Parallel Matrix Multiplication Design for Monocular SLAM

Mohd. Yamani Idris¹, Noorzaily Mohamed Noor³
Emran Mohd. Tamil⁴, Zaidi Razak⁵
¹,³,⁴,⁵Faculty of Computer Science & Information Technology,
University of Malaya,
Kuala Lumpur, Malaysia
yamani@um.edu.my, zaily@um.edu.my,
emran@um.edu.my, zaidi@um.edu.my

Abstract— As software profiling is conducted to determine which section of program demand high processing computation in monocular SLAM inverse depth estimation, matrix multiplication is identified to be one of the most time consuming process. The processing is more demanding when the number of features inserted to the image is increased. For that reason, this paper proposes a parallel matrix multiplier design which could accelerate the execution time. In this design, Field Programmable Gate Array (FPGA) technology which allows parallel design to be implemented is presented. The design manipulates existing classical matrix multiplication algorithm into an architecture that would enable data to be processed concurrently.

Keywords-FPGA, matrix multiplication, depth parametrization, Simultaneous Localization and Mapping (SLAM)

I. INTRODUCTION

Simultaneous Localization and Mapping (SLAM) is a study which involves an autonomous vehicle or robot to estimate its position with respect to objects in its environment and maps the object that it has encountered. In the early SLAM research, laser and sonar sensor have been a popular choice chosen by many researchers. However, due to the capability to acquire large amount of information, research on vision SLAM has emerged drastically. This emergence can be seen in various researches on vision SLAM feature detection techniques as reviewed in M.Y.I Idris et al [1]. The recent development is also motivated by the fact that vision sensors are compact, accurate, noninvasive, well understood, cheap and ubiquitous [2]. In vision SLAM area, monocular SLAM has been widely studied and presented. A monocular SLAM is a SLAM studies which focus on using a single camera to estimate the robot location and how the robot maps the environment. Although the motivation to use vision sensor to determine location and environment mapping is attractive, the computational cost has been one of the greatest drawback in SLAM implementation. The paper by R. Mungia et al [3] addressed the same problem when they noted that the process in real time is a major challenge in monocular SLAM since the computational cost grows rapidly when the number of features in the system state increases.

II. MONOCULAR SLAM AND ITS COMPUTATION

Feature detection and feature estimation are the main areas in SLAM research. In this paper, estimation or specifically depth estimation of a monocular SLAM is studied. In monocular SLAM, depth can be estimated by using a feature parallax where a parallax is a measured angle of an object or captured rays viewed from two different lines of sight. Davison et al [4,5] has successfully applied real-time monocular camera tracking, however their approach only make use of features that were close to the camera. The reason behind the inability to estimate distant feature depth is because of the straightforward Euclidean XYZ feature parameterization adoption. Euclidean XYZ feature parameterization for low parallax feature is not well represented by the Gaussian distribution implied in the EKF. Therefore, inverse depth representation is introduced to allow Gaussian distribution to cover uncertainty for both low and high parallax features. According to Civera et al [6], the ony drawback of the inverse depth scheme is the computational issue of increase state vector size where Inverse depth scheme needs six state vector parameters (equation 2) while Euclidean XYZ coding (equation 1) needs only three. To increase computational efficiency, Civera et al [6] restrict inverse depth to cases where XYZ encoding exhibits nonlinearity according to the linearity index. However, the estimation starts at infinity and “coming in” as the camera moves distant enough to determine adequate parallax. This means that high inverse depth computation is still in used until enough parallax is achieved.

\[
x_i = (X_i, Y_i, Z_i)^T \tag{1}
\]

\[
y_i = (x_i, y_i, z_i, \theta_i, \sigma_i, \rho_i)^T \tag{2}
\]

where:

- \(x_i, y_i, z_i\) = camera optical center
- \(\theta_i\) = azimuth
- \(\sigma_i\) = elevation
- \(\rho_i\) = inverse depth
III. INVERSE DEPTH PARAMETERIZATION AND MATRIX MULTIPLICATION

Another approach to improve computational speed is by introducing a co-processor that could aid in processing highly used functions. A form of dynamic program analysis to examine programs behavior called program profiling can be employed to analyze program’s behavior and detect which sections of a program demand high processing computation. As profiling is done (Figure 1), feature initialization as in equation 4 and equation 5 has shown significant time spent when number of features added to the image is high. The time spent is also significant in the covariance prediction where state update equation as in equation 3 is utilized. Covariance matrix computation starts with 13x13 diagonal matrix size where three values come from camera optical center position (rW; rx,ry,rz), four values from quaternion defining orientation (qWC; qRx,qy,qz), three values from linear velocity (vW; vx,vy,vz) and three from angular velocity (ωW; ωx, ωy, ωz) relative to world frame W and camera frame C respectively. When features are inserted, another six values (as in y2i in equation 2) will be added into the full state vector (equation 6). The insertion of more features will increase the size of the full state vector which also increases the matrix size for matrix multiplication. For instance, if one hundred features are included, a covariance matrix with size 100x100 will be used for the computation, (i.e. 100 (from number of features) x 6 (from y2i; 6-D vector) + 13 (i.e. from f2) = 613). Bigger matrix size will increase execution time spent by a processor.

![Figure 1: Software profiling](image)

![Classical algorithm is commonly used and it is the most important way matrices are multiplied. To multiply two matrices, the width of the first matrix must be equal the height of the second matrix (equation 7). Classical way of multiplying two matrixes is represented by equation 8. Although simple to implement, the square matrix multiplication of this method has complexity of O(n3) and suffers from poor locality where row i of A may have been force to leave from the cache by the time the inner-most loop completes. This will result on a poor performance and low efficiency [7].](image)

\[
C_{m,n} = A_{m,l} \times B_{l,n} \tag{7}
\]

\[
c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j} \tag{8}
\]

While computer add faster than they multiply, Winograd’s algorithm [8] as shown in equation 9 trade multiplications with additions. The α and β coefficients are computed only once for each row and column where only n² scalar multiplications are required. Therefore, the total number of scalar multiplications for Winograd’s algorithm is reduced from n×n×n (in classical method) to 1/2n²+n². However,
the number of additions is increase by \(1/2n^3\) from \((n-1)n^3−n^2\) in classical method.

\[
c_j = d_j - \alpha_i - \beta_j
\]

\[
d_j = \sum_{k=1}^{n/2}(a_{i,2k} + b_{2-4k,j}) \cdot (a_{i,2k-1} + b_{2k,j})
\]

\[
\alpha_i = \sum_{k=1}^{n/2}(a_{i,2k} \cdot a_{i,2k-1})
\]

\[
\beta_j = \sum_{k=1}^{n/2}(b_{2k,j} \cdot b_{2k-1,j}) \quad i,j=1,...,n
\]

(9)

Strassen’s algorithm [9] is a divide-and-conquer method which divides the given matrices into four sub-matrices and then recursively multiplying them to obtain the resultant matrix [10]. In this method, only seven multiplications and eighteen additions for each recursive call is needed. This makes the number of multiplications for an \(n\)-by-\(n\) matrix to be \(n^{\log_2(7)}\) and the number of additions is \((4.5)n^3\log_2 n\).

**Equation 10** shows the initial matrices \(A\) and \(B\) of size 2x2, partial multiplication result, S, and the resulting elements \(c_{i,j}\) used in Strassen’s algorithm.

\[
\begin{pmatrix}
    c_{11} & c_{12} \\
    c_{21} & c_{22}
\end{pmatrix}
= \begin{pmatrix}
    a_{11} & a_{12} \\
    a_{21} & a_{22}
\end{pmatrix}
\begin{pmatrix}
    b_{11} & b_{12} \\
    b_{21} & b_{22}
\end{pmatrix}
\]

\[
S_1 = (a_{11} + a_{12}) \cdot (b_{11} + b_{12})
\]

\[
S_2 = (a_{21} + a_{22}) \cdot b_{11}
\]

\[
S_3 = a_{11} \cdot (b_{12} - b_{22})
\]

\[
S_4 = a_{12} \cdot (b_{21} - b_{11})
\]

\[
S_5 = (a_{11} - a_{12}) \cdot (b_{11} + b_{12})
\]

\[
S_6 = (a_{21} - a_{22}) \cdot (b_{21} + b_{22})
\]

\[
c_{11} = S_1 + S_4 - S_5 + S_6
\]

\[
c_{12} = S_2 + S_3
\]

\[
c_{21} = S_2 + S_3
\]

\[
c_{22} = S_1 - S_4 + S_5 + S_6
\]

(10)

V. **MATRIX MULTIPLICATION ON FPGA**

Apart from algorithm improvement, the speed to calculate matrix multiplication can be improved by using Field Programmable Gate Array (FPGA). FPGA is an integrated circuit which can be reconfigured by the designer. It has been applied in many areas such as computer vision, digital signal processing and etc. The inherent advantage of FPGA comes from its internal structure which allows parallel execution. This advantage is able to allow considerable computational throughput even at low clock rates. Another significant advantage of this reconfigurable hardware is when the time intensive tasks are offloaded from software to hardware (FPGA). Besides that, FPGA is small (Actel FPGA package size is as small as 3x3mm) and only consume low power (as low as 2µW) [11]. This is suitable for monocular SLAM application where portability is essential.

Matrix Multiplication on FPGA has been studied by many researchers to improve the performance of numerous applications. Baravio I. et al [12], for instance have written a paper on different proposal to matrix multiplication based on FPGAs. They evaluated classical, Winograd and Strassen’s algorithm and proposed a new approach based on systolic arrays. A different design is proposed by Zhuo L. et al [13,14] which employ a linear architecture with small control logic. They also discussed the design tradeoffs among the area, the total storage size, the latency and the required memory bandwidth.

In this paper, yet another approach is proposed to improve matrix multiplication execution time. A classical method is manipulated into a parallel design to achieve higher processing speed. In the next section, the design of a classical matrix multiplication on FPGA is presented. Following the design, parallel design based on the classical design is presented. The parallel design uses the same concept as Strassen’s algorithm which divides the original matrix into smaller matrices and merges them at the end of the computation.

A. **Classical Matrix Multiplication on FPGA**

**Equation 7** and **equation 8** calculate matrix multiplication in a classical way. An example of the calculation is shown in **Example 1**. The design in FPGA is straightforward where a controller will feed Processing Element (PE) as in **Figure 2** with value of \(a\) (row) and \(b\) (column). The PE is a multiplier accumulator (MAC) which multiplies two numbers and adds the product to an accumulator. One PE is used to multiply and accumulate the input such in **Figure 3**. In this figure, \(t\) is denoted as the timing where \(t_{in}\) is the first processing to be computed. As can be seen, by using only one PE, 16\(n\) cycles (i.e. \(T_{in}\)) are needed to complete a 4x4 matrix multiplication. In the following section, this architecture will be improved with the introduction of seven sub processing elements to allow parallel execution of matrix multiplication.
Example 1

\[ A1 \times B1 = C1 \]

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\begin{bmatrix}
  b_{11} & b_{12} & b_{13} & b_{14} \\
  b_{21} & b_{22} & b_{23} & b_{24} \\
  b_{31} & b_{32} & b_{33} & b_{34} \\
  b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
= 
\begin{bmatrix}
  c_{11} & c_{12} & c_{13} & c_{14} \\
  c_{21} & c_{22} & c_{23} & c_{24} \\
  c_{31} & c_{32} & c_{33} & c_{34} \\
  c_{41} & c_{42} & c_{43} & c_{44}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  3 & 5 & 7 & 2 \\
  9 & 3 & 4 & 0 \\
  2 & 6 & 1 & 9 \\
  5 & 3 & 9 & 7
\end{bmatrix}
\begin{bmatrix}
  2 & 3 & 4 & 7 \\
  9 & 9 & 0 & 2 \\
  3 & 4 & 8 & 7 \\
  1 & 9 & 2 & 7
\end{bmatrix}
= 
\begin{bmatrix}
  74 & 100 & 72 & 94 \\
  57 & 70 & 68 & 97 \\
  70 & 145 & 34 & 96 \\
  71 & 141 & 106 & 153
\end{bmatrix}
\]

\[
c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} = 3(2) + 5(0) + 7(3) + 2(1) = 74
\]

\[
c_{34} = a_{31}b_{14} + a_{32}b_{24} + a_{33}b_{34} + a_{34}b_{44} = 2(7) + 6(2) + 1(7) + 9(7) = 96
\]

**Figure 2:** Classical Matrix Multiplication Block Diagram

---

\[ t_{1n} \mid t_{2n} \mid t_{3n} \mid t_{4n} \]

\[
\begin{array}{cccc}
3 & 5 & 7 & 2 \\
2 & 9 & 3 & 1 \\
3 & 5 & 7 & 2 \\
4 & 0 & 0 & 8 & 2 & 7 & 2 & 7 & 7
\end{array}
\]

---

\[ t_{5n} \mid t_{6n} \mid t_{7n} \mid t_{8n} \]

\[
\begin{array}{cccc}
9 & 3 & 4 & 0 \\
9 & 3 & 4 & 0 \\
9 & 3 & 4 & 0 \\
9 & 3 & 4 & 0 \\
7 & 2 & 7 & 7
\end{array}
\]

---

\[ t_{9n} \mid t_{10n} \mid t_{11n} \mid t_{12n} \]

\[
\begin{array}{cccc}
2 & 6 & 1 & 9 \\
2 & 9 & 3 & 1 \\
3 & 9 & 4 & 9 \\
4 & 0 & 8 & 2 & 7 & 2 & 7 & 7
\end{array}
\]

---

\[ t_{13n} \mid t_{14n} \mid t_{15n} \mid t_{16n} \]

\[
\begin{array}{cccc}
5 & 3 & 9 & 7 \\
5 & 3 & 9 & 7 \\
5 & 3 & 9 & 7 \\
5 & 3 & 9 & 7 \\
7 & 2 & 7 & 7
\end{array}
\]

---

\( n=4 \) (for this example)

**Figure 3:** Classical matrix multiplication timing cycle

---

**B. Matrix Multiplication Parallel Design**

The FPGA implementation of a classical matrix multiplication designed in the previous section has the ability to offload software processing to dedicated hardware. The offloading offers significant speed improvement over software implementation since most processing is done in register level. To further accelerate the design, parallel implementation on FPGA is considered. In a parallel design, a processing element with seven sub processing elements (sPE A-sPE G) is proposed. The proposed design is shown in **figure 4** with each sPE will process input \( a \) and \( b \) in parallel. The upper controller illustrated in the block diagram is used to enable register in each sPE while the lower controller is used to select which input line to be sent out via a multiplexer. The process starts at \( t_{1n} \) where at \( t_{1n} \) (as shown in **figure 5**), only sPE A is busy to process input \( a \) (i.e. 3,5,7,2) and \( b \) (i.e. 2,9,3,1). At the same time, register in sPE B is loaded with value \( a \) (i.e. 3,5,7,2) and sPE E is loaded with value \( b \) (i.e. 2,9,3,1). At \( t_{3n} \), three sub-processing elements (i.e. sPE A, sPE B, sPE E) start to work concurrently to produce three output values. sPE A process input \( a \) (i.e. 9,3,4,0) and \( b \) (i.e. 3,9,4,9) while sPE B and sPE E process the same input with the previous registered value. Similar process occurs at \( t_{5n} \) with five sub processing elements (i.e. sPE A, sPE B, sPE E, sPE C, sPE F) work simultaneously. Finally at \( t_{15n} \), all seven sub-processing elements start to work at the same time. Following the final process, all sixteen (one at \( t_{1n} \), three at \( t_{2n} \), five at \( t_{3n} \), and seven at \( t_{4n} \)) matrix multiplication results are obtained. This shows that in the parallel design, the computation is able to reduce the timing cycle from 16n cycle (in classical way) to only 4n cycles (**figure 5**). In this example, the multiplication is limited to 4x4 matrix size. The multiplication however can be further expanded to larger matrix by implying divide-
and-conquer technique. The breaking down of a larger matrix to simpler sub-problems will be combined at the end to solve the original problem. **Example 2** shows how divide-and-conquer algorithm can be used to solve larger matrix problems. In the example, a 16x16 matrix size is divided into 4x4 matrix size. The 4x4 matrix $An$ will multiply with 4x4 matrix $Bn$ and then the outcome will be added together to produce the final result.

**Example 2**

\[
\begin{bmatrix}
A_{1} & A_{2} \\
A_{3} & A_{4}
\end{bmatrix}
\begin{bmatrix}
B_{1} & B_{2} \\
B_{3} & B_{4}
\end{bmatrix}
= 
\begin{bmatrix}
C_{1} & C_{2} \\
C_{3} & C_{4}
\end{bmatrix}
\]

\[
\begin{bmatrix}
3 & 5 & 7 & 2 \\
9 & 3 & 4 & 0
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 8 & 7 \\
9 & 0 & 2
\end{bmatrix} = 
\begin{bmatrix}
2 & 3 & 4 & 1 \\
7 & 6 & 1
\end{bmatrix}
\]

**Figure 4:** Parallel Matrix Multiplication Block Diagram

<table>
<thead>
<tr>
<th></th>
<th>t_{1A}</th>
<th>t_{2A}</th>
<th>t_{3A}</th>
<th>t_{4A}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>7</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

**Figure 5:** Timing cycle for parallel matrix multiplication

VI. CONCLUSION

In this paper, a software profiling is conducted to determine which section of the program in monocular SLAM inverse depth estimation demand high processing computation. As profiling is done, the execution of matrix multiplications calculation is found to be one of the most time consuming process. This is more evident when the number of features (such as interest point) added to the image is increased. The increase in the inserted features also increases the matrix multiplication size. Matrix multiplication is used in many functions in inverse depth calculation. This includes feature covariance and covariance prediction computation. For that reason, a high speed co-processor design using FPGA
technology is proposed to offload software processing to a dedicated fast matrix multiplier hardware processing. The fast matrix multiplier utilizes possible parallel architecture available in the FPGA to allow each sub-processing element to work concurrently. At the same time, a divide-and-conquer algorithm is also employed in the design to permit larger matrix problem to be solved using smaller sub-problems (e.g. 4x4 matrix). For a 4x4 matrix (i.e. n=4), the parallel design is able to reduce the timing cycle from 16n cycle (in classical implementation) to 4n cycles.

ACKNOWLEDGMENT

This project is funded by Ministry of Science Technology and Innovation (MOSTI) under e-Science Fund (11-02-03-1047) and University of Malaya Research Grant (RG014-09ICT)

REFERENCES


[9] Strassen, Volker, Gaussian Elimination is not Optimal, Numerische Mathematik 13, p. 354-356, 1969


