ACSIJ Advances in Computer Science: an International Journal, Vol. 3, Issue 2, No.8 , March 2014
ISSN : 2322-5157
www.ACSIJ.org

ACSIJ
WWW.ACSIJ.ORG

Geometric Transformations via Matrix Multiplications Using Hardware/Software Co-design

Tai-Chi Lee
Department of Computer Science and Information Systems
Saginaw Valley State University
University Center, MI 48710
e-mail: *lee@svsu.edu*

Abstract

The standard methods of transformations of a geometric object in n-dimensional space are often expressed in the form of an n x n matrix multiplication by the an n x 1column vector, where the n x n matrix and the vector represent the transformation and the point in the homogeneous coordinate system respectively. This enable us to represent a series of transformations in terms of a single composite transformation in the resulting product matrix of each sequent transformation through the matrix maultiplications, where each individual matrix may be a translation, rotation, or scaling, or the cobinations of all the above. Therefore the matrix multiplications play an important role in such operation. In this paper, we first study the computational complexicity of matrix multiplications. Then we employ the hardware/software codesign on the matrix multiplications during their intensive computationally processes. In our codesign, we exploit the highly parallel nature of matrix multiplications, which cannot be exploited in our purely software implementation[4]. The hardware part of our codesign system is responsible for performing the arithmetic operations. This includes the matrix multiplier and adder, which perform concurrent multiplication and addition operations of matrix multiplication. Our matrix multiplier and adder are modeled in VHDL and runs on an ARC-PCI FPGA board [1].

Key Words: VHDL, FPGA, Multiplier, Transformation, Codesign, Product matrix, Composite trannsformation.

1. Introduction

Matrix multiplication plays an important role in applications such as geometric transformation, bipartite graph determination (non-existence of odd cycles), Economics (Leontief input-output model), power-invariant transformations (power systems), and genetics modeling (Markov chains). Therefore the computational complexity of Matrix Multiplications deserve some attentions.

Consider the following n × n matrix multiplication:

Given two n × n matrices, **A** and **B**, where

$$\mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & \dots & . & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} \quad \mathbf{B} = \begin{vmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ . & . & \dots & \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{vmatrix}$$

By the definition, the product matrix C is given as:

$$\mathbf{C} = \begin{vmatrix} c_{11} & c_{11} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ . & . & \dots & . \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{vmatrix}$$

where $c_{ij} = a_{i1} b_{1j} + a_{i2} b_{2j} + \dots + a_{in} b_{nj}$, $1 \le i, j \le n$

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.

2. Complexity of Matrix Multiplications

The C language code for n × n matrix multiplication may be given as follows:

```
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] = aij;
            b[i][j] = bij;
        }
    }

    // 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];
        }
     }
    }
}
```

The 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 [9].

In this purely software implementation, an n × 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 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 for n = 2,

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \times \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix}$$

Block 1

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$

Block 2

$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

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

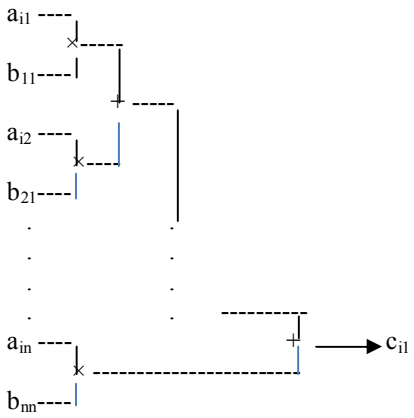
For an n × n matrix multiplication, each sequential block (see ith Block below) is composed of one parallel multiplication and (n-1) addition cycle, so 1+(n-1) arithmetic computation cycles are required for each block . And an additional cycle is required to clock data through the matrix multiplier. So a total of (n+1) clock cycles are required for each block. Therefore, the total number of clock cycles for such partitioning for an 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.

Block i

$$c_{i1} = a_{i1}b_{11} + a_{i2}b_{21} + \ldots + a_{in}b_{n1}$$
$$c_{i2} = a_{i1}b_{12} + a_{i2}b_{22} + \ldots + a_{in}b_{n2}$$
$$\vdots$$
$$c_{in} = a_{i1}b_{1n} + a_{i2}b_{2n} + \ldots + a_{in}b_{nn}$$

The multiplier's operations resulted in lst entry c_{i1} 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.

2.1 Test Results and Analysis for 3 × 3

We implemented an unsigned, 4-bit, 3 × 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 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

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

implies $n = 4.02 \cong 5$

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

Secondly, we 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 x 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

$f(n) = 26n^2 + 4n$, which is $O(n^2)$

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

Breakeven Point for Total Processing Time

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

implies $n = 92.4 \approx 93$

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 1 shows the performance comparison of our two implementations.

2.2 Performance Comparison

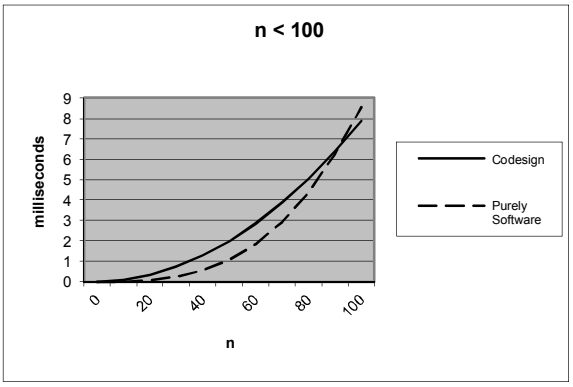


Fig. 1. Performance comparison of codesign vs. purely software for n<100.

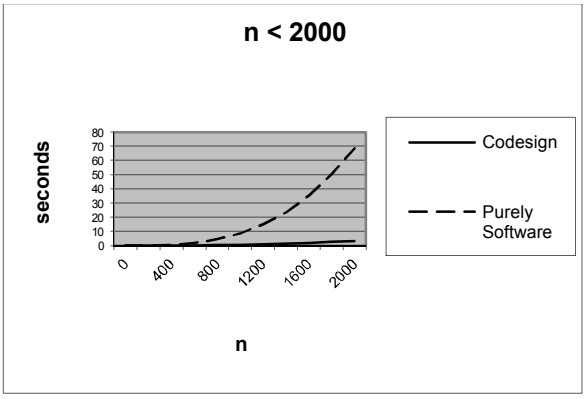


Fig. 2. Performance comparison of codesign vs. purely software for *n<2000*.

A significant observation in Figure 2 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.

3. Hardware Implementation

The hardware part of our codesign system is responsible for performing the arithmetic operations [3]. 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 [5]. 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 NT device driver to communicate with the board. Figure 3 shows our codesign system interaction.

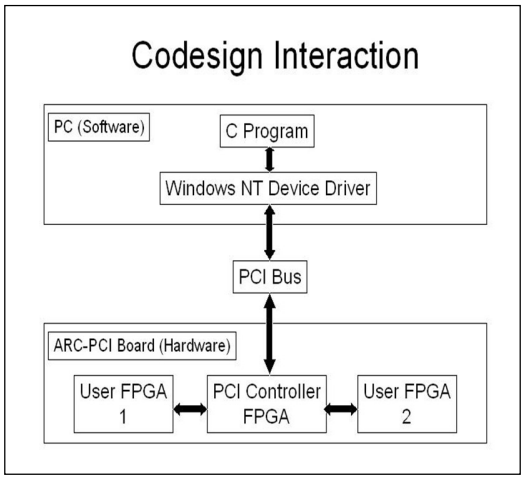


Figure 3. Layout of Codesign Scheme.

In this section, we consider a 4 × 4 matrix multiplication on our proposed SMSBS(n,m,b) (Shared Memory Split Bus System). See the figure below:

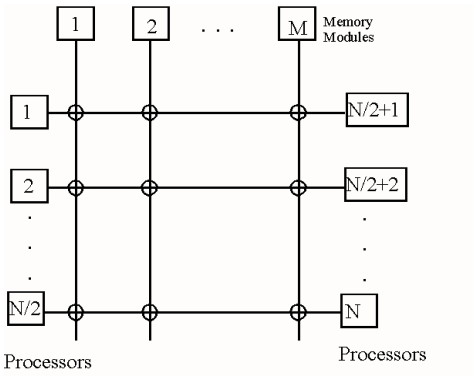


Figure 4.

The multiplication of two matrices is done on a machine whose architecture is shown above, where n, m, b are numbers of processors, memory modules, and buses

respectively. The efficiency of memory access can be found in [6].

In this example, the data is distributed to the memory modules to expedite a well bus-partitioning for buses as the processors requests these memory modules. With this partition, the SBS offers a favorable case for the bandwidth. Suppose, we have a matrix a 4 × 4 matrix A to be multiplied by a 4 × 4 matrix B in which A and B are given as:

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad B=\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

The matrix multiplication $\boldsymbol{C = A * B}$ can be performed on a SMSBS with n=m=4 using the following algorithm:

Algorithm

Step 1: P_1 Read a_{11} ...a_{14} from M_1 and
copy to P_2
P_3 Read a_{21} ...a_{24} from M_2 and
copy to P_4
P_5 Read a_{31} ...a_{34} from M_3 and
copy to P_6
P_7 Read a_{41} ...a_{44} from M_4 and
copy to P_8

Step 2: P_1 Read b_{11} ...b_{41} from M_1 and
copy to P_5
P_2 Read b_{12} ...b_{42} from M_2 and
copy to P_6
P_3 Read b_{13} ...b_{43} from M_3 and
copy to P_7
P_4 Read b_{14} ...b_{44} from M_4 and
copy to P_8

Step 3:
P_5 Read b_{14} ...b_{44} from M_4 and
copy to P_1
P_6 Read b_{13} ...b_{43} from M_3 and
copy to P_2
P_7 Read b_{12} ...b_{42} from M_2 and
copy to P_3
P_8 Read b_{11} ...b_{41} from M_1 and
copy to P_4

Step 4: (P_1, P_2, P_3, P_4) and (P_5, P_6, P_7, P_8) perform concurrent multiplication and addition of the partial products.

Step 5: P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8 store the resulting partial sums in M_1, M_2, M_3, M_4

End of Algorithm

The algorithm was implemented by using ModSim II [7], an object-oriented programming language. For the various cases in terms of the number of PE's, the matrix size q, and k, the number of columns read from the second matrix, the results we obtained are shown below.

# of PEs	matrix size q	k	# of steps	total time units
8	4	2	48	56
16	4	4	40	40
16	8	2	160	176
32	8	4	128	112
64	8	8	112	80
32	16	2	576	608
64	16	4	448	352
128	16	8	384	224
256	16	16	352	160

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 Figure 4.

4. Geometric Transformation—Mathematical Background

A geometric transformation is a function that takes a point (or vector) and maps that point (or vector) into another point (or vector). Using homogeneous coordinates, we can work with the representations of points and vectors in such that a geometric transformation can always be written in terms of the two representations, **u** and **v**, as a matrix multiplication:

v= Au, where **A** is a square matrix

For example, in 3D homogeneous coordinates [2], **A** is an 4 × 4 matrix of the form:

$$\mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{vmatrix},$$

$\mathbf{u} = [\ u_x, u_y, u_z, 1\ ]^T$, $\mathbf{v} = [\ v_x, v_y, v_z, 1\ ]^T$.

In particular, for a translation of a point with a displacements, d_x, d_y, d_z with respect to the origin, **A** takes the form:

$$\mathbf{A} = \begin{vmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

For a rotation of a point about the z-axis by an angle θ, **A** takes the form:

$$\mathbf{A} = \begin{vmatrix} \cos\theta & -\sin\theta & 0 & 1 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

And for scaling with a fixed point at the origin and the scaring factors, s_x, s_y, s_z we have **A** of the form:

$$\mathbf{A} = \begin{vmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

However, in general it depends on the nature of transformations the entries in the matrix **A** can be more complex expressions, which often increases the overhead of computations. Therefore, to speed up the matrix multiplications it requires an efficient algorithm that not only exploits the parallelisms of the computations it must also employs a well designed hardware approach. This is where our hardware/software co-design comes into play.

For example, assume for each i, the n × n matrix $\mathbf{A_i}$ represents some transformation. Then for a sequence of transformations, $\mathbf{A_1}$, $\mathbf{A_2}$,… $\mathbf{A_k}$, we form a composite transformation **C** [8], which by definition is a product matrix of $\mathbf{A_1}, \mathbf{A_2}, \ldots \mathbf{A_k}$. That is,

$$\mathbf{C} = \mathbf{A_1} \times \mathbf{A_2} \times \ldots \times \mathbf{A_k}.$$

Hence, arriving a single composite transformation by multiplying a sequence of transformation matrices can be more efficiently carried out by our hardware/software co-design using FPGA-based computing platform as described above.

5. 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.

6. References

[1] 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.

[2] Angel, Edward, Interactive Computer Graphics- A Top-Down Approach Using OpenGL, by Pearson Addison Wesley, 2005.

[3] 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.

[4] 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.

[5] Lee, Tai-Chi, *Building* An FPGA-Based Computing Platform, Proceedings of the 2012 International Conference on Frontier in Education: Computer Science & Computer Engineering, pp 522-527, July 16-19, 2012, Las Vegas, NV.

[6] 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.

[7] MODSIM II – The Language for Object-Oriented Programming Tutorial, CACI Product Company.

[8] Mortenson, E. Michael, Geometric Transformations for 3D Modeling, 3rd edition 2007 by Industrial Press, ISBN 978-0-8311-3338-2

[9] 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.