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.

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

**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 GFIops/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

\[ T = XYZ \]  

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

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, \( \frac{1}{2} \), 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 PCs 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

\[
\begin{array}{c|c|c|c|c}
\text{Input 1 (32 bit)} & \text{Input 2 (32 bit)} & \text{Output 64 bit} \\
\hline
0 & 0 & \ldots & 0 & x_0(0) & \ldots & x_{(N-1)}(N-1) \\
0 & \ldots & \ldots & \ldots & 0 & \ldots & \ldots \end{array}
\]

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 (5) 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 (6) format) with seven decimal digits could in addition represent 1.234567, 123456.7, 0.00001234567, 12345670000000000, 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

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{Multiplier} & \text{Input 1 (32 bit)} & \text{Input 2 (32 bit)} & \text{Output 64 bit} \\
\hline
0 & 0 & \ldots & 0 & x_0(0) & \ldots & x_{(N-1)}(N-1) \\
0 & \ldots & \ldots & \ldots & 0 & \ldots & \ldots \end{array}
\]
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 talked 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.

Conversion of Floating Number Into IEEE Single Precision Format

Ex:- Convert 6.75 into single precision format

Solution - 6=110 in binary

\[0.75 \times 2 = 1.5\]

\[0.5 \times 2 = 1.0\]

\[0.0 \times 2 = 0.0\]

\[0.0 \times 2 = 0.0\]

110.1100000000000000000000000000000 =1.

1011000000000000000000000000000 * 2

Exponent = 127 + 2= 129 or 10000001 in binary

Mantissa = 10110000000000000000000

6.75 in 32 bit floating point IEEE representation:-

<table>
<thead>
<tr>
<th>SIGN</th>
<th>EXPONENT</th>
<th>MANTISSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000001</td>
<td>1011000000000000000000000000000</td>
</tr>
</tbody>
</table>

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-
are vectors. We need to calculate vector G. Broadcast 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 $O(m^2)$ using $m$ PEs for $m \times m$ MM. Also the layout is simple as all the interconnections are localized and are within a PE or in between adjacent PEs only. For $m \times m$ 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 $m \times m$ MM in $m^2 + 2m$ 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 in 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. FIFOA 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](image)

**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 \(O(n^3)\) MM algorithm, energy efficiency is given by \(2n^3\) energy consumed by the design. Energy consumed by the design = time taken by the design - average. Fig. 3.2: Data layout for storing matrices 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 \(t_{RC}\). 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 \(m_t_{RC}\). 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 \(m_t_{RC}\). 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 \(m^2+2m\) cycles. Therefore, the total time for computation is \(O(n^3m^3)\). Total energy consumed by the arithmetic units is \(O(n^3)\), where constant units of energy is assumed to be consumed for performing an arithmetic operation. Amount of on-chip memory is \(O(m^2)\) and off-chip memory is \(O(n^2)\). Energy consumed by the on-chip memory units is \(O(n^3m)\), 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 \((m^5n^5)\), 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 \(O(n^3m^3)\). 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 \(O(n^3m)\), 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 $2n^3$. Energy efficiency of the design is $O(m/n^2)$ as $n \gg m$; 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.

References:


