# Performance of PGA (Programmable Graph Architecture) for Matrix Multiplications

Muling Peng, Sreekanth Ramani, K. Wendy Tang
*Department of Electrical and Computer Engineering*
*State University of NY at Stony Brook*
muling@ece.sunysb.edu, sramani@ic.sunysb.edu, wtang@ece.sunysb.edu

A. Yavuz Oruç
*Department of Electrical and Computer Engineering*
*University of Maryland at College Park*
*yavuz@eng.umd.edu*

**Abstract**

*Matrix multiplication is one of the primary operations in linear algebra and is useful in a wide spectrum of multimedia applications including signal and image processing. In this paper we review a novel computer architecture for matrix multiplications. This novel architecture is based on graph theory, hence the name Programmable Graph Architecture. We discuss the performance of PGA via time and space complexity and time-processor product for parallel implementation. We further compare the PGA performance with Cannon's algorithm and show that the PGA architecture outperforms the existing method for matrix with any dimension.*

**Keyword: computer architecture, matrix multiplication, Cayley graphs**

## 1. Introduction

Recently, we introduce the novel algorithms and architectures for matrix operations on configurable devices [1,2]. This new family of architecture is based on Cayley Graphs, hence the name Programmable Graph Architecture (PGA). The motivation to propose this new model of computation is to explore the potential performance advantages of transforming matrix operations into spatial graph routing problems. This is conceptually similar to the transformation techniques used in [5-9] but we rely on the isomorphism between linear matrix groups and Cayley graphs rather than those between arithmetic groups and permutation groups.

Our model works through the use of modular $p$ arithmetic; that is a matrix is iteratively broken down into modular $p$ matrices. Operations in the original matrix domain are then translated into operations in the modular $p$ matrix domain. It is in this domain of modular $p$ matrices that we construct a Cayley graph. To connect between the original matrix domain and the modular $p$ matrix domain, we need five tables to store one-to-one mapping information.

Through computer simulations, we observe that the time complexity of our PGA algorithm is of O ($P$), where $P$ is the average path length of a Cayley graph, instead of the dimension of the original matrix. And the number of processors needed is $\left(M/N\right)^3 \cdot e^2 \cdot N_{SDm}$, where $M$ is the dimension of the original matrix, $N$ is the dimension of the base matrix for the corresponding Cayley graph, $e$ is an integer, and $N_{SDm}$ is the number of source-destination multiplication pairs. The time-processor product for the PGA algorithm will be $\Omega\left(P \cdot e^2 \cdot N_{SDm} \cdot \left(M/N\right)^3\right)$. Standard serial matrix multiplication computation will require a time complexity of $O\left(M^3\right)$, while the most updated parallel computing algorithms can be performed in $O\left(M\right)$ operations with $M^2$ processors, or in $O\left(\log M\right)$ operations with $M^3$ processors [11,12]. The time-processor product for these parallel methods, however, is all nearly $\Omega\left(M^3\right)$, which is worse than our PGA approach. In this paper, we also show that the time-processor product for the PGA approach is less than that of the standard parallel computation for any matrix dimension with proper setting of PGA coefficients.

This paper is organized as follows: in section 2, we review the existing matrix multiplication algorithms. Section 3 provides a description of the PGA

Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR'06)
0-7695-2613-6/06 $20.00 © 2006 IEEE

algorithm. Section 4 discusses the performance of PGA and Section 5 is the conclusion of the paper.

## 2. Overview of existing matrix multiplication algorithms

Matrix multiplication is one of the primary operations in linear algebra and is useful in a wide spectrum of multimedia applications including signal and image processing. In recent years, a number of algorithms have been developed to reduce the computational complexity from $O(M^3)$. Initial algorithms were sequential, with Strassen(1969) showing that the complexity is reduced to $O(M^{2.807})$. [14] Further improvements reduced the order to $O(M^{2.376})$ was achieved by Coppersmith and Winograd(1990) [15]. To speed up computation, parallel matrix multiplication algorithms have been developed which mostly involve decomposition of the matrices and parallelizing the standard algorithm. A common complexity measure is therefore, a time-processor product ($\Omega$) rather than just the order of time complexity. There have been a number of approaches; these include: the systolic algorithm using systolic arrays, Cannon's algorithm [16], PUMMA (Parallel Universal Matrix Multiplication)[17], SUMMA (Scalable Universal Matrix Multiplication)[18], DIMMA (Distribution Independent Matrix Multiplication)[19] and SRUMMA (Shared and Remote Memory based Universal Matrix Multiplication algorithm)[20]. Apart from the computational requirements (number of multiplications and additions) there are also storage requirements which must be considered. In the case of parallel algorithms, the communication overhead between processors also comes into play.

In Strassen's algorithm, the $M \times M$ matrices are divided into $4$ $M/2 \times M/2$ matrices and the result is obtained by recursively multiplying these. The limitation is that the matrix size is a power of $2$ which is overcome in the Winograd algorihm.

In parallel processor algorithms, the memory used is either distributed or shared or a combination of both. Distributed memory performance measurements are more complex as the partitioning of matrices across the machines affects the parallelism and the communication overhead. The time complexity in the case of Cannon's algorithm was $O(M)$ with $M \times M$ processors. [12,16]

In our proposed matrix multiplication via PGA (Programmable Graph Architecture), we take a novel approach in turning the original matrix multiplication operation into the fine grain computation of graph routing. In [1-4], we introduced a computational model that facilitates the transformations of matrix multiplication into physical layers of processors. Such a computation model is based on the isomorphism between linear matrix groups and Cayley graphs. This isomorphism allows us to map matrix multiplication directly into hardware without performing any row-column products. The potential benefits of this mapping from the matrix algebra domain to Cayley graphs in processor design, especially for multimedia applications, can be quite huge. In the following section, we provide a brief overview of our matrix multiplication via PGA algorithm.

## 3. Matrix Multiplication via PGA algorithm

Our PGA algorithm makes use of five pre-stored tables and memory mapping to deal with the matrix multiplication. The strength of the PGA algorithm comes from transforming the matrix multiplication into serial scalar additions through graph routing. The theoretical parts of the algorithm are mentioned in previous papers [1-4]. Here we briefly review the algorithm by the following examples:

**Step 1:** *Decompose the operand matrices:* Given two $M \times M$ matrices, $\mathbf{A}$, $\mathbf{B}$, where $M$ is a power of $2$, iteratively decompose $\mathbf{A} \times \mathbf{B}$ into products of $2 \times 2$ matrices. For example, let $A$ and $B$ be $4 \times 4$ matrices. Then, we can write

$$\mathbf{A} \times \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 88 & 127 & 11 & 12 \\ 111 & 5 & 15 & 16 \end{pmatrix} \times \begin{pmatrix} 126 & 54 & 3 & 4 \\ 37 & 26 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{A_1B_1} + \mathbf{A_2B_3} & \mathbf{A_1B_2} + \mathbf{A_2B_4} \\ \mathbf{A_3B_1} + \mathbf{A_4B_3} & \mathbf{A_3B_2} + \mathbf{A_4B_4} \end{pmatrix} \quad \text{Eq.(1)}$$

where

$$\mathbf{A_1} = \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix}, \mathbf{A_2} = \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix}, \mathbf{A_3} = \begin{pmatrix} 88 & 127 \\ 111 & 5 \end{pmatrix}, \mathbf{A_4} = \begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix}$$

$$\mathbf{B_1} = \begin{pmatrix} 126 & 54 \\ 37 & 26 \end{pmatrix}, \mathbf{B_2} = \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix}, \mathbf{B_3} = \begin{pmatrix} 9 & 10 \\ 13 & 14 \end{pmatrix}, \mathbf{B_4} = \begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix}$$

Thus, the product of two $M \times M$ matrices ($M$ is a power of $2$), can be computed through a sequence of multiplications of $2 \times 2$ matrices. In the following steps, we illustrate how $2 \times 2$ matrices can be computed in parallel through Cayley graph routing.

**Step 2:** *Choose a prime number p and factor the $2 \times 2$ matrices into $2 \times 2$ matrices with entries mod p:* For example, for $p = 5$, consider the following $2 \times 2$

Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR'06)
0-7695-2613-6/06 $20.00 © 2006 IEEE

matrix multiplication $\mathbf{A_3}$ and $\mathbf{B_1}$, choosing from Eq. (1):

$$\mathbf{A}_3 \times \mathbf{B}_1 = \begin{pmatrix} 88 & 127 \\ 111 & 5 \end{pmatrix} \times \begin{pmatrix} 126 & 54 \\ 37 & 26 \end{pmatrix}$$

Then,

$$\mathbf{A}_3 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}_{\mathbf{A}_3,1} p^3 + \begin{bmatrix} 3 & 0 \\ 4 & 0 \end{bmatrix}_{\mathbf{A}_3,2} p^2 + \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}_{\mathbf{A}_3,3} p + \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix}_{\mathbf{A}_3,4}$$

$$\mathbf{B}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}_{\mathbf{B}_1,1} p^3 + \begin{bmatrix} 0 & 2 \\ 1 & 1 \end{bmatrix}_{\mathbf{B}_1,2} p^2 + \begin{bmatrix} 0 & 0 \\ 2 & 0 \end{bmatrix}_{\mathbf{B}_1,3} p + \begin{bmatrix} 1 & 4 \\ 2 & 1 \end{bmatrix}_{\mathbf{B}_1,4}$$

$$\mathbf{A}_3 \times \mathbf{B}_1 = \cdots + \mathbf{A}_{3,3}\mathbf{B}_{1,2} p^3 + \cdots + \mathbf{A}_{3,4}\mathbf{B}_{1,2} p^2 + \cdots + \mathbf{A}_{3,4}\mathbf{B}_{1,4} \qquad \text{Eq.(2)}$$

Eq. (2) reveals that the matrix product $\mathbf{A_3} \times \mathbf{B_1}$ can be computed by performing a set of $2 \times 2$ mod $p$ matrix multiplications. As in this example, only these three terms listed are non-singular matrices and the rest of these mod $p$ matrices are singular. For singular matrices, the Cayley graph approach cannot be used for matrix multiplications because these matrices are not in *GL(2,p)*, and therefore not in the corresponding Cayley graph. We solve this problem by expressing a singular matrix as a sum or difference of two nonsingular matrices. More specifically, there are five possible patterns of *2×2* singular matrices. They are:

$$\begin{pmatrix} 0 & 0 \\ x & y \end{pmatrix}, \begin{pmatrix} x & y \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} x & 0 \\ y & 0 \end{pmatrix}, \begin{pmatrix} 0 & x \\ 0 & y \end{pmatrix}, \begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix}$$

These singular *2x2* matrices can be expressed as the sum or difference of two non-singular ***Zp*** matrices:

$$\begin{pmatrix} 0 & 0 \\ x & y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ x & y+1 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ if } y \neq p-1; \text{ or}$$

$$\begin{pmatrix} 0 & 1 \\ x+1 & y \end{pmatrix} - \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \text{ if } y = p-1, x \neq p-1; \text{ or}$$

$$(p-1)\left\{ \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right\}, \text{ if } x = y = p-1$$

The matrices $\begin{pmatrix} x & y \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} x & 0 \\ y & 0 \end{pmatrix}, \begin{pmatrix} 0 & x \\ 0 & y \end{pmatrix}$ can be similarly expressed. The last pattern of singular matrices can be expressed as:

$$\begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix} = \begin{pmatrix} x_1 & 0 \\ 0 & x_4 \end{pmatrix} + \begin{pmatrix} 0 & x_2 \\ x_3 & 0 \end{pmatrix}$$

The elements of any rows or columns in $\begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix}$ could not be both zeroes.

Thus, all the singular matrices in Eq. (2) can be represented as the sum of two no-singular matrices. Eq. (2) will become multiple non-singular matrices pair multiplication. For example:

$$\mathbf{A}_{1,1}\mathbf{B}_{1,1} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \left\{ \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right\} \times \left\{ \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right\}$$

**Step 3:** *Multiply 2×2 mod p matrices via graph routing in a Cayley Graph:* In this step, we multiply the non-singular sub-matrices in Eq. (2). These multiplications are computed using an integer representation of a Cayley graph constructed over the GL(*2,p*) group of matrices [1-4]. For *p=5*, there are 480 number of nodes in the graph. These nodes are labeled as node ❶ to node ❹❼❾. Every node of the graph corresponds to a *2×2 mod p* non-singular matrix. Multiplication of any two *2×2 mod p* non-singular matrices corresponds to routing between a source and destination node of the graph, hence we call this process the source-destination multiplication, SDm. For example, node ❹❶❶×❸❺❺ is a source destination multiplication, SDm. The mapping of these matrices and the corresponding integer node label in the graph is stored as Table 1. This table consists of $n = p \times (p-1) \times (p^2-1)$ entries to index the $n$ vertices with the $n$ matrices, where $n$ is the order of the matrix group.

Step 3.1: *Transform 2×2 mod p non-singular matrices into integers:* Every non-singular matrix has a corresponding integer node label which is stored in Table 1. As in the example of $\mathbf{A}_{3,3}\mathbf{B}_{1,2}$ in Eq. (2):

$$\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}_{\mathbf{A}_{3,3}} \underset{\text{matrix}}{\overset{\text{integer}}{\rightleftharpoons}} ❹❶❶, \begin{bmatrix} 0 & 2 \\ 1 & 1 \end{bmatrix}_{\mathbf{B}_{1,2}} \underset{\text{matrix}}{\overset{\text{integer}}{\rightleftharpoons}} ❸❺❺$$

Thus,

$$\begin{aligned} \left(\mathbf{A}_{3,3} \times \mathbf{B}_{1,2}\right)_{\text{matrix domain}} &= \left(❹❶❶ \times ❸❺❺\right)_{\text{Integer Domain}} \\ &= \left\{ ❹❶❶ \times \left[\text{the routing path of } ❸❺❺\right] \right\}_{\text{Part I}} \\ &\quad - \left\{ ❹❶❶ \times \left[\text{the quotient matrix of } ❸❺❺\right] \times p \right\}_{\text{Part II}} \end{aligned}$$

Eq (3)

In the above example, it is clear that the computation of $\mathbf{A}_{3,3}\mathbf{B}_{1,2}$ is divided into 2 parts. The computation of these two parts is described in more details in Steps 3.2 and Step3.3:

Step 3.2: *Multiply source node with the routing path of the destination node (Part I):* Table 2 consists of *nD* entries where *D* is the diameter of the graph. It keeps track of the paths from node ❶ to all the other nodes in the graph. A path is identified as a sequence of generators as defined by Cayley graphs. More detailed description of these generators can be found in [1-4] and is not repeated here. By looking up Table 2, the path of the destination node can be identified. In our example, the routing path of ❸❺❺ can be identified from the 355th entry of Table 2, and in this

Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR'06)
0-7695-2613-6/06 $20.00 © 2006 IEEE

case, the sequence is $\mathbf{G_1}, \mathbf{G_6^{-1}}, \mathbf{G_7^{-1}}$, where $\mathbf{G_1}, \mathbf{G_6^{-1}}, \mathbf{G_7^{-1}}$ are generators.

Once we have identified the sequence of generators as a path to the destination node, we need to multiply the source node with this set of generators. This is the core of the algorithm that transforms a vector operation into a scalar operation. In the integer domain of the Cayley graph, multiplying a source node with generators corresponds to a series of modular *n* additions. Each of the generators corresponds to an integer and these integers are stored in Table 3 [1-2]. The size of this table is of O($\delta$) where $\delta$ is the degree of the underlying graph. In this example, $\delta=14$.

Since the underlying Cayley graph is constructed over a group of modular *p* matrices, namely GL(*N,p*), the quotient of the multiplication product need to be identified from a table. Table 4 of our database stores the quotient matrices for the product of the matrix with each of the generators. Its size is therefore, $n\delta N^2$ where *n* is the number of nodes, $\delta$ is the degree of the Cayley graph, and *N* is the dimension of the base matrix.

In our example, the source node is ❹❶❶, from Table 3, generator $\mathbf{G_1}$ corresponds to integer 270. From the 401th entry of Table 4, the quotient matrix is $\begin{bmatrix}0 & 0\\ 1 & 0\end{bmatrix}$. Hence $❹❶❶\times\mathbf{G_1}$ is

$$\mathbf{Q}_{❹❶❶}\times p+\left(401+270_{\mathbf{G_1}}\right)_{\bmod 480}=\begin{bmatrix}0 & 0\\ 5 & 0\end{bmatrix}+❶❾❶$$

Using the method outlined here, the entire computation of Part I can be summarized as follow:

$$\begin{aligned}
\text{Part I} &= ❹❶❶\times\mathbf{G_1}\times\mathbf{G_6^{-1}}\times\mathbf{G_7^{-1}}\\
&=\left\{\mathbf{Q}_{❹❶❶}\times p+\left(401+270_{\mathbf{G_1}}\right)_{\bmod 480}\right\}\times\mathbf{G_6^{-1}}\times\mathbf{G_7^{-1}}\\
&=\left\{\begin{bmatrix}0 & 0\\ 5 & 0\end{bmatrix}+❶❾❶\right\}\times\mathbf{G_6^{-1}}\times\mathbf{G_7^{-1}}\\
&=\left\{\begin{bmatrix}0 & 0\\ 5 & 0\end{bmatrix}+❶❾❶\right\}\times\mathbf{G_6^{-1}}\times\mathbf{G_7^{-1}}\\
&=\left\{\begin{bmatrix}0 & 0\\ 5 & 15\end{bmatrix}+\mathbf{Q}_{❹❶❶}\times p+\left(191+237_{\mathbf{G_6^{-1}}}\right)_{\bmod 480}\right\}\times\mathbf{G_7^{-1}}\\
&=\left\{\begin{bmatrix}0 & 0\\ 5 & 15\end{bmatrix}+\begin{bmatrix}1 & 2\\ 1 & 1\end{bmatrix}\times p+❶❷❸\right\}\times\mathbf{G_7^{-1}}=\left\{\begin{bmatrix}0 & 0\\ 5 & 15\end{bmatrix}+❷❽❼\times p+❹❷❽\right\}\times\mathbf{G_7^{-1}}\\
&=\begin{bmatrix}0 & 0\\ 30 & 55\end{bmatrix}+\mathbf{Q}_{❹❶❶}\times p^2+\left(287+172_{\mathbf{G_7^{-1}}}\right)_{\bmod 480}\times p+\mathbf{Q}_{❹❶❶,\mathbf{G_7^{-1}}}\times p+\left(428-3_{\mathbf{G_7^{-1}}}\right)_{\bmod 480}\\
&=\begin{bmatrix}0 & 0\\ 30 & 55\end{bmatrix}+\begin{bmatrix}0 & 1\\ 0 & 1\end{bmatrix}\times p^2+❹❺❾\times p+\begin{bmatrix}0 & 0\\ 1 & 3\end{bmatrix}\times p+❹❷❺\\
&=\begin{bmatrix}0 & 0\\ 30 & 55\end{bmatrix}+\begin{bmatrix}0 & 25\\ 0 & 25\end{bmatrix}+\begin{bmatrix}20 & 15\\ 10 & 0\end{bmatrix}+\begin{bmatrix}0 & 0\\ 5 & 15\end{bmatrix}+\begin{bmatrix}0 & 2\\ 1 & 0\end{bmatrix}=\begin{bmatrix}20 & 42\\ 46 & 95\end{bmatrix}
\end{aligned}$$

Step 3.3: *Fetch the quotient matrix of the destination node (Part II):* Table 5 is used to store the quotient matrix of the product of the set of generator matrices from node ❶ to each node in the graph. Its entries are therefore, *n*, where *n* is the number of nodes. The quotient matrix of the destination node is retrieved from the entry of its node number in Table 5. If entries in the quotient matrix are greater than *p*, we have to repeat step 2 to factor the matrix into entries mod *p*. In our example, the equation of Part II becomes:

$$\begin{aligned}
\text{Part II} &= ❹❶❶\times\begin{bmatrix}4 & 8\\ 1 & 2\end{bmatrix}\times p=❹❶❶\times\left\{\begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\times p+\begin{bmatrix}4 & 3\\ 1 & 2\end{bmatrix}\right\}\times p\\
&=\begin{bmatrix}0 & 1\\ 0 & 2\end{bmatrix}\times p^2+\begin{bmatrix}4 & 3\\ 9 & 8\end{bmatrix}\times p=\begin{bmatrix}20 & 40\\ 45 & 90\end{bmatrix}
\end{aligned}$$

Step 3.4: *Sum up Part I and Part II:* Once the answers of Part I and Part II are available, the answer of two matrix multiplications is Part I subtracted by Part II. In our example:

$$\begin{bmatrix}1 & 0\\ 2 & 1\end{bmatrix}_{\mathbf{A_{3,3}}}\times\begin{bmatrix}0 & 2\\ 1 & 1\end{bmatrix}_{\mathbf{B_{1,2}}}=\begin{bmatrix}20 & 42\\ 46 & 95\end{bmatrix}-\begin{bmatrix}20 & 40\\ 45 & 90\end{bmatrix}=\begin{bmatrix}0 & 2\\ 1 & 5\end{bmatrix}$$

## 4. Performance of PGA

Assume we need to perform a matrix multiplication for two *M×M* grey level matrices. Using our PGA algorithm, we need to choose a base matrix with dimension *N×N* to form a Cayley graph.

Here are $\left(M/N\right)^3$ numbers of base matrix multiplications needed to compute. From our example in Eq. (1), the dimension of the original matrix is *M=4*, and the dimension of the base matrix is *N=2*, the number of base matrix multiplication is $\left(4/2\right)^3 = 8$. If we adopt larger number of *N*, there are less numbers of base matrix multiplications. A small change of *N* will, however, dramatically increase the size of the number of nodes in the Cayley graph. Our experiments primarily use *N=2* as the base matrix dimension.

For performance, we consider three parameters: *time complexity, space consumption*, and *time-processor product* to measure the performance of PGA algorithm. In the following sections, we described the time and space complexity of our PGA approach. We also compare the time-processor product measure of PGA with the latest parallel matrix multiplication algorithms. We found that, indeed, our PGA approach has better performance with the proper setting of PGA coefficients.

### 4. 1 Time complexity measure

As indicated in Eq (3), in our PGA approach, the computation of the multiplication of two *N×N* non-

Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR'06)
0-7695-2613-6/06 $20.00 © 2006 IEEE

singular matrices is divided into two parts. Part I is concerned with the multiplication of the source node with the generators of the graphs; and Part II is the multiplication of the source node with the quotient of the destination node. The following describes the time complexity analysis for Part I and Part II.

4.1.1 Matrix multiplication between sources nodes and generators (Part I)

The routing sequence from the identity matrix to the multiplier matrix is actually the combination of generators. That's why this part can be regarded as multiplication with generators as multipliers. Once the dimension and modulus $p$ are set, the length of the routing sequence for each node is controlled by the degree of the graph. In general, the larger degree gives the smaller path length, but their relationship is non-linear.

Now, let's take a closer look at the process of multiplication with generators as multipliers. The amounts of processes needed depend on the number of generators in the routing sequence. Each process contains one addition for a multiplicand node plus GCR constant, and $N^2$ additions for the quotients. For our experiments, it is $2^2=4$ additions here. If the quotient is singular or has entries greater than modulus $p$, there are extra expanded terms for the next multiplication process. For example, if singular quotient is encountered, there are two more additions for multiplicand nodes plus GCR constants, and two more $N^2$ additions for quotients. Even though we have these expanded and add-on terms, the computing time of this part can still grow only with the path lengths of the graph. Figure 1 illustrates the relationship between the average number of additions and average path lengths for different graphs. The average number of additions is computed by averaging the total number of additions for all possible $n^2$ multiplication pairs. As expected, the average number of additions is proportional to the average path length.

Figure 1 contains four different Cayley graphs with $p$ ranges from *5* to *13*. Each point in the graph corresponding to a particular value of $p$ with a specific degree as indicated on the graph. From the graph, we observe that the average number of additions is linear with the average path length, i.e., average number of addition is of O($P$), where $P$ is the average path length.

4.1.2 Matrix Multiplication between source node and quotient of destination node (Part II)

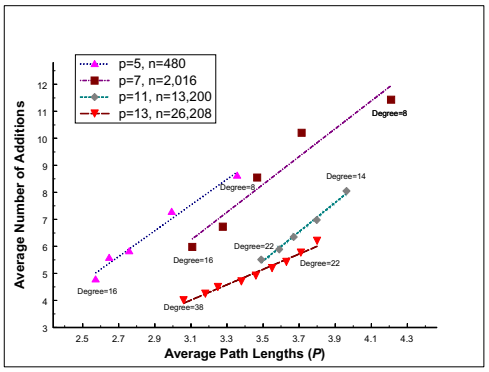


**Figure 1: Time complexity of PGA has O(P) relation.**

In this part, the computation is divided into several source-destination pairs. If the computation is achieved in serial, the time complexity should be the product of the number of SDm pairs with the average path length of the graph. If the computation is achieved in parallel, the number of processors needed corresponds to the number of source-destination pairs. The number of source-destination pairs is affected by the size of the graph. In general, a large network requires the graph having a larger degree so that the average path length is small which corresponds to a smaller quotient in Table 5. Such smaller quotients in general guarantees a smaller number of source-destination pairs. In our experiments by computing all possible $n^2$ multiplication pairs, we have the following results: for p=13, $N_{SDm}$=4.6 at $\delta$=32; $N_{SDm}$=6.3 at $\delta$=34; $N_{SDm}$=3.9 at $\delta$=36; $N_{SDm}$=3.5 at $\delta$=38; where $N_{SDm}$ is the number of SDm pairs, and $\delta$ is the degree of the graph.

4.2 Space consumption measure

As described in Section 3, there are five tables. As a summary, these five tables are:

Table 1: This table of size $nN^2$ consists of $n$ entries to pair the $n$ nodes with the $n$ matrices.

Table 2: This table consists of $nD$ entries where $D$ is the diameter of the graph to keep track of the paths from node 0 to all the other nodes in the graph.

Table 3: This table stores GCR constants with size $q\delta$.

Table 4: This table of size $n\delta N^2$ is used to store the quotient matrices for the product of the matrix corresponding to each node with each of the generators.

Table 5: This table of size $nN^2$ is used to store the quotient matrix of the product of generator matrices from node ❶ to each node in the graph.

The total size of these tables is, therefore,

Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR'06)
0-7695-2613-6/06 $20.00 © 2006 IEEE

$$\text{Tables Sizes} = n \times \left[ N^2 \times (2 + \delta) + D \right] + q\delta$$

Where $n$ is the number of nodes in the graph, $N$ is the dimension of the matrix, $\delta$ is the degree of the graph, $D$ is the diameter of the graph, and $q$ is the class of the graph.

The following diagram illustrates the space consumption versus the number of nodes in the graph with various degrees. As expected, the space requirement grows linearly with the size of the graphs.

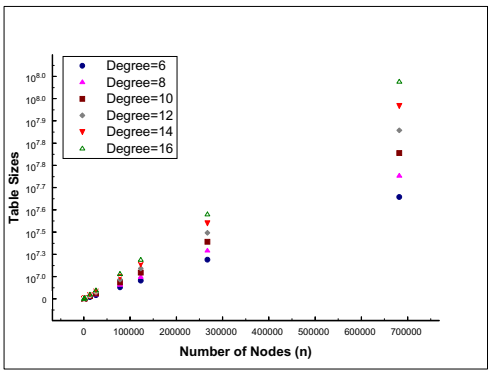


**Figure 2: The table sizes versus the number of nodes in the graph with various degrees.**

4.3 Time-processor product

For parallel implementation, time-processor product is a common performance measure. From the beginning of the section, we know there are $\left(M/N\right)^3$ numbers of base matrix multiplications, and in Section 4.1.2, we further confirm that each base matrix multiplication needs the number of SDms. Nevertheless, there is still another issue we need take into consideration, and that is matrix entries greater than modulus $p$. Since, it is a matrix with grey level entries, the power of modulus $p$ will determine the number of expanded terms. The number of expanded terms is equal to the smallest integer $e$ such that $p^e > 128$. For instance, for *p=11, e=3*, and for *p=13, e=2*. To combine these three factors, the processor units are required for parallel computation as the following equation:

$$\text{Number of Processor} = \left(M/N\right)^3 \times e^2 \times N_{SDm}$$

where $N_{SDm}$ is the number of source-destination multiplication.

To combine the time and processor factors, the time-processor product for the PGA algorithm will be:

$$\Omega\left( P \cdot e^2 \cdot N_{SDm} \cdot \left(M/N\right)^3 \right)$$

**The comparison between PGA and the existing algorithms**

In comparing the performance of PGA with other existing algorithms, we consider the time-processor product for PGA and other algorithms. Existing serial or parallel matrix multiplication is generally measured only with the number of multiplication, but our PGA algorithm does not involve multiplication but only contains integer additions. Since we consider the grey level matrix, we can assume seven scalar additions is equal to one scalar multiplication. Therefore, the time complexity of our algorithm can be O(*P/7*). We computed the time-processor product for four sets of PGA graphs: $N$=2, $p$=13, and $\delta$=32 to 38, These computations involves O(*P/7*) where *P*=3.*38, 3.25, 3.18* and *3.06* respectively for $\delta$=32 to 38. They are implemented on $7.8M^3$ , $10.1M^3$ , $6.2M^3$, $5.4M^3$ processors, respectively. As expected, the time-processor product for our PGA algorithm grows with matrix dimension. Figure 3 shows the comparison between our PGA algorithm and Cannon's algorithm. The plot shows the time-processor product for our PGA algorithm for $p$=13, $\delta$=34 and 38. Those for $\delta$=32 and $\delta$=38 are similar to $\delta$=34 and $\delta$=38 respectively, and are therefore omitted in the figure. As described in Section 2, Cannon's algorithm implements on $M^2$ processors with O($M$) time. The time-processor product is $\Omega(M^3)$.

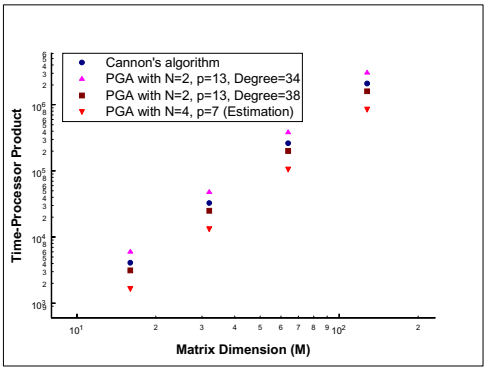


**Figure 3: The performance comparison of our PGA algorithms v.s Cannon's algorithm.**

As shown in Figure 3, our PGA algorithm for $p$=13, $\delta$ =38 outperforms Cannon's method for any matrix dimension, but not for $\delta$ =34. It tells the coefficient setting of our PGA algorithm is crucial to govern the performance for matrix multiplication. Hence, we have an estimation result for $N$=4, $p$=7 on the figure to show that the different setting can significantly

influence the performance. For example, for matrix with dimension $2^7$, the time-processor product for our PGA for $N$=2, $p$=13, $\delta$ =38 is $2^{20.6}$; for $\delta$=34 is $2^{21.5}$; and for $N$=4, $p$=7 is $2^{19.7}$ whereas that of Cannon is $2^{21}$. Thus, our PGA algorithm can be tuned by choosing different coefficients to tower over any other existing algorithms.

## 5. Conclusion

In this paper, we reviewed our PGA matrix multiplication algorithms and discuss its performance in time complexity, space complexity, and the time-processor product. We also compare the time-processor product of our PGA algorithm with that of Cannon's algorithm. Indeed, our PGA algorithm outperforms existing methods with the proper setting of the PGA coefficients.

## References


[1] K. Wendy Tang, A. Yavuz Oruç, "Programmable Graph Architecture (PGAs) For Matrix Operations", *2003 Conference on Information Science and Systems*, The John Hopkins University, March 2003.

[2] M. Peng, K. Wendy Tang and A. Yavuz Oruc, "Matrix Multiplication via Programmable Graph Architectures", Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems, MIT Cambridge, MA, November 9-11, 2004.

[3] B.W. Arden and K.W. Tang, ``Representations and Routing of Cayley Graphs", *IEEE Transactions on Communications,* 39(11):1533-1537, November, 1991.

[4] K.W. Tang, *Dense and Symmetric Interconnection Networks*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, 1991, University of Rochester, Rochester, NY.

[5] D.V. Chudnovsky and G.V. Chudnovsky and M.M. Denneau, Regular Graphs with Small Diameter as Models for Interconnection Networks, IBM Research Division, RC 13484(60281), February, 1988.

[6] A. Yavuz Oruç, Vinod Peris, and M. Y. Oruç, System and method for performing fast algebraic operations on a permutation network, US Patent No. 5,270,956, Dec. 14, 1993.

[7] A. Yavuz Oruç, Vinod Peris, and M. Yaman Oruç, Parallel Modular Arithmetic on a Permutation Network, Proc. of 1991 Int. Conference on Parallel Processing, August, 1991, Vol 1., pp. 706-707.

[8] M. B. Lin and A. Yavuz Oruç, Constant-time Inner Product and Matrix Computations on Permutation Network Processors, IEEE Transactions on Computers, 1994, pp. 1429-1434.

[9] Liang Fang and A. Yavuz Oruç, Matrix Computations on Permutation Networks. Proc. of 29th Annual Conference on Information Sciences and Systems, Mar 1995, pp. 804-809.

[10] J. J. Rotman, The Theory of Groups. 2nd Ed. Allyn and Bacon, Pub. 1Boston, 1973, p 156.

[11] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, T. Turnbull, "Implementation of Strassen's Algorithm for Matrix Multiplication", IEEE Transactions on Computers, 1996.

[12] K. Li, Y. Pan, S. Q. Zheng, "Fast and Processor Efficient Parallel Matrix Multiplication Algorithms on a Linear Array with a Reconfigurable Pipelined Bus System", IEEE Transactions on Parallel and Distributed Systems, Vol 9. No. 8, 1998.

[13] W. Y. Tu, H. K. Chau, Muling Peng, Sangjin Hong, Alex Doboli, K.Wendy Tang, A. Yavuz Oruç, "Design Study of (2x2) Processing Core Architecture for Cayley Graph Matrix Multiplcations", 2004.

[14] Volker Strassen, "Gaussian elimination is not optimal". *Numerische Mathematik*, 14(3):354-356, 1969.

[15] D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Processing", In *Proceeding of the Nineteenth Annual ACM Symposium on Theory of Computing*, Page 1-6, 1987.

[16] L.E. Cannon, "A Cellular Computer to Implement the Kalman Filter Algorithm," PhD thesis, Montana State Univ., 1969.

[17] J. Choi, J. J. Dongarra, and D. W. Walker, PUMMA: Parallel Universal Matrix Multiplication Algorithms, Concurrency: Practice and Experience, Vol. 6, No. 7, pages 543-570, October 1994.

[18] Robert van de Geijn and Jerrell Watts, ``SUMMA: Scalable Universal Matrix Multiplication Algorithm," Department of Computer Sciences, The Unversity of Texas, TR-95-13, April 1995. Also: LAPACK Working Note #96 , May 1995.

[19] Hyuk-Jae Lee, J.A.B. Fortes, "Toward Data Distribution Independent Parallel Matrix Multiplication", Proc. Int. Parallel Processing Sym., p436 --440, Apr 1995.

[20] M Krishnan, J Nieplocha, "SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems", *IEEE Parallel and Distributed Processing Symposium*, 2004.



Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR'06)

0-7695-2613-6/06 $20.00 © 2006 IEEE