Parallel Stereo Vision Algorithm

Paul Cockshott, Susanne Oehler, Tian Xu, Paul Siebert, Gerardo Aragon

***Abstract*—Integrating a stereo-photogrammetric robot head into a real-time system requires software solutions that rapidly resolve the stereo correspondence problem. The stereo-matcher presented in this paper uses therefore code parallelisation and was tested on three different processors with x87 and AVX. The results show that a 5mega pixels colour image can be matched in 5,55 seconds or as monochrome in 3,3 seconds.**

I. Introduction

This paper presents initial efforts to solve the stereo-correspondence problem in real time manner for the robot head used in the Clothes Perception and Manipulation project (www.clopema.eu). This means that generating 2.5D range maps should be obtained in an on-line manner. The stereo correspondence problem comprises the task of locating for each pixel on one image of a stereo pair, the corresponding location on the other image of the pair. Therefore the problem is solved by constructing a displacement field (also termed parallax or disparity map) that maps points on, for example, the left image to the corresponding location on the right image: $I_l(x, y) \rightarrow I_r(x', y')$. In this case the displacement field is usually expressed as two disparity maps $Dx(x, y)$ and $Dy(x, y)$ each storing horizontal and vertical displacements respectively that map each pixel in the left imageIl into the corresponding location of the right imageIr, as follows:

$$I_r(x', y') = I_l(x + D_x(x, y), y + D_y(x, y))$$

It should be noted that the displacement values stored in these maps are both real valued and signed. Knowing the correspondences between stereo pair images and also the geometric configuration of the cameras that captured the stereo pair, by means of camera calibration, it then becomes possible to recover range values, for each matching stereo-pair of pixel valid associated with a valid disparity estimate, through a process of triangulation.

A. The Robot Head

The robot head comprises two Nikon DSLR cameras (D5100) that are capable of capturing images at 16 mega pixels. These are mounted on two pan and tilt units (PTU-D46) with their corresponding controllers (Figure I.1). The cameras are separated by a pre-defined baseline for optimal stereo capturing. The hardware is interfaced to a Intel Core i7-3930K computer at 3.20 GHz with 16GB in RAM running Fedora 17. Camera and controller drivers are developed under ROS (www.ros.org). The robot head is able to verge its cameras on a point [1] and it will feature the stereo-matcher described in this paper.

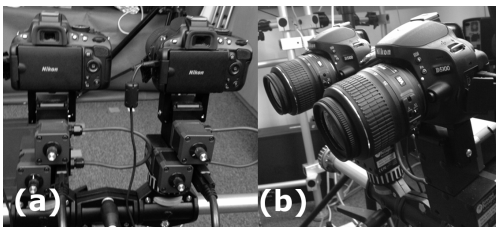

Figure I.1. Side (a) and back (b) view of the robot head.

II. The legacy software and its history.

The research underpinning all versions of image matching algorithm reported here have their origins in the Multi Scale Signal Matching (MSSM) algorithm proposed by Jin and Mowforth [2, 3]. MSSM operates by tracing dense, local, matches over a discrete scale-space: essentially an image pair is reduced in progressively in detail by iterated blurring operations. The most blurred versions of the stereo-pair, containing least detail and hence scope for matching ambiguity, are matched first and then subsequent matches at finer scales progressively resolve the detail removed by the blurring operations to produce a final full-resolution set of displacement, or disparity, estimates between corresponding locations on the matched images.

Jin's MSSM[2], formulated a coarsely quantised scale-space ($\sqrt{2}$ blur factor between scales) that required of the order of a few tens of image scales to be computed and therefore fewer matching operations were required to traverse from a coarse solution through to a full-resolution match and used a Gaussian spatial weight to a Pearson cross-correlation window used for local matching.

Typically, in excess of two days of CPU time were required to match a 256x512 stereo-pair using a SUN 3 workstation (Motorola 68000 based processor) in 1989. Urquart modified the MSSM to incorporate the semi-pyramid[4] and as a result MSSM could then match the 256x512pixel stereo-pair in the order of one hour. Van Hoff, further improved the MSSM algorithm by incorporating a fully regular image pyramid sampled with a half-octave reduction factor between levels[5].

Table I
TYPICAL EXECUTION TIMES, IN SECONDS, FOR MSSM IN 1993 [7]

Image Size (pixels)	256×256	512×512	576×768
Sun SPARCstation LX	32.7s	130.9s	215.0s
IBM RS6000 320H	15.5s	64.5s	104.6s
Sun SPARCstation 10	10.7s	50.5s	71.2s

Minor modifications[6] further improved the execution rate as listed in Table 1.

The original C system was recoded in Java as C3D incorporating Van Hoff's improved version of MSSM [8, 9].

III. LEGACY CODE - PARALLELISATION STRATEGY

C3D is based on research outputs from the past 25 years and has been used for numerous research projects involving stereo-photogrammetric camera systems. For the current objective its employment is no longer adequate due its slow processing times. To achieve acceptable real-time timings it has been decided to create new parallelised software.

In order not to lose the research inputs, which had led to the creation of C3D, the source code of the latter is being used as the design basis and benchmark for the new parallelised software. Due to the variety of different authors and the time span over, which C3D got developed, earlier documentation of the implementation turned out not to be fully consistent with the source code of the current version of C3D. It was necessary to extract the essential parts of the source code and transcribe their functionalities. C3D employs external libraries and follows an object-oriented design, which makes it difficult to translate directly into a parallelised form. We decided to first transcribe the source code into a set of Matlab scripts, removing the external libraries' influences and its object orientated nature. These Matlab scripts can be tested for output validity against the Java software. More importantly they provide design and documentation for a new parallelised version, as for instance, described in section V.

In this first phase of parallelisation, we decided to focus on the stereo-matching algorithm of C3D, without transcribing the C3D scale-space pyramid code. The C3D stereo-matcher takes two images, as input and outputs a horizontal disparity matrix, a vertical disparity matrix and a confidence matrix of the same size as the input images. The matching process is as followed:

1) First a pyramid representation with a given number of pyramid levels for the left and right image is produced. For the smallest level of the pyramid the horizontal disparity, vertical disparity and confidence matrix are initialised to zero.
2) Starting at the smallest pyramid level *Step 2 will be repeated for each pyramid level*:
   a) *Step 2a to 2h is repeated for each iteration on the current pyramid level*: The sampling step ($sampStep$) is initialised to one by default, unless $totIter > 1$ and $curIter < 7$. The sampling step is calculated as follows: $sampStep = (totIter - curIter - 1) * \frac{0.9}{totIter-1} + 1$ where $totIter$ is total iteration and $curIter$ is the iteration counter.
   b) *Step 2b to 2h is repeated twice for each iteration (1 iteration = 2 match cycles)*: Five different warped right image instances are created by applying bi-linear interpolation using the horizontal and vertical disparities with and without sampling step shift. The first is created using the horizontal and vertical disparities without shifts. For the other four, horizontal and vertical $\pm sampStep$ is added to the disparities before warping.
   c) For the left image and each of the five warped images a standard deviation matrix is calculated by multiplying the images with themselves, i.e. squaring them, and convolving the result horizontally and vertically with a Gaussian kernel.
   d) With the same Gaussian kernel, covariance matrices are calculated multiplying the warped images with the left and then convolving.
   e) Based on the standard deviation and covariance matrices the correlation coefficients are calculated.
   f) Having obtained five correlation coefficient matrices, 2nd order polynomial maximisation is applied to the corresponding elements of the matrices. This is essentially the same as in section **??**. In case of a confidence value greater than one and the zero point correlation coefficient (c_0) greater than zero it readjusts the disparity ($disp$) to be: $disp = disp * \frac{1-c_0}{conf-c_0}$ followed by overwriting the confidence with 1. Should C3D fail to fit a polynomial, it sets the disparity to zero and the corresponding confidence to 0.4.
   g) The newly computed horizontal and vertical disparities matrices are multiplied by the sampling step and added on to the initial horizontal and vertical matrices. To obtain again one confidence matrix it computes $ConfHV = ConfH \otimes ConfV$ weights this against the initial confidence ($InitConf$) : $InitConf = ConfHV + 0.75 * (InitConf - ConfHV)$
   h) The disparity matrices are then weighted by the confidence matrix $Disp_{xy} = Disp_{xy} \otimes InitConf$. This is followed by smoothing using the local confidence matrix as a convo-

lution kernel. The same process is then applied to the confidence matrix. *The weighted smoothing process (2h) is repeated twice. If the steps 2b to 2h have been run twice the process will either step to the next iteration, starting at 2a or in case the total number of iterations for the pyramid level has been reached move to the next pyramid level (3).*

3) After the matching iterations the resulting horizontal and vertical disparity matrices and the confidence matrix are smoothed once more with a 3x3 average filter, before being expanded to the size of the next pyramid level. OR
4) The final (largest) level of the pyramid is returned as output.

IV. The Hardware target and the Compiler

We have used Vector Pascal [10, 11] as our source language because it is an array language of a similar semantic level to Matlab thus easing the prototyping process, it targets and automatically parallelises code for a range of parallel processor chips including the multi-core AVX machines we are using; and, we have access to the developers we can have new code generators added for additional target machines as we buy them whilst leaving the core vision code unchanged. In Vector Pascal all operators are overloaded so that they can operate on arrays and vectors as well as scalars. Using compiler flags, a single program can be compiled, with differing levels of parallelism, to target a range of microprocessors.

Each array assignment statement is evaluated by the compiler for parallelisation in two ways:

1) If it is a two dimensional array and it uses only basic arithmetic operations or pure functions (side effect free) on the right hand side, then the work on rows of the array is interleaved between different n processors so that processor j handles all rows i such that $i \bmod n = j$.
2) If the right hand side contains no function calls and operates on adjacent array elements, then the compiler generates SIMD code.
3) If the expression on the right is a conditional one with no function calls it is evaluated using boolean masks to allow SIMD execution (Algorithm 1).

At the start of the project we evaluated 3 architectures: a 6 core Intel Ivy bridge (3.2GHz, 12MB cache) and an 8 core AMD Bulldozer (3.1GHz, 8MB cache). Both the Ivy Bridge and the Bulldozer utilise the AVX instruction set, capable of operating on vectors of 8 single precision floating point values. We also tested an AMD Fusion system, a chip with a GPU on the same die as a pair of x86 cores. We developed two Vector Pascal code generators for the Fusion, one using the Virtual SIMD approach [12] and the other translating directly to OpenCL. Although the Virtual SIMD approach is known to work with the Cell processor, initial trials showed the performance of both techniques on the Fusion architecture was poor. This appears to be due to the cost of transfers between the two address spaces of the machine. We thus rejected this target architecture.

Algorithm 1 Translation of Pascal conditional expressions to AVX instructions.

```
var a,b,c:array[1..8] of real;
begin
 a:=if c<b then c*c else b;
=======
 vmovdqu   YMM0, [PmainBase+  -96];
 vcmplps   yMM7, YMM0, [PmainBase+ -64];
 vmovdqu   YMM6, [PmainBase+ -96];
 vmulps    YMM0, YMM6, YMM6;
 vandps    YMM0, YMM0, yMM7;
 vpandnpd  YMM1, yMM7, [PmainBase+ -64];
 vorpsb    YMM0, YMM0, YMM1;
 vmovdqu   [PmainBase+ -32], YMM0;
```

In the longer term our preferred target architecture is the Intel MIC which combines over 50 x86 cores each of which can operate on vectors of 16 single precision floating point numbers. Provided a Linux system with shared memory is available on this machine, experience shows that production of a code generator for it is task of about a month or two.

V. The parallel Algorithm

Although the Pascal Compiler does not require explicit parallelism, the programmer, by expressing their algorithm in terms of array operations, supplies implicit parallelisation hints. Our aim was to derive an algorithm that was similar to our existing one, but is expressed in a data parallel fashion.

Like our Java algorithm, it uses a scale-space pyramid structure to enhance the efficiency of search. It attempts to perform a match moving from the coarsest to the finest level of detail. The key targets for parallelisation were: the process of extracting disparities at any given scale; and, the process of creating the scale-space pyramid from the initial images.

The basic algorithm is the same as was applied in C3D except that instead of using explicit loops for convolution, sub-sampling etc., we express everything in terms of operators over matrices of floating point numbers.

The basic step in forming a pyramid is to repeatedly convolve layers with a 5 wide blurring kernel separable to prevent aliasing and then sub-sample.

Given a starting image A as a two dimensional array of pixels, the algorithm first forms a temporary array T whose elements are the weighted sum of adjacent rows. Then in a second phase it sets the original image

Algorithm 2 The parallel correlation function. The PM messages indicate lines with both SIMD and multi-core parallelism. Storage management code is not shown.

```
  PROCEDURE  computesim(var sim,l,r:image);
  var bottom:pimage;i:integer;
  const k:kernel=(0.0816,0.218,0.3032,0.218,0.0816);
  BEGIN
PM  for i:= 0 to l.maxplane do sim[i]:=l[i]*r[i];
    (* got products for the top term
       do summation using convolution  *)
    pconv(sim,k);
    (* now we compute the l2 norm of l
       and divide the top by it*)
PM  for i:= 0 to l.maxplane do bottom^[i]:=l[i]*l[i];
    pconv(bottom^,k);
    for i:= 0 to l.maxplane do
PM    sim[i]:=sim[i]/sqrt (bottom^[i]);
    (* repeat  with the r l2 norm *)
PM  for i:= 0 to l.maxplane do bottom^[i]:=r[i]*r[i];
    pconv(bottom^,k);
    for i:= 0 to l.maxplane do
PM    sim[i]:=sim[i]/ sqrt(bottom^[i]);
PM  for i:= 0 to l.maxplane do sim[i]:=sim[i]*sim[i];
   END;
```

to be the weighted sum of the columns of the temporary array.

The bulk of the actual work in the convolution is done by calls to a parallel multiply accumulate procedure which is called twice to perform vertical and horizontal passes.

```
MA(T[lo..hi],
    p[0..hi-lo+0],p[1..hi-lo+1],
    p[2..hi-lo+2],p[3..hi-lo+3],
    p[4..hi-lo+4],kerr);
...
MA(p[bm..tm,lo..hi],t[bm..tm,0..hi-lo+0],
    t[bm..tm,1..hi-lo+1],
    t[bm..tm,2..hi-lo+2],
    t[bm..tm,3..hi-lo+3],
    t[bm..tm,4..hi-lo+4],kerr);
```

where `T` and `p` are image planes. `T[lo..hi]` selects rows `lo` to `hi` of the image. The multiply accumulate procedure has the form

```
   PROCEDURE MA(VAR a,p,q,r,s,t:plane;k:kernel);
   BEGIN
PM a:= p*k[1]+q*k[2]+r*k[3]+s*k[4]+t*k[5];
   END;
```

It multiplies the whole image plane term by the factor and adds it to the image plane acc. The `PM` generated in the listing file indicates that the line is parallelised across cores and uses SIMD.

Algorithm 2 shows code used to perform a parallel correlation between two images.

VI. Complexity analysis.

Denote the complexity in floating point operations of a match on a given level of the pyramid as $M(l)$

Assume that the base image is of size x, y the number of pixels at level l will be

$$p(l) = \frac{xy}{2^{l-1}} \tag{VI.1}$$

since the linear dimensions of each level reduce by a factor of $\sqrt{2}$ between levels and the base level is 1. The cost of matching a layer is the cost of doing the basic match at that level $m(l)$ plus the cost of propagating down the disparities from higher layers into layer l that is $d(l)$ so

$$M(l) = d(l) + m(l) \tag{VI.2}$$

The cost of propagating down the disparities is the cost of interpolating each pixel I times the number of pixels in layer time 3 to account for the disparity image having 3 planes, plus the time to rescale the x and y planes of the disparity (2 multiplies per pixel)

$$d(l) = 2p(l) + 3Ip(l) \tag{VI.3}$$

The interpolation process computes a matrix bfrom a matrix a so that one destination pixel b_{ij} requires 20 maths operators per colour to compute it from 4 pixels in the level above as follows

$$\begin{aligned} b_{ij} = \quad & (a_{\left\lfloor \frac{i}{\sqrt{2}} \right\rfloor \left\lfloor \frac{j}{\sqrt{2}} \right\rfloor}(1 - (\frac{j}{\sqrt{2}} - \left\lfloor \frac{j}{\sqrt{2}} \right\rfloor))+ \\ & a_{\left\lfloor \frac{i}{\sqrt{2}} \right\rfloor \left\lfloor 1+\frac{j}{\sqrt{2}} \right\rfloor}(\frac{j}{\sqrt{2}} - \left\lfloor \frac{j}{\sqrt{2}} \right\rfloor))(1 - (\frac{i}{\sqrt{2}} - \left\lfloor \frac{i}{\sqrt{2}} \right\rfloor)) \\ & +(a_{\left\lfloor 1+\frac{i}{\sqrt{2}} \right\rfloor \left\lfloor \frac{j}{\sqrt{2}} \right\rfloor}(1 - (\frac{j}{\sqrt{2}} - \left\lfloor \frac{j}{\sqrt{2}} \right\rfloor))+ \\ & a_{\left\lfloor 1+\frac{i}{\sqrt{2}} \right\rfloor \left\lfloor 1+\frac{j}{\sqrt{2}} \right\rfloor}(\frac{j}{\sqrt{2}} - \left\lfloor \frac{j}{\sqrt{2}} \right\rfloor))(\frac{i}{\sqrt{2}} - \left\lfloor \frac{i}{\sqrt{2}} \right\rfloor) \end{aligned} \tag{VI.4}$$

Register optimisation should be able to reduce the number of these operations but these operations are not be done by SIMD instructions since the source pixels in matrix a are not regularly spaced. We can thus deduce that, unoptimised,

$$d(l) \approx 182p(l) \tag{VI.5}$$

The basic within a level match breaks down into the component costs

1) Warping the image $w(l)$
2) Forming correlation images for the 5 offsets $s(l)$ on 3 colour planes
3) Polynomial interpolation $q(l)$
4) Smoothing $s(l)$

$$m(l) = w(l) + s(l) + q(l) + s(l) \tag{VI.6}$$

Warping requires 16 floating point operations per pixel on each of 3 colour planes so we have

$$w(l) = 48p(l) \tag{VI.7}$$

Warping is again unsuitable for SIMD code. If we designate the cost of doing a correlation as $c(l)$ then we have $s(l) = 15c(l)$and the correlation requires 8 floating point operations (two of which are $\surd$) per pixel plus 3 image convolutions with a 5 element kernel. If we designate the cost of convolution by $k(l)$ we have

$$s(l) = 15(8p(l) + k(l)) \quad \text{(VI.8)}$$

Polynomial interpolation requires 16 floating point ops per pixel so

$$q(l) = 16p(l) \quad \text{(VI.9)}$$

Smoothing requires 2 floating point ops per pixel plus 2 convolutions over 3 planes this amounts to

$$s(l) = 6(p(l) + k(l)) \quad \text{(VI.10)}$$

Separable 2D convolution with a 5 element kernel requires 20 floating point operations per pixel so we have

$$k(l) = 20p(l) \quad \text{(VI.11)}$$

Substituting into VI.6 we have

$$m(l) = 48p(l) + 140p(l) + 16p(l) + 121p(l) = 325p(l) \quad \text{(VI.12)}$$

Substituting (VI.12) and (VI.5) into (VI.2) we get

$$M(l) = 507p(l) = 507\frac{xy}{2^{l-1}} \quad \text{(VI.13)}$$

Summing this over n pyramid layers we get

$$\sum_{l=1}^{n} 507\frac{xy}{2^{l-1}} \approx 1000xy \quad \text{(VI.14)}$$

since the sequence 1, $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ tends to 2. So for a 5 megapixel image, the number of floating point operations is of the order 5×10^9 for the matching. On top of this we have to calculate the number of operations necessary to create the two image pyramids. The pyramid construction process at present involves a convolution of each layer by a kernel to blur it followed by a sub-sampling. The sub-sampling requires just over 2 floating point operations per pixel per colour, and the convolution cost is given in (VI.11). There is a complication involved in that the bottom layer is not sub-sampled. The total floating point cost of constructing each pyramid is thus

$$\sum_{l=2}^{n} 6p(l) + \sum_{l=1}^{n} 3kl = 126xy \quad \text{(VI.15)}$$

Given that we have two pyramids we can compute the total floating point cost for matching two 2448x2050 colour images as being 12722448x2050=6.4E+09. This implies a minimum data throughput of 51GB for the algorithm.

The working sets for the basic operations of the algorithm are large. For instance for convolution we have 3 planes, each of 20MB which have to be horizontally convolved to a similar sized buffer and then vertically convolved back to the original planes so 120MB of data has to be processed at the bottom pyramid layer. This is far larger than the cache size. We have an option for the convolution routine to adapt to the known cache size and divide the image into stripes that will fit into the cache. We present results of this below. Total active data size for the two source pyramids and the output pyramid is 360MB, in addition temporary buffers of up to a further 180MB are used during the algorithm. Transfers to and from the caches

VII. Results and Conclusions

We tested speed and precision for the Vector Pascal matcher and the C3D matcher using fractal-noise image pairs. The fractal noise images (2448x2050 pixels) were generated with different noise frequency patterns and warped using bi-linear interpolation for a set of predefined disparities maps. For the stereo-matching the two original images served as the left and the warped images as the right image. To evaluate the precision of the new matcher and C3D, the output disparities have been compared against the predefined disparities, calculating the root mean square (RMS) error over the differences of disparities. The speed of the parallelised matcher was tested for two different instruction sets, AVX and x87, two different image formats, monochrome and colour, and run in parallel over different number of cores for the three processors presented in section IV.

A. Speed results

The x87 served as the speed benchmark for each processor. The Intel 6 and AMD 8 run colour matches 2.2 and 2.7 times as fast with AVX, respectively while multi-core parallelisation gives further accelerations of 2.4 and 2.6 times, respectively. In overall, the AMD 8 was the slowest as observed in Figure VII.1. Hyper-threading in Intel 6 provided an overall speed-up of 3.2 times with respect to using a single core. Cache optimisation of the convolve function (Algorithm 2) gave a further speedup of 1.3 times with respect to no cache optimised code, as depicted in Figure VII.2.

C3D was run on the Intel 6 with Java SE7, for 1 iteration at 16.10 seconds and for the standard number of 10 iterations at 38.93 seconds while matching monochrome images. Comparing the Intel 6 AVX 12 core version with C3D's standard setup, the new matcher is 22.8 times faster. Even using colour images it is approximately 13 times faster. Considering the MSSM timings using the 1992 SPARCstation LX of 215 seconds over 576x768 pixels, in 20 years we have become 1430 times faster. This is almost exactly in line with Moore's law growth rates.

B. Accuracy results

C3D is slightly more accurate than the Pascal version, with an RMS error of 0.41 pixels, compared to the

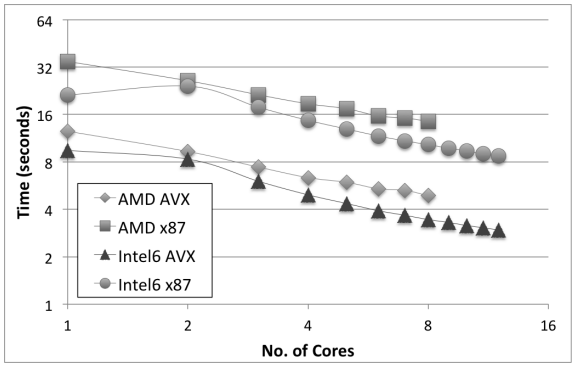

Figure VII.1. Speed performance for the Intel 6 and AMD architectures with the AVX and x87 instruction sets while matching colour images.

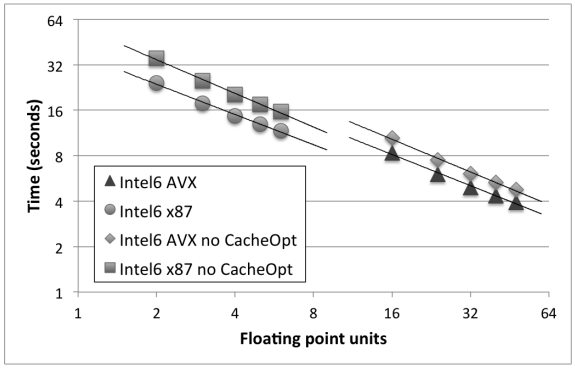

Figure VII.2. Log-log graphic of the speed performance vs Floating Point Units (FPU) of the Intel 6 with both instruction sets. Trendlines for the Intel 6 AVX and x87 instructions sets are $y = 54.6x^{-0.68}$ and $y = 37.6x^{-0.66}$, respectively; and for the Intel 6 without cache optimisation for both instruction sets are $75.5x^{-0.71}$ and $y = 57.5x^{-0.74}$, respectively.

0.83 pixels (see Table II). We attribute the difference to the slightly different scale-space pyramids used. However, the new matcher achieves more accurate results over colour images than over monochrome, which is interesting as C3D can only match monochrome images.

Overall, the 6 core Ivy Bridge has shown the best performance in terms of balancing speed and accuracy. The performance of the new matcher demonstrates the potential feasibility of integrating a stereo-robot head into a real-time system. Furthermore the new matcher should be able to run with minimal changes on the Intel MIC, possibly improving further its timings. In terms of accuracy C3D is still leading, hence, we shall proceed with transcribing and parallelising its pyramid code.

Table II
COMPARATIVE ACCURACY OF MATCHING

		Java C3D		Monochrome		Colour	
		1 Iter	10 Iter	AVX	x87	AVX	x87
Img	Disp	RMS	RMS	RMS	RMS	RMS	RMS
1	1	0.00	0.01	0.29	0.50	0.09	0.11
	2	0.06	0.02	0.29	0.50	0.09	0.11
	3	0.05	0.02	0.29	0.50	0.09	0.108
	4	0.33	0.32	0.66	1.26	0.32	0.47
	5	0.95	0.92	5.35	6.52	1.51	1.44
	6	1.17	1.15	13.12	13.55	3.0	2.81
2	1	0.01	0.01	0.47	0.541	0.10	0.10
	2	0.06	0.02	0.48	0.55	0.10	0.10
	3	0.04	0.02	0.48	0.55	0.10	0.10
	4	0.38	0.36	0.75	0.94	0.32	0.44
	5	0.97	0.94	1.92	2.11	1.44	1.38
	6	1.19	1.16	4.79	4.39	2.76	2.32
μ	RMS	0.43	**0.41**	2.41	2.66	**0.83**	**0.79**
σ	RMS	0.47	**0.46**	3.65	3.75	**1.04**	**0.93**

Disparities (1)none, (2), 1 pixel horizontal, (3) 1 pixel vertical, (4) 10 pixels horizontal, (5) sin horizontal, (6) sin horizontal and vertical.

REFERENCES

[1] G. Aragon-Camarasa, H. Fattah, and J. P. Siebert, "Towards a unified visual framework in a binocular active robot vision system," *Robotics and Autonomous Systems*, vol. 58, pp. 276–286, Mar. 2010.

[2] J. Zhengping, *On the multi-scale iconic representation for low-level computer vision systems*. PhD thesis, The Turing Institute and The University of Strathclyde, 1988.

[3] Z. Jin and P. Mowforth, "A discrete approach to signal matching," tech. rep., 1989.

[4] C. Urquhart, *The active stereo probe: the design and implementation of an active videometrics system*. PhD thesis, University of Glasgow, 1997.

[5] A. van Hoff, "Efficient computation of gaussian pyramids," tech. rep., Turing Institute, 1992.

[6] A. van Hoff, "An efficient implementation of MSSM," tech. rep., Turing Institute, 1992.

[7] C. Urquhart and J. P. Siebert, "Towards real-time dynamic close range photogrammetry," vol. 2067, p. 240251, 1993.

[8] J. Siebert and C. Urquhart, "C3D: a novel vision-based 3d data acquisition system," in *Proceedings of the Mona Lisa European workshop, combined real and synthetic image processing for broadcast and video production, Hamburg, Germany*, pp. 170–180, 1994.

[9] J. Siebert and S. Marshall, "Human body 3d imaging by speckle texture projection photogrammetry," *Sensor Review*, vol. 20, no. 3, pp. 218–226, 2000.

[10] T. Turner, "Vector Pascal: a computer programming language for the FPS-164 array processor," tech. rep., Iowa State Univ. of Science and Technology, Ames (USA), 1987.

[11] P. Cockshott and G. Michaelson, "Orthogonal parallel processing in Vector Pascal," *Computer Languages, Systems & Structures*, vol. 32, no. 1, pp. 2–41, 2006.

[12] W. Cockshott, Y. Gdura, and P. Keir, "Two alternative implementations of automatic parallelisation," in *CPC 2012 16th Workshop on Compilers for Parallel Computing*, 2012.