CS480/580 Project Final Report

GPU Utilization for Complex Mechatronic Systems

Zhenishbek Zhakypov, Bahadır Pehlivan, Cem Kocagil, Can İnce

Introduction

Mechatronic systems are widely employed in our daily life such as robots and many other smart electro-mechanic systems. In order to represent such systems relevant mathematical description of the linear or non-linear dynamics and kinematics of the physical system is required which may be an uneasy task. Moreover such systems demand an appropriate linear or non-linear control algorithms in order for it to become truly smart, accurate and robust. However implementation of such mathematical expressions on a regular sequentially executed microcontroller or processor unit may result in computational overheads especially in the case of complex mechatronic systems with many degrees of operation and they have smaller workspace and complex structure that require more computational power. The main purpose of this project is to utilize GPU unit to compute necessary kinematics and control algorithms of a multiple link robot manipulator or other mechatronic systems concurrently for fast and reliable response.

Problem Formulation

Robotic manipulator

Manipulation is one of the crucial functions in manufacturing since parts should be moved or manipulated by some exertion of external actions if necessary. Manipulator mechanisms can be used for gripping, pick-place, assembly or injection purposes. The three degrees of freedom (3-DOF) manipulator robot is depicted in Figure 1. It is a parallel kinematics robot and the main advantage of such parallel mechanisms over serial robots is their more precise, faster and stiffer characteristics. The robot is compact in size with 40 mm^3 operational space volume. It has three metallic arms mounted to backlash-free DC

motors at the base and connected to triangular frame on the other end (end effector) with omnidirectional motion capability in $x - y - z$ coordinates. Parallel robots provide the advantages of high stiffness, low inertia and high speed capability at the expense of smaller workspace, complex mechanical design, difficult forward kinematics and control algorithm. The kinematics of the parallel mechanisms is more complex when compared to the serial ones and needs more computational power.

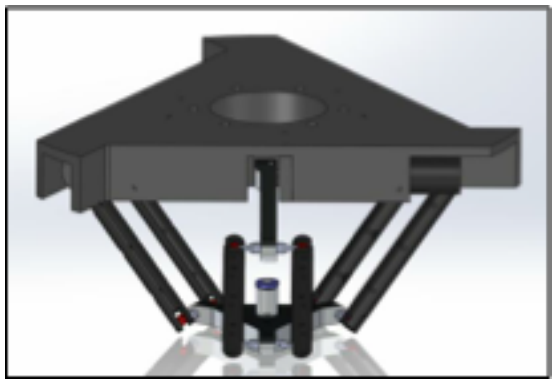

Figure 1. 3-DOF parallel kinematic robot

Robot kinematics

Kinematics of the system is a mathematical relation between the angles of each motor rotation (joint or configuration space) and end effector position in $x - y - z$ Cartesian coordinate frame (task or operation space). The theoretical forward kinematics of the robot has been derived based on geometric relations of the robot links and rotational angles.

Forward Kinematic Equation

$$D_i = -L_B^2 + L_A^2 + R^2 + 2RL_A cos\alpha_i$$

$$E_i = 2(R + L_A cos\alpha_i)cos\theta_i$$

$$F_i = 2(R + L_A cos\alpha_i)sin\theta_i = E_i tan\theta_i$$

$$G_i = 2L_A sin\alpha_i$$

$$H_1 = E_1G_2 - E_1G_3 - E_2G_1 + E_2G_3 + E_3G_1 - E_3G_2$$

$$H_2 = -E_1F_2 + E_1F_3 + E_2F_1 - E_2F_3 - E_3F_1 + E_3F_2$$

$$H_3 = -E_1D_2 + E_1D_3 + E_2D_1 - E_2D_3 - E_3D_1 + E_3D_2$$

$$H_4 = F_1D_2 - F_1D_3 - F_2D_1 + F_2D_3 + F_3D_1 - F_3D_2$$

$$H_5 = -F_1G_2 + F_1G_3 + F_2G_1 - F_2G_3 - F_3G_1 + F_3G_2$$

$$L = \frac{H_5^2 + H_1^2}{H_2^2} + 1$$

$$M = 2\frac{H_5H_4 + H_1H_3}{H_2^2} - \frac{H_5E_1 + H_1F_1}{H_2} - G_1$$

$$Q = \frac{H_4^2 + H_3^2}{H_2^2} - \frac{H_4E_1 + H_3F_1}{H_2} + D_1$$

$$R_A - R_B = R$$

$$x = z\frac{H_5}{H_2} + \frac{H_4}{H_2}$$

$$z = \frac{-M \pm \sqrt{M^2 - 4LQ}}{2L}$$

$$y = z\frac{H_1}{H_2} + \frac{H_3}{H_2}$$

R_A, R_B, L_A *and* L_B *are known*

Inverse Kinematic Equation

$R_A - R_B = R$ $\qquad R_A, R_B, L_A \text{ and } L_B \text{ are known}$

$$Q_i = 2x\cos\theta_i + 2y\sin\theta_i$$

$$S = \frac{1}{L_A}(-x^2 - y^2 - z^2 + L_B^2 - L_A^2 - R^2)$$

$$\tan\frac{\alpha_i}{2} = \frac{2z \pm \sqrt{4z^2 - 4R^2 - S^2 + Q_i^2\left(1 - \frac{R^2}{L_A^2}\right) + Q_i(-2\frac{RS}{L_A} - 4R)}}{-2R - S - Q_i(\frac{R}{L_A} - 1)}$$

Methodology

Above mentioned expressions can be implemented in a GPU architecture and computed concurrently. The block diagram of the overall system is depicted in Figure 2 below.

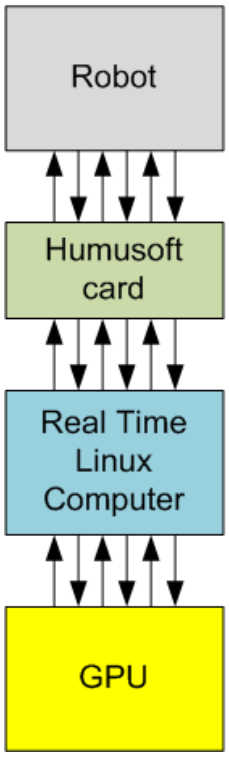

Figure 2. System block diagram

In this project only the kinematic model is implemented on GPU for the sake of proof of concept.

Inverse Kinematics

Serial Code

```
x_2= x^2 ,   y_2=y^2,    z_2=z^2
Q1=2x * cos(theta1) + 2y * sin(theta1)
Q2=2x * cos(theta2) + 2y * sin(theta2)
Q3=2x * cos(theta3) + 2y * sin(theta3)
Q1_2 = Q1^2,  Q2_2 = Q2^2,   Q3_2 = Q3^2
T1=2z+sqrt(4*z_2+4*R_2...+Q1*..)/(...-Q1*...)
T2=2z+sqrt(4*z_2+4*R_2...+Q2*..)/(...-Q2*...)
T3=2z+sqrt(4*z_2+4*R_2...+Q3*..)/(...-Q3*...)
Alpha1 = -2*atan(T1)
Alpha2 = -2*atan(T2)
Alpha3 = -2*atan(T3)
```

If we look at serial and parallel functions we can understand how parallelism provides us efficiency. Above we have pseudo code of serial version of Inverse Kinematics Function. Normally it takes x, y and z coordinates as inputs and gives alpha1, alpha2, alpha3 as outputs by using mathematic formulas. In order to gain these alphas it first gain 3 Q values and 3 T values. Also the function does taking square operation 6 times. When we look at the parallel one - which is below- it operates with 3 threads. Thanks to these 3 threads every thread just find 1 Q and 1 T values. Also we take square just two times and we finally gain 1 alpha from every thread. So this parallelization provide us approximately 3 times faster algorithm to calculate alphas.

Parallel Code

```
const int index = blockIdx.x * blockDim.x +
threadIdx.x;
pows[INDEX] = coords[INDEX]*coords[INDEX]
Q = 2 * coords[0] * cos(THETA * INDEX) + 2 * coords[1]
* sin(THETA * INDEX);
Q_2 = pow(Q,2);
T_sqrt = sqrt(4 * pows[2] +... + Q_2 * …..+ Q * …...);
T = (2 * coords[2] + T_sqrt) / (….........-Q *
….......);
alphas[INDEX] = -2 * atan(T);
```

Forward Kinematics

Serial Code

```
D1=....+..*cos(alpha1)
E1=..+..cos(alpha1)+ cos(theta1)
F1=  ..+..cos(alpha1)+ sin(theta1)
G1=...*sin(alpha1)
D2=....+..*cos(alpha2)
E2=..+..cos(alpha2)+ cos(theta2)
F2=  ..+..cos(alpha2)+ sin(theta2)
G2=...*sin(alpha2)
D3=...+..*cos(alpha3)
E3=..+..cos(alpha3)+ cos(theta3)
F3=..+..cos(alpha3)+ sin(theta3)
G3=...*sin(alpha3)
```

Above Function is the serial version of Forward Kinematics Function. We make 3 calculations for each D, E, F and G. But when we look at the parallel one below, we just write one line for each letter. So again we have 3 times faster algorithm.

Parallel Code

```
index = blockIdx.x * blockDim.x + threadIdx.x;;
const float alpha = alphas[index];
defg[0][index] =.......+ ……..... * cos(alpha);
defg[1][index] = 2 * (R + LA * cos(alpha)) * cos(THETA
* index);
defg[2][index] = 2 * (R + LA * cos(alpha)) * sin(THETA
* index);
defg[3][index] = …...* sin(alpha);
```

Computational Results

Implementation of the kinematic equations with conventional sequential and parallel programs is performed. When we look at the difference between serial and parallel codes we see that the parallel code is faster than the serial one. However, because of some issues about GPU hardware it doesn't provide expected time efficiency. But when used with more than one robot, efficiency in time grows exponentially.
Note that the tests were done in Linux with an emulator, not on an actual GPU hardware and the computation time was 8 times faster than CPU.

After the computation, forward and inverse kinematics results are as following;

```
Input coords: 4.4239, 2.60742, -54.1189
*     inverse   kinematics   result   angles:   0.100001,
0.200001, 0.3

Input alphas: 0.1, 0.2, 0.3
*    Forward kinematics result coords: 4.4239, 2.60742,
-54.1189
```

Conclusion & Future Work

Kinematics of a parallel robot is presented and the results shows that parallel programming on GPU outperforms the serial one in terms of computation time when utilized for big number of multi DOF robotic systems such as factories. Optimization in parallel programming, further utilization of GPU for more complex tasks such as controller, integration and trajectory generation is considered as future work.