A GRAPH COMPUTATION ON FPGA-BASED PLATFORM

Tai-Chi Lee

Department of Computer Science and Information Systems
Saginaw Valley State University
7400 Bay Road, University Center, MI 48710, USA

Abstract: In this paper, we first exploit the nature of parallel computations in the matrix multiplications, where the computational complexity of purely software approach vs. hardware/software code-sign are compared. Then the significance of entries of powers of adjacency matrix for a given graph is presented, which led to the determination of a bipartite graph. The calculations for the powers of an adjacency matrix are computational intensive. Even with a high speed computer, the computations of powers of a matrix in general can be a long process and time consuming. So a desirable hardware implementation of matrix multiplication using FPGA-Based computing platform is proposed, see [5]. 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, Our matrix multiplier is modeled in VHDL and runs on an ARC-PCI FPGA board, see [1]. The purpose of the software part of our co-design system is to provide I/O to the hardware, see [3], [6]. 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 co-design and purely software implementation, as well as the performance comparison of existing parallel implementations.

AMS Subject Classification: 68W10, 68U10

Key Words: graph, VHDL, FPGA, co-design, multiplier, cycle

Received: December 26, 2013

© 2014 Academic Publications, Ltd.
url: www.acadpubl.eu

Figure 1: $V1 = \{A, C\}$, $V2 = \{B, D\}$.

1. Introduction

For quite some time, the problem of determining a bipartite graph has interested many theorists as well as practitioners. Although a lot of research has been done, much of the effort was made to the proof of its characterization. Only recently more powerful machines and software become available, people have begun to utilize computational methods for proving or disproving the unsolved problems. Even so, due to the complexity of algorithms and the limitation of computing power, some attempts of such approaches succeeded and the others failed. In this paper, we developed an algorithm to perform multiplications of adjacency matrices for a given graph. The multiplication is performed on a machine which provides the capability of parallel computations and therefore, reduces the number of steps of computations as required in the serial computations. This leads to prove the existence of an odd cycle – closed path of odd length in a given graph. Thereby, it determines if a graph is bipartite.

A graph G is bipartite if it is possible to partition its vertices into two sets $V1$ and $V2$ such that every edge of G joins a vertex in $V1$ to a vertex in $V2$, see [2]. This class of graphs is of particularly important in the study of graph theory and is related to the well known coloring problem. One of its characterization is that a nontrivial graph is bipartite if and only if it contains no odd cycles, see [2]. Our proposed algorithm is to find odd cycles of a given graph, thus determine whether or not the given graph is a bipartite. A bipartite graph of order 4 is shown in Figure 1.

It can be shown that for each integer $k \geq 1$ the entry a_{ij} of an adjacency matrix A^k is the number of paths of length k . Therefore, in particular a_{ii} represents the number of cycles of length k . Thus, it suffices to say that if the diagonal entries of all odd powers of adjacency matrices are zero then the the

given graph is bipartite. Since the longest path between two vertices of a graph of order n is $n-1$ and the k th power of its adjacency matrix shows the existence of all paths of length k . Therefore, by computing all odd powers of the adjacency matrix up to $n-1$ and examining all the diagonal entries we can determine the existence of odd cycles.

Others applications that require large, fast matrix multiplication are economics (Leontief input-output model), power-invariant transformations (power systems), cryptography, and genetics modeling (Markov chains), and fractal image compressions.

2. The Computational Complexity of Matrix Multiplications

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 $n \times n$ matrix multiplication, see [4], [5].

Given two $n \times n$ matrices, A and B , where

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

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

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix},$$

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

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. The C language code for $n \times n$ 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 co-design, which is the simultaneous design of hardware and software subsystems [2].

In this purely software implementation, an $n \times n$ 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 a $n \times n$ matrix multiplication, each sequential block (see i th Block below) is composed of one parallel multiplication cycle and $(n-1)$ addition cycles, 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 a matrix multiplication is $f(n) = n(n+1) = n^2 + n$, which is $O(n^2)$, a slight improvement of one order over the purely software approach.

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

$$c_{i1} = a_{i1}b_{11} + a_{i2}b_{21} + \cdots + a_{in}b_{n1},$$

$$c_{i2} = a_{i1}b_{12} + a_{i2}b_{22} + \cdots + a_{in}b_{n2},$$

$$\vdots$$

$$c_{in} = a_{i1}b_{1n} + a_{i2}b_{2n} + \cdots + a_{in}b_{nn}.$$

The multiplier's operations resulted in 1st 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. The Test Results and Analysis

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

First, we will examine the arithmetic computation part of our co-design. 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 co-design 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 co-design arithmetic computation rate of 33 MHz . We would like to find n for the break-even point in arithmetic computation time for our co-design and purely software implementations. Our purely software arithmetic computation time is $(2n^3 - n^2 \text{ cycle seconds}) / (233,000,000 \text{ cycles})$.

Our co-design arithmetic computation time is $(4n \text{ cycle seconds}) / (33,000,000 \text{ cycles})$. 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 co-design outperforms the purely software implementation for $n >= 5$. In our 3×3 matrix multiplication test, our purely software implementation slightly outperforms our co-design. Secondly, we will examine the data communication part of our co-design. Our co-design 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 co-design, 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 co-design is

$$f(n) = (2 \times 9)n^2 + (1 \times 8)n^2 = 26n^2.$$

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

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

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 co-design outperforms our purely software implementation for $n \geq 93$. This explains why our purely software implementation is much faster than our co-design for $n = 3$. Figure 2 shows the performance comparison of our two implementations.

Performance Comparison

Figure 2: Performance comparison of co-design vs. purely software for $n < 100$.

Figure 3: Performance comparison of co-design vs. purely software for $n < 2000$.

A significant observation in Figure 3 is that for $n = 2000$, our co-design 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 co-design 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.

2.2. Related Work

Comparison of our co-design 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 $n = 2000$, see [5], compared to 3.2 seconds for $n = 2000$ with our co-design. Strassen and Winograd algorithms on a 4 processor system have an execution time of 12 seconds for $n = 1200$, see [6], compared to 1.1 seconds for $n = 1200$ with our co-design. Figure 4 shows the performance comparison of these parallel implementations.

3. Implementation

The hardware part of our co-design 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 co-design 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 1 shows our co-design system interaction [10].

In this section, we consider a 4×4 matrix multiplication on our proposed SMSBS(n, m, b) (Shared Memory Split Bus System) [8]. See the figure below:

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

Figure 4: Performance comparison of co-design with BMR-Strassen and Strassen/Winograd.

buses respectively. The efficiency of memory access can be found in [3]. 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 SMSBS 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:

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

The matrix multiplication $C = AB$ can be performed on a SMSBS with $n = m = 4$ using the following algorithm:

Algorithm (Note: The following P_i 's and M_i 's denote the processors and shared memories respectively)

Step 1: P_1 Read $a_{11} \dots a_{14}$ from M_1 and

copy to P_2

P_3 Read $a_{21} \dots a_{24}$ from M_2 and

Figure 5: Layout of co-design scheme.

Figure 6

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_l 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 simulated by using ModSim II - object-oriented programming language [9], and will be implemented on a FPGA-Based computing platform [5]. For the various cases, in terms of the number of processors P_i 's, the matrix size q , and k , the number of columns read from the second matrix, the results we obtained are shown in Table 1.

4. Conclusion

We have shown that a working co-design for matrix multiplication can be implemented with a PC and a PCI-interfaced FPGA board. Our co-design for $n \times n$ matrix multiplication outperforms our purely software implementation for $n \geq 93$. Our performance results are favorable to existing parallel matrix

# of Pi's	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

Table 1

multiplication implementations on multi-processor systems with distributed-memory, see [10].

4. Acknowledgments

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] 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] Behzad, Mehdi, Chartrand, Gary, and Lesniak-Foster, Linda, *Graphs & Digraphs*, Wadsworth International Group (1979), ISBN: 0-534-98021-X.
- [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, Lebeck, Alvin R, *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] Golub, Gene, Van Loan, Charles, *Matrix Computations*, Baltimore: Johns Hopkins University Press (1996).
- [6] Lee, Tai-Chi, Building an FPGA-based computing platform, In: *Proceedings of the 2012 International Conference on Frontier in Education: Computer Science & Computer Engineering*, pp. 522-527, July 16-19, 2012, Las Vegas, NV.
- [7] Lee, Tai-Chi, Using software emulation in FPGAs to improve co-design development, In: *Proceedings of the Fourth International Conference of Applied Mathematics and Computing*, pp. 167-168, Aug. 12-18, 2007, Bulgaria.
- [8] Luo, Qingshan and John B. Drake, (Ed-s.), *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.
- [9] *MODSIM II – The Language for Object-Oriented Programming Tutorial*, CACI Product Company.
- [10] Thomas, Donald E., and Adams, Jay K., A model and methodology for hardware-software co-design, *IEEE Design & Test of Computers*, **10**, No. 3 (1993), 6-15.