# Implementation of a robust 3D-image reconstruction algorithm on a TMS320C67x DSP

Marc Leeman, Rudy Lauwereins, Marc Pollefeys

*Abstract*— **This paper describes optimisations done on the implementation of a metric 3D-image reconstruction algorithm on a TMS320C6701 DSP. From the optimised code, an abstraction is made for a description on a higher level. The algorithm takes a series of uncalibrated 2D images taken with a handheld camera and produces a 3D-image of the scene.**

*Keywords*— **3D-model, reconstruction, 'C67x, memory optimisation, memory access, harris, corner detection, loop transformation**

## I. Introduction

3D-image reconstruction is both ane interesting as challenging topic to work on. On one hand, there is an ever increasing interest in the use of these techniques in all kinds of fields of application and on the other hand, most of the available techniques are limited by their extensive and as such expensive setup or the ability to capture and reconstruct only small artifacts.
The algorithm developped by Pollefeys et al.[1] solved these issues by starting from 2D images made with a standard hand-held camera. Furthermore, the algorithm doesn't have any limitations as to the size of the scene to be reconstructed: going from detailed sculptures in a temple wall to Roman theatres in archeology.
Between having a working algorithm and having the algorithm ready for customer appliences, a lot of implementation, transformation and optimisation work has to be done. Due to the heavy computational requirements of the algorithm, the most powerful and at that time soon available DSP, the choice was made for the Texas Instruments TMS320C67x floating point DSP. This paper will focus on one of the first steps in the 3D-image reconstruction algorithm.

## II. 3D-image Reconstruction Algorithm

### A. introduction

Contrary to other 3D-reconstruction algorithms that mostly use some callibration information, this method starts with no a-priory knowled of the scene that is to be reconstructed. A such, the costly setup for retrieving callibration is omitted. Next to this, this method is usable for a wide range of applications and capture techniques: from reconstructing large Roman theatres from areal footage to the capture of detailed temple carvings in a South-East Asean temple.
The method for reconstruction retrieves the information for reconstructing the 3D scene gradually. in the following section, an overview of the different steps will be given to situate the in this paper elaborated module [2].

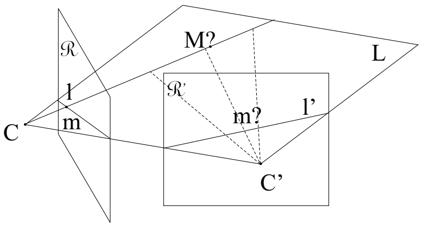

Fig. 1. epipolar geometry

### B. Projective Reconstruction

Initially, the images of the scene are completely unrelated. The only assumption made is that the consecutive image do not differ too much. This allows th restrict the search range for finding corresponding features in different images and match the features fhrough intensity cross-correlation.
Not all features are as easily matched. An *interest point detector* should satisfy two criteria:

- the extraction of the points should be as much as possible independent of camera pose and illumination changes.
- the neighbourhood of the selected points should contain as much information as possible to allow matching.

From the existing cornerdetectors, the Harris corner detector gives the best results according to the two mentioned criteria [3].
The first two images of the sequence are used to determine a reference frame. In both images, the N most prominent corners are detected using the Harris cornerdetector.
Finding matches in pairs of pictures is done in two phases. First, the corners are the matched using normalised cross correlation. Since the images are not supposed to differ that much, it can be expected that the matching points are to be found in the same region. Using pictures that differ relatively more would only make the matching effort larger since the matching corners are to be found in larger areas. In the initial phase, the threshold value for accepting a corner is set high to avoid false matches.
These matches are used to find the *epipolar geometry* or the *Fundamental Matrix*. The epipolar geometry describes the relationship that the location of a certain point is limited in one picture if it has been identified in another. For a point $m$ on the picture $R$ taken with a camera with its focal point at $C$, its corresponding position on the second picture $R'$ is limited along its *epipolar line* $l'$. Of course, a similar reasoning goes for $m'$ and $l$ (cf. figure 1).
The $3 \times 3$ $F$ matrix has 9 unknows, but 7 matches suffice for determining the matrix. 7 matches are selected from the available list using the RANSAC algorithm, which

stands for *RANdom SAmple Consensus.* $F$ is obtainded by solving the set of equations

elaborate alg?

$$x_{2i}^T \times F \times x_{1i} = 0 \qquad (1)$$

for the matches $x_{1i} \leftrightarrow x_{2i}$.
This newly acquired information is now used to refine the $F$ matrix by minimising squares of the distance of the points to the epiplolar line. This is repeated until the $F$ converges. While the first phase of finding matches in a pair of images was done in an area arount a candidate, the epipolar geometry is used to refine the search and add additional points to the list of matches. The strategy for finding the new matches can best be seen in figure 1. When looking for a matching point on the second image, the search can be restricted to a small band along the epipolar line of that point on the second image. Because the threshold value was set very high, and the restricted search area in the first phase, a lot of good correspondences were rejected, but are found here.
At this point, the 3D structure can be built using triangulation. In order to do this, the projection matrices ($P_1$, $P_2$) for the found matches have to be calculated. The $P$ matrices are found similarly to th e$F$ matrices, with a linear initialisation and a non-linear convergence step.
With the $P$ matrices and the point coordinates the 3D points is *projected backwards*: with every point on the image, a straight line can be associated, going through $C$ in image 1 and the projection on the image. The original position of the 3D point has to be on the intersection of the straight lines from the two pictures. Due to noise, the straight lines will probably not interesect in one single point. Therefore, the reconstructed point is set at the location that minimises the squared sum to the two projection lines.
More images are added in *chains*: a new image is linked and compared with the previous one processed. This is done for the first two images, the corners are found, the matches and epipolar geometry are calculated.
Finding the projection matrix $P$ is somewhat different as with the initialisation though. Instead of using the correspondences between the current image and the previous one, only those points are used that are already mapped to a 3D point are used. As with the $F$ matrix, the there is an initialisation using *RANSAC* and a non-linear refinement. In this case, a sample of 6 matches is needed to compute $P$. This non-linear refinement is needed because the new $P$ matrix will determine the new 3D structure and should be as accurate as possible.
With the new $P$ matrix, the 3D points are refined. When initialising the 3D structure, the *uncertainty elipsoid* is relatively large, especially if the corner between the two images is small. The uncertainty elipsoid is the area around the reconstructed 3D point where the real 3D point is located. By gaining more information (adding more views), this will become smaller. Since this elipsoid is used for accepting a point as a match, the projection matrices will be calculated with more precision as images are added. The adaptation of the elipsoids is done using the *Kalman filter.*

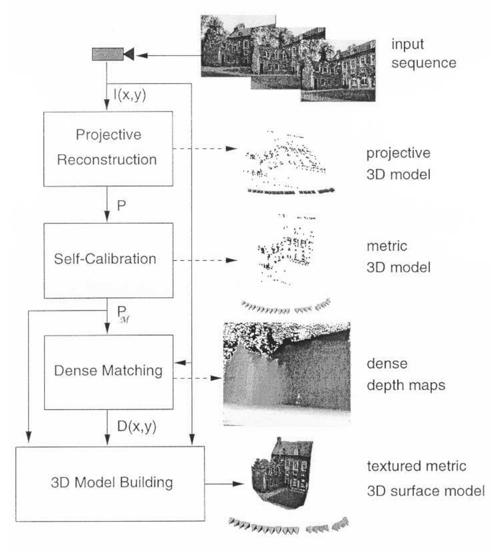

Fig. 2. 3D algorithm global structure

The entire procedure for finding the $P$ matrices is redone with the new estimates for the 3D points.
While adding new images, new 3D corners will be found, while others are not found anylonger. The central idea for retaining a corner will be that a 3D point will be accepted is if it is found in three consecutive images. Points are considered erroneous is they are outside a $3 \times \sigma$ band. Because these points can distort the standard deviation, the removal procedure is repeated until no more points are outside this band [4].

## C. Self-Calibration

In order to get from a projective to a metric reconstruction, the model has to be Because the images of the scene are not calibrated, the camera calibration, orientation and position are unknown which entails the following:
• the intrinsic camera parameters are not known: different camera settings could be used for every view.
• the extrensic camera parameters are not know: the position and orientation of the camera is unknown and can change for every image taken.
The only assumption made is that the scene itself is static. However, by accepting some constraints on these parameters (use of square pixels, estimate of focal length and expected principal point), the problem is reduced and the projective model is upgraded to a metric model using *flexible selfcalibration* [2].

## D. Dense Matching

At this point in the algorithm, a series of important scene points are reconstructed and a reconstruction can be made with interpolation. But now, the algorithm has a series of calibrated images, which enables the use of stereo algorithms that will find matches for most of the pixels.
When considering the epipolar geometry that is retrieved, a match of a particular point lies on its epipolar line on the other image. Therefore, the image pairs are rectified so

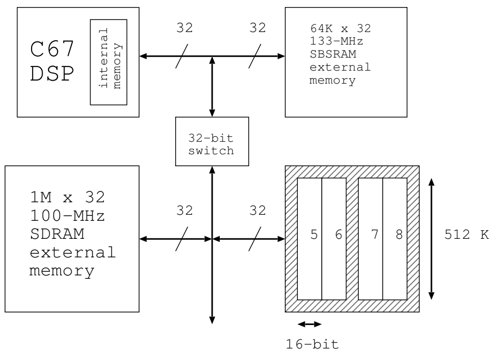
C67 DSP
internal memory
32
32
64K x 32 133-MHz SBSRAM external memory
32-bit switch
1M x 32 100-MHz SDRAM external memory
32
32
5
6
7
8
512 K
16-bit

Fig. 3. 'C67 with External Memory

that epipolar lines coincide with the scan lines to increase computational efficiency. For every pixel, the normilised cross correlation used along the scan lines. Matching ambiguities are solved be exploiting an ordering constraing in the dynamic programming approach. Pairwise disparity estimation allows computing theimage to image correspondence between adjacent rectified image pairs. These corespondences are now fused into a common 3D model through *controlled correspondence linking.* Starting from a reference viewpoint, the correspondeces between adjacent images are linked in a chain. The depth for a image point from the reference view is computed from the correspondence linking that delivers two lists of images correspondences relative to the reference, one linking down and one linking up. The result of this is a dense 3D map. Finally, texture is added on the dense map for visualisation. A parametric surface model is used to get spacial coherence between the points in the model. Morphological filtering removes spurious and tiny regions. To smooth the surface and to interpolate small surface gaps in regions that could not be measured, a bounded thin plate model is used with a second order spline.

insert figure to make this clear?

For avoiding bias towards one single image, the texture from the image sequence is fused in a similar way [1].

## III. The Texas Instruments TMS320C67x Memory Architecture

In order to fully comprehend the used optimisations on the *harris cornerdetector*, the used 'C67 memory architecture of the EVM will shortly laid out. For a full description, see [5] and [6].

The 'C6x EVM has two levels of external memory:

1. 256 kB SBSRAM (64K × 32-bit words of 7.5 ns (133 MHz))
2. 8 Mb SDRAM (1M × 32-bit words of 10 ns (100 MHz))

The SBSRAM on the EVM is directly connected to the DSP, without glue logic or buffers. A 32-bit bus switch is used to isolate the SDRAM from the DSP itself and the SBSRAM. The switch enables running the SBSRAM at 133 MHz. The SDRAM is divided in two banks. Each of these banks in turn is divided in two banks of two times 16 bit (cf. figure 3).

The used board has 64K internal memory. The *internal* memory banks are actually interleaved. Because each bank is single ported, only one access per cycle is allowed to a certain memory bank. Two accesses occur in a single cycle result in a memory stall: all pipeline operations are stopped for one cyle while the second value is read from memory. When the memory banks are not accessed at the same time, it is possible to access *all* banks using two parallel LDDW instructions and as such fetching 128 bits in one cycle.

## IV. Transformations

During the re-implementation, a number of transformations were immediately identified as critical areas in memory usage for the target platform. In what follows, the transformations of the HARRIS corner detector for finding significant points in the pictures are elaborated.

### A. The HARRIS corner detector

Finding the corners with the harris cornerdector is actually done in different substeps. Fist, all pixels the images are given their *cornerness* value. This property is an indication for a certain point as how much it satisfies to the cornerness criteria. The pixels with the highest cornerness values will be retained by the algorithm. The cornerness value used is defined as:

$$c = (f_{xx} \times f_{yy} - f_{xy}^2) - 0.04 \times (f_{xx} + f_{yy})^2 \qquad (2)$$

with $c$ the cornerness factor and $f(x, y)$ the intensity of a pixel at coordinates $(x, y)$. $f_{xx}$, $f_{yy}$ and $f_{xy}$ are the second dirivatives to $x$, to $y$ and to $x$ and $y$. In practice, finite derivatives are computed, using 1D Laplacian masks with a size of 5.

After the detection and selection of the most prominent corners, Harris' algorithm adds one more step for finding the *sub-pixel precision.* A point has a intensity $f(x, y)$. The intensity values is only defined for $x$ and $y$ values that are coordinates of pixels. When four neighbouring pixels are found with high cornerness values $f(x, y)$ (the most prominent corners are found or the pixels with the highest cornerness values), their function values can be used to find a maximum with quadratic fitting somewhere in between these four pixels. This maximum can be considered as a corner with sub-pixel precision.

Only images without texture of very textured images can cause problems. When using images whithout much texture, the corner detector finds not enough corners. On the other hand, when images are very textured, too much similar corners are found on very short distances from each other, making it virtually impossible to distinct them from one another. Another, perhaps less evident result of these properties is that corners can be concentrated in a couple regions of an image that contain highly textured objects. A typical example is a scene containing a building (which is the object intended for reconstruction), containing a couple of trees in the background. When running the *pure* cornermatcher on these kinds of images, an very high portion of (meaninless) corners will be found in the corners, skipping the interesting ones in the building itself.

The straightforward yet effective solution is to spit the image in different areas where the corner detector has to find a fraction of the global number in each.

## B. Mathematical Implementation: used resources

This program is ported from a object oriented C++ environment to functional C. The consistency of the variables used in the classes is ensured here by using *state variables.* This solution has also been proposed in the guidelines TEXAS INSTRUMENTS for software portability in DSP envirments [7]. The state variable collects the buffers and properties used and adapted throughout the program. Instead of calling a certain objects method, a function is called with one parameter being the pointer to the state object. The mapping of the classes onto the state variables is straightforward as shown in the following definitions (C++):

```
struct DR_MEMORYSTR{
    GL_BOOLEAN_ARRAY_STR *image_corner_max_ptr;
    GL_FLOAT_ARRAY_STR *image_corner_ptr;
    GL_FLOAT_ARRAY_STR *image_fxx_ptr;
    GL_FLOAT_ARRAY_STR *image_fxy_ptr;
    GL_FLOAT_ARRAY_STR *image_fyy_ptr;
    GL_FLOAT_ARRAY_STR *image_scratch_ptr;
    GL_IMAGE_UCHAR_STR *image_ptr;
    GL_INT_ARRAY_STR *image_gradx_ptr;
    GL_INT_ARRAY_STR *image_grady_ptr;
};
```

and the corresponding state variable is (C):

```
typdef struct imagevars{
    TUCharMatrix image;
    TFloatMatrix corners;
    TUCharMatrix cornersmax;
    TIntegerMatrix gradx;
    TIntegerMatrix grady;
    TFloatMatrix fxx;
    TFloatMatrix fxy;
    TFloatMatrix fyy;
} TImageVars;
```

The names of the arrays and types are self explaining and the correspondences can quickly be made. The original C++ implementation assigns one additional memory chunk with *image_scratch_ptr*. Contrary to the other buffers, this one is only used in one particular method. In the C implementation, this array was made local in the particular function instead of making it global and hogging up the memory for the duration of the program. This technique is exactly what is meant with the term *mathematical implementation*: a collection of variables is needed throughout the program or module, so they are declared globally thus avoiding any possible consistency problems. In the following discussion and experiments, a grey scale image is used of $480 \times 640$.

The DR_MEMORY_STR structure in the original program uses 9000 $((480 \times 640) \times (1+4+4+4+4+4+1+4+4) = 9216000$ ) kbytes. Its C counterpart initialises *only* 7800 kbytes but equally occupies a maximum 9000 kbytes in the function where the additional buffer is needed. Obviously one approach would be to create the $480 \times 640$ buffers only when needed and free them immediately afterwards. This good programming practice actually reuses the available memory space.

Some other smaller arrays are used, but compared to the ones described previously, their size is negligible.

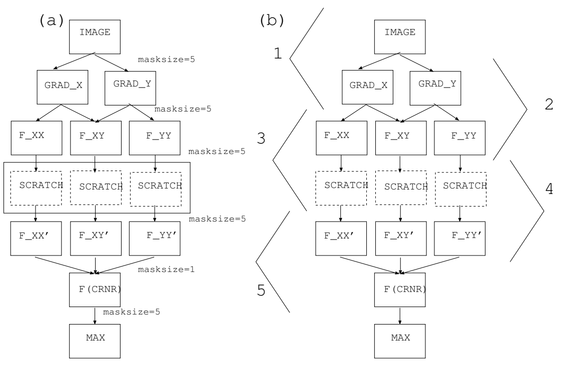

Fig. 4. Dependency of the buffers used in the Harris corner detection

## C. Proposed Optimisation

### C.1 Buffer-usage: original implementation

Figure 4 shows the *data dependency queue.* Any data structure in the graph is created from the previous level. The buffers in this case are (cf. the C and C++ definitions):

- the image: (1 byte) unsigned char.
- the gradients (GRAD_X and GRAD_Y): right and down edge detection filters with the mask [-2,1,0,1,2].
- second order derivatives (F_XX, F_XY and F_YY)
- second order derivatives (F_XX', F_XY' and F_YY') after horizontal and vertical smoothing. For this calculation a buffer SCRATCH is used to store the results of on smoothing. The second smoothing writes the results back to the original memory of the second derivatives. At this point, the programmers of the original algorithm used one optimisation. Though the consistent use of this technique could dramatically reduce the memory footprint, it will be shown that this is still far from the optimal result.
- scratch
- the cornerness values: $(f_{xx} \times f_{yy} - f_{xy}^2) - s \times (f_{xx} + f_{yy})^2$
- the local maxima in the cornerness in a $5 \times 5$ mask.

Since the image, the cornerness values and the local maxima are needed in an iterative process later on, the efforts will currently be focused on the 'buffers' in between. As they are not needed anywhere else, they are by definition temporary.

### C.2 Buffer-usage: good programming

A better approach than to allocate memory for every intermediate image-size array is to re-use the allocated arrays that are not needed anymore. Since all arrays are calculated only from the previous level, the bottle neck will be the largest memory occupied by the combined matrices of two subsequent levels. Figure 4 shows the five transitions that can determine the required memory space.

- transition 1: the memory here is $480 \times 640 \times (1byte + (2 \times 4bytes))$ or 2700 kB. Since the image has to be retained for later calculations, only 2400 kB is relevant as buffer space.
- transition 2: after the edge detection in one direction, the second derivatives are calculated. the required space for this phase is $480 \times 640 \times ((2 \times 4bytes) \times (3 \times 4bytes))$ or 6000 kB.

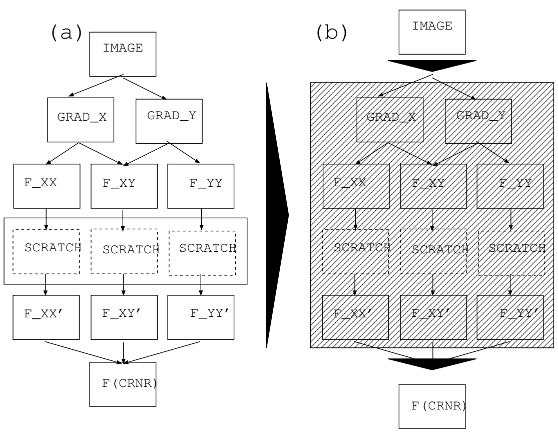

Fig. 5. black box data

• transition 3 and transition 4: the smoothing with a Gaussian mask is split in two parts because of the use of the intermediate SCRATCH memory. On the other hand, the double smoothing is done sequentially for each buffer: horizontal smoothing is done on F_XX and stored in SCRATCH. Then, vertical smoothing is performed in the SCRATCH and written back in the memory allocated for F_XX, ...The memory space needed for transition 4 or transition 5 is $480 \times 640 \times ((3 \times 4bytes) + 4bytes)$ or 4800 kB.
• transition 5: again only part of the involved arrays can be considered as buffers since the cornerness array has to be reused in the iterative process later on. The actual memory requirements are $480 \times 640 \times (3 \times 4bytes)$ or 3600 kB.

The computation that consumes the most memory is clearly the second transition with 5 32-bit image-size arrays. So, in comparison with the initial estimate of the C and C++ structures, the required space is 6000 kB $+480 \times 640 \times (1byte + 4bytes + 1byte)$ or 7800 kB. The only memory saved is an image sized array of floats (the SCRATCH buffer, since the free space of GRAD_X and GRAD_Y can be reused). The data barely fits in the TI's 'C67x external memory, without considering even program memory.

## C.3 Buffer-usage: minimal overhead

The previous section showed that simply reusing the buffers was not sufficient for implementation of the harris corner detector on a 'C67 EVM. Optimisation has to be done on a finer granularity. The concept is that the buffers that are only constructed on an intermediate basis have no particular meaning for the algorithm. It is not needed to retain their internal logical structure. Instead levels 2 to 5 are considered as a *black box* or *data pool*: the internal structure is irrelevant, but the data is needed for obtaining the result (cf. figure 5). Since the *only* meaningful matrices are the image and the matrix with the corresponding cornerness of each pixel, a value will only be kept as long as it is needed for the computation of the lower level value. All computations require a one dimensional mask with size $N$ (5 in this case) and are done on a $X \times Y$ matrix ($480 \times 640$ here) with an element size of $e$ **bytes**. When the arrays are passed through horizontally first (i.e. a double `for` loop), only $N$ elements are needed from the previous level for a new value when using the horizontal mask and

$$B = (Y \times (N-1) + \frac{N-1}{2}) \times e \tag{3}$$

for the vertical mask[1]. The calculation of the vertical mask clearly is more consuming and will be the one requiring the most buffer space. By delaying the index of the calculation for the next level using the horizontal maks by a minimum of $\frac{N-1}{2} + 1$ *slots*, its required pixels have already been calculated. The usage of the elements is depicted in figure 6. This was for a transition from level $i$ to level $i+1$. For the applicating a $N$ size mask sequentially from level $i$ to level $i+n$, the required data in the higher levels will grow in a rate linearly proportional to the masksize $N$: each level will add $N-1$ rows. It is not hard to see that for obtaining data elements in layer $i+n$

$$B_n = (Y \times n \times (N-1) + \frac{N-1}{2}) \times e \tag{4}$$

elements ($B_n$: buffer at $n$ layers previously) are needed in layer $i$. Of course, the final *total* buffer size depends on the actual *function* implemented in the *black box*. In the harris corner detector case, the buffer size at the sixth layer (*cornerness*) equals to

$$\begin{aligned}
B_{total} &= 3 \times 4 + 3 \times B_1 + 3 \times B_2 + 2 \times B_3 \\
&= 3 \times 4 \\
&\quad +3 \times (640 \times 1 \times (5-1) + 2) \times 4) \\
&\quad +3 \times (640 \times 2 \times (5-1) + 2) \times 4) \\
&\quad +2 \times (640 \times 3 \times (5-1) + 2) \times 4) \\
&= 12 + 30744 + 61464 + 61456 = 153664 bytes
\end{aligned}$$

Note that nor parts of the image, nor parts of the cornerness matrix are part of the buffer as indicated in the *black box* approach. Since the operations are symmetric in the horizontal or the vertical way the dimensions can be switched. When taking the 480 dimension as *horizontal*, 115264 bytes are required. By re-evaluating the required data for producing the *cornerness* values, an improvement of 80 or 60 has been realised (9000 kB → 150 kB or 112 kB). The total required memory (including the image, cornerness and local maxima) can be kept on the 'C67 EVM now (1950 kB) But the buffers, where the bulk of the computations are done on, still do not fit in the internal memory.

## C.4 Buffer-usage: interleaved loops

What was done in the previous section was moving part of the control (the `for` loops) to a higher level. The *horizontal* loop was moved to the outside.

When just looking at the buffersizes, another optimisation can be made. In the next discussion, only the elements

[1]all used formulas are approximations, and do not keep track of outer border exceptions

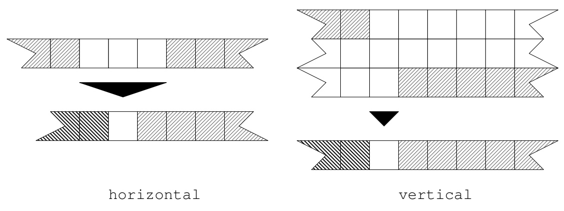

Fig. 6. Pixels needed and processed before a new one can be calculated

that are needed to construct the ultimate *cornerness* values. When at each level a horizontal mask and a vertical mask is used, the required data are a diamond structure as indicated in figure 7. Instead of keeping the *diamond* structure, it makes sense computationally (for re-use of the values and for reducing the index calculation) to take the square figure. Not only is the memory management easier, but from the $(1 + (N - 1) \times n)^2$ used in timeslot $i$, $(1 + (N - 1) \times n)^2 - (1 + (N - 1) \times n)$ can be reused in the next iteration. Obviously, this goes for every level in the buffer hierarchy. In this case,

$$B_n = (1 + (N - 1) \times n)^2 \times e \tag{5}$$

are needed at level $n$. The required bufferspace of the Harris cornerdetector is reduced to

$$\begin{aligned} B_{total} &= 3 \times 4 + 3 \times B_1 + 3 \times B_2 + 3 \times B_3 \\ &= 3 \times 4 \\ &\quad 3 \times (1 + (5 - 1) \times 1)^2 \times 4 \\ &\quad 3 \times (1 + (5 - 1) \times 2)^2 \times 4 \\ &\quad 2 \times (1 + (5 - 1) \times 3)^2 \times 4 \\ &= 12 + 300 + 972 + 1352 = 2636 bytes \end{aligned}$$

In this case, the buffer size is no longer dependent on the image size. This result is significant when reconstructing hi-resolution 3D-scenes and the memory becomes more of a bottleneck. Furthermore, the buffers size is somewhat over 2 KB, which easily fits in the 64 KB internal memory. This approach will be very efficient in reducing the accesses to external memory (cf. infra). Unfortunately, it comes at a price: the buffers can only be reused in one direction: the direction in which the data is constructed (e.g. horizontal). The previous figures are correct when the data dependency graph is interleaved at eacht transition and if a horizontal and vertical mask is needed at each stage. In the Harris cornerdetector, this is not the case, so one last optimisation can be made

$$\begin{aligned} B_{total} &= 3 \times 4 + 3 \times 5 \times 4 + 3 \times B_1 + 2 \times B_2 \\ &= 3 \times 4 + 3 \times 5 \times 4 \\ &\quad 3 \times (1 + (5 - 1) \times 1)^2 \times 4 \\ &\quad 2 \times (1 + (5 - 1) \times 2)^2 \times 4 \\ &= 12 + 60 + 300 + 648 = 1020 bytes \end{aligned}$$

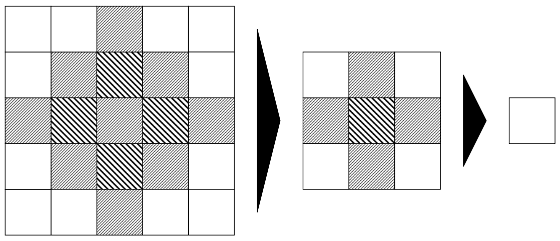

Fig. 7. Minimum amount of pixels needed for a values

## C.5 Buffer-Usage: Internal Memory limitation

The two previous parts were the trade-offs between
• "*large*" buffers and minimum and cycles needed for calculating the cornerness
• a minimum of buffers but a large cycle overhead for recalculating intermediate values.

Because of the low ammount of internal memory of the 'C67 and the memory consuming properties of the harris cornerdetector, the fastest option (i.e. keeping the image, results and required buffer bands in internal memory)can not be implemented using a picture as small as 480 × 640. Even if different boards with larger internal memories were to be used, the optimisation would most likely be limited to that particular image size since the buffersize grows liniarly with $15 \times (Y \times N)$ for the Harris cornerdetector. The optimisations had to work to the optimium of accessing every pixel of the gray scale image only once in external memory. The solution lied in the *localness* of every transition in the dependency graph. All transformations are done on a series of values that are adjacent to the position of the one which is created. This means that it is possible to split the image into different parts without compromising the cornerness result. For finding the cornerness values at the edges of the parts, some part of the image will have to be replicated, since the mask grows $2 \times \frac{N-1}{2}$ pixels for every level transition. At figure 8, the final result will only be of the dark shaded area. the white planes are the pixels that have to be copied. It is pretty evident that this overhead increases as the number of vertical stripes in which the picture is divided increases. For the cornerdetector, $3 \times \frac{N-1}{2}$ or 6 pixels are dropped on the outer borders of the image, in analogy, 6 will have to be replicated in the innner edges of the image parts. Summarised, for good performance the buffers (as well as the part of the image worked on and the part of the result worked on) have to be located in internal memory, or:

$$\left(\frac{X \times Y}{p} + (N - 1) \times X \times n\right) \times e_{in} + B_{total}^{part} + \left(\frac{X \times Y}{p}\right) \times e_{out} \tag{6}$$

The first term is the fraction of the image is processed when using $p$ parts, while the second is the second is the amount of pixels that has to be added to obtain an $p^{th}$ of the result (last term). Of course, the buffer size of the part of the image worked on has to be added.

First, the DMA is left out of consideration. Filling up the internal memory with as much data as possible to reduce overhead gave an image of 15 or 20 wide or about 32 im-

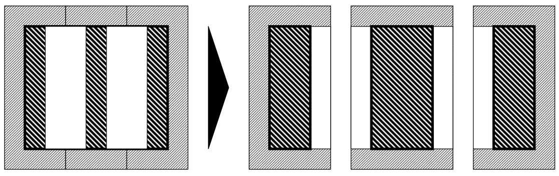

Fig. 8. Replication overhead

age stripes. The 32 stands for either the 640 dimension as the outer loop, or the 480: there is an almost perfect trade off between the buffersizes and the size in memory for the cornerness. The only factor that impedes the choice for having the *short* side up for division is the increased overhead. The overhead (i.e. the total amount of replicated pixels) grows linearly with number of partions of the image. Since this number of replicated pixels in determined by
1. the algorithm
2. the length of the *cut* in the image, or the length of the undivided side
it grows faster if the short side is divided. The *cornerness* values in internal memory are actually responsible for the small parts that have to be processed due to their large element size (`floats`). In this case, the replicated pixels are about $\frac{2}{3}$ of the pixels for which the cornerness is computed. Figure 9 shows the memory needed when the original image is divided in different parts. The point at which the required space is lower than 64 kB (30 parts) is at the right-hand side of graph. At this point, the program code is reorganised in such a manner that the data is available for constantly providing the 'C67 core.
Not quite. This is true only in this respect hat the DSP is fully occupied while processing a part of the image. After each stripe, DMA channels are started to
1. move the computed *cornerness* values to external memory. Though they are still needed afterwards, the internal memory is needed for further computation and has to be freed.
2. move a new band of the original gray scale image into internal memory
When one band of an image is processed, the next one should be available in internal memory. The adjusted evaluation that has to be made is:

$$2 \times (\frac{X \times Y}{p} + (N-1) \times X \times n) \times e_{in}$$
$$+ B_{total}^{part} + 2 \times (\frac{X \times Y}{p}) \times e_{out}$$

When doubling the relative size of the memory needed for the input and output values, this effect is more accute. Here 640 is used for the width and only bands of 10 pixels can be used. The replicated data is larger since 6 pixels on both sides have to be added.
The formula can be modified so that instead of only considering the division of the image in one dimension, horizontal as wel as vertical partitions can be used:

$$((1 + (N-1)^2 \times n^2) \times \frac{X \times Y}{p_x \times p_y} + (N-1) \times n$$

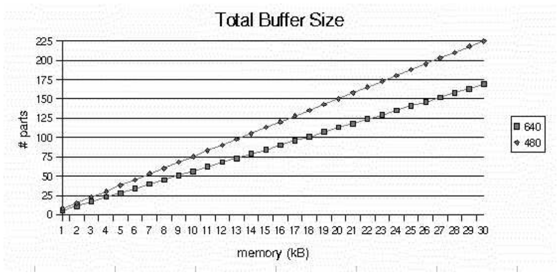

Fig. 9. Image segmentation: Internal Memory Occupation: total access overhead

$$\times (\frac{X}{p_x} + \frac{Y}{p_y})) \times e_{in} + B_{total}^{part} + (\frac{X \times Y}{p_x \times p_y}) \times e_{out}$$

$p_x$ is the number of partitions vertically and $p_y$ is the same for the horizontal direction. This version is still without DMA, but as expected, the results are not that much better than the code where the image was only divided in one direction. Adjusting the calculations for using DMA only stressed these results.

## *D. Memory Accesses*

Up until here, the program code has been reshuffled for fast execution of the *harris cornerdetector.* For this, the data where the 'C67 was working on at that time, had to be quickly accessibe and be put in internal memory. These optimisations also resulted in far less external memory accesses.
In the original mathematical implementation, the harris cornerdetector was done on image size buffers: some masks are applied on a480 × 640 image sized buffer, and a new one is produced. Since there is no possibility to buffer these sizes in internal memory where the data is easily accessible for the processor, the values have to be fetched from external memory and written to external memory. The masks are all of size 5 untill the actual construction of the *cornerness*, so every pixel in the buffers in the 4 level (from IMAGE to F_XX is accessed 5 times. Together with accessing every pixel in the F_AB' layer one time, this makes almost 15 million read accesses and over 3.5 million write accesses totals almost 18.5 million accesses.
This figure assumes that every pixel is fetched from external memory when it is needed and written back to external memory when it is computed. When using DMA, every pixel is fetched and stored in internal memory and then the next level is written back out. A quick glance (without keeping track for the needed overlaps because the entire buffers from level $i_{l-1}$ and level $i_p$ do not fit in internal memory, it is assumed the internal memory is large enough) still gives almost 8.5 million accesses. The memory accesses are in this case relatively independent from the buffer reuse, at least at the top level. Though the memory space is reduced in the *good programming* approach, while leaving the global structure untouched, the accesses stay the same. They are merely done to another place in memory.
When the **minimal overhead** or the **fully interleaved loops** approaches are used, the pixels of the image are re-

tained as long as they are needed (cf. figure 6) and the intermediate values are only kept for computing the final cornerness. The image pixels are read, the cornerness is computed in one pass and afterwards written to memory. The accesses are just over 600,000 or an improvement of 30. Of course, there's only one problem: the TI EVM doesn't have enough internal memory to provide for these large chunks of required memory.
When keeping in account the *internal memory* limitation of the EVM, the total amount of memory accesses is the number of pixels in the image, the number of output values (cornerness) AND the number of replicated pixels on the edges. The total number of accesses with the optimised code is about 790,000 or an improvement more than 20 in comparison with the original mathematical implementation.

## V. CONCLUSION

- memory becomes bottle neck
- note on overhead
- due to memory bottle neck, more accesses to external memory

## REFERENCES

[1] Marc Pollefeys, Reinhard Koch, Maarten Vergauwen, and Luc Van Gool, "Metric 3d surface reconstruction from uncalibrated image sequences," in *Lecture Notes in Computer Science*. Proc. SMILE Workshop (post-ECCV'98), 1998, vol. 1506, pp. 139 – 153., Springer-Verlag.
[2] Marc Pollefeys, *Self-Calibration and metric 3D reconstruction from uncalibrated image sequences*, Ph.D. thesis, K.U.Leuven, Kardinaal Mercierlaan 94, 3001 Heverlee, May 1999.
[3] C. Schmid, R. Mohr, and Bauckhage C., "Comparing and evaluating interest points," Proc. International conference on Computer Vision, 1998, pp. 230 – 235, Narosa Publishing House.
[4] A. Verbiest and M. Vergauwen, "Robuuste 3d reconstructie uit meerdere beelden," M.S. thesis, K.U.Leuven, Kardinaal Mercierlaan 94, 3001 Heverlee, 1997.
[5] Texas Instruments, *TMS320C6201/6701 Evaluation Module Technical Reference*, Texas Instruments, Nice, France, December, 1998.
[6] Texas Instruments, *The TMS320C6000 CPU and Instruction Set Reference Guide*, Texas Instruments, Nice, France, March, 1999.
[7] Texas Instruments, *eXpressDSP Algorithm Standard Rules and Guidelines*, Texas Instruments, Nice, France, September, 1999.