IMPLEMENTATION AND EVALUATION OF A POTHOLE DETECTION SYSTEM ON TI C6678 DIGITAL SIGNAL PROCESSOR

Chee Kin Chan, Yuan Gao, Zhen Zhang, and Naim Dahnoun

University of Bristol,
Merchant Venturers' Building, Woodland Road, Clifton, BS8 1UB
phone: + (0044) 7889909608, email: yuan.gao@bristol.ac.uk
web: http://www.bristol.ac.uk/

ABSTRACT

This paper describes the implementation of a previously proposed pothole detection system on Texas Instruments C6678 Digital Signal Processor (DSP). The system employs disparity maps as input followed by surface fitting and Connected Component Labelling (CCL) for pothole detection. Optimisations of the matrix calculation in the system algorithm have been made DSP friendly. Memory management has also been applied for high resolution input images as well as compiler optimisation for efficient code pipelining. The paper also discusses the use of parallel and multi-core programming of IPC and OpenMP API on TI C6678. The core system sections have been evaluated in terms of runtime and accuracy under different optimisation approaches.

1. INTRODUCTION

Across the world, potholes in roads have been one of the major problems for drivers and this has started to worsen in recent years. Vehicles suffer from pothole related damage, while the driver's safety is compromised by accidents caused by potholes. Furthermore, the cost of dealing with potholes has been rising over recent years [1, 2].

Many pothole detection systems have been developed in order to enhance drivers' safety or to assist authorities in maintaining roads efficiently ([3-5]). Amount these approaches, A stereo vision based system shows the highest cost/performance ratio. The advantages in cost and robustness of a stereo vision system comes with the challenge of achieving real-time processing capability. This is essential for a pothole detection system since, as a moving vehicle updates new environmental information rapidly, the system needs to detect and evaluate potholes in real-time to reduce data storage. A stereo vision based real-time pothole detection system faces the main problems in intense calculation during the disparity map computation as well as complex processing algorithms in detection/verification stages.

A pothole detection algorithm is proposed in [6], employing the novel disparity calculation method described in [7]. During the disparity calculation, the disparity value search range for a pixel at the image line is restrained within a set of candidate values from the neighbouring pixels at a lower image line. This concept results in a 90% computation cut compared to most of the basic block matching algorithms so that real-time processing becomes possible. The block diagram of the pothole detection system is shown in Figure 1. The theory and implementation of the disparity calculation method have been stated in [7]. This paper mainly focuses on the implementation of the actual detection stages, including surface fitting and connected component labelling (highlighted stages in Figure 1). In the surface fitting stage, the disparity maps are converted into a real world coordinates domain (Euclidean Domain) point cloud. By taking a global approach, a 3D quadratic surface module fitting the majority of the points is calculated to represent the road surface. The parameters defining the surface are obtained through the bi-square weighted robust least-squares method. With the road surface points fitted by the estimated quadratic surface module, points left out are likely to be part of the potholes. Those points are then grouped through Connected Component Labelling (CCL) so that surface areas represented by the pixel groups are pothole candidates.

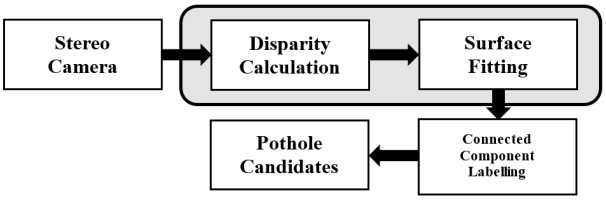

Figure 1 – Pothole Detection System. The highlighted system blocks are the implementation stages discussed in this paper.

The implementation platform is the Texas Instruments C6678 Multi-Core SoC Digital Signal Processor (TMS320C6678). The image processing algorithm was coded entirely in C to make it portable for various platforms especially for DSP. The surface fitting process involves heavy matrix calculations to compute the parameters of the 3D surface function. In order to reach the global best fit, the error function is computed and the parameters are updated according to the HAT function in each iteration to converge into the final result. The HAT function describes the influences of the given values on the estimated data modules [9]. The challenge is to optimise the algorithm for DSP to process matrix calculations efficiently. The communication between different components in the system is done through the Ethernet. This is achieved by employing the Network Development Kit (NDK) [14] [ref]. TheC6678 contains 8 CorePacs and it supports its own proprietary Inter-Processor Communication (IPC) and OpenMP API for parallel programming. During the implementation, multi-core programming has been employed to achieve an efficienct executing time. The paper is organised in the following fashion: Section 2 discusses the implementation of the detection stages. Section 3 states the optimisation made specifically for the DSP platform. Section 4 discusses the Multi-core Processing utilised during implementation. Section 5 concludes the paper.

2. IMPLEMENTATION OF THE POTHOLE DETECTION STAGES

To find the pothole, the approach was to find the discrepancies between the fitted plane and the actual plane, given a disparity map. Since road surfaces are generally smooth and continuous surfaces, a pothole area with uneven geometrical characteristics can be detected once the road surfaces are excluded. Disparity

978-1-4799-6843-5/14/$31.00 ©2014 IEEE

maps (*u,v,disparity value*) produced through stereo-vision input devices need to be converted into the Euclidean domain (*x,y,z*) before applying surface fitting.

2.1 Surface Fitting

2.1.1 Theory

A second-order quadratic surface model is used, since it is capable of simulation of most road surface conditions. The surface model is represented in Equation 1.

$$z = a_0 d + a_1 x + a_2 y + a_3 xy + a_4 x^2 + a_5 y^2 \quad (1)$$

$a_0 \ldots a_6$ are the coefficients for each variable. The sum of least squares is normally used to fit the planes in Equation 2.

$$S = \sum_{i=0}^{n} (z_i - \bar{z}_i) \quad (2)$$

where:

z_i = Actual z-value

$\bar{z}_i$ = Estimated z-value

The estimated plane fits the actual plane best when the partial derivative with respect to all the variables in Equation 1 is equal to zero.

$$\frac{\partial S}{\partial a_0} = \frac{\partial S}{\partial a_1} = \frac{\partial S}{\partial a_2} = \frac{\partial S}{\partial a_3} = \frac{\partial S}{\partial a_4} = \frac{\partial S}{\partial a_5} = 0 \quad (3)$$

The linear equation for each mentioned partial derivatives can be represented in matrix form as shown in Equation 4. The coefficients are computed using matrix algebra. S_{xy} corresponds to $S_{xy} = \sum_{i=1}^{n} \omega_i x_i y_i$. The initial weights for each element, ω_i, were set to 1.

$$\begin{bmatrix} n & S_x & S_y & S_{x^2} & S_{xy} & S_{y^2} \\ S_x & S_{x^2} & S_{xy} & S_{x^3} & S_{x^2y} & S_{xy^2} \\ S_y & S_{xy} & S_{y^2} & S_{x^2y} & S_{xy^2} & S_{y^3} \\ S_{x^2} & S_{x^3} & S_{x^2y} & S_{x^4} & S_{x^3y} & S_{x^2y^2} \\ S_{xy} & S_{x^2y} & S_{x^2y} & S_{x^2y} & S_{x^2y} & S_{x^2y} \\ S_{y^2} & S_{xy^2} & S_{y^3} & S_{x^2y^2} & S_{xy^3} & S_{y^4} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} S_z \\ S_{xz} \\ S_{yz} \\ S_{x^2z} \\ S_{xyz} \\ S_{y^2z} \end{bmatrix} \quad (4)$$

(Matrix A) (Matrix a) (Matrix S)

With the coefficients of the quadratic function obtained, the road surface can be estimated and the points belonging to the road can be grouped.

2.1.2 Implementation

The calculation of the function coefficients requires solving multiple linear equations, which is a very computationally expensive process in the DSP environment. During our implementation, QR decomposition is employed for its numerical accuracy and low computation. Using QR decomposition, an original matrix, say A, is decomposed into an orthogonal matrix, say Q, and an upper triangle matrix, say R. A can be obtained again by multiplying Q with R. The mathematical details can be found in [8]. Following this approach, Equation 4 can be expressed in the following way:

$$A * a = QR * a = S \quad (5)$$

Since the inverse of the unity matrix Q is simply its Hermitian transpose, Equation 5 can be converted as:

$$R * a = Q^H S \quad (6)$$

The upper-triangular matrix R makes it very simple to calculate coefficient a_5, which results in simpler calculation a_4 in the next step. Following this "from bottom to the top" calculation order, shown in Equation 7, the complexity of calculating surface parameters has been greatly reduced.

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} & r_{16} \\ 0 & r_{22} & r_{23} & r_{24} & r_{25} & r_{26} \\ 0 & 0 & 33 & r_{34} & r_{35} & r_{36} \\ 0 & 0 & 0 & r_{44} & r_{45} & r_{46} \\ 0 & 0 & 0 & 0 & r_{55} & r_{56} \\ 0 & 0 & 0 & 0 & 0 & r_{66} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} qs_0 \\ qs_1 \\ qs_2 \\ qs_3 \\ qs_4 \\ qs_5 \end{bmatrix} \quad (7)$$

(Matrix R) (Matrix a) (Matrix $Q^H S$)

In order to calculate the inverse of the upper triangular matrix, R, a simple loop was used rather than solving the linear equations as the matrix size was small at 6x6. Following the matrix decomposition philosophy, the computation process has been made real-time processing friendly, especially for the platform of embedded systems such digital signal processors. In order to compute Q and R, the Householder Reflection algorithm is used as the decomposing method. The matrix in Equation 4 is treated as having the form *Ax=b*. The Householder matrix, H, is shown in Equation 8.

$$H = I - \frac{2}{u^T u} u u^T \quad (8)$$

The first column of matrix A,c_1, is used as a reflector. It is reflected about a plane, given its normal vector *u*.

$$u = c_1 - ||c_1|| 2^{e_1} \quad (9)$$

Then, the transformation reflects onto the first standard basic vector, where the first entry is the only non-zero element.

$$e_1 = [1\ 0\ 0 \ldots]^T \quad (10)$$

In order to avoid numerical cancellation errors, we set:

$$u = c_1 + sign(a_1) ||c_1|| 2^{e_1} \quad (11)$$

Normalising with the Euclidean Norm of the first column, the equation for H in (8) is now:

$$H = I - \frac{2}{v^T v} v v^T \quad (12)$$

Applying H on c_1 will modify (9) into:

$$Hu = c_1 - sign(a_1) ||c_1|| 2^{e_1 c_1} \quad (13)$$

Applying H to matrix A zeroes all sub-diagonal elements for the first column:

$$H_1 A = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ 0 & * & * \\ 0 & * & * \end{pmatrix} \quad (14)$$

For the second column of A, only the first two are non-zero. The first column of the sub-matrix (denoted *) is needed to calculate *H* instead of the entire second column. The new *H* is inserted into an (m * n) identity to obtain H_2:

$$H_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & H & * \\ 0 & * & * \end{pmatrix} \quad (15)$$

This step is repeated until all sub-diagonal elements of A are removed to obtain the upper triangle matrix, R.

$$R = H_n \ldots \ldots H_3 H_2 H_1 A \quad (16)$$

To obtain the orthogonal matrix Q, the Householder matrix H is multiplied in reserve order, for every column.

$$Q = H_1 H_2 H_3 \ldots \ldots H_n \quad (17)$$

During our implementation, the source code of the QR

decomposition based on [8] has been modified to fit to the C6678 DSP platform.

2.2 Robust Fitting with Weighted Least Squares

2.2.1 Theory

A single iteration is insufficient to provide the best fit due to the outliers in the data. To minimise the influence of the outliers, weights were introduced to each data point. The weights determine how relevant the points are to the fit. They are updated using bi-square weighting and repeated until it converges. For bi-square weights, the value for each weight is given based on its distance from the central fitted line [9]. Points further from the fitted line have lower weight values. The concept is demonstrated in Figure 2, where the outlier data points are labelled in red.

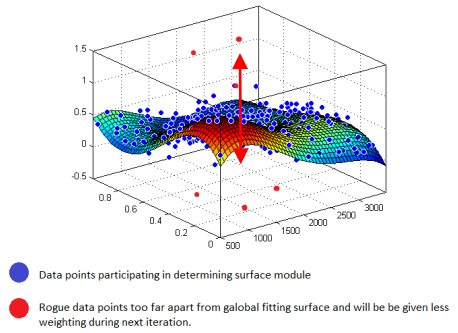

Figure 2 – Weight alteration of data points during fitting.

The residuals (actual z value - estimated z value) are adjusted based on Equation 18, where *h* is the leverage value obtained from the HAT matrix.

$$r_{adj} = \frac{r_i}{\sqrt{1-h_i}} \quad (18)$$

The adjusted residuals are further normalised using Equation 19:

$$u = \frac{r_{adj}}{K_s} \quad (19)$$

where u contains the normalised residuals, K is the tuning constant (0.463) and s is the Median Absolute Deviation (MAD) of the residuals. Finally, the weights are updated with Equation 20, where ω_i is the weight value for the data point.

$$\omega_i = \begin{cases} (1-(u_i)^2)^2 & |u_i| < 1 \\ 0 & |u_i| < 1 \end{cases} \quad (20)$$

The leverage values of the residuals are represented in the form of the HAT matrix. The HAT matrix, H, is obtained using Equation 21,

$$H = X(X^TX)^{-1}X^T \quad (21)$$

where X is the matrix with values $[(data\ length\ (x)(y)(x^2)(xy)(y^2))]$, and H is known as the HAT matrix.

2.2.2 Implementation

The leverage values are computed once and stored in the memory to reduce computation because they remain the same in all iterations. To conserve memory on the DSP, $X(X^TX)$ and X^{-1} were computed and stored separately. This arrangement only requires $2 * (matrix\ length * 6)$ memory as compared to$(matrix\ length)^2$.

. Median Absolute Deviation (MAD) is the median of the absolute difference between the median of the values and the values themselves. The MAD is required to compute the leverage values Equation 22 shows how the MAD is obtained.

$$Median\ Absolute\ Deviation = median\ (|median\ (data\ values) - data\ values|) \quad (22)$$

The simplest approach to compute the median is to sort the data, take the middle element for the odd length and the average of the middle elements for the even length. In this situation, a choice between merge sort [10], quick-sort [11] and quick-select needs to be made.

Merge sort divides the array into sub-arrays. It, then, sorts each sub-array and merges them. It repeats the mentioned steps until a single element is left. The quick sort algorithm consists of two operations, swapping and partitioning. It has a worst complexity of n^2 and an average complexity of $n \log n$. The performance of quick sort is dependent on the location of its pivot. Quick sort uses less memory, allowing it to fit within the cache. To make quick sort scalable, an iterative version of it was used instead of the conventional recursive quick sort. The quick-select algorithm has an average runtime of $O(n)$, similar to quicksort. It reduces computation by iterating the side of interest in the list of data values.

During our implementation, a benchmark was performed to compare the performances when finding the MAD using different sorting algorithms. The quick-sort function written in C was replaced by the *qsort()* function in *stdlib* because the recursion gets stuck, due to the small program stack on the DSP. The replaced function has a larger overhead, but executes on the DSP. The comparison results are shown in Figure 3. The quick-select algorithm was the fastest compared to quick sort and merge sort.

Sorting Method	Array Size	Average Completion Time (s)
Quick-sort	*636710*	*0.506*
Merge-sort	*636710*	*0.160*
Quick-select	*636710*	*0.013*

Figure 3 – Sorting Algorithm Benchmark on C6678.

2.3 Connected Component Labelling (CCL)

CCL finds the connected regions in an image. The implementation employed the 8-way connectivity CCL. The residuals are converted back into image pixels in binary using a simple threshold. As part of the CCL, the image is scanned along a row until it comes to a pixel x with value 255 (white pixel) [12]. Four neighbours around x are scanned, as shown in Figure 4, when this is true in the 8-ways version. The labelling of x is based on the following conditions:

- Assign a new label if the neighbours are all 0.
- Assign a label to x if only one neighbour has a value of 255.
- Assign a label to x and remember its equivalences if two or more neighbours have values of 255.

Figure 4 – 8-way Connected Components.

3. OPTIMISATION

To further improve the speed on the DSP, many options are available. By providing the compiler with pragmas/compiler directives, better memory placement and alignment, and using intrinsics, a larger degree of optimisation can be achieved. Multi-core processing can be employed to divide the workload between idle processors.

3.1 Pragmas

Pragmas are compiler directives that provide the compiler with more information about the code to improve the optimisation. Figure 5 provides a description of the pragmas used in the project.

Pragmas	Description
#pragma MUST_ITERATE (A,B,C)	Informs the compiler that the loop is executed a minimum of A times, with a maximum of B iterations, and that the number of iterations is a multiple of C.
#pragma DATA_SECTION (symbol, "*section name*")	Specifies the data section.
#pragma UNROLL (X)	Informs the compiler to unroll the loop X times.

Figure 5 – C6678 Pragmas [13].

3.2 Memory Management

In the C6678 DSP, a heap instance manages the actual memory, such as instances of HeapMem, HeapBuf or HeapMultiBuf. HeapMem is non-deterministic because each allocation request takes an arbitrary time [14]. However, it is able to allocate a memory of arbitrary size.

HeapBuf manages a fixed-size buffer, with equally sized blocks that can be allocated. HeapBuf replaced HeapMem where possible. It is very fast, deterministic, and ideal for objects that have a very similar size. Allocating and freeing memory from the HeapBuf instance takes the same time due to the deterministic nature.

3.3 Intrinsics

Intrinsics are a set of functions that are built in to compilers, so they are compiler specific. They can be in-lined compared to the library equivalent of the function. Intrinsics are very useful for Single Instruction Multiple Data (SIMD) operation. The C6678 is capable of SIMD operations which enables the processing of multiple data with a single instruction. This reduces the amount of instructions needed to execute the same amount of data. Figure 6 shows the intrinsic for SIMD operations during our implementation.

Intrinsic	Description
_fto128 (float src1, float src2, float src3, float src4)	Creates _x128_t from float (reg+3, reg+2, reg+1, reg+0)
_x128_t _qmpy32 (_x128_t, src1, _x128_t src2)	Four-way SIMD 320bit single precision multiply producing four 32-bit single precision results.
Float _get32f_128 (int X)	Extracts register X from register quad

Figure 6 – C6678 SIMD Instructions [14].

Due to the size of the matrix, data sets have to be placed on the DDR3 external memory. This bottlenecks the memory access to 64-bit data at a time, meaning SIMD operations greater than 64-bit are not possible. An alternative to SIMD would be to distribute the processor workload to other cores. This is directly related to Multicore processing, which is discussed in the next section.

4. MULTI-CORE PROCESSING

The TI C6678 contains 8 CorePac DSP cores. It supports its own proprietary Inter-Processor Communication (IPC) and OpenMP for parallel programming. Software can benefit in terms of speed and efficiency through parallelism as more data can be processed concurrently.

4.1 OpenMP

OpenMP[15] is an API for creating multi-threaded applications and its support is dependent on the compiler for the platform. The API is called in the form of compiler directives. The OpenMP API uses the Inter-Processor Communication (IPC) to get data across different cores. The C6678 DSP supports the OpenMP 3.0 standard.

OpenMP divides the workload with a master thread, forking the workload to worker threads. At the end of the computation, the worker threads would terminate. The model is illustrated in Figure 7.

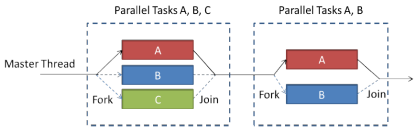

Figure 7 – OpenMP Fork Join Model.

OpenMP can be migrated to different platforms with little to no modification if they support the same OpenMP API version. It is worth noticing that OpenMP performs well on large data sets, which works well with a high resolution input sensor running at a high frame rate. This is the main reason that OpenMP is employed during our implementation.

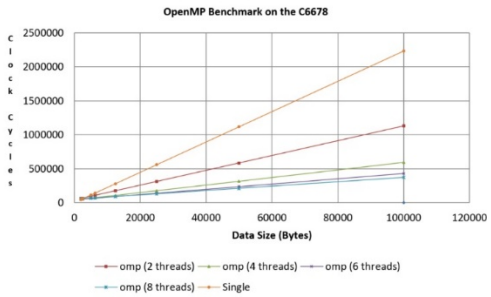

Figure 8 – OpenMP Benchmark on the C6678.

4.2 API Calls

The compiler interprets the pragmas/directives for OpenMP when the compiler options for OpenMP are enabled. Most compilers that do not support OpenMP ignore the pragmas, allowing the code to run normally. In the OpenMP model, threads communicate with each other by sharing variables. Synchronisation clauses were used to eliminate race-conditions and to preserve the correctness of the algorithm. In addition, OpenMP provides locks to prevent multiple threads from accessing the same variable together, as well as a schedule clause for Work Sharing. Further details are given in [16].

4.3 Parallelising For Loops

Many *for* loops in C typically can be parallelised to improve its performance. To optimise loops for OpenMP, the loop that requires the most computation is identified and rewritten to execute independently without dependencies. Lastly, the appropriate OpenMP directives are applied to the loop. Code 1

shows pragmas used in *for* loops during the implementation. Directives are normally added before the loop for the shorter version without the need to define a parallel region. This could be shortened to Code 2.

```
#pragma omp parallel
{
    #pragma omp for
    {
        //Code Block
    }
}
```

Code 1 – Original "for" loop.

```
#pragma omp parallel for
// For Loop
```

Code 2 – Shortened parallel "for" loop.

4.4 Parallelising matrix multiplication

Matrix multiplication is the most computationally intensive function within the surface fitting algorithm. It can be parallelised because it has no dependencies. Code 3 shows, during our implementation, the use of OpenMP to parallelise matrix multiplication.

```
# pragma omp parallel for private ( i, j, k )
        for ( i = 0; i < m; ++i){
            for ( j = 0; j < n; ++j){
                for ( k = 0; k < m2; ++k){
                    result[i * n + j] += mat_x[i * n + k] * mat_y[k * n
                        + j];
                }
            }
        }
```

Code 3 – Matrix Multiplication with OpenMP.

The reduction clause caters for repetitive operations on the same variable, such as MAC. The compiler makes a copy of the variable and initialises it based on the type of operation [38]. Each local copy is then updated. At the end of the parallel region, local copies are combined into a single value and assigned to the designated variable. It was useful when converting the partial derivative of the sum of least squares into its linear equation form in Equation 4. Code 4 shows an example of OpenMP on MACs.

```
double acc=0.0, A[MAX];
int i;
#pragma omp parallel for reduction (+:acc)
for (i=0;i< MAX; i++) {
    acc += A[i];
}
```

Code 4 – Code reduction example in MACs.

4.5 RESULTS OF OPTIMISATION

Figure 9 shows the results of the implementation on both the PC and DSP. The PC used is a Macbook Air 2013 and the DSP TMS320C6678. On both platforms, the application was built in release mode with maximum compiler optimisation. OpenMP implementation almost doubles the frame rate on the PC. However, on the DSP, OpenMP actually slows down the frame rate. It is assumed that the compiler optimisation is not improved with OpenMP.

Platform	Resolution (pixels)	Average FPS without OpenMP	Average FPS with OpenMP
PC	1070 x 743	1.4	3.3
DSP	1070 x 743	3.3	0.9*

Figure 9 – Results of the implementation on hardware

5. CONCLUSION

This paper has demonstrated the implementation of a pothole detection system on the Texas Instruments C6678 Multi-Core Digital Signal Processor. The algorithm used for the surface fitting method employed in the original pothole detection system has been discussed and several modifications have been made to reduce the computation. Different approaches of robust fitting with weighted least squares are compared and the most computational efficient method is implemented. Finally, multi-core processing with the assistance of OpenMP further reduces the processing time in the hardware platform. The detection stages of the pothole detection system, post optimisation, have reached 5 frames per second. In order to make the complete system real-time applicable, the disparity calculation stage needs to be optimised and implemented in the multi-core DSP environment. This is not included in this paper. Optimisations can still be made in linear assembly or assembly to further reduce the computation time.

REFERENCES

[1] BBC News, "millions of cars' damaged by potholes," http://www.bbc.co.uk/news/uk-21770969/, 2013, Accessed: 2013-11-01.

[2] BBC News, "Potholes which damaged cars prompt council pay-outs," http://www.bbc.co.uk/news/ukengland-19367206/, 2013, Accessed: 2013-11-01.

[3] Q. Li, M. Yao, X. Yao, and B. Xu, "A real-time 3d scanning system for pavement distortion inspection," *Measurement Science and Technology*, vol.21, no. 1, pp. 015702, 2010.

[4] A. Mednis, G. Strazdins, R. Zviedris, G. Kanonirs, and L. Selavo, "Real time pothole detection using android smartphones with accelerometers," in Distributed *Computing in Sensor Systems and Workshops* (DCOSS), 2011 International Conference on, 2011, pp. 1–6.

[5] E.Salari, E. Chou, and J. J Lynch, "Pavement distress evaluation using 3d depth information from stereo vision," Tech. Rep., 2012.

[6] Z. Zhang, X. Ai, C.K. Chan and N. Dahnoun, "An efficient algorithm for pothole detection using stereo vision," *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May. 2014

[7] Z. Zhang, X. Ai, and N. Dahnoun, "Efficient disparity calculation based on stereo vision with ground obstacle assumption," *European Signal Processing Conference (EUSIPCO)*, 2013, vol., no., pp.1,5, 9-13 Sept. 2013

[8] F. Charles, H. G. Golub, "The QR Factorization", *Matrix Computation (3rd ed.)* 1996.

[9] K.S. Arun, T.S. Huang, S.D. Blostein, "Least-Squares Fitting of Two 3-D Point Sets,"*Pattern Analysis and Machine Intelligence*, IEEE Transactions on , vol.PAMI-9, no.5, pp.698,700, Sept. 1987.

[10] T. Cormen, R. Rivest, C. Stein. "2.3.1 The divide-and-conquer approach", In *Introduction to Algorithms (3ed ed.)* 2009.

[11] T. Cormen, R. Rivest, C. Stein. "Chapter 7 Quicksort", In *Introduction to Algorithms (3ed ed.)* 2009.

[12] T. Walker, R. Fisher, S. Perkins, and E. *Wolfart. Connected Components Labelling*. 2003

[13] Texas Instruments. *TMS320C6000 Optimizing Compiler v7.6 User's Guide*. 2013. url: http://www.ti.com/lit/ug/spru187v/spru187v.pdf.

[14] Texas Instruments. *TI SYS/BIOS v6.35 Real-time Operating System User's Guide*. 2013. url: http://www.ti.com/lit/ug/spruex3m/spruex3m.pdf.

[15] Texas Instruments. *SYS/BIOS Inter-Processor Communication (IPC) 1.25 Users Guide*. 2012. url: http://www.ti.com/lit/ug/sprugo6e/sprugo6e. pdf.

[16] Larry Meadows Tim Mattson. *A Hands-on Introduction to OpenMP*. 2008. url: http://openmp.org/mp-documents/omp-hands-on-SC08.pdf