by-step process of FPGA design. The FPGA design methodology is used as a guideline for the hardware realization of algorithms. A number of design flows are used by different FPGA vendors but all are basically similar in sequence of tasks performed. These steps are common in all FPGA EDA tools and are essential in today's FPGA design process. The EDA tools like Xilinx Integrated Software Environment (ISE), Altera's Quartus II and Mentor Graphics' FPGA Advantage plays a very important role in obtaining an optimized digital circuit using FPGA [13-14]. A typical FPGA design flow followed in this work is shown in fig. 2. In this flow, design Entry is used to describe the algorithm/circuit that has to be implemented onto the FPGA device. There are two standard approaches to specify the FPGA designs: HDL-based and Schematic based depending upon the complexity of FPGA design. However, for complex and computationally intensive algorithms HDL-based (VHDL or Verilog) design entry is the dominant method used by FPGA designers. After specifying the design using HDLs or Schematic, the designer needs to validate the logical correctness of the design. This is performed using functional or behavioral simulation. The design involves the computation of G = AC, where A is a matrix, C and G are vectors. We need to calculate vector G. Broadcast algorithm is adopted for the matrix-vector multiplication. The matrix– vector multiplication is performed by broadcasting rows of matrix A and multiplying the corresponding column elements of vector C. Following operations are involved:

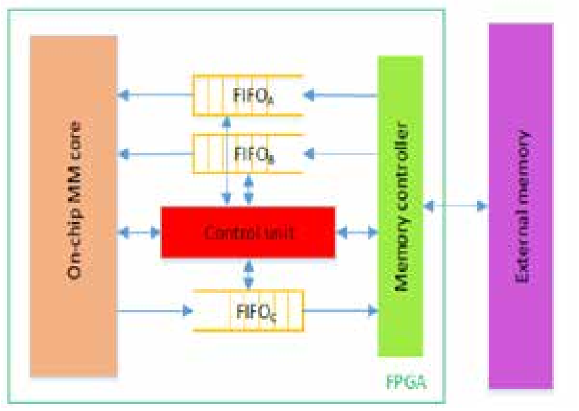
- ❖ Reading individual row elements of matrix A and individual column elements of vector C
- ❖ Storing them in internal buffers row and column wise respectively
- ❖ Multiplying row and column elements.
- ❖ Accumulating the multiplier output and writing back the results to the output buffers.

The input and output buffers are implemented on the FPGA. The matrix-vector multiplications involve multiply and accumulate operations. The multiply-accumulate unit consists of a multiplier and adder. The row and the column elements are supplied as the two inputs to the multiplier. The output of the multiplier is directly given to the adder as one of the inputs. The previous output of the adder is fed back as the second input to the adder. The multiply-accumulate unit takes each element of the matrix A in row major format and each element of vector C, multiplies them and adds the result to the running total. This process is repeated till the last element of row A and column C. The values are fed in a sequential manner. If the reset signal is asserted high, the contents of registers A and C are cleared. After a delay, as determined by the implementation results, the first element of vector G is available at the serial output and this output is stored in on-chip memory. This operation is repeated and the process continues until all the rows of matrix A are processed. Finally, the output vector G is available with all the elements stored in the memory locations.

In this work, we adapt the architecture and algorithm proposed in [9] for on-chip MM core (denoted On-chip MM). This algorithm achieves an optimal latency of O(m2) using m PEs for m_m MM. Also the layout is simple as all the interconnections are localized and are within a PE or in between adjacent PEs only. For m_m MM, On-chip MM consists of m identical PEs.

Each PE includes a multiplier and adder, 4 registers, and 2 m-word local memories. The On-chip MM architecture can perform a m_m MM in m2 + 2m cycles. To compute C = AB, matrix B is fed into the design in row major order, starting from the top row. Matrix A is fed into the design m cycles behind matrix B in column major order, starting from the left most column. The partial results are accumulated at each PE in an auxiliary memory of size m elements, CBuf. Once the multiplication of A and B is completed, an m elements size memory at each PE, COBuf, is used to transfer the resulting matrix C in the column major order, starting from the left most column. The detailed algorithm is described in [9].

Large-Scale Matrix Multiplication
Large-scale MM can be efficiently performed using tiled/blocked MM. Detailed procedure for tiled/blocked MM is shown in Algorithm 1 [6]. The input matrices and the product matrix are stored in external DRAM. The overall architecture for performing large-scale MM is shown in Fig. 2. As DRAM and the On-chip MM core can operate at different frequencies, FIFO's are used to synchronize the data access from the On-chip MM core to the DRAM. FIFOA and FIFOB are used to synchronize the reading of the input matrices A and B, respectively, from the DRAM memory to the On-chip MM core. FIFOC is used to synchronize the writing of the product matrix C, from the On-chip MM core to the DRAM memory. The control unit coordinates the operation of the entire design. As the timing of the operations in MM can be predetermined, control unit can be implemented using several counters. During a block operation, the partial results are kept in the On-chip MM core and are accumulated with the results in the next iteration.



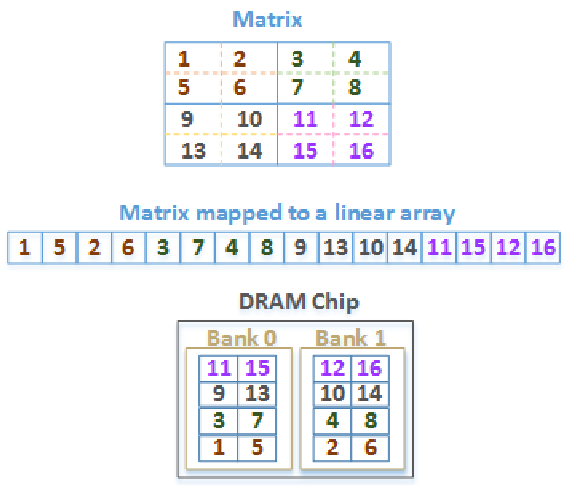
Large-Scale Matrix Multiplication
The partial results are accumulated in CBuf array at each PE of the On-chip MM core. Once a block of output sub-matrix C is produced, it is transferred to COBuf array at each PE of the On-chip MM core; eventually it is streamed out to DRAM using the FIFOC. During the streaming out process, the inputs for the next

Performance Metric
We consider energy efficiency as the metric for performance evaluation. Energy Efficiency is defined as the number of operations per unit energy consumed. When multiplying two nxn matrices using O(n3) MM algorithm, energy efficiency is given by 2n3=energy consumed by the design. Energy consumed by the design = time taken by the design _ average Fig. 3.2: Data layout for storing matrices power dissipation of the design. Alternatively energy efficiency of the design is Power efficiency (number of operations per second=Watt).

Data layout
In block matrix multiplication, input matrices are accessed in two dimensional blocks. Canonical data layouts such as row/column-major layouts store the adjacent elements in row/column in contiguous locations. Therefore, elements of a block are not stored in contiguous locations. So, for accessing next row/column elements of a block, we may need to access another row of the chip. Also if the adjacent row/column of the block is stored

Volume : 4 | Issue : 12 | December 2015 • ISSN No 2277 - 8179

in the same bank then tRC delay needs to be incurred in accessing the data.



Data Layout for Storing Matrices

Using the block data layout [15], we present a data layout to store the input matrices in DRAM. To describe the data layout, we map the matrix elements to a linear array. Also the locations of a DRAM chip are given a linear index. The matrix elements in the linear array are stored in DRAM chips at the corresponding indexes. As input and output matrices are transferred between DRAM and On-chip MM core in blocks of size m _ m, these matrices are divided into two dimensional blocks of size m _ m. An example of 4_4 matrix divided into 2_2 blocks is shown in Fig. 3. The adjacent blocks in a row (column) of matrices A and C (matrix B) are mapped to contiguous locations in the linear array. The adjacent block rows (columns) are mapped to contiguous locations in the linear array. The blocks of A and C matrices (B matrix) are stored in column major format (row-major format). An example data layout for a 4 _ 4 A matrix divided into 2 _ 2 blocks and stored in a DRAM chip with G = 2, I = 4, and H = 2 is shown in Fig. 3. In this data layout, elements within the blocks are stored according to the input data access order of the program. This data layout exploits the spatial locality in the data accesses of the MM core. Unlike the row/column major layout, in this data layout, the elements in the block of a matrix are stored in contiguous locations. Therefore, for an activated row, the number of accesses to it are maximized. By maximizing the number of accesses to an activated row, the total number of row activations required are minimized; hence reducing the energy consumed by the DRAM. In the row/column major layout, as the rows/columns of a block are not contiguously stored, the next accessed DRAM row can be in the same bank. The minimum time required for activating the row in the same bank is tRC. In the worst case the next accessed row in the block can always be located in the same bank and the row activation time cannot be overlapped with the data access. In this case, the overhead of row activation times for rows in a block is m_tRC. With the proposed data layout the next accessed DRAM is another bank and row activation can be overlapped with the data access. Therefore, when compared to the row/column major layout our proposed data layout can reduce the time for accessing one block of data by up to m_tRC. Minimizing the DRAM access time helps us to reduce energy by using the activation schedule described in Section V.

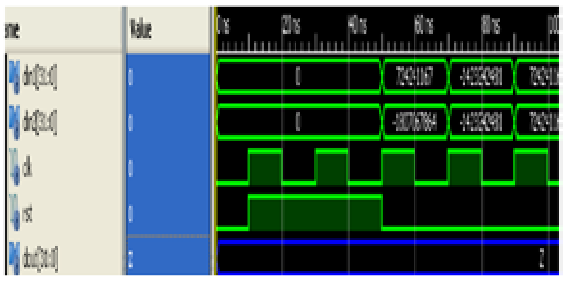
Energy Dissipation Analysis

In this subsection, we present an asymptotic analysis for the energy dissipation. We consider the architecture and algorithm described in Section IV. For n_n MM with m_m block size, number of block MM operations performed are (n=m)3. Each block MM operation takes m2+2m cycles. Therefore, the total time for computation is O(n3=m). Total energy consumed by the arithmetic units is O(n3), where constant units of energy is assumed to be consumed for performing an arithmetic operation. Amount of on-chip memory is O(m2) and off-chip memory is O(n2). Energy consumed by the on-chip memory units is O(n3m), where constant units of energy is assumed to be consumed in a cycle for a unit of storage. Energy consumed by the off-chip memory units is O(n5=m), where constant units of energy is assumed to be consumed in a cycle for a unit of storage in a DRAM chip which is operating in active power mode. Therefore, total energy consumed by the memory units is O(n3 _ (m + n2=m)). All the communication is in between the adjacent PEs, in between the components within the PEs, in between the on-chip and external memory, or in between adjacent units such as memory controller, FIFOs Fig. 4: Activation schedule for external memory only. Therefore the energy consumed for communication is O((n3=m) _ m), where unit energy is assumed to be energy for transferring a word of data within a PE, between adjacent PEs or between adjacent components. Total energy consumed by the design is sum of the computation, memory, and communication energies. The total number of operations performed in MM is 2n3. Energy efficiency of the design is O(m=n2) as n >> m; memory energy limits the scalability of the large scale MM design.

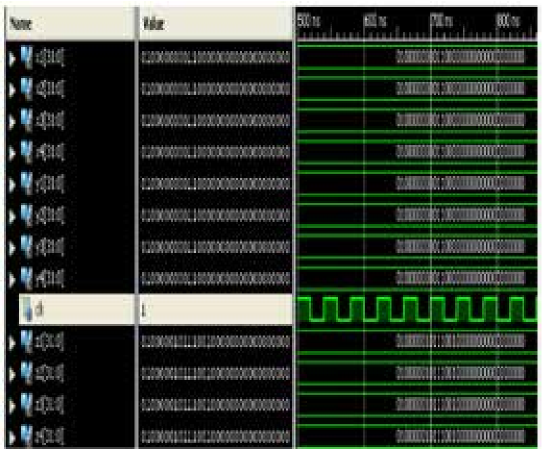
Conclusion

In previous work done the estimated dynamic power for the single precision 512*512 matrix multiplier was 17.2W. In our project the estimated dynamic power for the single precision 1024*1024 matrix multiplier is minimized up to 9.591W. In this paper we used floating point multiplier for that we use IEEE converter and with the help of this assembly we have easily generated output with minimum power and optimized structure. Initially we have taken two matrices and with the help of this floating point multiplier we able to generate output matrix and related simulation results.

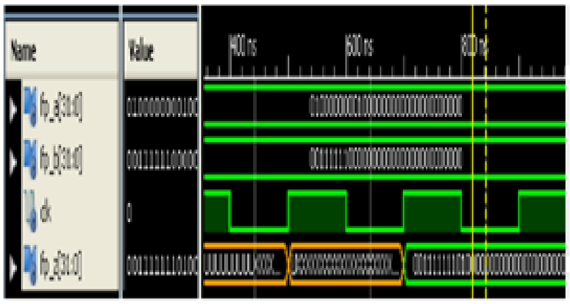
Simulation Results of 1024 Serial Data



Simulation Result of 1024 Inputs



Simulation Result of Floating Point Matrix Multiplier



REFERENCE

[1] Kiran Kumar Matam and Viktor K. Prasanna , "Energy-Efficient Large-Scale Matrix Multiplication, on FPGAs" 978-1-4799-2079-2/13/$31.00 c 2013 IEEE [2] Syed M. Qasim, Ahmed A. Telba and Abdulhameed Y. AlMazroo, "FPGA Design and Implementation of Matrix Multiplier Architectures for Image and Signal Processing Applications" IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.2, February 2010 [3] B.-Y. Jan1 and J.-L. Huang2, "A Fault-Tolerant PE Array Based Matrix Multiplier Design" 978-1-4577-2081-9/12/$26.00 ©2012 IEEE [4] Riya Saini*, R.D.Daruwala** , "Efficient Implementation of Pipelined Double Precision Floating Point Multiplier" International Journal of Engineering Research and Applications (IJERA) Vol. 3, Issue 1, January -February 2013, pp.1676-1679 [5] Rong Lin , "A Reconfigurable Low-Power High- Performance Matrix Multiplier Design" [6] Richard Dorrance, Fengbo Ren, Dejan Markovic, " A Scalable Space Matrix – Vector Multiplication Kernel for Energy-Efficient Sparse-Blas on FPGAs" [7] "Improved matrix multiplier design for high-speed digital signal processing applications" Prabir Saha1, Arindam Banerjee2, Partha Bhattacharyya3, Anup Dandapat1 www.ietdl.org [8] Franc¸ois Le Gall , "Faster Algorithms for Rectangular Matrix Multiplication" 2012 IEEE 53rd Annual Symposium on Foundations of Computer Science [9] A. Amira and F. Bensaali. An FPGA based parameterizable matrix product implementation. In Proc. of IEEE Workshop on Signal Processing Systems, pages 75-79, 2002. [10] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. In IEEE TPDS, 13(11), 1105-1123, 2002. [11] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In Proc. of 18th IPDPS, page 92. IEEE, 2004. [12] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. In IEEE TPDS, 18(4):433-448, 2007. [13] R. Scrofano, S. Choi, and V. K. Prasanna, "Energy Efficiency of FPGAs and Programmable Processors for Matrix Multiplication," in Proc. of IEEE Intl. Conf. on Field Programmable Technology, pp. 422-425, 2002. [14] J. Jang, S. Choi, and V. K. Prasanna, "Energy and Time Efficient Matrix Multiplication on FPGAs," IEEE Trans. On Very Large Scale Integration (VLSI) Systems, Vol. 13, No. 11, pp. 1305-1319, 2005. [15] S. Belkacemi, K. Benkrid, D. Crookes, and A. Benkrid, "Design and implementation of a high performance matrix multiplier core for Xilinx Virtex FPGA," in Proc. of IEEE Intl. Workshop on Computer Architectures for Machine Perception, pp. 156-159, 2003. [16] J. Jang, S. Choi, and V. K. Prasanna, "Energy efficient matrix multiplication on FPGAs," in Proc. of 12th Intl. Conf. on Field Programmable Logic and Applications, pp. 534-544, 2002. [17] L. Jianwen and J. C. Chuen, "Partially Reconfigurable Matrix Multiplication for Area and Time Efficiency on FPGAs," in Proc. of Euromicro Symp. on Digital System Design , pp. 244-248, 2004. [18] S. A. Alshebeili, "Computation of higher-order cross moments based on matrix multiplication", Journal of the Franklin Institute, 338, pp. 811-816, 2001. [19] S. Choi, V. K. Prasanna, and J. Jang, "Minimizing energy dissipation of matrix multiplication kernel on Virtex-II," in Proc. of SPIE, Vol. 4867, pp. 98-106, 2002. [20] F. Bensaali, A. Amira, and A. Bouridane, "An FPGA based coprocessor for large matrix product implementation," in Proc. of IEEE Intl. Conf. on Field Programmable Technology, pp. 292-295, 2003.