Resource and Delay Efficient Matrix Multiplication using Newer FPGA Devices

Scott J. Campbell
Department of ECE
University of Colorado, Boulder
Boulder, CO 80309

Sunil P. Khatri
Department of ECE
Texas A&M University
College Station TX 77843

ABSTRACT

Matrix multiplication is a fundamental building block for many applications including image processing, coding, and digital signal processing. This paper presents a delay and resource efficient methodology for implementing integer and floating point matrix multiplication using FPGAs. We present a scalable architecture based on new FPGA features that provides a significant reduction in total computation time and resource utilization over previous solutions. The implementation of our algorithm for various matrix dimensions using Xilinx FPGAs is also described. When compared to the best previously reported method, our approach achieves an improvement in the parallelization of 60% for 64-bit floating point computations.

Categories and Subject Descriptors: B.6.1 [Logic Design]:Design Styles

General Terms: Algorithms, Design

Keywords: FPGA, Matrix, Multiplier, Floating Point

1. INTRODUCTION

Matrix multiplication is a fundamental operation for many applications including image processing, coding, and digital signal processing (DSP). Because thee operations are vital, and processors implement matrix multiplication in $O(n^3)$ run-time, many parallel methods have been developed to reduce this complexity [1] [2] [3]. While these parallel processing methods have reduced the total computation time, *they are applicable only for small matrix dimensions since they require significant device resources.* However, recent advances in Field Programmable Gate Array (FPGA) technology now allow for more efficient implementation of parallel matrix multiplication algorithms.

The contribution of this paper is to present a methodology for implementing integer and floating point matrix multipliers using FPGAs which take advantage of the new FPGA technology. As our results illustrate, our new method performs significantly faster, while utilizing much fewer FPGA resources than previous solutions. The remainder of this paper is organized as follows: Section 2 discusses prior work in this area. In Section 3 we describe our new matrix multiplication design. Section 4 presents experimental results comparing our method with previous implementations. Finally, conclusions are drawn in Section 5.

GLSVLSI'06, April 30–May 2, 2006, Philadelphia, PA, USA.

2. PREVIOUS WORK

There has been extensive previous work in the area of designing a parallel implementation of the matrix multiplication operation. Recent work has focused on using FPGA technology to implement these algorithms.

In [1] Jang et al. presented an integer/fixed-point multiplication design that reduced the overall resource utilization and latency of the operation over previous designs such as [2] by taking advantage of data re-use. This design was extended to serve as a co-processor for the on-chip microprocessors of Xilinx Virtex-II family in [4] which maintained the operating frequency while reducing the resource utilization of the design. In [3] a systematic algorithm for floating-point matrix operations was presented which expanded on the work in [1] to decrease latency. Finally, [5] extended the floating-point algorithm work in [3] to consider multiple FPGA's and overall system design.

While these prior works have systematically improved the resource utilization and operating frequency over prior designs, *their algorithms all require intercommunication between the processing elements (PE's).* This communication requires extensive routing and logic, and limits the number of PE's that can be implemented as well as the operating frequency of the design. The aforementioned limitations have practically restricted the prior work to realizing very small parallelization factors p. In particular, our approach achieves an improvement of 60% in the value of the parallelization factor for a 64-bit floating point matrix multiplication compared to the best known prior approach [3].

New advances in FPGA technology allow more efficient implementation of integer and floating-point multiplication and addition over previous generations of devices. For example, the Xilinx Virtex-4 family introduced the dedicated DSP-48 block which consists of a 18×18 multiplier coupled with a 48-bit adder / accumulator[7]. In addition, the Virtex-4 family includes dedicated FIFOs which allows efficient implementation of both synchronous and asynchronous FIFOs without requiring additional logic resources. With the larger versions of Virtex-4 device family including up to 512 of the DSP-48 blocks, and 320 BlockRams [6], the family is ideal for implementing large dimension matrix multipliers.

3. OUR APPROACH

Several methods have been explored to decrease the algorithmic complexity of the matrix multiplication operation as discussed in Section 2. In order to achieve maximum data re-use, these methods have focused on using complex PE's to both store and transfer intermediate results throughout the processing array. While this improves the latency of the system, it is not delay or resource optimal due to data sharing between the PE's. *To the best of our knowledge, our method is the first to perform matrix multiplication with PE's that operate in isolation from each other.*

3.1 New Parallel Model

In order to optimize FPGA architecture resource use, the data from input matrices A and B should be re-used. Optimal data re-use occurs when data is read from memory for matrices A and B exactly once. By simultaneously reading one column of matrix A and one row of matrix B, and performing all multiply operations based on those values before additional memory reads, optimal data re-use occurs. Data read in this sequence allows one partial product term of every element in output matrix C to be computed per clock cycle. This process is shown in Figure 1.

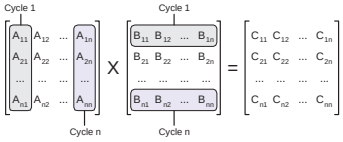

Figure 1: Parallel Matrix Multiplication Sequence

3.1.1 Multiplier Array Architecture

In order to meet the memory read requirements, an array structure of PE's is required. For example, Figure 2 shows the array structure required when input matrix A has l elements, and B has m elements.

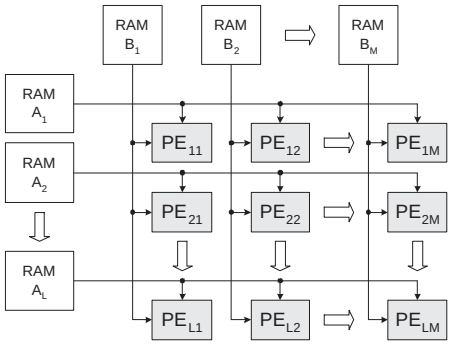

Figure 2: Our Matrix Multiplier Array

In this array configuration, lm product terms are produced each clock cycle from the $l + m$ matrix elements. In the sequel, we assume that $m = l$, allowing for a regular multiplier array structure. In all but the smallest of matrix dimensions n where $m = n$, it is assumed that $m = n/k$ where k is an integer. This array configuration allows the algorithm to be applied to matrices of non-square dimension. *It is important to note that in our approach, no intercommunication exists between the PE's as required in previous architectures. This novel structure significantly reduces the routing resources required in the design and removes the associate performance penalty. In addition, because each PE operates independently of other PE's in the array, the operating frequency of the PE array is independent of matrix dimension.*

3.1.2 PE Structure

The structure of the PE is shown in Figure 3.

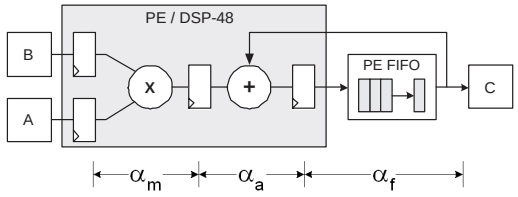

Figure 3: Our Processing Element (PE) Structure

The PE structure consists of one input each from matrix A and B, a multiplier accumulator (MAC), and a result FIFO. In Figure 3, the multiplier latency is denoted as α_m, while that of the adder and FIFO are denoted as α_a and α_f respectively. The inputs from matrices A and B, containing one word each per clock cycle, are implemented using dedicated routes from the BlockRam memory associated with the multiplier. By having dedicated memory connections for each PE, multiplexing between several inputs sources is not required, and the so the routing and resource delay penalty is eliminated.

During the computation of output matrix element C_{ij} the product term $A_{ik} \cdot B_{ki}$ must be available at the output of the adder during the same clock cycle as the product term $A_{i(k+1)} \cdot B_{(k+1)i}$ is available from the multiplier. For matrix arrays of dimension $k^2 < \alpha_a$, α_i idle cycles are required between successive input memory loads. The final product term requires α_a of latency in the adder before it is stored in memory. The latency of entire operation L_m in these systems is given by Equation (3).

3.1.3 Memory Structure

The memory hierarchy consists of input matrices A and B, and output Matrix C FIFO storage. Each matrix is partitioned into m BlockRam banks. This structure has one bank of A and B feeding one PE array row and column respecitvely. Each bank of A stores $k = (n/m)$ words of each column in A, for every row of A. Each bank of B stores $k = (n/m)$ words of each row in B, for every column of B. By storing data in this manner, matrices of arbitrary dimension can be implemented. Because the data remains in FIFO memory only until used, additional data can be loaded from external memory to accomodate array dimensions larger than internal storage would allow.

Each BlockRam has a depth d based on the width of the internal data. To determine the number of BlockRams required per matrix word, the input data word width (W_{mat}) is divided by the BlockRam width (W_{bram}) and rounded up. This is multiplied by the number of words in the matrix n^2 and divided by the FIFO depth d. The total number of input BlockRams required for matrices A and B are as presented in Equation (1).

$$InputBlockRams = 2 * \lceil (\lceil (W_{mat}/W_{bram}) \rceil * (n^2)/d) \rceil \quad (1)$$

The output memory C for each PE, as seen in Figure 3, contains a FIFO to synchronize the product terms. The alignment FIFO within the BlockRam for each PE, the number of FPGA fabric resources is significantly reduced. Because the FIFO contains the contents of the Multiply and Accumulate (MACC) operation, no additional latency cycles are required to transfer the results from the PE to the C matrix memory.

For large matrices, each PE FIFO stores k^2 terms of output matrix C. The number of PE BlockRams required is the width of the input word (W_{PE})divided by the BlockRam (W_{bram}) width and rounded up. Note that the data width at the output of the matrix MAC operation is wider than that of the input matrices, and will thus require more BlockRams per word. This value is multiplied by the number of PE words (k^2) and divided by the FIFO depth d. The total number of BlockRams required to store the result is given in Equation (2).

$$PEBlockRams = \lceil (W_{PE}/W_{bram}) * (k^2/d) \rceil \quad (2)$$

3.2 System Latency and Computation Time

This section discusses the latency and computation time of the new system. The latency is defined as the time between reading the first elements from the input matrices, A_{11} and B_{11}, and writing the first element C_{11} to the result matrix. The total computation time is the time elapsed between reading the first elements from the input matrices, A_{11} and B_{11}, and writing the final result matrix element C_{nn} to memory.

3.2.1 System Latency

In our design , where $k = n/m$, it requires k^2 clock cycles to read the input data which produces the product terms for one column of A multiplied by one row of B. With n rows and n columns in our design, and each column of A and B read simultaneously, there are nk^2 clock cycles required to read the input matrices from memory. This means that after $[(n-1)k^2+1]$ clock cycles the elements A_{1n} and B_{n1} which are required for the final product term of C_{1n} will be read from memory. Given that the multiplier latency is α_m and the adder latency is α_a, the first result C_{1n} will be presented to the PE FIFO after $[(n-1)k^2+1]+\alpha_m+\alpha_a$ clock cycles. Because the PE FIFO also serves as the output matrix C storage, no additional clock cycles are required to store the result. The latency formula of our system is given in Equation (3).

$$L_m = [(n-1)k^2+1]+\alpha_m+\alpha_a \quad (3)$$

3.2.2 Total Computation Time

The input matrix elements A_{nn} and B_{nn} required for the final product term of C_{nn} are read from memory after nk^2 clock cycles. Given the multiplier latency of α_m and the adder latency of α_a, the final result C_{nn} will be written to the PE FIFO after $nk^2+\alpha_m+\alpha_a$ clock cycles. Again, with no addition time required for storing result C_{nn} to memory, the total computation time is given in Equation (4).

$$Total\ \ Computation\ \ Time = nk^2+\alpha_m+\alpha_a \quad (4)$$

While the total computation time of prior approaches is dependent on the PE storage capacity [3], Equation (4) shows the total computation time in our architecture is entirely independent of PE storage capacity, and depends only on the number of PE's available in the device. In addition, because the adder and multiplier latency has only $O(1)$ affect on total computation time, computation time is largely independent of latency.

4. EXPERIMENTAL RESULTS

In this section we will discuss the implementation of our matrix multiplication algorithm in a Xilinx Virtex-4 XC4VSX55 -12 FPGA. This device is optimized for DSP applications, and contains 512 DSP-48 blocks combined with 320 BlockRam components [6]. The discussion will include resource utilization, performance metrics for both integer and floating-point matrix multiplications, and a comparison of our algorithm with recently reported designs.

4.1 Integer Fixed Point Implementation

This section discusses the resource utilization and algorithm performance for integer and fixed point systems. These systems are characterized by efficient multiplier and adder implementation in the PE, and have low result latency for both those functions.

The memory and DSP-48 resource utilization of the matrix multiplication implementation depends on matrix word size and the dimension of the processor array. Table 1 shows the resource utilization for the 16-bit integer multiplication.

n	m	DSP-48	Inp BRAM	PE BRAM	Tot BRAM
8	8	64	16	128	144
16	8	64	32	128	160
32	8	64	32	128	160
64	8	64	32	128	160
128	8	64	128	128	256

Table 1: 16-bit Integer Resource Utilization

The performance of the matrix multiplier is dependent on the system operating frequency, latency, and total computation time. The BlockRam and DSP-48 frequency is well characterized, and operates at 400 MHz for the -12 speedgrade for the XC4VSX55 [6]. The matrix multiplication latency is given in Equation (3) while the total computation time is given by Equation (4). Table 2 lists the latency and total computation time for various matrix dimension operating at 400 MHz.

n	Lat Cycles	Tot Cycles	Lat Time (ns)	Total (ns)
8	10	10	25	25
16	63	66	158	165
32	499	514	1248	1285
64	4035	4098	10088	10245
128	32515	32770	81288	81925

Table 2: 16-bit Integer Performance

4.2 Double Precision Implementation

This section discusses the double precision floating point implementation of the algorithm compliant with IEEE specifications [8]. Floating point operations typically contain complex multiply and add functions. The complex nature of these operations causes the latency to be much higher than integer implementations.

The Xilinx 64-bit double precision Floating Point Operator Core requires 16 DSP-48 blocks and 396 slices per core [9]. Because of the large DSP-48 requirement, the PE

array dimension m was chose to be 4 for a total of 16 PE's. In addition, the 64-bit double precision floating point format requires two 36-bit BlockRam words of storage for each matrix word. The resource requirements for the double precision implementation are shown in Table 3.

n	Inp RAM	DSP-48	PE RAM	Slice	Tot RAM
8	16	256	32	6336	48
16	16	256	32	6336	48
32	16	256	32	6336	48
64	32	256	32	6336	64
128	128	256	64	6336	192

Table 3: Double Precision Resource Utilization

The performance of the Xilinx double precision Xilinx Floating Point Operator Core is listed as 279 MHz [9]. For the purposes of this experiment we chose the operating frequency to be 250 MHz. Table 4 shows the performance of our algorithm for various array dimensions. In this table, the latency of the matrix multiplication is given in Equation (3) while the total computation time given by Equation (4).

n	Lat Cycle	Tot Cycle	Lat Time (ns)	Tot Time (ns)
8	118	129	472	516
16	274	289	1096	1156
32	2018	2081	8072	8324
64	16162	16417	64648	65668
128	130082	131105	520328	524420

Table 4: Double Precision Performance

4.3 Integer Performance Comparison

In an attempt to normalize the results across the different designs, we use p to represent the number of parallel PE's in the design, and n to represent the dimension of the matrices. In addition, we extrapolate the resources required for a 16-bit 8×8 matrix multiplication based on the previously published work [1] [4]. Table 5 shows the results of our comparison for the integer multiplier.

Design	Computation Cycles	Area (CLB's)
Our Method	$n^3/p + \alpha_m + \alpha_a$	128
Corsonello et al. [4]	$11n^3/p + 5n^2$	981
Jang et al. [1]	$n^3/p + 2n^2/p$	560 est.

Table 5: Integer Performance Comparison

Table 5 shows that our method decreases the total integer matrix computation time by eliminating the n^2 component of the complexity equation present in the other solutions. This complexity reduction is made possible by our removing the connection between PE's, which allows the PE's to operate in isolation from one another, combined with our unique method of reading the input matrix data. *In addition, the CLB resource requirement for the PE's is reduced significantly because our method utilizes the built-in hardware FIFOs in the FPGA, and also because we utilize the same BlockRam for the local PE memory and for storing the result matrix C. Finally, the previous approaches requires the FPGA CLBs to communicate over the FPGA routing fabric, possibly yielding a lower clock rate than our method, in which each PE operates independently (without any communication between PEs).*

4.4 Floating Point Performance Comparison

The results presented below are based on normalizing the results of other designs to our notation. Table 6 presents the comparison of our design to recently published work for the 8×8 double precision floating point multiplication.

Design	Computation Cycles	Area Slices
Our Method	$n^3/p + \alpha_m + \alpha_a$	6336
Prasanna et al. 1	$n^3/p + p + n + \alpha_m + \alpha_a - 1$	19045
Prasanna et al. 2	$n^3/p + 2p + n + \alpha_m + \alpha_a - 2$	33589

Table 6: 8×8 Double Precision Performance

Table 6 shows that our method dramatically decreases the total large array computation time by eliminating the $(p+n)$ complexity component present in the other solutions [3]. It is important to note that for $p = n^2$ other designs require an additional n^2 complexity over our solution. In addition, our solution also significantly reduces the resource utilization over other designs.

Additionally, we performed a study to compare the parallelization factor p for [3] with our method. Using the same FPGA device (the Xilinx Virtex-4 XC4VSX55 -12), the method of [3] achieves a p value of 10 for 64-bit floating point computations. **Our method, on the other hand, achieves a p value of 16 for 64-bit floating point computations. This represents an improvement of 60% over the approach of [3]. Additionally, the approach of [3] requires the FPGA CLBs to communicate over the FPGA routing fabric, possibly yielding a lower clock rate than our method, in which each PE operates independently.**

5. CONCLUSIONS

In this paper we have presented a delay and resource efficient implementation of the matrix multiplication algorithm for fixed-point and floating-point designs. We take advantage of recent FPGA technology to significantly reduce the resource utilization and total computation time over previous methods. These advances have been made possible by our novel algorithm which removes the intercommunication between parallel processing elements (PEs), and allows each PE to operate in isolation. This algorithm allows the implementation using matrices of arbitrary dimension, and scales in performance with more DSP-48 blocks and memory.

6. REFERENCES

[1] J. Jang, S. Choi, and V. Prasanna, "Area and time efficient implementations of matrix multiplication on FPGAs,," in *Proceedings, IEEE International Conference on Field Programmable Technology (FPT)*, pp. 93–100, Dec 2002.

[2] A. Amira and F. Bensaali, "An FPGA based parameterizable system for matrix product implementation," in *IEEE Workshop on Signal Processing Systems (SIPS)*, pp. 75–79, Oct 2002.

[3] L. Zhuo and V. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on FPGAs," in *Proceedings, International Parallel and Distributed Processing Symposium*, p. 92, Apr 2004.

[4] P. Corsonello, S. Perri, and P. Zicari, "A matrix product coprocessor for FPGA embedded soft processors," in *Proceedings, International Symposium on Signals, Circuits and Systems (ISSCS)*, vol. 2, pp. 489–492, July 2005.

[5] J. Jang, S. Choi, and V. Prasanna, "Energy- and time-efficient matrix multiplication on FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, pp. 1305–1319, Nov 2005.

[6] Xilinx, "Virtex-4 FPGA users guide." www.xilinx.com.

[7] Xilinx, "XtremeDSP design considerations." www.xilinx.com.

[8] I. of Electrical and E. Engineers, "IEEE 754 standard for binary floating-point arithmetic," 1984.

[9] Xilinx, "Floating point operator v1.0." www.xilinx.com.