Matrix Multiplication on FPGA-Based Platform

Tai-Chi Lee, Mark White, and Michael Gubody

Abstract—**In this paper, the implementation of matrix multiplication using FPGA-Based computing platform is investigated. Because the highly parallel nature of matrix multiplication it makes an ideal application for using such platform. The computations are done in parallel by multipliers and adders, which are implemented on multiple FPGA boards. The major challenge of this task is I/O interfaces between PC and FPGA board. In our approach, we adopt the software/hardware codesign to exploit the highly parallel nature of operations, which cannot be exploited in our purely software implemen-tation. Our matrix multiplier is modeled in VHDL and runs on an ARC-PCI FPGA board. The purpose of the software part of our codesign system is to provide I/O to the hardware. This part is implemented on a PC with a C program and a device driver to communicate with the board. We present the performance comparison of our codesign and purely software implementation, as well as the performance comparison of existing parallel implementations.**

Examples of applications that require large, fast matrix multiplication are bipartite graph determination (non-existence of odd cycles), Economics (Leontief input-output model), power-invariant transformations (power systems), Cryptography, and Genetics modeling (Markov chains), and Fractal image compressions .

Index Terms—**VHDL, FPGA, Codesign, Multiplier, Cycle**

I. Introduction

Matrix multiplication is an important operation in applications such as bipartite graph determination (non-existence of odd cycles), Economics (Leontief input-output model), power-invariant transformations (power systems), Cryptography, and genetics modeling (Markov chains). Consider the following n × n matrix multiplication [1].

Manuscript received June 17, 2013; revised August 20, 2013. This work was supported in part by the Foundation Resource Grant at Saginaw Valley State University.

Tai Chi Lee is with the Department of Computer Science & Information Systems, Saginaw Valley State University, University Center, MI 48710 USA (phone: 989-964-4483; e-mail: lee@svsu.edu).

Mark H. White is a student in the Department of Computer Science & Information Systems, Saginaw Valley State University, University Center, MI 48710 USA (e-mail: mhwhite@svsu.edu).

Michael J. Gubody is a student in the Department of Computer Science & Information Systems, Saginaw Valley State University, University Center, MI 48710 USA (email: mjgubody@svsu.edu.

An *n x n* Matrix Multiplication Example for *n =2*

$$\begin{vmatrix} a11 & a12 \\ a21 & a22 \end{vmatrix} \times \begin{vmatrix} b11 & b12 \\ b21 & b22 \end{vmatrix} = \begin{vmatrix} c11 & c12 \\ c21 & c22 \end{vmatrix}$$

$$c11 = a11b11 + a12b21$$
$$c12 = a11b12 + a12b22$$
$$c21 = a21b11 + a22b21$$
$$c22 = a21b12 + a22b22$$

As shown above, the multiplication of matrix *a* by matrix *b* consists of many multiplication and addition operations, which can be easily modeled in a software program The C language code for *n* × *n* matrix multiplication may be given as follows:

```
#define N 2

void main() {

    unsigned int a[N][N], b[N][N], c[N][N];
    unsigned int i, j, k;

    // initialize matrix values
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            a[i][j] = 15;
            b[i][j] = 15;
        }
    }

    // do matrix multiplication
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
        c[i][j] = a[i][N - 1] * b[N - 1][j];
        for (k = 0; k < (N - 1); k++) {
                c[i][j] += a[i][k] * b[k][j];
        }
            }
    }
}
```

In particular, let $a_{ij} = b_{ij} = 5$ *, for all i = 1,2, 3 and j = 1,2, 3* then the resulting matrices are:

$$\begin{vmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{vmatrix} \times \begin{vmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \\ 5 & 5 & 5 \end{vmatrix} = \begin{vmatrix} 75 & 75 & 75 \\ 75 & 75 & 75 \\ 75 & 75 & 75 \end{vmatrix}$$

ISBN: 978-988-19252-3-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

This purely software implementation of matrix multiplication is accomplished through iterative processing. Observation of the matrix multiplication equations shows that the multiplications can be performed concurrently, and then the additions can be performed concurrently. This parallelism can be exploited to increase processing speed via a codesign, which is the simultaneous design of hardware and software subsystems [2].

II. IMPLEMENTATION

The hardware part of our codesign system is responsible for performing the arithmetic operations. This includes the matrix multiplier, which performs concurrent multiplication and addition operations of matrix multiplication. Our matrix multiplier is modeled in VHDL and runs on an ARC-PCI FPGA board [3]. The purpose of the software part of our codesign system is to provide I/O to the hardware. This part is implemented on a PC with a C program and a Windows XP device driver to communicate with the board. Figure 1 shows our codesign system interaction.

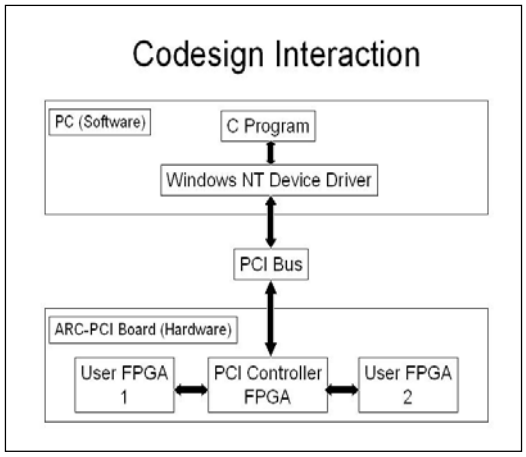


Fig. 1. Layout of Codesign Sceme.

In our purely software implementation, an n x n matrix multiplication requires n^3 multiplications and $(n^2 * (n - 1))$ additions. We define $f(n)$ as the total number of arithmetic operations required. Therefore,

$$f(n) = n^3 + (n^2 * (n - 1)) \\ += 2n^3 - n^2$$

The complexity is of $O(n^3)$.

In an ideal hardware implementation of matrix multiplication, all of the multiplications can be performed in parallel by multipliers on multiple FPGA boards, which take one clock cycle and then all of the additions can be performed concurrently by adders after that. Since the result can be computed in these two sets of concurrent arithmetic operations, $f(n) = 1+(n-1) = n$, which has the complexity of $O(n)$.

This ideal method may require an impractically large amount of hardware. A more realistic algorithm takes advantage of the parallel nature of matrix multiplication, but partitions the algorithm into groups of sequential block operations. For an $n \times n$ matrix, we use a partitioning scheme that divides the algorithm into n distinct sequential blocks. The following shows an example of our partitioning scheme.

Sequential Block Partitioning Example (n = 2)

$$\begin{vmatrix} a11 & a12 \\ a21 & a22 \end{vmatrix} \times \begin{vmatrix} b11 & b12 \\ b21 & b22 \end{vmatrix} = \begin{vmatrix} c11 & c12 \\ c21 & c22 \end{vmatrix}$$

Block 1

$$c11 = a11b11 + a12b21$$
$$c12 = a11b12 + a12b22$$

Block 2

$$c21 = a21b11 + a22b21$$
$$c22 = a21b12 + a22b22$$

Each sequential block is composed of one parallel multiplication and one parallel addition cycle, so 2 arithmetic computation cycles are required for 2x2 matrix multiplication. And two additional cycles are required to clock data through the matrix multiplier. So a total of 6clock cycles is required for 2x2 matrix multiplication.

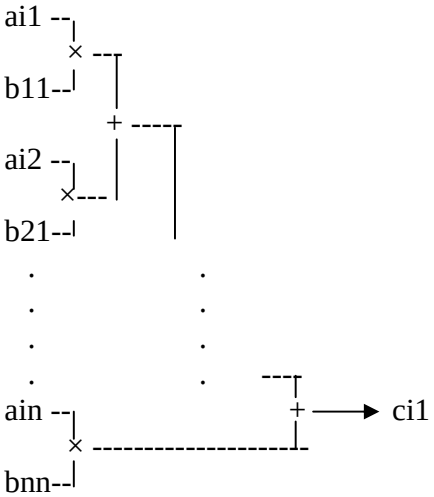
For $n \times n$ matrix multiplication, each sequential block (see ith Block below) is composed of one parallel multiplication and (n-1) addition cycle, so $1+(n-1)$ arithmetic computation cycles are required for each block . And an additional cycle is required to clock data through the matrix multiplier. So a total of $(n+1)$ clock cycles is required for each block. Therefore, the total number of clock cycles for such partitioning for n x n matrix multiplication is $f(n) = n*(n+1) = n^2 + 1$, which is $O(n^2)$, a slight improvement of one order over the purely software approach.

The following shows the ith Block containing the ith row entries of the product matrix C.

ISBN: 978-988-19252-3-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
ci1 =  ai1b11+ai2b21+…+ainbn1
ci2 =  ai1b12+ai2b22+…+ainbn2
                    .
                    .
                    .
cin =  ai1b1n+ai2b2n+…+ainbnn
```

The multiplier's operations resulted in lst entry ci1 of the block i can be shown as follows:



Note that, if the partition blocks are executed in parallel with one cycle to clock data to all multipliers at the same time, then the complexity would have been reduced to $f(n) = 1+(n-1)+1 = (n+1)$, which is $O(n)$, an improvement of two orders over purely software approach, but at a greater cost of hardware.

III. Test Results and Analysis

We implemented an unsigned, 4-bit, 3 x 3 matrix multiplier in VHDL for testing our codesign. In our purely software implementation, we have $f(n) = 2n^3 = 54$ arithmetic cycles. In our codesign, we have $f(n) = 4n = 12$ arithmetic cycles. Our purely software implementation took 10 μs to run, whereas our codesign took 120 μs to run. In this case where $n = 3$, our purely software implementation greatly outperforms our codesign. We will show how our codesign outperforms our purely software implementation as n increases.

First, we will examine the arithmetic computation part of our codesign. In our test PC, the CPU runs at 233 MHz, and the ARC-PCI board runs at the PCI bus frequency of 33 MHz. We know that our parallel-oriented codesign has fewer arithmetic computation cycles than our serial-oriented purely software implementation, but our purely software arithmetic computation rate of 233 MHz is faster than our codesign arithmetic computation rate of 33 MHz. We would like to find n for the break-even point in arithmetic computation time for our codesign and purely software implementations. Our purely software arithmetic computation time is ($2n^3 - n^2$ cycle seconds) / (233,000,000 cycles). Our codesign arithmetic computation time is (4n cycle seconds) / (33,000,000 cycles). The following shows the breakeven point in the arithmetic computation time for our two implementations.

Breakeven Point for Arithmetic Computation Time

$$\frac{2n^3 - n^2}{233} = \frac{4n}{33}$$

$$n = 4.02 \cong 5$$

Our codesign outperforms our purely software implementation for $n >= 5$. In our 3 x 3 matrix multiplication test, our purely software implementation slightly outperforms our codesign.

Secondly, we will examine the data communication part of our codesign. Our codesign also requires time that our purely software implementation does not: PCI bus time to transfer data between the ARC-PCI board and the PC. In our codesign, there are $3n^2$ PCI bus data transfers for an $n \times n$ matrix multiplication. $2n^2$ of these transfers are writes (data from the PC to the ARC-PCI board), and n^2 of these transfers are reads (data from the ARC-PCI board to the PC). A write takes at least 9 PCI cycles, and a read takes at least 8 PCI cycles [4]. Therefore, the total number of data communication cycles for our codesign is

$$f(n) = (2 * 9)n^2 + (1 * 8)n^2 = 26n^2$$

Adding the number of data communication cycles to the number of arithmetic computation cycles for our codesign, we now have

$$f(n) = 26n^2 + 4n, \text{ which is } O(n^2)$$

The following shows the breakeven point in the total processing time for our two implementations.

ISBN: 978-988-19252-3-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Breakeven Point for Total Processing Time

$$\frac{2n^3 - n^2}{233} = \frac{26n^2 + 4n}{33}$$

$$n = 92.4 \approx 93$$

After factoring in the data communication overhead, our codesign outperforms our purely software implementation for $n >= 93$. This explains why our purely software implementation is much faster than our codesign for $n = 3$. Figure 2 shows the performance comparison of our two implementations.

Performance Comparison

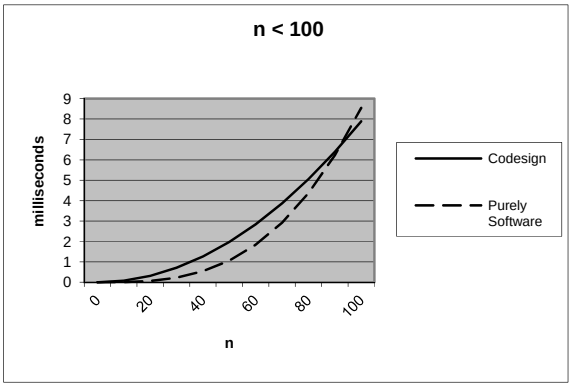


Fig. 2. Performance comparison of codesign vs. purely software for n<100.

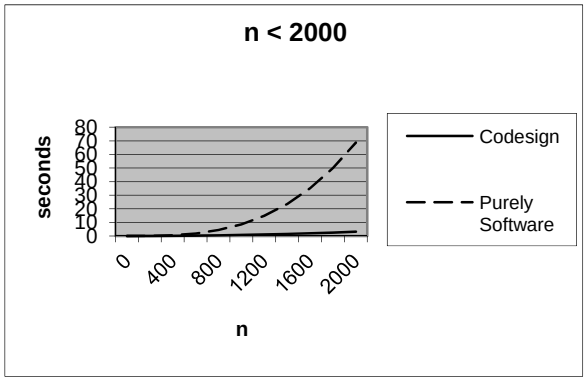


Fig. 3. Performance comparison of codesign vs. purely software for n<2000.

A significant observation in Figure 3 is that for $n = 2000$, our codesign takes 3.2 seconds to perform the matrix multiplication, compared to 68.7 seconds for our purely software implementation. The processing times in the graphs of this figure do not include system bus time, because this time is approximately equal in both of the implementations. These times are also estimates because they do not consider caching, branch prediction, pipelining, etc.

It is important to observe the computer architecture speed relationship for future considerations. As the CPU speed increases over time, the peripheral bus speed must also increase in order for our codesign to maintain significant speedup over our purely software implementation. In the future, the system and bus speeds in computers should naturally grow along with the CPU speed to achieve overall system performance gain.

IV. Related Work

Comparison of our codesign to existing parallel matrix multiplication implementations on multi-processor systems shows favorable performance results for us. A BMR-Strassen algorithm on a 64 processor system has an implementation time of 25 seconds for $n = 2000$ [5], compared to 3.2 seconds for $n = 2000$ with our codesign. Strassen and Winograd algorithms on a 4 processor system have an execution time of 12 seconds for $n = 1200$ [6], compared to 1.1 seconds for $n = 1200$ with our codesign. Figure 4 shows the performance comparison of these parallel implementations.

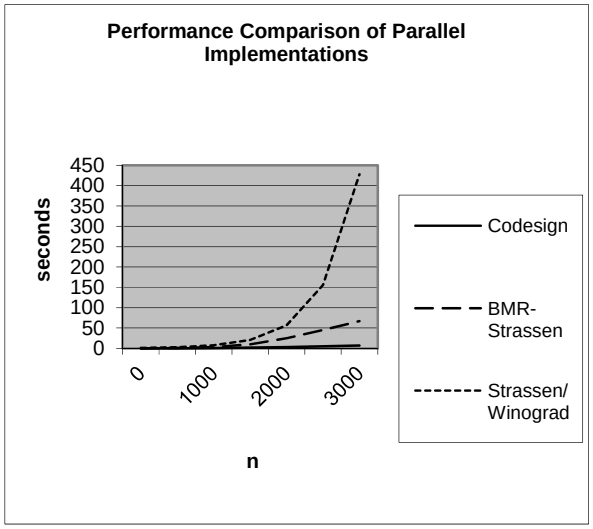


Fig. 4. Performance comparison of codesign with BMR-Strassen and Strassen/Winograd.

V. Conclusion

We have shown that a working codesign for matrix multiplication can be implemented with a PC and a PCI-interfaced FPGA board. Our codesign for n x n matrix multiplication outperforms our purely software implementation for $n >= 93$. Our performance results are favorable to existing parallel matrix multiplication implementations on multi-processor systems.

ISBN: 978-988-19252-3-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

ACKNOWLEDGMENT

We would especially like to thank SVSU Foundation for providing the support for this work and the Altera ARC-PCI software used in the study.

REFERENCES

[1] Golub, Gene, and Charles Van Loan, eds. *Matrix Computations*. Baltimore: Johns Hopkins University Press, 1996.

[2] Thomas, Donald E., and Jay K. Adams, eds. *A Model and Methodology for Hardware-Software Codesign*. IEEE Design & Test of Computers, 10(3) 1993: 6–15.

[3] Altera Corporation, San Jose, California. *The Altera Reconfigurable Computer with PCI interface (ARC-PCI)*. This reconfigurable computing platform is targeted towards researchers who want to investigate the benefits of reconfigurable computing; in other words, to improve the performance of computing systems by using applications to adapt computing hardware. February 1998.

[4] Bishop, William D. *Configurable Computing for Mainstream Software Applications*. Ph.D. Thesis, Parallel and Distributed Systems (PADS) research group, Department of Electrical & Computer Engineering, University of Waterloo, Ontario, Canada, February 2003.

[5] Luo, Qingshan and John B. Drake, eds. *A Scalable Parallel Strassen's Matrix Multiplication Algorithm for Distributed-Memory Computers*. Proceedings of the 1995 ACM Symposium on Applied Computing, Nashville, Tennessee, USA. New York: ACM Press, 1995: 221–226. ISBN: 0-89791-658-1.

[6] Chatterjee, Siddhartha, and Alvin R. Lebeck, eds. *Recursive Array Layouts and Fast Parallel Matrix Multiplication*. Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures, Saint Malo, France, 1999. New York: ACM Press, 1999: 222–231. ISBN: 1-58113-124-0.